CIS 301: Lecture Notes on Program Verification

Torben Amtoft
Department of Computing and Information Sciences
Kansas State University

February 26, 2008

These notes are written as a supplement to [1, Sect. 16.5], but can be read independently. Section 6 is inspired by Chapter 16 in [4], an excellent treatise on the subject of program construction; also our Section 8 is inspired by that book. The proof rules in Section 7 are inspired by the presentation in [5, Chap. 4]. Section 10 is inspired by [3].

Contents

1 Hoare Triples

To reason about correctness we shall consider *Hoare triples*, of the form

$$\begin{array}{l}\{\phi\}\\ \quad P\\ \{\psi\}\end{array}$$

saying that if ϕ (the *precondition*) holds prior to executing program code P then ψ (the *postcondition*) holds afterwards. Here ϕ and ψ are *assertions*, written in First Order Logic.

Actually, the above description is ambiguous: what if P does not terminate? Therefore we shall distinguish between

partial correctness: *if* P terminates *then* ψ holds;

total correctness: P *does* terminate *and then* ψ holds.

In these notes, we shall interpret a Hoare triple as denoting *partial correctness*, unless stated otherwise.

2 Software Engineering

In light of the notion of Hoare triples, one can think of software engineering as a 3-stage process:

1. *Translate* the demands D of the user into a *specification* (ϕ_D, ψ_D).
2. *Write* a program P that satisfies the specification constructed in 1.
3. *Prove* that in fact it holds that

$$\begin{array}{l}\{\phi_D\}\\ \quad P\\ \{\psi_D\}\end{array}$$

When it comes to software practice, 1 is a huge task (involving numerous discussions with the users) and hardly ever done completely, whereas 2 obviously has to be done. Until recently, 3 was almost never carried out, but the current trend in industry is towards increased use of formal verification. A noteworthy example of this is the ASTREE system[1] which has been used to prove, completely automatically, the absence of certain kind of errors in the flight control software of the Airbus A340 and A380 airplanes.

When it comes to academic discourse, 1 is an interesting task but only briefly touched upon (Section 3) in CIS 301. Instead, we shall focus on 3 (Sections 5 and 7), but also give a few basic heuristics for how to do 2 and 3 *simultaneously* (Section 6).

3 Specifications

3.1 Square root

Suppose the user demands

Compute the square root of `x` and store the result in `y`.

As a first attempt, we may write the specification

$$\begin{array}{l} \quad P \\ \{\mathtt{y}^2 = \mathtt{x}\} \end{array}$$

We now remember that we cannot compute the square root of negative numbers and therefore add a precondition:

$$\begin{array}{l} \{\mathtt{x} \geq 0\} \\ \quad P \\ \{\mathtt{y}^2 = \mathtt{x}\} \end{array}$$

Then we realize that if `x` is not a perfect square then we have to settle for an approximation (since we are working with integers):

$$\begin{array}{l} \{\mathtt{x} \geq 0\} \\ \quad P \\ \{\mathtt{y}^2 \leq \mathtt{x}\} \end{array}$$

[1]ASTREE has a web page at `http://www.astree.ens.fr`, and a description of its core was published in [2].

On the other hand, this is too liberal: we could just pick y to be zero. Thus, we must also specify that y is the largest number that does the job:

$$\begin{array}{l}\{\mathtt{x} \geq 0\} \\ \quad P \\ \{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 > \mathtt{x}\}\end{array}$$

which seems a sensible specification of the square root program. (Which entails that y has to be non-negative. Why?)

3.2 Factorial

Now assume that the user demands

> Ensure that y contains the factorial[2] of x.

This might give rise to the specification

$$\begin{array}{l}\{\mathtt{x} \geq 0\} \\ \quad P \\ \{\mathtt{y} = \mathrm{fac}(\mathtt{x})\}\end{array}$$

Well, it's not hard to write a program satisfying this specification:

$$\begin{array}{l}\{\mathtt{x} \geq 0\} \\ \quad \mathtt{x} := 4; \\ \quad \mathtt{y} := 24 \\ \{\mathtt{y} = \mathrm{fac}(\mathtt{x})\}\end{array}$$

The user may respond:

> Hey, that's cheating! You were not allowed to modify x.

[2]Remember that the factorial function is defined by

$$\begin{array}{rcl}\mathrm{fac}(0) & = & 1 \\ \mathrm{fac}(n+1) & = & (n+1)\mathrm{fac}(n) \text{ for } n \geq 0\end{array}$$

and thus fac(0) = 1, fac(1) = 1, fac(2) = 2, fac(3) = 6, fac(4) = 24, etc.

Well, if not, that better has to be part of the specification! But how to incorporate such demands?

One approach is to augment specifications with information about which identifiers[3] are allowed to be modified; in the above case, we would exclude `x` from that set. Another approach is to allow specifications to contain *logical* variables[4]: using the logical variable x_0 to denote the initial (and un-changed) value of the identifier `x`, a program computing factorials can be specified as follows:

$$\begin{array}{l}\{\mathtt{x} = x_0 \geq 0\}\\ \quad P\\ \{\mathtt{y} = \mathrm{fac}(x_0) \wedge \mathtt{x} = x_0\}\end{array}$$

Likewise, the specification of the square root program can be augmented so as to express that `x` must not be modified.

4 A Simple Language

For the next sections, we shall consider programs P written in a simple language, omitting[5] many desirable language features—such as procedures, considered in Section 9, and arrays, considered in Section 8. A program P is (so far) just a command, with the syntax of commands given by[6]

$$\begin{array}{rcl} C & ::= & x := E \\ & \mid & C_1;\ C_2 \\ & \mid & \mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2\ \mathtt{fi} \\ & \mid & \mathtt{while}\ B\ \mathtt{do}\ C\ \mathtt{od} \end{array}$$

[3]We shall use the term "identifier" for what is often called a "program variable", so as to avoid confusion with the variables of First Order Logic. To further facilitate that distinction, we shall always write identifiers in typewriter font.

[4]We shall write logical variables with a subscript, so as to emphasize that they do not occur in programs.

[5]Still, our language is "Turing-complete" in that it can *encode* all other features one can imagine!

[6]Note that for an assignment we use the symbol ":=", rather than the "=" used in many programming languages, so as to emphasize its directional nature. Also note that we need a *delimiter* for conditionals since otherwise a command `if` B `then` C_1 `else` C_2; C_3 would be ambiguous (is C_3 part of the "else" branch or not?); we shall use `fi` for that purpose since curly brackets are already in use for writing pre- and post-conditions. Similarly, we use `od` as a delimiter for loops.

Programs are thus constructed from

- *assignments* of the form $x := E$, the effect of which is to store the value of E in x;
- *sequential compositions* of the form $C_1;\ C_2$, the effect of which is to first execute C_1 and next execute C_2;
- *conditionals* of the form `if` B `then` C_1 `else` C_2 `fi`, the effect of which is to execute C_1 if B evaluates to true, but execute C_2 if B evaluates to false;
- *while loops* of the form `while` B `do` C `od`, the effect of which is to iterate executing C as long as B evaluates to true[7].

We have employed some auxiliary syntactic constructs:

- x stands for *identifiers* like `x`, `y`, `z`, etc;
- E stands for *integer expressions* of the form n (a constant), x (an identifier), $E_1 + E_2$, $E_1 - E_2$, etc;
- B stands for *boolean tests* of the form $E_1 < E_2$, $E_1 \leq E_2$, $E_1 \neq E_2$, etc.

Next, we shall discuss how to verify a claim that

$$\begin{array}{l}\{\phi\}\\ \quad P\\ \{\psi\}\end{array}$$

5 Loop Invariants

For the purpose of verification, the notion of *loop invariants* is crucial.

[7]Note that C may never be executed, if B is initially false, and that the loop may never terminate, if B always evaluates to true.

5.1 Motivating Example

We look at the following program for computing the factorial function; the program does not modify the value of x so we can safely write its specification without employing a logical variable x_0, cf. Section 3.

```
{x ≥ 0}
    y := 1;
    z := 0;
    while z ≠ x do
        z := z + 1;
        y := y * z
    od
{y = fac(x)}
```

The core of the above program is a loop, whose

- *test* is given by $\mathtt{z} \neq \mathtt{x}$;
- *body* is given by $\mathtt{z} := \mathtt{z} + 1;\ \mathtt{y} := \mathtt{y} * \mathtt{z}$;
- *preamble* is given by $\mathtt{y} := 1;\ \mathtt{z} := 0$.

There are many mistakes we could have made when writing that program: for instance we could have reversed the two lines in the loop body (in which case y would be assigned zero and keep that value forever), or we could have written the loop test as $\mathtt{z} \leq \mathtt{x}$ (in which case y would end up containing $\text{fac}(x+1)$).

Let us now convince ourselves that what we wrote is correct. We might first try a simulation: if say $x = 4$, the situation at the entry of the loop is:

	x	y	z
After 0 iterations	4	1	0
After 1 iterations	4	1	1
After 2 iterations	4	2	2
After 3 iterations	4	6	3
After 4 iterations	4	24	4

and then $\mathtt{z} = \mathtt{x}$ so that the loop terminates, with y containing the desired result $24 = \text{fac}(\mathtt{x})$. This may boost our confidence in the program, but still a general proof is needed. Fortunately, the table above may help us in that endeavor. For it suggests that it is always the case that $\mathtt{y} = \text{fac}(\mathtt{z})$.

Definition 5.1. A property which holds each time the loop test is evaluated is called an *invariant* for the loop. □

We now annotate the program with our prospective loop invariant:

```
{x ≥ 0}
    y := 1;
    z := 0;
{y = fac(z)}
    while z ≠ x do
        z := z + 1;
        y := y * z
{y = fac(z)}
    od
{y = fac(x)}
```

Of course, we must prove that what we have found is indeed a loop invariant:

Proposition 5.2. *Whenever the loop entry is reached, it holds that* $\mathtt{y} = \mathrm{fac}(\mathtt{z})$. □

The proof has two parts:

- Establishing the invariant;
- Maintaining the invariant.

Establishing the invariant. We must check that when the loop entry is first reached, it holds that $\mathtt{y} = \mathrm{fac}(\mathtt{z})$. But since the preamble assigns `y` the value 1 and assigns `z` the value 0, the claim follows from the fact that $\mathrm{fac}(0) = 1$.

Maintaining the invariant. Next we must check that if $\mathtt{y} = \mathrm{fac}(\mathtt{z})$ holds *before* an iteration then it also holds *after* the iteration. With $\mathtt{y}'$ denoting the value of `y` *after* the iteration, and $\mathtt{z}'$ denoting the value of `z` *after* the iteration, this follows from the following calculation:

$$\mathtt{y}' = \mathtt{y}\mathtt{z}' = \mathtt{y}(\mathtt{z} + 1) = \mathrm{fac}(\mathtt{z})(\mathtt{z} + 1) = \mathrm{fac}(\mathtt{z} + 1) = \mathrm{fac}(\mathtt{z}').$$

For the third equality we have used the assumption that the invariant holds *before* the iteration, and for the fourth equality we have used the definition of the factorial function.

Completing the correctness proof. We have shown that every time the loop test is evaluated, it holds that $\mathtt{y} = \mathrm{fac}(\mathtt{z})$. If (when!) we eventually exit the loop then the loop test is `false`, that is[8] $\mathtt{z} = \mathtt{x}$. Therefore, if (when) the program terminates it holds that $\mathtt{y} = \mathrm{fac}(\mathtt{x})$. This shows that our program satisfies its specification, in that partial correctness holds (cf. Section 1). Moreover, in this case total correctness is not hard to prove: since `x` is initially[9] non-negative, and since `z` is initialized to zero and incremented by one at each iteration, eventually `z` will equal `x`, causing the loop to terminate.

5.2 Proof Principles for Loop Invariants

From the previous subsection we see that three steps are involved when proving that a certain property ψ is indeed a useful invariant for a loop:

1. we must show that the code before the loop establishes ψ;
2. we must show that ψ is maintained after each iteration;
3. we must show that if the loop test evaluates to `false`, ψ is sufficient to establish the desired postcondition.

5.3 Proof Principles for Loop Termination

In general, termination of a loop can be proved by exhibiting a *termination function*: an integer expression E_t such that *(i)* E_t is never negative; *(ii)* the value of E_t decreases for each iteration. To see that this ensures termination, let the initial value of E_t be c; then the loop will terminate after c iterations or less. For assume otherwise, that $c+1$ iterations are performed: then the value of E_t will become negative, due to (ii), which is impossible, due to (i). For the program in Section 5.1, it is easy to see that a suitable termination function is given as $\mathtt{x} - \mathtt{z}$.

[8]If the loop test had been $\mathtt{z} < \mathtt{x}$ rather than $\mathtt{z} \neq \mathtt{x}$, at termination we would only have $\mathtt{z} \geq \mathtt{x}$, so in order to deduce the desired $\mathtt{z} = \mathtt{x}$, we would have to add to the invariant the fact that $\mathtt{z} \leq \mathtt{x}$.

In general, it is often easier to prove the correctness of a loop it its test is expressed using "$\neq$" rather than "$\leq$". On the other hand, when working with *floating points* (instead of integers) one should *never* use tests containing "$=$" or "$\neq$"!

[9]It is interesting that the proof of partial correctness does not use the precondition $\mathtt{x} \geq 0$.

6 Developing a Correct Program

In Section 5, we considered the situation where we must prove the correctness of a program which has *already* been written for a given specification. This two-step approach has some drawbacks:

- it gives us no clue about how actually to *construct* programs;
- if the program in question has been developed in an unsystematic way, perhaps by someone else, it may be hard to detect the proper loop invariant(s).

In this section, we shall illustrate that it is often possible to write a program *together* with the proof of its correctness.

For that purpose, we look at the square root specification from Section 3, where we shall solemnly promise that the program to be constructed will not modify the value of x.

```
{$\mathtt{x} \geq 0$}
    $P$
{$\mathtt{y}^2 \leq \mathtt{x} \land (\mathtt{y}+1)^2 > \mathtt{x}$}
```

It seems reasonable to assume that P should be a loop, possibly with some preamble. With ϕ the (yet unknown) invariant of that loop, and with B the (yet unknown) test of the loop, we have the skeleton

```
{$\mathtt{x} \geq 0$}
    ???
{$\phi$}
    while $B$ do
        ???
{$\phi$}
    od
{$\mathtt{y}^2 \leq \mathtt{x} \land (\mathtt{y}+1)^2 > \mathtt{x}$}
```

We now face the main challenge: to come up with a suitable invariant ϕ, the form of which will direct the remaining construction process. In order to justify the postcondition, we must ensure that

$$\mathtt{y}^2 \leq \mathtt{x} \land (\mathtt{y}+1)^2 > \mathtt{x} \text{ is a logical consequence of } \phi \land \neg B. \tag{1}$$

There are at least two ways to achieve that, to be described in the next two subsections.

6.1 Deleting a Conjunct

A simple way to satisfy (1) is to define

$$\begin{aligned}\phi &= \mathtt{y}^2 \le \mathtt{x} \\ B &= (\mathtt{y}+1)^2 \le \mathtt{x}\end{aligned}$$

That is, we follow the following general recipe:

- let the loop test be the negation of one of the conjuncts of the postcondition;
- let the loop invariant be the remaining conjuncts of the postcondition.

Our prospective program now looks like

```
{$x \geq 0$}
    ???
{$y^2 \leq x$}
    while $(y+1)^2 \leq x$ do
        ???
{$y^2 \leq x$}
    od
{$y^2 \leq x \wedge (y+1)^2 > x$}
```

Thanks to the precondition $\mathtt{x} \geq 0$, initializing y to zero will establish the loop invariant. Thanks to the loop test $(\mathtt{y}+1)^2 \leq \mathtt{x}$, incrementing y by one will maintain the loop invariant. We end up with the program

```
{$x \geq 0$}
    $y := 0$
{$y^2 \leq x$}
    while $(y+1)^2 \leq x$ do
        $y := y+1$
{$y^2 \leq x$}
    od
{$y^2 \leq x \wedge (y+1)^2 > x$}
```

This program will clearly always terminate, but is rather inefficient. We shall now describe a method which in this case results in a more efficient program.

6.2 Replacing an Expression By an Identifier

Let us consider another way of satisfying (1). First observe that the postcondition involves the expression $\mathtt{y}$ as well as the expression $\mathtt{y} + 1$. It might be beneficial to loosen the connection between these two entities, by introducing a new identifier $\mathtt{w}$ which eventually should equal $\mathtt{y} + 1$ but in the meantime may roam more freely. Note that the postcondition is implied by the formula

$$\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x} \wedge \mathtt{w} = \mathtt{y} + 1$$

containing three conjuncts. It is thus tempting to apply the previous technique of "deleting a conjunct", resulting in

$$\begin{aligned} \phi &= \mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x} \\ B &= \mathtt{w} \neq \mathtt{y} + 1 \end{aligned}$$

Our prospective program now looks like

```
{$\mathtt{x} \geq 0$}
    ???
{$\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x}$}
    while $\mathtt{w} \neq \mathtt{y} + 1$ do
        ???
{$\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x}$}
    od
{$\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y} + 1)^2 > \mathtt{x}$}
```

To establish the loop invariant, we must not only initialize $\mathtt{y}$ to zero but also initialize $\mathtt{w}$ so that $\mathtt{w}^2 > \mathtt{x}$: clearly, $\mathtt{x} + 1$ will do the job.

For the loop body, it seems a sensible choice to modify *either* $\mathtt{y}$ *or* $\mathtt{w}$. This can be expressed as a conditional of the form

```
if $B'$ then
    $\mathtt{y} := E_1$
else
    $\mathtt{w} := E_2$
fi
```

We must check that each branch maintains the invariant, and therefore perform a case analysis:

- if B' is true, we must require that ${E_1}^2 \leq \mathtt{x}$;
- if B' is false, we must require that ${E_2}^2 > \mathtt{x}$.

Let E be an arbitrary expression; then these demands can be satisfied by stipulating

$$\begin{aligned} E_1 &= E \\ E_2 &= E \\ B' &= E^2 \leq \mathtt{x} \end{aligned}$$

We have thus constructed the program

```
{$\mathtt{x} \geq 0$}
    $\mathtt{y} := 0$;
    $\mathtt{w} := \mathtt{x} + 1$;
{$\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x}$}
    while $\mathtt{w} \neq \mathtt{y} + 1$ do
        if $\mathtt{E}^2 \leq \mathtt{x}$
        then
            $\mathtt{y} := E$
        else
            $\mathtt{w} := E$
        fi
{$\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x}$}
    od
{$\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 > \mathtt{x}$}
```

which is partially correct, no matter how E is chosen! But of course, we also want to ensure termination, and hopefully a quick such! For that purpose, we pick

$$E = (\mathtt{y} + \mathtt{w})\ \mathtt{div}\ 2$$

where a `div` b (for positive b) is the largest integer c such that $bc \leq a$. With that choice, it is not difficult to see that `y` and `w` will get closer to each other for each iteration, until eventually $\mathtt{w} = \mathtt{y} + 1$. This shows total correctness. Even more, the program runs much faster than our first attempt!

7 Well-Annotated Programs and Valid Assertions

We have argued that annotating a program with loop invariants is essential for the purpose of verification (and also to understand how the program works!) It is often beneficial to provide more fine-grained annotations.

Example 7.1. For the factorial program from Sect. 5.1, a fully annotated version looks like

```
{x ≥ 0}                          (A)
{1 = fac(0)}                     (B)
    y := 1;
{y = fac(0)}                     (C)
    z := 0;
{y = fac(z)}                     (D)
    while z ≠ x do
{y = fac(z) ∧ z ≠ x}             (E)
{y(z + 1) = fac(z + 1)}          (F)
        z := z + 1;
{yz = fac(z)}                    (G)
        y := y * z
{y = fac(z)}                     (H)
    od
{y = fac(z) ∧ z = x}             (I)
{y = fac(x)}                     (J)
```

We shall soon see that this program is in fact *well-annotated.* □

We first define what it means for an assertion to be *valid.* There are several cases:

Logical consequence. If the assertion $\{\psi\}$ immediately follows the assertion $\{\phi\}$, and ψ is a logical consequence of ϕ, then ψ is valid.

Trying to conform with the notation used in [1], we can write this rule as

$$\begin{array}{lll} & \{\phi\} & \\ \triangleright & \{\psi\} & \qquad\qquad \textbf{Implies} \text{ (if } \psi \text{ logical consequence of } \phi) \end{array}$$

saying that the marked assertion is valid.

Of course, in order to trust that ψ holds, we must at some point also establish that ϕ is valid!

Example 7.2. Referring back to Example 7.1, note that thanks to this rule

- assertion (B) is valid, since it is a mathematical fact and therefore surely a logical consequence of assertion (A);
- assertion (F) is valid, since if by assertion (E) we have $y = \text{fac}(z)$ then $y(z+1) = \text{fac}(z)(z+1) = \text{fac}(z+1)$;
- assertion (J) is valid, since it is a logical consequence of assertion (I). □

Rule for While loops. We have the rule

```
  {ψ}
      while B do
▷ {ψ ∧ B}                    WhileTrue
      ...
  {ψ}
      od
▷ {ψ ∧ ¬B}                   WhileFalse
```

saying that if ψ is a loop invariant then

- at the beginning of the loop body, the loop test has just evaluated to `true` and therefore $\psi \wedge B$ will hold;
- immediately after the loop, the loop test has just evaluated to `false` and therefore $\psi \wedge \neg B$ will hold.

Note that we are still left with the obligation to show that the two ψ assertions (one before the loop, the other at the end of the loop body) are valid.

Example 7.3. Referring back to Example 7.1, note that assertions (E) and (I) are valid, thanks to this rule. □

Rule for Conditionals. We have the rule

```
  {$\phi$}
      if $B$
      then
▷ {$\phi \wedge B$}                IfTrue
      ...
  {$\psi$}
      else
▷ {$\phi \wedge \neg B$}           IfFalse
      ...
  {$\psi$}
      fi
▷ {$\psi$}                         IfEnd
```

saying that if ϕ holds before a conditional command then

- at the beginning of the `then` branch, $\phi \wedge B$ will hold;
- at the beginning of the `else` branch, $\phi \wedge \neg B$ will hold;

and also saying that ψ holds after the conditional command if ψ holds at the end of both branches.

Again, we are left with the obligation to show that the initial ϕ assertion is valid, and that the ψ assertions concluding each branch are valid.

Observe that this rule is quite similar to the rule $\vee$ **Elim** from propositional logic!

Rule for Assignments We would surely expect that for instance it holds that

```
{$\mathtt{y} = 5$}
    $\mathtt{x} := \mathtt{y} + 2$
{$\mathtt{x} = 7 \wedge \mathtt{y} = 5$}
```

and it seems straightforward to go from precondition to postcondition. But now consider

$\{\mathtt{y} + 2\mathtt{z} \leq 3 \wedge \mathtt{z} \geq 1\}$
    $\mathtt{x} := \mathtt{y} + \mathtt{z}$
$\{???\}$

where it is by no means a simple mechanical procedure to fill in the question marks: what does the precondition imply concerning the value of $\mathtt{y} + \mathtt{z}$?

It turns out that we shall formulate the proper rule *backwards*: if we assign x the expression E, and we want $\psi(x)$ to hold *after* the assignment, we better demand that $\psi(E)$ holds *before* the assignment! This motivates the rule[10]

$\{\psi(E)\}$
    $\mathtt{x} := E$
$\triangleright$ $\{\psi(x)\}$ **Assignment**

Referring back to our first example, we have

$\{\mathtt{y} = 5\}$
$\{\mathtt{y} + 2 = 7 \wedge \mathtt{y} = 5\}$ **Implies**
    $\mathtt{x} := \mathtt{y} + 2$
$\{\mathtt{x} = 7 \wedge \mathtt{y} = 5\}$ **Assignment**

And referring back to our second example, we have

$\{\mathtt{y} + 2\mathtt{z} \leq 3 \wedge \mathtt{z} \geq 1\}$
$\{\mathtt{y} + \mathtt{z} \leq 2\}$ **Implies**
    $\mathtt{x} := \mathtt{y} + \mathtt{z}$
$\{\mathtt{x} \leq 2\}$ **Assignment**

since it is easy to check that if $y + 2z \leq 3$ and $z \geq 1$ then $y + z \leq 2$.

EXAMPLE 7.4. Referring back to Example 7.1, note that assertions (C), (D), (G), and (H) are valid, thanks to this rule. □

[10]We let $\psi(x)$ denote a formula where x is possibly free, and let $\psi(E)$ denote the result of substituting E for *all* free occurrences of x.

Well-annotation. We are now done with all the rules for validity. Note that there is no need for a rule for sequential composition $C_1; C_2$, since in

```
{$\phi$}
    $C_1$;
{$\phi_1$}
    $C_2$
{$\phi_2$}
```

the validity of each ϕ_i $(i = 1, 2)$ must be established using the form of C_i. But there is a rule for all other language constructs, and also a rule **Implies** that is not related to any specific language construct.

We are now ready to assemble the pieces:

Definition 7.5. We say that an annotated program

```
{$\phi$}
    ...
{$\psi$}
```

is *well-annotated* iff all assertions, *except* for the precondition ϕ, are valid. □

Theorem 7.6. *Assume that the annotated program*

```
{$\phi$}
    ...
{$\psi$}
```

is in fact well-annotated. Then the program is partially correct wrt. the specification (ϕ,ψ). □

From Examples 7.2, 7.3, and 7.4. we infer that the program in Example 7.1 is well-annotated. We can write

$\{\mathtt{x} \geq 0\}$	(A)	
$\{1 = \text{fac}(0)\}$	(B)	**Implies**
$\quad \mathtt{y} := 1;$		
$\{\mathtt{y} = \text{fac}(0)\}$	(C)	**Assignment**
$\quad \mathtt{z} := 0;$		
$\{\mathtt{y} = \text{fac}(\mathtt{z})\}$	(D)	**Assignment**
$\quad \mathtt{while}\ \mathtt{z} \neq \mathtt{x}\ \mathtt{do}$		
$\{\mathtt{y} = \text{fac}(\mathtt{z}) \land \mathtt{z} \neq \mathtt{x}\}$	(E)	**WhileTrue**
$\{\mathtt{y}(\mathtt{z}+1) = \text{fac}(\mathtt{z}+1)\}$	(F)	**Implies**
$\qquad \mathtt{z} := \mathtt{z} + 1;$		
$\{\mathtt{yz} = \text{fac}(\mathtt{z})\}$	(G)	**Assignment**
$\qquad \mathtt{y} := \mathtt{y} * \mathtt{z}$		
$\{\mathtt{y} = \text{fac}(\mathtt{z})\}$	(H)	**Assignment**
$\quad \mathtt{od}$		
$\{\mathtt{y} = \text{fac}(\mathtt{z}) \land \mathtt{z} = \mathtt{x}\}$	(I)	**WhileFalse**
$\{\mathtt{y} = \text{fac}(\mathtt{x})\}$	(J)	**Implies**

It is important to stress that once we have found a loop invariant (which is in general hard), computing the annotations is a quite mechanical process. In the above program, where the invariant appears as assertions (D) and (H), we

1. use **WhileTrue** and **WhileFalse** to compute (E) and (I);
2. repeatedly use **Assignment** "backwards" so as to compute first (C) and next (B), and so as to compute first (G) and then (F).

Now that all assertions are in place, what is left is to check that the parts "fit together", that is, verify that

1. (A) logically implies (B);
2. (E) logically implies (F);
3. (I) logically implies (J).

If either of these checks fails, the proposed invariant was not suitable. Notice the close relationship to the steps outlined in Section 5.2.

Example 7.7. The program developed in Section 6.1 can be well-annotated:

```
{$x \geq 0$}
{$0^2 \leq x$}                                  Implies
    $y := 0$
{$y^2 \leq x$}                                  Assignment
    while $(y+1)^2 \leq x$ do
{$y^2 \leq x \wedge (y+1)^2 \leq x$}            WhileTrue
{$(y+1)^2 \leq x$}                              Implies
        $y := y+1$
{$y^2 \leq x$}                                  Assignment
    od
{$y^2 \leq x \wedge (y+1)^2 > x$}               WhileFalse
```

□

EXAMPLE 7.8. The program developed in Section 6.2 can be well-annotated:

```
{$x \geq 0$}
{$0^2 \leq x \wedge (x+1)^2 > x$}                          Implies
    $y := 0$;
{$y^2 \leq x \wedge (x+1)^2 > x$}                          Assignment
    $w := x+1$;
{$y^2 \leq x \wedge w^2 > x$}                              Assignment
    while $w \neq y+1$ do
{$y^2 \leq x \wedge w^2 > x \wedge w \neq y+1$}            WhileTrue
        if $E^2 \leq x$
        then
{$y^2 \leq x \wedge w^2 > x \wedge w \neq y+1 \wedge E^2 \leq x$}   IfTrue
{$E^2 \leq x \wedge w^2 > x$}                              Implies
            $y := E$
{$y^2 \leq x \wedge w^2 > x$}                              Assignment
        else
{$y^2 \leq x \wedge w^2 > x \wedge w \neq y+1 \wedge E^2 > x$}      IfFalse
{$y^2 \leq x \wedge E^2 > x$}                              Implies
            $w := E$
{$y^2 \leq x \wedge w^2 > x$}                              Assignment
        fi
{$y^2 \leq x \wedge w^2 > x$}                              IfEnd
    od
{$y^2 \leq x \wedge w^2 > x \wedge w = y+1$}               WhileFalse
{$y^2 \leq x \wedge (y+1)^2 > x$}                          Implies
```

□

8 Arrays

Until now, we have only considered simple data structures like integers; in this section we shall consider *arrays*. An array can hold a sequence of values (just like a linked list can), where each element of that sequence can be accessed, and mutated, directly (unlike what is the case for a linked list, where one has to follow a chain of pointers).

Below is depicted an array a with 5 elements: 7,3,9,5,2.

0	1	2	3	4
7	3	9	5	2

We thus have $a[0] = 7$, $a[1] = 3$, etc.

Individual elements of arrays can be updated; after issuing the command `a`$[3] := 8$ the array a will now look like

0	1	2	3	4
7	3	9	8	2

In our examples, we shall assume a universe without negative numbers; this greatly improves readability, as otherwise assertions of the form $j \geq 0$ would have to be inserted numerous places.

We shall talk about two arrays being *permutations* of each other if they contain the same elements, though perhaps in different order. This is, e.g., the case for the two arrays given below:

0	1	2	3	4
8	3	9	8	2

0	1	2	3	4
3	9	2	8	8

We shall write $\text{perm}(a_1, a_2)$ if a_1 and a_2 are permutations of each other.

8.1 Verifying Programs Reading Arrays

Let us first consider programs which are *read-only* on arrays. For such programs, the verification principles from the previous sections carry through unchanged[11].

Example 8.1. Let us construct a program that decides whether the first `k` elements of the array `a` are sorted in increasing order, and stores the answer

[11]For programs manipulating arrays, loop invariants and other properties will almost certainly contain quantifiers, whereas for programs without arrays, invariants can often be expressed in propositional logic.

in the boolean identifier $\mathtt{b}$. We assume that $\mathtt{k} \geq 1$, and that $\mathtt{a}$ indeed has at least $\mathtt{k}$ elements. The desired postcondition can be expressed as

$$\mathtt{b} = \forall j (j < \mathtt{k} - 1 \rightarrow \mathtt{a}[j] \leq \mathtt{a}[j+1])$$

We shall need a loop, and it seems reasonable to guess that with $\mathtt{i}$ a counter, its invariant should be

$$\mathtt{i} \leq \mathtt{k} \ \wedge \ \mathtt{b} = \forall j (j < \mathtt{i} - 1 \rightarrow \mathtt{a}[j] \leq \mathtt{a}[j+1])$$

This invariant can be established by a preamble that assigns 1 to $\mathtt{i}$ and $\mathtt{true}$ to $\mathtt{b}$. To see this, note that then the first conjunct will hold due to the assumption $1 \leq \mathtt{k}$, and that in the second conjunct, the right hand side will be vacuously true (as $j < 0$ is impossible).

We might be tempted to let the loop test be $\mathtt{i} \neq \mathtt{k}$ since its negation, together with the invariant, trivially implies the postcondition. But observe that also $\neg\mathtt{b}$, together with the invariant, will imply the postcondition: for if $\forall j (j < \mathtt{i} - 1 \rightarrow \mathtt{a}[j] \leq \mathtt{a}[j+1])$ is false, and $\mathtt{i} \leq \mathtt{k}$ holds, then surely also $\forall j (j < \mathtt{k} - 1 \rightarrow \mathtt{a}[j] \leq \mathtt{a}[j+1])$ is false. We infer that a loop test whose negation is $\mathtt{i} = \mathtt{k} \vee \neg\mathtt{b}$ will be sufficient to establish (using a proof by cases) the postcondition from the invariant. By de Morgan, such a loop test is

$$\mathtt{i} \neq \mathtt{k} \wedge \mathtt{b}$$

Finally, we must write a loop body that maintains the invariant while making progress towards termination. For the latter purpose, we increment $\mathtt{i}$ by 1 at the end of the loop body. Using the “prime” notation, we thus have $\mathtt{i}' = \mathtt{i} + 1$. Note that whenever the loop body is executed, $\mathtt{b}$ is true (by the loop test) and thus $\mathtt{a}[j] \leq \mathtt{a}[j+1]$ for all $j < \mathtt{i} - 1$ (by the loop invariant). There are two cases:

- If $\mathtt{a}[\mathtt{i} - 1] \leq \mathtt{a}[\mathtt{i}]$, then $\mathtt{a}[j] \leq \mathtt{a}[j+1]$ for all $j \leq \mathtt{i} - 1$, that is for all $j < \mathtt{i}' - 1$. Thus, nothing else is needed to maintain the invariant.
- Otherwise, $\mathtt{a}[j] \leq \mathtt{a}[j+1]$ does *not* hold for all $j < \mathtt{i}' - 1$. To maintain the invariant, we need to assign $\mathtt{false}$ to $\mathtt{b}$.

By putting the pieces together, we see that we have constructed the program below. We simultaneously proved its correctness, following the steps mentioned in Section 5.2.

```
i := 1;
b := true;
while i $\neq$ k $\wedge$ b do
    if a[i $-$ 1] $\leq$ a[i]
    then
    else b := false
    fi;
    i := i $+$ 1
od
```

□

Example 8.2. Let us construct a program that stores in $\mathtt{m}$ the maximum of the first $\mathtt{k}$ elements of the array $\mathtt{a}$, that is the maximum of $\mathtt{a}[0], \ldots, \mathtt{a}[\mathtt{k}-1]$. We assume that $\mathtt{k} \geq 1$, and that $\mathtt{a}$ indeed has at least $\mathtt{k}$ elements.

The desired postcondition can be expressed as

$$\forall j (j < \mathtt{k} \rightarrow \mathtt{a}[j] \leq \mathtt{m}) \quad \wedge \quad \exists j (j < \mathtt{k} \wedge \mathtt{a}[j] = \mathtt{m})$$

We shall need a loop, and it seems reasonable to guess that its test should be $\mathtt{i} \neq \mathtt{k}$ and its invariant should be

$$\phi : \forall j (j < \mathtt{i} \rightarrow \mathtt{a}[j] \leq \mathtt{m}) \quad \wedge \quad \exists j (j < \mathtt{i} \wedge \mathtt{a}[j] = \mathtt{m})$$

since then the loop invariant, together with the negation of the loop test, will imply the postcondition. With the aim of establishing and maintaining the invariant ϕ, we construct the following program:

```
i := 1;
m := a[0];
while i $\neq$ k do
    if a[i] $>$ m
    then
        m := a[i];
        i := i $+$ 1
    else
        i := i $+$ 1
    fi
od
```

To prove the correctness of that program, we annotate it:

```
$\{\mathtt{k} \geq 1\}$
$\{\forall j(j < 1 \rightarrow \mathtt{a}[j] \leq \mathtt{a}[0]) \wedge$
 $\exists j(j < 1 \wedge \mathtt{a}[j] = \mathtt{a}[0])\}$                    Implies(A)
    $\mathtt{i} := 1;$
$\{\forall j(j < \mathtt{i} \rightarrow \mathtt{a}[j] \leq \mathtt{a}[0]) \wedge$
 $\exists j(j < \mathtt{i} \wedge \mathtt{a}[j] = \mathtt{a}[0])\}$                    Assignment
    $\mathtt{m} := \mathtt{a}[0];$
$\{\phi\}$                                                      Assignment
    while $\mathtt{i} \neq \mathtt{k}$ do
$\{\phi \wedge \mathtt{i} \neq \mathtt{k}\}$                                            WhileTrue
        if $\mathtt{a}[\mathtt{i}] > \mathtt{m}$
        then
$\{\phi \wedge \mathtt{i} \neq \mathtt{k} \wedge \mathtt{a}[\mathtt{i}] > \mathtt{m}\}$                              IfTrue
$\{\forall j(j < \mathtt{i}+1 \rightarrow \mathtt{a}[j] \leq \mathtt{a}[\mathtt{i}]) \wedge$
 $\exists j(j < \mathtt{i}+1 \wedge \mathtt{a}[j] = \mathtt{a}[\mathtt{i}])\}$                  Implies(B)
            $\mathtt{m} := \mathtt{a}[\mathtt{i}];$
$\{\forall j(j < \mathtt{i}+1 \rightarrow \mathtt{a}[j] \leq \mathtt{m}) \wedge$
 $\exists j(j < \mathtt{i}+1 \wedge \mathtt{a}[j] = \mathtt{m})\}$                      Assignment
            $\mathtt{i} := \mathtt{i}+1$
$\{\phi\}$                                                      Assignment
        else
$\{\phi \wedge \mathtt{i} \neq \mathtt{k} \wedge \mathtt{a}[\mathtt{i}] \leq \mathtt{m}\}$                              IfFalse
$\{\forall j(j < \mathtt{i}+1 \rightarrow \mathtt{a}[j] \leq \mathtt{m}) \wedge$
 $\exists j(j < \mathtt{i}+1 \wedge \mathtt{a}[j] = \mathtt{m})\}$                      Implies(C)
            $\mathtt{i} := \mathtt{i}+1$
$\{\phi\}$                                                      Assignment
        fi
$\{\phi\}$                                                      IfEnd
    od
$\{\phi \wedge \mathtt{i}=\mathtt{k}\}$                                            WhileFalse
$\{\forall j(j < \mathtt{k} \rightarrow \mathtt{a}[j] \leq \mathtt{m}) \wedge$
 $\exists j(j < \mathtt{k} \wedge \mathtt{a}[j] = \mathtt{m})\}$                        Implies
```

Below we shall show the validity of (A) and (B) and (C); it is then an easy exercise to check the validity of the rest of the assertions.

To see that (A) is valid, observe that 0 is the only j such that $j < 1$.

To see that (B) is valid, we must prove that

$$\forall j(j < \mathtt{i} \rightarrow \mathtt{a}[j] \leq \mathtt{m}) \text{ and} \tag{1}$$
$$\exists j(j < \mathtt{i} \wedge \mathtt{a}[j] = \mathtt{m}) \text{ and} \tag{2}$$
$$\mathtt{a}[\mathtt{i}] > \mathtt{m} \tag{3}$$

implies

$$\forall j(j < \mathtt{i} + 1 \rightarrow \mathtt{a}[j] \leq \mathtt{a}[\mathtt{i}]) \text{ and} \tag{4}$$
$$\exists j(j < \mathtt{i} + 1 \wedge \mathtt{a}[j] = \mathtt{a}[\mathtt{i}]). \tag{5}$$

To establish (4), let j be given such that $j < \mathtt{i} + 1$: if $j = \mathtt{i}$, the claim is trivial; otherwise, $j < \mathtt{i}$ and the claim follows from (1) and (3). For (5), we can use $j = \mathtt{i}$.

To see that (C) is valid, we must prove that

$$\forall j(j < \mathtt{i} \rightarrow \mathtt{a}[j] \leq \mathtt{m}) \text{ and} \tag{6}$$
$$\exists j(j < \mathtt{i} \wedge \mathtt{a}[j] = \mathtt{m}) \text{ and} \tag{7}$$
$$\mathtt{a}[\mathtt{i}] \leq \mathtt{m} \tag{8}$$

implies

$$\forall j(j < \mathtt{i} + 1 \rightarrow \mathtt{a}[j] \leq \mathtt{m}) \text{ and} \tag{9}$$
$$\exists j(j < \mathtt{i} + 1 \wedge \mathtt{a}[j] = \mathtt{m}). \tag{10}$$

To establish (9), let j be given such that $j < \mathtt{i} + 1$. If $j = \mathtt{i}$, the claim follows from (8). Otherwise, $j < \mathtt{i}$ and the claim follows from (6). Finally, (10) follows from (7). $\square$

8.2 Verifying Programs Updating Arrays

Next we consider programs which also *write* on arrays, that is, contain commands of the form $\mathtt{a}[\mathtt{i}] := E$. For such assignments, we want to apply the proof rule

```
  {$\psi(E)$}
      x := $E$
▷ {$\psi(x)$}                    **Assignment**
```

But if we apply that rule *naively* to the assignment $\mathtt{a}[2] := \mathtt{x}$ and the post-condition $\forall j(j < 10 \rightarrow \mathtt{a}[j] > 5)$, substituting the right hand side of the assignment for the left hand side, we would infer (since $\mathtt{a}[2]$ does not occur in the postcondition) that the following program is well-annotated:

$\{\forall j(j < 10 \rightarrow \mathtt{a}[j] > 5)\}$
    $\mathtt{a}[2] := \mathtt{x}$
$\{\forall j(j < 10 \rightarrow \mathtt{a}[j] > 5)\}$

This is clearly unsound, as can be seen by taking $\mathtt{x} = 3$.

Instead, the proper treatment is to interpret an assignment $\mathtt{a}[\mathtt{i}] := E$ as being really the assignment

$$\mathtt{a} := \mathtt{a}\{\mathtt{i} \mapsto E\}$$

That is, we assign to $\mathtt{a}$ an array that is like $\mathtt{a}$, except that in position $\mathtt{i}$ it behaves like E. More formally, we have

$$\begin{array}{rcll} \mathtt{a}\{\mathtt{i} \mapsto E\}[j] & = & E & \text{if } j = \mathtt{i} \\ \mathtt{a}\{\mathtt{i} \mapsto E\}[j] & = & \mathtt{a}[j] & \text{if } j \neq \mathtt{i} \end{array}$$

Then, in the above example, we get the well-annotated program

$\{\forall j(j < 10 \rightarrow \mathtt{a}\{2 \mapsto \mathtt{x}\}[j] > 5)\}$
    $\mathtt{a}[2] := \mathtt{x}$
$\{\forall j(j < 10 \rightarrow \mathtt{a}[j] > 5)\}$

where the precondition can be simplified to

$$\forall j((j < 10 \land j \neq 2) \rightarrow \mathtt{a}[j] > 5) \land \mathtt{x} > 5$$

which is as expected.

EXAMPLE 8.3. Consider the program below which swaps $\mathtt{a}[\mathtt{p}]$ and $\mathtt{a}[\mathtt{q}]$.

```
t := a[p];
a[p] := a[q];
a[q] := t
```

We would expect that with precondition $\mathtt{a}[\mathtt{p}] < \mathtt{a}[\mathtt{q}]$ we have postcondition $\mathtt{a}[\mathtt{q}] < \mathtt{a}[\mathtt{p}]$. This is indeed the case, as can be seen from the well-annotation below. Here a_2 denotes $\mathtt{a}\{\mathtt{q} \mapsto \mathtt{t}\}$, a_1 denotes $\mathtt{a}\{\mathtt{p} \mapsto \mathtt{a}[\mathtt{q}]\}\{\mathtt{q} \mapsto \mathtt{t}\}$, and a_0 denotes $\mathtt{a}\{\mathtt{p} \mapsto \mathtt{a}[\mathtt{q}]\}\{\mathtt{q} \mapsto \mathtt{a}[\mathtt{p}]\}$. The application of **Implies** is justified since $a_0[\mathtt{q}] = \mathtt{a}[\mathtt{p}]$ and also (even if $\mathtt{p} = \mathtt{q}$) $a_0[\mathtt{p}] = \mathtt{a}[\mathtt{q}]$.

$\{\mathtt{a}[\mathtt{p}] < \mathtt{a}[\mathtt{q}]\}$	
$\{a_0[\mathtt{q}] < a_0[\mathtt{p}]\}$	**Implies**
$\quad\mathtt{t} := \mathtt{a}[\mathtt{p}];$	
$\{a_1[\mathtt{q}] < a_1[\mathtt{p}]\}$	**Assignment**
$\quad\mathtt{a}[\mathtt{p}] := \mathtt{a}[\mathtt{q}];$	
$\{a_2[\mathtt{q}] < a_2[\mathtt{p}]\}$	**Assignment**
$\quad\mathtt{a}[\mathtt{q}] := \mathtt{t}$	
$\{\mathtt{a}[\mathtt{q}] < \mathtt{a}[\mathtt{p}]\}$	**Assignment**

$\square$

As a larger example, let us construct a program that rearranges the first $\mathtt{k}$ elements of an array $\mathtt{a}$ such that the highest element is placed in position number 0.

The desired postcondition can be expressed as follows:

$$\forall j(j < \mathtt{k} \to \mathtt{a}[j] \leq \mathtt{a}[0]) \ \wedge \ \mathrm{perm}(\mathtt{a}, a_0)$$

where the logical variable a_0 denotes the initial value of a; the latter condition $\mathrm{perm}(\mathtt{a}, a_0)$ is also part of the precondition. We shall need a loop, and it seems reasonable to guess that its test should be $\mathtt{i} \neq \mathtt{k}$ and its invariant should be

$$\psi : \forall j(j < \mathtt{i} \to \mathtt{a}[j] \leq \mathtt{a}[0]) \ \wedge \ \mathrm{perm}(\mathtt{a}, a_0)$$

since then the loop invariant, together with the negation of the loop test, will imply the postcondition. With the aim of establishing and maintaining the invariant ψ, we construct the following program:

```
i := 1;
while i ≠ k do
    if a[i] > a[0]
    then
        t := a[0];
        a[0] := a[i];
        a[i] := t;
        i := i + 1
    else
        i := i + 1
    fi
od
```

To prove the correctness of this program, we annotate it, as done in Fig. 1. Below we shall show the validity of (D); it is then an easy exercise to check the validity of the rest of the assertions.

Let $a' = \mathtt{a}\{0 \mapsto \mathtt{a}[\mathtt{i}]\}\{\mathtt{i} \mapsto \mathtt{a}[0]\}$; we must prove that

$$\forall j(j < \mathtt{i} \rightarrow \mathtt{a}[j] \leq \mathtt{a}[0]) \text{ and} \quad (11)$$
$$\text{perm}(\mathtt{a}, a_0) \text{ and} \quad (12)$$
$$\mathtt{a}[\mathtt{i}] > \mathtt{a}[0] \quad (13)$$

implies

$$\forall j(j < \mathtt{i} + 1 \rightarrow a'[j] \leq a'[0]) \text{ and} \quad (14)$$
$$\text{perm}(a', a_0) \quad (15)$$

Clearly a' is a permutation of $\mathtt{a}$, so (15) follows from (12). To show (14), let $j < \mathtt{i}+1$ be given; we must show that $a'[j] \leq a'[0]$ which is trivial if $j = 0$ so assume that $0 < j < \mathtt{i} + 1$. Since $a'[0] = \mathtt{a}[\mathtt{i}]$, our task can be accomplished by showing that

$$a'[j] \leq \mathtt{a}[\mathtt{i}].$$

We do so by a case analysis on the value of j. If $j = \mathtt{i}$, the claim follows from (13) since $a'[j] = \mathtt{a}[0]$. Otherwise, $0 < j < \mathtt{i}$ and therefore $a'[j] = \mathtt{a}[j]$; the claim thus boils down to showing $\mathtt{a}[j] \leq \mathtt{a}[\mathtt{i}]$ which follows from (11) and (13).

```
$\{\mathrm{perm}(\mathtt{a}, a_0)\}$
$\{\forall j (j < 1 \rightarrow \mathtt{a}[j] \leq \mathtt{a}[0]) \ \wedge\ \mathrm{perm}(\mathtt{a}, a_0)\}$   **Implies**
    $\mathtt{i} := 1;$
$\{\psi\}$                                                    **Assignment**
    while $\mathtt{i} \neq \mathtt{k}$ do
$\{\psi \wedge \mathtt{i} \neq \mathtt{k}\}$                                      **WhileTrue**
        if $\mathtt{a}[\mathtt{i}] > \mathtt{a}[0]$
        then
$\{\psi \wedge \mathtt{i} \neq \mathtt{k} \wedge \mathtt{a}[\mathtt{i}] > \mathtt{a}[0]\}$                       **IfTrue**
$\{\forall j (j < \mathtt{i} + 1 \rightarrow \mathtt{a}\{0 \mapsto \mathtt{a}[\mathtt{i}]\}\{\mathtt{i} \mapsto \mathtt{a}[0]\}[j] \leq \mathtt{a}\{0 \mapsto \mathtt{a}[\mathtt{i}]\}\{\mathtt{i} \mapsto \mathtt{a}[0]\}[0])$
 $\wedge\ \mathrm{perm}(\mathtt{a}\{0 \mapsto \mathtt{a}[\mathtt{i}]\}\{\mathtt{i} \mapsto \mathtt{a}[0]\}, a_0)\}$         **Implies**(D)
            $\mathtt{t} := \mathtt{a}[0];$
$\{\forall j (j < \mathtt{i} + 1 \rightarrow \mathtt{a}\{0 \mapsto \mathtt{a}[\mathtt{i}]\}\{\mathtt{i} \mapsto \mathtt{t}\}[j] \leq \mathtt{a}\{0 \mapsto \mathtt{a}[\mathtt{i}]\}\{\mathtt{i} \mapsto \mathtt{t}\}[0])$
 $\wedge\ \mathrm{perm}(\mathtt{a}\{0 \mapsto \mathtt{a}[\mathtt{i}]\}\{\mathtt{i} \mapsto \mathtt{t}\}, a_0)\}$            **Assignment**
            $\mathtt{a}[0] := \mathtt{a}[\mathtt{i}];$
$\{\forall j (j < \mathtt{i} + 1 \rightarrow \mathtt{a}\{\mathtt{i} \mapsto \mathtt{t}\}[j] \leq \mathtt{a}\{\mathtt{i} \mapsto \mathtt{t}\}[0])$
 $\wedge\ \mathrm{perm}(\mathtt{a}\{\mathtt{i} \mapsto \mathtt{t}\}, a_0)\}$                             **Assignment**
            $\mathtt{a}[\mathtt{i}] := \mathtt{t};$
$\{\forall j (j < \mathtt{i} + 1 \rightarrow \mathtt{a}[j] \leq \mathtt{a}[0])$
 $\wedge\ \mathrm{perm}(\mathtt{a}, a_0)\}$                                 **Assignment**
            $\mathtt{i} := \mathtt{i} + 1$
$\{\psi\}$                                                    **Assignment**
        else
$\{\psi \wedge \mathtt{i} \neq \mathtt{k} \wedge \mathtt{a}[\mathtt{i}] \leq \mathtt{a}[0]\}$                       **IfFalse**
$\{\forall j (j < \mathtt{i} + 1 \rightarrow \mathtt{a}[j] \leq \mathtt{a}[0])$
 $\wedge\ \mathrm{perm}(\mathtt{a}, a_0)\}$                                 **Implies**
            $\mathtt{i} := \mathtt{i} + 1$
$\{\psi\}$                                                    **Assignment**
        fi
$\{\psi\}$                                                    **IfEnd**
    od
$\{\psi \wedge \mathtt{i} = \mathtt{k}\}$                                      **WhileFalse**
$\{\forall j (j < \mathtt{k} \rightarrow \mathtt{a}[j] \leq \mathtt{a}[0]) \ \wedge\ \mathrm{perm}(\mathtt{a}, a_0)\}$   **Implies**
```

Figure 1: A well-annotated program for putting the highest array value first.

9 Procedures

A convenient feature, present in almost all programming languages, is the ability to define *procedures*; these are "named abstractions" of commonly used command sequences. In these notes, we shall consider procedure declarations of the form[12]

```
proc p (var x, y)
    local ...
    begin
        C
    end
```

where the procedure p has *body* C and *formal parameters* `x` and `y`; the body may refer to these parameters and possibly also to the *local identifiers* (declared after `local`) but *not* to any other ("global") identifiers.

A program P is now a sequence of procedure declarations, followed by a command (running the program amounts to executing that command). The syntax of commands was defined in Section 4 and is now extended to include *procedure calls*:

$$\begin{array}{rcl} C & ::= & \ldots \\ & | & \texttt{call}\ p(x_1, x_2) \end{array}$$

Here x_1 and x_2 are the *actual parameters*; note that they must be identifiers and we shall even require them to be distinct.

As an example, consider the procedure `swap` with declaration

```
proc swap (var x, y)
    local t
    begin
        t := x;
        x := y;
        y := t
    end
```

The following code segment contains a call of `swap`; after the call, we would expect that `z` $= 7$ and that `w` $= 3$.

[12]The generalization to an arbitrary number of formal parameters is immediate.

```
    z := 3;
    w := 7;
    call swap(z, w)
```

This example shows that our parameter-passing mechanism[13] is “call-by-reference” (as indicated by the keyword `var`): what is passed to the procedure is the “location” of the actual parameter, not just its value.

The body of a procedure may contain calls to other procedures. A procedure may even call itself (directly or indirectly), in which case we say that it is *recursive*. In Section 5.1 we implemented the factorial function using *iteration* (that is, `while` loops); below is an implementation which uses recursion and which thus more closely matches the recursive definition (given in Footnote 2) of the factorial function.

```
proc fact (var x, y)
    local t,r
    begin
        if x = 0
        then
            y := 1
        else
            t := x − 1;
            call fact(t, r);
            y := x ∗ r
        fi
    end
```

9.1 Contracts

As is the case for a program, also a procedure should come with a specification, which can be viewed as a “contract” for its use. For example, we might want a procedure `twice` with the contract[14]

[13]It is not difficult to extend our theory to other parameter passing mechanisms.

[14]Alternatively, one often uses the term “summary”.

```
proc twice (var x, y)
∀a, b
{x = a ∧ y = b}
    C
{x = 2a ∧ y = 2b}
```

This contract promises that for a call to `twice`, the following property holds for the identifiers provided as actual parameters: no matter what their values were *before* the call, their values *after* the call will be twice as big.

The natural way to implement `twice` is

```
proc twice (var x, y)
    begin
        x := 2 * x;
        y := 2 * y
    end
```

Note that this would *not* work if we had not required the actual parameters to be *distinct* identifiers, as the command `call twice(w, w)` would in effect multiply `w` by 4.

The contract for `swap` is as follows:

```
proc swap (var x, y)
∀a, b
{x = a ∧ y = b}
    C
{x = b ∧ y = a}
```

and we can easily verify that its implementation fulfills that contract: for arbitrary a and b, we have

```
{x = a ∧ y = b}
{y = b ∧ x = a}                    Implies
    t := x;
{y = b ∧ t = a}                    Assignment
    x := y;
{x = b ∧ t = a}                    Assignment
    y := t
{x = b ∧ y = a}                    Assignment
```

The contract for `fact` is as follows:

$\texttt{proc}\ \texttt{fact}\ (\texttt{var}\ \texttt{x}, \texttt{y})$
$\forall a$
$\{\texttt{x} = a \land \texttt{x} \geq 0\}$
$\quad C$
$\{\texttt{y} = \mathrm{fac}(a)\}$

To verify that the implementation of `fact` satisfies that specification, we must first address how to reason about procedure calls.

9.2 Rule for Procedure Calls

Given a procedure p with contract

$\texttt{proc}\ p\ (\texttt{var}\ \texttt{x}, \texttt{y})$
$\forall a_1, a_2$
$\{\phi_1(\texttt{x}, \texttt{y}, a_1, a_2)\}$
$\quad C$
$\{\phi_2(\texttt{x}, \texttt{y}, a_1, a_2)\}$

We might expect that for calls of p, we have the rule

$$\begin{array}{ll} & \{\phi_1(x_1, x_2, c_1, c_2)\} \\ & \quad \texttt{call}\ p(x_1, x_2) \\ \triangleright & \{\phi_2(x_1, x_2, c_1, c_2)\} \end{array}$$

While this rule is sound (since x_1 and x_2 denote distinct identifiers), it is not immediately useful, in that assertions unrelated to the procedure call are forgotten afterwards. To allow such an assertion ψ to be remembered, we propose the rule

$$\begin{array}{ll} & \{\phi_1(x_1, x_2, c_1, c_2) \land \psi\} \\ & \quad \texttt{call}\ p(x_1, x_2) \\ \triangleright & \{\phi_2(x_1, x_2, c_1, c_2) \land \psi\} \end{array}$$

We must require that ψ is indeed unrelated to the procedure call; due to our assumption that the body C manipulates no global identifiers, it is sufficient to demand that the identifiers denoted by x_1 and x_2 do not occur in ψ. To see the need for this restriction, consider the purported annotation below (where the role of ψ is played by the assertion $2\mathtt{w} = 14$):

$\{\mathtt{z} = 3 \wedge \mathtt{w} = 7 \wedge 2\mathtt{w} = 14\}$
    `call swap(z, w)`
$\{\mathtt{z} = 7 \wedge \mathtt{w} = 3 \wedge 2\mathtt{w} = 14\}$

This annotation is incorrect since after the call, $2\mathtt{w}$ equals 6 rather than 14.

As an extra twist, it is convenient (as we shall see in our examples) to allow c_1 and c_2 to be existentially quantified. We are now ready for

Definition 9.1. Assuming that x_1 and x_2 denote *distinct* identifiers which are *not* free in ψ, we have the following proof rule for procedure calls:

$\{\exists c_1 \exists c_2 (\phi_1(x_1, x_2, c_1, c_2) \wedge \psi)\}$
    `call` $p(x_1, x_2)$
$\triangleright$ $\{\exists c_1 \exists c_2 (\phi_2(x_1, x_2, c_1, c_2) \wedge \psi)\}$ **Call**

$\square$

EXAMPLE 9.2. Calling `twice` with arguments `z` and `w` satisfying $\mathtt{z} \leq 4$ and $\mathtt{w} \geq 7$, establishes $\mathtt{z} \leq 8$ and $\mathtt{w} \geq 14$. This is formally verified by the following well-annotation, where in the application of **Call**, the role of ψ is played by the assertion $c_1 \leq 4 \wedge c_2 \geq 7$.

$\{\mathtt{z} \leq 4 \wedge \mathtt{w} \geq 7\}$
$\{\exists c_1 \exists c_2 (\mathtt{z} = c_1 \wedge \mathtt{w} = c_2 \wedge c_1 \leq 4 \wedge c_2 \geq 7)\}$ **Implies**
    `call twice(z, w)`
$\{\exists c_1 \exists c_2 (\mathtt{z} = 2c_1 \wedge \mathtt{w} = 2c_2 \wedge c_1 \leq 4 \wedge c_2 \geq 7)\}$ **Call**
$\{\mathtt{z} \leq 8 \wedge \mathtt{w} \geq 14\}$ **Implies**

$\square$

EXAMPLE 9.3. Calling `swap` with arguments `z` and `w` such that $\mathtt{z} > \mathtt{w}$, establishes $\mathtt{w} > \mathtt{z}$. This is formally verified by the following well-annotation, where in applying **Call**, the role of ψ is played by the assertion $c_1 > c_2$.

$\{\mathtt{z} > \mathtt{w}\}$
$\{\exists c_1 \exists c_2 (\mathtt{z} = c_1 \wedge \mathtt{w} = c_2 \wedge c_1 > c_2)\}$ **Implies**
    `call swap(z, w)`
$\{\exists c_1 \exists c_2 (\mathtt{z} = c_2 \wedge \mathtt{w} = c_1 \wedge c_1 > c_2)\}$ **Call**
$\{\mathtt{w} > \mathtt{z}\}$ **Implies**

$\square$

We are now ready to prove that `fact` fulfills its contracts. That is, given a, we must prove

```
{$\mathtt{x} = a \wedge \mathtt{x} \geq 0$}
        if $\mathtt{x} = 0$
        then
            $\mathtt{y} := 1$
        else
            $\mathtt{t} := \mathtt{x} - 1$;
            call $\mathtt{fact}(\mathtt{t}, \mathtt{r})$;
            $\mathtt{y} := \mathtt{x} * \mathtt{r}$
        fi
{$\mathtt{y} = \mathrm{fac}(a)$}
```

But this follows from the following well-annotation:

```
{$\mathtt{x} = a \wedge \mathtt{x} \geq 0$}
        if $\mathtt{x} = 0$
        then
{$\mathtt{x} = a \wedge \mathtt{x} \geq 0 \wedge \mathtt{x} = 0$}                          IfTrue
{$1 = \mathrm{fac}(a)$}                                      Implies
            $\mathtt{y} := 1$
{$\mathtt{y} = \mathrm{fac}(a)$}                                      Assignment
        else
{$\mathtt{x} = a \wedge \mathtt{x} \geq 0 \wedge \mathtt{x} \neq 0$}                          IfFalse
{$\exists c(\mathtt{x} - 1 = c \wedge \mathtt{x} - 1 \geq 0 \wedge \mathtt{x} = a \wedge \mathtt{x} = c + 1)$} Implies
            $\mathtt{t} := \mathtt{x} - 1$;
{$\exists c(\mathtt{t} = c \wedge \mathtt{t} \geq 0 \wedge \mathtt{x} = a \wedge \mathtt{x} = c + 1)$}         Assignment
            call $\mathtt{fact}(\mathtt{t}, \mathtt{r})$;
{$\exists c(\mathtt{r} = \mathrm{fac}(c) \wedge \mathtt{x} = a \wedge \mathtt{x} = c + 1)$}           Call
{$\mathtt{xr} = \mathrm{fac}(a)$}                                     Implies
            $\mathtt{y} := \mathtt{x} * \mathtt{r}$
{$\mathtt{y} = \mathrm{fac}(a)$}                                      Assignment
        fi
{$\mathtt{y} = \mathrm{fac}(a)$}                                      IfEnd
```

10 Secure Information Flow

Assume we are dealing with two kinds of identifiers: those of high security (classified); and those of low security (non-classified). Our goal is that users with low clearance should not be able to gain information about the values of the classified identifiers. In the following, this notion will be made precise.

For the sake of simplicity, let us assume that there are only two identifiers in play: `l` (for low) and `h` (for high). We want to protect ourselves against an attacker (spy) who

- knows the initial value of `l`;
- knows the program that is running;
- can observe the final value of `l`;
- can *not* observe intermediate states of program execution.

A program is said to be *secure* if such an attacker cannot detect anything about the initial value of `h`.

10.1 Examples

The program below is *not* secure.

$$\texttt{l} := \texttt{h} + 7 \tag{2}$$

For by subtracting 7 from the final value of `l`, the attacker gets the initial value of `h`. On the other hand, the program below is clearly secure.

$$\texttt{l} := \texttt{l} + 47 \tag{3}$$

One rotten apple does not always spoil the whole barrel; having the insecure program in (2) as a preamble may still yield a secure program as in

$$\texttt{l} := \texttt{h} + 7;\ \texttt{l} := 27 \tag{4}$$

since we assumed that the attacker cannot observe intermediate values of `l`. Also the following program is secure:

$$\texttt{h} := \texttt{l} \tag{5}$$

For even though the attacker learns the *final* value of `h` (as it equals the initial value of `l` which is known), he is still clueless about the *initial* value of `h`.

The following program is just a fancy way of writing $\mathtt{l} := \mathtt{h} + 7$ (since we do not care about the final value of `h`)

$$\mathtt{l} := 7;\ \mathtt{while}\ \mathtt{h} > 0\ \mathtt{do}\ \mathtt{h} := \mathtt{h} - 1;\ \mathtt{l} := \mathtt{l} + 1\ \mathtt{od} \tag{6}$$

and is therefore insecure. Also, the following program is insecure

$$\mathtt{if}\ \mathtt{h} = 6789\ \mathtt{then}\ \mathtt{l} := 0\ \mathtt{else}\ \mathtt{l} := 1\ \mathtt{fi} \tag{7}$$

since if the final value of `l` is zero, we know that `h` was initially 6789.

10.2 Specification

By putting quantifiers in front of Hoare triples, we can express security formally:

Definition: The program P is secure iff

$$\begin{array}{l} \forall l_0\ \exists l_1\ \forall h_0 \\ \qquad \{\mathtt{l} = l_0 \wedge \mathtt{h} = h_0\} \\ \qquad\qquad P \\ \qquad \{\mathtt{l} = l_1\} \end{array}$$

To put it another way, the final value (l_1) of `l` must depend only on the initial value (l_0) of `l` and *not* on the initial value (h_0) of `h`.

To characterize *in*security, we negate the sentence above, and then repeatedly apply de Morgan's laws. This results in

$$\begin{array}{l} \exists l_0\ \forall l_1\ \exists h_0\ \neg \\ \qquad \{\mathtt{l} = l_0 \wedge \mathtt{h} = h_0\} \\ \qquad\qquad P \\ \qquad \{\mathtt{l} = l_1\} \end{array}$$

If P terminates, then $\neg\{\phi\}\ P\ \{\mathtt{l} = l_1\}$ is equivalent to $\{\phi\}\ P\ \{\mathtt{l} \neq l_1\}$. Therefore we arrive at:

Observation: A terminating program P is insecure iff

$$\exists l_0\ \forall l_1\ \exists h_0 \quad \{\mathtt{l} = l_0 \wedge \mathtt{h} = h_0\}\ P\ \{\mathtt{l} \neq l_1\}$$

To put it another way, a program is insecure if for all possible final values of `l`, there exists an initial value of `h` that produces a different final value for `l`.

10.3 Examples Revisited

We first address the programs that are secure, and show that they do indeed meet the requirement stated in our Definition. In each case, we are given some l_0 and must find l_1 such that

$$\forall h_0 \quad \{\mathtt{l} = l_0 \wedge \mathtt{h} = h_0\}\ P\ \{\mathtt{l} = l_1\}$$

For the program in (3), we choose l_1 as $l_0 + 47$; this does the job since

$$\forall h_0 \quad \{\mathtt{l} = l_0 \wedge \mathtt{h} = h_0\}\ \mathtt{l} := \mathtt{l} + 47\ \{\mathtt{l} = l_0 + 47\}$$

For the program in (4), we can choose l_1 as 27; for the program in (5), we simply choose l_1 as l_0.

We next address the programs that are *not* secure, and show (cf. our Observation) that no matter how l_1 has been chosen, we can find h_0 such that

$$\{\mathtt{l} = l_0 \wedge \mathtt{h} = h_0\}\ P\ \{\mathtt{l} \neq l_1\}$$

For the programs in (2) and (6), we can just pick an h_0 different from $l_1 - 7$, say $h_0 = l_1$. For clearly we have

$$
\begin{array}{l}
\{\mathtt{l} = l_0 \wedge \mathtt{h} = l_1\} \\
\qquad \mathtt{l} := \mathtt{h} + 7 \\
\{\mathtt{l} \neq l_1\}
\end{array}
$$

For the program in (7), we proceed by cases on l_1: if l_1 is zero, then we can choose (among many possibilities) h_0 to be 2345 since

$$
\begin{array}{l}
\{\mathtt{l} = l_0 \wedge \mathtt{h} = 2345\} \\
\qquad \mathtt{if}\ \mathtt{h} = 6789\ \mathtt{then}\ \mathtt{l} := 0\ \mathtt{else}\ \mathtt{l} := 1\ \mathtt{fi} \\
\{\mathtt{l} \neq 0\}
\end{array}
$$

Alternatively, if l_1 is one, then we choose h_0 to be 6789 since

$$
\begin{array}{l}
\{\mathtt{l} = l_0 \wedge \mathtt{h} = 6789\} \\
\qquad \mathtt{if}\ \mathtt{h} = 6789\ \mathtt{then}\ \mathtt{l} := 0\ \mathtt{else}\ \mathtt{l} := 1\ \mathtt{fi} \\
\{\mathtt{l} \neq 1\}
\end{array}
$$

(If l_1 is neither zero nor one, we can choose any value for h_0.)

10.4 Declassification

A severe limitation of our theory is exposed by the last example (7) which is considered insecure even though very little information may actually be leaked to the attacker. Think of `h` as denoting a PIN code, with the attacker testing whether it happens to be 6789; if the PIN codes were selected randomly, the chance of the test revealing the PIN code is very small (1 to 10,000). It is currently an important challenge for research in (language based) security to formalize these considerations!

10.5 Data Integrity

We might consider an alternative interpretation of the identifiers `l` and `h`: `l` denotes a licensed entity, whereas `h` denotes a hacked (untrustworthy) entity. The integrity requirement is now:

Licensed data should *not* depend on hacked data.

It is interesting to notice that the framework described on the preceding pages covers also that situation! In particular, a program satisfies the above integrity requirement if and only if it is considered secure (according to our Definition). For example, (4) is safe as the licensed identifier `l` will eventually contain 27 which does not depend on hacked data, whereas (7) is unsafe as the value of the hacked identifier `h` influences the value of the licensed identifier.

References

[1] Jon Barwise and John Etchemendy. *Language, Proof and Logic.* CSLI Publications, 1999.

[2] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jerome Feret, Laurent Mauborgne, Antoine Mine, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation (PLDI'03)*, pages 196–207, 2003.

[3] Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *2nd International Conference on Security in Pervasive Computing (SPC 2005)*, volume 3450 of *LNCS*, pages 193–209. Springer-Verlag, 2005.

[4] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[5] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge University Press, 2nd edition, 2004.