Christoph Lüth, Serge Autexier

Korrekte Software: Grundlagen und Methoden

Sommersemester 2018

Lecture Notes

Universität Bremen

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Last revision: 01/08/2018.

Contents

Preliminary Notes

Status of this document:

- Chapter 4 (Equivalence of denotational and operational semantics) is *missing*.
- Chapter 8 (Modelling and Specification) is *missing*.
- Chapter 9 requires *revisions* (examples need to be checked, some parts missing).
- Chapter 10 requires *major revisions* (most parts missing).

Chapter 1

Introduction

1.1 Why "correct software"?

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

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

1.1.2 Software Disasters in Space

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

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

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

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

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

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

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

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

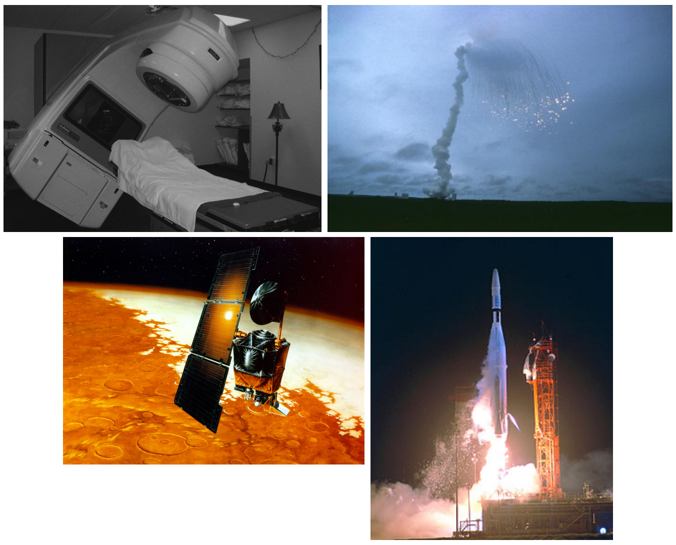

Figure 1.1: Software Disasters (from top left, clockwise): The Therac-25; Ariane-5 exploding on its maiden flight; Atlas booster carrying Mariner-1 taking off; artistic rendition of the Mars Climate Orbiter

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

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

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

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

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

The Mariner-1 had to be destroyed because the guidance system of the Atlas booster rocket carrying Mariner-1 was faulty. The guidance system was taking radar measurements and turning them into control commands for the rocket. It turned out that the programmer had missed on overbar[1] (as in $\overline{\dot{R}_n}$) 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 agency do in fact have a very good culture of learning from errors; thus,

[1]This is sometimes, and incorrectly, referrede to as a missing hyphen.

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

1.1.3 Not-so-well-known Software Disasters

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

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

1.1.4 Software Correctness and Safety

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

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

1.2 Semantics

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

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

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

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

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

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

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

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

  Concretely, we model programs by partial functions between states (mapping variable names to values, as above). It is easy to see how this works for the first two lines in our factorial program, but modellign 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 note in correspondence with the popular C compilers. However, there is certified C compiler, safecert, which has been proven (certified) to be correct with respec to its denotational semantics.

Chapter 2

Operational Semantics

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

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

2.1 Introduction to C0

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

We give the *abstract* syntax. 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 assginments; control statements, which are conditional (**if**) and iteration (**while**); and structured statements, which are the sequencing and the empty statement.

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

Note how just like we do not have parentheses to group expressions, we also do not have brackets ({...}) to group statements. Statements grouped with brackets are called compound statements in C.

Also, in concrete syntax of C, the semicolon is used to terminate basic statements, not to concatenate them; the difference is that in C, we need to write

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

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

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

2.2 State

The basis of all semantics (not only the operational semantics) is the *program state*. Formally, the program state is a partial 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 1 (Locations, Values and System State)

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

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

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

The notation $X \rightharpoonup Y$ denotes a partial map, or partial function[1] from X to Y. For a partial function f, we write $f(x) = \bot$ if the f is undefined at x, and we write $Dom(f) \stackrel{def}{=} \{x \mid f(x) \neq \bot\}$ for the *domain* of f, *i.e.* the subset of X where f is defined (returns a value from Y).

For a state $\sigma \in \Sigma$, a location $x \in \mathbf{Loc}$ and a value $n \in \mathbf{V}$, we write $\sigma[x/n]$ for the functional update of the state σ at location x with value n. That is, $\sigma[x/n]$ is a new state which for any $y \in \mathbf{Loc}$ is defined as

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

[1]A map is a finite function, *i.e.* a function where the domain is finite.

2.3 Evaluating Expressions

Given a state σ, an arithmetic expression a either evaluates to an integer $n \in \mathbb{Z}$ (a value), or an undefined error value $\bot$. We write this as

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

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

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

Definition 2 (Strict Function) *A function $f : X \rightarrow Y$ is* strict *if $f(\bot) = \bot$, i.e. if an undefined argument makes the function value undefined as well.*

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

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

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

Here is an alternative way to write this down (for the conjunction only):

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

$$\frac{\langle b_1, \sigma \rangle \rightarrow_{Bexp} \bot}{\langle b_1 \,\&\&\, b_2, \sigma \rangle \rightarrow_{Bexp} \bot}$$

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

Example 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*,*

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

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

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

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

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

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

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

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

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

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

Figure 2.1: Rules to evaluate arithmetic expressions

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

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

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

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

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

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

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

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

$$\frac{\langle b_1, \sigma\rangle \rightarrow_{Bexp} t_1 \qquad \langle b_2, \sigma\rangle \rightarrow_{Bexp} t_2}{\langle b_1 \,\&\&\, b_2, \sigma\rangle \rightarrow_{Bexp} t} \quad t = \begin{cases} \mathit{true} & t_1 = t_2 = \mathit{true} \\ \mathit{false} & t_1 = \mathit{false} \text{ or } (t_1 = \mathit{true} \text{ and } t_2 = \mathit{false}) \\ \bot & \text{otherwise} \end{cases}$$

$$\frac{\langle b_1, \sigma\rangle \rightarrow_{Bexp} t_1 \qquad \langle b_2, \sigma\rangle \rightarrow_{Bexp} t_2}{\langle b_1 \,||\, b_2, \sigma\rangle \rightarrow_{Bexp} t} \quad t = \begin{cases} \mathit{true} & t_1 = t_2 = \mathit{false} \\ \mathit{false} & t_1 = \mathit{true} \text{ or } (t_1 = \mathit{false} \text{ and } t_2 = \mathit{true}) \\ \bot & \text{otherwise} \end{cases}$$

Figure 2.2: Rules to evaluate boolean expressions

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:

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

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):

$$\langle !(x == 0)\ \&\&\ (z/x == 0),\sigma\rangle \rightarrow_{Bexp} ? \tag{2.1}$$
$$\langle (z/x == 0)\ ||\ x == 0,\sigma\rangle \rightarrow_{Bexp} ? \tag{2.2}$$
$$\langle y - z * ((a+1)/2),\sigma\rangle \rightarrow_{Aexp} ? \tag{2.3}$$

2.4 Evaluating Statements

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

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

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 1 (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 fragmen:

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

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

2.5 Equivalence

One application of operatioanl 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

$$\frac{\langle a, \sigma\rangle \rightarrow_{Aexp} n \in \mathbf{Z}}{\langle x = a, \sigma\rangle \rightarrow_{Stmt} \sigma[n/x]} \qquad \frac{\langle a, \sigma\rangle \rightarrow_{Aexp} \perp}{\langle x = a, \sigma\rangle \rightarrow_{Stmt} \perp}$$

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

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

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

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

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

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

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

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

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

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

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

Figure 2.3: Rules to evaluate statements

boolean expressions — two expressions are equivalent if they always evaluate to the same value under all states.

Formally:

Definition 3 (Equivalence) *Given two arithmetic expressions a_1, a_2, two boolean expressions b_1, b_2 are two programs c_0, c_1 respectively. They are* equivalent *iff:*

$$a_1 \sim_{Aexp} a_2 \quad \textit{iff} \quad \forall \sigma, n. \langle a_1, \sigma \rangle \rightarrow_{Aexp} n \Leftrightarrow \langle a_2, \sigma \rangle \rightarrow_{Aexp} n \tag{2.4}$$

$$b_1 \sim_{Bexp} b_2 \quad \textit{iff} \quad \forall \sigma, b. \langle b_1, \sigma \rangle \rightarrow_{Bexp} b \Leftrightarrow \langle b_2, \sigma \rangle \rightarrow_{Bexp} b \tag{2.5}$$

$$c_0 \sim_{Stmt} c_1 \quad \textit{iff} \quad \forall \sigma, \sigma'. \langle c_0, \sigma \rangle \rightarrow_{Stmt} \sigma' \Leftrightarrow \langle c_1, \sigma \rangle \rightarrow_{Stmt} \sigma' \tag{2.6}$$

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

For a longer example, we show that

$$\textbf{while } (b)\ c \sim \textbf{if } (b)\ \{c; \textbf{while } (b)\ c\}\ \textbf{else } \{\} \tag{2.7}$$

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 expressions `b` and `c` evaluate, starting with `b`:

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

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

- Case 2: $\langle b, \sigma \rangle \rightarrow_{Bexp} 1$:
  - Case 2.1: $\langle c, \sigma \rangle \rightarrow_{Stmt} \sigma'$, $\sigma' \neq \perp$.

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

  - Case 2.2: $\langle c, \sigma \rangle \rightarrow_{Stmt} \perp$

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

- Case 3: $\langle b, \sigma \rangle \rightarrow_{Bexp} \perp$:

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

□

2.6 Summary

- Operational semantics are 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. Neither does operational semantics have to be deterministic — expressions may evaluate to more than one possible value. The semantics of full C is non-deterministic for expressions [7], but our semantics for C0 is deterministic.
- 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 2.2 is the starting point.

In general, the denotional semantics is written, for a program c, as $[\![c]\!]$. The denotional semantics should be compositional, that is the semantics of a compound statement is given by composing the semantics of the basic statements:

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

But before we can proceed with the semantics, we need some mathematical preliminaries.

3.1 Fixed Points

Given a function $f : AA \to A$, a *fixed point* of f is an $a \in A$ such that $f(a) = a$.

Examples:

- For $f(x) = \sqrt{x}$ the fixed points are 0 und 1; similarly for $f(x) = x^2$.
- For a sorting function, all sorted lists are fixed points.

We will need fixed points to give a semantics to while-loops (iteration). To see why, recall that we have proven before (2.7) that

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

which suggest that for the denotional semantics

$$[\![\textbf{while}\ (b)\ c]\!] = [\![\textbf{if}\ (b)\ c; \textbf{while}\ (b)\ c\ \textbf{else}\ \{\}]\!]$$

This can be read as a *recursive* definition for the semantics of the **while**-loop, defining the left-hand side by the right-hand side. This means we define something in its own terms, which is a bit like chasing your own shadow. However, mathematically recursive definitions are no problem, and they are solved by fixed points.

Thus, we need to construct fixed points. We do this *inductively*, so we need something to describe how to construct sets step-by-step until they do not change anymore.

3.1.1 Rules and Rule Instances

Generally speaking, a rule allows us to derive a *conclusion* from a set of *premisses*. This is written as

$$\frac{y}{x_1,\ldots,x_n}(n \geq 0).$$

With $n = 0$, the rule states that y holds without any precondition. The conclusion and premisses may contain variables; these can be instantiated to give a *rule instance*. Instantiation has to be uniform throughout the rule (*i.e.* a variable has to be instantiated with the same term in all premisses and the conclusion), giving us a rule instance.

We use rules to define define sets *inductively* — that is, sets that we cannot define directly but we give the rules by which to form them. Here is an example. Consider the following rules:

$$\frac{-}{2^2} \qquad \frac{-}{2^3} \qquad \frac{n \quad m}{n \cdot m} \tag{3.1}$$

Instances of the rules are, *e.g.*

$$\frac{-}{4} \qquad \frac{-}{8} \qquad \frac{4 \quad 8}{32} \qquad \frac{4 \quad 4}{16} \qquad \frac{16 \quad 32}{512} \qquad \frac{3 \quad 5}{15} \qquad \ldots$$

These rules form a set, starting with 4 and 8 (because these are rule instances without any premisses), then 32 (because both 4 and 8 are in the set, so is 32, then 16, then 512, then 15 *etc.*). We can define this process formally:

Definition 4 (Rule Application) *Let R be a set of rule instances, and B a set. Then, we define*

$$\begin{aligned}
\hat{R}(B) &\overset{def}{=} \{y \mid \exists x_1,\ldots,x_k \in B.\, \frac{x_1,\ldots,x_k}{y} \in R\} \\
\hat{R}^0(B) &\overset{def}{=} B \\
\hat{R}^{i+1}(B) &\overset{def}{=} \hat{R}(\hat{R}^i(B))
\end{aligned}$$

For the rules in (3.1), we get

$$\begin{aligned}
\hat{R}^0(\emptyset) &= \emptyset \\
\hat{R}^1(\emptyset) &= \hat{R}(\emptyset) = \{4,8\} \\
\hat{R}^2(\emptyset) &= \{16,32,64,4,8\} \\
\hat{R}^3(\emptyset) &= \{128,256,512,1024,2048,4096,16,32,64,4,8\} \\
\hat{R}^{i+1}(\emptyset) &= \{2^{2k+3l} \mid 1 \leq k+l \leq 2^i\}
\end{aligned} \tag{3.2}$$

To show (3.2), we use induction on i. For the induction base, $i = 0$, hence we get $k = 1, l = 0$ or $k = 0, l = 1$, then $\hat{R}^1 = \{2^2, 2^3\}$. For the induction step, we get:

$$\begin{aligned}
\hat{R}^{i+1}(\emptyset) &= \hat{R}(\hat{R}^i(\emptyset) \\
&= \hat{R}(\{2^{2k+3l} \mid 1 \leq k+l \leq 2^{(i-1)}\} \\
&= \{2^2\} \cup \{2^3\} \cup \{2^{2k_1+3l_1} \cdot 2^{2k_2+3l_2} \mid 1 \leq k_1 + l \leq 2^{i-1}, 1 \leq k_2 + l_2 \leq 2^{i-1}\} \\
&= \{2^2, 2^3, 2^{2k_1+3l_1+2k_2+3l_2} \mid 1 \leq k_1 + l_1 + k_2 + l_2 \leq 2^{i-1} + 2^{i-1}\} \\
&= \{2^{2(k_1+k_2)+3(l_1+l_2)} \mid 1 \leq (k_1+k_2)+(l_1+l_2) \leq 2 \cdot 2^{i-1}\} \\
&= \{2^{2(k_1+k_2)+3(l_1+l_2)} \mid 1 \leq (k_1+k_2)+(l_1+l_2) \leq 2^i\}
\end{aligned}$$

You may see where this is going. Rules give us a single step to form a set; the rule application $\hat{R}^i$ iterates this procedure. We now want to apply rules until the set they define inductively does not change anymore (*i.e.* we have reached a fixed point). The following definition formalizes this concept of "not changing anymore":

Definition 5 (Closed under R) *Given a set R of rules, a set S is closed under R (R-closed) iff*

$$\hat{R}(S) \subseteq S$$

Applying a rule only adds to the set being constructed. The general concept is monotonicity:

Definition 6 (Monotone Function) *Given a function $f : A \to B$, then f* is monotone *iff*

$$\forall A_1, A_2. A_1 \subseteq A_2 \Rightarrow \{f(a_1) \mid a_1 \in A_1\} \subseteq \{f(a_2) \mid a_2 \in A_2\}$$

For a rule, the operation $\hat{R}$ is function, mapping B to $\hat{R}(B)$. (We are being a bit imprecise here about the domain and range of $\hat{R}$. Suffice it to say these are the *terms* built using the language we describe the rules in, *e.g.* numbers in example (3.1). This is explained in detail in *e.g.* [8] or [9].)

Lemma 1 (Rule Application is monotone) *For a set of rules R, the induced operation $\hat{R}$ is monotone.*

Now, given a set of rules, we can apply the rule instances until the set does not change (*i.e.* grow) anymore. The following lemma explains the construction:

Lemma 2 (Smallest Fixed Point) *Let R be a set of rules, and let $A_i \overset{def}{=} \hat{R}^i(\emptyset)$ for $i \in \mathbb{N}$, and $A \overset{def}{=} \bigcup_{i \in \mathbb{N}} A_i$. Then:*

(a) A ist closed under R,

(b) $\hat{R}(A) = A$, and

(c) A ist die smallest set closed under R.

Proof.

(a) A is closed under R:
Assume $\frac{x_1,\ldots,x_k}{y} \in R$ and $x_1,\ldots,x_k \subseteq A$. Since$A = \bigcup_{i \in \mathbb{N}} A_i$, there is a j such that $x_1,\ldots,x_k \subseteq A_j$. Hence we have:

$$\begin{aligned} y \in \hat{R}(A_j) &= \hat{R}(\hat{R}^j(\emptyset)) \\ &= \hat{R}^{j+1}(\emptyset) \\ &= A_{j+1} \subseteq A. \end{aligned}$$

(b) $\hat{R}(A) = A$:
We show the equality by showing inclusion in both directions.

- $\hat{R}(A) \subseteq A$:
  With A closed under R, we have $\hat{R}(A) \subseteq A$.

- $A \subseteq \hat{R}(A)$:
  Let $y \in A$. Because $A = \bigcup_{i \in \mathbb{N}} A_i$, it menas there is $n > 0$ with $y \in A_n$. Further, y cannot be in $\emptyset$, so it has to be added at some n, *i.e.* we can assume that $y \notin A_{n-1}$.
  Hence there must be a rule instance $\frac{x_1,\ldots,x_k}{y} \in R$ with $x_1,\ldots,x_k \subseteq A_{n-1} \subseteq A$.
  Because $\hat{R}$ is monotone, we have $\hat{R}(A_{n-1}) \subseteq \hat{R}(A)$.
  With $y \in A_n = \hat{R}(A_{n-1})$, it follows that $y \in \hat{R}(A)$.

(c) A is the smallest set closed under R, *i.e.* for any other set B closed under R we have $A \subseteq B$. We show by induction over n that $A_n \subseteq B$; it follows that $A \subseteq B$.

- Basisfall: $A_0 = \emptyset \subseteq B$. (The empty set is a subset of all sets.)
- Induktionsschritt: With B closed under R, we have : $\hat{R}(B) \subseteq B$.
  Our induction assumption is that $A_n \subseteq B$. Then $A_{n+1} = \hat{R}(A_n) \subseteq \hat{R}(B) \subseteq B$ because $\hat{R}$ is monotone, and B is closed under R.

□

3.1.2 Least Fixed Point Operator

We call the operator taking R to the set A as defined in Lemma 2 the smallest fixed point operator, as property (a) says A is a fixed point, and property (c) says it is the smallest one.

Definition 7 (Smallest Fixed Point Operator) *Given a set R of rules, the* smallest fixed point *of R is defined as*

$$\mathit{fix}(()\hat{R}) \overset{def}{=} \bigcup_{n \in N} \hat{R}^n(\emptyset)$$

Consider the rule set R from (3.1) again. We already saw that $\hat{R}^{i+1}(\emptyset) = \{2^{2k+3l} \mid 1 \leq k+l \leq 2^i\}$. With that, we can conclude that

$$\mathit{fix}(\hat{R}) = \{2^{2k+3l} \mid 1 \leq k+l\}$$

This is equivalent to

$$\mathit{fix}(\hat{R}) = \{2^j \mid 2 \leq j\}$$

because any $n \geq 2$ can be written as $2k+3l$ for some $k,l \geq 1$ (easy if n is even, if n then take $n = (n-3)+3$ where $n-3$ is even), and on the other hand, $2k+3l$ is at least 2 if $1 \leq k+l$ ($k=1, l=0$). But observe that $\hat{R}^i(\emptyset) \neq \{2^j \mid 2 \leq j \leq 2^i\}$.

We are now equipped to define the denotational semantics of our language.

3.2 Denotational Semantics

In general:

- each artithmetic expression a :**Aexp** is denoted by a partial function $\Sigma \rightharpoonup \mathbf{Z}$,
- each boolean expression b :**Bexp** is denoted by a partial function $\Sigma \rightharpoonup \mathbb{B}$, and

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

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

$$\mathscr{B}[\![a]\!] : \mathbf{Bexp} \rightarrow (\Sigma \rightharpoonup \mathbb{B})$$

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

Figure 3.1: Denotional semantics for expressions

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

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

Figure 3.2: Denotional semantics for statements

- each statement c :**Stmt** is denoted by a partial function $\Sigma \rightharpoonup \Sigma$.

Figure 3.1 gives the denotational semantics for expressions. Division by 0 is explicitly left undefined, and the denotations of conjunction and disjunction are non-strict on the right: if the left argument denotes 0 (for conjuction) or 1 (for disjunction), then the right argument is not evaluated at all.

Figure 3.2 gives the denotational semantics for statements, using the fixed point operator for the iteration. There, **Id** is the identity relation, and $R_1 \circ R_2$ is the composition operator for relations, defined as

$$\mathbf{Id}_X \overset{def}{=} \{(x,x) \mid x \in X\}$$
$$R_2 \circ R_1 \overset{def}{=} \{(x,z) \mid \exists y.\, (x,y) \in R_1 \land (y,z) \in R_2\}$$

Considering relations as partial functions, these are the identity function and function composition. The obvious properties hold, such as associativity of function composition $R_3 \circ (R_2 \circ R_1) = (R_3 \circ R_2) \circ R_1$ and the unit laws $\mathbf{Id} \circ R = R = R \circ \mathbf{Id}$.

Chapter 5

The Floyd-Hoare Logic

In this chapter, we introduce a third semantics, or another mathematical view of "what a program does". The Floyd-Hoare logic, also sometimes referred to as axiomatic semantics, differs from the operational semantics (which formalises the execution of a program) and denotional 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 and the general enhanced confidence by giving a third view of the elusive "meaning" of a program and showing it coincides with the other two? Well, "dreimal ist Bremer Recht" and all that, but consider again the example program in Figure 1.2 on page 9. It computes the factorial of the input variable n in the variable p, but how can we *prove* that? We could calculate the semantics, *e.g.* using the denotational semantics, but we will run into two difficulties:

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

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, and prove that these hold.

5.2 Basic Ingredients of Floyd-Hoare Logic

The basic ingredients of the Floyd-Hoare logic are:

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

- 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 assertion, and what it means for assertions to hold.

5.2.1 Assertions

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

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

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

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

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*.)

Definition 8 (Interpretation and Environment) *An interpretation $I \in$ **Inprt** is a partial map $I : \textbf{Var} \rightharpoonup \mathbb{B}$ which assigns integer values to logical variables.*

*An environment $\Gamma \in$ **Env** maps*

- *each n-ary logical function f to an n-ary function $\Gamma(f) : \mathbf{Z}^n \rightharpoonup \mathbf{Z}$, and*

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

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

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

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

Figure 5.1: Denotional semantics for assertions

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

Note how logical functions and predicates are mapped to total[2] functions which do not take the current state σ as a parameter, and hence do not depend on it; they are stateless (or pure). Logical functions and predicates may seem a bit obscure right now, but please bear with us for the time being.

Logical variables, however, deserve some explanation. Essentially, they allow us to formulate specifications which are invariant over the state. For example, to write down 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).

To define the denotational semantics of an assertion b, we need an environment (which maps the functions and predicates to a semantic meaning), and an interpretation. Figure 5.1 gives the additional equations to interpret the new constructs. Recall from page 11 our notations for partial functions; in particular, for an interpretation I we write $I[n/x]$ for updating the interpretation at the variable x with the (new) value n. Figure 5.1 shows the extension of the denotational semantics for expressions to assertions.

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. Formally, an assertion $b \in \mathbf{Assn}$ *holds* in a state σ with an assignment I, written $\sigma \models^I b$ iff

$$\sigma \models^I \text{ iff } \mathit{DenBvb}^I(\sigma) = \mathit{true} \tag{5.1}$$

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

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

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

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

Points to note:

- The following holds $\models \{\mathbf{1}\}$ **while** $(\mathbf{1})\ \{\,\}\ \{\mathbf{1}\}$, because even though for all states σ (and assignments I), we have $\sigma \models^I \mathbf{1}$, but there is not state σ' such that $(\sigma, \sigma') \in \mathscr{C}[\![$**while** $(\mathbf{1})\ \{\,\}]\!]$.
- For exactly the same reason, $\models [\mathbf{1}]$ **while** $(\mathbf{1})\ \{\,\}\ [\mathbf{1}]$ does not hold.
- However, both $\models \{\mathbf{0}\}$ **while** $(\mathbf{1})\ \{\,\}\ \{\mathbf{1}\}$ and $\models [\mathbf{0}]$ **while** $(\mathbf{1})\ \{\,\}\ [\mathbf{1}]$ hold; in fact, $\models \{\mathbf{0}\}\,c\,\{Q\}$ and $\models [\mathbf{0}]\,c\,[Q]$ hold for any c and Q, because an implication is true when the premisse is false ($\mathit{false} \Longrightarrow \phi$ is always true).

[2]One might want partial functions here, but that would make the logical partial.

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

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

If we spell out definition (5.2), we get

$$\begin{aligned}
&\models \{x = X\}\, x = x+1;\, \{X < x\} \\
&\Longleftrightarrow \forall I.\forall \sigma.\, \sigma \models^I \mathscr{B}_v[\![x = X]\!]^I_\Gamma \wedge \exists \sigma'.\, (\sigma, \sigma') \in \mathscr{C}[\![x = x+1]\!] \Longrightarrow \sigma' \models^I \mathscr{B}_v[\![X < x]\!]^I_\Gamma && (5.4)\\
&\Longleftrightarrow \forall I.\forall \sigma.\, \sigma(x) = I(X) \wedge \sigma' = \sigma[\sigma(x)+1/x] \Longrightarrow I(X) < \sigma'(x) && (5.5)\\
&\Longleftrightarrow \forall I.\forall \sigma.\, \sigma(x) = I(X) \Longrightarrow I(X) < \sigma(x)+1 && (5.6)\\
&\Longleftrightarrow \forall I.\, I(X) < I(X)+1 && (5.7)
\end{aligned}$$

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

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

In praxis, 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 a motorway at top speed. You almost always want "my program always returns the correct result".

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, or is valid (*i.e.* a program is correct). What is needed are some *syntactic rules* to show the validity of a Hoare triple. In logic, such a set of rules is called a calculus.

The rules of the Floyd-Hoare calculus are given in Figure 5.2. There is one rule for each construct of the language: assignment, case distinction, iteration, sequencing, and the empty statement.

It is perfectly natural (but wrong) to think that the assignment rule should have the substitution in the postcondition. But 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 have the expression e when reading x. By this rule, assignment in the programming language gets translated into substitution in logical propositions. The changes in the state by assigning values to variables are reflected by substituting the corresponding values in the assertions over that state. In other words, program execution is translated to logical manipulation of a formula.

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

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

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

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

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

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

Figure 5.2: The rules of the Floyd-Hoare calculus

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

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

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

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

5.3.1 A Notation for Proofs

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

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

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

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

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

```
// {$P$}
// {$P_1$}
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 5.2.
- 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 as well: whenever there is an assertion following immediately following another assertion, this means the weakening rule is applied.

It follows that programs annotated with linear correctness proofs must have a specific form: it starts with an annotation, followed by possibly more annotation, followed by a sequence of statements, followed by one or more annotations.[3]

Figure 5.3 shows another proof in the linear notation. The principle should be clear by now. 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.

[3]Call the nonterminal for statements C, and for annotations A, then this is described by the grammar $A^+(CA^+)+$.

```
// {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 5.3: A small example of a correctness proof in the linear notation.

5.4 Conclusion

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.

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.

Chapter 6

Finding Invariants and the Correctness of the Floyd-Hoare Calculus

We have seen the rules of the Floyd-Hoare calculus, now let us make them work on slightly larger examples.

6.1 Finding Invariants

Consider the motivating example from Figure 1.2 on page 1.2 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 "actual" 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:

$$p = (c-1)! \tag{6.1}$$

- 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 $c > n$.

  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

$$p = (c-1)! \wedge c-1 \leq n \tag{6.2}$$

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

Figure 6.1: The factorial example (mathematical notation)

```
/** { 1 == factorial(0) && 0 <= n } */
/** { 1 == factorial(1- 1) && 1 <= 1 && 1-1 <= n } */
p= 1;
/** { p == factorial(1- 1) && 1 <= 1 && 1-1 <= n } */
c= 1;
/** { p == factorial(c- 1) && 1 <= c && c-1 <= n } */
while (c<= n) {
  /** { p == factorial(c-1) && 1<= c && c-1 <= n && c <= n } */
  /** { p*c == factorial(c-1)* c && 1 <= c && c <= n } */
  /** { p*c == factorial(c) && 1 <= c && c <= n } */
  /** { p*c == factorial((c+1)- 1) && 1 <= c+1 && (c+1)-1 <= n } */
  p= p*c;
  /** { p == factorial((c+1)-1) && 1<= c+1 && (c+1)-1 <= n } */
  c= c+1;
  /** { p == factorial(c-1) && 1<= c && c-1 <= n } */
  }
/** { p == factorial(c-1) && 1<= c && c-1 <= n && !(c <= n) } */
/** { p == factorial(c-1) && 1<= c && c-1 <= n && c > n } */
/** { p == factorial(c-1) && 1<= c && c-1 == n } */
/** { p == factorial(n) } */
```

Figure 6.2: The factorial example (machine-readable notation)

- We now try to show that the loop body preserves (6.2), 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[1] 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:

$$n! = \begin{cases} 1 & n = 0 \\ (n-1)! \cdot n & n > 0 \end{cases} \tag{6.3}$$

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

$$p = (c-1)! \wedge c-1 \leq n \wedge c > 0 \tag{6.4}$$

Of course, we need to repeat the calculation now that (6.4) 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.

Figures 6.1 and 6.2 show the full proof of the factorial example. (Figure 6.1 uses a mathematical notation for the assertions, which is easier readable, and Figure 6.2 uses a machine-readable notation for assertions, which can be parsed by our tool. These two are completely equivalent, the mathematical notation is just easier to read, so we will use it most of the time.) 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 (6.3); 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 (6.3) (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.

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

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

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

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

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

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

```
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.

6.2 More Examples

Figures 6.3,6.4 and 6.5 show more examples and invariants. Points to note:

- In the second variation in Figure 6.3, we need to initialise c with 0, not with 1.
- In the third variation in Figure 6.3 we do not need $0 \leq y$ because of partial correctness. For $y < 0$, the loop never terminates so the postcondition is vacuous. If we replace the loop condition with $y > 0$, then this would be required as the loop would now terminate immediately with $y > 0$, $y = Y$ (as y has not changed) and $x = 1$, and clearly $x \neq 2^Y$ (consider *e.g.* $y = -1$).
- Figure 6.4 is a variation of 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.
- Figure 6.5 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 $s - t \leq a$.

6.3 Soundness and Completeness of the Floyd-Hoare-Logic

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

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

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

- Does $\vdash \{P\}\,c\,\{Q\}$ imply $\models \{P\}\,c\,\{Q\}$, meaning all Floyd-Hoare triples we derive are correct? This is the *soundness* or *correctness* of the Floyd-Hoare calculus.

```
// $\{0 \leq y\}$
// $\{1 = 2^0 \land 0 \leq y\}$
// $\{1 = 2^{1-1} \land 1-1 \leq y\}$
x= 1;
// $\{x = 2^{1-1} \land 1-1 \leq y\}$
c= 1;
// $\{x = 2^{c-1} \land c-1 \leq y\}$
while (c <= y) {
 // $\{x = 2^{c-1} \land c-1 \leq y \land c \leq y\}$
 // $\{x = 2^{c-1} \land c \leq y\}$
 // $\{2 \cdot x = 2 \cdot 2^{c-1} \land c \leq y\}$
 //
$\{2 \cdot x = 2^{(c+1)-1} \land (c+1)-1 \leq y\}$
 x= 2*x;
 // $\{x = 2^{(c+1)-1} \land (c+1)-1 \leq y\}$
 c= c+1;
 // $\{x = 2^{c-1} \land c-1 \leq y\}$
}
// $\{x = 2^{c-1} \land c-1 \leq y \land \neg(c \leq y)\}$
// $\{x = 2^{c-1} \land c-1 \leq y \land c-1 \geq y\}$
// $\{x = 2^y\}$
```

```
// $\{0 \leq y\}$
// $\{1 = 2^0 \land 0 \leq y\}$
x= 1;
// $\{x = 2^0 \land 0 \leq y\}$
c= 0;
// $\{x = 2^c \land c \leq y\}$
while (c < y) {
 // $\{x = 2^c \land c \leq y \land c < y\}$
 // $\{2 \cdot x = 2 \cdot 2^c \land c < y\}$
 // $\{2 \cdot x = 2^{c+1} \land c+1 \leq y\}$
 c= c+1;
 // $\{2 \cdot x = 2^c \land c \leq y\}$
 x= 2*x;
 // $\{x = 2^c \land c \leq y\}$
}
// $\{x = 2^c \land c \leq y \land \neg(c < y)\}$
// $\{x = 2^c \land c \leq y \land c \geq y\}$
// $\{x = 2^y\}$
```

```
// $\{y = Y \land 0 \leq y\}$
// $\{1 = 2^{Y-y} \land Y = y\}$
x= 1;
// $\{x = 2^{Y-y}\}$
while (y != 0) {
   // $\{x = 2^{(Y-y)} \land y \neq 0\}$
   // $\{2 \cdot x = 2^{(Y-y)+1}\}$
   // $\{2 \cdot x = 2^{Y-(y-1)}\}$
   x= 2*x;
   // $\{x = 2^{Y-(y-1)}\}$
   y= y-1;
   // $\{x = 2^{Y-y}\}$
}
// $\{x = 2^{Y-y} \land \neg(y \neq 0)\}$
// $\{x = 2^{Y-y} \land y = 0\}$
// $\{x = 2^Y\}$
```

Figure 6.3: Computing powers of 2 in three variations.

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

Figure 6.4: Computing the integer quotient and remainder.

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

Figure 6.5: Computing the integer square root.

- Does $\models \{P\}\,c\,\{Q\}$ impy $\vdash \{P\}\,c\,\{Q\}$, meaing when a Floyd-Hoare holds we will be able to derive it? This is the *completeness* of the Floyd-Hoare calculus.

6.3.1 Soundness

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

Theorem 1 (Soundness of Floyd-Hoare logic) *The calculus for the Floyd-Hoare logic is sound:*

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

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

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

Six cases missing.

$\square$

Lemma 3 (Substitution Lemma) *For any state $\sigma \in \Sigma$, assignment I, assertion $B \in$ **Assn**, arithmetic expression $e \in$ **Aexp** and variable $x \in$ **Loc**, we have*

$$sigma \models^I B[e/x] \Longleftrightarrow \sigma[\mathscr{A}[\![e]\!](\sigma)/x] \models^I B \tag{6.5}$$

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

$$\mathscr{A}_v[\![a[e/x]]\!]^I(\sigma) = \sigma[\mathscr{A}_v[\![e]\!]x^I(sigma)/x] \tag{6.6}$$

which is proven by structural induction on a. $\square$

6.4 Conclusion

We have seen how to conduct basic proofs in the Floyd-Hoare calculus — in particular, how to find invariants, which is the hard part — and we have shown important “meta-properties” of the calculus, in particular its soundness.

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

Chapter 7

Structured Datatypes

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

7.1 Datatypes

7.1.1 Arrays

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

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

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

7.1.2 Chars and Strings

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

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

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

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

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

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

7.2 Extending C0

7.2.1 Syntax

To be able to refer to structured datatypes, we need a syntax to express values of this type.[2] This means that was an identifier before can now be a structured value, e.g. denoting an array access or record selection. Array access and record selection are not ordinary operators like addition or subtraction, because they can appear on the left-hand side of an assignment as well. Such values, which semantically denote somewhere where we can store a value, are called lvalues[3] (with l for left-hand side). We introduce a syntactic class **Lexp** for expressions denoting such values, and extend the abstract syntax for C0 from page 2.1 as follows:

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

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

7.2.2 The State

Before we consider the semantics, we need to extend our notion of state as well. We have given a state model in Section 2.2 before: it mapped *locations* to *values* , with the locations being identifiers and values being integers (Definition 1), Now that we have structured addresses, locations need to be more than just identifiers. The C language has a fairly low-level memory model, with addresses being counted in bytes; for the time being, we can be more abstract and talk about locations which are either directly an identifier (as before), or an array access, or a record selection:

Definition 9 (Locations, Values and System State (revisited))

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

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

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

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

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

[2] Note that we do not need a language to write down declarations just yet, but since in C type declaration is supposed to look like type usage, the two are similar.

[3] These should actually be called l-expressions, as they are *syntactic* entities, not semantic ones as the designation "values" might connotate.

7.2.3 Operational Semantics

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

1. first, we need to give a meaning to lvalues, obviously given by locations, and
2. second, we need a meaning for an arithmetic expression which is an lvalue, which is given by accessing the (ambient) state at the location denoted by the lvalue.n

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

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

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

7.2.4 Denotational Semantics

For the denotational semantics, we need to give a meaning to lvalues — and unsurprisingly, these are locations. Figure 7.2 defines the denotational semantics of lvalues. Because the state is now a partial map $\mathbf{Loc} \rightharpoonup \mathbf{V}$, the rule to give meaning to an lvalue as an arithmetic expression needs to change slightly (7.1). The rule to give meaning to assignments also needs slight change (7.2).

7.2.5 Floyd-Hoare Calculus

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

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

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

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

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

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

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

$$\frac{\langle m, \sigma\rangle \rightarrow_{Lexp} l \qquad \langle a, \sigma\rangle \rightarrow_{Aexp} \perp}{\langle m[a], \sigma\rangle \rightarrow_{Lexp} \perp}$$

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

$$\frac{\langle m, \sigma\rangle \rightarrow_{Lexp} l \qquad l \in \mathrm{Dom}(\sigma)}{\langle m, \sigma\rangle \rightarrow_{Aexp} \sigma(l)}$$

$$\frac{\langle m, \sigma\rangle \rightarrow_{Lexp} l \qquad l \notin \mathrm{Dom}(\sigma)}{\langle , \sigma\rangle \rightarrow_{Aexp} \perp}$$

$$\frac{\langle m :: \tau, \sigma\rangle \rightarrow_{Lexp} l \qquad \langle e :: \tau, \sigma\rangle \rightarrow v \qquad \tau \text{ elementary}}{\langle m = e, \sigma\rangle \rightarrow_{Stmt} \sigma[v/l]}$$

Figure 7.1: Rules to evaluate lvalues and expressions

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

$$\begin{aligned}
\mathscr{L}[\![x]\!] &= \{(\sigma, x) \mid \sigma \in \Sigma\} \\
\mathscr{L}[\![m[a]]\!] &= \{(\sigma, l[i]) \mid (\sigma, l) \in \mathscr{L}[\![m]\!], (\sigma, i) \in \mathscr{A}[\![a]\!]\} \\
\mathscr{L}[\![m.i]\!] &= \{(\sigma, m.i) \mid (\sigma, l) \in \mathscr{L}[\![m]\!]\}
\end{aligned}$$

$$\mathscr{A}[\![m]\!] = \{(\sigma, \sigma(l)) \mid \sigma \in \Sigma, (\sigma, l) \in \mathscr{L}[\![m]\!]\} \tag{7.1}$$

$$\mathscr{C}[\![m = e]\!] = \{(\sigma, \sigma[v/l]) \mid (\sigma, l) \in \mathscr{L}[\![m]\!], (\sigma, v) \in \mathscr{A}[\![e]\!]\} \tag{7.2}$$

Figure 7.2: Denotational semantics for lvalues

```
 1  // $\{n \leq 0\}$
 2  // $\{(\forall j. 0 \leq j < 0 \longrightarrow a[j] = j) \wedge 0 \leq n\}$
 3  i= 0;
 4  // $\{(\forall j. 0 \leq j < i \longrightarrow a[j] = j) \wedge i \leq n\}$
 5  while (i< n) {
 6     // $\{(\forall j. 0 \leq j < i \longrightarrow a[j] = j) \wedge i \leq n \wedge i < n\}$
 7     // $\{(\forall j. 0 \leq j < i \longrightarrow a[j] = j) \wedge i = i \wedge i+1 \leq n\}$
 8     // $\{((\forall j. 0 \leq j < i \longrightarrow a[j] = j) \wedge a[i] = i \wedge i+1 \leq n)[a[i]/i]\}$
 9     a[i]= i;
10     // $\{(\forall j. 0 \leq j < i \longrightarrow a[j] = j) \wedge a[i] = i \wedge i+1 \leq n\}$
11     // $\{(\forall j. 0 \leq j < i+1 \longrightarrow a[j] = j) \wedge i+1 \leq n\}$
12     // $\{((\forall j. 0 \leq j < i \longrightarrow a[j] = j) \wedge i \leq n)[i+1/i]\}$
13     i= i+1;
14     // $\{(\forall j. 0 \leq j < i \longrightarrow a[j] = j) \wedge i \leq n\}$
15     }
16  // $\{\forall j. 0 \leq j < n \longrightarrow a[j] = j\}$
```

Figure 7.3: Initialising an array

In line 8 there is a more involved substitution:

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

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

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

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

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

7.3 Example Programs

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

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

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

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

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

7.3.1 Finding the maximum element in an array

Figure 7.4 shows an example of finding the maximum element in an array. This needs the array to be non-empty (precondition in line 1). We use (7.5) from line 20 to 21 and from line 13 to 14. From line 12 to 13, we use transitivity of less-equal to go from $a[j] \leq a[r]$ and $a[r] \leq a[i]$ to $a[j] \leq a[i]$.

7.3.2 Finding a zero element in an array

We consider three variations of finding a zero element in array. We start with Figure 7.5, where the specification is $r \neq -1 \longrightarrow 0 \leq r < n \wedge a[r] = 0$, *i.e.* if the result r is not -1, then it is between 0 and n, and $a[r]$ is zero. We can verify a simple implementation that iterates through the array and finds the first such zero element. The proof needs, going from line 12 to 13, a reverse of *modus ponens*, where we keep the antecedent A:

$$(A \wedge B) \Longrightarrow ((A \longrightarrow B) \wedge A \wedge B)$$

From line 10 to 12, we use some weakening and simple inequality reasoning — specifically, $0 \leq i < n$ implies $i \neq -1$ — to set up this rule. The fact that we drop the implication $r \neq -1 \Longrightarrow 0 \leq < i \wedge a[r] = 0$

```
// $\{0 \le n\}$
// $\{(-1 \neq -1 \longrightarrow 0 \le -1 < 0 \land a[-1] = 0) \land 0 \le 0 \le n\}$
i= 0;
r= -1;
// $\{(r \neq -1 \longrightarrow 0 \le r < i \land a[r] = 0) \land 0 \le i \le n\}$
while (i< n) {
   // $\{(r \neq -1 \longrightarrow 0 \le r < i \land a[r] = 0) \land 0 \le i \le n \land i < n\}$
   // $\{(r \neq -1 \longrightarrow 0 \le r < i \land a[r] = 0) \land 0 \le i < n\}$
   if (a[i] == 0) {
      // $\{(r \neq -1 \longrightarrow 0 \le r < i \land a[r] = 0) \land 0 \le i < n \land a[i] = 0\}$
      // $\{0 \le i < n \land a[i] = 0\}$
      // $\{(i \neq -1 \land 0 \le i \land a[i] = 0) \land 0 \le i+1 \le n\}$
      // $\{(i \neq -1 \longrightarrow 0 \le i \land a[i] = 0) \land 0 \le i+1 \le n\}$
      // $\{(i \neq -1 \longrightarrow 0 \le i < i+1 \land a[i] = 0) \land 0 \le i+1 \le n\}$
      r= i;
      // $\{(r \neq -1 \longrightarrow 0 \le r < i+1 \land a[r] = 0) \land 0 \le i+1 \le n\}$
      }
   else {
      // $\{(r \neq -1 \longrightarrow 0 \le r < i \land a[r] = 0) \land 0 \le i < n \land a[i] \neq 0\}$
      // $\{(r \neq -1 \longrightarrow 0 \le r < i+1 \land a[r] = 0) \land 0 \le i+1 \le n\}$
      }
   // $\{(r \neq -1 \longrightarrow 0 \le r < i+1 \land a[r] = 0) \land 0 \le i+1 \le n\}$
   i= i+1;
   // $\{(r \neq -1 \longrightarrow 0 \le r < i \land a[r] = 0) \land 0 \le i \le n\}$
   }
// $\{(r \neq -1 \longrightarrow 0 \le r < i \land a[r] = 0) \land 0 \le i \le n \land i \ge n\}$
// $\{r \neq -1 \longrightarrow 0 \le r < n \land a[r] = 0\}$
```

Figure 7.5: Finding a zero element in an array by index: weak specification

```
 1  // $\{0 \leq n\}$
 2  // $\{(-1 \neq -1 \longrightarrow 0 \leq -1 < 0 \wedge a[-1] = 0) \wedge (-1 = -1 \longrightarrow \forall j. 0 \leq j < 0 \longrightarrow a[j] \neq 0) \wedge 0 \leq 0 \leq n\}$
 3  i= 0;
 4  r= -1;
 5  // $\{(r \neq -1 \longrightarrow 0 \leq r < i \wedge a[r] = 0) \wedge (r = -1 \longrightarrow \forall j. 0 \leq j < i \longrightarrow a[j] \neq 0) \wedge 0 \leq i \leq n\}$
 6  while (i< n) {
 7    // $\{(r \neq -1 \longrightarrow 0 \leq r < i \wedge a[r] = 0) \wedge (r = -1 \longrightarrow \forall j. 0 \leq j < i \longrightarrow a[j] \neq 0) \wedge 0 \leq i \leq n \wedge i < n\}$
 8    // $\{(r \neq -1 \longrightarrow 0 \leq r < i \wedge a[r] = 0) \wedge (r = -1 \longrightarrow \forall j. 0 \leq j < i \longrightarrow a[j] \neq 0) \wedge 0 \leq i < n\}$
 9    if (a[i] == 0) {
10      // $\{(r \neq -1 \longrightarrow 0 \leq r < i \wedge a[r] = 0) \wedge (r = -1 \longrightarrow \forall j. 0 \leq j < i \longrightarrow a[j] \neq 0) \wedge 0 \leq i < n \wedge a[i] = 0\}$
11      // $\{0 \leq i < n \wedge a[i] = 0\}$
12      // $\{i \neq -1 \wedge 0 \leq i < i+1 \wedge a[i] = 0 \wedge 0 \leq i < n\}$
13      // $\{(i \neq -1 \longrightarrow 0 \leq i < i+1 \wedge a[i] = 0) \wedge (i = -1 \longrightarrow \forall j. 0 \leq j < i+1 \longrightarrow a[j] \neq 0) \wedge 0 \leq i+1 \leq n\}$
14      r= i;
15      // $\{(r \neq -1 \longrightarrow 0 \leq r < i+1 \wedge a[r] = 0) \wedge (r = -1 \longrightarrow \forall j. 0 \leq j < i+1 \longrightarrow a[j] \neq 0) \wedge 0 \leq i+1 \leq n\}$
16      }
17    else {
18      // $\{(r \neq -1 \longrightarrow 0 \leq r < i \wedge a[r] = 0) \wedge (r = -1 \longrightarrow \forall j. 0 \leq j < i \longrightarrow a[j] \neq 0) \wedge 0 \leq i < n \wedge a[i] \neq 0\}$
19      // $\{(r \neq -1 \longrightarrow 0 \leq r < i \wedge a[r] = 0) \wedge (r = -1 \longrightarrow \forall j. 0 \leq j < i \longrightarrow a[j] \neq 0 \wedge a[i] \neq 0) \wedge 0 \leq i < n\}$
20      // $\{(r \neq -1 \longrightarrow 0 \leq r < i+1 \wedge a[r] = 0) \wedge (r = -1 \longrightarrow \forall j. 0 \leq j < i+1 \longrightarrow a[j] \neq 0) \wedge 0 \leq i+1 \leq n\}$
21      }
22    // $\{(r \neq -1 \longrightarrow 0 \leq r < i+1 \wedge a[r] = 0) \wedge (r = -1 \longrightarrow \forall j. 0 \leq j < i+1 \longrightarrow a[j] \neq 0) \wedge 0 \leq i+1 \leq n\}$
23    i= i+1;
24    // $\{(r \neq -1 \longrightarrow 0 \leq r < i \wedge a[r] = 0) \wedge (r = -1 \longrightarrow \forall j. 0 \leq j < i \longrightarrow a[j] \neq 0) \wedge 0 \leq i \leq n\}$
25    }
26  // $\{(r \neq -1 \longrightarrow 0 \leq r < i \wedge a[r] = 0) \wedge (r = -1 \longrightarrow \forall j. 0 \leq j < i \longrightarrow a[j] \neq 0) \wedge 0 \leq i \leq n \wedge i \geq n\}$
27  // $\{(r \neq -1 \longrightarrow 0 \leq r < n \wedge a[r] = 0) \wedge (r = -1 \longrightarrow \forall j. 0 \leq j < n \longrightarrow a[j] \neq 0)\}$
```

Figure 7.6: Finding a zero element in an array: complete specification

from the invariant (line 10 to 11) corresponds to the fact that in the positive branch of the conditional, we will overwrite the value r in line 15 anyway.

But this specification is (too) weak. It does not allow us to conclude that if the program returns $r = -1$, there is no zero element; in fact, the simple assignment `r=` -1 satisfies the specification, because *false* implies everything (*ex falso quodlibet*):

```
// $\{true\}$
// $\{false \longrightarrow 0 \leq r < n \wedge a[r] = 0\}$
// $\{-1 \neq -1 \longrightarrow 0 \leq r < n \wedge a[r] = 0\}$
r= -1;
// $\{r \neq -1 \longrightarrow 0 \leq r < n \wedge a[r] = 0\}$
```

To strengthen the specficiation, we also need to say what holds in case of $r = -1$— that all elements $a[j]$ are not zero:

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

We strengthen the specification accordingly in Figure 7.6. The proof is a bit more involved in the positive branch of the conditional. Firstly, from line 10 to 11, we drop the part of the invariant which makes a statement about r (as in the proof in Figure 7.5 above). We then use the *modus ponens* going from line 12 to 13 to conclude $i \neq -1 \longrightarrow a[i] = 0$ from $i \neq -1 \land a[i] = 0$, but we also use a proof by contradiction (*reductio ad absurdum*)

$$A \Longrightarrow (\neg A \longrightarrow B)$$

to conclude $i = -1 \longrightarrow \forall j. \ldots$ from $\neg(i = -1)$. Further, we make use of (7.5) in line 19 to 20.

As a final variation, the reader is invited to change both program and specification such that the program finds the *smallest* index of a zero element.

7.4 Conclusions

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

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

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

Chapter 9

Verification Condition Generation

Consider the following simple program fragment (taken from the integer square root program on page 6.5):

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

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

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

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

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

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

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

As we can see, all we need is the postcondition, and we can derive a precondition from that. This works, because the assignment rule (as given in Figure 5.2 on page 5.2) has an "open" precondition, *i.e.* the schematic rule is

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

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

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

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

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

9.1 Reasoning Backwards

Looking at the other rules (in Figure 5.2, we can see the following:

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

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

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

which written as the single rule is

$$\frac{A \wedge b \Longrightarrow B \quad \vdash \{B\}\, c\, \{A\} \quad A \wedge \neg b \Longrightarrow C}{\vdash \{A\}\ \textbf{while}\ (b)\ c\, \{C\}} \tag{9.1}$$

This rule always matches any postcondition, but we need to prove the two implications in the precondition — these are our *verification conditions* — and we need to get the invariant A from somewhere (it is not given by the precondition). This is to be expected: finding the invariant is the creative (read: difficult) part of proofs in the Floyd-Hoare logic, and we could not reasonably expect any automation to alleviate us of this burden. Formally, we can not give a finite formula in the general case (a construction can be given using Gödel's β-predicate, see *e.g.* [9]).

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

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

With this, we can write the while-rule as

$$\frac{I \wedge b \Longrightarrow B \quad \vdash \{B\}\, c\, \{A\} \quad I \wedge \neg b \Longrightarrow C}{\vdash \{I\}\ \textbf{while}\ (b)\ /{*}{*}\,\textbf{inv}\ I\ {*}/\ c\, \{C\}} \tag{9.2}$$

The implications of rule (9.2) become the *verification conditions*. Verification conditions are state-free formula of first-order logic with induction, which represent the logical reasoning underlying the program correctness. They are proven by an external prover.

The astute reader may have noticed the weakening $I \wedge b \Longrightarrow B$ in the premise of rule (9.2). What is this for? If we apply a rule with premisses — such as the while-rule, or the if-rule — and it matches on the postcondition, it will generate new Hoare triples which we have to prove, as given by the premisses of the rule. For the premisses, the postcondition does not need to be schematic — it is given by instantiating the postcondition of the conclusion — but the precondition needs be schematic, as it is instantiated by proving the newly occuring Hoare triples.

This is also the reason why the if-rule as given in Figure 5.2 is not entirely suitable for backwards reasoning:

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

To use this would require that the derived precondition of the positive branch (c_0) and the negative branch (c_1) are $A \wedge b$ and $A \wedge \neg b$, respectively; but we cannot be sure of this. We need a rule where the precondition of both premisses is arbitrary. By composing with weakening (again), we get such a rule. For this, we need a bit of boolean formula gymnastics. In general, given propositions A_0, A_1 and B, we have

$$(A_0 \wedge B) \vee (A_1 \wedge \neg B) \wedge B \Longleftrightarrow (A_0 \wedge B \wedge B) \vee (A_1 \wedge \neg B \wedge B) \Longleftrightarrow (A_0 \wedge B) \vee \mathit{false} \Longleftrightarrow A_0 \wedge B \tag{9.3}$$

and similary, $(A_0 \wedge B) \vee (A_1 \wedge \neg B) \wedge \neg B \Longleftrightarrow A_1 \wedge \neg B$. This suggests that if the two preconditions of the positive and negative branch are A_0 and A_1, respectively, to use $A_0 \wedge B) \vee (A_1 \wedge \neg B$ as the derived weakest precondition for backwards reasoning, because we get the derived rule (where the two equivalences for the weakening rules are given by (9.3)), and hence do not need to be proven:

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

which as a single rule reads

$$\frac{\vdash \{A_0\}\, c_0\, \{B\} \quad \vdash \{A_1\}\, c_0\, \{B\}}{\vdash \{(A_0 \wedge b) \vee (A_1 \wedge \neg b)\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}.} \tag{9.4}$$

9.2 Weakest Preconditions

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

Definition 10 *Weakest Precondition Given an assertion $Q \in$ **Assn** and a statement $c \in$ **Stmt**, the* weakest precondition *is the assertion $P \in$ **Assn** such that*

$$\models \{P\}\, c\, \{Q\} \Longleftrightarrow P \Longrightarrow \mathrm{wp}(c, Q) \tag{9.5}$$

This is equivalent to the more obvious

$$\models \{\mathrm{wp}(c, Q)\}\, c\, \{Q\} \tag{9.6}$$

$$\models \{P\}\, c\, \{Q\} \Longrightarrow (P \Longrightarrow \mathrm{wp}(c, Q)) \tag{9.7}$$

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

As mentioned above, we cannot hope to calculate the weakest precondition as a finite formula in general, because of the while-rule. However, with our trick of *annotating* the program with the invariants we can compute the *approximative weakest precondition* along with a set of verification conditions which have to proven separately. This formalises the process from above.

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

Figure 9.1: The approximative weakest precondition and verification conditions

9.2.1 Approximative Weakest Preconditions

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

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

Here, $\mathrm{awp}(c,Q)$ is the approximative weakest precondition and $\mathrm{wvc}(c,Q)$ the set of verification condition. We define these functions by induction on the structure of the statement, see Figure 9.1. Together, they satisfy (9.6), but not (9.7), in the sense that if all verification conditions (or equivalently, their conjunction) hold, then the approximative weakest precondition is a precondition for the statement c and the postcondition Q:

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

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

The definition in Figure 9.1 can be read off the Floyd-Hoare rules, taking into account the derived rules for case distinction (9.4) and iteration (9.2). For example, the while-rule generates two verification conditions — the weakenings. The awp is given by the annotated invariant, and the recursively calculated awp is used in the verification conditions.

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

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

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

Figure 9.2: Factorial function with annotated invariant

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

9.3 Examples

When calculating the awp and the verification conditions, we use the notation awp_n for the awp generated for line n (*i.e.* the precondition of line n and postcondition of line $n-1$), and vc_n for a verification condition generated at line n. Figure 9.2 is the by now well-known factorial example, with the invariant annotated. Here are the resulting awps and verification conditions:

$$\begin{aligned}
awp_6 &: p = ((c+1)-1)! \wedge ((c-1)+1) \leq n\\
awp_5 &: p \cdot c = ((c+1)-1)! \wedge ((c-1)+1) \leq n\\
awp_3 &: p = (1-1)! \wedge (1-1) \leq n\\
awp_2 &: 1 = (1-1)! \wedge (1-1) \leq n
\end{aligned}$$

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

Verification conditions can get very lengthy. Figure 9.3 is the program which finds the maximal element in an array from Figure 7.4, with invariant annotated as required. We can see that the actual source code is much shorter, and the good news is that from that source code we can actually derive all information which in Figure 7.4 had to be painstakingly annotated by hand. But this comes with a price in the form of lengthy verification conditions:

$$\begin{aligned}
awp_{11} &: (\forall j. 0 \leq j < i+1 \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n\\
awp_{10} &: (\forall j. 0 \leq j < i+1 \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n\\
awp_7 &: (\forall j. 0 \leq j < i+1 \longrightarrow a[j] \leq a[i]) \wedge 0 \leq i < n\\
awp_6 &: ((\forall j. 0 \leq j < i+1 \longrightarrow a[j] \leq a[i]) \wedge 0 \leq i < n \wedge a[r] < a[i]) \vee\\
&\quad ((\forall j. 0 \leq j < i+1 \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n \wedge \neg(a[r] < a[i]))
\end{aligned}$$

```
// {$0 < n$}
i= 0;
r= 0;
while (i< n) /** inv $(\forall j. 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n$ */
 {
   if (a[r] < a[i]) {
      r= i;
      }
   else {
      }
   i= i+1;
   }
// {$(\forall j. 0 \leq j < n \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n$}
```

Figure 9.3: Finding the maximal element in an array

Chapter 10

Forwards with Floyd-Hoare!

10.1 Examples

10.1.1 The Factorial Example

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

We first calculate the strongest postconditions, simplifying them as far as possible on-the-fly:

$$
\begin{array}{ll}
asp_2 & \exists V. 0 \le n[V/p] \wedge p = (1[V/p]) \\
 & \rightsquigarrow \exists V. \le n \wedge p = 1 \\
 & \rightsquigarrow 0 \le n \wedge p = 1 \\
asp_3 & \exists V. (0 \le n \wedge p = 1)[V/c] \wedge c = (1[V/c]) \\
 & \rightsquigarrow \exists V. 0 \le n \wedge p = 1 \wedge c = 1 \\
 & \rightsquigarrow 0 \le n \wedge p = 1 \wedge c = 1 \\
asp_4 & \neg(c \le n) \wedge p = (c-1)! \wedge c-1 \le n \\
asp_5 & \exists V_1. (p = (c-1)! \wedge (c-1) \neq n \wedge c \le n)[V_1/p] \wedge p = (p \cdot c[V_1/p] \\
 & \rightsquigarrow \exists V_1. V_1 = (c-1)! \wedge (c-1) \neq n \wedge c \le n \wedge p = V_1 \cdot c \\
 & \rightsquigarrow c-1 \le n \wedge c \le n \wedge p = (c-1)! \cdot c \\
asp_6 & \exists V_2. (c-1 \le n \wedge c \le n \wedge p = (c-1)! \cdot c)[V_2/c] \wedge c = (c+1)[V_2/c] \\
 & \rightsquigarrow \exists V_2. V_2 - 1 \le n \wedge V_2 \le n \wedge p = (V_2-1)! \cdot V_2 \wedge c = V_2 + 1 \\
 & \rightsquigarrow \exists V_2. V_2 - 1 \le n \wedge V_2 \le n \wedge p = (V_2-1)! \cdot V_2 \wedge V_2 = c - 1 \\
 & \rightsquigarrow c-2 \le n \wedge c-1 \le n \wedge p = (c-2)! \cdot (c-1)
\end{array}
$$

Notes:

- To simplify asp_5, we use the equational form of the invariant ($p = \ldots$), which by substitution turns into $V_1 = \ldots$.

- To simplfiy asp_6, we rephrase $c = V_2 + 1$ into $V_2 = c - 1$ and use that equation. We also use some elementary arithmetic $(c-1) - 1 = c - 2$, but not the recursive definition of the factorial $(n-1)! \cdot n = n!$, as this is not part of the simplification.

We then compute the verification conditions:

$$\begin{array}{ll} vc_4 & \{asp_3 \longrightarrow p = (c-1)! \wedge c - 1 \leq n,\ asp_6 \longrightarrow p = (c-1)! \wedge c - 1 \leq n\} \\ vc_8 & asp_4 \longrightarrow p = n! \end{array}$$

Proving the verification conditions is actually pretty straightforward, but needs a lemma: when the precondition is an equation of the form $x = t$, then we can substitute all x occuring the conclusion with t:

$$(x = t \longrightarrow P) \Longrightarrow P[t/x]$$

The verification conditions vc_4 are two implications with a conjuction as conclusion, so this simplifies to four separate implications, whereas the reasoning for vc_8 is by now familiar:

$$\begin{array}{l} vc_{4.1} : asp_3 \longrightarrow p = (c-1)! \\ \quad \Longleftrightarrow 0 \leq n \wedge p = 1 \wedge c = 1 \longrightarrow p = (c-1)! \\ \quad \Longleftrightarrow 0 \leq n \longrightarrow 1 = (1-1)! \\ \quad \Longleftrightarrow 0 \leq n \longrightarrow \mathit{true} \\ vc_{4.2} : asp_3 \longrightarrow c \leq n \\ \quad \Longleftrightarrow 0 \leq n \wedge p = 1 \wedge c = 1 \longrightarrow c - 1 \leq n \\ \quad \Longleftrightarrow 0 \leq n \longrightarrow 1 - 1 \leq n \\ \quad \Longleftrightarrow 0 \leq n \longrightarrow 0 \leq n \\ \quad \Longleftrightarrow \mathit{true} \\ vc_{4.3} : asp_6 \longrightarrow p = (c-1)! \\ \quad \Longleftrightarrow c - 2 \leq n \wedge c - 1 \leq n \wedge p = (c-2)! \cdot (c-1) \longrightarrow p = (c-1)! \\ \quad \Longleftrightarrow c - 1 \leq n \wedge p = (c-2)! \cdot (c-1) \longrightarrow p = (c-1)! \\ \quad \Longleftrightarrow \mathit{true} \\ vc_{4.4} : asp_6 \longrightarrow c - 1 \leq n \\ \quad \Longleftrightarrow c - 2 \leq n \wedge c - 1 \leq n \wedge p = (c-2)! \cdot (c-1) \longrightarrow c - 1 \leq n \\ \quad \Longleftrightarrow c - 1 \leq n \longrightarrow c - 1 \leq n \\ \quad \Longleftrightarrow \mathit{true} \\ \quad vc_8 : asp_4 \longrightarrow p = n! \\ \quad \Longleftrightarrow \neg(c \leq n) \wedge p = (c-1)! \wedge c - 1 \leq n \longrightarrow p = n! \\ \quad \Longleftrightarrow n \leq c - 1 \wedge c - 1 \leq n \wedge p = (c-1)! \longrightarrow p = n! \\ \quad \Longleftrightarrow n = c - 1 \wedge p = (c-1)! \longrightarrow p = n! \\ \quad \Longleftrightarrow \mathit{true} \end{array}$$

10.1.2 Finding the Maximum Element

First attempt (as before):

```
// {0 < n}
i= 0;
r= 0;
while (i != n) /** inv (∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n */ {
  if (a[r] < a[i]) {
    r= i;
  }
  else {
  }
  i= i+1;
}
// {(∀j.0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Strongest preconditions:

$$\begin{aligned}
asp_3 &= 0 < n \land i = 0 \land r = 0\\
asp_9 &= (\exists V_1.\,(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[V_1]) \land 0 \leq V_1 < n \land i \neq n \land a[V_1] < a[i] \land r = i) \lor\\
&\quad ((\forall j.0 \leq j < i \longrightarrow a[j] < a[r]) \land 0 \leq r < n \land i \neq n \land a[i] \leq a[r])\\
asp_{10} &= \exists V_2.\, asp_9[V_2/i] \land i = V_2 + 1\\
&= (\exists V_1.\,(\forall j.0 \leq j < i-1 \longrightarrow a[j] \leq a[V_1]) \land i-1 \neq n \land a[V_1] < a[i-1] \land r = i-1) \lor\\
&\quad ((\forall j.0 \leq j < i-1 \longrightarrow a[j] < a[r]) \land 0 \leq r < n \land i-1 \neq n \land a[i-1] \leq a[r])
\end{aligned}$$

Verification conditions (excerpt, the final one is missing but as before):

$$\begin{aligned}
vc_{4.1} &= asp_3 \longrightarrow (\forall j.0 \leq j < i \longrightarrow a[j] < a[r]) \land 0 \leq r < n\\
&= 0 < n \longrightarrow (\forall j.0 \leq j < 0 \longrightarrow a[j] < a[0]) \land 0 \leq 0 < n\\
&= 0 < n \longrightarrow 0 < n\\
vc_{4.2} &= asp_{10} \longrightarrow (\forall j.0 \leq j < i \longrightarrow a[j] < a[r]) \land 0 \leq r < n
\end{aligned}$$

Consider $vc_{4.2}$ in more detail: to show $A \lor B \longrightarrow C_1 \land C_2$, we have to show $A \longrightarrow C_1, A \longrightarrow C_2, B \longrightarrow C_1$ and $B \longrightarrow C_2$, hence $vc_{4.3}$ decomposes into:

$$(\exists V_1.\,(\forall j.0 \leq j < i-1 \longrightarrow a[j] \leq a[V_1]) \land i-1 \neq n \land a[V_1] < a[i-1] \land r = i-1) \longrightarrow \forall j.0 \leq j < i \longrightarrow a[j] < a[r]$$
$$(\exists V_1.\,(\forall j.0 \leq j < i-1 \longrightarrow a[j] \leq a[V_1]) \land i-1 \neq n \land a[V_1] < a[i-1] \land r = i-1) \longrightarrow 0 \leq r < n$$
$$(\forall j.0 \leq j < i-1 \longrightarrow a[j] < a[r]) \land 0 \leq r < n \land i-1 \neq n \land a[i-1] \leq a[r] \longrightarrow \forall j.0 \leq j < i \longrightarrow a[j] < a[r]$$
$$(\forall j.0 \leq j < i-1 \longrightarrow a[j] < a[r]) \land 0 \leq r < n \land i-1 \neq n \land a[i-1] \leq a[r] \longrightarrow 0 \leq r < n$$

Of these, the first one and the last two can be shown, but the second one is a problem: to show $0 \leq r < n$, we need $0 \leq i-1 < n$ (with $r = i-1$), but how can we show that? So going forward, we really need to strengthen the invariant to include that $0 \leq i$ and $i < n$; the latter can be achieved by changing the loop condition, but then we need $i \leq n$ as invariant (note $i < n$ is not an invariant) to show that after the loop $i = n$.

```
// $\{0 < n\}$
i= 0;
r= 0;
while (i< n) /** inv $(\forall j. 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n \wedge 0 \leq i \leq n$ */ {
  if (a[r] < a[i]) {
    r= i;
    }
  else {
    }
  i= i+1;
  }
// $\{(\forall j. 0 \leq j < n \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n\}$
```

With this, we get the following strongest preconditions:

$$\begin{aligned}
asp_9 &= (\exists V_1.(\ldots) \wedge 0 \leq V_1 < n \wedge 0 \leq i < n \wedge a[V_1] < a[i] \wedge r = i) \vee \\
&\quad ((\ldots) \wedge 0 \leq r < n \wedge 0 \leq i < n \wedge a[i] \leq a[r]) \\
asp_{10} &= \exists V_2.\, asp_9[V_2/i] \wedge i = V_2 + 1 \\
&= (\exists V_1.(\forall j. 0 \leq j < i-1 \longrightarrow a[j] \leq a[V_1]) \wedge 0 \leq i-1 < n \wedge a[V_1] < a[i-1] \wedge r = i-1) \vee \\
&\quad ((\forall j. 0 \leq j < i-1 \longrightarrow a[j] < a[r]) \wedge 0 \leq r < n \wedge 0 \leq i-1 < n \wedge a[i-1] \leq a[r])
\end{aligned}$$

We get a new verification condition $vc_{4.2}$, which decomposes to:

$$\begin{aligned}
&(\exists V_1.(\forall j. 0 \leq j < i-1 \longrightarrow a[j] \leq a[V_1]) \wedge 0 \leq i-1 < n \wedge a[V_1] < a[i-1] \wedge r = i-1) \longrightarrow \forall j. 0 \leq j < i \longrightarrow a[j] < a[r] \\
&(\exists V_1.(\forall j. 0 \leq j < i-1 \longrightarrow a[j] \leq a[V_1]) \wedge 0 \leq i-1 < n \wedge a[V_1] < a[i-1] \wedge r = i-1) \longrightarrow 0 \leq i \\
&(\exists V_1.(\forall j. 0 \leq j < i-1 \longrightarrow a[j] \leq a[V_1]) \wedge 0 \leq i-1 < n \wedge a[V_1] < a[i-1] \wedge r = i-1) \longrightarrow 0 \leq r < n \\
&\qquad \forall j. 0 \leq j < i-1 \longrightarrow a[j] < a[r]) \wedge 0 \leq r < n \wedge 0 \leq i-1 < n \wedge a[i-1] \leq a[r] \longrightarrow \forall j. 0 \leq j < i \longrightarrow a[j] < a[r] \\
&\qquad \forall j. 0 \leq j < i-1 \longrightarrow a[j] < a[r]) \wedge 0 \leq r < n \wedge 0 \leq i-1 < n \wedge a[i-1] \leq a[r] \longrightarrow 0 \leq i \\
&\qquad \forall j. 0 \leq j < i-1 \longrightarrow a[j] < a[r]) \wedge 0 \leq r < n \wedge 0 \leq i-1 < n \wedge a[i-1] \leq a[r] \longrightarrow 0 \leq r < n
\end{aligned}$$

Now, the previously problematic third verification condition proves easily. The new verification conditions (second and fifth) are easy as well ($0 \leq i-1$ implies $0 \leq i$), and the first and third — the ones containing the meat of the argument, so to speak — are as before.

Bibliography

[1] D. Burke. All circuits are busy now: The 1990 AT&T long distance network collapse. Technical Report CSC440-01, California Polytechnic Stae University, Nov. 1995. `http://users.csc.calpoly.edu/~jdalbey/SWE/Papers/att_collapse.html`.

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

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

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

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

[6] J.-L. Lions. Ariane 5 flight 501 failure — report by the enquiry board. Technical report, Ariane 501 Inquiry Board, July 19th 1996. `http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf`.

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

[8] D. van Dalen. *Logic and Structure*. Springer Verlag, 4 edition, 2004.

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