Christoph Lüth, Serge Autexier

Korrekte Software: Grundlagen und Methoden

Sommersemester 2021

Lecture Notes

Last revision: 15/07/2021.

Contents

Preliminary Notes

Status of this document:

- Chapter 4 (Equivalence of denotational and operational semantics) is *missing*.
- In Chapter 10, Section **??** (Function calls) is *missing*.
- In Chapter 11, Section 11.7 (Verification Conditions with References) is *missing*.

Chapter 1

Introduction

1.1 Why "correct software"?

1.1.1 Well-known Software Disasters # 1: Therac-25

The Therac-25 was a novel, computer-controlled radiation therapy machine which between June 1985 and January 1987 massively overdosed six people (with a radiation dose of 4000 – 20000 rad, where 1000 rad is considered to be lethal), leading to five casualties. The overdoses were the result of several design errors, where one of the root problems was the software being designed by a single programmer who was also responsible for testing [5, Appendix A]. These incidents are thought to be the first casualties directly caused by malfunctioning software.

1.1.2 Software Disasters in Space

The Ariane-5 exploded on its maiden flight (Ariane Flight 501) on June 4th 1996 in Kourou, French-Guayna. How did that happen? The inquiry which was held after the incident reconstructed the exact sequence of events, backwards from the disaster [6]:

(1) Self-destruction was triggered due to an instability.

(2) The instability was due to wrong steering movements (rudder).

(3) The steering movements resulted from the on-board computer trying to compensate for an (assumed) wrong trajectory.

(4) The trajectory was calculated wrongly because the own position was wrong.

(5) The own position was wrong because positioning system had crashed.

(6) The positioning system had crashed because transmission of sensor data to ground control failed with integer overflow.

(7) The integer overflow occurred because input values were too high.

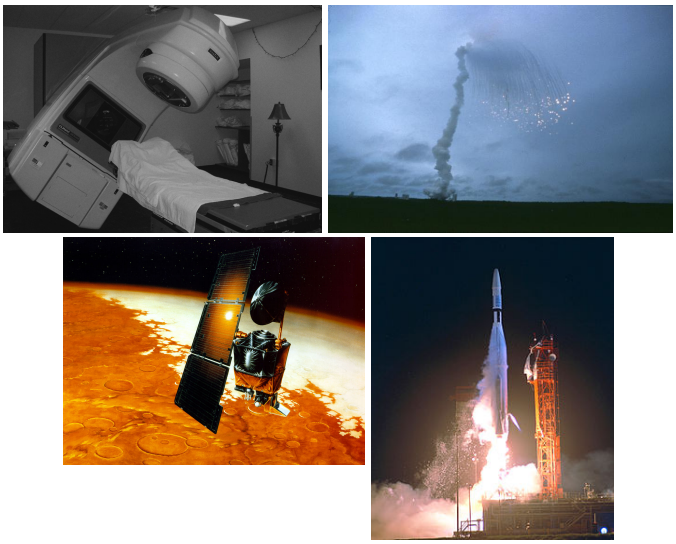

Figure 1.1: Software Disasters (from top left, clockwise): The Therac-25; Ariane-5 exploding on its maiden flight; Atlas booster carrying Mariner-1 taking off; artistic rendition of the Mars Climate Orbiter

(8) The input values were too high because the positioning system was integrated unchanged from predecessor model, Ariane-4.

(9) This assumption was not documented because it was satisfied tacitly with Ariane-4.

(10) The positioning system was redundant, but both systems failed within milliseconds because they ran exactly the same software (systematic error).

(11) Furthermore, the transmission of data to ground control was not necessary; it was only included to allow faster restart if the start had to be interrupted.

The Ariane-5 incident was comprehensively investigated afterwards. It was both spectacular and costly, to the tune of 500 mio Euro. Other software disasters in space include the loss of the Mariner-1 spacecraft 294 seconds after launch on August 27th 1962, and the Mars Climate Orbiter.

The Mariner-1 had to be destroyed because the guidance system of the Atlas booster rocket carrying Mariner-1 was faulty. The guidance system was taking radar measurements and turning them into control commands for the rocket. It turned out that the programmer had missed on overbar[1] (as in $\overline{R_n}$) which stood for smoothing the measurements (taking the average over several samples). Coupled with the failure of a secondary radar system, this lead to the control system working with faulty data, for which it tried to compensate wildly, leading to a rocket which was effectively out of control.

The Mars Climate Orbiter failure was more simple: one of the subcontractors was working with imperial measures, whereas the rest of the system (and NASA) was (and is) using metric units. Thus, the navigation software was using wrong values to calculate the course and steering commands for the craft, which subsequently went for too low into the Mars atmosphere and was lost.

It should be pointed out that software disasters in space are so well-known because they tend to be spectacular, and because space agencies do in fact have a very good culture of learning from errors; thus,

[1] This is sometimes, and incorrectly, referred to as a “missing hyphen”.

after each of these disasters an enquiry was held trying to establish the exact causes of the failure. This is how these errors become so well-known, as opposed to errors in closed commercial applications which tend to be hushed up (or, in the case of consumer products, are just conformant to expectation).

1.1.3 Not-so-well-known Software Disasters

On January 15th 1990, the AT&T long distance network (the telephone backbone of the US back then) began to fail on a large scale, losing up to a half of the calls routed though this network. Between 2:25pm and 11:30 pm, AT&T lost more than $ 60 mio in unconnected calls (not counting losses by *e.g.* hotels and airlines counting on the network for their reservation systems). This was a genuine software bug which caused network nodes to reboot and take down neighbouring nodes with them [1]. The software in question was written in C, thus this incident is highly relevant for this course.[2] A more recent telephone-related incident was the outage on October 4th, 2016 in the US, which was caused by an an operator leaving empty an input on the wrong assumption it would be ignored when in fact it was not [3], although here we have a bad user interface instead of a genuine software bug.

On a related note, there was the Wall Street crash from October 19th, 1987, when the Dow-Jones fell by 508 points, losing nearly a quarty of its value; apparently, the greatest loss on a single day. This could be traced to trading programs (a novelty back then) selling stock automatically (due to falling prices, which were caused in the day by an SEC investigation into insider trading) which lead to falling prices, which lead to a self-reinforcing feedback loop as trading programs were trying to sell more and more stock, effectively overwhelming the market, which lead to a widespread panic. Not a software disaster as such, as there was no faulty software involved, but a disaster caused by unintended (“emerging”) effects of software.

1.1.4 Software Correctness and Safety

Incorrect software cannot be safe, but safety is more than correct software. In fact, most of the disasters above were more than software not functioning as it was specified; for disasters on that scale, the whole system design process has to be flawed in one way or another (see [2]).

However, that does not mean we should not care about software correctness, quite the contrary.The functional safety standard, IEC 61508, defines safety as “freedom from unacceptable risks of physical injury or of damage to the health of people, either directly, or indirectly as a result of damage to property or to the environment” [4, §3.1], and goes on to define *functional safety* as the part of the overall safety that depends on a system or equipment operating correctly in response to its inputs. Thus, correct software is a prerequisite of functional safety which is a part of the overall safety of a system.

1.2 Semantics

In general, semantics means assigning a *meaning* to some concrete (syntactic) construct. Here, we talk about programs, so we assign meanings to *programs*. For example, consider the program in Figure 1.2. What does it compute? If we look at it, we will convince ourselves it computes (in p) the factorial (of n). Semantics is concerned with making this statement precise.

What could the meaning of a program be, and how do we model that in mathematical terms?

[2] Or not — the problem was caused by one of the problematic features of C which are not in the subset covered here, and which in fact is ruled out by safety-directed subsets of C such as MISRA-C.

```
p= 1;
c= 1;
while (c <= n) {
  p = p * c;
  c = c + 1;
}
```

Figure 1.2: An example program. What does it do?

- It could be what the program *does* — then, we have to describe the action of the program somehow. We do so in terms of actions of an abstract machine, *i.e.* we give an abstract notion of the *state* of a machine as a map of adresses to values, and describe how the program changes that. This is called *operational semantics*.

  Concretely, the abstract machine is a map of variable names to values. In our example, this starts with say $n \mapsto 3$ and p, c undefined, and enters the loop with a state $n \mapsto 3, p \mapsto 1, c \mapsto 1$. The loop condition (and any other expression) is always evaluated with respect to the current state, so we enter the loop; after the first loop iteration, we get $n \mapsto 3, p \mapsto 1, c \mapsto 2$, and then after two more iterations $n \mapsto 3, p \mapsto 6, c \mapsto 4$, at which point we exit the loop.

- We could do so by assigning, to each program, a mathematical entity which describes this program. Since programs take inputs and give us outputs, it would seem natural to describe programs as partial functions. (This, of course, works best with functional languages, but we can also use it with C0.) This is called *denotational semantics*.

  Concretely, we model programs by partial functions between states (mapping variable names to values, as above). It is easy to see how this works for the first two lines in our factorial program, but modelling the while loop requires the mathematical construction of a fixpoint, which we will explore in depth later.

- Finally, we can describe a program by all the properties that is has. (This is sometimes called extensionality.) For our program, it would mean to specify what it exactly computes, *e.g.* stating that the example program in Figure 1.2 calculates the factorial, $p = n!$. This is called *axiomatic semantics*.

All three semantics can be considered as different *views* on the same syntactic entity. The semantics should *agreee* in the sense that for a given input, they should state that the output (result) is the same: the semantics should be *equivalent*.

It should be pointed out that what the program actually does when it runs is something else, because it depends on things such as the compiler used, the underlying machine *etc.*, but hopefully agrees with the semantics. Only for a few programming languages such as the functional language Standard ML, a subset of Java, and C have the mathematical semantics been fully worked out. For full C, this is surprisingly complex; it has been done, but not in correspondence with the popular C compilers. However, there is certified C compiler, safecert, which has been proven (certified) to be correct with respec to its denotational semantics.

Chapter 2

Operational Semantics

Operational semantics describes programs by what they do. For imperative programs, this means the program has an implicit or ambient state (*i.e.* the state is not explicitly written down, programs only refer to the state or change parts of it), and the operational semantics aims to capture this in a mathematical precise way. In particular, it makes the notion of state explicit and central to the semantics.

First of all, we need to fix some notation. We write $\mathbb{Z}$ for the set of all integers, and $\mathbb{B} = \{\mathit{false}, \mathit{true}\}$ for the set of all boolean values. (These are the mathematical entities representing integer and boolean expressions. Note that for clarity, $\mathbb{B}$ and $\mathbb{Z}$ are disjoint, *i.e.* we do not use 0 for *false* and 1 for *true* as in the programming language.)

2.1 Introduction to C0

We first introduce the tiny subset of C that we want to consider. We call our language C0 (that name is not unique, see *e.g.* []), and this is the first development stage.

We give the *abstract* syntax here. That means, as opposed to a concrete syntax, it lacks for example parentheses to group expressions, or brackets to group statements, and does not specify operator priorities. Moreover, it is not efficiently parseable (being not regular).

We first give expressions (**Exp**), which are either arithmetic expressions **Aexp** (integer-valued), and boolean expressions **Bexp** (boolean-valued):

Aexp	$a ::= \mathbf{Z} \mid \mathbf{Idt} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2$
Bexp	$b ::= \mathbf{1} \mid \mathbf{0} \mid a_1 == a_2 \mid a_1 < a_2 \mid !b \mid b_1 \,\&\&\, b_2 \mid b_1 \,\|\|\, b_2$
Exp	$e := \mathbf{Aexp} \mid \mathbf{Bexp}$

Here, **Z** are integers; again, our abstract syntax means we do not give a concrete grammar which for integers might look like this

$$\mathbf{Z} ::= -^{?}(0|1|2|3|4|5|6|7|8|9)^{+}$$

which means "an optional minus sign (indicated by $-^{?}$) followed by a non-empty sequence of digits, i.e. characters $0,\ldots,9$ (the non-empty sequence is written as $^{+}$)". **Idt** are the identifiers, *i.e.* variable

names. Concretely, in C these start with a non-digit (an underscore or a letter), followed by a sequence of non-digits or digits.

In concrete examples, we use more relational operators, all of which can be given in terms of the ones given above, and thus can be considered *syntactic sugar*. These are

$$b ::= a_1 \mathrel{!}= a_2 \mid a_1 <= a_2 \mid a_1 > a_2 \mid a1 >= a2$$

In a concrete program, an expression $a_1 \mathrel{!}= a_2$ is parsed in the abstract syntax term $!(a_1 == a_2)$, and $a_1 <= a_2$ is parsed as $a_1 < a_2 \,||\, a_1 == a_2$, and $a_1 > a_2$ as $a_2 < a_1$.

With expressions, we can give statements (**Stmt**). These fall into three groups: basic statements, which are assignments; control statements, which are conditional (**if**) and iteration (**while**); and structured statements, which are the sequencing and the empty statement.

$$\textbf{Stmt} \quad c ::= \textbf{Idt} = \textbf{Exp} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ (b)\ c \mid c_1; c_2 \mid \{\,\}$$

Just like we do not have parentheses to group expressions, we also do not have brackets ({...}) to group statements. Such statements grouped with brackets are called compound statements in C. Also, in the concrete syntax of C, the semicolon is used to terminate basic statements, not to concatenate them; the difference is that in C, we need to write

```
if (x == 0) {x= 99; z= 0; } else { y= z/x; z= 1;}
```

instead of `if (x == 0) { x= 99; z= 0 } else { y= z/x; z= 1}`: both compound statements need to end in a semicolon.

Presently, statements are our programs (and all programs are statements); we do not consider function definitions yet.

2.2 Partial Functions

Partial functions are a fundamental concept of our semantics. A partial function from X to Y is written as $X \rightharpoonup Y$, and maps each x to at most one y. We define partial functions as right-unique relations:

Definition 2.1 (Partial Function) *For two sets X and Y, a* partial function $f : X \rightharpoonup Y$ *is a subset of $f \subseteq X \times Y$ such that for all $x \in X, y_1, y_2 \in Y$*

$$(x, y_1) \in f, (x, y_2) \in f \Longrightarrow y_1 = y_2 \tag{2.1}$$

For a partial function f, the domain *of f is the subset of X where f is defined (returns a value from Y):*

$$dom(f) = \{x \mid \exists y.\, (x, y) \in f\}$$

The right-uniqueness property (2.1) means we can write $f(x)$ for applying f to x and this is well-defined

$$f(x) = y \Longleftrightarrow (x, y) \in f.$$

We furthermore write $f(x) = \bot$ if f is undefined at x, *i.e.* $x \notin dom(f)$). For a function $f : X \rightharpoonup Y$, a value $x \in X$ and a value $n \in Y$, we define the *functional update* of the function f at location x with value n, written as $f[x \mapsto n]$, as the function

$$f[x \mapsto n] \stackrel{def}{=} \{(y, m) \mid x \neq y, (y, m) \in f\} \cup \{(x, n)\}$$

or alternatively as the function which for any $y \in X$ is defined as

$$f[x \mapsto n](y) \stackrel{def}{=} \begin{cases} n & \text{if } x = y \\ f(y) & \text{otherwise} \end{cases}$$

Finite partial functions (functions where the domain X is finite) are also called finite *maps*. To denote maps, we use the notation $\langle x_1 \mapsto n_1, x_2 \mapsto n_2 \rangle$ *etc.*; in particular, we use $\langle\rangle$ for the empty map $\langle\rangle = \emptyset$.

2.3 State

The basis of all semantics (not only the operational semantics) is the *program state*. Formally, the program state is a partial map from *locations* to *values*. The values are what programs evaluate to, or what we can compute. When we expand our language, we will both extend the notion of locations and values, but for the time being we define:

Definition 2.2 (Locations, Values and System State)

The values *are given by integers,* $\boldsymbol{V} \stackrel{def}{=} \mathbb{Z}$

The locations *are given by identifiers,* $\boldsymbol{Loc} \stackrel{def}{=} \boldsymbol{Idt}$

The system state *is a partial map from locations to values:* $\Sigma \stackrel{def}{=} \boldsymbol{Loc} \rightharpoonup \boldsymbol{V}$.

Exercise 2.1 *(i) Give a state which assigns the value* 6 *to the identifier a, and* 2 *to the idenifier c.*

(ii) Which of the following are valid states:

(A) $\langle x \mapsto 1, a \mapsto 3 \rangle$

(B) $\langle x \mapsto y, b \mapsto 6 \rangle$

(C) $\langle x \mapsto y, b \mapsto 6, y \mapsto 2 \rangle$

(D) $\langle x \mapsto 3, b \mapsto 6, y \mapsto 2 \rangle$

(iii) Calculate the following state updates

(A) $\langle x \mapsto 1, a \mapsto 3 \rangle [y \mapsto 1] = ?$

(B) $\langle x \mapsto 1, a \mapsto 3 \rangle [x \mapsto 3] = ?$

(C) $\langle x \mapsto 1, a \mapsto 3 \rangle [x \mapsto 3][y \mapsto 1][x \mapsto 4] = ?$

2.4 Evaluating Expressions

As mentioned above, statements and expressions are always evaluated with respect to an ambient state. Given a state σ, an arithmetic expression a either evaluates to an integer $n \in \mathbb{Z}$ (a value), or an undefined error value $\perp$. We write this as

$$\langle a, \sigma \rangle \rightarrow_{Aexp} n \mid \perp.$$

$$\frac{}{\langle i, \sigma\rangle \to_{Aexp} [\![i]\!]}$$

$$\frac{x \in \mathbf{Idt}, x \in Dom(\sigma), \sigma(x) = v}{\langle x, \sigma\rangle \to_{Aexp} v} \qquad \frac{x \in \mathbf{Idt}, x \notin Dom(\sigma)}{\langle x, \sigma\rangle \to_{Aexp} \bot}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \qquad n_i \in \mathbb{Z}}{\langle a_1 + a_2, \sigma\rangle \to_{Aexp} n_1 + n_2}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma n_2\rangle \qquad n_1 = \bot \text{ or } n_2 = \bot}{\langle a_1 + a_2, \sigma\rangle \to_{Aexp} \bot}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \qquad n_i \in \mathbb{Z}}{\langle a_1 - a_2, \sigma\rangle \to_{Aexp} n_1 - n_2}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \qquad n_1 = \bot \text{ or } n_2 = \bot}{\langle a_1 - a_2, \sigma\rangle \to_{Aexp} \bot}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \qquad n_i \in \mathbb{Z}}{\langle a_1 * a_2, \sigma\rangle \to_{Aexp} n_1 \cdot n_2}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \qquad n_1 = \bot \text{ or } n_2 = \bot}{\langle a_1 * a_2, \sigma\rangle \to_{Aexp} \bot}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \qquad n_i \in \mathbb{Z}, n_2 \neq 0}{\langle a_1 / a_2, \sigma\rangle \to_{Aexp} n_1 \div n_2}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \qquad n_1 = \bot, n_2 = \bot \text{ or } n_2 = 0}{\langle a_1 / a_2, \sigma\rangle \to_{Aexp} \bot}$$

Figure 2.1: Rules to evaluate arithmetic expressions

$$\frac{}{\langle \mathbf{1}, \sigma \rangle \to_{Bexp} true} \qquad \frac{}{\langle \mathbf{0}, \sigma \rangle \to_{Bexp} false}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad n_i \neq \bot, n_1 = n_2}{\langle a_1 == a_2, \sigma \rangle \to_{Bexp} true}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad n_i \neq \bot, n_1 \neq n_2}{\langle a_1 == a_2, \sigma \rangle \to_{Bexp} false}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad n_1 = \bot \text{ or } n_2 = \bot}{\langle a_1 == a_2, \sigma \rangle \to_{Bexp} \bot}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad n_i \neq \bot, n_1 < n_2}{\langle a_1 < a_2, \sigma \rangle \to_{Bexp} true}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad n_i \neq \bot, n_1 \geq n_2}{\langle a_1 < a_2, \sigma \rangle \to_{Bexp} false}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad n_1 = \bot \text{ or } n_2 = \bot}{\langle a_1 < a_2, \sigma \rangle \to_{Bexp} \bot}$$

$$\frac{\langle b, \sigma \rangle \to_{Bexp} true}{\langle !b, \sigma \rangle \to_{Bexp} false} \qquad \frac{\langle b, \sigma \rangle \to_{Bexp} false}{\langle !b, \sigma \rangle \to_{Bexp} true} \qquad \frac{\langle b, \sigma \rangle \to_{Bexp} \bot}{\langle !b, \sigma \rangle \to_{Bexp} \bot}$$

$$\frac{\langle b_1, \sigma \rangle \to_{Bexp} false}{\langle b_1 \,\&\&\, b_2, \sigma \rangle \to_{Bexp} false} \qquad \frac{\langle b_1, \sigma \rangle \to_{Bexp} \bot}{\langle b_1 \,\&\&\, b_2, \sigma \rangle \to_{Bexp} \bot} \qquad \frac{\langle b_1, \sigma \rangle \to_{Bexp} true \qquad \langle b_2, \sigma \rangle \to_{Bexp} t}{\langle b_1 \,\&\&\, b_2, \sigma \rangle \to_{Bexp} t}$$

$$\frac{\langle b_1, \sigma \rangle \to_{Bexp} true}{\langle b_1 \,||\, b_2, \sigma \rangle \to_{Bexp} true} \qquad \frac{\langle b_1, \sigma \rangle \to_{Bexp} \bot}{\langle b_1 \,||\, b_2, \sigma \rangle \to_{Bexp} \bot} \qquad \frac{\langle b_1, \sigma \rangle \to_{Bexp} false \qquad \langle b_2, \sigma \rangle \to_{Bexp} t}{\langle b_1 \,||\, b_2, \sigma \rangle \to_{Bexp} t}$$

Figure 2.2: Rules to evaluate boolean expressions

2.4.1 Arithmetic Expressions

Figure 2.1 gives the rules to evalute arithmetic expressions. Some notational points:

- Note that we distinguish the values, which are integers $\mathbb{Z}$ from the integer literals **Z** as written in the program. Similarly, boolean expressions evaluate to $\mathbb{B}$, and we distinguish the semantic values *true* and *false* from the syntactic literals **1** and **0**. This distinction may seem a little fussy at first, but we need to be careful to distinguish our semantic world from our syntactic one.
- For an integer literal i, $[\![i]\!] \in \mathbb{Z}$ is the corresponding integer in $\mathbb{Z}$.
- $a \div b$ is the integer division of $a, b \in \mathbb{Z}$, and $a \cdot b$ is obviously the product.

2.4.2 Strictness

It is important to realize that if one of the arguments of an arithmetic operator such as $+, -, *, /$ evaluates to $\bot$ (*i.e.* produces an error), than the whole expression fails to evaluate. We call such an operator *strict*:

Definition 2.3 (Strict Function) *A partial function $f : X \rightharpoonup Y$ is* strict *if $f(\bot) = \bot$, i.e. if an undefined argument makes the function value undefined as well.*

For a partial function $f : X_1 \times \ldots \times X_n \rightharpoonup Y$, f is strict in the i-th position if $x_i = \bot$ implies $f(x_1, \ldots, x_n) = \bot$, *i.e.* an undefined argument at i-th position makes the function value undefined as well.

2.4.3 Non-Determinism

Note that an expression or statement could, in theory, evaluate to more than one value or successor state. Indeed, the semantics of full C is non-deterministic [7], but our semantics for C0 is deterministic. This follows from the equivalence with the denotational semantics that we will show in the next section, so we do not prove it separately here.

2.4.4 Boolean Expressions

Similarly, given a state σ, an boolean expressions either evaluates to a *boolean value true* or *false*, or an undefined error value $\bot$. We write this as

$$\langle b, \sigma \rangle \rightarrow_{Bexp} true \mid false \mid \bot$$

Figure 2.2 gives the rules to evaluate boolean expressions. The rules to evaluate the boolean literals, relational operators and negation are no great surprise. However, the rules to evaluate logical conjunction and disjunction deserve closer attention: they specify that if the left argument evaluates to false, the whole conjunction is false (and similarly, if the left argument of a disjunction evaluates to true, the whole disjunction is true), even if the right argument would evaluate to undefined. In other words, conjunction and disjunction are *non-strict* in their second (right) argument. This is the way it is defined in C and most other programming languages, so our rules model this behaviour.

Example 2.1 (Evaluating an Expression) *An evaluation is constructed as an inference tree, from the bottom up. As an example, consider $\sigma \stackrel{def}{=} \{x \mapsto 6, y \mapsto 5\}$. We now want to evaluate the expression*

$(x+y)*(x-y)$ under σ. For this, we first apply the rule for $*$ from Figure 2.1, which means we have to evaluate $x+y$ and $x-y$. To evaluate these, we have to evaluate x and y. These evaluate to 6 and 5, respectively, allowing us to fill in the values for $x+x$ and $x-y$ and, ultimately, the whole expression. Written as an inference tree, we obtain:

$$\dfrac{\dfrac{\langle x,\sigma\rangle \to_{Aexp} 6 \qquad \langle y,\sigma\rangle \to_{Aexp} 5}{\langle x+y,\sigma\rangle \to_{Aexp} 11} \qquad \dfrac{\langle x,\sigma\rangle \to_{Aexp} 6 \qquad \langle y,\sigma\rangle \to_{Aexp} 5}{\langle x-y,\sigma\rangle \to_{Aexp} 1}}{\langle (x+y)*(x-y),\sigma\rangle \to_{Aexp} 11}$$

Exercise 2.2 *As an exercise, let $\sigma \stackrel{def}{=} \{x \mapsto 0, y \mapsto 3, z \mapsto 7\}$ and try evaluating the following expressions and note how the undefinedness propagates (or not):*

$$\langle !(x == 0)\ \&\&\ (z/x == 0), \sigma\rangle \to_{Bexp} ? \tag{2.2}$$

$$\langle (z/x == 0)\ ||\ x == 0, \sigma\rangle \to_{Bexp} ? \tag{2.3}$$

$$\langle y - z*((a+1)/2), \sigma\rangle \to_{Aexp} ? \tag{2.4}$$

2.5 Evaluating Statements

Under a given state σ_1, a statement evaluates to a new state σ_2 or an error $\bot$, written as

$$\langle c, \sigma_1\rangle \to_{Stmt} \sigma_2 \mid \bot$$

Figure 2.3 gives the rules to evaluate statements. It is instructive to see how undefinedness propagates:

- The assignment becomes undefined if the right-hand side is undefined. The left-hand side need not be defined (*i.e.* the location does not need be in the domain of σ).
- The concatenation operator (;) is strict.
- The conditional is strict in the condition (if the condition is undefined, the whole conditional is), but not in the two branches: if the condition evaluates to *true*, the negative branch is not evaluated at all. This is *fundamental* in all programming languages, because the conditional operator is needed to guard against undefinedness, such as in this code fragment:

```
if (x == 0) {
  y= 0;
  } else {
  y= y/x;
  }
```

- Similarly, the iteration is strict in the condition, but not in the body: if the condition evaluates to *false*, the body is not evaluated at all (for very much the same reasons).

2.5.1 Undefinedness

As we have seen, some operations are strict and some are not. However, strictness refers to propagation of undefinedness, so where does undefinedness *originate* from? In the operational semantics, looking at the rules, there are only two rules which *cause* undefinedness:

$$\frac{\langle a,\sigma\rangle \to_{Aexp} n \in \mathbb{Z}}{\langle x = a,\sigma\rangle \to_{Stmt} \sigma[x \mapsto n]} \qquad \frac{\langle a,\sigma\rangle \to_{Aexp} \bot}{\langle x = a,\sigma\rangle \to_{Stmt} \bot}$$

$$\frac{}{\langle \sigma, \{\,\}\rangle \to_{Stmt} \sigma}$$

$$\frac{\langle c_1,\sigma\rangle \to_{Stmt} \sigma' \neq \bot \qquad \langle c_2,\sigma'\rangle \to_{Stmt} \sigma'' \neq \bot}{\langle c_1;c_2,\sigma\rangle \to_{Stmt} \sigma''}$$

$$\frac{\langle c_1,\sigma\rangle \to_{Stmt} \bot}{\langle c_1;c_2,\sigma\rangle \to_{Stmt} \bot}$$

$$\frac{\langle c_1,\sigma\rangle \to_{Stmt} \sigma' \neq \bot \qquad \langle \{c_2\},\sigma'\rangle \to_{Stmt} \bot}{\langle c_1;c_2,\sigma\rangle \to_{Stmt} \bot}$$

$$\frac{\langle b,\sigma\rangle \to_{Bexp} \mathit{true} \qquad \langle c_1,\sigma\rangle \to_{Stmt} \sigma'}{\langle \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2,\sigma\rangle \to_{Stmt} \sigma'}$$

$$\frac{\langle b,\sigma\rangle \to_{Bexp} \mathit{false} \qquad \langle c_2,\sigma\rangle \to_{Stmt} \sigma'}{\langle \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2,\sigma\rangle \to_{Stmt} \sigma'}$$

$$\frac{\langle b,\sigma\rangle \to_{Bexp} \bot}{\langle \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2,\sigma\rangle \to_{Stmt} \bot}$$

$$\frac{\langle b,\sigma\rangle \to_{Bexp} \mathit{false}}{\langle \textbf{while}\ (b)\ c,\sigma\rangle \to_{Stmt} \sigma}$$

$$\frac{\langle b,\sigma\rangle \to_{Bexp} \mathit{true} \qquad \langle c,\sigma\rangle \to_{Stmt} \sigma' \qquad \langle \textbf{while}\ (b)\ c,\sigma'\rangle \to_{Stmt} \sigma''}{\langle \textbf{while}\ (b)\ c,\sigma\rangle \to_{Stmt} \sigma''}$$

$$\frac{\langle b,\sigma\rangle \to_{Bexp} \mathit{true} \qquad \langle c,\sigma\rangle \to_{Stmt} \bot}{\langle \textbf{while}\ (b)\ c,\sigma\rangle \to_{Stmt} \bot}$$

$$\frac{\langle b,\sigma\rangle \to_{Bexp} \bot}{\langle \textbf{while}\ (b)\ c,\sigma\rangle \to_{Stmt} \bot}$$

Figure 2.3: Rules to evaluate statements

(1) division by zero, or

(2) reading from an undefined location *i.e.* an identifier which has not been written to before.

In particular, a non-terminating evaluation is *not* undefined. To wit, consider the evaluation of this program:

$$\langle x = 0; \textbf{while } (x == 0)\ \{\}, \langle\rangle \rangle \to_{Stmt} ?$$

2.6 Equivalence

One application of operational semantics is to reason about program equivalence. (This can be used in compilers to show that certain optimisations are correct.) We say two programs c_0, c_1 are *equivalent* if they affect the same state changes. Of course, this also needs a notion of equivalence for arithmetic and boolean expressions — two expressions are equivalent if they always evaluate to the same value under all states.

Formally:

Definition 2.4 (Equivalence) *Given two arithmetic expressions a_1, a_2, two boolean expressions b_1, b_2, and two programs c_0, c_1 respectively. They are* equivalent *iff:*

$$a_1 \sim_{Aexp} a_2 \quad \textit{iff} \quad \forall \sigma, n.\, \langle a_1, \sigma\rangle \to_{Aexp} n \Leftrightarrow \langle a_2, \sigma\rangle \to_{Aexp} n \tag{2.5}$$

$$b_1 \sim_{Bexp} b_2 \quad \textit{iff} \quad \forall \sigma, b.\, \langle b_1, \sigma\rangle \to_{Bexp} b \Leftrightarrow \langle b_2, \sigma\rangle \to_{Bexp} b \tag{2.6}$$

$$c_0 \sim_{Stmt} c_1 \quad \textit{iff} \quad \forall \sigma, \sigma'.\, \langle c_0, \sigma\rangle \to_{Stmt} \sigma' \Leftrightarrow \langle c_1, \sigma\rangle \to_{Stmt} \sigma' \tag{2.7}$$

For example, `A || (A && B)` and `A` are equivalent; this can be shown by considering all possible combinations of all possible values $\bot, \mathit{false}, \mathit{true}$ for `A`, `B`; the interesting cases here are with `B` evaluating to $\bot$ and `A` evaluating to $\mathit{false}, \mathit{true}$.

For a longer example, we show that

$$\textbf{while } (b)\ c \sim \textbf{if } (b)\ \{c; \textbf{while } (b)\ c\}\ \textbf{else } \{\} \tag{2.8}$$

Proof. To show this, first let $w \stackrel{def}{=} \textbf{while } (b)\ c$. We need to show for arbitrary but fixed σ, σ' that $\langle w, \sigma\rangle \to_{Stmt} \sigma'$ iff $\langle \textbf{if } (b)\ c; w \textbf{ else } \{\}, \sigma\rangle \to_{Stmt} \sigma'$.

The proceeds by a case distinction over how the expressions `b` and `c` evaluate, starting with `b`:

- Case 1: $\langle b, \sigma\rangle \to_{Bexp} \mathit{false}$. Then

$$\langle \textbf{while } (b)\ c, \sigma\rangle \to_{Stmt} \sigma$$
$$\langle \textbf{if } (b)\ \{c; w\} \textbf{ else } \{\}, \sigma\rangle \to_{Stmt} \langle \{\}, \sigma\rangle \to_{Stmt} \sigma$$

- Case 2: $\langle b, \sigma\rangle \to_{Bexp} \mathit{true}$. Then we distinguish two sub-cases:
  - Case 2.1: $\langle c, \sigma\rangle \to_{Stmt} \sigma''$, $\sigma'' \neq \bot$. Also assume that $\langle w, \sigma''\rangle \to_{Stmt} \sigma$ where σ' is possibly $\bot$. Then:

$$\overbrace{\langle \textbf{while } (b)\ c}^{w}, \sigma\rangle \to_{Stmt} \langle c, \sigma\rangle \to_{Stmt} \sigma'' \text{ with } \langle w, \sigma''\rangle \to_{Stmt} \sigma' \text{ we get } \langle w, \sigma\rangle \to_{Stmt} \sigma'$$
$$\langle \textbf{if } (b)\ \{c; w\} \textbf{ else } \{\}, \sigma\rangle \to_{Stmt} \langle \{c; w\}, \sigma\rangle \to_{Stmt} \langle c, \sigma\rangle \to_{Stmt} \sigma'' \text{ with } \langle c, \sigma\rangle \to_{Stmt} \sigma'', \langle w, \sigma''\rangle \to_{Stmt} \sigma'$$

– Case 2.2: $\langle c, \sigma\rangle \to_{Stmt} \bot$. Then:

$$\langle \overbrace{\textbf{while } (b)\ c}^{w}, \sigma\rangle \to_{Stmt} \langle c, \sigma\rangle \to_{Stmt} \bot$$
$$\langle \textbf{if } (b)\ \{c; w\}\ \textbf{else } \{\}, \sigma\rangle \to_{Stmt} \langle \{c; w\}, \sigma\rangle \to_{Stmt} \langle c, \sigma\rangle \to_{Stmt} \bot$$

- Case 3: $\langle b, \sigma\rangle \to_{Bexp} \bot$. Then:

$$\langle \textbf{while } (b)\ c, \sigma\rangle \to_{Stmt} \bot$$
$$\langle \textbf{if } (b)\ \{c; w\}\ \textbf{else } \{\}, \sigma\rangle \to_{Stmt} \bot$$

□

2.7 Summary

- Operational semantics is a way to describe the meaning of a program by its evaluation. The evaluation is expressed by describing the state transition of the program.
- Operational semantics is given by rules (originally, operational semantics was known as *structured operational semantics*). There is one rule for each syntactical construct.
- The operational semantics defines how *expressions* evaluate to *values*, and how *programs* evaluate one state into a successor state.
- Operational semantics is *partial*: not every program evaluates to a successor state, not every expressions evaluates to a value. This is a feature, not a bug — partiality is necessary for Turing equivalence.
- Our operational semantics is *deterministic* — each expression evaluates to at most one value, each statements to at most one successor states.
- Operational semantics can be used to show *equivalence* of programs or expressions.

Chapter 3

Denotational Semantics

In denotational semantics, we give the meaning of each program in terms of a mathematical entity, in particular a *partial function* between states. Hence, the notion of state as defined in Section 2.3 is the starting point.

In general, the denotional semantics is written, for a program c, as $[\![c]\!]$. The denotional semantics is *compositional*, that is the semantics of a structured statement can be given in terms of its components; for example, the semantics of the compound statement is given by composing the semantics of the basic statements:

$$[\![c_1;c_2]\!] = [\![c_1]\!] \circ [\![c_2]\!]$$

This is not the case for the operational semantics.

The denotations of our programs (and expressions) are partial functions, as defined in Section 2.2. In particular:

- each arithmetic expression $a \in \mathbf{Aexp}$ is denoted by a partial function $[\![a]\!]_{\mathscr{A}} : \Sigma \rightharpoonup \mathbb{Z}$,
- each boolean expression $b \in \mathbf{Bexp}$ is denoted by a partial function $[\![b]\!]_{\mathscr{B}} : \Sigma \rightharpoonup \mathbb{B}$, and
- each statement $c \in \mathbf{Stmt}$ is denoted by a partial function $[\![c]\!]_{\mathscr{C}} : \Sigma \rightharpoonup \Sigma$.

3.1 Denotational Semantics of Expressions

Figure 3.1 gives the denotational semantics for expressions. The arithmetic operators are denoted by their semantic counterparts (note that for division, $[\![a_1/a_2]\!]_{\mathscr{A}}$ denotes the *integer division* $[\![a_1]\!]_{\mathscr{A}} \div [\![a_2]\!]_{\mathscr{A}}$). The denotional semantics of boolean expressions is entirely similar, except that disjunction and conjunction are — just like we have seen in the operational semantics — non-strict on the right (in their second argument): if $[\![b_1]\!]_{\mathscr{B}}(\sigma) = \mathit{false}$ for some b_1 and σ, then $[\![b_1 \,\&\&\, b_2]\!]_{\mathscr{B}}(\sigma) = \mathit{false}$ even if $[\![b_2]\!]_{\mathscr{B}}(\sigma)$ is undefined (*i.e.* $[\![b_2]\!]_{\mathscr{B}} = \bot$).

Figure 3.1 gives the semantics as a relation. This relation is, in fact, a partial function:

Lemma 3.1 (Denotational Semantics of Expressions)

(i) The relation $[\![-]\!]_{\mathscr{A}}$ as defined in Figure 3.1 is right-unique, and hence a partial function.

$$[\![a]\!]_{\mathscr{A}} : \mathbf{Aexp} \to (\Sigma \rightharpoonup \mathbb{Z})$$

$$\begin{array}{rcl}
[\![n]\!]_{\mathscr{A}} & = & \{(\sigma, [\![n]\!]) \mid \sigma \in \Sigma\} \\
[\![x]\!]_{\mathscr{A}} & = & \{(\sigma, \sigma(x)) \mid \sigma \in \Sigma, x \in Dom(\sigma)\} \\
[\![a_0 + a_1]\!]_{\mathscr{A}} & = & \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in [\![a_0]\!]_{\mathscr{A}} \wedge (\sigma, n_1) \in [\![a_1]\!]_{\mathscr{A}}\} \\
[\![a_0 - a_1]\!]_{\mathscr{A}} & = & \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in [\![a_0]\!]_{\mathscr{A}} \wedge (\sigma, n_1) \in [\![a_1]\!]_{\mathscr{A}}\} \\
[\![a_0 * a_1]\!]_{\mathscr{A}} & = & \{(\sigma, n_0 \cdot n_1) \mid (\sigma, n_0) \in [\![a_0]\!]_{\mathscr{A}} \wedge (\sigma, n_1) \in [\![a_1]\!]_{\mathscr{A}}\} \\
[\![a_0 / a_1]\!]_{\mathscr{A}} & = & \{(\sigma, n_0 \div n_1) \mid (\sigma, n_0) \in [\![a_0]\!]_{\mathscr{A}} \wedge (\sigma, n_1) \in [\![a_1]\!]_{\mathscr{A}} \wedge n_1 \neq 0\}
\end{array}$$

$$[\![b]\!]_{\mathscr{B}} : \mathbf{Bexp} \to (\Sigma \rightharpoonup \mathbb{B})$$

$$\begin{array}{rcl}
[\![\mathbf{1}]\!]_{\mathscr{B}} & = & \{(\sigma, \mathit{true}) \mid \sigma \in \Sigma\} \\
[\![\mathbf{0}]\!]_{\mathscr{B}} & = & \{(\sigma, \mathit{false}) \mid \sigma \in \Sigma\} \\
[\![a_0 == a_1]\!]_{\mathscr{B}} & = & \{(\sigma, \mathit{true}) \mid \sigma \in \Sigma, (\sigma, n_0) \in [\![a_0]\!]_{\mathscr{A}}(\sigma), (\sigma, n_1) \in [\![a_1]\!]_{\mathscr{A}}, n_0 = n_1\} \\
 & & \cup \{(\sigma, \mathit{false}) \mid \sigma \in \Sigma, (\sigma, n_0) \in [\![a_0]\!]_{\mathscr{A}}(\sigma), (\sigma, n_1) \in [\![a_1]\!]_{\mathscr{A}}, n_0 \neq n_1\} \\
[\![a_0 < a_1]\!]_{\mathscr{B}} & = & \{(\sigma, \mathit{true}) \mid \sigma \in \Sigma, (\sigma, n_0) \in [\![a_0]\!]_{\mathscr{A}}(\sigma), (\sigma, n_1) \in [\![a_1]\!]_{\mathscr{A}}, n_0 < n_1\} \\
 & & \cup \{(\sigma, \mathit{false}) \mid \sigma \in \Sigma, (\sigma, n_0) \in [\![a_0]\!]_{\mathscr{A}}(\sigma), (\sigma, n_1) \in [\![a_1]\!]_{\mathscr{A}}, n_0 \geq n_1\} \\
[\![!b]\!]_{\mathscr{B}} & = & \{(\sigma, \mathit{true}) \mid \sigma \in \Sigma, (\sigma, \mathit{false}) \in [\![b]\!]_{\mathscr{B}}\} \\
 & & \cup \{(\sigma, \mathit{false}) \mid \sigma \in \Sigma, (\sigma, \mathit{true}) \in [\![b]\!]_{\mathscr{B}}\} \\
[\![b_1 \;\&\&\; b2]\!]_{\mathscr{B}} & = & \{(\sigma, \mathit{false}) \mid \sigma \in \Sigma, (\sigma, \mathit{false}) \in [\![b_1]\!]_{\mathscr{B}}\} \\
 & & \cup \{(\sigma, t_2) \mid \sigma \in \Sigma, (\sigma, \mathit{true}) \in [\![b_1]\!]_{\mathscr{B}}, (\sigma, t_2) \in [\![b_2]\!]_{\mathscr{B}}\} \\
[\![b_1 \;||\; b_2]\!]_{\mathscr{B}} & = & \{(\sigma, \mathit{true}) \mid \sigma \in \Sigma, (\sigma, \mathit{true}) \in [\![b_1]\!]_{\mathscr{B}}\} \\
 & & \cup \{(\sigma, t_2) \mid \sigma \in \Sigma, (\sigma, \mathit{false}) \in [\![b_1]\!]_{\mathscr{B}}, (\sigma, t_2) \in [\![b_2]\!]_{\mathscr{B}}\}
\end{array}$$

Figure 3.1: Denotional semantics for expressions

$$[\![c]\!]_{\mathscr{C}} : \mathbf{Stmt} \to (\Sigma \rightharpoonup \Sigma)$$

$$\begin{array}{rcl}
[\![x = a]\!]_{\mathscr{C}} & = & \{(\sigma, \sigma[x \mapsto n]) \mid \sigma \in \Sigma \wedge (\sigma, n) \in [\![a]\!]_{\mathscr{A}}\} \\
[\![c_1; c_2]\!]_{\mathscr{C}} & = & [\![c_1]\!]_{\mathscr{C}} \circ [\![c_2]\!]_{\mathscr{C}} \\
[\![\{\,\}]\!]_{\mathscr{C}} & = & \mathbf{Id}_{\Sigma} \\
[\![\textbf{if } (b)\ c_0 \textbf{ else } c_1]\!]_{\mathscr{C}} & = & \{(\sigma, \sigma') \mid (\sigma, \mathit{true}) \in [\![b]\!]_{\mathscr{B}} \wedge (\sigma, \sigma') \in [\![c_0]\!]_{\mathscr{C}}\} \\
 & & \cup \{(\sigma, \sigma') \mid (\sigma, \mathit{false}) \in [\![b]\!]_{\mathscr{B}} \wedge (\sigma, \sigma') \in [\![c_1]\!]_{\mathscr{C}}\} \\
[\![\textbf{while } (b)\ c]\!]_{\mathscr{C}} & = & \mathit{fix}(\Gamma_{b,c}) \\
 & & \text{where } \Gamma_{b,c}(f) \stackrel{def}{=} \{(\sigma, \sigma') \mid (\sigma, \mathit{true}) \in [\![b]\!]_{\mathscr{B}} \wedge (\sigma, \sigma') \in [\![c]\!]_{\mathscr{C}} \circ f\} \\
 & & \qquad\qquad \cup \{(\sigma, \sigma) \mid (\sigma, \mathit{false}) \in [\![b]\!]_{\mathscr{B}}\}
\end{array}$$

Figure 3.2: Denotional semantics for statements

(ii) The relation $[\![-]\!]_{\mathscr{B}}$ as defined in Figure 3.1 is right-unique, and hence a partial function.

Proof. To show this lemma (for (i)), we have to show right-uniqueness (2.1), *i.e.*: given $(\sigma,n) \in [\![a]\!]_{\mathscr{A}}, (\sigma,m) \in [\![a]\!]_{\mathscr{A}}$ then $n = m'$.

This is shown by *structural induction* over a. The induction base cases are $a \equiv n$ for some literal $n \in \mathbf{Z}$, and $a \equiv x$ for some identifier $x \in \mathbf{Idt}$. We consider the latter: because the system state σ is itself right-unique, if $(n,x) \in \sigma$ and $(m,x) \in \sigma$, then $n = m$.

For the induction step, consider the case $a \equiv a_1 * a_2$. The induction assumption is that if $(\sigma,n_i) \in [\![a_i]\!]_{\mathscr{A}}$ and $(\sigma,m_i) \in [\![a_i]\!]_{\mathscr{A}}$ then $n_i = m_i$ (for $i = 1,2$). Now let $(\sigma,n) \in [\![a_1 * a_2]\!]_{\mathscr{A}}$ with $n = n_1 \cdot n_2$ and $(\sigma,n_1) \in [\![a_1]\!]_{\mathscr{A}}, (\sigma,n_2) \in [\![a_2]\!]_{\mathscr{A}}$. Assume there is $(\sigma,m) \in [\![a_1 * a_2]\!]_{\mathscr{A}}$ with $m = m_1 \cdot m_2$ and $(\sigma,m_1) \in [\![a_1]\!]_{\mathscr{A}}, (\sigma,m_2) \in [\![a_2]\!]_{\mathscr{A}}$. Now, we can apply the induction assumption to $(\sigma,m_1) \in [\![a_1]\!]_{\mathscr{A}}$ and $(\sigma,n_1) \in [\![a_1]\!]_{\mathscr{A}}$ to conclude that $m_1 = n_1$, and similarly, $m_2 = n_2$. Hence $n = n_1 \cdot n_2 = m_1 \cdot m_2 = m$ as required.

The other operators, and the right-uniqueness of $[\![-]\!]_{\mathscr{B}}$ ((ii) above) are proven analogously; there are no hidden traps. □

Lemma 3.1 allows us to write the denotation in a compositional style. Let $s \stackrel{def}{=} \langle a \mapsto 7, b \mapsto 3 \rangle$, then

$$\begin{aligned}
[\![5 * (a+b)]\!]_{\mathscr{A}}(s) &= [\![5]\!]_{\mathscr{A}}(s) \cdot ([\![a+b]\!]_{\mathscr{A}}(s)) \\
&= [\![5]\!]_{\mathscr{A}}(s) \cdot ([\![a]\!]_{\mathscr{A}}(s) + [\![b]\!]_{\mathscr{A}}(s)) \\
&= 5 \cdot (s(a) + s(b)) = 5 \cdot (7+3) = 50
\end{aligned}$$

One thing to notice is that this equational style hides the undefinedness. Consider the simple expression $[\![a/0]\!]_{\mathscr{A}}(s)$. Remember that $[\![a/0]\!]_{\mathscr{A}}(s) = \bot$ is shorthand for $s \notin dom([\![a/0]\!]_{\mathscr{A}})$, *i.e.* the state s does not get mapped to anything at all— the arithmetic expression $[\![a/0]\!]_{\mathscr{A}}$ has no value under s. The notation $x = \bot$ must *not* be read as an equation, but rather as a predicate on x. In particular, we are not allowed to conclude that if $x = \bot$, then $f(x) = \bot$; that only holds for *strict* functions (recall Definition 2.3 on page 15).

Note that all arithmetic expressions are strict, so with some caution we may write equational evaluations like the following, with the same state s as above:

$$\begin{aligned}
[\![(a+b)/(a-7)]\!]_{\mathscr{A}}(s) &= [\![a+b]\!]_{\mathscr{A}}(s) \div [\![a-7]\!]_{\mathscr{A}}(s) \\
&= ([\![a]\!]_{\mathscr{A}}(s) + [\![b]\!]_{\mathscr{A}}(s)) \div ([\![a]\!]_{\mathscr{A}}(s) - [\![7]\!]_{\mathscr{A}}(s)) \\
&= (7+3) \div (7-7) = 10 \div \bot = \bot
\end{aligned}$$

However, disjunction and conjunction are not strict in their second argument, so writing

$$[\![a \,\&\&\, b]\!]_{\mathscr{B}}(s) = [\![a]\!]_{\mathscr{B}}(s) \wedge b(s)$$

requires even more caution. The usual conjunction $\wedge$ only considers two values, and does not deal with undefinedness at all. If we make undefinedness part of the logic, we get a three-valued logic, which makes exactly that difference between strict and non-strict conjunction.[1] Here, our semantic conjunction

[1] In these three-valued logis, conjunction and disjuctions are either defined strict in both arguments, or non-strict in both; the latter are called Kleene logics.

and disjunction have the following truth tables:

$\wedge$	$\bot$	*false*	*true*
$\bot$	$\bot$	$\bot$	$\bot$
false	*false*	*false*	*false*
true	$\bot$	*false*	*true*

$\vee$	$\bot$	*false*	*true*
$\bot$	$\bot$	$\bot$	$\bot$
false	*false*	*false*	*false*
true	$\bot$	*false*	*true*

	$\neg$
$\bot$	$\bot$
false	*true*
true	*false*

It is a bit of a stretch to use $\wedge$ and $\vee$ for this (a classical case of abuse of notation), but introducing a different notation would also be clumsy. We proceed with caution, noting that progress in mathematics is only achieved by careful abuse of notation[2].

Example 3.1 *Consider the state $s = \langle x \mapsto 3, y \mapsto 4\rangle$ and the expression $a = 7 * x + y$. To calculate the denotational semantics as a relation, we can deduce*

$$\begin{array}{rl} & (s,7) \in [\![7]\!]_{\mathscr{A}} \\ & (s,3) \in [\![x]\!]_{\mathscr{A}} \\ (s,7) \in [\![7]\!]_{\mathscr{A}}, (s,3) \in [\![x]\!]_{\mathscr{A}} \Longrightarrow & (s,21) \in [\![7*x]\!]_{\mathscr{A}} \\ & (s,4) \in [\![y]\!]_{\mathscr{A}} \\ (s,21) \in [\![7*x]\!]_{\mathscr{A}}, (s,4) \in [\![y]\!]_{\mathscr{A}} \Longrightarrow & (s,25) \in [\![7*x+y]\!]_{\mathscr{A}} \end{array}$$

We can write this in a more linear style:

$$\dfrac{\dfrac{\begin{array}{l}(s,7) \in [\![7]\!]_{\mathscr{A}} \\ (s,3) \in [\![x]\!]_{\mathscr{A}}\, foo\end{array}}{(s,21) \in [\![7*x]\!]_{\mathscr{A}}} \quad (s,4) \in [\![y]\!]_{\mathscr{A}}\, foo}{(s,25) \in [\![7*x+y]\!]_{\mathscr{A}}}$$

This is the same as the equational derivation written as follows:

$$\begin{aligned} [\![7*x+y]\!]_{\mathscr{A}}(s) &= [\![7*x]\!]_{\mathscr{A}}(s) + [\![y]\!]_{\mathscr{A}}(s) \\ &= [\![7]\!]_{\mathscr{A}}(s) \cdot [\![x]\!]_{\mathscr{A}}(s) + [\![y]\!]_{\mathscr{A}}(s) \\ &= 7 \cdot s(x) + s(y) = 7 \cdot 3 + 4 = 21 + 4 = 25 \end{aligned}$$

Exercise 3.1 *In the style of Example 3.1, calculate the denotational semantics of the expression*

$$b = (7 == x) \,||\, (x/0 == 1)$$

with the state $s = \langle x \mapsto 7\rangle$.

3.2 Denotational Semantics of Statements

A statement c is denoted by a partial function $[\![c]\!]_{\mathscr{C}} : \Sigma \rightharpoonup \Sigma$ from states to states, also called a state transformer. Note that Σ is a partial function itself, so this is a higher-order function. But we know Haskell so this does not scare us.

Going throught the definition in Figure 3.2, the assignment operation denotes the funcional update of the system state.

To denote structured statements, we need to compose partial functions. Partial functions are composed as relations according to the following definition:

[2] "Citation needed" — who said that?

Definition 3.1 (Composition of Relations) *For two relations $R \subseteq X \times Y, S \subseteq Y \times Z$, their* composition *is a relation $R \circ S \subseteq X \times Z$ defined as*

$$R \circ S \stackrel{def}{=} \{(x,z) \mid \exists y \in Y.\, (x,y) \in R \wedge (y,z) \in S\}$$

One might think the empty statement denotes the empty relation $\emptyset$, but that would be wrong — the empty relation denotes *no* state transition at all, but the empty statement maps each state to itself. This is given by the *identity relation* on a set X, defined as

$$\mathbf{Id}_X \stackrel{def}{=} X \times X = \{(x,x) \mid x \in X\} \tag{3.1}$$

The obvious properties of function composition and identity hold, such as associativity of function composition $R_1 \circ (R_2 \circ R_3) = (R_1 \circ R_2) \circ R_3$ and the unit laws $\mathbf{Id} \circ R = R = R \circ \mathbf{Id}$. Note that given partial functions R and S, the domain of $R \circ S$ (considered as a partial function) is at most that of R but maybe smaller, *i.e.* $dom(R \circ S) \subseteq dom(R)$.

The denotation of the conditional is straightforward, but how should while-loops be denoted? Recall that in Chapter 2, we had equation (2.8) which unfolds a while loop, *i.e.* $w \sim$ **if** (b) $\{c; w\}$ **else** $\{\}$ for $w \stackrel{def}{=}$ **while** (b) c. This should hold in denotational semantics as well, so the following equation should hold:

$$[\![w]\!]_{\mathscr{C}} = [\![\textbf{if } (b) \ \{c; w\} \ \textbf{else } \{\}]\!]_{\mathscr{C}} \tag{3.2}$$

This unfolds the loop once. If we unfold the loop arbitrarily often, this should be the semantics of the while loop. To solve a recursive equation like (3.2), we need the semantic device a *fixed point*; this needs some introduction.

3.3 Fixed Points

Definition 3.2 (Fixed Point) *For a partial function $f : X \rightharpoonup X$, a* fixed point *is $x \in X$ such that $x = f(x)$.*

Examples for fixed points are, for $f(x) = \sqrt{x}$, the points 0 and 1, and for $f(x) = x^2$, 0 and 1 as well. But not all functions have a fixed point — examples being *e.g.* $f(x) = x + 1$, which has no fixed point in $\mathbb{Z}$, or $f(x) = \mathbb{P}(X)$, which has no fixed point at all. On the other hand, some functions have more than one fixed point; for a sorting function on lists, all sorted lists are fixed points. Given a function $f : X \rightharpoonup X$, we say $fix(f)$ is the *least* fixed point of f (if it exists). The construction of fixed points is technically a bit intricate, as we need to exactly state the criteria under which functions admit a fixed point, and define some sort of order under which to determine the least fixed point, so we defer this for the time being to Section 3.8.

It should be clear that a recursive equation like (3.2) describes a fixed point, the fixed point in question being the denotation of $[\![w]\!]_{\mathscr{C}}$ for $w \stackrel{def}{=}$ **while** (b) c. More precisely, let Γ denote the unfolding — it takes the denotation of an arbitrary statement s, and unfolds the loop once (note Γ takes b and c, *i.e.* the loop condition and the loop body, as parameters), then:

$$\begin{aligned} \Gamma_{b,c}([\![s]\!]_{\mathscr{C}}) &\stackrel{def}{=} [\![\textbf{if } (b) \ \{c; s\} \ \textbf{else } \{\}]\!]_{\mathscr{C}} \\ [\![w]\!]_{\mathscr{C}} &\stackrel{def}{=} fix(\Gamma_{b,c}) \end{aligned} \tag{3.3}$$

Equation (3.3) is not quite precise yet, because it mixes the abstract syntax (**if** (b) c **else** $\{\}$) with the denotation, but it gives the gist; Figure 3.2 gives the precise definition. Note how $\Gamma_{b,c}$ takes a denotation and returns a denotation

$$\Gamma_{b,c} : (\Sigma \rightharpoonup \Sigma) \to (\Sigma \rightharpoonup \Sigma)$$

so it is a higher-order function, called the *approximation functional*.[3]

We can give a more elementary account of the construction of the fixed point. Given the approximation functional $\Gamma_{b,c}$ from above, the fixed point is constructed with a series of approximations $\Gamma^i : \Sigma \rightharpoonup \Sigma$ which correspond to unfoldings of the loop, starting with the empty relation (no unfolding at all):

$$\begin{aligned} \Gamma^0 &\stackrel{def}{=} \emptyset \\ \Gamma^{i+1} &\stackrel{def}{=} \Gamma_{b,c}(\Gamma^i) \\ [\![w]\!]_{\mathscr{C}} &\stackrel{def}{=} fix(\Gamma) = \bigcup_{i\in\mathbb{N}} \Gamma^i \end{aligned} \tag{3.4}$$

More precisely, Γ^i is the semantics of the loop having been unfolded at most $i-1$ times if the loop condition holds; more precisely, it corresponds to the loop condition being tested i times. Hence, Γ^i is undefined for states s in which the loop has to be unfolded more than $i-1$ times before termination is reached (which includes those states where the loop does not terminate at all). We now describe this in more detail.

3.4 The Fixed Point at Work

Consider the following simple program:

```
x= 0;
while (n > 0) {
  x= x+ n;
  n= n- 1;
  }
```

This obviously calculates the sum from 0 to `n` in `x`. To demonstrate how the semantics of the while-loop is calculated, we look at its construction for several different states.

First, we need to calculate the semantics of the approximation functional, called Γ above. Recall Γ takes a denotation $f : \Sigma \rightharpoonup \Sigma$ and returns another denotation, so for our program we define Γ as

$$\begin{aligned} \Gamma(f) \stackrel{def}{=} \; & \{(\sigma,\sigma) \mid \sigma(n) \le 0\} \cup \\ & \{(\sigma, \sigma[x \mapsto \sigma(x)+\sigma(n)][n \mapsto \sigma(n)-1]) \mid \sigma(n) > 0\} \end{aligned}$$

written in a functional style as

$$\Gamma(f)(\sigma) \stackrel{def}{=} \begin{cases} \sigma & \sigma(n) \le 0 \\ f(\sigma[x \mapsto \sigma(x)+\sigma(n)][n \mapsto \sigma(n)-1]) & \sigma(n) > 0 \end{cases} . \tag{3.5}$$

Now, consider a specific state $s \stackrel{def}{=} \langle x \mapsto 0, n \mapsto 0\rangle$. Then

$$\begin{aligned} \Gamma^0(s) &= \bot \\ \Gamma^1(s) &= \Gamma(\Gamma^0)(s) = \langle x \mapsto 0, n \mapsto 0\rangle \\ \Gamma^2(s) &= \Gamma(\Gamma^1)(s) = \langle x \mapsto 0, n \mapsto 0\rangle \end{aligned}$$

[3] Functional is a somewhat dated term for higher-order functions.

Γ^0 is simple undefined everywhere. $\Gamma(\Gamma^0)(s)$ is s if the loop condition is not satisfied, which is the case here ($n \leq 0$, so $\Gamma^1(s) = s$. For $\Gamma^2(s) = \Gamma(\Gamma^1(s))$, the loop condition is not satisfied either, leading to $\Gamma^2(s) = s$, and the same holds for all other $i \geq 1$. In general, if $\Gamma^i(\sigma) = \sigma'$ then $\Gamma^j(\sigma) = \sigma'$ for all $j > i$: once the loop terminates, nothing changes anymore. Let us consider some other states:

$$\Gamma^0(\langle x \mapsto 0, n \mapsto 3\rangle) = \bot$$
$$\Gamma^1(\langle x \mapsto 0, n \mapsto 3\rangle) = \Gamma^0(\langle x \mapsto 3, n \mapsto 2\rangle) = \bot$$
$$\Gamma^2(\langle x \mapsto 0, n \mapsto 3\rangle) = \Gamma^1(\langle x \mapsto 3, n \mapsto 2\rangle) = \Gamma^0(\langle x \mapsto 5, n \mapsto 1\rangle) = \bot$$
$$\Gamma^3(\langle x \mapsto 0, n \mapsto 3\rangle) = \Gamma^2(\langle x \mapsto 3, n \mapsto 2\rangle) = \Gamma^1(\langle x \mapsto 5, n \mapsto 1\rangle) = \Gamma^0(\langle x \mapsto 6, n \mapsto 0\rangle) = \bot$$
$$\Gamma^4(\langle x \mapsto 0, n \mapsto 3\rangle) = \Gamma^3(\langle x \mapsto 3, n \mapsto 2\rangle) = \Gamma^2(\langle x \mapsto 5, n \mapsto 1\rangle) = \Gamma^1(\langle x \mapsto 6, n \mapsto 0\rangle) = \langle x \mapsto 6, n \mapsto 0\rangle$$

To summarise these calculations in a table:

s	$\Gamma^0(s)$	$\Gamma^1(s)$	$\Gamma^2(s)$	$\Gamma^3(s)$	$\Gamma^4(s)$
$\langle x \mapsto 0, n \mapsto -1\rangle$	$\bot$	$\langle x \mapsto 0, n \mapsto -1\rangle$	$\langle x \mapsto 0, n \mapsto -1\rangle$	$\langle x \mapsto 0, n \mapsto -1\rangle$	$\langle x \mapsto 0, n \mapsto -1\rangle$
$\langle x \mapsto 0, n \mapsto 0\rangle$	$\bot$	$\langle x \mapsto 0, n \mapsto 0\rangle$	$\langle x \mapsto 0, n \mapsto 0\rangle$	$\langle x \mapsto 0, n \mapsto 0\rangle$	$\langle x \mapsto 0, n \mapsto 0\rangle$
$\langle x \mapsto 0, n \mapsto 1\rangle$	$\bot$	$\bot$	$\langle x \mapsto 1, n \mapsto 0\rangle$	$\langle x \mapsto 1, n \mapsto 0\rangle$	$\langle x \mapsto 1, n \mapsto 0\rangle$
$\langle x \mapsto 0, n \mapsto 2\rangle$	$\bot$	$\bot$	$\bot$	$\langle x \mapsto 3, n \mapsto 0\rangle$	$\langle x \mapsto 3, n \mapsto 0\rangle$
$\langle x \mapsto 0, n \mapsto 3\rangle$	$\bot$	$\bot$	$\bot$	$\bot$	$\langle x \mapsto 6, n \mapsto 0\rangle$

As we can see, $\Gamma^i(s)$ is defined if the loop terminates for the state s in $i-1$ steps. Now consider a slight change in our program:

```
x= 0;
while (n != 0) {
  x= x+ n;
  n= n- 1;
  }
```

Now the loop does not terminate for negative numbers anymore:

$$\Gamma^1(\langle x \mapsto 0, n \mapsto -1\rangle) = \Gamma^0(\langle x \mapsto -1, n \mapsto -2\rangle) = \bot$$
$$\Gamma^2(\langle x \mapsto 0, n \mapsto -1\rangle) = \Gamma^1(\langle x \mapsto -1, n \mapsto -2\rangle) = \Gamma^0(\langle x \mapsto -3, n \mapsto -3\rangle = \bot$$
$$\Gamma^3(\langle x \mapsto 0, n \mapsto -1\rangle) = \Gamma^2(\langle x \mapsto -1, n \mapsto -2\rangle) = \Gamma^1(\langle x \mapsto -3, n \mapsto -3\rangle = \Gamma^0(\langle x \mapsto -6, n \mapsto -4\rangle) = \bot$$

As we can see, there is no i such that $\Gamma^i(\langle x \mapsto 0, n \mapsto -1\rangle$ is defined. Operationally, this is because the loop does not terminate for $n < 0$ (because if $n < 0$ then $n - 1 < 0$ as well). It is interesting (if you like these sort of things) to compare $\Gamma^2(\langle x \mapsto 0, n \mapsto -1\rangle)$ and $\Gamma^2(\langle x \mapsto 0, n \mapsto 2\rangle)$. They both equal $\bot$ — they are both undefined, but the former is so because the loop *will never terminate*, whereas the latter is undefined because the loop has *not terminated yet*, *i.e.* it needs to be unfolded further; so semantically, there is no difference between a loop which has not terminated yet, and one which never will. Here is the

table from above for this case (the calculations for the lower rows remain just as they were before):

s	$\Gamma^0(s)$	$\Gamma^1(s)$	$\Gamma^2(s)$	$\Gamma^3(s)$	$\Gamma^4(s)$
$\langle x \mapsto 0, n \mapsto -2\rangle$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\langle x \mapsto 0, n \mapsto -1\rangle$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\langle x \mapsto 0, n \mapsto 0\rangle$	$\bot$	$\langle x \mapsto 0, n \mapsto 0\rangle$	$\langle x \mapsto 0, n \mapsto 0\rangle$	$\langle x \mapsto 0, n \mapsto 0\rangle$	$\langle x \mapsto 0, n \mapsto 0\rangle$
$\langle x \mapsto 0, n \mapsto 1\rangle$	$\bot$	$\bot$	$\langle x \mapsto 1, n \mapsto 0\rangle$	$\langle x \mapsto 1, n \mapsto 0\rangle$	$\langle x \mapsto 1, n \mapsto 0\rangle$
$\langle x \mapsto 0, n \mapsto 2\rangle$	$\bot$	$\bot$	$\bot$	$\langle x \mapsto 3, n \mapsto 0\rangle$	$\langle x \mapsto 3, n \mapsto 0\rangle$
$\langle x \mapsto 0, n \mapsto 3\rangle$	$\bot$	$\bot$	$\bot$	$\bot$	$\langle x \mapsto 6, n \mapsto 0\rangle$

3.5 Properties of the Fixed Point

We can now show that (2.8) actually holds in the denotational semantics, *i.e.* that for $w \stackrel{def}{=} \textbf{while}\ (b)\ c$ we have

$$[\![w]\!]_{\mathscr{C}} = [\![\textbf{if}\ (b)\ \{c; w\}\ \textbf{else}\ \{\}]\!]_{\mathscr{C}} \tag{3.6}$$

Proof.

$$\begin{aligned}
[\![w]\!]_{\mathscr{C}} &= \mathit{fix}(\Gamma_{b,c}) && \text{by def. of } [\![w]\!]_{\mathscr{C}} \\
&= \Gamma_{b,c}(\mathit{fix}(\Gamma_{b,c})) && \text{property of the fixed-point} \\
&= \Gamma_{b,c}([\![w]\!]_{\mathscr{C}}) && \text{by def. of } [\![w]\!]_{\mathscr{C}} \\
&= \{(\sigma,\sigma') \mid (\sigma,\mathit{true}) \in [\![b]\!]_{\mathscr{B}} \wedge (\sigma,\sigma') \in [\![c]\!]_{\mathscr{C}} \circ [\![w]\!]_{\mathscr{C}}\} \\
&\quad \cup \{(\sigma,\sigma) \mid (\sigma,\mathit{false}) \in [\![b]\!]_{\mathscr{B}}\} && \text{by def. of } \Gamma_{b,c} \\
&= \{(\sigma,\sigma') \mid (\sigma,\mathit{true}) \in [\![b]\!]_{\mathscr{B}} \wedge (\sigma,\sigma') \in [\![c;\ w]\!]_{\mathscr{C}}\} \\
&\quad \cup \{(\sigma,\sigma) \mid (\sigma,\mathit{false}) \in [\![b]\!]_{\mathscr{B}} \wedge (\sigma,\sigma) \in [\![\{\}]\!]_{\mathscr{C}}\} && \text{by def. of } [\![c;w]\!]_{\mathscr{C}}, [\![\{\}]\!]_{\mathscr{C}} \\
&= [\![\textbf{if}\ (b)\ \{c;w\}\ \textbf{else}\ \{\}]\!]_{\mathscr{C}} && \text{by def. of } [\![\textbf{if}\ldots]\!]_{\mathscr{C}}
\end{aligned}$$

□

Note how the proof involves only *equational reasoning* using properties of the semantics; we make no reference to evaluation here.

Another interesting property to prove is that the loop condition does not hold in the target state. More precisely: let $w \stackrel{def}{=} \textbf{while}\ (b)\ c$, then $[\![b]\!]_{\mathscr{B}}([\![w]\!]_{\mathscr{C}}(\sigma)) = \mathit{false}$, which we can write in the relational style as

$$(\sigma,\sigma') \in [\![w]\!]_{\mathscr{C}} \Longrightarrow (\sigma',\mathit{false}) \in [\![b]\!]_{\mathscr{B}} \tag{3.7}$$

Proof. This is proven via the fixed point construction. We know $[\![w]\!]_{\mathscr{C}} = \mathit{fix}(\Gamma_{b,c}) = \bigcup_{n\in\mathbb{N}} \Gamma^i$ (in our notation from above). We show by induction over n that for all $n \in \mathbb{N}$, if $(\sigma,\sigma') \in \Gamma^n$ then $(\sigma',\mathit{false}) \in [\![b]\!]_{\mathscr{B}}$:

- The base case is vacuous, since $\Gamma^0 = \emptyset$, so there are no σ,σ' such that $(\sigma,\sigma') \in \Gamma^0$.
- For the step case, assume if $(\sigma,\sigma') \in \Gamma^n$ then $(\sigma',\mathit{false}) \in [\![b]\!]_{\mathscr{B}}$, and consider $(\sigma,\sigma') \in \Gamma^{n+1}(\emptyset)$. Unfolding the definition of Γ^{n+1}, we get either $(\sigma,\sigma) \in \Gamma(\Gamma^n)$ if $(\sigma,\mathit{false}) \in [\![b]\!]_{\mathscr{B}}$ (then the thesis follows with $\sigma' = \sigma$), or $(\sigma,\sigma') \in \Gamma(\Gamma^n)$ if $(\sigma,\mathit{true}) \in [\![b]\!]_{\mathscr{B}}$ and there is σ'' such that $(\sigma,\sigma'') \in [\![c]\!]_{\mathscr{C}}$ and $(\sigma'',\sigma') \in \Gamma^n$, but then we can use the induction assumption to derive $(\sigma',\mathit{false}) \in [\![b]\!]_{\mathscr{B}}$.

□

3.6 Undefinedness

There is an important distinction in the way undefinedness is handled in the operational and in the denotational semantics. In the operational semantics, undefined was handled *explicitly*: arithmetic expressions evaluated to a number, or $\bot$, so for example $n/0$ evaluates to $\bot$ for all states σ. In the denotational semantics, undefinedness is handled *implictly:* $n/0$ is not mapped to *anything*, *i.e.* $[\![n/0]\!]_{\mathscr{A}} = \emptyset$.[4]

Moreover, the denotational semantics handles undefinedness (division by zero, access to uninitialised variables) and non-termination uniformly, whereas the operational semantics distinguishes between the two: the former is modelled by $\bot$, the latter by a non-terminating rule evaluation.

3.7 Summary

- In denotational semantics, we map each expression and statement to a mathematical entity, its *denotation*. The denotations are partial functions between system states for statements, and partial functions from system states to integers or booleans for expressions.
- The denotational semantics is *compositional*: we can compose the meaning of a whole program by composing the meaning (denotation) of its components.
- The denotation of the while-loop requires the mathematical construction of a fixed point; essentially, the loop is unfolded as often as required until it terminates.
- In our denotational semantics, erroneous expressions such as division by zero, and non-terminating while-loops are handled by the partiality of the semantics: both are simply undefined. This is in contrast to the operational semantics, where erroneous expressions evaluate to an error element, $\bot$, but non-terminating while-loop are non-terminating evaluation sequences.

Of course, the question presenting itself immediately is, are operational and denotational semantics equivalent? In lieu of the missing Chapter 4, we refer the intrigued reader to [8, Chapter 5.3].

3.8 Appendix: Constructing Fixed Points — cpos and lubs

As mentioned above, there is no guarantee that the least fixed point of the construction (3.4) always exists. Clearly, some functions do not have a fixed point, so we need a way to determine which functions admit fixed points and which not. We look into the necessary mathematical constructions in this section; it is not really necessary in order to understand the denotational semantics, so the less inclined reader may want to skip this on first reading. For the full technical details, readers may want to refer to *e.g.* [8, Chapter 4].

Technically, these constructions arise from the theory of complete partial orders (cpos) which arose from the work of Dana Scott to construct a model for the Lambda-calculus. We need some preliminaries first.

Definition 3.3 (Partial Order) *A* partial order *on a set P is a relation $\sqsubseteq \subset P \times P$ which is*

- *reflexive, i.e. $x \sqsubseteq x$ for all $x \in P$;*

[4] To add to the confusion, we use the symbol $\bot$ in two slightly different ways — as a symbol for undefinedness in the operational semantics, and as $f(x) = \bot$ to denote $x \notin dom(f)$ in the denotational semantics. Remember what we said about careful abuse of notation?

- *transitive, i.e. if* $x \sqsubseteq y, y \sqsubseteq z$ *then* $x \sqsubseteq z$*;*
- *anti-symmetric, i.e. if* $x \sqsubseteq y$ *and* $y \sqsubseteq x$ *then* $x = y$*.*

Definition 3.4 (ω-Chain) *An ω-chain in a partial order $(P, \sqsubseteq)$ is given by a set of elements $X = \{x_i \in P\}_{i \in \omega}$ such that*

$$x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq x_3 \ldots$$

Here, $\omega = \{0, 1, 2, \ldots\}$ is the smallest infinite ordinal, or the set of natural numbers.

Definition 3.5 (Least Upper Bound) *Given an ω-chain X in a partial order P, then $p \in P$ is the* least upper bound *of X, written $p = \bigsqcup X$, iff*

(i) $\forall q \in X.q \sqsubseteq p$ *(p is an upper bound of the chain), and*

(ii) $\forall r.(\forall q \in X.q \sqsubseteq r) \Longrightarrow q \sqsubseteq r$ *(p is smaller than all other upper bounds r of X).*

Definition 3.6 (Complete Partial Order) *A* complete partial order *(cpo) $(X, \sqsubseteq)$ is a partial order $(P, \sqsubseteq)$ such that all ω-chains X have a least upper bound $\bigsqcup X \in P$.*

Word of warning here: the terminology is a bit of a mess. Some authors require cpo's to have a least element (this was Scott's original definition I believe), while others do not; for disambiguation, the terms bottomless cpos and pointed cpos (for those without and with a least element) are sometimes used. Further, completeness can be defined not in terms of lubs of ω-chains but rather in terms of least upper bounds of directed sets (where a directed set is partially ordered set $(P, \sqsubseteq)$ such that for all $x, y \in P$ there is $z \in P$ such that $x \in P, y \in P$). These two definitions are equivalent: clearly, an ω-chain is a directed set, but a directed set can also be given in terms of ω-chains such that their least upper bounds are equal.

Given a cpo $(X, \sqsubseteq)$ without a least element, we can "adjoin" a least element (meaning we add an element not already in the set). This element is called $\bot$, and we write $(X, \sqsubseteq)_\bot$ for the resulting pointed cpo.

Lemma 3.2 (Discrete cpo) *Given any set X, the* discrete cpo *is given by $(X, \mathbf{Id}_X)$.*

Proof. The identity relation is trivially a partial order, and since chains are most of length 1 they also have a trivial least upper bound. □

Definition 3.7 (Monotone and Continuous Functions) *Given two cpos $(D, \sqsubseteq)$ and $(E, \sqsubseteq)$, a function $f : D \to E$ is called*

- monotonic *iff $x \sqsubseteq y \Longrightarrow f(x) \sqsubseteq f(y)$ (it preserves the order structure) and*
- continuous *iff moreover for all ω-chains X in D, $\bigsqcup f(x_i) = f(\bigsqcup X)$ (it preserves least upper bounds).*

A function which is both monotonic and continuous is called admissible.

Given two cpos $(D, \sqsubseteq)$ and $(E, \sqsubseteq)$, the set of all functions $D \to E$ as a partial order defined "pointwise" as follows: $f \sqsubseteq g$ iff $\forall x. f(x) \sqsubseteq g(x)$. This set is complete as well (*i.e.* forms a cpo); given an ω-chain of functions $F = \{f_i\}$, its least upper bound is the function defined as $(\bigsqcup F)(x) \stackrel{def}{=} \bigsqcup f_1(x) \sqsubseteq f_2(x) \sqsubseteq f_3(x) \ldots$.

An interesting point to note here is that the set of all *partial* functions from D to E is equivalent (bijective) to the set of all total functions from X to $Y_\perp$, if we map x to $\perp$ when $f(x)$ is not defined. (This is the deeper reason for the notation $f(x) = \perp$ used above.) Thus, when working with cpos we can use total functions between them, and still handle partiality. Also note that if $(D, \sqsubseteq)$ and $(E, \sqsubseteq)$ are discrete cpos then functions are ordered in terms of definedness, *i.e.* $f \subseteq g$ with f, g considered as relations iff $f \sqsubseteq g$ iff when $f(x)$ is defined for $x \in D$ then so is $g(x)$, and $f(x) = g(x)$.

In this context, monotonicity means that operations "preserve definedness", but what does continuity mean? The intuition behind continuitity is that operations have "finite arity", *i.e.* only depend on a finite number of arguments. This is crucial because for finite chains, the lub is always the largest element, so preservation is given by monotonicity; only for infinite chains, continuity is an extra requirement. Cpos were invented to model computable functions, so computing something on an infinite set X means that we compute on all finite subsets of X and take its union.

We are now in a position to construct fixed points. This is known as the Kleene fixed-point theorem:

Theorem 3.3 (Kleene Fixed-Point Theorem) *Given a cpo $(X, \sqsubseteq)$ with a least element $\perp \in X$, and a continuous partial function $f : X \rightharpoonup X$, the least fixed point of f is given as the least upper bound*

$$\mathit{fix}(f) = \bigsqcup_{n \in \omega} f^n(\perp)$$

Proof. We build an ω-chain by starting with $\perp$ and applying f:

$$F \stackrel{def}{=} \perp \sqsubseteq f(\perp) \sqsubseteq f^2(\perp) \sqsubseteq f^3(\perp) \sqsubseteq \ldots$$

We then need to show that $\bigsqcup F$ is the least fixed point of f:

- it is a fixed point: f is continuous, hence

  $$f(\bigsqcup F) = f(\bigsqcup_{i \in \omega} f^i(\perp)) = \bigsqcup_{i \in \omega} f(f^i(\perp)) = \bigsqcup_{i \in \omega} f^{i+1}(\perp) = \bigsqcup_{i \in \omega} f^i(\perp) = \bigsqcup F$$

  In the fourth step, we have the chain $f^1(\perp) \sqsubseteq f^2(\perp) \sqsubseteq \ldots$; since $\perp \sqsubseteq f(\perp)$ always holds we can add $\perp$ to this chain without changing its least upper bound.

- it is the least fixed point: assume there is another fixed point $x = f(x)$, then $\perp \sqsubseteq x$, hence $f(\perp) \sqsubseteq f(x) = x$, hence $f^i(\perp) \sqsubseteq x$, so x is an upper bound of F, and hence $\bigsqcup F \sqsubseteq x$.

□

To apply this theory to the denotational semantics of our language, it remains to be shown that all our functions are admissible, in particular the approximation functional Γ.

We start with making our data types into cpos. **Loc** and **V** are discrete cpos, and so $\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{V}$ is a cpo as well, ordered by definedness.

We then show that the denotations $[\![a]\!]_{\mathscr{A}} : \mathbf{Loc} \rightharpoonup \mathbf{V}$ and $[\![b]\!]_{\mathscr{B}} : \mathbf{Loc} \rightharpoonup \mathbb{B}$ are admissible, for any $a \in \mathbf{Aexp}$ and $b \in \mathbf{Bexp}$. This is done by structural induction on a and b, respectively. For monotonicity, we have to show that if $\sigma \sqsubseteq \tau$, then $[\![a]\!]_{\mathscr{A}}(\sigma) \sqsubseteq [\![a]\!]_{\mathscr{A}}(\tau)$. We consider two salient cases:

- For $x \in \mathbf{Idt}$ and $\sigma \sqsubseteq \tau$ if $x \in \mathit{dom}(\sigma)$ then $x \in \mathit{dom}(\tau)$ as well and $\sigma(x) = \tau(x)$, hence $[\![x]\!]_{\mathscr{A}}(\sigma) = [\![x]\!]_{\mathscr{A}}(\tau)$.

- For $a_1, a_2 \in \mathbf{Aexp}$, assume $[\![a_i]\!]_{\mathscr{A}}(\sigma) \sqsubseteq [\![a_i]\!]_{\mathscr{A}}(\tau)$ for $i = 1,2$ (which boils down to $[\![a_i]\!]_{\mathscr{A}}(\sigma) = [\![a_i]\!]_{\mathscr{A}}(\tau)$), then show $[\![a_1 + a_2]\!]_{\mathscr{A}}(\sigma) \sqsubseteq [\![a_1 + a_2]\!]_{\mathscr{A}}(\tau)$ which boils down $[\![a_1 + a_2]\!]_{\mathscr{A}}(\sigma) = [\![a_1 + a_2]\!]_{\mathscr{A}}(\tau)$.

Showing continuity is even less exciting. Chains in Σ consist of states $\sigma_0 \sqsubseteq \sigma_1 \sqsubseteq \ldots$ such that σ_{i+1} is defined in more locations than σ_i. Preserving the least upper bound is nearly trivial, since each operation only takes one $\sigma \in \Sigma$ as argument.

Finally, to show that the approximation functional Γ is admissible, we show that $[\![c]\!]_{\mathscr{C}} : \mathbf{Stmt} \to \Sigma \rightharpoonup \Sigma$ is admissible for all c except $c \equiv \textbf{while}\ (b)\ p$. This is done by case distinction on c. For monotonicity, note that the definedness of the state σ never decreases for any command.

Chapter 5

Floyd-Hoare Logic

In this chapter, we introduce a third semantics, or another mathematical view of "what a program does". This *axiomatic semantics*, better known as *Floyd-Hoare logic*, differs from the operational semantics (which formalises the execution of a program) and denotational semantics (which by translating a program into a mathematical entity formalises the exact meaning of the program) in that it formalises the result of the program (*i.e. what* the program does, not *how* the program does it).

5.1 Why Another Semantics?

Why do why need another kind of semantics anyway, apart from mathematical curiosity? Well, "dreimal ist Bremer Recht" and all that — there is general enhanced confidence to be gained by giving a third view of the elusive "meaning" of a program and showing it coincides with the other two.

Moreover, this semantics is specifically suited to state and prove *properties* about programs. Consider again the example program in Figure 5.1 (well known from earlier on page 9). It computes the factorial of the input variable n in the variable p, but how can we state and prove that?

We could calculate the semantics, *e.g.* using the denotational semantics, but we will run into three difficulties:

(i) First, the calculated semantics — both operational and denotational — is a very large term indeed, and it is hard to see how that term would imply the simple equality $p = n!$ that we want to prove. In other words, the two semantics we have introduced do not scale for proving properties of larger[1] programs.

(ii) Second, the denotational semantics of the while-loop is hard to handle. It calculuates a fixed point— how can we deal with that?

(iii) Third, variables may change their value, so assertions only hold in a specific program state. For example, the property $p = n!$ only holds once the while-loop has finished, not before.

Floyd-Hoare logic deals with these problems by *abstraction*. Instead of calculating every tiny change of every variable in the state, it allows us to state properties about program variables at certain points in the excecution, prove that these hold, and from that prove properties about the whole program.

[1] Even though the example program is hardly large— imagine calculating the semantics of a couple of thousand lines of C0.

```
p= 1;
c= 1;
while (c <= n) {
  p = p * c;
  c = c + 1;
}
```

Figure 5.1: The example program: calculates the factorial.

5.2 Basic Ingredients of Floyd-Hoare Logic

The basic ingredients of the Floyd-Hoare logic are:

- a language of state-based *assertions*, which allow us to specify properties of the program's state on an abstract level,
- a formalisation of program properties using *Floyd-Hoare tripels*, which specify properties which hold before and after a program is run (pre- and postcondition);
- and a *calculus* by which we can *prove* such properties without having to calculate the whole semantics of a program.

Thus, Floyd-Hoare logic translates the semantics of a program into a logical language. The big questions we will have to deal with are how to handle state change (the assignment statement) and iteration (while-loops) — more on that later. We first review the language of assertions, and what it means for assertions to hold.

5.2.1 Assertions

Assertions are essentially boolean expressions (Section 2.1) extended with a few key concepts:

- *Logical variables* which as opposed to program variables are not stateful, *i.e.* their value is given by an interpretation and cannot be changed. The set of logical variables is written as **Var**, and by convention we use capital names for them in our examples; in our implementation, logical and program variables are distinguished by static analysis (*i.e.* typing the program).
- Logical predicates and functions which are defined externally, and which represent the models used to specify the behaviour of the program (more on that later). Examples of these are the factorial, written as $n!$, or the summation $\sum_{i=1}^{n} i$.
- Implication and universal/existential quantification (which allows us to write down non-executable assertions) over logical variables.

We define the sets of assertions, **Assn**, and extended arithmetic expressions **Aexpv** by extending the defintions of **Bexp** and **Aexp** as follows:

$$\begin{array}{ll} \textbf{Aexpv} & a ::= \mathbf{Z} \mid \mathbf{Idt} \mid \mathbf{Var} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \mid a_1 / a_2 \mid f(e_1, \ldots, e_n) \\ \textbf{Assn} & b ::= \mathbf{1} \mid \mathbf{0} \mid a_1 == a_2 \mid a_1 < a_2 \mid !b \mid b_1 \,\&\&\, b2 \mid b_1 \,||\, b_2 \mid b_1 --> b_2 \mid \\ & \quad p(e_1, \ldots, e_n) \mid \textbf{\textbackslash forall}\, v.\, b \mid \textbf{\textbackslash exists}\, v.\, b \end{array}$$

$$[\![a]\!]_{\mathscr{A}_V} : \mathbf{Aexpv} \to \mathbf{Env} \to \mathbf{Intprt} \to (\Sigma \rightharpoonup \mathbf{N})$$

$$\begin{aligned}
[\![n]\!]^{\Gamma,I}_{\mathscr{A}_V} &= \{(\sigma,n) \mid \sigma \in \Sigma\} \\
[\![x]\!]^{\Gamma,I}_{\mathscr{A}_V} &= \{(\sigma,\sigma(x)) \mid \sigma \in \Sigma, x \in Dom(\sigma)\} \\
[\![v]\!]^{\Gamma,I}_{\mathscr{A}_V} &= \{(\sigma,I(x)) \mid \sigma \in \Sigma, v \in Dom(I)\} \\
[\![a_0 + a_1]\!]^{\Gamma,I}_{\mathscr{A}_V} &= \{(\sigma,n_0+n_1) \mid (\sigma,n_0) \in [\![a_0]\!]^{\Gamma,I}_{\mathscr{A}_V} \wedge (\sigma,n_1) \in [\![a_1]\!]^{\Gamma,I}_{\mathscr{A}_V}\} \\
[\![a_0 - a_1]\!]^{\Gamma,I}_{\mathscr{A}_V} &= \{(\sigma,n_0-n_1) \mid (\sigma,n_0) \in [\![a_0]\!]^{\Gamma,I}_{\mathscr{A}_V} \wedge (\sigma,n_1) \in [\![a_1]\!]^{\Gamma,I}_{\mathscr{A}_V}\} \\
[\![a_0 * a_1]\!]^{\Gamma,I}_{\mathscr{A}_V} &= \{(\sigma,n_0*n_1) \mid (\sigma,n_0) \in [\![a_0]\!]^{\Gamma,I}_{\mathscr{A}_V} \wedge (\sigma,n_1) \in [\![a_1]\!]^{\Gamma,I}_{\mathscr{A}_V}\} \\
[\![a_0 / a_1]\!]^{\Gamma,I}_{\mathscr{A}_V} &= \{(\sigma,n_0 \div n_1) \mid (\sigma,n_0) \in [\![a_0]\!]^{\Gamma,I}_{\mathscr{A}_V} \wedge (\sigma,n_1) \in [\![a_1]\!]^{\Gamma,I}_{\mathscr{A}_V} \wedge n_1 \neq 0\} \\
[\![f(a_1,\ldots,a_n)]\!]^{\Gamma,I}_{\mathscr{A}_V} &= \{(\sigma,\Gamma(f)(v_1,\ldots,v_n)) \mid (\sigma,v_i) \in [\![a_i]\!]^{\Gamma,I}_{\mathscr{A}_V}\}
\end{aligned}$$

$$[\![a]\!]_{\mathscr{B}} : \mathbf{Bexp} \to \mathbf{Env} \to \mathbf{Intprt} \to (\Sigma \rightharpoonup \mathbb{B})$$

$$\begin{aligned}
[\![\mathbf{0}]\!]^{\Gamma,I}_{\mathscr{B}_V} &= \{(\sigma,\mathit{false}) \mid \sigma \in \Sigma\} \\
[\![\mathbf{1}]\!]^{\Gamma,I}_{\mathscr{B}_V} &= \{(\sigma,\mathit{true}) \mid \sigma \in \Sigma\} \\
[\![a_0 == a_1]\!]^{\Gamma,I}_{\mathscr{B}_V} &= \{(\sigma,\mathit{true}) \mid \sigma \in \Sigma, (\sigma,n_0) \in [\![a_0]\!]^{\Gamma,I}_{\mathscr{A}_V}(\sigma), (\sigma,n_1) \in [\![a_1]\!]^{\Gamma,I}_{\mathscr{A}_V}, n_0 = n_1\} \\
&\quad \cup \{(\sigma,\mathit{false}) \mid \sigma \in \Sigma, (\sigma,n_0) \in [\![a_0]\!]^{\Gamma,I}_{\mathscr{A}_V}(\sigma), (\sigma,n_1) \in [\![a_1]\!]^{\Gamma,I}_{\mathscr{A}_V}, n_0 \neq n_1\} \\
[\![a_0 < a_1]\!]^{\Gamma,I}_{\mathscr{B}_V} &= \{(\sigma,\mathit{true}) \mid \sigma \in \Sigma, (\sigma,n_0) \in [\![a_0]\!]^{\Gamma,I}_{\mathscr{A}_V}(\sigma), (\sigma,n_1) \in [\![a_1]\!]^{\Gamma,I}_{\mathscr{A}_V}, n_0 < n_1\} \\
&\quad \cup \{(\sigma,\mathit{false}) \mid \sigma \in \Sigma, (\sigma,n_0) \in [\![a_0]\!]^{\Gamma,I}_{\mathscr{A}_V}(\sigma), (\sigma,n_1) \in [\![a_1]\!]^{\Gamma,I}_{\mathscr{A}_V}, n_0 \geq n_1\} \\
[\![!b]\!]^{\Gamma,I}_{\mathscr{B}_V} &= \{(\sigma,\mathit{false}) \mid \sigma \in \Sigma, (\sigma,\mathit{true}) \in [\![b]\!]^{\Gamma,I}_{\mathscr{B}_V}\} \\
&\quad \cup \{(\sigma,\mathit{true}) \mid \sigma \in \Sigma, (\sigma,\mathit{false}) \in [\![b]\!]^{\Gamma,I}_{\mathscr{B}_V}\} \\
[\![b_1 \,\&\&\, b2]\!]^{\Gamma,I}_{\mathscr{B}_V} &= \{(\sigma,\mathit{false}) \mid \sigma \in \Sigma, (\sigma,\mathit{false}) \in [\![b_1]\!]^{\Gamma,I}_{\mathscr{B}_V}\} \\
&\quad \cup \{(\sigma,t_2) \mid \sigma \in \Sigma, (\sigma,\mathit{true}) \in [\![b_1]\!]^{\Gamma,I}_{\mathscr{B}_V}, (\sigma,t_2) \in [\![b_2]\!]^{\Gamma,I}_{\mathscr{B}_V}\} \\
[\![b_1 \,||\, b_2]\!]^{\Gamma,I}_{\mathscr{B}_V} &= \{(\sigma,\mathit{true}) \mid \sigma \in \Sigma, (\sigma,\mathit{true}) \in [\![b_1]\!]^{\Gamma,I}_{\mathscr{B}_V}\} \\
&\quad \cup \{(\sigma,t_2) \mid \sigma \in \Sigma, (\sigma,\mathit{false}) \in [\![b_1]\!]^{\Gamma,I}_{\mathscr{B}_V}, (\sigma,t_2) \in [\![b_2]\!]^{\Gamma,I}_{\mathscr{B}_V}\} \\
[\![p(e_1,\ldots,e_n)]\!]^{\Gamma,I}_{\mathscr{B}_V} &= \{(\sigma,\Gamma(p)(v_1,\ldots,v_n)) \mid (\sigma,v_i) \in [\![e_i]\!]^{\Gamma,I}_{\mathscr{A}_V}\} \\
[\![b_1 --> b_2]\!]^{\Gamma,I}_{\mathscr{B}_V} &= \{(\sigma,\mathit{true}) \mid (\sigma,\mathit{false}) \in [\![b_1]\!]^{\Gamma,I}_{\mathscr{B}_V}\} \\
&\quad \cup \{(\sigma,t_2) \mid (\sigma,\mathit{true}) \in [\![b_1]\!]^{\Gamma,I}_{\mathscr{B}_V}, (\sigma,t_2) \in [\![b_2]\!]^{\Gamma,I}_{\mathscr{B}_V}\} \\
[\![\mathbf{\backslash forall}\, v; b]\!]^{\Gamma,I}_{\mathscr{B}_V} &= \forall x \in \mathbb{Z}.(\sigma,\mathit{true}) \in [\![b]\!]^{\Gamma,I[x\mapsto v]}_{\mathscr{B}_V} \\
[\![\mathbf{\backslash exists}\, v; b]\!]^{\Gamma,I}_{\mathscr{B}_V} &= \exists x \in \mathbb{Z}.(\sigma,\mathit{true}) \in [\![b]\!]^{\Gamma,I[x\mapsto v]}_{\mathscr{B}_V}
\end{aligned}$$

Figure 5.2: Denotional semantics for assertions

In what follows we use a more mathematical notation for assertions — but this is merely lexical sugar (*i.e.* we just replace the symbols):

$$\textbf{Assn} \qquad b ::= \mathit{true} \mid \mathit{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b2 \mid b_1 \vee b_2 \mid b_1 \longrightarrow b_2 \mid p(e_1, \ldots, e_n) \mid \forall v.\, b \mid \exists v.\, b$$

Logical variables, however, deserve some explanation. Essentially, they allow us to formulate specifications which are invariant over the state. For example, to specify that the statement `x= x+1` increases the value of `x`, we need to specify somehow the value of `x` before and after the statement. We do so by using a logical variable, say N: if the value of N before the statement is equal to `x`, than the value of `x` after the statement should now be larger than the value of N (which has not changed).

An assertion $b \in \textbf{Assn}$ holds in a state $\sigma \in \Sigma$ if its denotational semantics evaluates to *true*. To make this precise, we need to extend the denotational semantics for the missing constructs — that is easy, except that it requires that we give a meaning for the logical functions and predicates, and it moreover requires that we assign a value to the logical variables. This is usual in formal logic: to evaluate a formula, one first assigns values to the variables occuring in the formula, then calculates the evaluation. The formula $a = 4 \wedge b < 5$ is neither true nor false, but if we assign 4 to a and 6 to b, then it becomes *false*.

To define the denotational semantics of an assertion b, we further need an environment which maps the functions and predicates to a semantic meaning. This assigns a meaning to symbols like !; it remains fixed and very much in the background, so much so that we for the time being will omit it from our definitions (a bit like an implicit argument in the Scala programming langauge). We will later come back to the environment, however. Note how the environments map logical functions and predicates to total functions which do not take the current state σ as a parameter, and hence do not depend on it; they are stateless (or pure).

The formal definitions are as follows:

Definition 5.1 (Interpretation and Environment)

An interpretation $I \in \textbf{\textit{Intprt}}$ *is a partial map* $I : \textbf{\textit{Var}} \rightharpoonup \mathbb{Z}$ *which assigns integer values to logical variables.*

An environment $\Gamma \in \textbf{\textit{Env}}$ *maps*

- *each n-ary logical function* f *to an n-ary function* $\Gamma(f) : \mathbb{Z}^n \rightharpoonup \mathbb{Z}$*, and*
- *each n-ary logical predicate* p *to an n-ary predicate* $\Gamma(p) : \boldsymbol{Z}^n \rightharpoonup \mathbb{B}$.

Figure 5.2 gives the additional equations to interpret the new constructs. Recall from Section 2.2 our notations for partial functions; in particular, for an interpretation I we write $I[x \mapsto n]$ for updating the interpretation at the variable x with the (new) value n.

5.2.2 Floyd-Hoare Tripels, Partial and Total Correctness

We can now define what it means for an assertion to *hold* (*i.e.* to be true), with respect to a state and an assignment (omitting the environment, as advertised above). Formally, an assertion $b \in \textbf{Assn}$ *holds* in a state σ with an assignment I, written $\sigma \models^I b$,

$$\sigma \models^I b \text{ iff } [\![b]\!]^I_{\mathscr{B}v}(\sigma) = \mathit{true}. \tag{5.1}$$

The central notion of the Floyd-Hoare logic are *Floyd-Hoare triples* (also sometimes called partial/total correctness assertions), given as $\{P\}\,c\,\{Q\}$ and $[P]\,c\,[Q]$, where $P,Q \in \mathbf{Assn}$ and $c \in \mathbf{Stmt}$. Partial correctness means that if the programs starts in a state where the precondition P holds, and it terminates, then it does so in a state which satisfies the postcondition Q; total correctness means that if the program starts in a state where the precondition P holds, then it must terminate in a state where the postcondition Q holds. So total correctness is essentially partial correctness plus termination; in other words, for partial correctness, the termination of the program c is precondition, and for total correctness, it is part of the requirement.

We now define this formally. We write $\models \{P\}\,c\,\{Q\}$ to mean that the Hoare triple $\{P\}\,c\,\{Q\}$ holds, and define:

$$\models \{P\}\,c\,\{Q\} \iff \forall I.\,\forall \sigma.\,\sigma \models^I P \wedge \exists \sigma'.\,(\sigma,\sigma') \in [\![c]\!]_{\mathscr{C}} \Longrightarrow \sigma' \models^I Q \tag{5.2}$$

$$\models [P]\,c\,[Q] \iff \forall I.\,\forall \sigma.\,\sigma \models^I P \Longrightarrow \exists \sigma'.\,(\sigma,\sigma') \in [\![c]\!]_{\mathscr{C}} \wedge \sigma' \models^I Q \tag{5.3}$$

Points to note:

- The Hoare triple $\models \{\mathit{true}\}$ **while** $(\mathbf{1})\ \{\}\,\{\mathit{false}\}$ holds, because even though for all states σ (and interpretations I) we have $\sigma \models^I \mathit{true}$, there is no state σ' such that $(\sigma,\sigma') \in [\![$**while** $(\mathbf{1})\ \{\}]\!]_{\mathscr{C}}$; if there were such a state σ', it would have to satisfy the impossible, $\sigma' \models^i \mathit{false}$.
- For exactly the same reason, $\models [\mathit{true}]$ **while** $(\mathbf{1})\ \{\}\,[\mathit{false}]$ does not hold.
- However, both $\models \{\mathit{false}\}$ **while** $(\mathbf{1})\ \{\}\,\{\mathit{true}\}$ and $\models [\mathit{false}]$ **while** $(\mathbf{1})\ \{\}\,[\mathit{true}]$ hold; in fact, $\models \{\mathit{false}\}\,c\,\{Q\}$ and $\models [\mathit{false}]\,c\,[Q]$ hold for any c and Q, because an implication is true when the premisse is false ($\mathit{false} \Longrightarrow \phi$ is always true).

Note how it is important that the same assignment I is used to evaluate both the precondition P and the postcondition Q. This is what makes it possible to refer to variable values independent of the state. Consider the following triple

$$\models \{x = X\}\,x = x+1;\,\{X < x\}$$

If we spell out definition (5.2), we get

$$\begin{aligned}
&\models \{x = X\}\,x = x+1;\,\{X < x\}\\
&\iff \forall I.\,\forall \sigma.\,\sigma \models^I [\![x = X]\!]^{\Gamma,I}_{\mathscr{B}_v} \wedge \exists \sigma'.\,(\sigma,\sigma') \in [\![x = x+1]\!]_{\mathscr{C}} \Longrightarrow \sigma' \models^I [\![X < x]\!]^{\Gamma,I}_{\mathscr{B}_v}\\
&\iff \forall I.\,\forall \sigma.\,\sigma(x) = I(X) \wedge \exists \sigma'.\,\sigma' = \sigma[x \mapsto \sigma(x)+1] \Longrightarrow I(X) < \sigma'(x)\\
&\iff \forall I.\,\forall \sigma.\,\sigma(x) = I(X) \Longrightarrow I(X) < \sigma(x)+1\\
&\iff \forall I.\,I(X) < I(X)+1
\end{aligned}$$

We will in the following concentrate on partial correctness. As total correctness is partial correctness plus termination, proving partial correctness is a prerequisite for total correcness anyway. To show total correctness, this additionally needs two things:

1. *program safety* — the program should never run into error conditions, where the execution is undefined. In our current little language, the only error condition is division by zero, but later this will also include array access out of bounds, and illegal pointer dereferencing.
2. *termination* of while-loops and recursive functions.

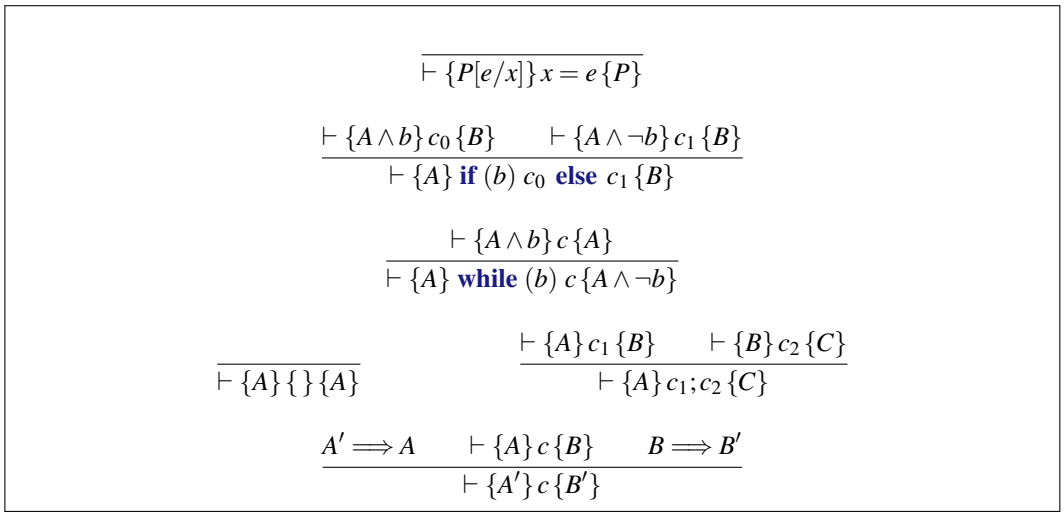

Figure 5.3: The rules of the Floyd-Hoare calculus

In practice, it is total correctness we want — proving with much effort that "my program would have given the correct result if it had not crashed" seems a bit weak, in particular when the program in question was supposed to control an autonomous car driving on the autobahn at top speed. You almost always want "my program always returns the correct result". If we consider partial correctness in the following, it is because there is a clear, nice separation of concerns: we can prove total correctness by proving partial correctness, plus termination and program safety; the proof of these can be done almost entirely separately.

5.3 The Rules of the Floyd-Hoare Calculus

Reconsidering the proof (**??**)–(**??**), it is clear that this is not the way to show that a Hoare triple holds (*i.e.* a program is correct). What is needed are some *syntactic rules* to show that a Hoare triple holds. In logic, such a set of rules is called a *calculus*.

The rules of the Floyd-Hoare calculus are given in Figure 5.3. There is one rule for each construct of the language: assignment, case distinction, iteration, sequencing, and the empty statement.

The assignment rules uses the notation $P[t/x]$ to denote the substitution of the variable x with the term t in an assertion P, the formal definition of which we elide here.[2]It is perfectly natural (but wrong) to think that the assignment rule should have the substitution in the postcondition. Yet it has to be in the precondition: if a predicate P has to hold in a state σ after assigning e to variable x, then it will yield the expression e when reading x, so it has to hold whenever we replace x with e. By this rule, assignment in the programming language gets translated into substitution in logical propositions. The changes in the state by assigning values to variables are reflected by substituting the corresponding values in the assertions over that state. In other words, program execution is translated to logical manipulation of a formula.

The rule for iteration has an assertion, A, for which we have to show that it is preserved by the body of

[2]The definition is completely straightforward in the absence of variable binders (such as the universal quantification, $\forall x.\, P$ which binds the variable x in P), and suprisingly complex in their presence; in our case, we bind only logical variables, but substitute only program variables, so the definition is easy.

the loop — if so, if it holds before the loop is entered, then it will hold after the loop has exited. This assertion is called the *invariant* of the loop. The invariant cannot be deduced from the program, it has to be given. Finding the invariant is one of the difficult parts of conducting correctness proofs with the Floyd-Hoare calculus; we will return to that problem later.

A special rule is the weaking rule, the bottom one: it brings logic into the proof, as opposed to the structural rules. To see why it holds, define the extension of an assertion P as the set of all states σ where P holds (for a fixed but arbitrary interpretation I). If P logically implies Q, then the extension of P is a subset of the extension of Q, or

$$P \Longrightarrow Q \text{ iff } \forall I.\forall \sigma.\sigma \models^I P \Longrightarrow \sigma \models^I Q$$

In fact, this can be taken as a semantic definition of logic implication for assertions. Thus, if $P \Longrightarrow Q$, we can replace P in the postcondition with Q (because if the program ends in a state σ where P holds, Q will also hold), and similary, Q in the precondition with P.

A special case of this is logical equivalence: we can always replace a pre- or postcondition with one which is logical equivalent. This allows equational reasoning in the assertions.

5.3.1 A Notation for Proofs

Writing down proofs in the calculus in the style of inference trees would be very tedious indeed. Assume we have the following schematic program c:

```
x= e;
while (x< n) {
  z= a;
  }
```

and we want to prove that it satisfies the Hoare triple $\vdash \{P\}\, c\, \{Q\}$ (for P, Q some schematic assertions). The inference tree of a typical proof could like this, where P_3 is the invariant of the while loop, and there are a few weakenings in between:

$$\dfrac{\dfrac{P \Longrightarrow P_1 \quad \vdash \{P_1\}\, x = e\, \{P_2\}}{\vdash \{P\}\, x = e\, \{P_2\}} \qquad \dfrac{P_2 \Longrightarrow P_3 \quad \dfrac{\dfrac{P_3 \Longrightarrow P_4 \quad \vdash \{P_4\}\, z = a\, \{P_3\}}{\vdash \{P_3 \wedge x < n\}\, z = a\, \{P_3\}}}{\vdash \{P_3\}\ \textbf{while}\ (x < n)\ldots \{P_3 \wedge x < n\}} \quad P_3 \wedge \neg(x < n) \Longrightarrow Q}{\vdash \{P_2\}\ \textbf{while}\ (x < n)\ldots \{Q\}}}{\vdash \{P\}\, c\, \{Q\}}$$

This will quickly become unreadable for even the most basic proofs. Instead, we use the following *linear* notation for that proof:

```
// {P}
// {P1}
x= e;
// {P2}
// {P3}
while (x< n) {
    // {P3 ∧ x < n}
    // {P4}
    z= a;
    // {P3}
```

```
    }
// {P_3 ∧ ¬(x < n)}
// {Q}
```

Our linear notation uses the following conventions:

- Assertions are annotated as comments into the program.
- For a statement c, the assertion P immediately preceding c and Q immediately following c form a Hoare triple $\vdash \{P\}\, c\, \{Q\}$, and must be derivable using the rules of the calculus from Figure 5.3. Specifically, this means:
  - For assignment:

    ```
    // {P[e/x]}
    x= e;
    // {P}
    ```

  - For the case distinction:

    ```
    // {P}
    if (b) {
      // {b ∧ P}
      ...
      // {Q}
      } else {
      // {¬b ∧ P}
      ...
      // {Q}
      }
    // {Q}
    ```

  - For the while-loop:

    ```
    // {P}
    while (b) {
      // {b ∧ P}
      ...
      // {P}
      }
    // {¬b ∧ P}
    ```

    For the latter two, the ellipses (...) are filled in with the branches of the conditional or the body of the while-loop respectively, annotated with assertions as required.
- The sequencing rule is used implicitly; for two statments $c_1; c_2$, and an assertion P immediately preceding c_1, and an assertion Q immediately following c_2, there must be an assertion R between c_1 and c_2, and we derive the Hoare triple $\vdash \{P\}\, c_1; c_2\, \{Q\}$ using this intermediate assertion R.
- The weakening rule is used implicitly: whenever there is an assertion followed immediately by another assertion, this means the weakening rule is applied.

Typically, at the end and start of the body of a while-loop and the two branches there will be some weakenings to bring the annotations into a form such that they match the rules.

```
1  // {x = X ∧ y = Y}
2  // {y = Y ∧ x = X}
3  z= x;
4  // {y = Y ∧ z = X}
5  x= y;
6  // {x = Y ∧ z = X}
7  y= z;
8  // {x = Y ∧ y = X}
```

Figure 5.4: A small example of a correctness proof in the linear notation.

```
// {true}
if (x < y) {
  // {x < y ∧ true}
  // {x ≤ y}
  // {true ∧ x ≤ y ∧ (true ∨ x = y)}
  // {x ≤ x ∧ x ≤ y ∧ (x = x ∨ x = y)}
  z = x;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
} else {
  // {¬(x < y) ∧ true}
  // {y ≤ x ∧ true ∧ (y = x ∨ true}
  // {y ≤ x ∧ y ≤ y ∧ (y = x ∨ y = y)}
  z= y;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
}
// {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
```

Figure 5.5: Another example for a correctness proof.

Programs annotated with linear correctness proofs must have a specific form: it starts with an annotation, followed by sequence of annotations and statements such that there is always at least one annotation between statements (no statements follow each other directly).

Figure 5.4 shows another proof in the linear notation. Note the difference between the program variables x and y, and the logical variables X, Y. We use two conventions:

- logical variable identifiers start with capital letters;
- a logical variable which has the capitalized name of a program variable usually serves to refer to the value of the program variable at an earlier point. Here, X and Y refer to the value of x and y before the statement is executed.

Another example using a case distinction is shown in Figure 5.5. The program calculates the minimum of two variables x and y. (Note the specification: it is not enough to specify that the minimum is smaller than $z < x$ and $z < y$, otherwise $-(x+y)$ would always be the minimum; we have to require the minimum is either x or y.) Figure 5.5 shows how to use weakening to massage the assertions into the form required by the if-statement; we use logical equivalences such as $(y = y) = \mathit{true}$, $\mathit{true} \vee P = \mathit{true}$, and $\mathit{true} \wedge P = P$.

```
// {0 ≤ n}
// {1 = 0! ∧ 0 ≤ n}
// {1 = (1−1)! ∧ 1 ≤ 1 ∧ 1−1 ≤ n}
p= 1;
// {p = (1−1)! ∧ 1 ≤ 1 ∧ 1−1 ≤ n}
c= 1;
// {p = (c−1)! ∧ 1 ≤ c ∧ c−1 ≤ n}
while (c<= n) {
  // {p = (c−1)! ∧ 1 ≤ c ∧ c−1 ≤ n ∧ c ≤ n}
  // {p∗c = (c−1)! ∗c ∧ 1 ≤ c ∧ c ≤ n}
  // {p∗c = c! ∧ 1 ≤ c ∧ c ≤ n}
  // {p∗c = ((c+1)−1)! ∧ 1 ≤ c+1 ∧ (c+1)−1 ≤ n}
  p= p∗c;
  // {p = ((c+1)−1)! ∧ 1 ≤ c+1 ∧ (c+1)−1 ≤ n}
  c= c+1;
  // {p = (c−1)! ∧ 1 ≤ c ∧ c−1 ≤ n}
  }
// {p = (c−1)! ∧ 1 ≤ c ∧ c−1 ≤ n ∧ ¬(c ≤ n)}
// {p = (c−1)! ∧ 1 ≤ c ∧ c−1 ≤ n ∧ c > n}
// {p = (c−1)! ∧ 1 ≤ c ∧ c−1 = n}
// {p = n!}
```

Figure 5.6: The factorial example.

The basic principle should be clear by now. We have not seen a full example involving a while-loop, and that is because it involves giving an invariant, so we turn to that problem now.

5.4 Finding Invariants

Consider the motivating example from Figure 5.1 on page 33 again. First, the specification is quite clear: after the program has run, p should be the factorial of n, *i.e.* $p = n!$ (no logical variables needed). Second, the precondition is simply that n should be larger or equal than zero (the factorial of negative integers is not defined).

When we try to construct a proof we run straight into a problem: the last statement is a while-loop, so to apply the while-rule we need an *invariant*. How do we find that?

Looking at the factorial example, the invariant can be constructed systematically as follows:

- The core of the invariant is $p = (c-1)!$. It describes that at each point in the loop we have computed the factorial up to $c-1$. Let us start with this:

$$p = (c-1)! \tag{5.4}$$

- Now, from the invariant and the negated loop condition we need to be able to derive the postcondition, because the while-loop is the final statement. So, here, from $\neg(c \leq n)$ and $p = (c-1)!$ we must be able to conclude that $p = n!$. Clearly, this means that $n = c-1$, but from $\neg(c \leq n)$ we can only conclude $c > n$. Something is lacking — we need a stronger invariant.

Essentially, because the loop body increment c only by 1, once the loop has finished, c must be $n+1$, rather than suddenly something much larger than n. In other words, the loop counter c does not jump, so $c-1$ is always smaller or equal than n. That makes our invariant

$$p = (c-1)! \wedge c - 1 \leq n \tag{5.5}$$

- We now try to show that the loop body preserves (5.5), by applying the assignment rule backwards over the two assignment statements. We get the transformed invariant

  $$p \cdot c = ((c+1)-1)! \wedge (c+1) - 1 \leq n$$

  which we can simplify trivially[3] to

  $$p \cdot c = c! \wedge c \leq n$$

  The second part of that conjunction is the loop condition (and that should not be surprising), but we need to be able to show that $p = (c-1)!$ implies $p \cdot c = c!$. Consider the recursive definition of the factorial function:

  $$n! = \begin{cases} 1 & n = 0 \\ (n-1)! \cdot n & n > 0 \end{cases} \tag{5.6}$$

  If we knew that $c > 0$ we could use that to conclude $c! = (c-1)! \cdot c$, hence $p = (c-1)! \Longrightarrow p \cdot c = (c-1)! \cdot c$. But is c larger than zero? Well, we start with c set to 1 and only increase c — so to be able to use this fact we need to add $c > 0$ to the invariant and get

  $$p = (c-1)! \wedge c - 1 \leq n \wedge c > 0 \tag{5.7}$$

  Of course, we need to repeat the calculation now that (5.7) is preserved by the loop body, which means that we also have to show $c + 1 > 0$ follows from $c > 0$. This is trivial; it is exactly the fact that we only increase c.

Figure 5.6 shows the full proof of the factorial example. The proof uses some weakenings which are just rearrangements, some trivial simplifications such as $1-1$ becomes 0 or $0! = 1$ (the first case of (5.6); an easy fact of linear arithmetic that from $a - 1 \leq b$ and $b > a$ we can conclude $a - 1 = b$ (line 19 to 20), and as explained above the second case of (5.6) (line 9 to 10). Note that when there is no weakening, the assertions must literally match the rules; so for example, the assertion following the while-loop in line 18 must be the invariant conjoint with the negated loop condition, *i.e.* $p = (c-1)! \wedge 1 <= c \wedge c - 1 \leq n \wedge \neg(c \leq n)$; that $\neg(c \leq n)$ is equivalent to $c > n$ needs to be introduced via an explicit weakening.

Finding an invariant is an approximative process. One takes a good guess, from the basic design of the algorithm, and then tries to refine it until the proof goes through. Here, we had three steps:

(i) Find the actual invariant: what has been calculated "up to here"?

(ii) Refine the invariant, such that from the invariant and the negated loop condition you are able to conclude the postcondition of the loop.

(iii) Show the invariant is preserved by the loop body, and if needed add further clauses to the invariant needed by weakening proofs in between.

[3] A simplification is an equality $s = t$, used to replace s with t. Here, we use equations like $(x+1) - 1 = x$.

Another remark is the loop here is a typical "counting loop", very much like a for-loop. In fact, for-loops are transformed to while-loops of this kind; our factorial example could be written more idiomatically, using the obivous syntactic sugar `c++` and **for**, as

```
p= 1;
for (c= 1; c<= n; c++) {
  p= p* c;
  }
```

For counting loops of this kind, where a variable c counts up to n (loop condition $c \leq n$), and afterwards something involving n should hold, the first part will be that proposition with n replaced by $c-1$, and the second part of the invariant will be $c < n$ so we can conclude $c+1 = n$ after the loop. This is what happened here.

Note that this loop runs from $c = 1$ to $c = n$. A more idiomatic loop in C (and Java) runs from $c = 0$ to $n-1$, written as

```
for (c= 0; c< n; c++) {
  ...
}
```

This starts counting from 0,[4] and will be used in conjuction with array access later on.

5.5 More Examples

Figure 5.7 is a variation of the factorial where we count downwards using the variable n as a counter. This requires a different precondition, which "saves" the value of the program variable n in the logical variable N. The invariant is also slightly more complicated. It needs the generalisation of the factorial function, the product from a to b, written as $\prod_{i=a}^{b} i$ or as $prod(a,b)$ here, and defined recursively as

$$prod(a,b) = \begin{cases} 1 & a > b \\ a \cdot prod(a+1,b) & a \leq b \end{cases} \tag{5.8}$$

A candidate for the invariant could now be that at all times p is the product from N (the original value of n) to the current value of n plus one (because we reduce n by one after multiplying with it in line 13:

$$p = prod(n+1,N) \tag{5.9}$$

A quick first check if we can conclude $p = N!$ from (5.13) reveals that we need to deduce $n = 0$, because then $p = prod(1,N) = N!$. However, we only know that $\neg(0 < n)$, *i.e.* $0 \geq n$. Thus, we add $0 \leq n$ to the invariant. (This very much like we had to add $c \leq n$ to the invariant before.)

When we check if the invariant $p = prod(n+1,N) \wedge 0 \leq n$ is preserved by the body of the while loop, we find that we do have to deduce $p \cdot n = prod(n,N)$ from $p = prod(n+1,N)$. We could do so using (5.8)

$$p \cdot n = prod(n,N) \tag{5.10}$$
$$n \cdot p = n \cdot prod(n+1,N) \tag{5.11}$$
$$p = prod(n+1,N) \tag{5.12}$$

[4] Edsgar Dijkstra, a famous Dutch-American computer scientist, used to claim that computer scientists should start couting with 0, but this is a mathematically questionable practice.

```
1  // {n = N ∧ 0 ≤ n}
2  // {n = N ∧ 1 = prod(n+1,N) ∧ 0 ≤ n}
3  // {1 = prod(n+1,N) ∧ 0 ≤ n ∧ n = N}
4  p= 1;
5  // {p = prod(n+1,N) ∧ 0 ≤ n ∧ n ≤ N}
6  while (0 < n) {
7     // {p = prod(n+1,N) ∧ 0 ≤ n ∧ n ≤ N ∧ 0 < n}
8     // {n · p = n · prod(n+1,N) ∧ n ≤ N ∧ 0 < n}
9     // {p · n = prod(n,N) ∧ 0 < n ∧ n ≤ N ∧ 0 < n}
10    p= p*n;
11    // {p = prod(n,N) ∧ n ≤ N ∧ 0 < n}
12    // {p = prod((n−1)n+1,N) ∧ n−1 ≤ N ∧ 0 ≤ n−1}
13    n= n−1;
14    // {p = prod(n+1,N) ∧ n ≤ N ∧ 0 ≤ n}
15    }
16 // {p = prod(n+1,N) ∧ 0 ≤ n ∧ ¬(0 < n)}
17 // {p = prod(n+1,N) ∧ 0 ≤ n ∧ n ≤ 0}
18 // {p = N!}
```

Figure 5.7: A variation on the factorial example: counting downwards.

```
// {0 ≤ a}
// {a = b · 0 + a ∧ 0 ≤ a}
r= a;
// {a = b · 0 + r ∧ 0 ≤ r}
q= 0;
// {a = b · q + r ∧ 0 ≤ r}
while (b <= r) {
   // {a = b · q + r ∧ 0 ≤ r ∧ b ≤ r}
   // {a = b · q + b + r − b ∧ b ≤ r}
   // {a = b · (q+1) + (r−b) ∧ 0 ≤ r−b}
   r= r−b;
   // {a = b · (q+1) + r ∧ 0 ≤ r}
   q= q+1;
   // {a = b · q + r ∧ 0 ≤ r}
   }
// {a = b · q + r ∧ 0 ≤ r ∧ ¬(b ≤ r)}
// {a = b · q + r ∧ 0 ≤ r ∧ r < b}
```

Figure 5.8: Computing the integer quotient and remainder.

```
// {0 ≤ a}
// {1−1 ≤ a ∧ 1 = 2·0+1 ∧ 1 = 0^2+1}
t= 1;
// {1−t ≤ a ∧ t = 2·0+1 ∧ 1 = 0^2+t}
s= 1;
// {s−t ≤ a ∧ t = 2·0+1 ∧ s = 0^2+t}
i= 0;
// {s−t ≤ a ∧ t = 2·i+1 ∧ s = i^2+t}
while (s <= a) {
  // {s−t ≤ a ∧ t = 2·i+1 ∧ s = i^2+t ∧ s ≤ a}
  // {t = 2·i+1 ∧ s = i^2+t ∧ s ≤ a}
  // {s ≤ a ∧ t+2 = 2·i+3 ∧ s = i^2+2·i+1}
  // {s+(t+2)−(t+2) ≤ a ∧ t+2 = 2·(i+1)+1 ∧ s+(t+2) = (i+1)^2+(t+2)}
  t= t+ 2;
  // {s+t−t ≤ a ∧ t = 2·(i+1)+1 ∧ s+t = (i+1)^2+t}
  s= s+ t;
  // {s−t ≤ a ∧ t = 2·(i+1)+1 ∧ s = (i+1)^2+t}
  i= i+ 1;
  // {s−t ≤ a ∧ t = 2·i+1 ∧ s = i^2+t}
  }
// {s−t ≤ a ∧ t = 2·i+1 ∧ s = i^2+t ∧ ¬(s ≤ a)}
// {i^2 ≤ a ∧ a < (i+1)^2}
```

Figure 5.9: Computing the integer square root.

but this needs $n \leq N$ as a side condition, so we need to add that to the invariant as well. The full invariant now is

$$p = prod(n+1, N) \wedge 0 \leq n \wedge n \leq N \tag{5.13}$$

The fully annotated program is shown in Figure 5.7. There are number of logical derivations happening in the weakenings, *e.g.* in line 3 we conclude $n \leq N$ from $n = N$, or in line 8 we drop $0 \leq n$ from the invariant because we can recover $0 \leq n$ from $0 < n$ later: when going down in a weakening, we can always remove parts of a conjunction which are not needed later on, because $A \wedge B \Longrightarrow A$.

Figure 5.8 is a differen counting loop, where we count down in steps of a instead of up in steps of 1. Subsequently, the $s - t \leq q$ part of the invariant is the equivalent of $c - 1 \leq n$ we encountered early, stating that "the loop does not jump (in steps larger than a)"; hence, $s - t \leq a$ follows as the transformed loop condition. What is peculiar here is that the invariant is remarkably close to the postcondition.

Finally, Figure 5.9 computes the integer square root of a, which is the number i such that $i^2 \leq a < (i+1)^2$. From that, let $s = (i+1)^2 = i^2 + 2 \cdot i + 1$ and $t = 2 \cdot i + 1$, hence $s = i^2 + t$ and $s - t \leq a$. Coming up with the invariant here is something like a dark art.

5.6 Summary

In this section, we have introduced the central notions of the Floyd-Hoare logic:

- *Assertions* are state-based predicates (or, in other terms, boolean functions with program variables and logical variables), which we use to specify which properties hold in a specific state.
- A Floyd-Hoare triple $\{P\}\, c\, \{Q\}$ consists of a state assertion P, the precondition, a statement (program) c, and an assertion Q, the postcondition.
- A Floyd-Hoare triple is valid, written $\models \{P\}\, c\, \{Q\}$, if in every state where P holds, and for which c terminates, Q holds afterwards. This is notion of *partial correctness*. For total correctness, c must terminate for every state in which P holds.
- A calculus of six rules (one for each construct of the programming language) allows us to derive judgements of the form $\vdash \{P\}\, c\, \{Q\}$. The rules suggest a backward-proof of correcntess. Most of the rules are straightforward, but the rule for the while loop needs an invariant finding which is not always easy.
- A linear notation makes proofs easier to write and read.
- We have seen how to find invariants, which is the hard part of proofs in the Floyd-Hoare-style.

One question which is open is how a judgement $\vdash \{P\}\, c\, \{Q\}$ relates to the semantic definition $\models \{P\}\, c\, \{Q\}$; ideally, we would hope that if we can derive the former the latter holds as well. This is the soundness or correctness of the Floyd-Hoare calculus, and we will adress it in the next section.

```
// {0 ≤ y}
x= 1;
c= 1;
while (c <= y) {
 x= 2*x;
 c= c+1;
}
// {x = 2^y}
```

```
// {0 ≤ y}
x= 1;
c= 0;
while (c < y) {
 c= c+1;
 x= 2*x;
}
// {x = 2^y}
```

```
// {y = Y ∧ 0 ≤ y}
x= 1;
while (y != 0) {
  x= 2*x;
  y= y−1;
}
// {x = 2^Y}
```

Figure 5.10: Computing powers of 2 in three variations.

Chapter 6

Correctness of the Floyd-Hoare Calculus

6.1 Syntactic Derivation and Semantic Validity

In Chapter 5, we have introduced the Floyd-Hoare logic and its proof calculus, with two notions:

- The semantic definition $\models \{P\}\,c\,\{Q\}$ of the validity of a Floyd-Hoare triple, which talks about program states, and
- the syntactic notion $\vdash \{P\}\,c\,\{Q\}$ of how we can derive that a Hoare triple holds by purely syntactic deriviation.

The question is, how are these related? In formal logic (and mathematics), this situation occurs quite often: one defines some notation, a semantics for it (what does it mean?) and syntactic rules to do calculations (a *calculus* for the logic). Then, we want to know if using the syntactic rules can we always get correct results, and can we get all results? In our situation, this means that the relationship $\vdash \{P\}\,c\,\{Q\} \quad \overset{?}{\rightsquigarrow} \quad \models \{P\}\,c\,\{Q\}$ has two directions:

- Does $\vdash \{P\}\,c\,\{Q\}$ imply $\models \{P\}\,c\,\{Q\}$, meaning all Floyd-Hoare triples we derive are correct? This is the *soundness* or *correctness* of the Floyd-Hoare calculus.
- Does $\models \{P\}\,c\,\{Q\}$ impy $\vdash \{P\}\,c\,\{Q\}$, meaning when a Floyd-Hoare holds we will be able to derive it? This is the *completeness* of the Floyd-Hoare calculus.

6.2 Soundness

We turn towards soundness first. Without further ado, let us state and prove that the calculus is sound. (Everything else would make the calculus useless. Who wants to extend effort into proving something that may or may not be true?)

Theorem 6.1 (Soundness of Floyd-Hoare logic) *The calculus for the Floyd-Hoare logic is sound:*

$$\vdash \{P\}\, c\, \{Q\} \Longrightarrow \models \{P\}\, c\, \{Q\}$$

Proof. The statement is proven by *rule induction*. Since the judgement $\vdash \{P\}\, c\, \{Q\}$ is derived by using the rules of the calculus, it is sufficient to prove that for each rule if it concludes $\vdash \{P\}\, c\, \{Q\}$ we can show $\models \{P\}\, c\, \{Q\}$, where we can assume this for the rules premisses.

This means we have six cases to consider. TO DO.

Six cases missing.

□

The proof needs the following lemma. It may look innocuous at first sight, but it formalises one the of fundamental insights of the Floyd-Hoare-Calculus, namely in the logic syntactic substitution models assignment (which is modelled in the denotational semantics by functional update $\sigma[- \mapsto -]$ of the state σ).

Lemma 6.2 (Substitution Lemma) *For any state $\sigma \in \Sigma$, assignment I, assertion $B \in$ **Assn**, arithmetic expression $e \in$ **Aexp** and variable $x \in$ **Loc**, we have*

$$\sigma \models^I B[e/x] \iff \sigma[x \mapsto [\![e]\!]_{\mathscr{A}}(\sigma)] \models^I B \tag{6.1}$$

Proof. By induction on the structure of B. (Don't do this as an exercise, it is boring.) This first needs a preliminary lemma like the above for extended arithmetic expressions $a \in$ **Aexpv**:

$$[\![a[e/x]]\!]^I_{\mathscr{A}_V}(\sigma) = [\![a]\!]^I_{\mathscr{A}_V}(\sigma[x \mapsto [\![e]\!]^I_{\mathscr{A}_V}(\sigma)]) \tag{6.2}$$

which is proven by structural induction on a. □

6.3 Conclusion

We have seen how to conduct basic proofs in the Floyd-Hoare calculus, and we have shown important "meta-properties" of the calculus, in particular its soundness.

But one gets soon a bit tired about writing programs which handle integers only (unless you life a very boring life, or just happen to like integers a lot), so in the next chapter we will turn towards richer data types.

Chapter 7

Structured Datatypes

Structured datatypes are types such as arrays, structures (also known as labelled records), and of course reference types (pointers). Pointers open a whole Pandora's box of their own, so we will defer looking into them until later, but we will now turn towards arrays and structures. It turns out they are quite easy to model, on an abstract level.

7.1 Datatypes

7.1.1 Arrays

At an abstract level, arrays are finite maps from an initial sequence $[0..n]$ of the naturals to a value type. They values can in turn be arrays, making the array multi-dimensional. In C0, arrays are declared like this:

```
int a[5];
int c[3][2];
```

Arrays always have a fixed, known length in C0. The second line above defines an array of length 3, which has arrays of length 2 as elements.

7.1.2 Chars and Strings

The datatype **char** is the type of basic, unsigned characters. It is a subset of **int**, meaning each element of **char** can be converted into an **int** (but not the other way around).

Strings are then just array of type **char**. The following two declarations with initialisations are equivalent:

```
char c[5] = "hello";
char c[5] = {'h', 'e', 'l', 'l', 'o', '\\0'};
```

Strings are supported fairly rudimentarily in C (and C0). They can be initialised, but not assigned, and all functions to handle strings are from the standard library, not from the language itself.

The types **char** and **int** are called *elementary types*.

Remark: C knows more elementary types, such as the floating point types, and integers of varying word length (**short**, **long** *etc.*), of both signed and unsigned variety[1]. There is a number of rather complicated rules describing the automatic conversions between them. All of this is semantically rather uninteresting, so we disregard it here in favour of just two elementary types.

7.2 Extending C0

7.2.1 Syntax

To be able to refer to structured datatypes, we need a syntax to express values of this type. This means that was an identifier before can now be a structured value, *e.g.* denoting an array access or record selection. Array access and record selection are not ordinary operators like addition or subtraction, because they can appear on the left-hand side of an assignment as well. Such expressions, which semantically denote somewhere where we can store a value, are called l-expressions[2] (with l for left-hand side). We introduce a syntactic class **Lexp** for l-expressions, and extend the abstract syntax for C0 from page 10 as follows:

Lexp	$l ::= \mathbf{Idt} \mid l[a] \mid l.\mathbf{Idt}$
Aexp	$a ::= \mathbf{Z} \mid \mathbf{C} \mid \mathbf{Lexp} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2$
Bexp	$b ::= \mathbf{1} \mid \mathbf{0} \mid a_1 == a_2 \mid a_1 < a_2 \mid !\, b \mid b_1 \,\&\&\, b_2 \mid b_1 \,\|\|\, b_2$
Exp	$e := \mathbf{Aexp} \mid \mathbf{Bexp}$

We have also introduced a new syntactic class **C** for characters, allowing us to write down strings as above. A **C** is a literal character written as `'x'` (no Unicode nonsense, but basic escape sequences like `'\n'` or `'\0'`).

7.2.2 The State

Before we consider the semantics, we need to extend our notion of state as well. We have given a state model in Section 2.3 before: it mapped *locations* to *values* , with the locations being identifiers and values being integers (Definition 2.2), Now that we have structured addresses, locations need to be more than just identifiers. The C language has a fairly low-level memory model, with addresses being counted in bytes; for the time being, we can be more abstract and talk about locations which are either directly an identifier (as before), or an array access, or a record selection:

Definition 7.1 (Locations, Values and System State (revisited))

The values *are given by integers,* $\boldsymbol{V} \stackrel{def}{=} \mathbb{Z}$

The locations *are as follows:* $\boldsymbol{Loc} ::= \boldsymbol{Idt} \mid \boldsymbol{Loc}[\mathbb{Z}] \mid \boldsymbol{Loc}.\boldsymbol{Idt}$

The system state *is a partial map from locations to values:* $\Sigma \stackrel{def}{=} \boldsymbol{Loc} \rightharpoonup \boldsymbol{V}$.

Note the difference between **Lexp** and **Loc**: the former is syntactic entity which has arbitrary arithmetic expressions for array access, the latter is a semantic entity which has concrete integers as array access.

[1] Unsigned integers are in fact natural numbers, bu they are not called that.

[2] In the C literature they are usually referred to as l-values, but as they are *syntactic* entities not semantic ones as the designation "values" might connotate, we do not follow this custom here; in our usage, an l-value is the semantic denotation of an l-expression.

7.2.3 Operational Semantics

To extend the operational and denotational semantics, we need to two steps:

1. first, we need to give a meaning to l-expressions in terms of locations, and

2. second, we need a meaning for an arithmetic expression which is an l-expression (this replaces the rule where an expression just comprises a single identifier); here, the semantics is given by accessing the (ambient) state at the location denoted by the l-expression.

For the operational semantics, the rules in Figure 7.1 define the evaluation of locations and expressions. Under a given state σ, an lvalue m evaluates to a location l or an error $\bot$, written as

$$\langle m, \sigma \rangle \to_{Lexp} l \mid \bot$$

The rule for expressions extend the rules given in Section 2.4, meaning they should be added to those in Figures 2.1 and 2.2, with the obvious exception that the rule evaluating an identifier as an expression is replaced by the rule evaluating an lvalue as an expression. Similarly, the rule evaluate an assignment extends and replaces the rule in Figure 2.3.

7.2.4 Denotational Semantics

The denotational semantics follows the operational semantics fairly closel. We need to give a meaning to l-expressions by locations by defining a function

$$[\![l]\!]_{\mathscr{L}} : \mathbf{Lexp} \to (\Sigma \rightharpoonup \mathbf{Loc})$$

Figure 7.2 gives the definition of this function. Because the state is now a partial map $\mathbf{Loc} \rightharpoonup \mathbf{V}$, the rule to give meaning to an lvalue as an arithmetic expression needs to change slightly (7.1). The rule to give meaning to assignments also needs slight change (7.2).

7.2.5 Types and Undefinedness

If an l-expression denotes a location which is not in the domain of the state, using this l-expression on the left-hand side of an assignment, or in an expression, renders the assignment or expression undefined. (Before, this could only happen if we use an undefined identifier, and that could be detected syntactically; now, we have errors such as accessing an array out of bounds.)

Note the l-expression *itself* may still be a defined location, only once we try to access the state at this location the semantics become undefined. Also, the semantics is fairly liberal— for example, we can write into an array at *any* location, there are no checks for array bounds on writing.

Also, assignment usually is restricted to assigning values of elementary types (in our case, integers). We do not follow this here, because we want to keep our language typeless — not so much because types are not needed, but because types are a bit of a Pandora's box to open. You start with integers and arrays, and before you know it you have dependent types and impredicative type universes.

$$\frac{x \in \mathbf{Idt}}{\langle x, \sigma \rangle \to_{Lexp} x}$$

$$\frac{\langle m, \sigma \rangle \to_{Lexp} l \qquad \langle a, \sigma \rangle \to_{Aexp} i \neq \bot}{\langle m[a], \sigma \rangle \to_{Lexp} l[i]} \qquad \frac{\langle m, \sigma \rangle \to_{Lexp} l \qquad \langle a, \sigma \rangle \to_{Aexp} \bot}{\langle m[a], \sigma \rangle \to_{Lexp} \bot}$$

$$\frac{\langle m, \sigma \rangle \to_{Lexp} l}{\langle m.i, \sigma \rangle \to_{Lexp} l.i}$$

$$\frac{\langle m, \sigma \rangle \to_{Lexp} l \qquad l \in Dom(\sigma)}{\langle m, \sigma \rangle \to_{Aexp} \sigma(l)} \qquad \frac{\langle m, \sigma \rangle \to_{Lexp} l \qquad l \notin Dom(\sigma)}{\langle , \sigma \rangle \to_{Aexp} \bot}$$

$$\frac{\langle m, \sigma \rangle \to_{Lexp} l \qquad \langle e, \sigma \rangle \to v}{\langle m = e, \sigma \rangle \to_{Stmt} \sigma[l \mapsto v]}$$

Figure 7.1: Rules to evaluate l-expressions and expressions

$$[\![l]\!]_{\mathscr{L}} : \mathbf{Lexp} \to (\Sigma \rightharpoonup \mathbf{Loc})$$

$$\begin{aligned}
[\![x]\!]_{\mathscr{L}} &= \{(\sigma, x) \mid \sigma \in \Sigma\} \\
[\![m[a]]\!]_{\mathscr{L}} &= \{(\sigma, l[i]) \mid (\sigma, l) \in [\![m]\!]_{\mathscr{L}}, (\sigma, i) \in [\![a]\!]_{\mathscr{A}}\} \\
[\![m.i]\!]_{\mathscr{L}} &= \{(\sigma, m.i) \mid (\sigma, l) \in [\![m]\!]_{\mathscr{L}}\} \\
[\![m]\!]_{\mathscr{A}} &= \{(\sigma, \sigma(l)) \mid \sigma \in \Sigma, (\sigma, l) \in [\![m]\!]_{\mathscr{L}}, l \in Dom(\sigma)\} \qquad (7.1) \\
[\![m = e]\!]_{\mathscr{C}} &= \{(\sigma, \sigma[l \mapsto v]) \mid (\sigma, l) \in [\![m]\!]_{\mathscr{L}}, (\sigma, v) \in [\![e]\!]_{\mathscr{A}}\} \qquad (7.2)
\end{aligned}$$

Figure 7.2: Denotational semantics for lvalues

7.2.6 Floyd-Hoare Calculus

The rules of the Floyd-Hoare calculus, perphaps surprisingly, do not need to change — but they need to be read differently. Specifically, the assignment rule which reads

$$\overline{\vdash \{P[e/m]\}\, m = e\, \{P\}}$$

Note how the precondition of the rule contains a substitution. Previously, this was a simple syntactic substitution, replacing all occurences of a variable (called say "x") by an expression. Now, the substitution becomes a *rewrite*— we need to replace all occurences of the lvalue expression m with the expression e. This may sound innocuous but the problem is that lvalues may contain an array access, which contains an (arbitrary) integer expression.

```
1  // {n ≤ 0}
2  // {(∀j.0 ≤ j < 0 ⟶ a[j] = j) ∧ 0 ≤ n}
3  i= 0;
4  // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n}
5  while (i< n) {
6    // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n ∧ i < n}
7    // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i = i ∧ i+1 ≤ n}
8    // {((∀j.0 ≤ j < i ⟶ a[j] = j) ∧ a[i] = i ∧ i+1 ≤ n)[a[i]/i]}
9    a[ i]= i ;
10   // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ a[i] = i ∧ i+1 ≤ n}
11   // {(∀j.0 ≤ j < i+1 ⟶ a[j] = j) ∧ i+1 ≤ n}
12   // {((∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n)[i+1/i]}
13   i= i+1;
14   // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n}
15   }
16 // {∀j.0 ≤ j < n ⟶ a[j] = j}
```

Figure 7.3: Initialising an array

Consider the simple example in Figure 7.3. We have spelt out the substitutions in that example. In line 10 there is a substitution of a simple variable i with the expression $i+1$, which easily computes as follows:

$$\begin{aligned}
&((\forall j.0 \le j < i \longrightarrow a[j] = j) \wedge i \le n)[i+1/i] \\
=\ &((\forall j.0 \le j < i \longrightarrow a[j] = j)[i+1/i] \wedge (i \le n)[i+1/i] \\
=\ &(\forall j.0 \le j < i \longrightarrow a[j] = j) \wedge i+1 \le n
\end{aligned}$$

In line 8 there is a more involved substitution:

$$\begin{aligned}
&((\forall j.0 \le j < i \longrightarrow a[j] = j) \wedge a[i] = i \wedge i+1 \le n)[i/a[i]] \\
=\ &(\forall j.0 \le j < i \longrightarrow a[j] = j)[i/a[i]] \wedge (a[i] = i)[i/a[i]] \wedge (i+1 \le n)[i/a[i]] \\
=\ &(\forall j.(0 \le j < i)[i/a[i]] \longrightarrow (a[j] = j)[i/a[i]]) \wedge i = i \wedge i+1 \le n && (7.3) \\
=\ &(\forall j.0 \le j < i \longrightarrow a[j] = j) \longrightarrow i = i \wedge i+1 \le n && (7.4)
\end{aligned}$$

Here, we can see two things: in (7.3), we substitute $a[i]$ with i, because $a[i]$ is (syntactically) equal to $a[i]$. In contrast, in (7.4), we do *not* substitute $a[j]$ with i because we know that $a[j]$ is not equal to $a[i]$, because the precondition states that $j < i$.

Note that this only works because we have weakened the substituted invariant $\forall j.0 \leq j < i+1 \longrightarrow a[j] = j$ in line 11 to $\forall j.0 \leq j < i \longrightarrow a[j] = j$ in line 10. If we had not performed this weakening, we would have a substitution like this:

$$(\forall j.0 \leq j \leq i \longrightarrow a[j] = j)[i/a[i]] = (\forall j.0 \leq j \leq i \longrightarrow a[j][i/a[i]] = j)$$

Here, we know that j is never equal to $a[i]$, so $j[i/a[i]] = j$, but we cannot reduce $a[j][i/a[i]]$ further. If we encounter a situtation like that, it is either a problem of a missing weakening such as above, or a problem of the specification. We will come back to this problem later, when we discuss forward and backward verification condition generation.

7.3 Example: Finding the maximum element in an array

A generally useful pattern is a theorem which allows us to extend a range:

$$(\forall j.0 \leq j < n \longrightarrow P(j)) \wedge P(n) \Longleftrightarrow \forall j.0 \leq j < n+1 \longrightarrow P(j) \tag{7.5}$$

If we know $P(j)$ holds for j from 0 up to less than n, and it holds for n itself, then it will hold from 0 up to less than $n+1$. This theorem is used in all proofs where a program iterates through an array; in fact, we have used this theorem going from line 10 to line 11 in Figure 7.3 above.

As a slightly more non-trivial example, we now want to verify a program which finds the maximum element in an array. For this, the array obviously has to be non-empty. The result should be the index r of the array which is a valid index, and for which the array elements at all other indices are at most equal. This can be specified as follows (with n the size of the array):

// $\{0 < n\}$
...
// $\{(\forall j.0 \leq j < n \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n\}$

A first implementation of this specification would be as follows

```
i= 0;
r= 0;
while (i< n) {
  if (a[r] < a[i]) {
    r= i;
    }
  else {
    }
  i= i+1;
  }
}
```

What could the invariant be? Obviously, r is the maximum up to index i. So let us try that:

$$(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]$$

As we have seen above, we first need to check we can prove the postcondition from the invariant and the negated loop condition

$$(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge \neg(i < n) \longrightarrow (\forall j.0 \leq j < n \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n$$

which is equivalent to showing

$$(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge n \leq i \longrightarrow (\forall j.0 \leq j < n \longrightarrow a[j] \leq a[r])$$
$$(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge n \leq i \longrightarrow 0 \leq r < n$$

We can straight away identify two shortcomings with the invariant: firstly, $n \leq i$ is not enough to derive $n = i$ (for the first implication), and secondly, there is no way to show $0 \leq r < n$ in the second implication. So we strengthen the invariant, requiring $i \leq n$ and $0 \leq r < n$. This clearly allows us to prove the postcondition, but is it preserved by the loop body? Try to informedly guess the answer before reading on.

As it turns out, it is not enough. The problematic clause is the positive branch of the case distinction. Consider this partial annotation:

```
1 // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ i ≤ n ∧ 0 ≤ r < n ∧ i < n}
2   if (a[r] < a[i]) {
3     // {(∀j.0 ≤ j < i+1 ⟶ a[j] ≤ a[i]) ∧ i+1 ≤ n ∧ 0 ≤ i < n}
4     r= i;
5     // {(∀j.0 ≤ j < i+1 ⟶ a[j] ≤ a[r]) ∧ i+1 ≤ n ∧ 0 ≤ r < n}
6     }
7   // {(∀j.0 ≤ j < i+1 ⟶ a[j] ≤ a[r]) ∧ i+1 ≤ n ∧ 0 ≤ r < n}
8   i= i+1;
9 // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ i ≤ n ∧ 0 ≤ r < n}
```

As we can see, we now have to show $(\forall j.0 \leq j < i+1 \longrightarrow a[j] \leq a[i]) \wedge i+1 \leq n \wedge 0 \leq i < n$ from the invariant which is equivalent to showing the following three from the invariant:

$$(\forall j.0 \leq j < i+1 \longrightarrow a[j] \leq a[i]$$
$$i+1 \leq n$$
$$0 \leq i < n$$

The first two are easy(ish), as we will demonstrate shortly, but the third, inocuous as it looks, actually can't be show — the invariant simply does not contain a lower limit on i:

$$(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge i \leq n \wedge 0 \leq r < n \wedge i < n$$

So we strengthen the invariant further and add $0 \leq i$. The fully annotated prgram can be found in Figure 7.4. We use (7.5) from line 20 to 21 and from line 13 to 14. From line 12 to 13, we use transitivity of less-equal to go from $a[j] \leq a[r]$ and $a[r] \leq a[i]$ to $a[j] \leq a[i]$.

7.4 Conclusions

We have extended our language, C0, to cover richer datatypes such as labelled records (**struct** in C) or arrays. On the syntactic side, this requires that on the left-hand side of an assignment there can be more than just identifiers, namely structured expressions (lvalues, **Lexp**). On the semantic side, this requires that the locations of our state are structured as well, necessitating a refinement of our notion of state such that the addresses can also be composed using labels (*e.g.* `a.x`) or array indices. This is an abstraction over a more low-level memory model, where all addresses are integers.

```
1  // {0 < n}
2  // {(∀j.0 ≤ j < 0 ⟶ a[j] ≤ a[0]) ∧ 0 ≤ 0 ≤ n ∧ 0 ≤ 0 < n}
3  i= 0;
4  // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[0]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ 0 < n}
5  r= 0;
6  // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
7  while (i< n) {
8    // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ i < n}
9    // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n}
10   if (a[r] < a[i]) {
11     // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n ∧ a[r] < a[i]}
12     // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ a[r] < a[i]) ∧ 0 ≤ i < n}
13     // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i < n}
14     // {(∀j.0 ≤ j < i+1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i+1 ≤ n ∧ 0 ≤ i < n}
15     r= i;
16     // {(∀j.0 ≤ j < i+1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i+1 ≤ n ∧ 0 ≤ r < n}
17     }
18   else {
19     // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n ∧ a[i] ≤ a[r]}
20     // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ a[i] ≤ a[r] ∧ 0 ≤ i < n ∧ 0 ≤ r < n}
21     // {(∀j.0 ≤ j < i+1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i+1 ≤ n ∧ 0 ≤ r < n}
22     }
23   // {(∀j.0 ≤ j < i+1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i+1 ≤ n ∧ 0 ≤ r < n}
24   i= i+1;
25   // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
26   }
27 // {(∀j.0 ≤ i < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ i ≥ n}
28 // {(∀j.0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Figure 7.4: Finding the maximal element in an non-empty array

The syntactic and semantic changes come together with both an operational and denotational semantics for lvalues. The changes to the Floyd-Hoare calculus remain minimal; in fact, it is just the assignment rule which needs to change, but in a round-about fashion: suddenly, substutition becomes more complicated as we can now not only substitute expressions for identifiers, but expressions for lvalues.

We have considered some small examples, and seen that the specifications grow rather large and convoluted. What we need now is threefold: first, some help with writing down correctness proofs in our calculus, second something to help us to conduct proofs, and third, something to state specifications more concisely. Most of the rule applications can be derived, so can we not automate this process?

Chapter 8

Verification Condition Generation

Consider the following simple program fragment (taken from the integer square root program on page 45):

```
1    t= t+ 2;
2    s= s+ t;
3    i= i+ 1;
```

When verifying this, we start with the postcondition holding after the last statement (which is the invariant)

$$\{s-t \leq a \wedge t = 2 \cdot i + 1 \wedge s = i^2 + t\}$$

and repeatedly apply the assignment rule to obtain new assertions as follows:

Before line 3 : $\{s-t \leq a \wedge t = 2 \cdot (i+1) + 1 \wedge s = (i+1)^2 + t\}$

Before line 2 : $\{s+t-t \leq a \wedge t = 2 \cdot (i+1) + 1 \wedge s+t = (i+1)^2 + t\}$

Before line 1 : $\{s+(t+2)-(t+2) \leq a \wedge t+2 = 2 \cdot (i+1) + 1 \wedge s+(t+2) = (i+1)^2 + (t+2)\}$

As we can see, all we need is the postcondition, and we can derive a precondition from that. This works, because the assignment rule (as given in Figure 5.3 on page 37) has an "open" postondition, *i.e.* the schematic rule is

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

and the postcondition P of the rule matches on *any* given postcondition. The same goes for the sequence and empty rules:

$$\frac{}{\vdash \{A\}\,\{\}\,\{A\}} \qquad \frac{\vdash \{A\}\, c_1\, \{B\} \quad \vdash \{B\}\, c_2\, \{C\}}{\vdash \{A\}\, c_1; c_2\, \{C\}}$$

and hence this works for any sequence of assignment statements— which what we have done above. So, can we use this to automate proofs of program correctness?[1]

[1] Another question is if this could work going forwards as well, but we postpone that question to the next chapter.

8.1 Reasoning Backwards

Looking at the other rules (in Figure 5.3, we can see the following:

- the if-rule has an open precondition, and thus looks suitable to be applied backwards as described (but see below),
- and the while-rule does not have an open precondition, it can only be applied backwards if the postcondition happens to be of the form $b \wedge I$, where b is the loop condition (and I the invariant), which is pretty unlikely.

So what to do? The general trick is that any rule can be given an open precondition by composing it with the weakening rule. For example, if we compose the while with two instances of the weakening rule, we get the following rule:

$$\dfrac{\dfrac{\dfrac{A \wedge b \Longrightarrow B \quad \vdash \{B\}\, c\, \{A\}}{\vdash \{A \wedge b\}\, c\, \{A\}}\ \textit{Weaken}}{\vdash \{A\}\ \textbf{while}\ (b)\ c\, \{A \wedge \neg b\}}\ \textit{While} \qquad A \wedge \neg b \Longrightarrow C}{\vdash \{A\}\ \textbf{while}\ (b)\ c\, \{C\}}\ \textit{Weaken}$$

which written as the single rule is

$$\dfrac{A \wedge b \Longrightarrow B \quad \vdash \{B\}\, c\, \{A\} \quad A \wedge \neg b \Longrightarrow C}{\vdash \{A\}\ \textbf{while}\ (b)\ c\, \{C\}} \tag{8.1}$$

This rule always matches any postcondition, but we need to prove the two implications in the precondition — these are our *verification conditions* — and we need to get the invariant A from somewhere (it is not given by the precondition). This is to be expected: finding the invariant is the creative (*i.e.* difficult) part of proofs in the Floyd-Hoare logic, and we could not reasonably expect any automation to alleviate us of this burden. Formally, we can not give a finite formula in the general case (a construction can be given using Gödel's β-predicate, see *e.g.* [8]).

Thus, to make correctness proofs automatically checkable, we need to somehow give the invariants for each while-loop manually. We do so by *annotating* the invariant to the while-loop. An annotation is a comment in a special format, which can be picked up by a tool checking the proof. On the other hand, the compiler completely ignores the comment and so it does not affect the semantics. The invariant annotation looks like this:

```
while (b) /** inv I; */ c
```

With this, we can write the while-rule as

$$\dfrac{I \wedge b \Longrightarrow B \quad \vdash \{B\}\, c\, \{I\} \quad I \wedge \neg b \Longrightarrow C}{\vdash \{I\}\ \textbf{while}\ (b)\ /{**}\ \textbf{inv}\ I\ {*}/\ c\, \{C\}} \tag{8.2}$$

The implications of rule (8.2) become the *verification conditions*. Verification conditions are state-free formula of first-order logic with induction, which represent the logical reasoning underlying the program correctness. They are proven by an external prover.

The astute reader may have noticed the weakening $I \wedge b \Longrightarrow B$ in the premisse of rule (8.2). What is this for? If we drop that weakening, we get the following rule:

$$\dfrac{\vdash \{I \wedge b\}\, c\, \{I\} \quad I \wedge \neg b \Longrightarrow C}{\vdash \{I\}\ \textbf{while}\ (b)\ /{**}\ \textbf{inv}\ I\ {*}/\ c\, \{C\}} \tag{8.3}$$

If we apply this rule, we have to show an implication — this is our verfication condition — and a we have to show that the precondition of the body c of the while-loop is $I \wedge b$. But if we calculate the precondition schematically, it could be anything! That means, it is not enough that the rules contain an open postcondition in the conclusion to be automatically applicable, the rules must also have open preconditions in the premisses.

This is the reason why the if-rule as given in Figure 5.3 is not entirely suitable for backwards reasoning:

$$\frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\ \{B\}}$$

To use this would require that the derived precondition of the positive branch (c_0) and the negative branch (c_1) are $A \wedge b$ and $A \wedge \neg b$, respectively; but it would be sheer luck if that were the case. Again, we need a rule where the precondition of both premisses is arbitrary and disjoint. By composing with weakening (again), we get such a rule. For this, we need a bit of boolean formula gymnastics. In general, given propositions A_0, A_1 and B, we have

$$(A_0 \wedge B) \vee (A_1 \wedge \neg B) \wedge B \iff (A_0 \wedge B \wedge B) \vee (A_1 \wedge \neg B \wedge B) \iff (A_0 \wedge B) \vee \mathit{false} \iff A_0 \wedge B \tag{8.4}$$

and similary, $(A_0 \wedge B) \vee (A_1 \wedge \neg B) \wedge \neg B \iff A_1 \wedge \neg B$. This suggests that if the two preconditions of the positive and negative branch are A_0 and A_1, respectively, to use $(A_0 \wedge B) \vee (A_1 \wedge \neg B)$ as the derived weakest precondition for backwards reasoning, because we get the derived rule (where the two equivalences for the weakening rules are given by (8.4)), and hence do not need to be proven:

$$\frac{\dfrac{\overline{(A_0 \wedge b) \vee (A_1 \wedge \neg b) \wedge b \iff A_0} \quad \vdash \{A_0\}\, c_0\, \{B\}}{\vdash \{(A_0 \wedge b) \vee (A_1 \wedge \neg b) \wedge b\}\, c_0\, \{B\}} \qquad \dfrac{\overline{(A_0 \wedge b) \vee (A_1 \wedge \neg b) \wedge \neg b \iff A_1} \quad \vdash \{A_1\}\, c_0\, \{B\}}{\vdash \{(A_0 \wedge b) \vee (A_1 \wedge \neg b) \wedge \neg b\}\, c_1\, \{B\}}}{\vdash \{(A_0 \wedge b) \vee (A_1 \wedge \neg b)\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\ \{B\}}$$

which as a single rule reads

$$\frac{\vdash \{A_0\}\, c_0\, \{B\} \quad \vdash \{A_1\}\, c_0\, \{B\}}{\vdash \{(A_0 \wedge b) \vee (A_1 \wedge \neg b)\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\ \{B\}}. \tag{8.5}$$

To sum up, if our rules satify the following three criteria we can completely mechanically apply them backwards:

(C-1) The postconditions of the conclusion of the rule is a single (“open”) variable.

(C-2) The preconditions of the premisses of the rule are single, disjoint variables.

(C-3) All variables occuring in the precondition of the conclusion, the postconditions of the premises, or side-conditions (such as weakenings) in the premisses must also occur in either the postcondition of the conclusion, or the precondition of one of the premises of the rule. We say the variables are *determined*.

By composing with the weakening rule we can always construct rules which satisfy the first two criteria; to satisfy the third criterion we need to extend the syntax of our language of our language to allow the user to provide the needed information.

8.2 Approximative Weakest Preconditions

To make the process of reasoning backwards formal, we define the *weakest precondition*: given a postcondition Q, and a statement c, we want the weakest precondition to be an assertion P such that $\models \{P\}\, c\, \{Q\}$ and also that all other P' for which $\models \{P'\}\, c\, \{Q\}$ we get $P \Longrightarrow P'$ (*i.e.* P is weaker than P').

Definition 8.1 (Weakest Precondition) *Given an assertion $Q \in$ **Assn** and a statement $c \in$ **Stmt**, the* weakest precondition *is the assertion $P \in$ **Assn** such that*

$$\models \{P\}\, c\, \{Q\} \Longleftrightarrow P \Longrightarrow \mathrm{wp}(c, Q) \tag{8.6}$$

This is equivalent to the more obvious

$$\models \{\mathrm{wp}(c, Q)\}\, c\, \{Q\} \tag{8.7}$$
$$\models \{P\}\, c\, \{Q\} \Longrightarrow (P \Longrightarrow \mathrm{wp}(c, Q)) \tag{8.8}$$

because if $\models \{P\}\, c\, \{Q\}$ and $P' \Longrightarrow P$ then always $\models \{P'\}\, c\, \{Q\}$.

As mentioned above, we cannot hope to calculate the weakest precondition as a finite formula in general. However, with rules satisfying the three criteria (C-1) to (C-3) we can compute the *approximative weakest precondition* along with a set of verification conditions which have to proven separately. This formalises the process from above.

We calculate the approximative weakest precondition together with a set of *verification conditions*, by giving two functions such that for any given command $c \in$ **Stmt** and postcondition $Q \in$ **Assn**

$$\mathrm{awp}(c, Q) \in \mathbf{Assn}$$
$$\mathrm{wvc}(c, Q) \in \mathbb{P}(\mathbf{Assn})$$

(where the notation $\mathbb{P}(\mathbf{Assn})$ means we calculate a set of assertions). Here, $\mathrm{awp}(c, Q)$ is the approximative weakest precondition and $\mathrm{wvc}(c, Q)$ the set of verification condition. We define these functions by induction on the structure of the statement, see Figure 8.2. Together, they satisfy (8.7), but not (8.8), in the sense that if all verification conditions hold (or more precisely their conjunction), then the approximative weakest precondition is a precondition (but not necessarily the weakest one) for the statement c and the postcondition Q. Thus, we have the following statement which amounts to the *correctness* of the functions *awp* und *svc*:

$$\bigwedge_{p \in \mathrm{wvc}(c,Q)} p \Longrightarrow \models \{\mathrm{awp}(c, Q)\}\, c\, \{Q\} \tag{8.9}$$

In other words, the functions are guaranteed to find a valid precondition, but there may be a weaker one. This is because the annotated invariants determine the precondition and verification conditions, and there may be different invariants which hold; clearly, to write down a logical statement, there is are many different, logically equivalent ways, *e.g.* $A \wedge B$ or $B \wedge A$.

Theorem 8.1 (Correctness of *awp* and *svc*) *The definitions in Figure 8.2 are correct in the sense of (8.9).*

Proof. We prove the correctness by structural induction on c. For each of the constructors of our language we have to show (**??**). For this, we use the equivalence of the semantic validity of Hoare tripels and their

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}} \qquad \frac{}{\vdash \{A\}\, \{\}\, \{A\}} \qquad \frac{\vdash \{A\}\, c_1\, \{B\} \qquad \vdash \{B\}\, c_2\, \{C\}}{\vdash \{A\}\, c_1; c_2\, \{C\}}$$

$$\frac{I \wedge b \Longrightarrow B \quad \vdash \{B\}\, c\, \{I\} \quad I \wedge \neg b \Longrightarrow C}{\vdash \{I\}\ \textbf{while}\ (b)\ c\, \{C\}} \qquad \frac{\vdash \{A_0\}\, c_0\, \{B\} \quad \vdash \{A_1\}\, c_0\, \{B\}}{\vdash \{(A_0 \wedge b) \vee (A_1 \wedge \neg b)\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

$$\frac{A' \Longrightarrow A \quad \vdash \{A\}\, c\, \{B\} \quad B \Longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$

Figure 8.1: Rules of the backward Floyd-Hoare calculus.

$$\begin{aligned}
\mathrm{awp}(\{\}, P) &\stackrel{def}{=} P \\
\mathrm{awp}(x = e, P) &\stackrel{def}{=} P[e/x] \\
\mathrm{awp}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{awp}(c_1, \mathrm{awp}(c_2, P)) \\
\mathrm{awp}(\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, P) &\stackrel{def}{=} (b \wedge \mathrm{awp}(c_0, P)) \vee (\neg b \wedge \mathrm{awp}(c_1, P)) \\
\mathrm{awp}(/{*}{*}\ \{q\}\ {*}/, P) &\stackrel{def}{=} q \\
\mathrm{awp}(\textbf{while}\ (b)\ /{*}{*}\ \textbf{inv}\ i\ {*}/\ c, P) &\stackrel{def}{=} i \\
\\
\mathrm{wvc}(\{\}, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(x = e, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{wvc}(c_1, \mathrm{awp}(c_2, P)) \cup \mathrm{wvc}(c_2, P) \\
\mathrm{wvc}(\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, P) &\stackrel{def}{=} \mathrm{wvc}(c_0, P) \cup \mathrm{wvc}(c_1, P) \\
\mathrm{wvc}(/{*}{*}\ \{q\}\ {*}/, P) &\stackrel{def}{=} \{q \longrightarrow P\} \\
\mathrm{wvc}(\textbf{while}\ (b)\ /{*}{*}\ \textbf{inv}\ i\ {*}/\ c, P) &\stackrel{def}{=} \mathrm{wvc}(c, i) \cup \{i \wedge b \longrightarrow \mathrm{awp}(c, i)\} \cup \{i \wedge \neg b \longrightarrow P\}
\end{aligned}$$

Figure 8.2: The approximative weakest precondition and verification conditions

syntactic derivability, *i.e.* $\models \{P\}\,c\,\{Q\} \Longleftrightarrow \vdash \{P\}\,c\,\{Q\}$ (for a concrete constructor c). Thus, to show (**??**) we have to show

$$\bigwedge_{p \in \mathrm{wvc}(c,Q)} p \Longrightarrow \vdash \{\mathrm{awp}(c,Q)\}\,c\,\{Q\}$$

This can be directly read off the backward rules in Figure 8.1.

For example, for the sequential composition we have to show

$$\bigwedge_{p \in \mathrm{wvc}(c_1;c_2,Q)} p \Longrightarrow \models \{\mathrm{awp}(c_1;c_2,Q)\}\,;\{c_1\}c_2Q$$

which is equivalent to

$$\bigwedge_{p \in \mathrm{wvc}(c_1,Q) \cup \mathrm{wvc}(c_2,Q)} p \Longrightarrow \vdash \{\mathrm{awp}(c_1,\mathrm{awp}(c_2,Q))\}\,c_1;c_2\,\{Q\}.$$

Then by the rule for sequential composition in Figure 8.1 (actually, this is the usual sequential rule from Figure **??**) this can be derived from

$$\vdash \{\mathrm{awp}(c_2,Q)\}\,c_2\,\{Q\} \quad \text{and} \quad \vdash \{\mathrm{awp}(c_1,\mathrm{awp}(c_2,Q))\}\,c_1\,\{\mathrm{awp}(c_2,Q)\}$$

which is equivalent to showing

$$\models \{\mathrm{awp}(c_2,Q)\}\,c_2\,\{Q\} \quad \text{and} \quad \models \{\mathrm{awp}(c_1,\mathrm{awp}(c_2,Q))\}\,c_1\,\{\mathrm{awp}(c_2,Q)\}$$

and that is precisely the induction assumption for the case of sequential composition. □

The assertion spec `/** {q} */` forces the current precondition to become P. It can be seen as an enforced weakening with an empty statement, and an empty statement (because the comment is essentially an empty statement):

$$\frac{\vdash \{q\}\,\{\}\,\{q\} \quad q \Longrightarrow P}{\vdash \{q\}\,\{\}/{*}*\,\{q\}\,*/\,\{P\}}$$

In practice, we will be given a Hoare tuple $\{P\}\,c\,\{Q\}$, and want to calculate wether it holds. Then if all verification conditions hold, and if P implies $\mathrm{awp}(c,Q)$, the Hoare tuple holds:

$$\{P \longrightarrow \mathrm{awp}(c,Q)\} \cup \mathrm{wvc}(c,Q) \Longrightarrow \models \{P\}\,c\,\{Q\}$$

That is, to verify a given program, we are only interested in the verification conditions, and calculate the awp only as an auxiliary. Let us consider some examples.

8.3 Simplifying Verification Conditions

The first example is an old friend: the factorial function, here annotated with the invariant:

```
1  /** { 0 ≤ n } */
2  p= 1;
3  c= 1;
4  while (c <= n) /** inv p= (c−1)! ∧ c−1 ≤ n; */ {
5    p = p * c;
6    c = c + 1;
7    }
8  /** { p = n! } */
```

When calculating the awp and the verification conditions, we use the following notation:

- We annotate the awp *before* the line it belongs to.
- We write down the generated verification conditions separately, using the notation vc_n for the VCs generated in line n. We also number the VCs consecutively, so we can refer to them later.
- We will then give a set of rules to simplify and break down the verification conditions, such that many of them can be proven trivially; a few will remain, which have to be proven manually (or an external prover).

Going back from the postcondition in our example, we arrive at the following with all awps annotated:

```
1  /** { 0 ≤ n } */
2  // {1 = (1−1)! ∧ 1−1 ≤ n}
3  p= 1;
4  // {p = (1−1)! ∧ 1−1 ≤ n}
5  c= 1;
6  // {p = (c−1)! ∧ c−1 ≤ n}
7  while (c <= n) /** inv p= (c−1)! ∧ c−1 ≤ n; */ {
8    // {p·c = ((c+1)−1)! ∧ (c+1)−1 ≤ n}
9    p = p * c;
10   // {p = ((c+1)−1)! ∧ (c+1)−1 ≤ n}
11   c = c + 1;
12   // {p = (c−1)! ∧ c−1 ≤ n}
13   }
14 /** { p = n! } */
```

The calculation of the awp yields the following verification conditions:

$$
\begin{array}{lll}
(1)\ p = (c-1)! \wedge c-1 \leq n \wedge c \leq n \longrightarrow p \cdot c = ((c+1)-1)! \wedge ((c-1)+1) \leq n & vc_4 \\
(2)\ p = (c-1)! \wedge c-1 \leq n \wedge \neg(c \leq n) \longrightarrow p = n! & vc_4 \\
(3)\ 0 \leq n \longrightarrow 1 = (1-1)! \wedge (1-1) \leq n & vc_1
\end{array}
$$

To simplify verification conditions (vcs), we use to the following rules:

(SIMP-1) A conjuction is simplified to two different vcs:

$$P \longrightarrow A \wedge B \rightsquigarrow P \longrightarrow A, P \longrightarrow B$$

(SIMP-2) Constant arithmetic expressions are evaluated and simple arithmetic laws are employed (such as $x+0 = x, x+a-a = a$).

(SIMP-3) The relations are normalised as follows:

$$
\begin{array}{lll}
x > y \rightsquigarrow y < x & \neg(x = y) \rightsquigarrow x \neq y & \neg(x < y) \rightsquigarrow y \leq x \\
x \geq y \rightsquigarrow y \leq x & \neg(x \neq y) \rightsquigarrow x = y & \neg(x \leq y) \rightsquigarrow y < x
\end{array}
$$

This means that after simplification there are no $>$, $\geq$ or negated relations left.

(SIMP-4) The following vcs are trivially proven:

- if the conclusion appears in the premises
- if the conclusion is simplified to *true*;
- if one of the premises is simplified to *false*.

With these rules, the verification conditions simplify as follows:

(1.1)	$p = (c-1)! \wedge c-1 \leq n \wedge c \leq n \longrightarrow p \cdot c = c!$	
(1.2)	$p = (c-1)! \wedge c-1 \leq n \wedge c \leq n \longrightarrow c \leq n$	Trivial.
(2)	$p = (c-1)! \wedge c-1 \leq n \wedge n < c \longrightarrow p = n!$	
(3.1)	$0 \leq n \longrightarrow 1 = 0!$	Holds with $0! = 1$
(3.2)	$0 \leq n \longrightarrow 0 \leq n$	Trivial.

As we can see, (1.2), (3.1) and (3.2) are trivial.

To prove (2): we can assume $n < c$, hence $n \leq c-1$, hence with $c-1 \leq n$, we can conclude $c-1 = n$ and with $p = (c-1)!$ we can conclude $p = n!$ as required.

To prove (1.1): from $p = (c-1)!$ we can conclude $p \cdot c = (c-1)! \cdot c$, and with $(n-1)! \cdot n = n!$, we can conclude $p \cdot c = c!$ as required. TO DO.

This needs the side condition that $0 < c$ which is still lacking.

□

8.4 Simplifying Disjunctions

The second example is the program which finds the maximal element in an array, which we have already seen in Section **??** (Figure 7.4), with only the invariant annotated. We can see that the actual source code is much shorter, and the good news is that from that source code we can actually derive all information which in Figure 7.4 had to be painstakingly annotated by hand. The initial source looks like this:

```
1  // {0 < n}
2  i= 0;
3  r= 0;
4  while (i< n) /** inv (∀j. 0≤ j< i ⟶a[j]≤ a[r]) ∧ 0≤ i≤ n ∧
   0≤ r< n */ {
5    if (a[r] < a[i]) {
6      r= i;
7      }
8    else {
9      }
10   i= i+1;
11   }
12 // {(∀j.0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Like the one above, this example will have three verification conditions: two for the while loop, and one for the initial precondition. The calculation of the vcs and the awp can be modularized; we can calculate two vcs without considering the loop body:

```
1  // {0 < n}
2  // {(∀j.0 ≤ j < 0 ⟶ a[j] ≤ a[0]) ∧ 0 ≤ 0 ≤ n ∧ 0 ≤ 0 < n}
3  i= 0;
4  // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[0]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ 0 < n}
5  r= 0;
6  // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
7  while (i< n) /** inv (∀j. 0≤ j< i ⟶a[j]≤ a[r]) ∧ 0≤ i≤ n ∧ 0≤ r< n */ {
8    if (a[r] < a[i]) {
9      r= i;
10     }
11   else {
12     }
13   i= i+1;
14   }
15 // {(∀j.0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

This gives us the following two vcs:

$$
\begin{array}{lll}
(1) & 0 < n \longrightarrow (\forall j.0 \le j < 0 \longrightarrow a[j] \le a[0]) \wedge 0 \le 0 < n \wedge 0 \le 0 < n & vc_1 \\
(2) & (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le i \le n \wedge 0 \le r < n \wedge \neg(i < n) & \\
 & \quad \longrightarrow (\forall j.0 \le j < n \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n & vc_{15}
\end{array}
$$

The loop body can be considered in isolation:[2]

```
1  while (i< n) /** inv (∀j. 0≤ j< i ⟶a[j]≤ a[r]) ∧ 0≤ i≤ n ∧ 0≤ r< n */ {
2    // {(a[r] < a[i] ∧ (∀j.0 ≤ j < i+1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i+1 ≤ n ∧ 0 ≤ i < n)
       ∨ (¬(a[r] < a[i]) ∧ (∀j.0 ≤ j < i+1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i+1 ≤ n ∧ 0 ≤ r < n)}
3    if (a[r] < a[i]) {
4      // {(∀j.0 ≤ j < i+1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i+1 ≤ n ∧ 0 ≤ i < n}
5      r= i;
6      // {(∀j.0 ≤ j < i+1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i+1 ≤ n ∧ 0 ≤ r < n}
7      }
8    else {
9      // {(∀j.0 ≤ j < i+1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i+1 ≤ n ∧ 0 ≤ r < n}
10     }
11   // {(∀j.0 ≤ j < i+1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i+1 ≤ n ∧ 0 ≤ r < n}
12   i= i+1;
13   // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
14   }
```

This gives the third verification condition:

$$
\begin{array}{lll}
(3) & (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le i \le n \wedge 0 \le r < n \wedge i < n & vc_2 \\
 & \quad \longrightarrow (a[r] < a[i] \wedge (\forall j.0 \le j < i+1 \longrightarrow a[j] \le a[i]) \wedge 0 \le i+1 \le n \wedge 0 \le i < n) & \\
 & \qquad \vee (\neg(a[r] < a[i]) \wedge (\forall j.0 \le j < i+1 \longrightarrow a[j] \le a[r]) \wedge 0 \le i+1 \le n \wedge 0 \le r < n) &
\end{array}
$$

[2] Note that in line 13 we have annotated a postcondition. This is of course the invariant, and not required by the definition of the awp; it is here purely for convenience.

Quite a handful! Even worse, it cannot be simplified meaningfully by our rules, because of the disjunction in the conclusion, so it is hard to prove as it is. As a situation like this will occur quite often with case distinctions, it seems worthwhile to consider more general solutions.

First, we come up with a strategy to prove propositions like (3) above, and then we consider a language extension by which to prevent verification conditions like that from arising in the first place.

In formal logic, we have the following general rule (known as the "elimination rule for the disjunction"):

$$\frac{P \vee Q \quad P \longrightarrow R \quad Q \longrightarrow R}{R} \tag{8.10}$$

This rule says that if we have disjunction $P \vee Q$, and we can prove some common proposition R from both P and Q, then surely R holds without a precondition. We can look at this as a case distinction: we consider both cases of the disjunction. If the disjunction is $B \vee \neg B$, this will hold for any B, so we get case distinction over a proposition B as a derived rule:

$$\frac{B \longrightarrow R \quad \neg B \longrightarrow R}{R} \tag{8.11}$$

Lemma 8.2 (Simplifying Case Distinctions) *To show the case distinction $(A \wedge B) \vee (C \wedge \neg B)$, it is sufficient to show $B \longrightarrow A$ and $\neg B \longrightarrow C$.*

Proof. With $B \longrightarrow A$ and $B \longrightarrow B$ (which always holds) we get $B \longrightarrow A \wedge B$,w and with that $B \longrightarrow (A \wedge B) \vee (C \wedge \neg B)$ (if we can show $P \longrightarrow Q$, we can always show $P \longrightarrow Q \vee R$). Similarly, from $\neg B \longrightarrow C$, we get $\neg B \longrightarrow C \wedge \neg B$ and hence $\neg B \longrightarrow (A \wedge B) \vee (C \wedge \neg B)$, so by rule (8.11) we get the required conclusion. □

Lemma 8.2 allows us to simplify the complicated case distinction $(A \wedge B) \vee (C \wedge \neg B)$ to the two implications $B \longrightarrow A$ and $\neg B \longrightarrow C$. More general, in the presence of assumptions, we have the following additional simplication rule:

(SIMP-5) A disjunction can be simplified as follows:

$$P \longrightarrow (A \wedge B) \vee (C \wedge \neg B) \rightsquigarrow P \wedge B \longrightarrow A, P \wedge \neg B \longrightarrow C$$

In our case, this means that verification condition (3) simplifies to

$$\begin{array}{ll} (3.1) & (\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq i \leq n \wedge 0 \leq r < n \wedge i < n \wedge a[r] < a[i] \\ & \longrightarrow (\forall j.0 \leq j < i+1 \longrightarrow a[j] \leq a[i]) \wedge 0 \leq i+1 \leq n \wedge 0 \leq i < n) \\ (3.2) & (\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq i \leq n \wedge 0 \leq r < n \wedge i < n \wedge a[i] \leq < a[r] \\ & \longrightarrow (\forall j.0 \leq j < i+1 \longrightarrow a[j] \leq a[r]) \wedge 0 \leq i+1 \leq n \wedge 0 \leq r < n) \end{array}$$

These look quite similar, but are substantially different— (3.1) talks about i in the conclusion where (3.2) talks about r. The two conditions simplify further. Collecting and simplifiying the other vcs (1), (2) gives

us the following:

(1.1)	$0 < n \longrightarrow \forall j.0 \leq j < 0 \longrightarrow a[j] \leq a[0]$	
(1.2)	$0 < n \longrightarrow 0 \leq 0 < n$	Trivial.
(1.3)	$0 < n \longrightarrow 0 \leq 0 < n$	Trivial.
(2.1)	$(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq i \leq n \wedge 0 \leq r < n \wedge n \leq i$ $\longrightarrow (\forall j.0 \leq j < n \longrightarrow a[j] \leq a[r])$	
(2.2)	$(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq i \leq n \wedge 0 \leq r < n \wedge n \leq i \longrightarrow 0 \leq r < n$	Trivial.
(3.1.1)	$(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq i < n \wedge 0 \leq r < n \wedge a[r] < a[i]$ $\longrightarrow (\forall j.0 \leq j < i+1 \longrightarrow a[j] \leq a[i])$	
(3.1.2)	$(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq i < n \wedge 0 \leq r < n \wedge a[r] < a[i] \longrightarrow 0 \leq i+1 \leq n$	
(3.1.3)	$(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq i < n \wedge 0 \leq r < n \wedge a[r] < a[i] \longrightarrow 0 \leq i < n$	Trivial.
(3.2.1)	$(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq i < n \wedge 0 \leq r < n \wedge a[i] \leq a[r]$ $\longrightarrow (\forall j.0 \leq j < i+1 \longrightarrow a[j] \leq a[r])$	
(3.2.2)	$(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq i < n \wedge 0 \leq r < n \wedge a[i] \leq a[r] \longrightarrow 0 \leq i+1 \leq n$	See 3.1.2
(3.2.3)	$(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq i < n \wedge 0 \leq r < n \wedge a[i] \leq a[r] \longrightarrow 0 \leq r < n)$	Trivial.

All in all, after simplification we have eleven proof obligations. Of these, five are trivially proven (these correspond to passing arounds facts related to variables which do not change).[3] Verification conditions which are not proven trivially are called *proof obligations*— six of them here. Four are proven quite easily (what mathematicians might colloquially refer to as trivial): (1.1) includes a precondition $0 \leq j < 0 \longrightarrow \ldots$ which is equivalent to $\mathit{false} \longrightarrow \ldots$ and so is always true; (2.1) allows us to conclude $n = i$ from $i \leq n$ and $n \leq i$, and with that the conclusion; and in (3.1.2) and (3.2.2) $i < n$ implies $i+1 \leq n$.

It remains to prove (3.1.1) and (3.2.1). The proofs also occur in the correctness proof in the last chapter, and they are the "algorithmic core" of the program, namely that when increasing the range from $0,\ldots,i$ to $0,\ldots,i+1$ we have to consider wether either the current maximum $a[r]$ is less than $a[i]$ (in that case, i is the index of the new maxium) or not.

For (3.2.1), from $a[i] \leq a[r]$ and $\forall j.0 \leq j < i \longrightarrow a[j] \leq a[i]$ we can conclude that $\forall j.0 \leq j < i+1 \longrightarrow a[j] \leq a[i]$, using the range extension theorem (7.5) (from page 55). For (3.1.1): from the assumption $a[r] \leq a[i]$ and $a[j] < a[i]$, we can conclude (by transitivity of $\leq$) that $\forall j.0 \leq j < i \longrightarrow a[j] \leq a[i]$, and we further have $a[i] \leq a[i]$ by reflexivity. Now, we can conclude by (7.5) that $\forall j.0 \leq j < i+1 \longrightarrow a[j] \leq a[i]$ as required.

8.5 Explicit Assertions

To prevent a verification condition which is a lengthy disjunction from occuring in the first place, consider that the disjunction comes from the more general form of the if-rule (8.5), which can be applied in any situation. But if the approximative weakest preconditions of the positive and negative branch are in fact of the form required by the original if-rule, there is no disjunction and a far simpler precondition, so it would appear to make sense to use that rule instead. Even in the face of Lemma 8.2, preventing such a lengthy disjunction from occuring is worthwhile, as the precondition gets propagated through the prececeding statements[4]— this has not happened in our example because the case distinction is the first statement of the while-loop.

[3] "Trivially proven" has a technical meaning here, meaning that the conclusion appears inthe precondition after simplification.

[4] Consider a case distinction within a case distinction!

Accodingly, we add the following derived clause to the definition of *awp*:

$$\text{awp}(\textbf{if } (b)\ c_0 \textbf{ else } c_1, P) \stackrel{def}{=} Q \quad \text{if awp}(c_0,P) = b \wedge Q \text{ and awp}(c_1,P) = \neg b \wedge Q \tag{8.12}$$

This easily follows from the definition in Figure 8.2 if we assume $\text{awp}(c_0,P) = b \wedge Q$ and $\text{awp}(c_1,P) = \neg b \wedge Q$, then $\text{awp}(\textbf{if } (b)\ c_0 \textbf{ else } c_1, P) = (b \wedge Q) \vee (\neg b \wedge Q) = (b \vee \neg b) \wedge Q = \mathit{true} \wedge Q = Q$.

As the approximative weakest preconditions of the two branches are calculated rather than given, if we want to use the simplified version (8.12) of the awp rule we have to force them into the required form by using an *explicit assertion*. This is done as follows:

```
1  while (i< n) /** inv (∀j. 0≤ j< i ⟶ a[j]≤ a[r]) ∧ 0≤ i≤ n ∧ 0≤ r< n */ {
2    if (a[r] < a[i]) {
3      /** {a[r]< a[i] ∧ (∀j. 0≤ j< i ⟶ a[j]≤ a[r]) ∧ 0≤ i< n ∧ 0≤ r< n } */
4      r= i;
5      }
6    else {
7      /** {¬(a[r]< a[i]) ∧ (∀j. 0≤ j< i ⟶ a[j]≤ a[r]) ∧ 0≤ i< n ∧ 0≤ r< n } */
8      }
9    i= i+1;
10   }
```

As a syntactic note, this requires us to write down the envisaged precondition Q twice, so it would seem more useful to consider an annotation to the if-statement where we give Q directly, like this:

```
/** {(∀j. 0≤ j< i⟼a[j]≤ a[r]) ∧ 0≤ i< n ∧ 0≤ r< n} */
if (a[r] < a[i]) {
  r= i;
  }
else {
  }
```

However, using explicit annotations is far more versatile, so we prefer to keep it as a primitve concept, and use the annotation to the if-statement as syntactic sugar.

To continue with our example, with the explicit annotation we get the following awps and verification conditions (using the syntactic sugar just introduced):

```
1  while (i< n) /** inv (∀j. 0≤ j< i ⟶ a[j]≤ a[r]) ∧ 0≤ i≤ n ∧ 0≤ r< n */ {
2    /** {(∀j. 0≤ j< i⟼a[j]≤ a[r]) ∧ 0≤ i< n ∧ 0≤ r< n} */
3    if (a[r] < a[i]) {
4      // {(∀j.0 ≤ j < i+1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i+1 ≤ n ∧ 0 ≤ i < n}
5      r= i;
6      // {(∀j.0 ≤ j < i+1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i+1 ≤ n ∧ 0 ≤ r < n}
7      }
8    else {
9      // {(∀j.0 ≤ j < i+1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i+1 ≤ n ∧ 0 ≤ r < n}
10     }
11   // {(∀j.0 ≤ j < i+1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i+1 ≤ n ∧ 0 ≤ r < n}
12   i= i+1;
13   // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
14   }
```

$$
\begin{array}{lll}
(4) & (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le i \le n \wedge 0 \le r < n \wedge i < n & \\
& \longrightarrow (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le i < n \wedge 0 \le r < n & vc_1 \\
(5) & a[r] < a[i] \wedge (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le i < n \wedge 0 \le r < n & \\
& \longrightarrow (\forall j.0 \le j < i+1 \longrightarrow a[j] \le a[r]) \wedge 0 \le i+1 \le n \wedge 0 \le i < n & vc_4 \\
(6) & \neg(a[r] < a[i]) \wedge (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le i < n \wedge 0 \le r < n & \\
& \longrightarrow (\forall j.0 \le j < i+1 \longrightarrow a[j] \le a[r]) \wedge 0 \le i+1 \le n \wedge 0 \le r < n & vc_9
\end{array}
$$

Simplifying further, we obtain the following:

$$
\begin{array}{lll}
(4.1) & (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le i \le n \wedge 0 \le r < n \wedge i < n \longrightarrow (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) & \text{Trivial.} \\
(4.2) & (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le i \le n \wedge 0 \le r < n \wedge i < n \longrightarrow 0 \le i < n & \text{Trivial.} \\
(4.3) & (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le i \le n \wedge 0 \le r < n \wedge i < n \longrightarrow 0 \le r < n & \text{Trivial.} \\
(5.1) & a[r] < a[i] \wedge (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le i < n \wedge 0 \le r < n & \\
& \longrightarrow (\forall j.0 \le j < i+1 \longrightarrow a[j] \le a[r]) & \\
(5.2) & a[r] < a[i] \wedge (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le i < n \wedge 0 \le r \le n \longrightarrow 0 \le i+1 \le n & \text{Trivial.} \\
(5.3) & a[r] < a[i] \wedge (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le i < n \wedge 0 \le r < n \longrightarrow 0 \le i < n & \text{Trivial.} \\
(6.1) & \neg(a[r] < a[i]) \wedge (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le i < n \wedge 0 \le r < n & \\
& \longrightarrow (\forall j.0 \le j < i+1 \longrightarrow a[j] \le a[r]) & \\
(6.3) & \neg(a[r] < a[i]) \wedge (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le i < n \wedge 0 \le r < n \longrightarrow 0 \le i+1 \le n & \\
(6.3) & \neg(a[r] < a[i]) \wedge (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le i < n \wedge 0 \le r < n \longrightarrow 0 \le r < n & \text{Trivial.}
\end{array}
$$

It is instructive to compare these nine proof obligations to the six proof obligations (3.1.1) to (3.2.3) from above. The first three (4.1) to (4.3) correspond to the weakening from the annotated invariant to the explicit assertion; as the assertion is exactly the invariant, this is trivial. The other six correspond exactly to (3.1.1) to (3.2.3) and are proven in the same way.

8.6 Conclusions

We do not have to annotate the whole program as we have seen in the previous chapter, we can indeed derive preconditions automatically by applying the rules of the Hoare calculus backwards, starting with a given postcondition.

However, while loops need an invariant which cannot be derived automatically. For this, we need to annotate the invariant to the while loop; this is done by a special kind of comment which can be picked up by the verification but is ignored by the semantics.

An annotated program allows us to derive a *approximative weakest precondition* together with a set of *verification conditions*. If we can prove the verification conditions, the program satisfies the annotated specification. To that end, we have considered some rules by which we can simplify verification conditions (they can become quite lengthy and unwieldy).

We further introduced explicit assertions, which allow us to introduce a given precondition at a certain point. This is helpful, as it keeps the precondition from getting too large.

Chapter 9

Forwards with Floyd-Hoare!

9.1 Going Forward

In Chapter 8 we have seen how applying our rules backwards gives us an algorithm to compute the approximate weakest precondition, and corresponding verification condition. However, this may generate unwieldy verification conditions, as we may not be able to simplifiy expressions (in particular substitutions).

What about going forward?

9.1.1 Rules

On page 60 we have given three criteria to apply rules backward. To apply rules forward, we just need to "dualise" these. A rules is applicable forward if it satisfies the following three criteria:

(F-1) The preconditions of the conclusion of the rule is a single ("open") variable.

(F-2) The postconditions of the premisses of the rule are single, disjoint variables.

(F-3) All variables occuring in the postcondition of the conclusion, the preconditions of the premises, or side-conditions (such as weakenings) in the premisses must also occur in either the precondition of the conclusion, or the postcondition of one of the premises of the rule.

If we look at our rules, it is clear that the assignment rule cannot be applied forward, and we shall adress this problem in a second. But looking at the other rules, we can see they are nearly all suitable as they are, the only small problem being the while-rule. Here, the precondition of the conclusion is precisely the invariant (which needs to be annotated). One could use the rule as is, but in practice the calculated postcondition of the statement before the while-loop will not be suitable as an invariant. So, just like when going backwards we need an annotated invariant, and we need a weakening towards that invariant. And secondly, in the postcondition of the premisse of the while-rule we again have the invariant where we need an open postcondition, so we get a second weakening. We obtain a rule which, just like when going forward, has two weakenings as preconditions, resulting in two verification conditions being generated per while-loop. The rules are given in Figure 9.1.

$$\frac{V \notin FV(P)}{\vdash \{P\}\, x = e\, \{\exists V.\, P[V/x] \wedge x = e[V/x]\}} \qquad \frac{}{\vdash \{A\}\, \{\,\}\, \{A\}} \qquad \frac{\vdash \{A\}\, c_1\, \{B\} \qquad \vdash \{B\}\, c_2\, \{C\}}{\vdash \{A\}\, c_1; c_2\, \{C\}}$$

$$\frac{A \Longrightarrow I \quad \vdash \{I \wedge b\}\, c\, \{B\} \quad B \Longrightarrow I}{\vdash \{A\}\ \textbf{while}\ (b)\ \ \textbf{inv}\ I\ \ c\, \{I \wedge \neg b\}} \qquad \frac{\vdash \{A \wedge b\}\, c_0\, \{B_1\} \quad \vdash \{A \wedge \neg b\}\, c_1\, \{B_2\}}{\vdash \{A\}\ \textbf{if}\ (b)\ /\!*\, c_0\ \textbf{else}\ c_1\, \{B_1 \vee B_2\}}$$

$$\frac{A' \Longrightarrow A \quad \vdash \{A\}\, c\, \{B\} \quad B \Longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$

Figure 9.1: Rules of the forward Floyd-Hoare calculus.

$$\begin{aligned}
\mathrm{asp}(P, \{\,\}) &\stackrel{def}{=} P \\
\mathrm{asp}(P, x = e) &\stackrel{def}{=} \exists V.\, P[V/x] \wedge x = (e[V/x]) \\
\mathrm{asp}(P, c_1; c_2) &\stackrel{def}{=} \mathrm{asp}(\mathrm{asp}(P, c_1), c_2) \\
\mathrm{asp}(P, \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1) &\stackrel{def}{=} \mathrm{asp}(b \wedge P, c_0) \vee \mathrm{asp}(\neg b \wedge P, c_1) \\
\mathrm{asp}(P, /\!*\, \{q\}\, *\!/) &\stackrel{def}{=} q \\
\mathrm{asp}(P, \textbf{while}\ (b)\ /\!*\ \textbf{inv}\ i\ *\!/\ c) &\stackrel{def}{=} i \wedge \neg b \\
\\
\mathrm{svc}(P, \{\,\}) &\stackrel{def}{=} \emptyset \\
\mathrm{svc}(P, x = e) &\stackrel{def}{=} \emptyset \\
\mathrm{svc}(P, c_1; c_2) &\stackrel{def}{=} \mathrm{svc}(P, c_1) \cup \mathrm{svc}(\mathrm{asp}(P, c_1), c_2) \\
\mathrm{svc}(P, \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1) &\stackrel{def}{=} \mathrm{svc}(P \wedge b, c_0) \cup \mathrm{svc}(P \wedge \neg b, c_1) \\
\mathrm{svc}(P, /\!*\, \{q\}\, *\!/) &\stackrel{def}{=} \{P \longrightarrow q\} \\
\mathrm{svc}(P, \textbf{while}\ (b)\ /\!*\ \textbf{inv}\ i\ *\!/\ c) &\stackrel{def}{=} \mathrm{svc}(i \wedge b, c) \cup \{P \longrightarrow i\} \cup \{\mathrm{asp}(i \wedge b, c) \longrightarrow i\} \\
\\
\mathrm{svc}(()\{P\}\, c\, \{Q\}) &\stackrel{def}{=} \{\mathrm{asp}(P, c) \longrightarrow Q\} \cup \mathrm{svc}(P, c)
\end{aligned}$$

Figure 9.2: Approximative strongest postcondition

We return to the problem of the assigment rule. Her, we need an alternative version suitable for forward reasoning. Historically, this is the older version which Floyd came up with originally before Hoare replaced it with the simpler backward rule:

$$\frac{V \notin FV(P)}{\vdash \{P\}\, x = e\, \{\exists V.\, P[V/x] \wedge x = e[V/x]\}} \tag{9.1}$$

Here, $FV(P)$ are the *free variables* in P.

The rule has an existential quantifier which makes for awkward reasoning, and in particular composition. It can be read as a reference to an indetermined previous value of a variable ("there once was a value of x"). When we compose assignment statements without any simplifcation there would be one existential quantifier for each assignment statement.

9.1.2 Forward Chaining

Consider two simple assignment statements:

```
// {0 ≤ b}
a= b+ 7;
b= 9;
```

We can use the forward rule to derive postconditions:

```
// {0 ≤ b}
a= b+ 7;
// {∃A. (0 ≤ b)[A/a] ∧ a = (b+7)[A/a]}
// {∃A. 0 ≤ b ∧ a = b+7}
```

The first thing to note here is that A does not occur inside the predicate at all. In that case, we can get rid of the existential quantor. Written as a rule:

$$\exists X.\, P \Longrightarrow P \quad \text{if } X \notin FV(P) \tag{9.2}$$

This corresponds to an assigment where the assigned variable is neither referred to in the precondition nor on the right-hand side of the assignment. Continuing, we get:

```
// {0 ≤ b}
a= b+ 7;
// {0 ≤ b ∧ a = b+7}
b= 9;
// {∃B. (0 ≤ b ∧ a = B+7)[B/b] ∧ b = 9[B/b]}
// {∃B. 0 ≤ B ∧ a = B+7 ∧ b = 9}
```

This time, cannot get rid of the existential quantifier. The final postcondition describes two things: firstly, the transformed precondition ($\exists B.\, 0 \leq B$ — "there once was a value which was larger than zero"), and secondly a symbolic description of the resulting state: a is what B once was (and which we know was larger than zero) plus 7, and b is 9.

Let us consider another example, this time with more self-references:

```
// {0 ≤ a}
a= b*a;
```

```
// {∃A1.(0 ≤ a)[A1/a] ∧ a = (b·a)SubstA1a}
// {∃A1.0 ≤ A1 ∧ a = b·A1}
a= a+5;
// {∃A2.(∃A1.0 ≤ A1 ∧ a = b·A1)[A2/a] ∧ a = (a+5)[A2/a]}
// {∃A2.(∃A1.0 ≤ A1 ∧ A2 = b·A1) ∧ a = A2+5}
```

To simplify this, consider that it existentially quantifies over a variable A_2 with the restriction that $A_2 = b \cdot A_1$. This means, "there is a value for A_2 such that $A_2 = b \cdot A_1$", so it actually gives us that value. Expressed as a rule, this is

$$\exists X.(P(X) \wedge X = t) \Longrightarrow P[t/X] \quad \text{if } X \notin FV(t) \tag{9.3}$$

This rule is applicable here, but we have to be careful: the scope of the existential quantifier A_1 does not extend to the final equation $a = A_2 + 5$, so it does not fit the pattern of (9.3). Fortunately, we can always extend (or limit) the scope of the quantifier as long as the variable quantified over does not occur in the term[1]

$$(\exists X.P) \wedge Q \Longleftrightarrow \exists X.P \wedge Q \quad \text{if } X \notin FV(Q) \tag{9.4}$$

This gives us

$$\begin{aligned} & \exists A_2.\exists A_1.0 \leq A_1 \wedge A_2 = b \cdot A_1 \wedge a = A_2 + 5 \\ \Longleftrightarrow & \exists A_1.0 \leq A_1 \wedge a = b \cdot A_1 + 5 \end{aligned}$$

As a final example, consider what happens if we bind the value of a variable to a logical variable using an equation of the form $x = X$:

```
// {a = A}
a= a+1;
// {∃A1.(a = A)[A1/a] ∧ a = (a+1)[A1/a],}
// {∃A1.A1 = A ∧ a = A1 + 1}
// {a = A+1}
```

The last simplification was with rule (9.3). As we can see, the equation $a = A$ made that rule applicable, and had the effect of giving a name to the existentially quantified variable A_1 and making it disappear. The postcondition says exactly what this program does: the value of a afterwards is the value of a before plus one.

Pulling all of this together, here are the simplification rules for existential quantifiers:

(SIMP-6) The scope of the existential quantifier should be made as large as possible:

$$(\exists X.P) \wedge Q \rightsquigarrow \exists X.P \wedge Q \quad \text{if } X \notin FV(Q)$$

(SIMP-7) Equalities are substituted:

$$P(X) \wedge X = t \rightsquigarrow P[t/X]$$

(SIMP-8) Quantifiying over an unused variable is not necessary:

$$\exists X.P \rightsquigarrow P \quad \text{if } X \notin FV(P)$$

Rule (9.3) is just a combination of the last two simplifications; the reason to keep them separate is that simplification of equations is useful on its own. (Note that we can only substitute equations $X = t$ where X is a logical variable; this does not work for state variables, where we need to keep this equation.)

[1] If X occured in Q, this would be called a "capture", as in "the variable X is caputered in Q.

9.1.3 Correctness of the Forward Assignment Rule

The forward assignment rule (9.1) is *not a derived rule* — we have to prove it correct just like we did with the original assignment rule.

Lemma 9.1 (Correctness of the forward assignment rule) *Rule (9.1) is correct:*

$$\models \{P\}\, x = e\, \{\exists V.\, P[V/x] \wedge x = (e[V/x])\} \tag{9.5}$$

Proof. To show (9.1), we unfold the definition of the Hoare triple, so we have to show

$$\begin{aligned} &\models \{P\}\, x = e\, \{\exists V.\, P[V/x] \wedge x = (e[V/x])\} \\ \Longleftrightarrow\ &\forall I.\, \forall \sigma.\, \sigma \models^I P \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![x = e]\!]_{\mathscr{C}} \Longrightarrow \sigma' \models^I \exists V.\, P[V/x] \wedge x = (e[V/x]) \end{aligned}$$

From the definition of the denotational semantics of the assigment, we get $\sigma[x \mapsto [\![e]\!]_{\mathscr{A}}]$ as the σ' which satisfies $(\sigma, \sigma') \in [\![x = e]\!]_{\mathscr{C}}$:

$$\Longleftrightarrow\ \forall I.\, \forall \sigma.\, \sigma \models^I P \Longrightarrow \sigma[x \mapsto [\![e]\!]_{\mathscr{A}}] \models^I \exists V.\, P[V/x] \wedge x = (e[V/x])$$

We can now use the Substitution Lemma 6.2 (demonstrating again its central role in modelling state update by syntactic substitution):

$$\begin{aligned} \Longleftrightarrow\ &\forall I.\, \forall \sigma.\, \sigma \models^I P \Longrightarrow \sigma \models^I (\exists V.\, P[V/x] \wedge x = (e[V/x]))[e/x] \\ \Longleftrightarrow\ &\forall I.\, \forall \sigma.\, \sigma \models^I P \Longrightarrow \sigma \models^I \exists V.\, (P[V/x])[e/x] \wedge e = ((e[V/x])[e/x]) \end{aligned}$$

In $P[V/x]$ all occurences of x have been replaced by V, so substituting x again will have no effect:

$$\Longleftrightarrow\ \forall I.\, \forall \sigma.\, \sigma \models^I P \Longrightarrow \sigma \models^I (\exists V.\, P[V/x] \wedge e = (e[V/x]))$$

Finally, we know there is a term which when we substitute it for V in P and e satisfies $P[V/x]$ and $e = e[V/x]$ respectively— it is x:

$$\begin{aligned} &\Longleftrightarrow \forall I.\, \forall \sigma.\, \sigma \models^I P \Longrightarrow \sigma \models^I P[x/x] \wedge e = (e[x/x]) \\ &\Longleftrightarrow \forall I.\, \forall \sigma.\, \sigma \models^I P \Longrightarrow \sigma \models^I P \wedge e = e \end{aligned}$$

□

9.2 Examples

9.2.1 The Factorial Example

We start with our old friend, the factorial function:

```
1  // {0 ≤ n}
2  p= 1;
3  c= 1;
4  while (c <= n) /** inv p= (c−1)! ∧ 0≤ c ∧ c−1≤ n; */ {
5    p = p * c;
6    c = c + 1;
7    }
8  // {p = n!}
```

We first calculate the strongest postconditions, using the same notation as for the weakest precondition. When going through a sequence of statements, we can either simplify the postconditions on the fly, or at the end — logically, it does not make any difference. However, with the forwards approach we can simplify conditions far more effectively[2], so we will take this approach in the following, and simplify postconditions as much as possible. In the first example, we will still write down the simplifications, but later on we only write down the simplified postcondition:

```
1  // {0 ≤ n}
2  p= 1;
3  // {∃P.(0 ≤ n)[P/p] ∧ p = (1[P/p])}
4  // {0 ≤ n ∧ p = 1}
5  c= 1;
6  // {∃C.(0 ≤ n ∧ p = 1)[C/c] ∧ c = (1[C/c])}
7  // {0 ≤ n ∧ p = 1c = 1}
8  while (c < n) /** inv p= (c−1)! ∧ 0< c ∧ c−1≤ n; */ {
9    // {p = (c−1)! ∧ 0 < c ∧ c−1 ≤ n ∧ c < n}
10   p = p * c;
11   // {∃P.(p = (c−1)! ∧ 0 < c ∧ c−1 ≤ n ∧ c < n)[P/p] ∧ p = (p·c[P/p])}
12   // {∃P.P = (c−1)! ∧ 0 < c ∧ c−1 ≤ n ∧ c < n ∧ p = P·c}
13   c = c + 1;
14   // {∃C.(∃P.P = (c−1)! ∧ 0 < c ∧ c−1 ≤ n ∧ c < n ∧ p = P·c)[C/c] ∧ c = (c+1[C/c])}
15   // {∃C.∃P.P = (C−1)! ∧ 0 < C ∧ C−1 ≤ n ∧ C < n ∧ p = P·C ∧ c = C+1}
16   // {∃C.0 < C ∧ C−1 ≤ n ∧ C < n ∧ p = (C−1)!·C ∧ c = C+1}
17   }
18 // {p = (c−1)! ∧ 0 < c ∧ c−1 ≤ n ∧ ¬(c < n)}
19 // {p = n!}
```

This gives us the three verification conditions:

$$(1)\ 0 \le n \wedge p = 1 \wedge c = 1 \longrightarrow p = (c-1)! \wedge 0 < c \wedge c - 1 \le n \qquad vc_4$$
$$(2)\ \exists C.0 < C < n \wedge p = (C-1)! \cdot C \wedge c = C+1 \longrightarrow p = (c-1)! \wedge 0 < c \wedge c - 1 \le n; \qquad vc_6$$
$$(3)\ p = (c-1)! \wedge 0 < c \wedge c - 1 \le n \wedge \neg(c \le n) \longrightarrow p = n! \qquad vc_7$$

We can simplify these according to our rules:

$$(1.1)\ 0 \le n \wedge p = 1 \wedge c = 1 \longrightarrow p = (c-1)!$$
$$(1.2)\ 0 \le n \wedge p = 1 \wedge c = 1 \longrightarrow 0 < c$$
$$(1.3)\ 0 \le n \wedge p = 1 \wedge c = 1 \longrightarrow c - 1 \le n$$
$$(2.1)\ \exists C.0 < C < n \wedge p = (C-1)! \cdot C \wedge c = C+1w \longrightarrow p = (c-1)!$$
$$(2.2)\ \exists C.0 < C < n \wedge p = (C-1)! \cdot C \wedge c = C+1 \longrightarrow 0 < c$$
$$(2.3)\ \exists C.0 < C < n \wedge p = (C-1)! \cdot C \wedge c = C+1 \longrightarrow c - 1 \le n$$
$$(3)\ p = (c-1)! \wedge 0 < c \wedge c - 1 \le n \wedge n > c \longrightarrow p = n!$$

[2] As we will see below, that does not necessarily mean they become small— it just means we can keep them from getting even larger.

Let us prove these:

- (1.1), (1.2) and (1.3) follow easily by substituting $c = 1$ and $p = 1$: (1.1) becomes $1 = 0!$, (1.2) becomes $0 < 1$ and (1.3) $0 \leq n$.

- (3) follows with $c - 1 = n$ (from $c - 1 \leq n$ and $n > c \Longleftrightarrow n \geq c - 1$).

- (2.1) is a bit hard to show. One trick here is to transform $c = C + 1$ into $c - 1 = C$, and then use (9.3) to get the premisses

$$0 < c - 1 < n \wedge p = ((c-1)-1)! \cdot (c-1) \tag{9.6}$$

  Then we can use the definition of the factorial to conclude $p = ((c-1)-1)! \cdot (c-1) = (c-1)!$. Similarly, from (**??**) we can conclude (2.2) (because $0 < c - 1 < c$) and (2.3).

In general, proving from an assumption $\exists x.P(x)$ is awkard, since we do not have much information about that x. In general, we can assume there is some fresh and arbitrary but fixed constant c such that $P(c)$ holds; if we can prove some R from this assumption, such that R does not contain references to c, then R holds. (This c is called a *Skolem constant*, after the logician who invented the technique.)

Here, this means introduce some Skolem constant b and get the assumption $0 < b < n \wedge p = (b-1)! \cdot b \wedge c = b + 1$. From this, we can conclude $p = b!$ by definition of the factorial, which is equivalent to $p = ((b+1)-1)!$, hence with $c = b + 1$ we get $p = b!$. (2.2) and (2.3) are shown similarly.

9.2.2 Finding the Maximum Element

As a second example, consider again the search for the maximum element in a non-empty array. The source looks like this:

```
1  // {0 < n}
2  i= 0;
3  r= 0;
4  while (i!= n) /** inv (∀j. 0≤ j < i ⟶ a[j]≤ a[r])∧ 0≤ i≤ n ∧ 0≤ r< n; */ {
5    if (a[r] < a[i]) {
6      r= i;
7    }
8    else {
9      }
10   i= i+1;
11   }
12 // {(∀j.0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

The strongest postconditions can be calculated as follows;

```
1  // {0 < n}
2  i= 0;
3  // {∃I.(0 < n)[I/i] ∧ i = 0[I/i]}
4  // {0 < n ∧ i = 0}
5  r= 0;
6  // {∃R.(0 < n ∧ i = 0)[R/r] ∧ r = 0[R/r]}
7  // {0 < n ∧ i = 0 ∧ r = 0}
8  while (i< n) /** inv (∀j. 0≤ j< i ⟶ a[j]≤ a[r]) ∧ 0≤ i≤ n ∧ 0≤ r< n; */ {
9    // {∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n}
10   if (a[r] < a[i]) {
11     // {∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n ∧ a[r] < a[i]}
12     r= i;
13     // {∃R.((∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n ∧ a[r] < a[i])[R/r] ∧ r = i[R/r]}
14     // {∃R.(∀j.0 ≤ j < i ⟶ a[j] ≤ a[R]) ∧ 0 ≤ i < n ∧ 0 ≤ R < n ∧ a[R] < a[i] ∧ r = i}
15     }
16   else {
17     // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n ∧ ¬(a[r] < a[i])}
18     }
19   // {∃R.(∀j.0 ≤ j < i ⟶ a[j] ≤ a[R]) ∧ 0 ≤ i < n ∧ 0 ≤ R < n ∧ a[R] < a[i] ∧ r = i
       ∨ (∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n ∧ ¬(a[r] < a[i])}
20   i= i+1;
21   // {∃I.(∃R.(∀j.0 ≤ j < i ⟶ a[j] ≤ a[R]) ∧ 0 ≤ i < n ∧ 0 ≤ R < n ∧ a[R] < a[i] ∧ r = i
       ∨ (∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n ∧ ¬(a[r] < a[i]))[I/i] ∧ i = (i+1)[I/i]}
22   // {∃I.∃R.(∀j.0 ≤ j < I ⟶ a[j] ≤ a[R]) ∧ 0 ≤ I < n ∧ 0 ≤ R < n ∧ a[R] < a[I] ∧ r = I
       ∨ (∀j.0 ≤ j < I ⟶ a[j] ≤ a[r]) ∧ 0 ≤ I < n ∧ 0 ≤ r < n ∧ ¬(a[r] < a[I]) ∧ i = I + 1}
23   }
24 // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ ¬(i < n)}
25 // {(∀j.0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

This gives us the following verification conditions:

(1) $0 < n \wedge i = 0 \wedge r = 0 \longrightarrow (\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n$ vc_3

(2) $(\exists I.((\exists R.(\forall j.0 \leq j < I \longrightarrow a[j] \leq a[R]) \wedge 0 \leq I < n \wedge 0 \leq R < n \wedge a[R] < a[I] \wedge r = I)$
$\vee ((\forall j.0 \leq j < I \longrightarrow a[j] \leq a[r]) \wedge 0 \leq I < n \wedge 0 \leq r < n \wedge a[I] \leq a[r])) \wedge i = I + 1)$
$\longrightarrow (\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n$ vc_{10}

(3) $(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq i \leq n \wedge 0 \leq r < n \wedge \neg(i < n)$
$\longrightarrow (\forall j.0 \leq j < n \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n$ vc_{11}

This looks foreboding, but us simplify (1) and (3) first:

(1.1) $0 < n \wedge i = 0 \wedge r = 0 \longrightarrow (\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r])$
(1.2) $0 < n \wedge i = 0 \wedge r = 0 \longrightarrow 0 \leq r < n$
(3.1) $(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq i \leq n \wedge 0 \leq r < n \wedge n \leq i \longrightarrow (\forall j.0 \leq j < n \longrightarrow a[j] \leq a[r])$
(3.2) $(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq i \leq n \wedge 0 \leq r < n \wedge n \leq i \longrightarrow 0 \leq r < n$

These all prove trivial or very easy along by now well-known lines — (1.1) has an empty premise, (3.1) follows with $n = i$.

However, verification condition (2) cannot be simplified meaningfully because of the topmost existential quantifier. However, this can be rectified as the existential quantifier distributes over the disjunction:

$$\exists x.(P \vee Q) \Longrightarrow (\exists x.P) \vee (\exists x.Q)$$

In Section 8.4, we have already seen how to deal with disjunctions in the conclusion; here, we have disjunctions in the premisses, so how can we simplify these? (**??**) gives us a clue: to show $P \vee Q \longrightarrow R$, we show $P \vee R$ and $Q \vee R$. This gives the following new simplification rules:

(SIMP-9) The scope of the existential quantification can be shrunk: $(\exists x.P \vee Q) \rightsquigarrow (\exists x.P) \vee (\exists x.Q)$

(SIMP-10) Disjunctions in the premisses give rise to a case distinction: $P \vee Q \longrightarrow R \rightsquigarrow P \longrightarrow R, Q \longrightarrow R$

(SIMP-11) Conjunction and disjunction are distributive: $(A_1 \vee A_2) \wedge B \rightsquigarrow (A_1 \wedge B) \vee (A_2 \wedge B)$ and $(A_1 \wedge A_2) \vee B \rightsquigarrow (A_1 \vee B) \wedge (A_2 \vee B)$

The last rule obviously has to applied judiciously. The goal is to get verification conditions into a form $C_1 \vee C_2 \vee \ldots C_n \longrightarrow D_1 \wedge D_2 \wedge \ldots D_n$, where C_i are conjunctions of elementary facts, and D_i are disjunctions; these then can be simplified using rule (SIMP-10).

To show the new simplification at work, here is the simplification of the premisse of (3):

$$\begin{aligned}
&(\exists I.((\exists R.(\forall j.0 \le j < I \longrightarrow a[j] \le a[R]) \wedge 0 \le I < n \wedge 0 \le R < n \wedge a[R] < a[I] \wedge r = I) \\
&\quad \vee ((\forall j.0 \le j < I \longrightarrow a[j] \le a[r]) \wedge 0 \le I < n \wedge 0 \le r < n \wedge a[I] \le a[r])) \wedge i = I+1) \\
\rightsquigarrow &\exists I.((\exists R.(\forall j.0 \le j < I \longrightarrow a[j] \le a[R]) \wedge 0 \le I < n \wedge 0 \le R < n \wedge a[R] < a[I] \wedge r = I) \wedge i = I+1) \\
&\quad \vee ((\forall j.0 \le j < I \longrightarrow a[j] \le a[r]) \wedge 0 \le I < n \wedge 0 \le r < n \wedge a[I] \le a[r] \wedge i = I+1) \\
\rightsquigarrow &(\exists I\, \exists R.(\forall j.0 \le j < I \longrightarrow a[j] \le a[R]) \wedge 0 \le I < n \wedge 0 \le R < n \wedge a[R] < a[I] \wedge r = I \wedge i = I+1) \\
&\quad \vee (\exists I.(\forall j.0 \le j < I \longrightarrow a[j] \le a[r]) \wedge 0 \le I < n \wedge 0 \le r < n \wedge a[I] \le a[r] \wedge i = I+1) \\
\rightsquigarrow &(\exists R.(\forall j.0 \le j < r \longrightarrow a[j] \le a[R]) \wedge 0 \le r < n \wedge 0 \le R < n \wedge a[R] < a[r] \wedge i = r+1) \\
&\quad \vee (\exists I.(\forall j.0 \le j < I \longrightarrow a[j] \le a[r]) \wedge 0 \le I < n \wedge 0 \le r < n \wedge a[I] \le a[r] \wedge i = I+1)
\end{aligned}$$

Using this, we get a VC which has has a disjunction as a premisse and a conjuction as a conclusion, which results in four simpler VCs:

$$\begin{aligned}
&(2.1)\ (\exists R.(\forall j.0 \le j < r \longrightarrow a[j] \le a[R]) \wedge 0 \le r < n \wedge 0 \le R < n \wedge a[R] < a[r] \wedge i = r+1) \longrightarrow (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \\
&(2.2)\ (\exists R.(\forall j.0 \le j < r \longrightarrow a[j] \le a[R]) \wedge 0 \le r < n \wedge 0 \le R < n \wedge a[R] < a[r] \wedge i = r+1) \longrightarrow 0 \le r < n \\
&(2.3)\ (\exists I.(\forall j.0 \le j < I \longrightarrow a[j] \le a[r]) \wedge 0 \le I < n \wedge 0 \le r < n \wedge a[I] \le a[r] \wedge i = I+1) \longrightarrow (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \\
&(2.4)\ (\exists I.(\forall j.0 \le j < I \longrightarrow a[j] \le a[r]) \wedge 0 \le I < n \wedge 0 \le r < n \wedge a[I] \le a[r] \wedge i = I+1) \longrightarrow 0 \le r < n
\end{aligned}$$

Here, (2.2) and (2.4) are proven trivially. For (2.1) and (2.3), we first replace the quantifiers with a skolem constant c_1 and c_2:

$$\begin{aligned}
&(2.1.1)\ (\forall j.0 \le j < r \longrightarrow a[j] \le a[c_1]) \wedge 0 \le r < n \wedge 0 \le c_1 < n \wedge a[c_1] < a[r] \wedge i = r+1 \longrightarrow (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \\
&(2.3.1)\ (\forall j.0 \le j < c_2 \longrightarrow a[j] \le a[r]) \wedge 0 \le c_2 < n \wedge 0 \le r < n \wedge a[c_2] \le a[r] \wedge i = c_2+1 \longrightarrow (\forall j.0 \le j < i \longrightarrow a[j] \le a[r])
\end{aligned}$$

Now the proof is as before, using (7.5). To show (2.1.1), use transitivity to obtain $\forall j.0 \le j < r \longrightarrow a[j] \le a[r]$ and with $a[r] \le a[r]$ and (7.5), obtain $\forall j.0 \le j < r+1 \longrightarrow a[j] \le a[r]$, and with $i = r+1$ the conclusion follows.

To show (2.3.1), use $\forall j.0 \le j < c_2 \longrightarrow a[j] \le a[r]$ and $a[c_2] \le a[r]$ to obtain $\forall j.0 \le j < c_2+1 \longrightarrow a[j] \le a[r]$ with (7.5), and then with $i = c_2+1$ the conclusion follows.

These two proof obligations form the algorithmic core of our small program; the rest is just book-keeping. It is instructive to compare these two to the corresponding simplified proof obligations from going backward (from page 68):

$$\begin{aligned}
&(3.1.1)\ (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le i < n \wedge 0 \le r < n \wedge a[r] < a[i] \longrightarrow (\forall j.0 \le j < i+1 \longrightarrow a[j] \le a[i]) \\
&(3.2.1)\ (\forall j.0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le i < n \wedge 0 \le r < n \wedge a[i] \le a[r] \longrightarrow (\forall j.0 \le j < i+1 \longrightarrow a[j] \le a[r])
\end{aligned}$$

Basically (and unsurprisingly), these are logically equivalent. The forward proof obligations have the range $\forall j.0 \leq j < c$ as the premisse, and show the range $\forall j.0 \leq j < i$ with $i = c+1$ for some constant which is the previous value of i, so they go from $i-1$ to i. The backward proof obligations have the range $\forall j.0 \leq j < i$ as the premisse, and the range $\forall j.0 \leq j < i+1$ as the conclusion, so they go from i to $i+1$.

9.3 Conclusions

We have investigated how to calculate correctness starting from the precondition, and going forwards. This needs a slightly more involved assignment rule, and results in more awkward verification conditions. The calculated approximate strongest postcondition is a symbolic representation of the program.

The advantages of forward calculation are that we can do more simplifications, because at each point in the forward calcution, we know all there is to know about the program up to this point.

This is not quite true: a while-loop always loses information which is not specified in the invariant. We should expand on this point, with an example etc; further, it is not specific to forward calculation.

Chapter 10

Functions and Procedures

10.1 Introduction

Notes:

- We talk about functions with side-effects here, not purely referential functions as in Haskell. (Those give easier proof rules, but make a more complicated semantic model, as we have to model higher-order functions.)
- Procedures are just functions of return type `void`.

Why functions?

- Functions are the smallest modularity concept (and the only in C).
- Functions model *state encapsulation* (a key concept in object-oriented languages) on a basic level.

To handle functions, we need four ingredients:

1. an extension of our language so we can declare and define functions,
2. a semantics of function definitions,
3. another extension of our language so we can specify how functions should behave, together with rules of how to prove them,
4. a semantics for function calls, and proof rules for function calls.

10.2 Function Definitions and their Semantics

Where previously we merely had statements, we now have function definitions. These consist of a header, which specificies the return type and parameter types, and the body, a block comprising declarations and the statement (most of the time, a sequence or compound statement).

We begin by extending our language from page 51:

$$\begin{aligned}
\textbf{FunDef} &::= \textbf{FunHeader FunSpec}^{+}\ \textbf{Blk}\\
\textbf{FunHeader} &::= \textbf{Type Idt}(\textbf{Decl}^{*})\\
\textbf{Decl} &::= \textbf{Type Idt}\\
\textbf{Blk} &::= \{\textbf{Decl}^{*}\ \textbf{Stmt}\}\\
\textbf{Type} &::= \textbf{char} \mid \textbf{int} \mid \textbf{Struct} \mid \textbf{Array}\\
\textbf{Struct} &::= \textbf{struct}\ \textbf{Idt}^{?}\ \{\textbf{Decl}^{+}\}\\
\textbf{Array} &::= \textbf{Type Idt}[\textbf{Aexp}]
\end{aligned}$$

The major addition is that statements now include the return statement which will cause some head-ache. A function can optionally return a value; the type of the return value must be compatible to the declared return type, obviously; if (and only if) the return type is **void**, the argument of the return statement may be omitted.[1]

$$\begin{aligned}
\textbf{Stmt}\qquad s ::= {} & l = e \mid c_1; c_2 \mid \{\} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2\\
& \mid \textbf{while}\ (b)\ /{**}\ \textbf{inv}\ P\ {*}/\ c \mid /{**}\ \{P\}\ {*}/\\
& \mid \textbf{return}\ a^{?}
\end{aligned}$$

The return statement breaks the sequential control flow; after a `return`-statement the control flow returns to the calling function. In other words, in the following short fragment

```
x= 3; return x; x= x+7;
```

the second assignment statement is never reached. We need to extend the semantics to handle this by introducing a return state as well along the normal sequential state.

$$\begin{aligned}
[\![.]\!]_{\mathscr{C}} &: \textbf{Stmt} \to \Sigma \rightharpoonup (\Sigma \cup \Sigma \times \mathbf{V}_U)\\
\mathbf{V}_U &\stackrel{def}{=} \mathbf{V} + \{*\}
\end{aligned}$$

So, a statement either has a next state as usual, or a *return state* along with a return value (a tuple (σ, v) where v is either a normal value (a number or character) or the unit value $*$ (when a function of type **void**returns no value).

Here is how the denotation of two functions composes:

$$\begin{aligned}
g \circ_S f = {} & \{(\sigma, \rho') \mid (\sigma, \sigma') \in f \wedge (\sigma', \rho') \in g\}\\
& \cup \{(\sigma, (\sigma', v)) \mid (\sigma, (\sigma', v)) \in f\}
\end{aligned}$$

The following two equations hold for this composition, and demonstrate how it works:

$$f(\sigma) = \sigma' \Longrightarrow g \circ_S f(\sigma) = g(\sigma') \tag{10.1}$$

$$f(\sigma) = (\sigma', v) \Longrightarrow g \circ_S f(\sigma) = (\sigma', v) \tag{10.2}$$

[1] We will not go into the specifics of how to specify well-typedness here, as this is not really the focus of these notes. There is a wealth of literature on static analysis and compiler construction which explains this.

If f ends in the normal sequential state $f(\sigma) = \sigma'$, then the value of the composition is whatever g returns for that state, either a normal sequential state or a tuple of return value and state. However, if f ends in return value and state, $f(\sigma) = (\sigma', v)$, then that is the value of the composition and g is not called at all.[2]

The denotional semantics given in Figure 10.1 uses this compositions throughout. The return state is only used, obviously, for the return statement. Note that in the definition of the appromixation functional for the while statement a return also terminates the loop.

We do not give any semantics for the function declarations, as they do not change the state (when we declare a variable, it is not inserted into the state — that only happens with the first assignment). Moreover, we are only interested in the return state of the function body, assuming the function body always ends in a return state. (To that end, we have to make sure by static analysis that the function body ends in a return statement — otherwise, for functions of non-void return type the return value would be undefined. For functions of void return type, we could have a "fall-through", *i.e.* the state *after* the last statement becomes the return statement, but this would needlessly complicate our exposition here.[3]

$$[\![.]\!]_{blk} : \mathbf{Blk} \to \Sigma \rightharpoonup (\Sigma \times V_U)$$
$$[\![d\ s]\!]_{blk} \stackrel{def}{=} \{(\sigma,(\sigma',v)) \mid (\sigma,(\sigma',v)) \in [\![s]\!]_{\mathscr{C}}\}$$

For a function definition, the parameters of the function become parameters of the semantics of the block.

$$[\![.]\!]_{fd} : \mathbf{FunDef} \to \mathbf{V}^n \rightharpoonup \Sigma \rightharpoonup \Sigma \times \mathbf{V}_U$$
$$[\![f(t_1\ p_1, t_2\ p_2, \ldots, t_n\ p_n)\ b]\!]_{fd} = \{((v_1,\ldots,v_n),\sigma,(\sigma',v)) \mid (\sigma[p_i \mapsto v_i]_{i=1,\ldots,n},(\sigma',v)) \in [\![b]\!]_{blk}\}$$

What does this mean? It means that the function parameters are essentially local variables which are initialised with the actual parameters when the function is called. TO DO.

Actually, this does not work. If we just use the *names* of the parameters (and local variables), then recursive function calls overwrite the values; we really *do* need to model local variables. This also means we have to (statically) map names to locations (in other words, $\times$ is not a valid location anymore). Interestingly enough, for the Floyd-Hoare rules this is of no concern.

10.3 Function Specifications and their Semantics

Function specifications are essentially Floyd-Hoare-tripels, with some additional syntactic sugar. This is the *behavioural specification* style originally proposed by Betrand Meyer for Eiffel for Eiffel[4], and used by languages such as JML, OCL, or ACSL.

The function specification is written between the function parameters and the function body (because it restricts the function parameters). It contains two specifications: a pre-specification and a post-specification.

$$\mathbf{FunSpec} ::= \texttt{/**}\ \texttt{pre}\ \mathbf{Assn}\ \texttt{post}\ \mathbf{Assn}\ \texttt{*/}$$

Compared to Floyd-Hoare-tripels as we have been using up to now, we have two additional constructs in the boolean expressions:

[2] Friends of the Haskell programming language may spot this is the compositional behaviour of the Maybe or Either monads.
[3] In fact, our tool adds a return statment to such functions as the last statement of the body if it is not there.
[4] I think...

$$
\begin{aligned}
[\![x = e]\!]_{\mathscr{C}} &= \{(\sigma, \sigma[l \mapsto a]) \mid (\sigma, l) \in [\![x]\!]_{\mathscr{L}}, (\sigma, a) \in [\![e]\!]_{\mathscr{A}}\} \\
[\![c_1; c_2]\!]_{\mathscr{C}} &= [\![c_2]\!]_{\mathscr{C}} \circ_S [\![c_1]\!]_{\mathscr{C}} \\
[\![\{\,\}]\!]_{\mathscr{C}} &= \mathbf{Id}_\Sigma \\
[\![\textbf{if } (b)\ c_0 \textbf{ else } c_1]\!]_{\mathscr{C}} &= \{(\sigma, \rho') \mid (\sigma, \mathit{true}) \in [\![b]\!]_{\mathscr{B}} \wedge (\sigma, \rho') \in [\![c_0]\!]_{\mathscr{C}}\} \\
&\quad \cup \{(\sigma, \rho') \mid (\sigma, \mathit{false}) \in [\![b]\!]_{\mathscr{B}} \wedge (\sigma, \rho') \in [\![c_1]\!]_{\mathscr{C}}\} \\
&\quad \text{mit } \rho' \in \Sigma \cup \Sigma \times \mathbf{V}_U \\
[\![\textbf{return } e]\!]_{\mathscr{C}} &= \{(\sigma, (\sigma, a)) \mid (\sigma, a) \in [\![e]\!]_{\mathscr{A}}\} \\
[\![\textbf{return}]\!]_{\mathscr{C}} &= \{(\sigma, (\sigma, *))\} \\
\\
[\![\textbf{while } (b)\ c]\!]_{\mathscr{C}} &= \mathit{fix}(\Gamma) \\
\Gamma(\psi) &\stackrel{def}{=} \{(\sigma, \rho') \mid (\sigma, \mathit{true}) \in [\![b]\!]_{\mathscr{B}} \wedge (\sigma, \rho') \in \psi \circ_S [\![c]\!]_{\mathscr{C}}\} \\
&\quad \cup \{(\sigma, \sigma) \mid (\sigma, \mathit{false}) \in [\![b]\!]_{\mathscr{B}}\}
\end{aligned}
$$

Figure 10.1: Denotational Semantics of Statements

```
int fac(int n)
/** pre  0 ≤ n;
    post \result == n!;
 */
{
 int p;
 int c;

 p= 1;
 c= 1;
 while (c<= n) /** inv p == (c- 1)! ∧ c≤ n+1 ∧ 0 < c; */ {
    p= p*c;
    c= c+1;
    }
 return p;
}
```

Figure 10.2: Example: the good old factorial

- We can use the constant **\result**to refer to the return value of the function. The type of **\result**is given by the return type of the function; obviously, for functions of void return type the constant cannot be used. Further, **\result**can only be used in the post-specification of the function.
- In the post-specification we can also refer to the initial value of a variable, *i.e.* the value at the start of the function. This is written as `l@pre` (for an l-expression `l`).

The second point deserves some explanation. In C (and most other similar languages such as Java), function parameters are in fact *local variables* which can be modified, and which are initialised with the value passed to the function. Consider the following example:

```
1  int foo(int x)
2  /** post   \result  == x+1; */
3  {
4    x= 0;
5    return 1;
6  }
```

Here, there is difference wether the `x` in the post-condition refers to the parameter `x` at the start of the function or at the end. However, the change to `x` in line 5 is not visible outside the function, so the function should *not* satisfy the given postcondition, as it always returns 1 and not whatever it is given plus 1. Thus, when we write `x` in the postcondition for a parameter `x` we really mean `x@pre`, or more precisely the value we pass to the function.

Thus, the denotational semantics of a specification (written as $[\![-]\!]_{\mathscr{B}sp}$) takes as arguments *two* states, the state when entering the function and the state when the function returns (where $[\![-]\!]_{\mathscr{A}sp}$ is an auxiliary function which evaluates the arithmetic expressions in the specification):

$$[\![-]\!]_{\mathscr{B}sp} : \mathbf{Env} \to \mathbf{Assn} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbb{B}$$
$$[\![-]\!]_{\mathscr{A}sp} : \mathbf{Env} \to \mathbf{Aexpv} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{V}$$

Figure 10.4 shows the denotational semantics for specifications. One point to note is that the semantics are *not* parameterized, that is the parameters of the function do not become parameters of the specification. This is because the function parameters are also local variables, and is discussed above we refer to their initial value.

We can now define what it means for a function definition to be *correct* — it satisfies the specification annotated to the definition.

Definition 10.1 (Correctness of Function Definition) *A funcion definition fd satisfies a function specification* ***pre*** *p* ***post*** *q in a context Γ*

$$\textbf{\textit{pre}}\ p\ \textbf{\textit{post}}\ q \models fd \iff \forall v_1, \ldots, v_n.\ [\![fd]\!]_{fd}\ \Gamma\ v_1 \ldots v_n \in [\![\textbf{\textit{pre}}\ p\ \textbf{\textit{post}}\ q]\!]_{\mathscr{B}sp}\ \Gamma \tag{10.3}$$

If the function fd is annotated with the specification, we say the function is correct.

Note that correctness is always with respect to a context Γ. This is because functions may call other functions (we will deal with that in Section 10.5), so correctness of one function always depends on correctness of other functions.

Now, a program in C is just a collection of functions. If we have a set of n functions $f_1, \ldots, f_n$ with specifications $sp_1, \ldots, sp_n$ annotated, then the correctness of the whole program is the correctness of each f_i within the context $\Gamma = \{(f_1, sp_1), \ldots, (f_n, sp_n)\}$ — that is, we assume all functions satisfy their specification, and then prove that each does. This seems like circular reasoning, but it really is not; we come back to that question later in Section 10.5.

10.4 Proof Rules for Function Specifications

We have seen how to write down function definitions, along with specifications, and we have defined their semantics. Following along the path of the Floyd-Hoare logic of the previous chapters, we now need to extend the proof rules such that we can *prove* semantic correctness according to Definition 10.1 syntactically.

The major problem here is that our new semantics $[\![-]\!]_{\mathscr{C}}$ from Figure 10.1 uses $\Sigma \rightharpoonup (\Sigma \cup \Sigma \times \mathbf{V}_U)$ as semantic domain, whereas the Hoare triples from Section 5.2.2 only use $\Sigma \rightharpoonup \Sigma$, so the first step is to accomodate the return state into the tripels. This is achieved by extending the Hoare tripels: the return state gets its own post-specification.

An *extended Floyd-Hoare tripel* $\{P\}\, c\, \{Q \mid Q_R\}$ is given by a pre-specification P, a program statement c, a post-specification Q (so far so good), and a return specification Q_R. We can extend the notion of partial and total correctness as follows:

Definition 10.2 (Partial and Total Correctness) *Given an extended Floyd-Hoare tripel $\{P\}\, c\, \{Q \mid Q_R\}$, we say c is partially or totally correct, written $\models \{P\}\, c\, \{Q \mid Q_R\}$ iff:*

$$
\begin{aligned}
&\Gamma \models \{P\}\, c\, \{Q \mid Q_R\} \iff \\
&\quad \forall \sigma.\, (\sigma, \mathit{true}) \in [\![P]\!]_{\mathscr{B}}\, \Gamma \;\wedge\; (\exists \sigma'.\, (\sigma, \sigma') \in [\![c]\!]_{\mathscr{C}} \Longrightarrow ((\sigma, (\sigma', *)), \mathit{true}) \in [\![Q]\!]_{\mathscr{B}sp}\, \Gamma) \\
&\qquad\qquad\qquad\qquad \vee \\
&\qquad\qquad\qquad\qquad (\exists \sigma', v.\, (\sigma, (\sigma', v)) \in [\![c]\!]_{\mathscr{C}} \Longrightarrow ((\sigma, (\sigma', v)), \mathit{true}) \in [\![Q_R]\!]_{\mathscr{B}sp}\, \Gamma) \\
&\Gamma \models [P]\, c\, [Q \mid Q_R] \iff \\
&\quad \forall \sigma.\, (\sigma, \mathit{true}) \in [\![P]\!]_{\mathscr{B}}\, \Gamma \Longrightarrow (\exists \sigma'.\, (\sigma, \sigma') \in [\![c]\!]_{\mathscr{C}} \wedge ((\sigma, (\sigma', *)), \mathit{true}) \in [\![Q]\!]_{\mathscr{B}sp}\, \Gamma) \\
&\qquad\qquad\qquad\qquad \vee \\
&\qquad\qquad\qquad\qquad (\exists \sigma', v.\, (\sigma, (\sigma', v)) \in [\![c]\!]_{\mathscr{C}} \wedge ((\sigma, (\sigma', v)), \mathit{true}) \in [\![Q_R]\!]_{\mathscr{B}sp}\, \Gamma)
\end{aligned}
$$

It seems only logical (and indeed it is) that the proof rules should also extend the post-specification with a return specification. The return specification does not do match; it is only passed around, because most of the statements (except for the return statement) return a sequential state rather than a return state. Subsequently, only the return statement makes use of the new return specification:

$$
\frac{}{\Gamma \vdash \{\Gamma\}\, Q\, \{\textbf{return} \mid P\}} \qquad \frac{}{\Gamma \vdash \{\Gamma\}\, Q[e/\textbf{\textbackslash result}]\, \{\textbf{return}\ e \mid P\}}
$$

There are couple of side conditions here: a return without argument is only allowed in the body of a function of return type **void**, but on the other hand, in a function of non-void type *all* return statements have to be provided with an expression of the appropriate type. (Again, these are static conditions which we assume to hold within the calculus, as we can check them before.)

The rules for the other statements are given comprehensively in Figure 10.5. The "top-level rule" kick-starting the correctness proof of a function body is the following and deserves some explanation:

$$
\frac{\Gamma \vdash \{\Gamma\}\, P \wedge x_i = x_i\, @\textbf{pre}\, \{c \mid \mathit{false}\}}{\Gamma \vdash f(x_1, \ldots, x_n) /{*}{*}\ \mathsf{pre}\ P\ \mathsf{post}\ Q\ {*}/\ \{ds\ c\}} \tag{10.4}
$$

```
int fac(int x)
/** pre   0 ≤ x;
    post   \result == x!; */
{
 int r = 0;

 if (x == 0) {
   return 1;
   }
 return r* x;
}
```

Figure 10.3: Shades of Haskell: the recursive factorial function

$$
\begin{aligned}
[\![.]\!]_{\mathscr{B}sp} &: \mathbf{Env} \to \mathbf{Assn} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbb{B} \\
[\![.]\!]_{\mathscr{A}sp} &: \mathbf{Env} \to \mathbf{Aexpv} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{V} \\
[\![!b]\!]_{\mathscr{B}sp}\,\Gamma &= \{((\sigma,(\sigma',v)),true) \mid ((\sigma,(\sigma',v)),false) \in [\![b]\!]_{\mathscr{B}sp}\Gamma\} \\
&\quad \cup \{((\sigma,(\sigma',v)),false) \mid ((\sigma,(\sigma',v)),true) \in [\![b]\!]_{\mathscr{B}sp}\Gamma\} \\
[\![x]\!]_{\mathscr{A}sp}\,\Gamma &= \{((\sigma,(\sigma',v)),\sigma'(x))\} \\
&\dots \\
[\![\mathsf{e@pre}]\!]_{\mathscr{B}sp}\,\Gamma &= \{((\sigma,(\sigma',v)),b) \mid (\sigma,b) \in [\![e]\!]_{\mathscr{B}}\,\Gamma\} \\
[\![\mathsf{e@pre}]\!]_{\mathscr{A}sp}\,\Gamma &= \{((\sigma,(\sigma',v)),a) \mid (\sigma,a) \in [\![e]\!]_{\mathscr{A}}\,\Gamma\} \\
[\![\,\mathsf{\backslash result}\,]\!]_{\mathscr{A}sp}\,\Gamma &= \{((\sigma,(\sigma',v)),v)\}
\end{aligned}
$$

$$
[\![\mathbf{pre}\ p\ \mathbf{post}\ q]\!]_{\mathscr{B}sp}\,\Gamma = \{(\sigma,(\sigma',v)) \mid (\sigma,true) \in [\![p]\!]_{\mathscr{B}}\,\Gamma \wedge ((\sigma,(\sigma',v)),true) \in [\![q]\!]_{\mathscr{B}sp}\,\Gamma\}
$$

Figure 10.4: Denotational semantics of function specifications

As mentioned above, when we refer to a parameter x in the post-specification, we mean the initial value of x, so we replace all occurences x_i in Q with x_i **@pre**. To enforce the meaning that x_i **@pre** is the value of x_i when calling the function, we add the equations $x_i = x_i$ **@pre** to the precondition.

Further, the initial sequential precondition is *false*. This ensures that the control flow of c never reaches the end of the statement without encountering a return statement (if it were, c would not be correct).

To see the new calculus in action, consider the following simple example (a corrected version from above):

```
1  int foo(int x)
2  /** post \result > x; */
3  {
4    // {x == x@pre}
5    // {x@pre < x+1}
6    x= x+1;
7    // {x@pre < x}
8    return x;
9    // {false | x < \result}
10 }
```

We have only writtn down the return specification in line 7, where it is actually used, and because the return specification never changes.[5] Line 9 is slightly confusing— this is the post specification **\result** $< x$, where **\result** is replaced with x, and x and is replaced with x **@pre**, as by the return rule. The x **@pre** is in fact a logical variable, and not changed by assiging to x (Line 6). It remains to prove that $x == x$ **@pre** implies x **@pre** $< x+1$ which is on the trivial side of easy proofs.

The rules in Figure 10.5 are neither suitable for forward nor for backward reasoning. Even though the assignment rule is given in the backward style, the top-level rule (10.4) does not have an open precondition in the assumption, but it is trivial to amend this by weakening. With that, we can derive a suitable definition of calculating the approximate weakest pre-condition (which also needs two different postconditions). The relevant clauses are:

$$\begin{aligned}
\mathrm{awp}(\Gamma, f(x_1,\ldots,x_n)/{*}{*}\ \mathsf{pre}\ P\ \mathsf{post}\ Q\ {*}/\ \{ds\ blk\}) &\stackrel{def}{=} \mathrm{awp}(\Gamma', blk, false, Q[x_i\ \mathbf{@pre}/x_i]) \\
\mathrm{awp}(\Gamma, \mathbf{return}\ e, Q, Q_R) &\stackrel{def}{=} Q_R[e/\mathbf{\backslash result}] \\
\mathrm{awp}(\Gamma, \mathbf{return}, Q, Q_R) &\stackrel{def}{=} Q_R
\end{aligned}$$

$$\begin{aligned}
\mathrm{wvc}(\Gamma, f(x_1,\ldots,x_n)/{*}{*}\ \mathsf{pre}\ P\ \mathsf{post}\ Q\ {*}/\ \{ds\ blk\}) &\stackrel{def}{=} \{(\Gamma \wedge P) \Longrightarrow P'[x_i/x_i\ \mathbf{@pre}]\} \\
&\quad \cup\ \mathrm{wvc}(\Gamma', blk, false, Q[x_i\ \mathbf{@pre}/x_i]) \\
\Gamma' &\stackrel{def}{=} \Gamma[f \mapsto \forall x_1,\ldots,x_n.\,(P,Q)] \\
P' &\stackrel{def}{=} \mathrm{awp}(\Gamma', blk, false, Q[x_i\ \mathbf{@pre}/x_i]) \\
\mathrm{wvc}(\Gamma, \mathbf{return}\ e, Q, Q_R) &\stackrel{def}{=} \emptyset
\end{aligned}$$

Now consider the following version of the factorial program:

[5] It could be weakened, just like the regular post-condition (cf. Figure 10.5), but there is no reason to do so.

```
1  int fac(int n)
2  /** pre 0 ≤ n;
3      post \result = n!; */
4  {
5   int p, c;
6
7   p= 1;
8   c= 1;
9   while (1) /** inv p = (c- 1)! ∧ 0< c; */ {
10     p= p*c;
11     if (c == n) {
12       return p;
13       }
14     c= c+1;
15     }
16 }
```

This is not recursive as in Figure 10.3 above, but it uses **return** to end a non-terminating while loop (and break the sequential control flow). Using the definition of *awp* above, we can derive the weakest preconditions:

```
1  int fac(int n)
2  /** pre 0 ≤ n;
3      post \result = n!; */
4  {
5   int p, c;
6   // {1 = (1−1)! ∧ 0 < 1 ∧ n = n@pre}
7   p= 1;
8   // {p = (1−1)! ∧ 0 < 1 ∧ n = n@pre}
9   c= 1;
10  // {p = (c−1)! ∧ 0 < c ∧ n = n@pre}
11  while (1) /** inv p = (c- 1)! ∧ 0< c ∧ n = n@pre; */ {
12    // {(c = n ∧ p·c = n@pre!) ∨ (c ≠ n ∧ p·c = ((c+1)−1)! ∧ 0 < c+1)}
13    p= p*c;
14    // {(c = n ∧ p = n@pre!) ∨ (c ≠ n ∧ p = ((c+1)−1)! ∧ 0 < c+1)}
15    if (c == n) {
16      // {p = n@pre!}
17      return p;
18      }
19    else {
20      // {p = ((c+1)−1)! ∧ 0 < c+1}
21      }
22    // {p = ((c+1)−1)! ∧ 0 < c+1}
23    c= c+1;
24    // {p = (c−1)! ∧ 0 < c}
25    }
26  // {false}
27 }
```

The precondition results in the following verification conditions:

(1) $0 \le n \wedge n = n\,@\mathbf{pre} \longrightarrow 1 = (1-1)! \wedge 0 < 1 \wedge n = n\,@\mathbf{pre}$
(2) $p = (c-1)! \wedge 0 < c \wedge n = n\,@\mathbf{pre} \wedge \mathit{true} \longrightarrow (c = n \wedge p \cdot c = n\,@\mathbf{pre}!) \vee (c \neq n \wedge p \cdot c = ((c+1)-1)! \wedge 0 < c+1)$
(3) $p = (c-1)! \wedge 0 < c \wedge n = n\,@\mathbf{pre} \wedge \neg \mathit{true} \longrightarrow \mathit{false}$

When we simplify these according to our rules, we get

(1.1)	$0 \leq n \wedge n = n\,@\mathbf{pre} \longrightarrow 1 = 0!$	Follows by definition of !
(1.2)	$0 \leq n \wedge n = n\,@\mathbf{pre} \longrightarrow 0 < 1$	Trivial.
(1.3)	$0 \leq n \wedge n = n\,@\mathbf{pre} \longrightarrow n = n\,@\mathbf{pre}$	Trivial.
(2.1)	$p = (c-1)! \wedge 0 < c \wedge n = n\,@\mathbf{pre} \wedge c = n \longrightarrow p \cdot c = n\,@\mathbf{pre}!$	
(2.2)	$p = (c-1)! \wedge 0 < c \wedge n = n\,@\mathbf{pre} \wedge c \neq n \longrightarrow p \cdot c = c!$	Follows by definition of !
(2.3)	$p = (c-1)! \wedge 0 < c \wedge n = n\,@\mathbf{pre} \wedge c \neq n \longrightarrow 0 < c+1$	Follows with $0 < c < +1$
(3.1)	$\mathit{false} \longrightarrow \mathit{false}$	Trivial.

Only (2.1) needs some explanation: from $p = (c-1)!$, we get $p \cdot c = n!$, and with $n = n\,@\mathbf{pre}$ we get the conclusion. This is the reason $n = n\,@\mathbf{pre}$ is part of the invariant; we must explicitly specify that n does not change during the course of the while-loop.

Note how the invariant on the other did not need to specify $c - 1 \leq n$ as before; this is because at the exit point of the loop we actually have $c = n$, and do not need to conclude $c - 1 = n$ from $c - 1 \geq n$ and $c - 1 \leq n$. On the other hand, the regular postcondition of the loop is *false*, so it first looks an impossible ask to prove that (verification condition (3), but because the loop never terminates (loop condition *true*) it holds rather trivially.

10.5 Function Calls

(To come.)

$$\frac{}{\Gamma \vdash \{\Gamma\}\, P\, \{\{\} \mid P\}} \qquad \frac{\Gamma \vdash \{\Gamma\}\, P\, \{c_1 \mid R\} \quad \Gamma \vdash \{\Gamma\}\, R\, \{c_2 \mid Q\}}{\Gamma \vdash \{\Gamma\}\, P\, \{c_1; c_2 \mid Q\}}$$

$$\frac{}{\Gamma \vdash \{\Gamma\}\, Q[e/x]\, \{l = e \mid Q\}} \qquad \frac{\Gamma \vdash \{\Gamma\}\, P \wedge b\, \{c \mid P\}}{\Gamma \vdash \{\Gamma\}\, P\, \{\textbf{while}\ (b)\ c \mid P \wedge \neg b\}}$$

$$\frac{\Gamma \vdash \{\Gamma\}\, P \wedge b\, \{c_1 \mid Q\} \quad \Gamma \vdash \{\Gamma\}\, P \wedge \neg b\, \{c_2 \mid Q\}}{\Gamma \vdash \{\Gamma\}\, P\, \{\textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \mid Q\}}$$

$$\frac{(\Gamma \wedge P) \longrightarrow P' \quad \Gamma \vdash \{\Gamma\}\, P'\, \{c \mid Q'\} \quad (\Gamma \wedge Q') \longrightarrow Q \quad (\Gamma \wedge R') \longrightarrow R}{\Gamma \vdash \{\Gamma\}\, P\, \{c \mid Q\}}$$

$$\frac{}{\Gamma \vdash \{\Gamma\}\, Q\, \{\textbf{return} \mid P\}} \qquad \frac{}{\Gamma \vdash \{\Gamma\}\, Q[e/\backslash\textbf{result}]\, \{\textbf{return}\ e \mid P\}}$$

$$\frac{(\Gamma \wedge P) \Longrightarrow P'[x_i/x_i\,@\textbf{pre}] \quad \Gamma \vdash \{\Gamma\}\, P'\, \{c \mid \mathit{false}\}}{\Gamma \vdash f(x_1, \ldots, x_n) /{*}{*}\ \mathsf{pre}\ P\ \mathsf{post}\ Q\ {*}/\ \{ds\ c\}}$$

Figure 10.5: The extended Floyd-Hoare calculus

Chapter 11

Memory Models and References

11.1 Introduction

In our language, functions and procedures can only have global effects (*i.e.* side effects on global variables). It is not possible to change a local variable using a function call. Moreover, since our values are only basic values (integers), we cannot even pass an array or structure into a function.

In C and most other languages, this problem is solved by passing around *references* to local variables and structured values. In C specifically, references are weakly typed (pointers a type τ are only compatible with pointers to a compatible type, but there is a type **void** $*$ which is compatible with a pointer to any type). Moreover, pointers and arrays are conflated using *pointer arithmetic*. This assumes that our memory is addressed in integers counting bytes. An object is a contagious part of memory large enough to hold the representation of an inhabitant of a given type. Now, given an expression p of type pointer to τ, the expression $p+n$ for an integer n denotes the adress p plus n times the size of an object of type τ in bytes, and $*(p+n)$ is the content of that adress. If we think of p as an array, $*(p+n)$ is the same as $p[n]$, accessing the array at index n.[1]

Thus, in C an array is just a pointer with static allocation of enough memory to hold a number of objects. References to arrays are just normal references, and no index checking is done; this results in fast, but error-prone, code. Java is different here; although arrays are objects (but no class instances[2]), and the JVM keeps track of the length of the array, and checks the bounds of array access at runtime. This makes array access more costly, but applications more stable (safe).

11.2 Extending C0 with References

Syntactically, references manifest themselves in two additional operators: the address operator &, which takes the address of an l-expression, and the dereferencing operator $*$, which turns an expression denoting an adress into an l-expression. Thus, we extend the syntax for expressions and l-expressions from page 51 as follows:

[1] This is exactly as the standard defines array access. It also states that addition is commutative, leading to the amusing chain of equations $a[i] = *(a+i) = *(i+a) = i[a]$; and indeed, one can write array access in that fashion, and any standard-compliant compiler will accept this, most while complaining loudly.

[2] https://docs.oracle.com/javase/specs/jls/se8/html/jls-10.html

Lexp	$l ::= \mathbf{Idt} \mid l[a] \mid l.\mathbf{Idt} \mid *a$
Aexp	$a ::= \mathbf{Z} \mid \mathbf{C} \mid \mathbf{Lexp} \mid \&l \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1/a_2$
Bexp	$b ::= \mathbf{1} \mid \mathbf{0} \mid a_1 == a_2 \mid a_1 < a_2 \mid !b \mid b_1 \,\&\&\, b_2 \mid b_1 \,\|\| \, b_2$
Exp	$e := \mathbf{Aexp} \mid \mathbf{Bexp}$

We also need to extend the syntax for types by a type of pointer to t for any type t. This is written just like the derefencing (in C, the syntax of types is as close as possible to expressions of that type[3]):

Type	$t ::= \mathbf{char} \mid \mathbf{int} \mid *t \mid \mathbf{struct}\ \mathbf{Idt}^? \ \{\mathbf{Decl}^+\} \mid t\ \mathbf{Idt}[a]$

To model the semantics of references, we need to change the model — and unfortunately, quite severely. To see why, consider the following example:

```
int x;
int *p;

p= &x;
x= 0;
// (1)
*p= 7;
// {x > 0}
```

Here, at point (1) there are two ways to refer to the location where `x` is stored, namely `x` and `*p`. This is called *aliasing*. For example, let us assume that we want to show that the postcondition $x = 7$. It is particularly troubling when calculating the verification conditions backwards. The backwards Hoare rule would let us deduce $(x > 0)[7/*p] = (x > 0)$. But that assertion is wrong, because the reference p is pointing to x and changes its value under the hood, so to speak.

When we define a semantics for locations we have to adress this problem. Recall that in Definition 7.1, we defined the state as a partial map from locations **Loc** to values **V**, where locatons were **Loc** ::= **Idt** | **Loc**[$\mathbb{Z}$] | **Loc**.**Idt** whereas values where just numbers. This is not enough anymore; firstly, we need values to be locations as well — to model an assignment like `p= &x` — and secondly, locations need to be flexible enough to handle effects like aliasing.

In order to develop a semantic model for referencing, we need to look at *memory models*.

11.3 Memory Models

The C standard describes a fairly low-level memory model, with addresses being counted in bytes. This is adequate when talking about compilers — which are the prime applications of a language standard — but not so when considering verification.

When describing memory models, the question is always how given declarations like the following are represented in the memory:

[3] A questionable design decision which leads to types which take some practice to parse.

```
int a;
struct {
  int x;
  int y[3]} b[2];
int c[3];
```

11.4 Axiomatic State Model

We model the state with two abstract datatypes, locations **Loc** and the state Σ.

The locations are a datatype **Loc** such that there are indeterminate locations (corresponding to the base locations l,m,n in Figure 11.3 above), and operations for array access and field selection (for **struct**), called *off* and *fld* respectively. These are charactized as follows:

$$\begin{array}{ll} \mathit{off} : \mathbf{Loc} \to \mathbf{Z} \to \mathbf{Loc} & \mathit{fld} : \mathbf{Loc} \to \mathbf{Idt} \to \mathbf{Loc} \\ \\ \mathit{off}(l,0) = l & \mathit{fld}(l,f) \neq l \\ \mathit{off}(\mathit{off}(l,a),b) = \mathit{off}(l,a+b) & \mathit{fld}(l,f) = \mathit{fld}(l,g) \Longrightarrow f = g \\ \mathit{off}(l,a) = l \Longrightarrow a = 0 & \mathit{fld}(l,f) = \mathit{fld}(m,f) \Longrightarrow l = m \\ \mathit{off}(l,a) = \mathit{off}(l,b) \Longrightarrow a = b & f \neq g \Longrightarrow \mathit{fld}(l,f) \neq \mathit{fld}(m,g) \end{array}$$

The state is modelled with the abstract datatype Σ with two operations characterised by four equations:

$$\begin{array}{c} \mathit{read} : \Sigma \to \mathbf{Loc} \rightharpoonup \mathbf{V} \\ \mathit{upd} : \Sigma \to \mathbf{Loc} \to \mathbf{V} \rightharpoonup \Sigma \\ \mathbf{V} \stackrel{\mathit{def}}{=} \mathbb{Z} + \mathbf{Loc} \end{array}$$

$$\mathit{read}(\mathit{upd}(\sigma,l,v),l) = v \tag{11.1}$$
$$l \neq m \Longrightarrow \mathit{read}(\mathit{upd}(\sigma,l,v),m) = \mathit{read}(\sigma,m) \tag{11.2}$$
$$\mathit{upd}(\mathit{upd}(\sigma,l,v),l,w) = \mathit{upd}(\sigma,l,w) \tag{11.3}$$
$$l \neq m \Longrightarrow \mathit{upd}(\mathit{upd}(\sigma,l,v),m,w) = \mathit{upd}(\mathit{upd}(\sigma,m,w),l,v) \tag{11.4}$$

These equations are *complete* characterisations of the state.

Given a state Σ, its *domain* $\mathit{dom}(\Sigma)$ is the set of all locations where it is defined:

$$\mathit{dom}(\sigma) = \{l \mid \exists v.\, \mathit{read}(\sigma,l) = v\} \tag{11.5}$$

There is a one particular state which is empty: nothing can be read from it, its domain is empty.

$$\begin{array}{c} \mathit{empty} : \Sigma \\ \mathit{dom}(\mathit{empty}) = \emptyset \end{array}$$

To model allocation and local variables, we need more operations. *fresh* describes *allocation*, it gives us

a location which nothing has been written into, and *rem* describes deallocation:

$$\begin{aligned} \mathit{fresh} &: \Sigma \to \mathbf{Loc} \\ \mathit{rem} &: \Sigma \to \mathbf{Loc} \to \Sigma \end{aligned}$$

$$\begin{aligned} \mathit{fresh}(\sigma) &\notin \mathit{dom}(\sigma) \\ \mathit{dom}(\mathit{rem}(\sigma,l)) &= \mathit{dom}(\sigma) \setminus \{l\} \\ l \neq m \Longrightarrow \mathit{read}(\mathit{rem}(\sigma,l),m) &= \mathit{read}(\sigma,m) \end{aligned}$$

11.5 Denotational Semantics

With these operations, we can give a denotational semantics for expressions involving expressions (Figure 11.4). The denotational semantics for statements remains largely as it was, except for minor changes for assignment (Figure 11.5).

TO DO.

Missing: definitions of the environment Γ, denotational semantics for declarations.

11.6 Floyd-Hoare Logic with References

11.6.1 Explicit State Predicates

Assertions (**Assn**) are state-dependent predicates, *i.e.* predicates with state variables. If we change the state model, we obviously need to change these too.

We do that by making the read and update operations *explicit* in our state predicates.

$\mathbf{Lexp}_s$	$l ::= \ldots \mid \cancel{*a}$
$\mathbf{Assn}_s$	$b ::= \ldots$
$\mathbf{Aexp}_s$	$a ::= \mathit{read}(S,l) \mid \mathbf{Z} \mid \mathbf{C} \mid l \mid \cancel{\&l} \mid \ldots \mid e\,@\mathbf{pre} \mid \ldots$
State	$S ::= \mathit{StateVar} \mid \mathit{upd}(S,l,e)$

State variables *StateVar* inclde the current statate σ, the pre-state ρ of the current function, and intermediate states $\rho_0, \rho_1, \rho_2, \ldots$.

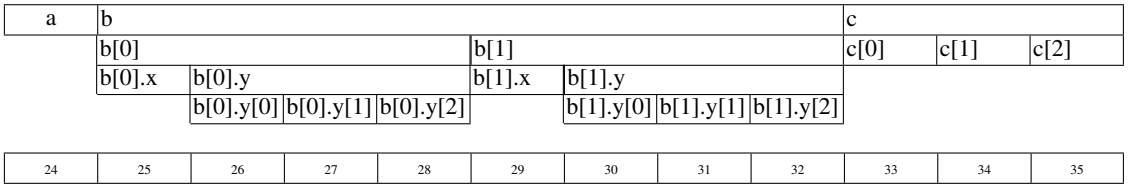

Figure 11.1: Memory modell as implemented by a compiler

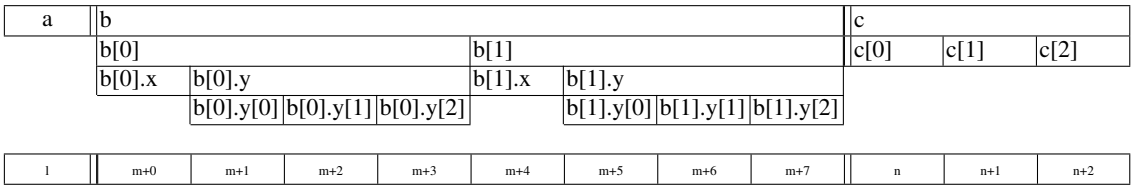

Figure 11.2: Memory modell according to the C standard

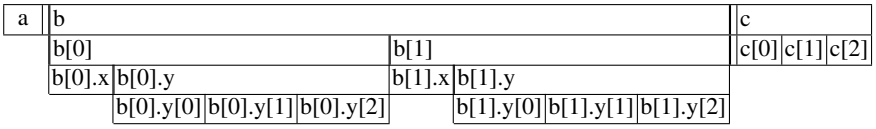

Figure 11.3: Symbolic memory modell

$$[\![-]\!]_{\mathscr{L}} : \mathbf{Env} \to \mathbf{Lexp} \to \Sigma \rightharpoonup \mathbf{Loc}$$

$$\begin{aligned}
[\![x]\!]_{\mathscr{L}}\ \Gamma &= \{(\sigma, \Gamma!x) \mid \sigma \in \Sigma\} \\
[\![e[a]]\!]_{\mathscr{L}}\ \Gamma &= \{(\sigma, \mathit{off}(l,i)) \mid (\sigma,l) \in [\![e]\!]_{\mathscr{L}}\ \Gamma, (\sigma,i) \in [\![a]\!]_{\mathscr{A}}\ \Gamma\} \\
[\![e.f]\!]_{\mathscr{L}}\ \Gamma &= \{(\sigma, \mathit{fld}(e,f)) \mid (\sigma,l) \in [\![\mathit{lexp}]\!]_{\mathscr{L}}\ \Gamma\} \\
[\![*e]\!]_{\mathscr{L}}\ \Gamma &= [\![e]\!]_{\mathscr{A}}\ \Gamma
\end{aligned}$$

$$[\![-]\!]_{\mathscr{A}} : \mathbf{Env} \to \mathbf{Aexp} \to \Sigma \rightharpoonup \mathbf{V}$$

$$\begin{aligned}
[\![n]\!]_{\mathscr{A}}\ \Gamma &= \{(\sigma,n) \mid \sigma \in \Sigma\} \quad \text{for } n \in \mathbb{N} \\
[\![e]\!]_{\mathscr{A}}\ \Gamma &= \{(\sigma, \mathit{read}(\sigma,l)) \mid (\sigma,l) \in [\![e]\!]_{\mathscr{L}}\ \Gamma\} \quad e \in \mathbf{Lexp} \text{ and } e \text{ not of array type} \\
[\![e]\!]_{\mathscr{A}}\ \Gamma &= \{(\sigma,l) \mid (\sigma,l) \in [\![e]\!]_{\mathscr{L}}\ \Gamma\} \quad e \in \mathbf{Lexp} \text{ and } e \text{ of array type} \\
[\![\&e]\!]_{\mathscr{A}}\ \Gamma &= \{(\sigma,l) \mid (\sigma,l) \in [\![e]\!]_{\mathscr{L}}\ \Gamma\} \\
[\![a_0 + a_1]\!]_{\mathscr{A}}\ \Gamma &= \{(\sigma, n_0 + n_1 \mid (\sigma,n_0) \in [\![a_0]\!]_{\mathscr{A}}\ \Gamma \wedge (\sigma,n_1) \in [\![a_1]\!]_{\mathscr{A}}\ \Gamma\} \\
[\![a_0 - a_1]\!]_{\mathscr{A}}\ \Gamma &= \{(\sigma, n_0 - n_1) \mid (\sigma,n_0) \in [\![a_0]\!]_{\mathscr{A}}\ \Gamma \wedge (\sigma,n_1) \in [\![a_1]\!]_{\mathscr{A}}\ \Gamma\} \\
[\![a_0 * a_1]\!]_{\mathscr{A}}\ \Gamma &= \{(\sigma, n_0 * n_1) \mid (\sigma,n_0) \in [\![a_0]\!]_{\mathscr{A}}\ \Gamma \wedge (\sigma,n_1) \in [\![a_1]\!]_{\mathscr{A}}\ \Gamma\} \\
[\![a_0 / a_1]\!]_{\mathscr{A}}\ \Gamma &= \{(\sigma, n_0 / n_1) \mid (\sigma,n_0) \in [\![a_0]\!]_{\mathscr{A}}\ \Gamma \wedge (\sigma,n_1) \in [\![a_1]\!]_{\mathscr{A}}\ \Gamma \\
&\qquad \wedge n_1 \neq 0\}
\end{aligned}$$

Figure 11.4: Denotational semantics for l-expressions and expressions.

11.6.2 Floyd-Hoare Rules

In the actual rules, we only need to change the assignment rule. We first need functions which convert an assertion to an explicit state predicate, called $-^\dagger$ for l-expressions and $-^\#$ for arithmetic expressions.

$$
\begin{array}{lll}
(-)^\dagger & : \mathbf{Lexp} \to \mathbf{Lexp}_s & \\
i^\dagger & = i \quad (i \in \mathbf{Idt}) & \\
l.id^\dagger & = l^\dagger.id & \\
l[e]^\dagger & = l^\dagger[e^\#] & \\
*l^\dagger & = l^\# &
\end{array}
\qquad
\begin{array}{lll}
(-)^\# & : \mathbf{Aexp} \to \mathbf{Aexp}_s & \\
e^\# & = \mathit{read}(\sigma, e^\dagger) \quad (e \in \mathbf{Lexp}) & \\
n^\# & = n & \\
v^\# & = v \quad (v \text{ a logical variable}) & \\
\&e^\# & = e^\dagger & \\
e_1 + e_2{}^\# & = e_1{}^\# + e_2{}^\# & \\
\mathbf{\backslash result}^\# & = \mathbf{\backslash result} & \\
e\,\mathbf{@pre}^\# & = e\,\mathbf{@pre} &
\end{array}
$$

We omit the denotional semantics here for time being.

In the Floyd-Hoare rules, the only major change is the assignment rule, where we need to replace the substitution with the *upd* operation. We also need to insert the conversion functions systematically (see Figure 11.6).

The weakening rule contains implications $P \Longrightarrow Q$ where P and Q are explicit state predicates. They are implicitly quantified universally over the state variables (in particular σ) which they contain, just like a normal predicate is quantified universally over the free variables it contains.

11.7 Verification Conditions

(To come.)

$$[\![-]\!]_{\mathscr{C}} : \mathbf{Env} \to \mathbf{Stmt} \to \Sigma \rightharpoonup \Sigma$$

$$[\![x = e]\!]_{\mathscr{C}}\,\Gamma = \{(\sigma, upd(\sigma, l, a)) \mid (\sigma, l) \in [\![x]\!]_{\mathscr{L}}\,\Gamma \wedge (\sigma, a) \in [\![e]\!]_{\mathscr{A}}\,\Gamma\}$$
$$[\![x = f(t_1, \ldots, t_n)]\!]_{\mathscr{C}}\Gamma = \{(\sigma, upd(\sigma', l, v)) \mid (\sigma, (\sigma', v)) \in \Gamma(f)(v_1, \ldots, v_n) \wedge (\sigma, v_i) \in [\![t_i]\!]_{\mathscr{A}}\Gamma \wedge (\sigma, l) \in [\![x]\!]_{\mathscr{L}}\Gamma\}$$

Figure 11.5: Denotational semantics for statements

$$\frac{}{\Gamma \vdash \{\Gamma\}\, Q[upd(\sigma, l^\dagger, e^\#)/\sigma]\, \{l = e \mid Q\}} \qquad \frac{}{\Gamma \vdash \{\Gamma\}\, Q[e^\#/\backslash\mathbf{result}]\, \{\mathbf{return}\ e \mid P\}}$$

$$\frac{}{\Gamma \vdash \{\Gamma\}\, P\, \{\{\} \mid P\}} \qquad \frac{\Gamma \vdash \{\Gamma\}\, P\, \{c_1 \mid Q\} \quad \Gamma \vdash \{\Gamma\}\, Q\, \{c_2 \mid R\}}{\Gamma \vdash \{\Gamma\}\, P\, \{c_1; c_2 \mid R\}}$$

$$\frac{\Gamma \vdash \{\Gamma\}\, P \wedge b^\# \{c \mid P\}}{\Gamma \vdash \{\Gamma\}\, P\, \{\mathbf{while}\ (b)\ c \mid P \wedge \neg b^\#\}} \qquad \frac{\Gamma \vdash \{\Gamma\}\, P \wedge b^\# \{c_1 \mid Q\} \quad \Gamma \vdash \{\Gamma\}\, P \wedge \neg b^\# \{c_2 \mid Q\}}{\Gamma \vdash \{\Gamma\}\, P\, \{\mathbf{if}\ (b)\ c_1\ \mathbf{else}\ c_2 \mid Q\}}$$

$$\frac{P \Longrightarrow P' \quad \Gamma \vdash \{\Gamma\}\, P'\, \{c \mid Q'\} \quad Q' \Longrightarrow Q \quad R' \Longrightarrow R}{\Gamma \vdash \{\Gamma\}\, P\, \{c \mid Q\}}$$

Figure 11.6: The backward rules of the Floyd-Hoare calculus with explicit state predicates.

Bibliography

[1] D. Burke. All circuits are busy now: The 1990 AT&T long distance network collapse. Technical Report CSC440-01, California Polytechnic Stae University, Nov. 1995. `http://users.csc.calpoly.edu/~jdalbey/SWE/Papers/att_collapse.html`.

[2] M. Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84–, Mar. 1997.

[3] G. Goldberg. The risks digest. `http://catless.ncl.ac.uk/Risks/30/64#subj1`, Apr. 2018.

[4] *IEC 61508 — Functional safety of electrical/electronic/programmable electronic safety-related systems. Part 0: Functional safety*. International Electrotechnical Commission, Geneva, Switzerland, 2000.

[5] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[6] J.-L. Lions. Ariane 5 flight 501 failure — report by the enquiry board. Technical report, Ariane 501 Inquiry Board, July 19th 1996. `http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf`.

[7] M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, Computer Laboratory, 1998.

[8] G. Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing. MIT Press, 1993.