Christoph Lüth, Serge Autexier

Korrekte Software: Grundlagen und Methoden

Sommersemester 2024

Lecture Notes

Universität Bremen

Deutsches Forschungszentrum für Künstliche Intelligenz
German Research Center for Artificial Intelligence

Last revision: 2024-05-15 09:58

Contents

1 Introduction **6**
1.1 Why “correct software”? . . . 6
1.1.1 Well-known Software Disasters # 1: Therac-25 . . . 6
1.1.2 Software Disasters in Space . . . 6
1.1.3 Not-so-well-known Software Disasters . . . 8
1.1.4 Software Correctness and Safety . . . 8
1.2 Semantics . . . 8

2 Operational Semantics **10**
2.1 Sets, Relations, Rules . . . 10
2.2 Partial Functions . . . 11
2.3 Introduction to C0 . . . 12
2.4 State . . . 13
2.5 Evaluating Expressions . . . 13
2.5.1 Arithmetic Expressions . . . 15
2.5.2 Boolean Expressions . . . 15
2.6 Evaluating Statements . . . 16
2.6.1 Undefinedness . . . 16
2.7 Equivalence . . . 18
2.8 Summary . . . 19

3 Denotational Semantics **20**
3.1 Denotational Semantics of Expressions . . . 20
3.2 Denotational Semantics of Statements . . . 23
3.3 Fixed Points . . . 24
3.4 The Fixed Point at Work . . . 25
3.5 Properties of the Fixed Point . . . 27

3.6 Undefinedness 28
3.7 Summary 28
3.8 Appendix: Constructing Fixed Points — cpos and lubs 28

5 Floyd-Hoare Logic **31**
5.1 Why Another Semantics? 31
5.2 Basic Ingredients of Floyd-Hoare Logic 32
5.2.1 Assertions 32
5.2.2 Floyd-Hoare Tripels, Partial and Total Correctness 34
5.3 The Rules of the Floyd-Hoare Calculus 36
5.3.1 Substitution 37
5.3.2 A Notation for Proofs 37
5.4 Finding Invariants 41
5.5 More Examples 43
5.6 Summary 46

6 Correctness of the Floyd-Hoare Calculus **49**
6.1 Syntactic Derivation and Semantic Validity 49
6.2 Soundness 49
6.3 Conclusion 50

7 Structured Datatypes **51**
7.1 Datatypes 51
7.1.1 Arrays 51
7.1.2 Chars and Strings 51
7.1.3 Structures 52
7.2 Memory Models 53
7.3 Extending C0 53
7.3.1 Syntax 53
7.3.2 The State 54
7.3.3 Operational Semantics 54
7.3.4 Denotational Semantics 56
7.3.5 Types and Undefinedness 56
7.3.6 Floyd-Hoare Calculus 56
7.4 Example: Finding the maximum element in an array 57
7.5 Conclusions 59

8 Verification Condition Generation **57**
8.1 Reasoning Backwards . . . 58
8.2 Approximative Weakest Preconditions . . . 60
8.3 Simplifying Verification Conditions . . . 62
8.4 Simplifying Disjunctions . . . 64
8.5 Explicit Assertions . . . 67
8.6 Conclusions . . . 69

9 Forwards with Floyd-Hoare! **70**
9.1 Going Forward . . . 70
9.1.1 Rules . . . 70
9.1.2 Forward Chaining . . . 72
9.1.3 Correctness of the Forward Assignment Rule . . . 74
9.2 Examples . . . 74
9.2.1 The Factorial Example . . . 74
9.2.2 Finding the Maximum Element . . . 76
9.3 Conclusions . . . 79

10 Exceptions and Errors **80**

11 Functions and Procedures **81**
11.1 Introduction . . . 81
11.2 Function Definitions and their Semantics . . . 81
11.3 Function Specifications and their Semantics . . . 83
11.4 Proof Rules for Function Specifications . . . 86
11.5 Function Calls . . . 90

12 Memory Models and References **92**
12.1 Introduction . . . 92
12.2 Extending C0 with References . . . 92
12.3 Memory Models . . . 93
12.4 Axiomatic State Model . . . 94
12.5 Denotational Semantics . . . 95
12.6 Floyd-Hoare Logic with References . . . 95
12.6.1 Explicit State Predicates . . . 95
12.6.2 Floyd-Hoare Rules . . . 97

12.7 Verification Conditions 97

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 [6, 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 [7]:

(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.

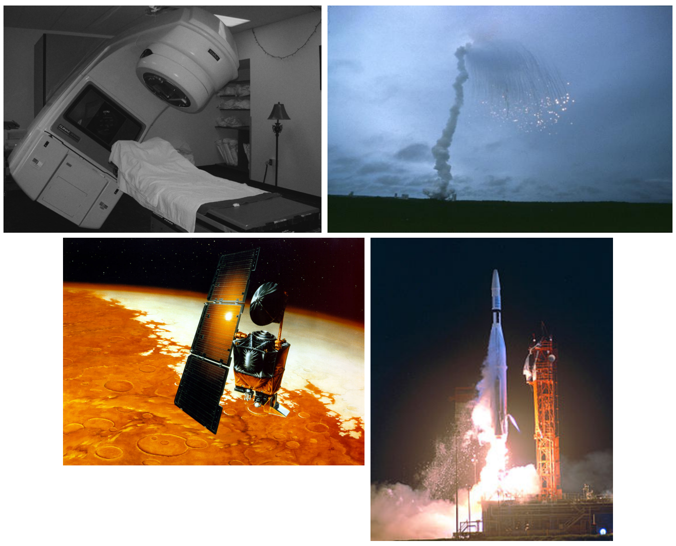
Figure 1.1: Software Disasters (from top left, clockwise): The Therac-25; Ariane-5 exploding on its maiden flight; Atlas booster carrying Mariner-1 taking off; artistic rendition of the Mars Climate Orbiter

(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 $\overline{\dot{R}_n}$) 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 [4], 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 [3]).

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" [5, §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 respect 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 Sets, Relations, Rules

Sets are one of the fundamental concepts of mathematics. They can be defined axiomatically, and this is the subjet of set theory; for an introduction, see one of the many excellent books on that subject. Here, we are not concerned with the finer points of set theory such as transfinite induction or inaccessible cardinals; for our purpose, it is enough that sets may be empty or contain elements, and there is the predicate $x \in X$, which is true if x is an element of the set X (with $x \notin X$ for the negation). We write $\{\phi(x) \mid x \in X\}$ for the subset of X for which ϕ holds (*set comprehension*), for a function $\phi : X \to \mathbb{B}$. Further, the *cartesian product* $X \times Y$ of two sets X, Y is the set of all possible pairs (x, y) with x from X and y from Y:

$$X \times Y \overset{\mathit{def}}{=} \{(x, y) \mid x \in X, y \in Y\}$$

A binary *relation* between two sets X and Y is a subset $R \subseteq X \times Y$; sometimes we write $x\,R\,y$ for $(x, y) \in R$ (the so-called infix notation).

We will use rules later to define relations. Without going into unnecessary technical details, such a rule is given as

$$\frac{\phi_1 \quad \cdots \quad \phi_n}{\psi}$$

for predicates $\phi_1, \ldots, \phi_n, \psi$, and it means that if $\phi_1, \ldots, \phi_n$ are true, then so should ψ. Usually, there is more than one rule. Here is an example for a relation $\mathrm{div} \subseteq \mathbb{Z} \times \mathbb{Z}$, and the corresponding predicate $\mathrm{div}(n, m) : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$:

$$\frac{}{\mathrm{div}(n,n)} \qquad\qquad \frac{n \le m \quad \mathrm{div}(n,m)}{\mathrm{div}(n,m+n)}$$

This formalizes the recursive definition $\mathrm{div}(n,m)$ (n divides m) which says that div is the smallest relation such that

- n divides itself, and
- n divides an m larger than n if n divides $m-n$;

With this, we can conclude that $\mathrm{div}(5,15)$:

$$\frac{5 \le 15 \qquad \dfrac{5 \le 10 \quad \mathrm{div}(5,5)}{\mathrm{div}(5,10)}}{\mathrm{div}(5,15)}$$

Crucially, we can *not* derive *e.g.* $\mathrm{div}(5,7)$ and hence (because div is the smalles relation satisfying the rules), we can deduce that $(5,7) \notin \mathrm{div}$.

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):*

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

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.* $x \notin \mathrm{dom}(f)$). 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

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

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

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

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 $\langle\rangle = \emptyset$.

One important property of partial functions is wether given an undefined argument the result is undefined as well.

Definition 2.2 (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.

If a function is non-strict in an argument x this means intuitively that there cases (*e.g.* given by the other parameters) where x is not used to compute the result.

2.3 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), 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):

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

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

$$b ::= a_1 \,!\!= a_2 \mid a_1 <= a_2 \mid a_1 > a_2 \mid a1 >= a2$$

In a concrete program, an expression $a_1 \,!\!= a_2$ 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.

2.4 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 2.3 (Locations, Values and System State)

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

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

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

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

2.5 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 may evaluate to an integer $n \in \mathbb{Z}$ (a value). We write this as

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

$$\frac{}{\langle i, \sigma \rangle \rightarrow_{Aexp} [\![i]\!]} \qquad \frac{x \in \mathbf{Idt}, x \in Dom(\sigma), \sigma(x) = v}{\langle x, \sigma \rangle \rightarrow_{Aexp} v}$$

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

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

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

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

Figure 2.1: Rules to evaluate arithmetic expressions

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

$$\frac{\langle a_1, \sigma \rangle \rightarrow_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \rightarrow_{Aexp} n_2 \qquad n_1 = n_2}{\langle a_1 == a_2, \sigma \rangle \rightarrow_{Bexp} \mathit{true}}$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \rightarrow_{Aexp} n_2 \qquad n_1 \neq n_2}{\langle a_1 == a_2, \sigma \rangle \rightarrow_{Bexp} \mathit{false}}$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \rightarrow_{Aexp} n_2 \qquad n_1 < n_2}{\langle a_1 < a_2, \sigma \rangle \rightarrow_{Bexp} \mathit{true}}$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \rightarrow_{Aexp} n_2 \qquad n_1 \geq n_2}{\langle a_1 < a_2, \sigma \rangle \rightarrow_{Bexp} \mathit{false}}$$

$$\frac{\langle b, \sigma \rangle \rightarrow_{Bexp} \mathit{true}}{\langle !b, \sigma \rangle \rightarrow_{Bexp} \mathit{false}} \qquad \frac{\langle b, \sigma \rangle \rightarrow_{Bexp} \mathit{false}}{\langle !b, \sigma \rangle \rightarrow_{Bexp} \mathit{true}}$$

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

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

Figure 2.2: Rules to evaluate boolean expressions

2.5.1 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 *quotient* (integer division) of $a, b \in \mathbb{Z}$ (*e.g.* $5 \div 3 = 1$), and $a \cdot b$ is obviously the product.

Undefinedness. Obviously, given an expression a, not for every state σ does a evaluate to a value n. This corresponds to situations where something "goes wrong", in particular when we refer to a variable which is not defined in the state, or when we divide by zero. In that case, there is no n such that $\langle a, \sigma \rangle \rightarrow_{Aexp} n$ and we write $\langle a, \sigma \rangle \rightarrow_{Aexp} \perp$.

That there is no n to evaluate a to in such a situation is a feature, not a bug. It corresponds to the way C (and many other low-level languages) handle such errors. Specifically, in C division by zero is defined to be "undefined behaviour" [2, §6.5.5 (5)], about which nothing must be assumed. Note there is no way to recover from such an error; we will investigate explicit error handling (*i.e.* exceptions) later.

Non-Determinism. Note that an expression could, in theory, evaluate to more than one value. Indeed, the semantics of full C is non-deterministic [8], but our semantics for C0 is deterministic, meaning it always evaluates to (at most) one value. 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, but the proof idea is quite simple: you prove for each rule that it is deterministic (*i.e.* that if $\langle a, \sigma \rangle \rightarrow_{Aexp} n_1$ and $\langle a, \sigma \rangle \rightarrow_{Aexp} n_2$, than $n_1 = n_2$), and that lets you deduce the result for whole deriviations. This is a process called *rule induction*, and we will come back it later.

2.5.2 Boolean Expressions

Similarly, given a state σ, an boolean expressions either evaluates to a *boolean value*, which is either *true* or *false*. We write this as

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

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 not evaluate *at all*. This is sometimes called *short-circuit evaluation*. In other words, conjunction and disjunction are *non-strict* in their second (right) argument; the operational 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* $\sigma \overset{def}{=} \{x \mapsto 6, y \mapsto 5\}$. *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:

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

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

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

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

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

2.6 Evaluating Statements

Under a given state σ_1, a statement evaluates to a new state σ_2, written as

$$\langle c,\sigma_1\rangle \rightarrow_{Stmt} \sigma_2.$$

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).

2.6.1 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 two situations which cause the evaluation of an arithmetic expression to be undefined:

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

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

$$\frac{\langle c_1,\sigma\rangle \rightarrow_{Stmt} \sigma' \qquad \langle c_2,\sigma'\rangle \rightarrow_{Stmt} \sigma''}{\langle c_1;c_2,\sigma\rangle \rightarrow_{Stmt} \sigma''}$$

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

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

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

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

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.

Moreover, a non-terminating program is undefined in this sense. To wit, consider the evaluation of this program:

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

There simple is no state σ' that this program evaluates, *i.e.* there is no state σ' such that we can derive

$$\langle x = 0; \textbf{while}\ (x == 0)\ \{\,\}, \langle\rangle\rangle \rightarrow_{Stmt} \sigma'.$$

2.7 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:*

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

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

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

For example, `A || (A && B)` and `A` are equivalent; this can be shown by considering all possible combinations of all possible values $\bot$, *false*, *true* for `A`, `B`; the interesting cases here are with `B` evaluating to $\bot$ and `A` evaluating to *false*, *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 $w \overset{def}{=} \textbf{while}\ (b)\ c$. We need to show for arbitrary but fixed σ, σ' that $\langle w, \sigma\rangle \rightarrow_{Stmt} \sigma'$ iff $\langle \textbf{if}\ (b)\ c; w\ \textbf{else}\ \{\,\}, \sigma\rangle \rightarrow_{Stmt} \sigma'$.

The proceeds by a case distinction over how the expression `b` evaluates:

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

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

- Case 2: $\langle b, \sigma\rangle \rightarrow_{Bexp} \textit{true}$. Then assume that $\langle c, \sigma\rangle \rightarrow_{Stmt} \sigma''$ (if there is no such σ'', then neither w nor $\textbf{if}\ (b)\ c; w\ \textbf{else}\ \{\,\}$ evaluates to anything), and we have

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

- Finally, note if $\langle b, \sigma \rangle$ does not evaluate to anything (*i.e.* there is no $v \in \mathbb{B}$ such that $\langle b, \sigma \rangle \rightarrow_{Bexp} v$, then neither w nor **if** (b) $c; w$ **else** $\{\}$ evaluate to anything.

$\square$

2.8 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 2.4 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.

Why is compositionality so important? It allows us to reason about *parts* of a program, like an expression or a group of statements. While reasoning directly about the properties of the denotation (the partial function from state to state) is clumsy at best, the denotational semantics allows us to bridge the gap between the operational semantics best describing the way the program works, and a coming semantics best suited to proving properties of the program.

Components of the Semantics

The denotations of our programs (and expressions) are partial functions, as defined in Section 2.2. In particular, just like operational semantics, we need semantics for the constituting parts of our language:

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

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

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

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

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

Figure 3.1: Denotional semantics for expressions

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

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

Figure 3.2: Denotional semantics for statements

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.

(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 $s \overset{def}{=} \langle a \mapsto 7, b \mapsto 3 \rangle$, 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 $s \notin \mathrm{dom}([\![a/0]\!]_{\mathscr{A}})$, *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 2.2 on page 12).

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 and disjunction have the following truth tables:

$\wedge$	$\perp$	*false*	*true*
$\perp$	$\perp$	$\perp$	$\perp$
false	*false*	*false*	*false*
true	$\perp$	*false*	*true*

$\vee$	$\perp$	*false*	*true*
$\perp$	$\perp$	$\perp$	$\perp$
false	*false*	*false*	*false*
true	$\perp$	*false*	*true*

	$\neg$
$\perp$	$\perp$
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*

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

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:

[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.

[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*

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

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

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

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.* $\mathrm{dom}(R \circ S) \subseteq \mathrm{dom}(R)$.

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.* $w \sim \mathbf{if}\ (b)\ \{c;w\}\ \mathbf{else}\ \{\}$ for $w \overset{def}{=} \mathbf{while}\ (b)\ c$. This should hold in denotational semantics as well, so the following equation should hold:

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

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 $\mathit{fix}(f)$ 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 $w \overset{def}{=} \mathbf{while}\ (b)\ c$. 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:

$$\Gamma_{b,c}([\![s]\!]_{\mathscr{C}}) \overset{def}{=} [\![\mathbf{if}\ (b)\ \{c;s\}\ \mathbf{else}\ \{\}]\!]_{\mathscr{C}} \tag{3.3}$$
$$[\![w]\!]_{\mathscr{C}} \overset{def}{=} \mathit{fix}(\Gamma_{b,c})$$

Equation (3.3) is not quite precise yet, because it mixes the abstract syntax ($\mathbf{if}\ (b)\ c\ \mathbf{else}\ \{\}$) 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):

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

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

$$\Gamma(f) \overset{def}{=} \begin{array}{l} \{(\sigma,\sigma) \mid \sigma(n) \le 0\} \cup \\ \{(\sigma, \sigma[x \mapsto \sigma(x)+\sigma(n)][n \mapsto \sigma(n)-1]) \mid \sigma(n) > 0\} \end{array}$$

written in a functional style as

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

Now, consider a specific state $s \overset{def}{=} \langle x \mapsto 0, n \mapsto 0 \rangle$. 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 $w \stackrel{def}{=}$ **while** (b) c we have

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

Proof.

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

□

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 $w \stackrel{def}{=}$ **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

The denotational semantics handles undefinedness *implicitly*, very much like the operational semantics. For example, $n/0$ is not mapped to anything in the denotational semantics, *i.e.* $[\![n/0]\!]_{\mathscr{A}} = \emptyset$, and it does not evaluate to anything in the operational semantics, *i.e.* there is no $v \in \mathbb{Z}$ such that $\langle n/0, \sigma\rangle \rightarrow_{Aexp} v$. The same holds for non-terminating programs.

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, $\perp$, 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 [9, 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.* [9, 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 $x \in P$;*
- *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 $x \sqsubseteq z, y \sqsubseteq z$). 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.[4]

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. $\square$

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 $(\bigsqcup F)(x) \overset{def}{=} \bigsqcup f_1(x) \sqsubseteq f_2(x) \sqsubseteq f_3(x) \ldots$.

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 $Y_\bot$, if we map x to $\bot$ when $f(x)$ is not defined. (This is the deeper reason for the notation $f(x) = \bot$ 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 $f(x) = g(x)$.

[4]It has to pointed out that in this context $\bot$ actually *is* an element we can calculate with, contrary to what we said above. I am sorry for the confusion, but the notation is heavily entrenched in the scientific literature by now.

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 $\bot \in X$, and a continuous partial function $f : X \rightharpoonup X$, the least fixed point of f is given as the least upper bound*

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

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

$$F \overset{def}{=} \bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq f^3(\bot) \sqsubseteq \ldots$$

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

- it is a fixed point: f is continuous, hence
$$f(\bigsqcup F) = f(\bigsqcup_{i \in \omega} f^i(\bot)) = \bigsqcup_{i \in \omega} f(f^i(\bot)) = \bigsqcup_{i \in \omega} f^{i+1}(\bot) = \bigsqcup_{i \in \omega} f^i(\bot) = \bigsqcup F$$
In the fourth step, we have the chain $f^1(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots$; since $\bot \sqsubseteq f(\bot)$ always holds we can add $\bot$ 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 $\bot \sqsubseteq x$, hence $f(\bot) \sqsubseteq f(x) = x$, hence $f^i(\bot) \sqsubseteq x$, 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 $x \in \mathrm{dom}(\sigma)$ then $x \in \mathrm{dom}(\tau)$ 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 $[\![c]\!]_{\mathscr{C}} : \mathbf{Stmt} \rightarrow \Sigma \rightharpoonup \Sigma$ is admissible for all c except $c \equiv$ **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 2.3) 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

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

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

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

$$\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}\}\\
[\![\textbf{\textbackslash 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}\\
[\![\textbf{\textbackslash 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 and arithmetic expressions

definitions of **Bexp** and **Aexp** as follows:

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

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

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

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

*An environment $\Gamma \in$ **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 $\Gamma(p) : \mathbf{Z}^n \rightharpoonup \mathbb{B}$.*

Figure 5.4 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 $b \in$ **Assn** *holds* in a state σ with an assignment I, written $\sigma \models^I b$,

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

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 $P,Q \in$ **Assn** and $c \in$ **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:

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

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

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** $(\mathbf{1})\ \{\,\}]\!]_{\mathcal{C}}$; 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

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

$$\iff \forall I.\forall \sigma.\, \sigma \models^I [\![x = X]\!]^{\Gamma,I}_{\mathcal{B}v} \wedge \exists \sigma'.(\sigma,\sigma') \in [\![x = x+1]\!]_{\mathcal{C}} \Longrightarrow \sigma' \models^I [\![X < x]\!]^{\Gamma,I}_{\mathcal{B}v} \tag{5.4}$$

$$\iff \forall I.\forall \sigma.\, \sigma(x) = I(X) \wedge \exists \sigma'.\sigma' = \sigma[x \mapsto \sigma(x)+1] \Longrightarrow I(X) < \sigma'(x) \tag{5.5}$$

$$\iff \forall I.\forall \sigma.\, \sigma(x) = I(X) \Longrightarrow I(X) < \sigma(x)+1 \tag{5.6}$$

$$\iff \forall I.\, I(X) < I(X)+1 \tag{5.7}$$

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.

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

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

$$\frac{\vdash \{A \land b\}\, c\, \{A\}}{\vdash \{A\}\ \textbf{while}\ (b)\ c\, \{A \land \neg b\}}$$

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

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

Figure 5.3: The rules of the Floyd-Hoare calculus

2. *termination* of while-loops and recursive functions.

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, and the crash involved not only the program but said car as well. 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 (5.4)–(5.7), 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

[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.

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 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 Substitution

The assigment rule models assignment by a substitution in the precondition. Substitutions in formulae with quantifiers (such as $\forall$ and $\exists$ here) can be tricky, but since we only substitute program variables, and quantifiers may only contain logical variables, we are fine here. **??** shows the lengthy formal definition of the function. Note that we use the notation $[e/x]$ for substitution in both assertions and arithmetic expressions. The salient clause is $y[e/x]$ in the case of $y = x$, because only here something really happens.

5.3.2 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:

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

$$
\begin{aligned}
\mathbf{0}[e/x] &\overset{def}{=} \mathbf{0}\\
\mathbf{1}[e/x] &\overset{def}{=} \mathbf{1}\\
(a_0 == a_1)[e/x] &\overset{def}{=} (a_0[e/x]) == (a_1[e/x])\\
(a_0 < a_1)[e/x] &\overset{def}{=} (a_0[e/x]) < (a_1[e/x])\\
(!b)[e/x] &\overset{def}{=} !(b[e/x])\\
(b_1 \,\&\&\, b2)[e/x] &\overset{def}{=} (b_1[e/x]) \,\&\&\, (b_2[e/x])\\
(b_1 -- > b2)[e/x] &\overset{def}{=} (b_1[e/x]) -- > (b_2[e/x])\\
(p(e_1,\ldots,e_n))[e/x] &\overset{def}{=} p(e_1[e/x],\ldots e_n[e/x])\\
(\forall v.\, b)[e/x] &\overset{def}{=} \forall v.\,(b[e/x]) \quad x \neq v, v \notin FV(e)\\
(\exists v.\, b)[e/x] &\overset{def}{=} \exists v.\,(b[e/x]) \quad x \neq v, v \notin FV(e)
\end{aligned}
$$

$$
\begin{aligned}
n[e/x] &\overset{def}{=} n\\
y[e/x] &\overset{def}{=} \begin{cases} e & x = y\\ y & x \neq y \end{cases}\\
(a_0 + a_1)[e/x] &\overset{def}{=} (a_0[e/x]) + (a_1[e/x])\\
(a_0 - a_1)[e/x] &\overset{def}{=} (a_0[e/x]) - (a_1[e/x])\\
(a_0 * a_1)[e/x] &\overset{def}{=} (a_0[e/x]) * (a_1[e/x])\\
(a_0 / a_1)[e/x] &\overset{def}{=} (a_0[e/x]) / (a_1[e/x])\\
(f(a_0,\ldots,a_n))[e/x] &\overset{def}{=} f(a_0[e/x],\ldots,a_n[e/x])
\end{aligned}
$$

Figure 5.4: Definition of the substitution function.

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

```
// {$P$}
// {$P_1$}
x= e;
// {$P_2$}
// {$P_3$}
while (x< n) {
    // {$P_3 \wedge x < n$}
    // {$P_4$}
    z= a;
    // {$P_3$}
    }
// {$P_3 \wedge \neg(x < n)$}
// {$Q$}
```

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) {
      // {$b \wedge P$}
      ...
      // {$Q$}
      } else {
      // {$\neg b \wedge P$}
      ...
      // {$Q$}
      }
    // {$Q$}
    ```

  - For the while-loop:

    ```
    // {$P$}
    while (b) {
      // {$b \wedge P$}
      ...
      // {$P$}
      }
    // {$\neg b \wedge P$}
    ```

```
// {x = X ∧ y = Y}
// {y = Y ∧ x = X}
z= x;
// {y = Y ∧ z = X}
x= y;
// {x = Y ∧ z = X}
y= z;
// {x = Y ∧ y = X}
```

Figure 5.5: 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 5.6: Another example for a correctness proof.

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.

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).

```
// $\{0 \leq n\}$
// $\{1 = 0! \land 0 \leq n\}$
// $\{1 = (1-1)! \land 1 \leq 1 \land 1-1 \leq n\}$
p= 1;
// $\{p = (1-1)! \land 1 \leq 1 \land 1-1 \leq n\}$
c= 1;
// $\{p = (c-1)! \land 1 \leq c \land c-1 \leq n\}$
while (c<= n) {
   // $\{p = (c-1)! \land 1 \leq c \land c-1 \leq n \land c \leq n\}$
   // $\{p*c = (c-1)!*c \land 1 \leq c \land c \leq n\}$
   // $\{p*c = c! \land 1 \leq c \land c \leq n\}$
   // $\{p*c = ((c+1)-1)! \land 1 \leq c+1 \land (c+1)-1 \leq n\}$
   p= p*c;
   // $\{p = ((c+1)-1)! \land 1 \leq c+1 \land (c+1)-1 \leq n\}$
   c= c+1;
   // $\{p = (c-1)! \land 1 \leq c \land c-1 \leq n\}$
   }
// $\{p = (c-1)! \land 1 \leq c \land c-1 \leq n \land \neg(c \leq n)\}$
// $\{p = (c-1)! \land 1 \leq c \land c-1 \leq n \land c > n\}$
// $\{p = (c-1)! \land 1 \leq c \land c-1 = n\}$
// $\{p = n!\}$
```

Figure 5.7: The factorial example.

Figure 5.5 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 5.6. 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 $z < y$, because $z \overset{def}{=} -(x+y)$ always satisfies this specification; we have to require the minimum is either x or y.) Figure 5.6 shows how to use weakening to massage the assertions into the form required by the if-statement; we use logical equivalences such as $(y = y) = \mathit{true}$, $\mathit{true} \lor P = \mathit{true}$, and $\mathit{true} \land P = P$.

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 32 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:
$$p = (c-1)! \tag{5.8}$$
- 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
$$p = (c-1)! \wedge c-1 \leq n \tag{5.9}$$
- We now try to show that the loop body preserves (5.9), 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:
$$n! = \begin{cases} 1 & n = 0 \\ (n-1)! \cdot n & n > 0 \end{cases} \tag{5.10}$$
  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
$$p = (c-1)! \wedge c-1 \leq n \wedge c > 0 \tag{5.11}$$
  Of course, we need to repeat the calculation now that (5.11) 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 5.7 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 (5.10); 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 (5.10) (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:

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

(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.

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

5.5 More Examples

Figure 5.8 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

$$prod(a,b) = \begin{cases} 1 & a > b \\ a \cdot prod(a+1,b) & a \leq b \end{cases} \tag{5.12}$$

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:

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

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

```
// {$n = N \land 0 \le n$}
// {$n = N \land 1 = prod(n+1,N) \land 0 \le n$}
// {$1 = prod(n+1,N) \land 0 \le n \land n = N$}
p= 1;
// {$p = prod(n+1,N) \land 0 \le n \land n \le N$}
while (0 < n) {
   // {$p = prod(n+1,N) \land 0 \le n \land n \le N \land 0 < n$}
   // {$n \cdot p = n \cdot prod(n+1,N) \land n \le N \land 0 < n$}
   // {$p \cdot n = prod(n,N) \land 0 < n \land n \le N \land 0 < n$}
   p= p*n;
   // {$p = prod(n,N) \land n \le N \land 0 < n$}
   // {$p = prod((n-1)n+1,N) \land n-1 \le N \land 0 \le n-1$}
   n= n-1;
   // {$p = prod(n+1,N) \land n \le N \land 0 \le n$}
   }
// {$p = prod(n+1,N) \land 0 \le n \land \neg(0 < n)$}
// {$p = prod(n+1,N) \land 0 \le n \land n \le 0$}
// {$p = N!$}
```

Figure 5.8: A variation on the factorial example: counting downwards.

```
// {$0 \le a$}
// {$a = b \cdot 0 + a \land 0 \le a$}
r= a;
// {$a = b \cdot 0 + r \land 0 \le r$}
q= 0;
// {$a = b \cdot q + r \land 0 \le r$}
while (b <= r) {
   // {$a = b \cdot q + r \land 0 \le r \land b \le r$}
   // {$a = b \cdot q + b + r - b \land b \le r$}
   // {$a = b \cdot (q+1) + (r-b) \land 0 \le r-b$}
   r= r-b;
   // {$a = b \cdot (q+1) + r \land 0 \le r$}
   q= q+1;
   // {$a = b \cdot q + r \land 0 \le r$}
   }
// {$a = b \cdot q + r \land 0 \le r \land \neg(b \le r)$}
// {$a = b \cdot q + r \land 0 \le r \land r < b$}
```

Figure 5.9: Computing the integer quotient and remainder.

```
// $\{0 \leq a\}$
// $\{1-1 \leq a \wedge 1 = 2 \cdot 0 + 1 \wedge 1 = 0^2 + 1\}$
t= 1;
// $\{1-t \leq a \wedge t = 2 \cdot 0 + 1 \wedge 1 = 0^2 + t\}$
s= 1;
// $\{s-t \leq a \wedge t = 2 \cdot 0 + 1 \wedge s = 0^2 + t\}$
i= 0;
// $\{s-t \leq a \wedge t = 2 \cdot i + 1 \wedge s = i^2 + t\}$
while (s <= a) {
  // $\{s-t \leq a \wedge t = 2 \cdot i + 1 \wedge s = i^2 + t \wedge s \leq a\}$
  // $\{t = 2 \cdot i + 1 \wedge s = i^2 + t \wedge s \leq a\}$
  // $\{s \leq a \wedge t+2 = 2 \cdot i + 3 \wedge s = i^2 + 2 \cdot i + 1\}$
  // $\{s+(t+2)-(t+2) \leq a \wedge t+2 = 2 \cdot (i+1) + 1 \wedge s+(t+2) = (i+1)^2 + (t+2)\}$
  t= t+ 2;
  // $\{s+t-t \leq a \wedge t = 2 \cdot (i+1) + 1 \wedge s+t = (i+1)^2 + t\}$
  s= s+ t;
  // $\{s-t \leq a \wedge t = 2 \cdot (i+1) + 1 \wedge s = (i+1)^2 + t\}$
  i= i+ 1;
  // $\{s-t \leq a \wedge t = 2 \cdot i + 1 \wedge s = i^2 + t\}$
  }
// $\{s-t \leq a \wedge t = 2 \cdot i + 1 \wedge s = i^2 + t \wedge \neg(s \leq a)\}$
// $\{i^2 \leq a \wedge a < (i+1)^2\}$
```

Figure 5.10: Computing the integer square root.

A quick first check if we can conclude $p = N!$ from (5.17) 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 (5.12)

$$p \cdot n = prod(n,N) \tag{5.14}$$
$$n \cdot p = n \cdot prod(n+1,N) \tag{5.15}$$
$$p = prod(n+1,N) \tag{5.16}$$

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

$$p = prod(n+1,N) \wedge 0 \leq n \wedge n \leq N \tag{5.17}$$

The fully annotated program is shown in Figure 5.8. 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 5.9 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 5.10 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 5.11: Computing powers of 2 in three variations.

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] Programming langauges — C. ISO/IEC Standard 9899:1999(E), 1999. Second Edition.

[3] M. Dowson. The Ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84–, Mar. 1997.

[4] G. Goldberg. The risks digest. `http://catless.ncl.ac.uk/Risks/30/64#subj1`, Apr. 2018.

[5] *IEC 61508 — Functional safety of electrical/electronic/programmable electronic safety-related systems. Part 0: Functional safety*. International Electrotechnical Commission, Geneva, Switzerland, 2000.

[6] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[7] 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`.

[8] M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, Computer Laboratory, 1998.

[9] G. Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing. MIT Press, 1993.