Introduction to Programming Logic

I.S.W.B. Prasetya
Dept. of Information and Computing Sciences
Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, the Netherlands

e-mail: wishnu@cs.uu.nl
URL: www.cs.uu.nl\staff\wishnu.html
## Contents

1 Introduction 7

2 Formulas 11

3 Logic and Proofs 17
   3.1 Style .................................................. 18
      3.1.1 The Implicit G Abbreviation ...................... 20
   3.2 Inference Rule ........................................... 20
      3.2.1 Substitution ........................................ 21
   3.3 Definition ............................................... 22
   3.4 Rewriting ................................................ 22
   3.5 Subproof ................................................ 24
   3.6 Equational Proof ........................................ 24
   3.7 Handling $\forall$ and $\exists$ .......................... 26
      3.7.1 Inferring a $\forall$-Formula ....................... 26
      3.7.2 Eliminating $\exists$ ................................ 27
   3.8 Some Useful Proof Techniques ......................... 28
   3.9 Exercise ................................................ 32
   3.10 Solution ............................................... 34

4 Lists 37
   4.1 Basic Notation and Properties ....................... 38
   4.2 Proofs ................................................ 41
   4.3 Exercises ............................................... 43
   4.4 Solution ............................................... 44

5 Program and Specification 47
   5.1 Program ................................................ 47
   5.2 Specification .......................................... 50
   5.3 Exercises ............................................... 55
   5.4 Solution ............................................... 56
## 6 The Basic Proof System

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Abstract Model</td>
<td>59</td>
</tr>
<tr>
<td>6.1.1</td>
<td>State</td>
<td>60</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Modelling Program as Relation</td>
<td>60</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Predicate as a Set of States</td>
<td>61</td>
</tr>
<tr>
<td>6.1.4</td>
<td>The Formal Meaning of Hoare Triples</td>
<td>62</td>
</tr>
<tr>
<td>6.1.5</td>
<td>Expressing Non-termination</td>
<td>63</td>
</tr>
<tr>
<td>6.2</td>
<td>Weakest Pre-condition</td>
<td>63</td>
</tr>
<tr>
<td>6.3</td>
<td>Loop Reduction Strategy</td>
<td>70</td>
</tr>
<tr>
<td>6.4</td>
<td>Organizing Your Proof</td>
<td>73</td>
</tr>
<tr>
<td>6.5</td>
<td>Example: Fibonacci Problem</td>
<td>76</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Step 1: Proof Plan</td>
<td>78</td>
</tr>
<tr>
<td>6.5.2</td>
<td>Step 3: Cross Checking with Init</td>
<td>79</td>
</tr>
<tr>
<td>6.5.3</td>
<td>Step 4: Verifying Invariance</td>
<td>80</td>
</tr>
<tr>
<td>6.5.4</td>
<td>Step 5: Loop Termination</td>
<td>83</td>
</tr>
<tr>
<td>6.5.5</td>
<td>Step 6: Initialization</td>
<td>84</td>
</tr>
<tr>
<td>6.5.6</td>
<td>Summarizing</td>
<td>84</td>
</tr>
<tr>
<td>6.6</td>
<td>On Iteratively Constructing I</td>
<td>85</td>
</tr>
<tr>
<td>6.7</td>
<td>Linear Scanning</td>
<td>86</td>
</tr>
<tr>
<td>6.7.1</td>
<td>Constructing Invariant</td>
<td>86</td>
</tr>
<tr>
<td>6.7.2</td>
<td>Loop Termination</td>
<td>87</td>
</tr>
<tr>
<td>6.7.3</td>
<td>Proving Invariance</td>
<td>89</td>
</tr>
<tr>
<td>6.7.4</td>
<td>Initialization</td>
<td>93</td>
</tr>
<tr>
<td>6.8</td>
<td>Invariants</td>
<td>94</td>
</tr>
<tr>
<td>6.8.1</td>
<td>Head and Tail Invariant</td>
<td>94</td>
</tr>
<tr>
<td>6.8.2</td>
<td>Loop for Searching a Solution</td>
<td>96</td>
</tr>
<tr>
<td>6.8.3</td>
<td>Iterative Implementation of Tail Recursion</td>
<td>96</td>
</tr>
<tr>
<td>6.9</td>
<td>Proof of the Distributivity of wp</td>
<td>98</td>
</tr>
<tr>
<td>6.10</td>
<td>Simultant Assignment</td>
<td>99</td>
</tr>
<tr>
<td>6.11</td>
<td>Reducing Program’s Specification</td>
<td>100</td>
</tr>
<tr>
<td>6.12</td>
<td>Updating Compound Structures</td>
<td>101</td>
</tr>
<tr>
<td>6.12.1</td>
<td>Updating Record</td>
<td>103</td>
</tr>
<tr>
<td>6.13</td>
<td>Advanced Features</td>
<td>104</td>
</tr>
<tr>
<td>6.13.1</td>
<td>assert and assume</td>
<td>104</td>
</tr>
<tr>
<td>6.13.2</td>
<td>Non-deterministic Assignment</td>
<td>105</td>
</tr>
<tr>
<td>6.13.3</td>
<td>Non-deterministic Choice</td>
<td>105</td>
</tr>
<tr>
<td>6.13.4</td>
<td>goto</td>
<td>105</td>
</tr>
<tr>
<td>6.13.5</td>
<td>Inner block</td>
<td>106</td>
</tr>
<tr>
<td>6.13.6</td>
<td>Expression with side effect</td>
<td>106</td>
</tr>
<tr>
<td>6.13.7</td>
<td>Exception</td>
<td>106</td>
</tr>
<tr>
<td>6.13.8</td>
<td>Alias</td>
<td>108</td>
</tr>
<tr>
<td>6.14</td>
<td>Exercise</td>
<td>110</td>
</tr>
<tr>
<td>6.15</td>
<td>Solution</td>
<td>115</td>
</tr>
</tbody>
</table>

## 7 Program Call

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Restrictions</td>
<td>134</td>
</tr>
<tr>
<td>7.2</td>
<td>Formal Treatment of Program Call</td>
<td>135</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Transforming Program Call</td>
<td>136</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Black Box Reduction</td>
<td>137</td>
</tr>
<tr>
<td>7.3</td>
<td>Functional Black Boxes</td>
<td>140</td>
</tr>
<tr>
<td>7.4</td>
<td>Recursion</td>
<td>142</td>
</tr>
<tr>
<td>7.5</td>
<td>Exercise</td>
<td>145</td>
</tr>
<tr>
<td>7.6</td>
<td>Solution</td>
<td>146</td>
</tr>
</tbody>
</table>
Introduction

Computers are probably the most revolutionary and successful invention in our history. In the last few decades, they have played a dominant role in creating the modern world that we know now. A world that in terms of productivity and comfort far exceeds the imagination of our ancestors. Increasingly, computers are becoming an integral part of our modern culture. They are everywhere, aiding us in every activity. What most people do not realize, is that dependency always has its other side. More and more aspects of our life are now controlled by computers. Some mistakes or failures made by the computers may just be annoying, but some may have a disastrous consequence for someone’s life. The danger comes from the way people produce software. The behavior of even a simple program is often very complex. The depth of its complexity far exceeds that of more traditional engineering products, like radios, cars, or bridges. On the other hand, for more than a century we have known a golden age of our industry. People have been successful in manufacturing and building almost anything, from paper clips to skyscrapers. This develops a certain style of engineering, and not surprisingly the same style is used in developing software.

Unfortunately, the traditional engineering approach is not suitable for dealing with the magnitude of complexity that we have to deal with in the software world. This often results in unreliable software. It is true that there are many applications where occasional failures are acceptable, but there are also applications where this is highly unacceptable. Think of the control systems of traffic lights, aeroplanes, or nuclear reactors. Think also of the protocols that allow internet access to bank accounts. These are applications whose failure may cause huge material or financial damage or even loss of human life. Such an application or system is often called critical system. It is obvious that such a system has to be developed with a lot of care.

To safe guard the quality of a program, what people do is ‘testing’ it. That is, we run the program on a number of real inputs and observe whether its behavior conforms to its specification. The approach is fast and cheap. You can find a lot of errors with testing, but unfortunately you cannot find them all. This is simply because in general you cannot test all possible inputs. E.g. the set of all possible inputs of a small program that takes a single integer as a parameter is theoretically infinitinely large. Even if we use the fact that integers are in the computer represented by a string
CHAPTER 1. INTRODUCTION

of e.g. 32 bits, there are still $10^{32}$ possible integers! It is not feasible to test all these possibilities. The fact is, testing is very incomplete. For critical systems this may not be a satisfactory solution.

This book introduces you to an alternative technique, called program verification. It uses mathematical reasoning to verify the correctness of a program. Its use requires a lot more effort than testing, but on the other hand the coverage is complete, which means that using this technique guarantees that a program to behave as specified with respect to all inputs.

This is not to say that program verification is the ultimate solution for writing correct software. Real applications are huge. Writing formal specifications for them and subsequently verifying them will be very expensive, even with the help of computers. However the technique discussed here can still be effectively applied to slices or models of real programs. Because, within its range, verification is always thorough, it often finds bugs that ordinary testing misses.

An example

To give you a glimpse to some of the issues we are going to discuss, let us consider the following small program to calculate the greatest common divisor of two positive integers.

An integer $d$ is said to be a divisor of another integer $x$ if it can divide $x$ (so, 3 is a divisor of 6, but not of 7). This $d$ is a common divisor of $x$ and $y$ if it is a divisor of $x$ and of $y$. Let us denote the greatest common divisor of two integers $x$ and $y$ by $\text{gcd} \, x \, y$. For example, $\text{gcd} \, 2 \, 6 = 2$ and $\text{gcd} \, 6 \, 9 = 3$.

The straightforward way to calculate $\text{gcd} \, x \, y$ is to try out all possible integers $d$ between 2 and $x/2$ (assuming $x < y$) to find out which ones are common divisors, and then compute the maximum. However, there is a faster way to do it. This is shown in Figure 1.1—this solution has been around for more than 2000 years. It is invented by an ancient Greek mathematician, Euclides in ± 320 BC. You may have seen this program before, but let us pretend that this is the first time we see it and we are given the task to check if it is correct.

```plaintext
Euclid(x,y : int) : int {
   while x<>y do if x>y then x:=x-y else y:=y-x ;
   return x
}
```

Figure 1.1: A program to compute the greatest common divisor.

Specification

Well, what we can do is to test the program. But even to test we first need a specification, which is a description of what the program is expected to do. When we test a program, we test if it meets its specification.

Ideally, a specification should not be ambiguous, or else testers and other programmers may misunderstand how to use your program and the consequence may be undesirable. Human languages (like English or Japanese) are unfortunately often ambiguous. So, they are actually not suitable for writing specifications. Instead, you should use a so-called specification language. Like programming languages, such a language have a strict syntax. This helps to avoid most ambiguity. Some even have formal descriptions of their semantics which exclude ambiguity. Here we will use a very simple specification language. Its original style is invented by C.A.R. Hoare back in 1969 [13]. In the Hoare style, we write a specification like this:
It says that if the value of \(x\) and \(y\) are positive, then the call \(\text{Euclid}(x,y)\) will return a value which is equal to \(\text{gcd} \ x \ y\).

The expressions between \{* \} are called assertions. The one preceding the program is called the pre-condition of the program, and the one after the program is called the post-condition. The triple of pre-condition, program, and post-condition is often called Hoare triple. This style of specification is used in many verification approaches.

**Proof**

In program verification, rather than testing we try to prove that the \text{Euclid} is correct. So, how can we do that?

We proceed in steps. First, rather than proving that \(\text{return} = \text{gcd} \ x \ y\) we can equivalently try to prove that the loop in \text{Euclid} terminates in this condition:

\[
x = \text{gcd} \ X \ Y
\]  \hspace{1cm} (1.1)

where \(X\) and \(Y\) represent the initial values of \(x\) and \(y\). The strategy now is to come up with some general property about \(x\) and \(y\) which will be maintained at the end of every iteration; e.g. this property:

\[
x > 0 \ \land \ y > 0
\]  \hspace{1cm} (1.2)

Such a property is also called loop invariant or simply invariant. Because it holds at the end of every iteration, it also holds when the loop terminates. So we may be able to use it to prove what we want, namely (1.1). We need to find a good invariant though. Unfortunately (1.2) does not contain enough information to prove (1.1). In general, finding the right invariant is not trivial.

We are not going to prove it here, but you can actually show that every iteration also maintains this:

\[
\text{gcd} \ x \ y = \text{gcd} \ X \ Y
\]  \hspace{1cm} (1.3)

Note that when the loop terminates \(x=y\) holds. Now, what would be the \text{gcd} of \(x\) and \(y\) if both are positive and equal? Well, \(x\) itself of course! (e.g. \(\text{gcd} \ 6 \ 6 = 6\)) So, after the last iteration we have:

\[
\text{gcd} \ x \ y = x
\]  \hspace{1cm} (1.4)

Together with (1.3) this now implies that \(x = \text{gcd} \ X \ Y\) holds when the loop ends. So, we are done.

We did skip few things in the proof above, such as the proof of (1.3) and the loop will actually terminate. We will show you the complete proof of \text{Euclid} later. The important thing to note here is that a proof such as the one above made no particular assumption about the initial values of \(x\) and \(y\), other than, in this case, that they have to be positive. Consequently, the proven correctness is guaranteed with respect to all possible (positive) inputs of the program!
Proof: Informal or Formal?

The proof in the previous section is an informal proof, because it is mostly written in plain English. Such a proof is often ambiguous and incomplete. Because we can just express anything in plain English, an informal proof is also error prone. Nevertheless, it allows us to present a proof abstractly, thus making it easy to understand. Proofs of rules and theorems in this book are presented in a mix of formal and informal parts. The main purpose of those proofs is to improve your understanding.

When verifying a real program we would have a very different purpose. The main purpose of verification is to check the program’s correctness, and not so much to try to understand why the program is correct. Of course if it does help your understanding then it is also good. Anyway, because a proof itself is also prone to errors, for verification we need a to construct a formal proof.

Such a proof must follow a certain set of rules that define precisely which proof steps you are allowed to use. This set of rules is called logic. It is usually also straightforward to write a program to check if a formal proof follows its logic; thus, it contains no error. Of course having those rules do mean that we are no longer free to write a proof in any way that is convenient to us; so, formal proofs do tend to be longer, more verbose, and are more difficult to write.

Can we also program a computer to construct (formal) proofs for us? Unfortunately, it has been shown that in general this is impossible. So, our human insight is still needed. In various limited cases, automatic proof construction is possible. Proof automation is also a very active research field. The number of tools to do it keeps growing and they are also getting smarter. The use of proof tools is however outside the scope of this book. We will focus more on learning the basic of programming logics. For this purpose the use of tools is not really necessary.

To use a logic effectively it is not enough to know its rules: you also have to understand their principles. If your attempt to prove the correctness of a program fails, it can be because the program is incorrect, or because your proof strategy is not good. In either case, you will have to analyze the proof to discover where it starts to go wrong, why it goes wrong, and what options you have to fix it. You cannot do this without a good understanding of how the logic works.

A skill that you also need to develop here is the skill to write a proof formally, because this is the kind of proofs that you have to deal with in an actual setup. Training it requires some discipline. However, as a programmer this should not be new to you: writing Java programs also requires discipline. You cannot deliver a program with a bad syntax. The compiler won’t accept it, so it won’t run. The same kind of discipline applies to formal proofs. If you write a formula in the wrong way, or use the logic in a wrong way, your proof is flawed, which is just as useless as a program with a bad syntax.

Despite all the above emphasis on formal proofs it is a mistake to dismiss the value of informal proofs. Despite the ambiguity and lack of precision, an informal proof is a good tool to abstractly document our insight and high level proof strategies. For verification there is no way around: you need a formal proof.

New Vocabulary

ambiguous/unambiguous; assertion; formal/informal proof; Hoare triple; invariant; post-condition; pre-condition; specification; specification language; testing; verification
Formulas

Expressions are texts like $x=0$ and $x + 1 > 0$. In programming expressions are used in assignments and to guard loops and conditional statements. In logics they are also called formulas. We have seen in Chapter 1 that formulas are used to specify the allowed initial and final states of a program. Simple formulas like above may be sufficient to specify simple programs. A more complicated program, such as a program to check if an array contains a zero, may however need more sophisticated formulas to specify. This chapter will introduce you to the language we will use to write our formulas. We will simply call it Form. Form generally follows the style common in the presentation of predicate logic e.g. [7, 11, 10, 6, 16]. It is a small language, but is sufficient for our purpose. For industrial use people use larger specification languages such as Z [8] or JML [18].

Operators

In the simplest form, a formula\(^1\) is just a constant like 1, 2, 3, or a variable like $x$ and $y$. Boolean constants are denoted by true and false. A formula can also be constructed with operators, like in $x + 1$. Allowed binary operators are shown in Table 2.1. They are listed in the decreasing order of their priority. For example, $\land$ has a higher priority (binds stronger) than $\lor$. So, $p \land q \lor r$ is read as $(p \land q) \lor r$. Operators listed in the same row has the same priority. We also have the unary operator $\neg$ (negation); it has a higher priority than any binary operator.

Note that the symbol = denotes the operator to test equality and that $=$ has the lowest priority\(^2\). We also allow some list operators: :, ++, and $\in$. They will be explained in Chapter 4.

The L and R flags in Table 2.1 indicate the ’associativity’ of an operator. L means that the operator is left associative, and R means that it is right associative. For example, $\ast$ is left

\(^1\)In literature e.g. [15] people distinguish, for a technical reason, between Boolean and non-Boolean formulas. The latter are called terms. We will ignore this distinction as it is not necessary for our discussions.

\(^2\)In particular, this is unlike in some programming languages, e.g. Java, where $=$ denotes assignment and where the operator used to test equality (usually $==$) has a higher priority.
Table 2.1: Binary operators in Form and their associativity. The operators are listed in decreasing priority.

associative, meaning that \( x - y - z \) is read as \((x - y) - z\). On the other hand, \( \Rightarrow \) is right associative, meaning that \( p \Rightarrow q \Rightarrow r \) is read as \( p \Rightarrow (q \Rightarrow r) \). Operators which are marked with LR, like \(+\), are both left and right associative. It means that \( x + y + z \) can be read as either \((x + y) + z\) or \(x + (y + z)\). The operators with no associativity flag are not associative. For example, \( =\) is not associative in Form, which means that you cannot write \( x = y = z \), because this is not meaningful in Form.

Note that in \( \Rightarrow \) is right associative and its priority is less than \( \land \) and \( \lor \). So, for example:

1. \( p \Rightarrow q \Rightarrow r \) means \( p \Rightarrow (q \Rightarrow r) \).
2. \( p \land q \Rightarrow r \) means \( (p \land q) \Rightarrow r \).
3. \( p \lor q \Rightarrow r \) means \( (p \lor q) \Rightarrow r \).

The mod and div Operators

For integers \( x \) and \( y \), the expression \( x \mod y \) returns the integer remainder of dividing \( x \) by \( y \); whereas \( x \div y \) returns the quotient. If \( x \) and \( y \) are both positives the meaning is quite clear. When they are negatives different languages may define the results differently. Here, our interpretation is as follows.

If \( y \neq 0 \), then \( x \mod y \) and \( x \div y \) are defined to be integers \( r \) respectively \( q \), such that:

\[
0 \leq r < |y| \land x = qy + r \tag{2.1}
\]

So, \(-5 \mod 3 = 1\), and \(-5 \mod -3 \) is also 1. However, \( 5 \mod -3 = 2 \).

Functions

You can define your own functions in Form, like:

\[
xor x y = (x \lor y) \land (x \neq y)
\]

However functions in Form are pure mathematical functions; they are not regarded as programs. They are introduced purely for the purpose of specifying programs, for example to express that a program implements a certain mathematical function. Functions application is written like \( f x y \) instead of the usual \( f(x, y) \), and has the highest priority. For example, \( f x y + 1 \) is read as \( (f x y) + 1 \).
Array

In imperative programming we often deal with arrays so we also want to be able to specify properties about them. We will use the usual array notation: if \( a \) is an array \( a[i] \) denotes the \( i \)-th element of \( a \). Just to simplify reasoning we will use idealized arrays. They are of infinite size; the indices run from \(-\infty \) to \(+\infty \). So, you do not have to worry about the meaning of \( a[i] \) if \( i \) turns out to be beyond \( a \)'s domain of indices. When dealing with real programs we should of course naively ignore this aspect.

Integer domain expressions

An integer domain expression, or simply a domain expression, is the term we will use for an expression like \( a \leq i < b \) and all its variations with different numeric relations; \( a \), \( i \), and \( b \) are of the integer type. They are abbreviations; for example:

1. \( a \leq b < c \) abbreviates \( a \leq b \land b < c \).
2. \( a \leq i, j < c \) abbreviates \( a \leq i < c \land a \leq j < c \).

The same abbreviation scheme is used for other combinations of \( <, \geq, \leq, \) and \( > \).

Conditional

A conditional formula is a formula of the form \( g \rightarrow e_1 \mid e_2 \). The \( g \) has to of type Boolean. If \( g \) is true then the meaning of this formula is equal to \( e_1 \) and else equal to \( e_2 \).

You may also write a conditional like this: \( \text{if } g \text{ then } e_1 \text{ else } e_2 \). Some people prefer this notation because of its familiar construct. If you use it, don’t confuse it with the if-then-else used in imperative programming languages. In the latter if-then-else is an action that can change the state of a program. In Form if-then-else is a formula; it specifies a value and does not change anything.

Quantified formula

A quantified formula is a formula like \( (\forall k : 0 \leq k < N : a[k] > 0) \). It means that for all \( k \) such that \( 0 \leq k < N \) the formula \( a[k] > 0 \) holds. A formula of this form is also called a universally quantified expression.

Another example of a quantified expression is \( (\exists k : 0 \leq k < N : a[k] > 0) \), which means that there exists a \( k \) in the domain \( 0 \leq k < N \) such that \( a[k] > 0 \) holds. An expression of this form is also called an existentially quantified formula.

The symbols \( \forall \) and \( \exists \) are called quantifiers (also called quantors or binders). The formula between the two colons in a quantified formula is called the domain of the formula. For example the domain of \( (\forall k : 0 \leq k < N : a[k] > 0) \) is \( 0 \leq k < N \). A domain is said to be full when it is equivalent to \( \text{true} \). It is said to be empty when it is equivalent to \( \text{false} \). For example, \( (\forall k : 0 \leq k < 0 : a[k] > 0) \) has an empty domain. When its domain is empty a quantified expression has a ‘special’ meaning:

\[ (\forall i : \text{false} : P \ i) = \text{true} \]
\[ (\exists i : \text{false} : P \ i) = \text{false} \]

It actually makes sense. For example in the case of \( \exists \): there is of course no \( i \) that can satisfy \( \text{false} \). In other words, the domain allows no \( i \), let alone an \( i \) satisfying \( P \ i \).

We also write \( (\forall i :: P \ i) \) (notice that the domain formula is omitted) as a shorthand for \( (\forall i : \text{true} : P \ i) \). Similarly, \( (\exists i :: P \ i) \) means \( (\exists i : \text{true} : P \ i) \). Do keep in mind that omitting the domain specification in a quantified formula is not the same as quantifying over an empty domain.
You can also quantify over multiple 'bound variables', e.g. \((\forall i, j : 0 \leq i, j < n : a[i] = a[j])\), which says that for all \(i\) and \(j\) in the specified domain the formula \(a[i] = a[j]\) holds. You can also have nested quantifications, e.g:

\[(\forall i : 0 \leq i < n : (\forall j : i \leq j < n : a[i] \leq a[j]))\]

(so what does the formula above say in plain English?)

The scope of quantification

In a quantified formula like \((\forall k : 0 \leq k < N : a[k] > x)\), the 'variable' \(k\) is called a \textit{bound variable}. Its scope is limited within the quantified formula. So, if we have an expression like:

\[(\forall k : 0 \leq k < N : a[k] > x) \land k\]

the \(k\) outside the quantification refers to something different than the \(k\) inside the quantification. Variables which are not bound are called \textit{free}. For example, \(x\) and \(N\) in the above formula are free.

The term 'variable' is indeed a bit confusing. We actually have two concepts of 'variable' here: 'variable' in mathematics and 'variable' in programming. In a mathematical formula such as \(x > 0 \Rightarrow x \geq 0\), the \(x\) is commonly called a variable. It is just a word mathematicians use for a place holder representing an arbitrary value. In most mathematical books, such a variable represents some abstract concept that has nothing to do with programs.

In formal specifications, we use mathematical formulas to specify what a program is supposed to do. So, you can expect that some variables in a specification represent actual variables of a program. For example \(x\) and \(y\) in the specification of the program Euclid from the previous chapter refer to Euclid's variables of the same names. So, they are program variables. From the mathematical point of view, a program variable occurring in a specification is just a special kind of a mathematical variable that also represents a physical place in the computer's memory where one can store an arbitrary value. When writing a specification we are thus not limited to just using program variables. E.g. the variables \(X\) and \(Y\) we introduced in the informal proof of Euclid are not program variables. They are variables that refer to arbitrary integers. A bounded variable is also an example of a variable which is not a program variable. We are allowed to use these kinds of variables so that we can conveniently express our intent in a specification.

\textbf{Shifted domain}

When you have a universally or existentially quantified expression you can shift some part of its domain like this:

1. \((\forall i : P i \land Q i : R i)\) is equivalent to \((\forall i : P i : Q i \Rightarrow R i)\)
2. \((\exists i : P i \land Q i : R i)\) is equivalent to \((\exists i : P i : Q i \land R i)\)

You can also shift the entire domain like this:

1. \((\forall i : P i : Q i)\) is equivalent to \((\forall i : P i : Q i)\)
2. \((\exists i : P i : Q i)\) is equivalent to \((\exists i : P i \land Q i)\)

\textbf{Example}

As an example, below we show a program. Formulas are used to specify the program's pre- and post-conditions. The specification says that when the the program \texttt{ELEM} is called in a state satisfying \(n \geq 0\), it will terminate in a state where:

\[
\text{return } = (\exists k : 0 \leq k < n : a[k] = x)
\]

In other words, \texttt{ELEM} tests whether \(x\) is an element of the array \(a\) in the domain \(0 \leq k < n\).
\{ \ast n \geq 0 \ \ast \} \\

ELEM(x,n:int, a:int[]) : bool \\
{ \var i : int ; \\
var found : bool ; \\
i:=0 \\
found:=false ; \\
while i<n \land \neg found do { found:=(a[i]=x) ; i:=i+1 } ; \\
return found \\
} \\
\{ \ast \text{return} = (\exists k : 0 \leq k < n \ : a[k] = x) \ \ast \} \\

\hline

New Vocabulary

associativity: left,right,non-; conditional; domain expression; quantified formula: universal, existential, domain, empty domain, full domain, without domain; quantifier; shifting domain; variable: bound, free
A logic is essentially just a collection of rules for inferring a formula from a set of other formulas. If a formula \( q \) can be inferred from another formula \( p \) by using a one rule or a composition of rules then we say that \( q \) follows from \( p \). The sequence of the rules used to do so is our proof. A logic is also a formal system: it describes its derivation rules precisely, so that there is no ambiguity as to when a rule can be applied, and what the results are. Proposition and predicate logics are examples of logics. There are many more logics —some are very general, others are specialized to specific purposes, e.g. to abstractly reason about security protocols. Even proposition and predicate logics have many variants. Later we will also use a logic to prove the correctness of a program. Such a logic is called programming logic.

There are also many programming logics, but often they are built on top of a predicate logic. The pure programming logic part is used to prove properties about executions; the underlying predicate logic is used to prove properties about data. An example of a data-level property is this:

*The first element of ascendingly sorted array, is also its minimum.*

This property holds for any array, and has nothing to do with program execution. The technique that we will use (Chapter 6) for proving program correctness works in fact by reducing a pair of program and specification to a set of data-level formulas. After the reduction we can fall back to the predicate logic to finish the proof.

This chapter will show you the style used in this book to prove predicate logic formulas. When you look in the literature, don’t be surprised to find many different proof styles. Which one to use is often a matter of taste; however, some styles are less suitable for introductory training (e.g. too low level, or use difficult notations). We will use a variation of the flag-pole style used in [21]. We will explain some of the basic proof rules. However, this book is not intended to be a complete introduction to the predicate logic —we assume you have been sufficiently exposed to it. For your convenience, the Appendix lists commonly used inference rules and equalities in predicate logic.
**Basic Terminology**

In mathematics a function that returns a Boolean value is called *predicate*. Here the notion corresponds to Form-formula of type Boolean, since such a formula can be seen as defining a function over its free variables as parameters.

A predicate \( p \) is *universally true*, also called *valid*, when it is true for all possible values (of the correct types) of its *free* variables. For example, this predicate:

\[
x > 0 \Rightarrow x \geq 0
\]  

(3.1)

is true for all possible values of \( x \), and therefore it is valid. On the other hand, \( x > 0 \) is not a valid predicate, because it is not true when \( x = 0 \).

A valid predicate usually captures a useful fact. To explicitly indicate that an expression \( p \) is valid, we write it like this: \( \vdash p \). When we loosely say "proving a formula \( p \)" we actually mean proving that \( p \) is valid. So, it as if \( p \) has an implicit outmost-level \( \forall \)-quantifier over its free variables. We also say "proving \( q \) follows from \( p \)" to mean proving \( \vdash p \Rightarrow q \).

Other frequently used notions are the notions of stronger and weaker. If \( \vdash p \Rightarrow q \) holds, then we say that \( p \) is *stronger* than \( q \). Conversely, \( q \) is *weaker* than \( p \). Notice that when \( p \) is valid, so is any \( q \) which is weaker than \( p \). Therefore, if we know that if \( p \) is stronger than \( q \), then proving \( \vdash p \) is sufficient to prove \( \vdash q \).

**3.1 Style**

The proof style that we are going to use is most suitable for proving expressions of this form:

\[
(p_1 \land \ldots \land p_n) \Rightarrow q
\]  

(3.2)

or this form:

\[
(\forall v_i, \ldots, v_k : p_1 \land \ldots \land p_n : q)
\]  

(3.3)

Notice that (3.3) can also be written as \( (\forall v_i, \ldots, v_k : p_1 \land \ldots \land p_n \Rightarrow q) \). So it is just a quantified form of (3.2). Any predicate can be converted to one of those forms.

As an example, let us prove that this formula is valid:

\[
(\forall i : 0 \leq i < n : a[i] > 0) \\
\land n > 0 \\
\land x > 0 \\
\Rightarrow \\
x \ast a[0] > 0
\]  

(3.4)

Intuitively, this is quite easy:

The universal quantification at the left hand side of the \( \Rightarrow \) says that \( a[i] > 0 \), for all \( i \) in the specified range. Since \( n > 0 \), this range is not empty. In particular, this implies that \( a[0] > 0 \). Since \( x > 0 \), then it follows that \( x \ast a[0] > 0 \).

This is still an informal proof. Figure 3.1 shows how we write this more formally. As you can see, a proof consists of three parts: its name, declaration, and body. The proof in Figure 3.1 is named \texttt{top}. Name is optional. The part between the name and the keyword \texttt{BEGIN} is the declaration part, and the part between \texttt{BEGIN} and \texttt{END} is the body.

In the declaration part you list what your assumptions are, and the formulas that you want to prove. The latter are called *goals*. Entries in the declaration can be labelled (e.g. \texttt{A1}, \texttt{A2}, \texttt{G} and so on) so that we can refer to them. An entry whose label begins with an \( A \) is an assumption. In the proof above you see three assumptions: \texttt{A1}, \texttt{A2}, and \texttt{G}. An entry whose label begins with a \( G \) is a goal.
3.1. STYLE

PROOF top

[A1:] \((\forall i:0 \leq i < n: a[i] > 0)\)

[A2:] \(n > 0\)

[A3:] \(x > 0\)

[G :] \(x \ast a[0] > 0\)

BEGIN

1 { trivially follows from A2 }

\(0 \leq 0 < n\)

2 { \(\forall\)-elimination on A1 using 1, taking \(i = 0\) }

\(a[0] > 0\)

3 { trivially follows from A3 and 2 }

\(x \ast a[0] > 0\)

END

Figure 3.1: This is how we write a formal proof in this book.

Consider a proof with \(A_1 \ldots A_n\) specified as the assumptions and \(g\) as a goal; the total formula you try to prove is:

\(A_1 \land \ldots \land A_n \Rightarrow g\)

If the proof is successful, that is what it proves\(^1\). You can also specify multiple goals. A proof is then successful if all of its goals are proven.

Recall that a logic is a system to infer new formulas from other formulas. This is what you do in a proof’s body. So, in the body you can write down new ‘facts’ (that is, formulas) which are inferred from the assumptions, or from the formulas you derived previously. In this way, all formulas you write down in a proof are logical consequences of the assumptions. A proof is therefore successful if you manage to derive all its goals somewhere in the proof body. Later, you may also open a subproof within a proof; things become a bit more complicated then, but we will return to this later.

We will call the set of assumptions plus the facts derived so far within a proof the proof-context or simply context of the proof. This context grows as you add more facts into a proof. What we said in the previous paragraph can also be rephrased in terms of proof-context: we can only extend a proof with a new formula if it can be inferred from the current proof-context. Note that if you have subproofs, the context does not include the formulas derived in subproofs. It does include formulas derived previously in the ancestor proofs.

It is convenient to number the inferred formulas in a proof body, so that we can refer to them by their numbers. The text between \{ and \} (printed blue, if you have a color version of this book) is a hint of how the formula below it is inferred. For example, in the proof in Figure 3.1 the first expression derived is \(0 \leq 0 < n\). The hint above it says that this follows trivially from A2.

The next question is, how do we infer a new fact? Strictly speaking we have to use a rule from whatever logic we use. Here it would be the predicate logic. But since we also use integers, floats, strings, and lists, we additionally also need logics about each of those data types. Since you would be familiar with simple reasoning about e.g. integers and simple boolean operators, we will allow some degree of informality. E.g. it is acceptable to consider the fact that \(n = 0\) implies \(n > 0\) to be trivial and use this to justify a proof step as we do in step 1 and 3 in Figure 3.1. Afterall, proving these kind of little facts are not what we mainly want to learn here. In the real setup you of

\(^1\)Be careful: if you have assumptions, a successful proof does not imply the validity of the goal. What you prove is that the assumptions imply the goal.
course cannot say this to a proof tool. If it wants a proof, then you have to deliver it. There are however many tools that can help you to automate the proving of simple facts.

### 3.1.1 The Implicit G Abbreviation

The declaration of a goal, e.g. as in Figure 3.1:

\[
[G:] \ x \cdot a[0] > 0
\]

also implicitly introduces the variable G as an abbreviation for the specified goal. If you have a large goal, this is convenient to avoid having to copy the entire formula in your proof steps. E.g. the proof in Figure 3.1 could also be written:

\begin{verbatim}
PROOF top
...
[G:] x * a[0] > 0
BEGIN
  1 ...
  2 ...
  3 { trivially follows from A3 and 2 } G
END
\end{verbatim}

### 3.2 Inference Rule

Another way to infer a formula, is to use an inference rule. Predicate logic has a number of rules. Here is one called the **Conjunction Rule** (also called \(\land\) Introduction Rule):

\[
\begin{array}{c}
P \\
Q \\
\hline
P \land Q
\end{array}
\] (3.5)

It is read like this: if \(P\) and \(Q\) are in the current proof-context, then you can infer \(P \land Q\) (and add it in your proof). The formulas above the line are also called the **premises** of the rule. The formula under the line is called the **conclusion**; it tells you which formula can be derived from the premises. You can also view an inference rule as a rule to reduce the conclusion to the premises: in order to prove the conclusion, it is sufficient to prove the premises.

Here is another example:

\[
\begin{array}{c}
P (\forall i : P i : Q i) \\
\hline
Q e
\end{array}
\] (3.6)

This rule can be explained more easily with an example. Suppose we already have \((\forall i : 0 \leq i < n : a[i] > 0)\) in the current proof-context. So, if we can infer that \(e\) it is inside the \(\forall\)-quantification’s domain, that is: \(0 \leq e < n\), then it follows that \(a[e] > 0\) holds.

The rule is used in step 2 in the proof in Figure 3.1: in step 1 we have concluded that 0 is inside the quantification’s range; hence by the rule above we can infer \(a[0] > 0\).

Since using the rule above we can infer a new formula as if by eliminating a \(\forall\) symbol, the rule is called \(\forall\)-Elimination. It is also called \(\forall\)-Specialization. The act of eliminating \(\forall\) in this way is called **specializing** or **instantiating** a \(\forall\)-formula. Be careful: there are some conditions in invoking this rule. They will be explained in Subsection 3.2.1.
Sometimes, we name the premises, just to make it convenient to refer to them, like in:

\[
\begin{align*}
C_1 & : P e \\
C_2 & : (\forall i : P i : Q i) \\
\hline
Q e
\end{align*}
\]

The names are printed in bold.

You can find a list of predicate logic’s inference rules in the Appendix.

### 3.2.1 Substitution

Recall the ∀-Elimination rule in (3.6). We show it again below, next to an alternative formulation of the rule:

\[
\begin{array}{c}
P e \\
(\forall i : P i : Q i) \\
\hline
Q e
\end{array} \quad \begin{array}{c}
P[e/i] \\
(\forall i : P i : Q i) \\
\hline
Q[e/i]
\end{array}
\] (3.7)

You see a new notation here, namely \(P[e/i]\). It means the formula that you obtain by replacing all free occurrences of \(i\) in \(P\) by \(e\). This operation is called substitution. Be careful not to confuse it with the array or list notations, which also use square brackets. Also, do not confuse the use of ‘/’ with the division operator, which is often denoted with the same symbol.

The rule to the left implicitly assumes that \(i\) is free in the formulas \(P\) and \(Q\). We implicitly treat \(P\) and \(Q\) as functions parameterized with \(i\). So, \(P e\) is the predicate we obtain by replacing \(i\) in \(P\) with \(e\). In other words, it is the same as \(P[e/i]\) in the second rule. The second rule on the right is more precise in the sense that it does not make the implicit assumption about treating formulas as functions. While more explicit, the second rule is also more cluttered. So, in this book we will stick to the style used in the first rule.

We will also need substitution to handle assignments in imperative programs — we will discuss this in Chapter 6.

Note that in a substitution \(P[e/i]\) only free occurrences of \(i\) in \(P\) should be replaced! So, for example:

\[((\exists i :: a[i]>x) \land i \geq 0) \ [0/i]\]

results in \((\exists i :: b[i]) \land a[i]>0\), and not \((\exists i :: b[0]) \land 0 \geq 0\).

A substitution \(P[e/i]\) is also not allowed if after the substitution there is a free variable in \(e\) that becomes bound, because the resulting formula would have a different meaning — one which is not your intention. So, for example, this substitution is not allowed:

\((\exists i :: a[i]>x) \ [i+1/x]\)

because it would result in \((\exists i :: a[i]+1>i+1)\), where \(i\), which occurs free in the substituting expression \(i+1\), becomes bound in the result.

You can however first rename the bound variable \(i\) in \((\exists i :: a[i]>x)\), for example to \(i'\). So, we obtain: \((\exists i' :: a[i']]>x)\). Renaming bound variables does not change the meaning of a quantified expression. After the renaming it is now safe to apply the substitution \([i+1/x]\), producing:

\((\exists i' :: a[i']>i+1)\).

Because of the above restriction, note that when using an inference rule like those in (3.7), no free variables in the substituting expression, i.e. the \(e\) in (3.7), should become bound in the inferred formula.
Proof:

\[ D1: \forall i : i < n : b[i] \]
\[ A : Q n \land b[n] \]
\[ G : Q(n+1) \]

\begin{enumerate}
\item \{ rewrite the 2nd conjuct of A with Quantification over Singleton (T A.4.10) \}
\[ Q n \land (\forall i : i = n : b[i]) \]
\item \{ rewrite 1 with D1 \}
\[ (\forall i : i < n : b[i]) \land (\forall i : i = n : b[i]) \]
\item \{ rewrite 2 with Domain Split (T 3.8.1) \}
\[ (\forall i : i < n \lor (i = n) : b[i]) \]
\item \{ rewrite 2 with i < n \lor (i = n) = i < n+1 \}
\[ (\forall i : i < n+1 : b[i]) \]
\item \{ rewrite 4 with D1 \}
\[ Q(n+1) \]
\end{enumerate}

Figure 3.2: Well chosen definitions can improve the readability of your proof.

### 3.3 Definition

You can also introduce definitions in the declaration part of a proof. They are convenient for abbreviating long expressions that we expect to occur several times in the proof. Well chosen definitions can make a complicated proof much easier to read. As an example, consider the formula below:

\[(\forall i : i < n : b[i]) \land b[n] \Rightarrow (\forall i : i < n+1 : b[i])\]  (3.8)

Intuitively, the formula is clearly valid. Figure 3.2 shows the formal proof. Entries in the declaration whose labels begin with a D are definitions. In Figure 3.2 D1 introduces the notation Q n to abbreviate (\(\forall i : i < n : b[i]\)).

The equality that specifies a definition can be thought to be implicitly universally quantified over all parameters of the defined name. So, in the definition of Q in Figure 3.2, the equation does not only specify the meaning of Q n for a single n, but also for all instances of n. So, it should be read:

\[(\forall n :: Q n = (\forall i : i < n : b[i]))\]

### 3.4 Rewriting

Rewriting is a frequently used operation in proofs. If you know that \(e_1 = e_2\), then you can replace some or all occurrences of \(e_1\) in E with \(e_2\). We use \(E[e_2/e_1]\) to denote the resulting formula. Within the current proof, the result of a rewrite is equivalent with the original formula. Rewriting is captured by the following rule:

\[
\begin{align*}
P & \\
\frac{e_1 = e_2}{P[e_2/e_1]} & (3.9)
\end{align*}
\]
Rewriting is a generalization of substitution (Section 3.2.1). A substitution replaces occurrences of a single variable. In rewriting you can replace arbitrary expressions. Like in substitution, there are a number of things that you should keep in mind:

1. As in substitution, when performing a rewrite $E[e_2/e_1]$, no free variables in $e_2$ should become bound after the rewrite.

2. When performing a rewrite $E[e_2/e_1]$, you can only replace an occurrence of $e_1$ in $E$ if in this occurrence no free variable of $e_1$ is bound. For example, supposed we have derived this expression:

$$(\forall i : 0 \leq i : b[i]) \land b[i]$$

If we also know that $b[i] = x$, we can replace $b[i]$ with $x$. So, we infer this:

$$( (\forall i : 0 \leq i : b[i]) \land b[i] ) [x/b[i]]$$

which gives:

$$(\forall i : 0 \leq i : b[i]) \land x$$

Notice that only the outer $b[i]$ is replaced. The $b[i]$ inside the quantification has its $i$ bound: the scope of this $i$ is limited to within the quantified expression. So, it does not mean the same thing as the $i$ in the derived fact $b[i] = x$. Therefore, using this equality to rewrite the inner $b[i]$ would be incorrect. So, this is not a correct result of the above rewrite:

$$(\forall i : 0 \leq i : x) \land x$$

If you have derived $P$, you can also replace its occurrences in $Q$ with true. Similarly, if you have derived $\neg P$, you can replace the occurrences of $P$ in $Q$ with false. These are captured by the following rules:

$$
\begin{array}{c|c}
P & \neg P \\
\hline
Q & Q[true/P] & Q[false/P] \\
\end{array}
$$

(3.10)

In the Appendix you can find a list of standard equalities in predicate logic. You can use them to rewrite the formulas in your proof. Using them needs a little explanation. Here is an example of the listed equalities:

$$\vdash \neg(\forall i : P i : Q i) = (\exists i : P i : \neg(Q i))$$

(3.11)

Implicitly, we mean that the equality holds for all instances of $P$ and $Q$. We could formulate it more precisely:

$$\vdash (\forall P, Q :: \neg(\forall i : P i : Q i) = (\exists i : P i : \neg(Q i)))$$

but the outermost $\forall$ clutters the formula. So, we will stick with the first formulation, and use as a convention that the theorems in this book are implicitly universally quantified over their free variables (a quite common convention in mathematics).

In effect, we can use (3.11) not only to rewrite an expression that is exactly equal to either sides of (3.11), but also to rewrite any expression whose pattern matches either sides of (3.11). For example, using (3.11) we can rewrite:

$$\neg(\forall i : i \geq 0 : i > 0) \to (\exists i : i \geq 0 : \neg(i > 0))$$
and vice versa. Sometimes, an equality only holds under certain circumstances. For example, a theorem called Domain Merging (Theorem 3.8.3—see Section 30) states the following equality:

\[ \vdash a \leq b \Rightarrow (a \leq i < b + 1 = a \leq i < b \lor (i = b)) \]  

(3.12)

Note that implicitly, as we have agreed above, the formula holds for all \( a, b, \) and \( i \). So, it allows you to rewrite e.g.:

\[ 0 \leq i < k + 1 \text{ to } 0 \leq i < k \lor (i = k) \]

but only if you know that \( 0 \leq k \). This kind of rewriting is called conditional rewriting, because you can only do it under certain circumstances.

### 3.5 Subproof

A proof may also contain subproofs. As an example, let us prove the following:

\[ \neg b \land (x^2 > 0 \Rightarrow y < 0) \land (y \leq 0 \Rightarrow b) \Rightarrow (x = 0) \]  

(3.13)

Informally this can be argued by contradiction as follows. Assume \( x \neq 0 \). It follows that \( x^2 > 0 \). By the assumption \( x^2 > 0 \Rightarrow y < 0 \) it follows that \( y < 0 \). Next, by the assumption \( y \leq 0 \Rightarrow b \) it follows that \( b \) holds. But this is in contradiction with the assumption \( \neg b \) ! So, the original assumption \( x \neq 0 \) must be wrong, and therefore \( x = 0 \).

The formal proof is shown in Figure 3.3. Notice that the first expression derived in top is justified by the subproof that follows below it.

The nesting in a proof can be several levels. For example, a proof \( P \) may contain a subproof \( Q \), which in turn contains a subproof \( R \).

Recall that a new formula can only be added to a proof if it is a logical consequence of the current proof-context. When we start a subproof \( Q \), its starting context consists of \( Q \)'s own assumptions plus the context of \( Q \)'s parent proof at the moment \( Q \) is opened.

Anything you derive inside \( Q \) potentially relies on \( Q \)'s assumptions. Keep in mind that the scope of these assumptions is limited within \( Q \). Therefore information derived inside a subproof cannot be used in the parent proof. For example, in Figure 3.3 within top we are not allowed to refer to any formula derived inside pot. However, we can (of course) in top refer to the conclusion of pot, which is formula 1 in top.

For labelling the proof elements (e.g. assumptions and derived formulas) in \( Q \) you can reuse labels already used in the parent proofs. We will use qualified notation when it is necessary to distinguish them. For example, in the subproof pot (Figure 3.3), \( A1 \) refers to pot's own assumption \( A1 \), whereas top.A1 refers to the \( A1 \) of top. Similarly, top.1 refers to the formula 1 in top.

### 3.6 Equational Proof

The flag pole proof style we use so far works fine for proving a formula of the form \( p \Rightarrow q \), but is less convenient for proving formulas of the form \( e_1 = e_2 \). A more convenient way to do it is by using an equational proof. The idea is that we rewrite \( e_1 \) to an equivalent formula and repeat this until we obtain \( e_2 \). As an example, consider this:

\[ \neg(\forall i : 0 \leq i : b[i] \Rightarrow c[i]) = (\exists i : 0 \leq i : b[i] \land \neg c[i]) \]  

(3.14)

One way to prove this in a pure flag pole style is by proving the double implications equivalent of the above equality. It can be shorter and nicer as shown in the following equational style:
3.6. EQUATIONAL PROOF

PROOF top
[A1:] \neg b
[A2:] x^2 > 0 \Rightarrow y < 0
[A3:] y \leq 0 \Rightarrow b
[G :] x = 0
BEGIN

1 \{ see proof pot \} \quad x \neq 0 \Rightarrow false

PROOF pot
[A1:] x \neq 0
[G :] false
BEGIN

1 \{ follows from A1 \} \quad x^2 > 0
2 \{ Modus Ponens (R A.1.2) on 1 and top.A2 \} \quad y < 0
3 \{ follows from 2 \} \quad y \leq 0
4 \{ Modus Ponens (R A.1.2) on 3 and top.A3 \} \quad b
5 \{ conjunction of 4 and top.A1 \} \quad false

END

2 \{ Contradiction Rule (R A.1.3) on 1 \} \quad x = 0

END

Figure 3.3: An example showing the use of a subproof.

EQUATIONAL PROOF top

\neg (\forall i : 0 \leq i : b[i] \Rightarrow c[i])

= \{ Negate \forall (T A.4.7) \}

(\exists i : 0 \leq i : \neg(b[i] \Rightarrow c[i]))

= \{ basic equalities of boolean connectors (T A.4.1) \}

(\exists i : 0 \leq i : b[i] \land \neg c[i])

END

In an equational proof the only thing you can do is to rewrite the current formula in the proof to another equivalent formula. So, every formula in the sequence above is equivalent to each other\footnote{Notice that if you are in the body of an EQUATIONAL PROOF, the formula in k+1-th step is equivalent with the previous one. In contrast, in the body of a flag pole PROOF each formula you infer is, by agreement, only a consequence of the current proof-context. For example if you infer e_1 in k-th step (in a flag pole proof) and e_2 in the next step, you cannot then claim that e_1 = e_2. If this is what you want, you have to explicitly try to infer (and write down) e_1 = e_2 in your k+1-th step.}.

An equational proof may have a declaration part containing assumptions. You don’t have to specify the goal. If \( f_1 \) and \( f_n \) are the first and the last formulas in an equational proof, the proof justifies the validity of \( f_1 = f_n \). However, if the proof has declared assumptions \( A_1, \ldots, A_k \), then it proves this instead:

\[ A_1 \land \ldots \land A_k \Rightarrow (f_1 = f_n) \]
3.7 Handling ∀ and ∃

The quantors ∀ and ∃ are used quite often. Section 3.2 already explains the rule (3.6) used to instantiate (eliminate) a ∀-quantification. The following rule can be used to infer a ∃-quantification. It is called ∃-Introduction Rule: in a sense it is the dual of the ∀-Elimination Rule (3.6). Informally, suppose we have already inferred a[i]=0. So, there is (obviously) a k such that a[k]=0. In other words, we have (∃x : a[x] = 0). Formally, in two variants:

\[
\frac{Q}{\exists i : Q[i]}
\]

\[
\frac{P e}{Q e}
\]

\[
\frac{\exists i : P i : Q i}{Q e}
\]

Inferring a ∀-quantification and eliminating ∃ are a bit more complicated.

3.7.1 Inferring a ∀-Formula

Suppose we have a top level proof with A as an assumption and g as a goal. If successful, it proves ⊨ A ⇒ g. That is, that A ⇒ g is valid. But we can infer more. Suppose i occurs free in A ⇒ g, then by the meaning of 'validity' the same proof also implies ⊨ (∀i : A ⇒ g). Or equivalently: ⊨ (∀i : A : g). For example, consider again the proof in Figure 3.3. It proves this formula:

\[-b \land (x^2 > 0 \Rightarrow y < 0) \land (y \leq 0 \Rightarrow b) \Rightarrow (x = 0)\]

However, by quantifying over the free variables x and y, it also proves this:

\[(\forall x, y : -b \land (x^2 > 0 \Rightarrow y < 0) \land (y \leq 0 \Rightarrow b) : x = 0)\]

Therefore, to prove a formula of the form:

\[(\forall x_1, \ldots, x_k : A_1 \land \ldots \land A_n : g)\]

we can instead just prove:

\[A_1 \land \ldots \land A_n \Rightarrow g\]

Doing this in a subproof is more tricky. Consider a proof P containing a subproof Q. Suppose Q proves A ⇒ g and suppose x is free in A ⇒ g. Can we also infer (∀x : A : g)? Well, it depends. If x is used in the parent proof P\(^3\) the proof in Q may refer to some proven fact about x in P. The latter means that g depends on a fact which is not stated in A. So, it would be incorrect to conclude (∀x : A : g); otherwise it is a sound conclusion. Since you have to keep track of references, this kind of inference can be quite error prone.

To guard against mistakes, we will introduce a so-called ANY flag. You can start a subproof Q with a keyword ANY followed by a list of variables. For example ANY x, y. It means that we start the proof Q by assuming nothing about x and y, other than the assumptions stated in Q itself. In particular, this limits the scope of x and y to Q only. It follows, when concluding the proof Q we can also universally quantify the conclusion over x and y.

There is one condition that you have to remember: since the scope of x and y is limited to Q, you are not allowed to refer to any formula in the higher level proofs that contains x or y as a free variable. It follows that using fresh variables every time you use the ANY flag is always safe (that you don’t actually need the flag anymore). When you conclude a subproof, you can only ∀-quantify over ANY variables. To do otherwise is potentially unsound, thus is forbidden.

As an example, consider the following formula:

\[(\forall i : 0 \leq i : a[i] > 0) \land x^2 > 0\]

\[\Rightarrow (\forall i : 0 \leq i : x * a[i] \neq 0)\]

\(^3\)More precisely: x occurs free in one of the formulas in the proof-context of P at the moment Q is opened.
3.7. HANDLING \( \forall \) AND \( \exists \)

A proof of its validity is shown in Figure 3.4. The subproof \( \text{pot} \) proves that \( 0 \leq i \) implies \( x \cdot a[i] \neq 0 \). However it also has an \( \text{ANY} \) \( i \) flag, which means we can also \( \forall \)-quantify over \( i \), thus drawing this conclusion instead (in step-2 of \( \text{top} \)):

\[
(\forall i: 0 \leq i: x \cdot a[i] \neq 0)
\]

Notice that the flagged variable \( i \) in the proof \( \text{pot} \) \textit{does} occur in the parent proof \( \text{top} \), namely in \( \text{top.A1} \). However it does not occur free in \( \text{top.A1} \), so it won’t pose any problem. If it does occur free in some formula \( f \) in \( \text{top} \) it is still not a problem as long as you do \textit{not} refer to \( f \) from within \( \text{pot} \).

### 3.7.2 Eliminating \( \exists \)

Sometimes you have to prove a goal with a \( \exists \)-formula as one of its assumptions. The information inside the \( \exists \)-formula is often more useful if we can somehow get rid of the \( \exists \). However, we have to be careful, because doing so may lead to an unsound conclusion. Consider the following example:

\[
(\exists i: 0 \leq i: b[i]) \land \neg b[x] \land (\forall i: b[i]: c[i])
\]

\[
\Rightarrow (\exists i :: c[i])
\]

The proof is shown in Figure 3.5. If you have a \( \exists \)-formula in the current proof-context, you can infer a new formula by replacing the \( \exists \) quantor with a \( \text{SOME} \) flag. You can also rename the bound variable. In Figure 3.5, \( A1 \) states the existence of an \( i \) satisfying \( 0 \leq i \) and \( b[i] \). The \( \text{SOME} \) flag in step-1 simply re-asserts this fact:

\[
[\text{SOME } i'] \ 0 \leq i' \land b[i']
\]

This may seem a bit pointless, but there is one important difference between \( \exists \) and \( \text{SOME} \): subsequent proof steps can now directly use the formula after the \( \text{SOME} \) flag. Before, it was not possible because it is 'packed' inside an \( \exists \) quantifier.
CHAPTER 3. LOGIC AND PROOFS

PROOF
[A1:] \( (\exists i : 0 \leq i : b[i]) \)
[A2:] \( \neg b k \)
[A3:] \( (\forall i : b[i] \Rightarrow c[i]) \)
[G :] \( (\exists i : c[i]) \)

BEGIN

1 \{\exists elimination on A1 \}
\[ \text{SOME } i' \ 0 \leq i' \land b[i'] \]

2 \{\land Elimination (R A.1.5) on 1 \}
\[ b[i'] \]

3 \{\forall Elimination (R A.1.10) on A3 and 2 \}
\[ c[i'] \]

4 \{\exists Introduction (R A.1.11) on 3 \}
\[ (\exists i' :: c[i']) \]

END

Figure 3.5: An example of how to eliminate \( \exists \).

The flag \text{SOME } i' introduces however a scope. From this point on, all references to \( i' \) refers to the \( i' \) that is introduced by the flag. In particular, if the current proof-context already had a variable \( i' \), after the \text{SOME } i' flag you are not allowed to use any assumed or derived facts on the ‘old’ \( i' \)—doing so would mean that you illegally assume something that is not there for the flagged \( i' \). If you consider this to be error prone, then just introduce a fresh variable on every \text{SOME}. For example, in Figure 3.5 introducing \text{SOME } i' in step-1 is allowed; so are \text{SOME } i or \text{SOME } j. Even this is allowed:

\[ [\text{SOME } k ] \ 0 \leq k \land b[k] \]

But be careful! Any \( k \) after this refers to the above \( k \). Now, notice that \( k \) already appears free in A2. So invoking A2 would now be an invalid step (in fact, A2 asserts something that is contradictive to the above asserted property of \( b[k] \)).

3.8 Some Useful Proof Techniques

Proof by contradiction

Sometimes, it is easier to prove a goal \( Q \) by contradiction. That is, we assume \( \neg Q \), and then show that this is not possible. Formally, this is done by invoking the following rule:

\[
\frac{\neg Q \Rightarrow \text{false}}{Q}
\]  \hspace{1cm} (3.18)

The premise \( \neg Q \Rightarrow \text{false} \) captures the fact that \( \neg Q \) is not possible. An example showing the use of this rule has been shown in Figure 3.3.

Case split

Consider the formula below:

\[
(a[0] = 0) \land (\forall i : 0 < i : 0 < a[i]) \land 0 \leq j
\]

\[
\Rightarrow
\]

\[ 0 \leq a[j] \]  \hspace{1cm} (3.19)
3.8. SOME USEFUL PROOF TECHNIQUES

PROOF top

[A1:] \( a[0] = 0 \)

[A2:] \((\forall i : 0 < i : 0 < a[i])\)

[A3:] \( 0 \leq j \)

[G :] \( 0 \leq a[j] \)

BEGIN

1 \{ follows trivially from A3 \} \( 0 < j \lor (j = 0) \)

2 \{ see subproof pot \} \( 0 < j \Rightarrow 0 \leq a[j] \)

PROOF pot

[A1:] \( 0 < j \)

[G :] \( 0 \leq a[j] \)

BEGIN

1 \{ \( \forall \)-Elimination on top, A2 using A1 \} \( 0 < a[j] \)

2 \{ follows trivially from 1 \} \( 0 \leq a[j] \)

END

3 \{ follows trivially from A1 \} \( (j = 0) \Rightarrow 0 \leq a[j] \)

4 \{ Case Split on 1,2,3 \} \( 0 \leq a[j] \)

END

Figure 3.6: An example of a proof using case split.

Informally this can be argued as follows. We only need to prove the implication for \( j \) satisfying \( 0 \leq j \). We split this in two cases: (1) if \( 0 < j \) then it is in the domain of the assumption \((\forall i : 0 < i : 0 < a[i])\), thus implying \( 0 < a[j] \); (2) if \( j = 0 \), the first assumption above says \( a[0] = 0 \). Both cases imply \( 0 \leq a[j] \).

The proof relies on analyzing two possible cases of \( j \). Distinguishing the two cases is necessary, because the ways the conclusion \( 0 \leq a[j] \) is inferred are different for each case. Formally, to do this kind of split-cases proof we invoke the following rule:

\[
\begin{align*}
P \lor Q \\
P &\Rightarrow R \\
Q &\Rightarrow R \\
\hline
R
\end{align*}
\]

So, we can infer \( R \) by first identifying two cases \( P \) and \( Q \) and proving \( P \lor Q \)—the latter is to show that \( P \) and \( Q \) sufficiently cover all possible cases. Then we prove that each of the cases implies \( R \).

The proof in Figure 3.6 shows an example. It is the formal proof of (3.19).

Splitting a quantified formula

Quantified formulas occur quite often in the specifications of programs that use arrays. During the proof, quite often we have to prove that an equality the form \( x \oplus e = (\forall i : P_1 i : Q i) \) follows from another equality: \( x = (\forall i : P_2 i : Q i) \); here \( \oplus \) is an operator or a binary function. A strategy that is often used to first prove:

\[
(\forall i : P_2 i : Q i) \oplus e = (\forall i : P_1 i : Q i)
\]

Then it follows that:
\[ x \oplus e = \{ \text{assumption} \} \]
\[ (\forall i : P_2 i : Q i) \oplus e = \{ \text{using (3.21)} \} \]
\[ (\forall i : P_1 i : Q i) \]

Notice that the strategy works even if we have another quantor instead of \( \forall \). There are a number of theorems that are often convenient for proving equalities between quantified formulas such as (3.21). Here are some examples (of the theorems):

**Theorem 3.8.1 : Domain Split**

1. \( \vdash (\forall i : P i \lor Q i : R i) = (\forall i : P i : R i) \land (\forall i : Q i : R i) \)
2. \( \vdash (\exists i : P i \lor Q i : R i) = (\exists i : P i : R i) \lor (\exists i : Q i : R i) \)

\( \square \)

The theorem above is called Domain Split because the equalities above can be used to 'split' a quantified formula on its domain. We can also split on its range, which is stated by the theorem below:

**Theorem 3.8.2 : Range Split**

1. \( \vdash (\forall i : P i : Q_1 i \land Q_2 i) = (\forall i : P i : Q_1 i) \land (\forall i : Q i : Q_2 i) \)
2. \( \vdash (\exists i : P i : Q_1 i \lor Q_2 i) = (\exists i : P i : Q_1 i) \lor (\exists i : Q i : Q_2 i) \)

\( \square \)

**Merging integer domains**

Recall that an integer domain expression is a formula of the form, for example, \( a \leq i < b \). Such a formula is used quite often, especially when specifying and proving the correctness of a program that uses arrays. Quite often, in a proof, such as in Figure 3.2, we need to convert a domain expression of the form \( a \leq i < b + 1 \) to \( a \leq i < b \lor (i = b) \). These two formulas are equal, but be careful, they are only equal if \( a, b, \) and \( i \) are all integers and if \( a \leq b \). This is expressed by the following theorem:

**Theorem 3.8.3 : Domain Merging**

Let \( a, b, \) and \( i \) be integers:

\[ \vdash a \leq b \Rightarrow (a \leq i < b + 1 = a \leq i < b \lor (i = b)) \]

\( \square \)

In a more general case, we may want to split \( a \leq i < c \) to:

\[ a \leq i < b \lor b \leq i < c \]

The two formulas are equivalent, but as before, they are only equivalent under a certain condition:

**Theorem 3.8.4 : Domain Merging**

Let \( a, b, c, \) and \( i \) be integers:

\[ \vdash a \leq b \land b \leq c \Rightarrow (a \leq i < c = a \leq i < b \lor b \leq i < c) \]

\( \square \)

Both equalities are also listed in the Appendix.
3.8. SOME USEFUL PROOF TECHNIQUES

Simplifying conditional formulas

Recall that a conditional formula is a formula of the form $P \rightarrow e_1 \mid e_2$. You may encounter this kind of formulas when proving the correctness of a program that contains an if-then-else statement. The theorem below lists a number of equalities which can be used to simplify a conditional formula.

**Theorem 3.8.5 : COND Conversion**

1. $\vdash P \Rightarrow (P \rightarrow e_1 \mid e_2 = e_1)$
2. $\vdash \neg P \Rightarrow (P \rightarrow e_1 \mid e_2 = e_2)$
3. $\vdash P \rightarrow e \mid e = e$
4. $\vdash f (P \rightarrow e_1 \mid e_2) = P \rightarrow f e_1 \mid f e_2$
5. $\vdash P \rightarrow e_1 \mid e_2 = \neg P \rightarrow e_2 \mid e_1$

Sometimes, we have to prove a formula of the form $P \rightarrow Q \mid R$. We can use the first two equalities above to simplify $P \rightarrow Q \mid R$ to either $Q$ or $R$. Unfortunately this does not always work, since we may not be able to derive either $P$ or $\neg P$. The following theorem states that we can equivalently prove $P \Rightarrow Q$ and $\neg P \Rightarrow R$ instead (but note that the theorem requires that $P$, $Q$, and $R$ are all predicates):

**Theorem 3.8.6 : COND Split**

Let $P$, $Q$, and $R$ be predicates:

$$\vdash P \rightarrow Q \mid R = (P \Rightarrow Q) \land (\neg P \Rightarrow R)$$

□

□□□

New Vocabulary

conclusion; COND coersion; COND split; conditional rewrite; domain merging; domain split; eliminating $\exists$; goal; inference rule; introducing $\forall$, $\exists$; predicate; premise; proof-context; range split; rewriting; specializing (eliminating, instantiating) $\forall$; stronger; subproof; substitution; universally true; valid; weaker
3.9 Exercise

1. Explain in words what the following formulas say:

   (a) $(\forall i, j :: (a[i] = a[j]) \Rightarrow (i = j))$
   (b) $\neg (\exists i :: 0 \leq i : a[i] \neq a[i + 1])$
   (c) $(\exists i :: (\forall j :: a[i] \leq a[j]))$
   (d) $(\forall i :: \neg (\exists j :: a[i] = b[j]))$

2. Write formulas expressing the following informal texts:

   (a) Within some domain $D$ the array $a$, when viewed as a set, is a 'subset' of array $b$.
   (b) There is a unique index $i$ in the domain $D$ such that $f(a[i]) = 0$.
   (c) The array $a$ contains at least two distinct elements.
   (d) The array $a$ has at most two elements which are equal to each other.

3. Explain in words what the following formula says, then prove it.

   $$(\forall i, j : 0 \leq i < n \land i \leq j < n : a[i] \leq a[j])$$
   $$\land$$
   $$0 < n$$
   $$\Rightarrow$$
   $$(\exists i : 0 \leq i < n : (\forall j : 0 \leq j < n : a[i] \leq a[j]))$$

   Actually, the implication is still valid even if we drop the $(\forall i, j : \ldots)$ conjunct at the left side of the $\Rightarrow$, though the proof is now more complicated and would require induction (see below).

4. Explain in words what this formula says then prove it.

   $$(\forall i : 0 \leq i : a[i] = a[i + 1])$$
   $$\Rightarrow$$
   $$(\forall i : 0 \leq i : a[i] = a[0])$$

   To prove you will need to use induction over positive integers. This is expressed by the following rule:

   $P 0$
   $$(\forall n : n \geq 0 : P n \Rightarrow P (n + 1))$$
   $$(\forall n : n \geq 0 : P n)$$

   where $n$ in the above rule in assumed to be of the integer type.

5. Suppose at some point during an execution of a program we know that this condition holds:

   $$r = (\forall k : 0 \leq k < i : b[k])$$

   A subsequent statement $S$ requires however that another condition should hold before it executes, namely:

   $$r \land b[i] = (\forall k : 0 \leq k < i + 1 : b[k])$$

   It is safe to continue with $S$ if the first condition above actually implies the second. Can you prove this? You will discover that your proof also need an additional assumption $i \geq 0$, why?
6. Suppose we mark some integers as 'blue' integers. We know the following facts about an array $a$:

(a) If $a$ contains a blue integer in the domain $0 \leq i < n$, then this integer is 2.
(b) In the domain mentioned above, $a$ does not contain any even integer.

Prove that $a$ does not contain (in the above domain) any blue integer. You can assume that you have a function $\text{blue}$ to test whether an integer is blue.

7. What does the following formula say? Prove it.

$$\neg(\forall i: 0 \leq i < n : b[i]) \land (\forall i : j \leq i < n : b[i]) \Rightarrow \neg(\forall i : 0 \leq i < j : b[i])$$

Notice that all occurrences of $i$ are bound, whereas $j$ is free.
3.10 Solution

1. (a) All elements in the array \( a \) are distinct.
   (b) All elements of the array \( a \) in the domain \( 0 \leq i \) are all the same.
   (c) The array \( a \) has a minimum element. (note: if we quantify over a finite domain, an array always has a minimum; however, this is not always true if we quantify over an infinite domain.)
   (d) \( a \) and \( b \) have no common element.

2. The formulas:
   (a) \((\forall i : D i : (\exists j : D j : a[i] = b[j]))\)
   (b) \((\exists i : D i : (f(a[i]) = 0) \land (\forall j : D j : (f(a[j]) = 0) : i = j))\)
   (c) \((\exists i, j : i \neq j : a[i] \neq a[j])\)
   (d) \((\forall i, j, k : (a[i] = a[j]) \land (a[j] = a[k]) : (i = j) \lor (j = k))\)

3. It says: "if the array \( a \) is ascending in the domain \( 0 \leq i < n \), for a positive \( n \), then \( a \) has a minimum element in that domain". Intuitively, this is quite obvious. The formal proof is shown below. Because the array is ascending, we know what its minimum is, namely \( a[0] \). This simplifies the proof:

PROOF top

[A1:] \((\forall i, j : 0 \leq i < n \land i \leq j < n : a[i] \leq a[j])\)

[A2:] \(0 < n\)

[G:] \((\exists i : 0 \leq i < n : (\forall j : 0 \leq j < n : a[i] \leq a[j]))\)

BEGIN

1 { see the subproof below } \((\forall j : 0 \leq j < n : a[0] \leq a[j])\)

PROOF

ANY j

[A:] \(0 \leq j < n\)

[G:] \(a[0] \leq a[j]\)

BEGIN

1 { \( \forall \)-elimination on top.A1 } \(0 \leq 0 < n \land 0 \leq j < n \Rightarrow a[0] \leq a[j]\)

2 { Modus Ponens on 1, using A and \( 0 \leq 0 < n \), which is implied by top.A2 } \(a[0] \leq a[j]\)

END

2 { \( \exists \)-introduction on 1; A2 implies \( 0 \leq 0 < n \) }

\((\exists i : 0 \leq i < n : (\forall j : 0 \leq j < n : a[i] \leq a[j]))\)

END

4. If within the specified domain every element in the array \( a \) is equal to the next one, then all elements of \( a \) in the domain is equal to \( a[0] \).

PROOF top

[A:] \((\forall i : 0 \leq i : a[i] = a[i + 1])\)

[G:] \((\forall i : 0 \leq i : a[i] = a[0])\)

BEGIN
3.10. SOLUTION

1 \{ trivial \} \ a[0] = a[0]

2 \{ see the subproof below \}
(\forall i : i \geq 0 : (a[i] = a[0]) \Rightarrow (a[i + 1] = a[0]))

PROOF
ANY i
[A1:] i \geq 0
[A2:] a[i] = a[0]
[G :] a[i + 1] = a[0]

BEGIN

1 \{ \forall\text{-elimination on top.A, using A1 } \} \ a[i] = a[i + 1]

2 \{ follows from A1 and 1 \} \ a[i + 1] = a[0]

END

3 \{ (positive integers) induction on 1 and 2 \}
(\forall i : 0 \leq i : a[i + 1] = a[0])

END

5. We are going to prove this with an equational proof. Notice that the Domain Merging step in the proof requires 0\leq i, which is not in the original assumption. This is added in the proof below.

Note that in general you should not casually add assumptions just so that you can complete your proof. By doing so you may not solving the original problem. The new assumptions may also turn out to be unrealistic. In some cases however, the original problem may turn out to be unsolvable. In this case adding assumptions may be reasonable to see under which 'wider' conditions the problem is solvable. Note also that adding assumptions also weaken the result that you obtain.

EQUATIONAL PROOF

[A1:] \ r = (\forall k : 0 \leq k < i : b[k])
[A2:] \ 0 \leq i

(\forall k : 0 \leq k < i + 1 : b[k])
= \{ \text{Domain Merging (Theorem A.4.16), justified by A2 } \}
(\forall k : 0 \leq k < i \lor (k = i) : b[k])
= \{ \text{Domain Split (Theorem A.4.12) } \}
(\forall k : 0 \leq k < i : b[k]) \land (\forall k : k = i : b[k])
= \{ \text{quantification over singleton (Theorem A.4.10) } \}
(\forall k : 0 \leq k < i : b[k]) \land b[i]
= \{ \text{rewrite with A1 } \}
\ r \land a[i]

END

6. This is what you have to prove:

(\forall i : 0 \leq i < n : \text{blue} (a[i]) \Rightarrow (a[i] = 2)) \land
(\forall i : 0 \leq i < n : \neg \text{even} (a[i]))
\Rightarrow
\neg (\exists i : 0 \leq i < n : \text{blue} (a[i]))
We are going to prove this by contradiction. Notice the subproof contra, where we assume the negation of the right hand side of the ⇒ above, and try to derive contradiction (false) out of it.

**PROOF top**

[A1:] \((\forall i: 0 \leq i < n: \text{blue}(a[i]) \Rightarrow (a[i] = 2))\)

[A2:] \((\forall i: 0 \leq i < n: \neg(\text{even}(a[i])))\)

[G :1] \(\neg(\exists i: 0 \leq i < n: \text{blue}(a[i]))\)

BEGIN

1 \{ see the subproof below \} \((\exists i: 0 \leq i < n: \text{blue}(a[i])) \Rightarrow \text{false}\)

**PROOF contra**

[A:] \((\exists i: 0 \leq i < n: \text{blue}(a[i]))\)

[G:] false

BEGIN

1 \{ \exists-elimination on A \}

[SOME k] \(0 \leq k < n \land \text{blue}(a[k])\)

2 \{ \forall-elimination on top.A1, using the first conjunct of 1 \}

\(\text{blue}(a[k]) \Rightarrow (a[k] = 2)\)

3 \{ Modus Ponens on 2, using the second conjunct of 1 \}

\(a[k] = 2\)

4 \{ \forall-elimination on top.A2, using the first conjunct of 1 \}

\(\neg(\text{even}(a[k]))\)

5 \{ contradiction between 3 and 4 \}

\(\text{false}\)

END

2 \{ Contradiction Rule on 1 \}

\(\neg(\exists i: 0 \leq i < n: \text{blue} a[i])\)

END

7. In words the formula says:

If \(b\) contains a false in the domain \(0 \leq i < n\) and furthermore in the domain \(j \leq i < n\) all its elements are true, then the false element must be in the domain \(0 \leq i < j\).

We’ll leave the proof for you.
In Chapter 2 we have seen that the existential quantifier $\exists$ is needed to specify a program that tests whether an array contains a value that satisfies a certain property. Similarly, the universal quantifier $\forall$ is needed to specify a program that tests whether all elements of an array satisfy a certain property. Unfortunately, with only $\exists$ and $\forall$ we still cannot specify a program that, for example, computes the sum of all elements in an array. To solve this problem we need to extend our expression language with more quantifiers. The most common notation used to denote the sum over some specified domain is, for example:

$$(\sum i : 0 \leq i < n : i)$$

or this:

$$\Sigma_{0 \leq i < n} i$$

where $i$ is assumed to be of type integer. Both denote the sum of all (integer) values $i$ over the domain $0 \leq i < n$. These notations has a drawback. An operation that we often do during a proof is splitting the domain of a quantification. For example, we may want to rewrite:

$$(\sum i : P \lor Q : i) \text{ to } (\sum i : P : i) + (\sum i : Q : i)$$

Unfortunately, the two expressions are not in general equal. They are only equal if $P$ and $Q$ specify disjoint domains, else the two separate sums in the formula to the right will add some $i$ twice. This often causes confusion and mistakes by students, so we decide to use a different notation. We will denote the sum of all integer values in the domain $0 \leq i < n$ as follows:

$$\text{SUM} [0 \ldots n]$$

where $\text{SUM}$ is an ordinary list function (rather than a quantifier) which can be defined recursively as follows:

$$\text{SUM} [] = 0$$

$$\text{SUM} (a : s) = a + \text{SUM} s$$
stating that the sum of an empty list is 0, and the sum of a list whose first element is \(a\) is equal to \(a\) plus the sum of the rest of the list. The function \(\text{SUM}\) satisfies this property:

\[
\text{SUM} (s \leftrightarrow t) = \text{SUM} s + \text{SUM} t
\]

where \(s \leftrightarrow t\) denotes the list obtained by concatenating \(s\) in front of \(t\). As before, the above equality expresses how to 'split the domain' of \(\text{SUM}\). But now the equality holds without restriction, so one is less likely to make a mistake.

To use the above kind of notation we will need to extend \(\text{Form}\), the language we use to write formulas, with a list notation. We will use a slightly modified notation borrowed from Haskell [26]. Haskell is a functional programming language. Lists are used a lot in functional programming. Do keep in mind that here we do not use the list notation to program, as you would do in Haskell. Instead, we use the notation purely for expressing specifications.

### 4.1 Basic Notation and Properties

**Basic operators**

The operator : is called the \(\text{cons}\) operator. For example \(x : s\) denotes the list obtained by adding \(x\) in front of \(s\). The operator \(\leftrightarrow\) is called \(\text{append}\). For example \(s \leftrightarrow t\) denotes the list obtained by appending \(s\) in front of \(t\). The formula \(x \in s\) means that \(x\) is an element of the list \(s\). Append and \(\in\) can be defined recursively as follows:

**Definition 4.1.1** : \(\leftrightarrow\) AND \(\in\)

\[
\begin{align*}
[] \leftrightarrow t &= t \\
(x : s) \leftrightarrow t &= x : (s \leftrightarrow t) \\
x \in [] &= \text{false} \\
x \in (y : s) &= (x = y) \lor x \in s
\end{align*}
\]

\(\checkmark\)

The \(\leftrightarrow\) operator has a higher priority than :, and : has a higher priority than \(\in\); \(\leftrightarrow\) is left and right associative\(^1\), whereas : is only right associative —see also Table 2.1 in Chapter 2.

**Concrete list and enumeration**

The notation like \([i,j,k]\) is used to denote a concrete list, in this case the list contains three elements: \(i\), \(j\), and \(k\). Empty list is denoted by \([]\). You can also specify lists using expressions of the form \([i \ldots j]\), where \(i\) and \(j\) are of the type integer. The notation is called \(\text{list enumeration}\); it denotes the list of all integers from \(i\) up to, but not including, \(j\). Below are a number of basic theorems about list enumeration.

**Theorem 4.1.2** : EMPTY ENUMERATION

\(\vdash j \leq i \Rightarrow (i \ldots j) = []\)

**Theorem 4.1.3** : SINGLETON ENUMERATION

\(\vdash [i \ldots i+1] = [i]\)

**Theorem 4.1.4** : ENUMERATION MEMBERSHIP

\(\vdash j \in [i \ldots k] = i \leq j < k\)

---

\(^1\)As a functional language \(\leftrightarrow\) is usually only left associative. This is for an efficiency reason. However here we use \(\leftrightarrow\) for expressing specifications where efficiency is not an issue.
Theorem 4.1.5: Enumeration Split

1. ⊢ i ≤ j ⇒ ([i...j] + 1) = [i...j] + [j]
2. ⊢ i ≤ j ≤ k ⇒ ([i...k] = [i...j] + [j...k])

List comprehension

You may be familiar with this notation from set theory: \{ f x \mid x ∈ A \} denoting the set of all f x where x is taken from the set A. The notation is called set comprehension. We will also use something similar for list, called list comprehension. We will write, for example:

\[ x \ast x \mid x \text{ from } s, x > 0 \]

to denote the list consisting of all x * x values, where x is taken from the list s, and moreover x should satisfy x > 0. This is a quite convenient notation. For example, to specify the sum of all positive elements of an array a in the domain 0 ≤ i < n we can write:

\[ \text{SUM} \left[ x \mid x \text{ from } a \mid 0 \ldots n \right], x > 0 \]

The list s in \[ e i \mid i \text{ from } s, P i \] is also called generator. The P is called the condition of the comprehension, and e is called the transformer of the comprehension.

Sometimes, we also write a comprehension without the P part, e.g.: \[ e i \mid i \text{ from } s \], which means that we have no condition to constrain the i's. Formally, you can consider \[ e i \mid i \text{ from } s \] to be an abbreviation of \[ e i \mid i \text{ from } s, \text{true} \]. Below you can find some basic theorems about list comprehension.

Theorem 4.1.6: Empty Comprehension

1. \[ e i \mid i \text{ from } [], P \] = []
2. \[ e i \mid i \text{ from } s, \text{false} \] = []
3. \[ (e i \mid i \text{ from } s, P i) = [] \] = (\(\forall i : i ∈ s : ¬(P i)\))

Theorem 4.1.7: Singleton Comprehension

1. \[ e i \mid i \text{ from } [x] \] = [e x]
2. \[ e i \mid i \text{ from } [x], P i \] = P x → [e x] | []

Theorem 4.1.8: Comprehension Membership

1. \( x ∈ [e i \mid i \text{ from } s, P i] \) = (\(∃i : i ∈ s \land P i : x = e i\))
2. \( x ∈ [i \mid i \text{ from } s, P i] \) = \( x ∈ s \land P x \)

Theorem 4.1.9: Comprehension Split

\[ e i \mid i \text{ from } (s ++ t), P i \] = [e i \mid i \text{ from } s, P] ++ [e i \mid i \text{ from } t, P]

A comprehension whose generator is another comprehension is called nested. The following theorem tells us how to ‘collapse’ a nested comprehension.

Theorem 4.1.10: Nested Comprehension

1. \[ e i \mid i \text{ from } [e_2 j \mid j \text{ from } s, P j], Q i \] = \[ e_1 (e_2 j) \mid j \text{ from } s, P j \land Q (e_2 j) \]
2. \[ e_1 i \mid i \text{ from } [e_2 j \mid j \text{ from } s, P j] \] = \[ e_1 (e_2 j) \mid j \text{ from } s, P j \]
Specifying array properties via list

If \( a \) is an array and \( s \) is a list, the notation \( a s \) denotes the list of all elements of \( a \) whose indices is taken from \( s \). It is called listed array. More precisely:

\[
a s = [a[i] \mid i \text{ from } s]
\]

For example \( a[0...n) \) denotes the list consisting of \( a[0], \ldots, a[n-1] \). The notation allows list functions like \( \text{SUM} \) to be used for specifying properties over arrays. For example, as in \( \text{SUM}(a[0...n)) > 0 \).

The analogous of Theorems 4.1.6 ... 4.1.5 for enumeration also hold for listed array. For example, for splitting a listed array we have:

**Theorem 4.1.11 : ARRAY Split**

1. \( i \leq j \Rightarrow (a[i...j+1) = a[i...j) ++ [a[j]]) \)
2. \( i \leq j \leq k \Rightarrow (a[i...k) = a[i...j) ++ a[j...k]) \)

**Some standard list functions**

We have mentioned \( \text{SUM} \), which is a function used to specify the sum of all elements in a list. We can also define \( \text{COUNT} \) to specify the number of elements in a list; \( \text{MAX} \) to specify the greatest element in a list; and \( \text{MIN} \) to specify the smallest element in a list. Their definitions are shown below.

**Definition 4.1.12 : SUM, COUNT, MAX, AND MIN**

\[
\begin{align*}
\text{SUM} & \quad [ ] \quad = \quad 0 \\
\text{SUM} & \quad (x : s) \quad = \quad x + \text{SUM} \ s \\
\text{COUNT} & \quad [ ] \quad = \quad 0 \\
\text{COUNT} & \quad (x : s) \quad = \quad 1 + \text{COUNT} \ s \\
\text{MAX} & \quad [x] \quad = \quad x \\
\text{MAX} & \quad (x : s) \quad = \quad x \ max \text{MAX} \ s \quad , \quad \text{provided } s \text{ is non-empty.} \\
\text{MIN} & \quad [x] \quad = \quad x \\
\text{MIN} & \quad (x : s) \quad = \quad x \ min \text{MIN} \ s \quad , \quad \text{provided } s \text{ is non-empty.}
\end{align*}
\]

Notice that \( \text{MAX} \) and \( \text{MIN} \) are not defined when the list is empty.

**Splitting function over list**

A function like \( \text{SUM} \) distributes over \( ++ \). That is:

\[
\text{SUM} \ (s ++ t) = \text{SUM} \ s + \text{SUM} \ t
\]

This is a useful properties when implementing \( \text{SUM} \). For example, it allows us to compute \( \text{SUM} \) recursively by splitting the input list. The property above suggests that it should not matter, in principle, where we split the list.

In Algebra people have a nice name for a function with a \( \text{SUM} \)-like property above: homomorphism\(^2\) More precisely, given a function \( f \) and two operators \( \oplus \) and \( \otimes \), the function \( f \) is a homomorphism if it satisfies:

\[
f \ (x \oplus y) = f \ x \otimes f \ y
\]

Many useful list functions are homomorphisms over \( ++ \), for example:

\(^2\)In Algebra a homomorphism is a structure perserving map from one set of values to another. In the case of \( \text{SUM} \), it maps lists to integers. The property above can be also interpreted as follows: \( \text{SUM} \) consistently maps lists constructed with \( ++ \) to integers constructed with \( + \).
4.2. PROOFS

**Theorem 4.1.13 : Homomorphism of List Functions**

1. \( \vdash x \in (s ++ t) \quad \Rightarrow \quad x \in s \lor x \in t \)
2. \( \vdash \text{SUM}(s ++ t) = \text{SUM} s + \text{SUM} t \)
3. \( \vdash \text{COUNT}(s ++ t) = \text{COUNT} s + \text{COUNT} t \)

Some functions are only homomorphic under a certain condition. For example \( \text{MAX}(s ++ t) \) is equal to \( \text{MAX} s \ \text{max} \ \text{MAX} t \), but only if both \( s \) and \( t \) are non-empty (since \( \text{MAX} \) is undefined on an empty list):

**Theorem 4.1.14 : Homomorphism of MAX and MIN**

1. \( \vdash s \neq [] \land t \neq [] \quad \Rightarrow \quad \text{MAX}(s ++ t) = \text{MAX} s \ \text{max} \ \text{MAX} t \)
2. \( \vdash s \neq [] \land t \neq [] \quad \Rightarrow \quad \text{MIN}(s ++ t) = \text{MIN} s \ \text{min} \ \text{MIN} t \)

4.2 Proofs

We will show several examples of how to prove properties about list. Consider a list \( s \). We will prove that if \( s \) contains an element \( x \) satisfying \( P \), then \( [e i \mid i \text{ from } s, P i] \) is not empty. In fact, the two statements are equivalent, as formulated by the theorem below:

**Theorem 4.2.1 : Non-empty Comprehension**

\( \vdash (\exists i : i \in s : P i) = \neg([e i \mid i \text{ from } s, P i] = []) \)

The proof is quite simple:

**EQUATIONAL PROOF**

\[
\neg([e i \mid i \text{ from } s, P i] = [] )
\]

\[
= \{ \text{rewrite with Theorem Empty Comprehension (T 4.1.6)} \} \\
\neg(\forall i : i \in s : \neg(P i))
\]

\[
= \{ \text{rewrite with Theorem Negate } \forall \ (T A.4.7) \} \\
(\exists i : i \in s : \neg(P i))
\]

\[
= \{ \text{trivial} \} \\
(\exists i : i \in s : P i)
\]

In the next example we will prove that \( \text{COUNT} s \) cannot be negative. Stated formally, we want to prove this:

\( (\forall s :: \text{COUNT} s \geq 0) \)

Intuitively, the property is obvious. To prove it formally is a bit more complicated. It cannot be proven using basic equalities like in the previous proof. It requires something else, namely induction.

The induction principle over lists (also called list induction) is analogous to the natural number induction. It says that to prove that \( P \) holds for all lists \( s \), it is sufficient to show \( P [] \) and \( P s \Rightarrow P (x : s) \), for arbitrary \( x \) and \( s \). Formally:
PROOF

\[ \forall s :: \text{COUNT } s \geq 0 \]
BEGIN

1 \{ follows from the definition of \text{COUNT} \} \quad \text{COUNT } [] \geq 0
2 \{ see the proof below \} \quad (\forall x, s :: \text{COUNT } s \geq 0 \Rightarrow \text{COUNT } (x : s) \geq 0)

BEGIN

1 \{ from the definition of \text{COUNT} \} \quad \text{COUNT } (x : s) = 1 + \text{COUNT } s
2 \{ follows from 1 \} \quad \text{COUNT } (x : s) \geq \text{COUNT } s
3 \{ follows from 1 and \text{A} \} \quad \text{COUNT } (x : s) \geq 0

END

3 \{ list induction on 1 and 2 \} \quad (\forall s :: \text{COUNT } s \geq 0)

END

Figure 4.1: An example of a proof using induction.

Rule 4.2.2: List Induction

\[
\begin{align*}
P [] \\
(\forall x, s :: P s \Rightarrow P (x : s)) \\
(\forall s :: P s)
\end{align*}
\]

The \( P s \) in the second premise is also called induction hypothesis.

To use induction to prove that \( \text{COUNT } s \geq 0 \), we have to prove two things: (1) \( \text{COUNT } [] \geq 0 \), which is quite trivial, and (2) \( \text{COUNT } s \geq 0 \) implies \( \text{COUNT } (x : s) \) is also at least 0. The complete proof is shown in Figure 4.1. The step marked with (*) is where we use the induction hypothesis (A1).

New Vocabulary

distributivity; homomorphism; lambda notation; list comprehension (generator, condition, transformer, nested); list enumeration; list induction
4.3 Exercises

1. Write formulas to express the following texts:

   (a) \( k \) is the index pointing to a maximum element of an array \( a \) in the domain \( 0 \leq i < n \).

   (b) In the domain \( 0 \leq i < n \), the array \( a \) has at least two maximum elements.

   (c) \( k \) is the length of the longest prefix of an array \( a \) within the domain \( 0 \leq i < n \) that consists only of zeros.

2. Prove that if \( s \) contains only even integers then \( \text{SUM} s \) is also even.

3. The functions \( \text{map} \) and \( \text{filter} \) are defined as follows:

\[
\begin{align*}
\text{map} \ f \ [] & = [] \\
\text{map} \ f \ (x : s) & = f \ x : \text{map} \ f \ s \\
\text{filter} \ p \ [] & = [] \\
\text{filter} \ p \ (x : s) & = (p \ x \to [x] \mid []) ++ \text{filter} \ p \ s
\end{align*}
\]

Function composition \( \circ \) is defined as follows:

\[
(f \circ g) \ x = f \ (g \ x)
\]

Prove the following properties:

(a) \( \text{map} \ f \ (\text{map} \ g \ s) = \text{map} \ (f \circ g) \ s \)

(b) \( \text{map} \ f \ (\text{filter} \ p \ s) = [f \ x \mid x \text{ from} \ s, p \ x] \)

4. Consider a function \( f \) defined recursively as follows:

\[
\begin{align*}
f \ [] & = e \\
f \ (x : s) & = x \oplus f \ s
\end{align*}
\]

Prove that if \( \oplus \) is associative with \( e \) as its unit, then \( f \) is a homomorphism:

\[
f \ (s \oplus t) = f \ s \oplus f \ t
\]

Based on this, give some examples of a useful \( f \) which is homomorphic over \( \oplus \).

5. Theorem 4.1.11 is used to split an 'enumerated' array. Prove it.
4.4 Solution

1. (a) \[ a[k] = \text{MAX}(a[0...n]) \]
   (b) OPEN \[ \text{COUNT} \[ i \mid i \text{ from } [0...n], a[i] = \text{MAX}(a[0...n]) \] \geq 2 \]
   (c) \[ k = \text{MAX} \[ i \mid i \text{ from } [0...n+1], (\forall j : 0 \leq j < i : a[j] = 0) \]

2. PROOF

   \[ \text{P} s = (\forall x : x \in s : \text{even } x) \]
   \[ \text{G} (\forall s :: P s \Rightarrow \text{even} \( \text{SUM} s \)) \]

   BEGIN

   \begin{enumerate}
   \item { follows from def. of \text{SUM} } \text{even} \( \text{SUM} [\] \)
   \item { \lor-introduction on 1 } \text{¬P } [\] \lor \text{even} \( \text{SUM} [\] \)
   \item { rewrite 2 using } \text{p } \Rightarrow \text{q } = \text{¬p } \lor \text{q } \text{P } [\] \Rightarrow \text{even} \( \text{SUM} [\] \)
   \item { see proof below } \text{(\forall x,s :: (P s \Rightarrow \text{even}(\text{SUM} s)) \Rightarrow (P (x : s) \Rightarrow \text{even}(\text{SUM} (x : s))))} \]
   \end{enumerate}

   PROOF Pinduct

   \begin{enumerate}
   \item \text{ANY x,s} \]
   \item \text{P s} \Rightarrow \text{even}(\text{SUM} s) \]
   \item \text{P (x : s)} \Rightarrow \text{even}(\text{SUM} (x : s)) \]
   \end{enumerate}

   see the subproof below

   \begin{enumerate}
   \item { rewrite proof above } \text{(\forall x,s :: (P s \Rightarrow \text{even}(\text{SUM} s)) \Rightarrow (P (x : s) \Rightarrow \text{even}(\text{SUM} (x : s))))} \]
   \end{enumerate}

   \begin{enumerate}
   \item { \text{rewrite with def. of P } } \text{even}(\text{SUM} [\] \)
   \item { \text{rewrite with def. of ∩ } } \text{(\forall y : y \in s \lor (y = x) : even y)} \]
   \item { domain split on 2, and quantification over singleton } \text{(\forall y : y \in s : even y) } \land \text{even x} \]
   \item { rewrite with def. of P } \text{P s } \land \text{even x} \]
   \item \text{Modus Ponens on Pinduct.A using first conjunct of 4 } \text{even}(\text{SUM} s) \]
   \item { follows from 5 and second conjunct and 4 } \text{even(x + SUM s)} \]
   \item { rewrite with def. of SUM } \text{even}(\text{SUM} (x : s)) \]
   \end{enumerate}

   END

   \begin{enumerate}
   \item { list induction on 3 and 4 } \text{(\forall s :: P s \Rightarrow \text{even}(\text{SUM} s))} \]
   \end{enumerate}

   END

3. Here is the proof of the first property \text{map f (map g s) = map (f ◦ g) s}. We introduce some abbreviations to save some writing.

   PROOF

   \[ \text{D1} : \text{L s = map f (map g s)} \]
   \[ \text{D2} : \text{R s = map (f ◦ g) s} \]
   \[ \text{G} (\forall s :: \text{L s} = \text{R s}) \]

   BEGIN

   END

   \begin{enumerate}
   \item \text{Here is the proof of the first property map f (map g s) = map (f ◦ g) s. We introduce some abbreviations to save some writing.} \]
   \end{enumerate}
4.4. SOLUTION

1 \{ see the equational proof below \} \quad L s = R s

EQUATIONAL PROOF

L s = \{ def. of L \}
map f (map g s) = \{ def. of map \}
map f s = \{ def. of map \}
\[
= \{ \text{def. of } R \}
\]

END

2 \{ see the subproof below \} \quad (\forall x, s : L s = R s : L (x : s) = R (x : s))

PROOF Pinduct

[ANY x, s]
[A:] L s = R s
[G:] L (x : s) = R (x : s)

See the equational proof below:

EQUATIONAL PROOF

L (x : s) = \{ def. of L \}
map f (map g (x : s)) = \{ def. of map \}
map f (g x : map g s) = \{ def. of map \}
f (g x) : map f (map g s)
= \{ def. of \circ \text{ and main.D1} \}
(f \circ g) x : L s = \{ assumption Pinduct.A \}
(f \circ g) x : R s = \{ def. of R \}
(f \circ g) x : map (f \circ g) s = \{ def. of map \}
map (f \circ g) (x : s) = \{ def. of R \}
R (x : s)

END

3 \{ list induction on (1) and (2) \}

(\forall s :: L s = R s)

END

The proof of the second property is left to you.

4. We can prove the homomorphism of f by induction over s:

PROOF \[G:] (\forall s :: f (s+ t) = f s \oplus f t)\]

BEGIN
1 \{ \text{ see the subproof below } \}
\[
f ([|]+ + t) = f t
\]

\text{EQUATIONAL PROOF } \text{Pbase}

\[
f ([|]+ + t) = \{ \text{ definition of } ++ \} f t = \{ e \text{ is the unit element of } \oplus \} e \oplus f t \]

\text{END}

2 \{ \text{ see the subproof below } \}
\[
(\forall x, s :: (f (s++ t) = f s \oplus f t) \Rightarrow (f ((x : s)++ t) = f (x : s) \oplus f t))
\]

\text{PROOF } \text{Pinduct}

\text{[ANYx, s]}

\text{[A::] } f (s++ t) = f s \oplus f t

\text{[G::] } f ((x : s)++ t) = f (x : s) \oplus f t

\text{See the equational proof below:}

\text{EQUATIONAL PROOF}

\[
f ((x : s)++ t) = \{ \text{ definition of } ++ \} f (x : (s++ t)) = \{ \text{ definition of } f \} x \oplus (f (s++ t)) = \{ \text{ the assumption } A \} x \oplus (f s \oplus f t) = \{ \oplus \text{ is associative } \} (x \oplus f s) \oplus f t = \{ \text{ definition } f \} f (x : s) \oplus f t \]

\text{END}

3 \{ \text{ list induction 1 and 2 } \}
\[
(\forall s :: f (s++ t) = f s \oplus f t)
\]

\text{END}

It follows that \text{SUM} and \text{COUNT} are homomorphic over ++. Another example is the following function to 'flatten' a list of lists:

\[
\begin{align*}
\text{concat } [] &= [] \\
\text{concat } (s : ss) &= s++ \text{concat } ss
\end{align*}
\]
program and Specification

To discuss programs we will assume a small and simple programming language which we will just call uPL. It has the basic features of an imperative language, but lacks sophisticated features such as pointers, exception, and concurrency. This is a deliberate choice to let you focus on the basic principles. Of course in practice, when you deal with real programs, you also have to deal with advanced features, for which you will need a programming logic which is more advanced than uPL’s logic. You can find such a logic in the literature, e.g. [23] to handle C programs, or [14] for Java programs.

All programs we will discuss in this book are toy programs; these are small programs designed for exposing the principles and the use of our proof techniques. They are not specifically made to solve real problems, but you should not immediately dismiss their value because of this. In fact, proving the correctness of some toy programs is more challenging than that of many real programs.

5.1 Program

As an example here is again the program Euclid; it is already in of a uPL:

```plaintext
Euclid(x,y : int) : int {
    while x<>y do if x>y then x:=x-y else y:=y-x ;
    return x
}
```

Available statements in uPL are assignments, sequence, while loop, if-then-else, and a limited form of program call; goto and exception are excluded. Available types are basic types and compound types. Basic types are () (void), int, bool, char, and string. Compound types are array and record. In Appendix C you can find uPL’s EBNF grammar.
CHAPTER 5. PROGRAM AND SPECIFICATION

Array

Arrays can be one dimensional or multi-dimensional. For example, `int[]` is the type of one dimensional arrays of integers and `int[][]` is the type of two dimensional arrays of integers.

The `i`-th element of a one dimensional array `a` is denoted by `a[i]`. If `a` is two dimensional, the notation `a[i][j]` is used to refer to its elements.

To keep uPL’s logic simple we will assume that all uPL arrays are infinitely large. So, there can be no error because an array is accessed outside its range. Of course, in a real setup you should not rely on such an assumption.

Record

A record type can be defined like this:

```
record Person = {name:string; age:int}
```

The notation used to access the fields of a record is quite standard: if `r` is a value of type `Person`, `r.name` and `r.age` refer to the values stored in `r`’s fields.

Operators

Most imperative programming languages only allow simple expressions in their programs. In particular, they do not typically allow us to write a quantified expression like `(∃i :: a[i] = 0)`. So, we will not assume it either in uPL. Available uPL operators are listed below —most of them are quite standard. The operators `∧`, `∨`, and `¬` are used to write respectively the Boolean `and`, `or`, and `not` operators. Other languages often denote them by `&&`, `||`, and `!`. All uPL operators are also Form operators, and all uPL expressions are also Form expressions. However, not all Form expressions are also uPL expressions. In particular, as said above, quantified expressions are not allowed in uPL.

```
∧ (L)    ∨ (LR)    =
* (LR)    + (LR)    mod, div, max (LR), min (LR)
<, >, ≠, ≤, ≥
```

The operators are listed in the decreasing order of priority. The flag L and R specify their associativity —see Chapter 2 for their meaning.

Note that the symbol `=` denotes the operator used to test equality. It does not denote assignment as for example in Java. It is not associative, and unlike in many programming languages, `=` in uPL has a low priority.

Local Variables

uPL allows you to declare local variables. In the example below `h` is declared as a local variable:

```
SWAP (OUT x,y:int) : ()
{ var h : int ;
  h := x ; x := y ; y := h }
```
5.1. PROGRAM

Aliasing

A computer’s memory can be thought to consists of data cells. Program variables represent data stored in these cells. To access a specific cell, the computer must know which cell it is. So, each cell has an address, also called reference or pointer. This address is however specified in a binary string, which is too difficult for human to remember. So, we use symbolic names like $x$ and $y$ which are easier to remember. A compiler maps these names to binary addresses pointing to where the corresponding data are stored in the actual memory.

In Java an assignment $x=y$ will make the value of $x$ equal to $y$. However when $x$ and $y$ represent objects the actual values of these variables are pointers to their respective objects. The assignment will actually just copy the pointer to $y$’s object to $x$. So now both point to the same object. We also say that $x$ becomes an alias of $y$. Aliasing improves efficiency and is a powerful feature, but it also greatly complicates reasoning. If you change $y$, the logic will have to take into account that the change will also influence all aliases of $y$. In general the aliases cannot be determined statically (that is, just by analyzing the syntax of a program).

For simplicity in uPL we do not have aliasing. More specifically: two distinct variables names, e.g. $x$ and $y$, represent two totally disjoint sets of memory cells. So, an assignment to one will not affect the other. It also follows that a uPL assignment $x:=y$ does not copy a pointer as in Java. Instead, it copies the value of $y$ to $x$, even if $y$ is a large data structure.

Of course for languages where aliasing is pervasive, like Java, this non-aliasing assumption is unrealistic — you would need a more powerful logic to deal with the issue.

Parameter Passing

Parameters are used to pass data to and from a program. The most common ways to pass parameters to a program are the so-called pass-by-value and pass-by-reference. Consider a program with header $P(a)$. Suppose we want to pass an expression $x+y$ as a parameter to $P$ in the call $P(x+y)$. To pass $x+y$ by value means that it is first evaluated (computed), and the resulting value is copied to $a$, which acts as a local variable to $P$. Since $a$ is local to $P$, whatever $P$ does to $a$ will not influence any variable of its caller.

Now consider a program with header $Q(b)$, which is called within another program $R$:

```plaintext
R() {var x:int ... ; Q(x) ; ... }
```

Rather than passing the value of $x$ to $Q$ we can also pass its reference (address). This is called pass-by-reference. However, unlike in the pass-by-value scheme, changes made by $Q$ on this $x$ will affect the $x$ in $R$. In this respect pass-by-reference is more error prone. On the other hand, it is more efficient, especially if you have to pass large data.

There are some technical subtleties involved in the logical treatment of pass-by-reference, in particular in conjunction with aliasing and concurrency issues. In uPL we decide not to have pass-by-reference parameters. Instead we have pass-by-copy-restore.

When a parameter $x$ is passed by copy-restore to a program $P$, it is first evaluated. The result is copied to a local variable of $P$, as in pass-by-value. However, when $P$ is about to return, the final value of this local variable is copied over to $x$ — this is called, perhaps a bit misleading, restore action. This scheme is just as inefficient as pass-by-value. However, under some conditions it can be equivalently replaced by pass-by-reference.

Furthermore, in uPL a parameter of an array type is by default passed by copy-restore, except when it is marked with the `READ` keyword. Parameters of other types are by default passed by value, unless if they are marked with the `OUT` keyword.

A uPL program can return a value. The usual `return` statement is used to return a value. However, to simplify the logic `return` can only appear as the last statement in the body of a uPL program.
Specification: this program searches an element in \textit{a}, returning the position where it can be found.

\begin{verbatim}
BS(READ a:int[], i,s:int) : int
{ var m,mid,k : int ;
k := 0 ;
m := s ;
while m \neq k + 1 do
    { mid := (k+m) div 2 ;
        if a[mid]<=i then k:=mid else m:=mid } ;
    if a[k]=i then skip else k=-1 ;
return k
}
\end{verbatim}

Figure 5.1: An example of an informal specification

Program Call

To simplify the logic the syntax of program call will be restricted. We will discuss this in Chapter 7.

Scoping

To simplify the logic, any variable in a uPL program should be either a local variable or one of the program's parameters. This means that a uPL program is not allowed to access a global variable, unless it is passed to it as a parameter.

5.2 Specification

A specification of a program is just a piece of text describing what we expect the program will do. Many people just use plain English to write a specification, e.g. as in Figure 5.1. Such a specification is called \textit{informal}. Most specifications in practice are informal, which is not surprising because informal specifications are quite easy to write. However, we cannot verify a program with such a specification because it is often imprecise and ambiguous. For example, in the specification in Figure 5.1 it is not clear, unless we read the code: where does 'it' refer to? Does it mean \textit{i} or perhaps \textit{s}? And what to expect when the searched element cannot be found.

Rather than plain English, we can use a formal language like \textit{Form} to express specifications. Sentences of a formal language has a well-defined meaning. A specification expressed in a formal language is called a \textit{formal specification}.

You can express a lot with \textit{Form}; but it is still a a small specification language. In practice you may prefer more sophisticated languages, such as OCL (Object Constraint Language) [28], JML (Java Modelling Language) [18], or Z [8]. OCL is part of the UML methodology. JML is used to specify Java programs. Z is based on a set theory and is more generic.

There is by the way a trade-off to be made in the choice of the specification language. A more expressive specification language allows more delicate properties of a program to be expressed. But usually these properties are also more difficult to prove. The situation is reversed if we choose for a less expressive language.

We will specify programs and statements with so-called Hoare triples [13]. Let \( S \) be a uPL program or statement, and \( P,Q \) be \textit{Form} predicates. A Hoare triple is a triple of the form:

\[ \{ \ast P \ast \} \ S \ { \ast Q \ast \} \]
5.2. SPECIFICATION

In the context of such a specification, $P$ is called the *pre-condition* and $Q$ is called the *post-condition*. Such a specification states that if $S$ is executed in a state that satisfies $P$, it will terminate in a state that satisfy $Q$. The pre-condition can be thought to specify allowed initial states to execute $S$. The post-condition specifies its possible final states. Trivial examples of valid specifications are these:

\[
\begin{align*}
\{\ast x = 0 \ast\} & \quad x := x+1 \quad \{\ast x = 1 \ast\} \\
\{\ast x \geq 0 \ast\} & \quad y := x \quad \{\ast y \geq 0 \ast\}
\end{align*}
\]

An example of a specification which is not valid:

\[
\{\ast \text{true} \ast\} \; \text{skip} \; \{\ast x = 1 \ast\}
\]

We have used the term ‘valid specification’. More precisely, a specification is *valid* if all possible executions (runs) of the specified program satisfy the specification. Otherwise it is *invalid*. Validity is usually shown by giving a mathematical proof. Invalidity is shown by giving a *counter example*, which in this case is a run that does not satisfy the specification. In terms of programming, such a counter example is called a *bug*. Note that *testing*, even if we do that with tools, can show invalidity, but it cannot prove validity.

We also say that a program *satisfies* a specification to mean that the specification is a valid specification. The *correctness* of a program is always expressed with respect to some specification. So, saying ”program $S$ is correct” does not make sense unless you also give a specification which $S$ has to satisfy. As a side note, any program $S$ satisfies this specification:

\[
\{\ast \text{false} \ast\} \; S \; \{\ast Q \ast\}
\]

So, any $S$ is trivially ‘correct’ in this sense (so, it is also not a useful specification, except for a theoretical purpose).

In addition to using *Form* predicates to specify initial and final states of a program, we can also use it to specify its possible states at any control point. Here is an example:

\[
\begin{align*}
\{\ast x > 0 \ast\} & \quad y := x \\
\{\ast y > 0 \ast\} & \quad y := y - 1 \quad \{\ast y \geq 0 \ast\}
\end{align*}
\]

With the predicate $y > 0$ between the two assignments we mean to say that when the program is in that control location, its state should satisfy the predicate. So, in the example above the specification is quite obviously valid.

A *Form* predicate used inside $\{\ast - \ast\}$ is also called an *assertion*: it asserts some constraint on the program’s state at that point. Pre- and post-conditions are assertions too.

A specification can be extended by inserting new assertions in it. For example, we have this specification (it does not say much about the final state, but it does say that $S_1; S_2$ terminates):

\[
\{\ast \text{true} \ast\} \; S_1; \; S_2 \; \{\ast \text{true} \ast\}
\]

We can extend it to:

\[
\{\ast \text{true} \ast\} \; S_1; \; \{\ast x = 0 \ast\} \; S_2 \; \{\ast \text{true} \ast\}
\]

which also demands that after $S_1$ we have $x = 0$. Inserting a new assertion is safe in the sense that it cannot make an invalid specification valid. However, it may make your specification ‘stronger’. That is, it demands more, and thus is also more difficult, perhaps even impossible, to satisfy.
Total and Partial Correctness

Remember that \( \{ * P * \} S \{ * Q * \} \) also means that \( S \) terminates when executed in a state satisfying \( P \). In some setups we may deliberately choose to assume termination rather than requiring the proof of it. In the literature, such an interpretation of Hoare triple is also called partial correctness interpretation. It may be acceptable when, for example, we are dealing with programs whose termination is obvious. The default notion of correctness used by uPL includes termination, and is called total correctness. As an example, the following specification is not always valid under total correctness:

\[
\{ * P * \} S \{ * \text{true} * \}
\]

because \( S \) may not terminate. It is however valid under partial correctness: \( S \) is assumed to terminate; so it will terminate in some state \( s \); and any state satisfies the \text{true} predicate.

Black Box and White Box

Some specification may not reveal the code of the program being specified. Such a specification is called black box specification. In this context, the term white box is used to refer to a specification that does reveal the program code. An example of a black box specification:

\[
\{ * 0 \leq \text{n} * \} \text{SUMN} (\text{n} : \text{int}) \{ * \text{return} = \text{n} \ast (\text{n} - 1) \text{div} 2 * \}
\]  

(5.1)

We only see the header of \text{SUMN}, but not its body. Despite this fact, the specification still reveals useful information: we know that when it is called with a non-negative \( \text{n} \) it will terminate, and we know exactly what the return value will be.

Obviously you cannot prove the validity of a black box specification —for that you need its full code. However, once a specification is proven, there are a number of good reasons to use it as a black box:

- By relying only on the specification rather than the code, you force yourself to reason more abstractly.
- If you only rely on the specification you can still change the program code, even after the program is deployed in a running application, as long as its specification is maintained.
- A black box informs your customers what the functionality of your program is, without having to expose the source code.

Initial Values

In a specification we sometimes need to refer to the initial values of some variables. Consider a program \( P \) with the following header:

\[
P \quad \text{(OUT x:int)} : \quad ()
\]

We want to specify that \( P \) should increase the value of \( x \). There is no way we can specify this in the post-condition without referring to the initial value of \( x \). One way to do it is to allow a notation like this:

\[
\{ * \text{true} * \} P (x) \{ * x > \text{old} x * \}
\]

where \text{old} \( x \) refers to the value of \( x \) when it is passed to \( P \). Although convenient, the use of the \text{old} operator complicates the logic [24]. To keep it simple, here we will use another trick. We add so-called auxiliary variables. These are fresh variables added to the program. We can subsequently extend the program, for example with assignments to these variables. However you can only extend the program in a way that does not change the the original control flow. With respect to
the original set of variables, the new and the old program should be equivalent. Auxiliary variables are used to record the values of actual program variables at certain control points.

Let us now introduce an auxiliary variable \( X \) to record the initial value of \( x \). Here is now how we specify the program \( P \):

\[
\{ \text{true} \} \quad X := x; \quad P(x) \quad \{ \ast X > x \ast \}
\]

As another example, here is how you can specify a program for swapping the value of two variables. \( X \) and \( Y \) below are auxiliary variables.

\[
\{ \text{true} \} \quad X, Y := x, y; \quad \text{SWAP(OUT } x, y : \text{int}) \quad \{ \ast (x = y) \wedge (y = X) \ast \}
\]

Specifying a Program

When specifying a statement, a (free) variable \( x \) in the post-condition refers to the value of the program variable \( x \) when the statement terminates. When specifying a full program we use a slightly different convention. A program is different from a statement (consult uPL grammar in Appendix C) because it has a name, it may have parameters, and it can be called by another program. Firstly, in a program's post-condition we can use the keyword \texttt{return} to refer to the value returned by the program. We have seen an example of this in the specification of \texttt{SUMN} in (5.1).

The second difference can be better explained with an example. Consider this contrived program:

\[
\text{copy}(x : \text{int}, \text{OUT } y : \text{int}) : \text{int} \\
\{ \ y := y + x \\
\quad ; x := y - x \\
\quad ; y := y - x \\
\quad ; \text{return } x \}
\]

The program copies the value of \( x \) to \( y \), and return the original value of \( y \). Notice that \( x \) is passed by value. So, even though \texttt{copy} changes its value, it does so only on its own copy of \( x \). The caller will see that \( x \) is unchanged. For this reason we use the convention that if \( x \) appears in the post-condition of \texttt{copy}, it refers to its value when it is passed to \texttt{copy}. The same convention applies to read-only pass-by-copy-restore parameter.

With the above convention, we can specify \texttt{copy} as follows:

\[
\{ \text{true} \} \quad Y := y; \quad \text{copy}(x, \text{OUT } y) \quad \{ \ast (y = x) \wedge (\text{return} = Y) \ast \}
\]

Notice that the \( x \) in the post-condition refers to the value of \( x \) when it is passed, whereas \( y \) in the post-condition refers to the final value of \( y \).

Typically, we also require that a program's specification is \textit{closed}. That is, it does not mention any variable other than the variables in the program's formal parameters or the auxiliary variables used to remember the parameters' initial values. This makes sure that such a specification can be reused in any context.

\textbf{ASSUMING} Section

Consider again the code of \texttt{BS} in Figure 5.1. The program is used to check if the value of \( i \) occurs in \( a[0 \ldots s] \). If it does, a non-negative value is returned, and otherwise a negative value is returned. We can easily express this formally:

\[
\text{return} < 0 = i \in a[0 \ldots s]
\]

However this does not cover all functionalities of the program. If \( i \) can be found, the return value also points to where \( i \) can be found in \( a \). This can be expressed as follows:

\[
\text{return} \geq 0 \Rightarrow (a[\text{return}] = i)
\]
You may notice that BS implements the binary search strategy. It only works if \( n \geq 0 \) and if \( a \) is sorted ascendingly. These facts seem to be 'forgotten' in BS's informal specification. We now want to make it explicit in our formal specification. The following formula can be used to express that \( a \) is sorted ascendingly:

\[
(\forall p, q: 0 \leq p < s \land 0 \leq q < s : p \leq q \Rightarrow a[p] \leq a[q])
\]

Figuring out the intuition behind a quantified formula like this can be quite hard. This can be improved with abbreviations. For example, we can introduce the abbreviation \texttt{sorted a s} to stand for the formula above. You can choose an abbreviation’s name that gives a hint about its meaning. This often improves the readability of a specification. If you need to document the abbreviations explicitly in your specification, you can do so in the \texttt{ASSUMING} section — see the example below.

\[
\{ * n > 0 \land \texttt{sorted a s} * \}
\]

BS(READ \( a: \text{int}[] \), \( i, s:\text{int} \) ) : \text{int} \{ \text{program body} \} \{ *( \text{return}<0 = \text{found a s i} ) \land ( \text{return}\geq0 \Rightarrow (a[\text{return}] = i) )* \} \]

\textbf{ASSUMING}

\texttt{sorted a s} = (\forall p, q: 0 \leq p < s \land 0 \leq q < s : p \leq q \Rightarrow a[p] \leq a[q])

\texttt{found a s i} = i \in a[0...s]

\hfill \Box

\textbf{New Vocabulary}

\textit{address; alias; assertion; ASSUMING section; auxiliary variable; non-aliasing assumption; partial correctness; pass by copy-restore; pass by reference; pass by value; pointer; reference; specification: black box, white box, valid, invalid, satisfy, correctness, counter example; total correctness}
5.3 Exercises

1. Consider the black box specifications below. What does each program do according to its specification? Express it in plain english.

   (a) \{ \textit{true} \}
   \[
   X := x; \text{inc}(d: \text{int}, \text{OUT} x: \text{int})\\
   \{ (x = X + d) \land (\text{return} = X) \}
   \]

   (b) \{ \textit{true} \}
   \[
   \text{zero}(\text{READ} a: \text{int}[], n, k: \text{int})\\
   \{ \text{return} = (\forall i: 0 \leq i < n: (a[i] = 0) \Rightarrow (i = k)) \}
   \]

   (c) \{ \textit{true} \}
   \[
   \text{find}(\text{READ} a: \text{int}[], n, x: \text{int})\\
   \{ x \in a[0 \ldots n] \Rightarrow (a[\text{return}] = x) \}
   \]

   (d) \{ \textit{true} \}
   \[
   \text{isINC}(\text{READ} a: \text{int}[], n: \text{int})\\
   \{ \text{return} = (\forall i, j: 0 \leq i, j < n: a[i] \geq a[j] \Rightarrow i \geq j) \}
   \]

   (e) \{ \textit{true} \}
   \[
   \text{maxTprefLen}(\text{READ} b: \text{bool}[], n: \text{int})\\
   \{ \text{return} = \text{MAX} \{ i \mid i \text{ from } [0..n], \text{allT} b[i] \}\}
   \]
   \text{ASSUMING} \quad \text{allT} b \text{ right} = (\forall j: 0 \leq j < \text{right}: b[j])

2. What do you think the following uPL program does? Capture it in a formal specification.

   \[
   \text{iP} \ (n: \text{int}) : \text{bool}\\
   \{ \text{var} \ d : \text{int} ;\\
   \quad \text{var} \ b : \text{bool} ;\\
   \quad b := \text{true} ; \quad d := 2 ;\\
   \quad \text{while} \ b \land d < n \ \text{do}\\
   \quad \{ \ b := n \text{ mod } d \neq 0 \\
   \quad \quad ; \ d := d + 1 \}\} ;\\
   \text{return} \ b
   \}

3. Write a specification for a program \text{getMAX}. When given an array \( a \) and a positive integer \( n \), the program should return the greatest element of \( a \) in the domain \( 0 \leq i < n \).

4. We decide to extend \text{getMAX} so that it also checks if the computed maximum is unique. That is, if \( m \) is the returned value, then the program also checks if \( m \) only occurs once in \( a \) within the the domain \( 0 \leq i < n \). Extend your specification to also reflect this additional functionality.

5. Give a specification for a program that reverses the order of the elements of an array in the domain \( 0 \leq i < n \).
5.4 Solution

1. (a) \texttt{inc} increases \( x \) by \( d \) and returns the original value of \( x \). Notice that \( x \) is passed by copy-restore and \( d \) by value.

   (b) If \( a \) contains a ‘0’, the program \texttt{zero} checks if within the specified domain \( a[k] \) is the only ‘0’ in \( a \). Note that if a ‘0’ is not found, the specification says that a \texttt{true} is returned.

   (c) If the array \( a \) contains \( x \) in the specified domain, \texttt{find} will return an index pointing to where \( x \) can be found. The specification does not specify what the return value is if \( x \) is not found.

   (d) \texttt{isINC} checks whether within the specified domain the array \( a \) is sorted in a strictly increasing order. Notice that a \texttt{false} is returned if \( a \) contains duplicate elements (within the domain). For example, if \( n = 2 \) and \( a[0] = a[1] \), the quantified formula in the post-condition would imply:

   \[
   (a[0] \geq a[1]) \Rightarrow 0 \geq 1
   \]

   which is false.

   (e) \texttt{maxTprefLen} returns the length of longest prefix of the array \( b \) whose elements are all \texttt{true}.

2. The program tests if \( n \) is a prime number (not efficiently, but that is not the issue now). Note that it requires \( n \) to be bigger than 2. The specification:

   \[
   \{ \ast \ n > 2 \ast \} \ \text{isP}(n) \ \{ \ast \ \text{return} = \text{isPrime} n \ast \}
   \]

   \textbf{ASSUMING}

   \text{isPrime} k = (\forall d: d > 0: (d \texttt{canDivide} k) \Rightarrow (d = 1) \lor (d = k))

   \text{d canDivide} k = (k \texttt{mod} d = 0)

3. We will assume \texttt{getMAX} to have the following header:

   \texttt{getMAX(READ a:int[], n:int) : int}

   The specification: \[
   \{ \ast n > 0 \ast \} \ \text{getMAX}(a,n) \ \{ \ast \ \text{return} = \text{MAX}(a[0...n]) \ast \}
   \]

4. We extend the header of \texttt{getMAX} to:

   \texttt{getMAX(READ a:int[], n:int, OUT b:bool) : int}

   Its new specification:

   \[
   \{ \ast n > 0 \ast \}
   \text{getMAX}(a,n,b)
   \{ \ast (\text{return} = \text{MAX}(a[0...n])) \land (b = \text{unique} a \ n \text{return}) \ast \}
   \]

   \textbf{ASSUMING}

   \text{unique a n x = COUNT}[a[k] \mid k \text{ from } [0...n], a[k] = x] \leq 1

5. Assume the following header:

   \texttt{REV(n:int, a:int[])} : ()
The specification:

\{ \textit{true} \}

\texttt{A:=a ; REV(n,a)}

\{ \textit{(\forall k: 0\leq k < n: a[k] = A[n-1-k])} \}
In Chapter 5 you have learned how to precisely specify what a program is required to do. Once specified, we still have to show that the program is correct with respect to its specification. Informal reasoning is useful to motivate why the program is correct (or otherwise), but it is not something that we really can rely on, since it is ambiguous and lacks in precision.

In formal reasoning you will have to deal with formulas. Typical operations that you want to do are rearranging a formula, eliminating some parts of it, or adding new parts to it. In principle, you can do these operations without being aware of what your formulas mean. However, trying out different operations without having an idea of what you are doing or where you want to go is like walking in a thick forest without a compass — it is unlikely that you can succeed in finding a proof in this way. The compass that you need is abstraction. Literally it means a visionary idea. Abstraction is the mental picture that you have in your mind of what the formulas in your proof mean. This mental picture is much simpler than the formulas they represent, and thus is easier to comprehend. It allows you to make a mental plan or strategy of how to solve a problem. So, you will need your skill to think abstractly. Once you have your strategy, you can work it out in the form of a formal proof. A formal proof is very precise. It is much more reliable. However, being very precise also means that you have to deal with a lot more details than in informal reasoning. You will have to learn the skill to handle these details.

6.1 Abstract Model

Before we proceed with discussing our programming logic, it is useful to first look at models of programs. A model is a simplified representation of a real thing. Models are thus much easier to understand. For us they help in explaining some of the more technical aspects of the programming logic.

---

1 In computer science, the term abstraction is also used technically to mean a simplified, and usually formal, representation of something. In a sense, a visionary idea is a also simplified view of the implementation of the idea.
Because a model and the program it represents are two different things, you may want to denote them differently. For example, if \( S \) is a program, then perhaps we can write \(|S|\) to denote the model of \( S \). However, this clutters our formulas and hinders our discussion. So we decide not to do so. That is, we write \( S \) to refer to both the program \( S \) and its model. Hopefully from the context it would be obvious which one is meant in a particular use.

### 6.1.1 State

A program have states. At any moment it is in a particular state, and each step of the program causes it to go from one state to another. The actual state of a program is very complex, comprising of the content of the memories of your computer, the content of various registers of your computer’s processors, etc. However, abstractly the *state* of a program comprises of the value of all its variables at a particular moment. For example if we recall the program `Euclid`:

```plaintext
Euclid(x,y : int) : int {
    while x<>y do if x>y then x:=x-y else y:=y-x ;
    return x
}
```

the state of this program is the value \( x \) and \( y \) at a given moment. Let us write e.g. \( \{x=0, y=1\} \) to denote a state where the value of \( x \) is 0, and that of \( y \) is 1.

So, we can say for example that if \( \{x=4, y=8\} \) is the state of `Euclid` as we enter its body, then \( \{x=2, y=2\} \) would be its state just before it returns.

Each statement in a program can be thought to move the program from one state to another. The state where a program or statement is executed on is called its *initial state* or *starting state*. The state at which it stops is called *terminal state* or *final state*.

### 6.1.2 Modelling Program as Relation

In mathematics, a *relation* on a domain \( A \) is a function of type \( A \rightarrow \text{Pow}(A) \), where \( \text{Pow}(A) \) denotes the set of all subsets of \( A \) (it is the so-called *power set* of \( A \)). So, if \( R \) is a relation on \( A \), \( y \in R x \) means that \( R \) relates \( x \) to \( y \). A relation can also be interpreted as a function that returns multiple possible results. So if \( R x = U \) then firstly \( U \) is a set. It is also a subset of \( A \). Furthermore, \( U \) can be seen as specifying all possible results of applying the ‘function’ \( R \) to \( x \).

We can abstractly model a program as a relation on states. It is a simple and easy to understand model. For simplicity, we will for now assume that all our programs terminate.

Let \( \sum \) be the set of all possible states that a program \( S \) may have. The program \( S \) can be modelled as a relation on \( \sum \). So, it is a function of type:

\[
\sum \rightarrow \text{Pow}(\sum)
\]

So, if \( s \) is a state in \( \sum \), then \( S s = V \) means that \( V \) is the set of all possible final states of \( S \) when it is executed on the state \( s \). Since we assumed \( S \) to terminate, \( V \) will contain at least one element. For example, in terms of the above kind of models the `skip` statement is defined like this:

\[
\text{skip } s = \{s\}
\]

It says that if we execute `skip` on a state \( s \), then there is only one possible final state, which is \( s \) itself.

Other statements, e.g. an assignment is also a relation on \( \sum \). So, we can apply it to a state to get its possible final states, e.g.:

\[
(x := x+1) \{x=0, y=1\} = \{(x=1, y=1)\}
\]
6.1. ABSTRACT MODEL

Most (non-interactive) sequential programs are deterministic. If we repeatedly started such a program in the same state \( s \), the program will always behave in the same way; thus ending in the same state. Programs that do not have this property are called non-deterministic. In terms of the above model, \( S \) is deterministic if \( S \) contains exactly one state. Otherwise, if for some \( s \) there are more that one (terminal) state in \( S \), then \( S \) is non-deterministic.

Interactive or concurrent programs programs are typically non-deterministic. Non-determinism may simplify or complicate analysis. In the case of concurrent programs, it is usually the latter.

We can now define the meaning of some standard statements in terms of models as explained above:

**Definition 6.1.1 : Models of Skip, Conditional, and Sequence**

1. \( \text{skip} \ s = \{ s \} \)

2. \( (\text{if} \ g \ \text{then} \ S_1 \ \text{else} \ S_2) \ s = g s \rightarrow S_1 \ s \upharpoonright S_2 \ s \)

3. \( (S_1 ; S_2) \ s = \cup \{ S_2 \ t \mid t \in S_1 \ s \} \)

\[ \square \]

where \( \cup \{ X \} \) means the union of all sets in \( X \). So, \( \cup \{ U, V \} = U \cup V \).

Notice that the above definition excludes exceptions. You can see this in e.g. the definition of \( S_1;S_2 \), which simply passes on the final state \( t \) after \( S_1 \) to \( S_2 \), whereas if we are to allow exception then \( S_2 \) may be skipped if \( S_1 \) throws an exception. We will not have exception in \( uPL \). In practice you will of course have to deal with that, and would need a more powerful logic that that of \( uPL \) to deal with it.

Because assignments affects states, defining their models requires us to first come up with a model of states. If we for example model a state as a function from variable names to their value, and an expression as a function from state to value, then we can model assignments as follows:

**Definition 6.1.2 : Model of Assignment**

\[ (x := e) \ s = (\lambda v. v = x \rightarrow e \ s \upharpoonright s x) \]

\[ \square \]

6.1.3 Predicate as a Set of States

When we write a predicate like \( x > y \) as e.g. a pre-condition, we intend to interpret it over states. That is, when given a particular state the value of the predicate would be either true or false. Because we will use this concept frequently later, let us also introduce a notation for that.

For a predicate \( P \) and a state \( s \) we will write \( P \ s \) to mean the value of the predicate \( P \) when interpreted on the state \( s \). For example we can write:

\[ (x > y) \ \{ x = 1, \ y = 0 \} \]

By this we mean the value of the predicate \( x > y \) interpreted on the state \( \{ x = 1, \ y = 0 \} \). This value is in thus true.

The term 'a state \( s \) satisfies a predicate \( P \)' is thus the same as '\( P \ s \) is true'.

A predicate, such as \( x > y \), can be thought to specify a set of states. So if we talk about a program \( S \) whose set of possible states is \( \sum \), a predicate \( P \) specifies a set containing all state \( s \in \sum \) such that \( s \) satisfies \( P \). So, this set:

\[ \{ s \mid s \in \sum, \ P \ s \} \]

This is called the set interpretation of the predicate \( P \). Such an interpretation explains why we use predicates to specify programs. E.g. in \( \{ \ast \ P \ast \} \ S \{ \ast \ Q \ast \} \) the predicate \( P \) is interpreted as specifying the set of allowed initial states, and \( Q \) the set of all possible final states.
The set interpretation of a predicate is an equivalent interpretation. It is sometimes useful to make certain things easier to understand. In fact, in programming logic predicates are often treated as if they are just sets. For example, let $P$ be a predicate, people often say "the set $P$" to actually mean the set of states specified by the predicate $P$. People also say "a state $s$ is in $P$" to actually mean that $s$ is in the set of states specified by the predicate $P$ (or equivalently, $s$ is a state such that $P$ is true).

Recall that we have introduced the notation $P s$ to mean that the value of the predicate $P$ is true when interpreted on the state $s$. When $P$ is interpreted as a set, then $P s = s \in P$.

**Theorem 6.1.3**

Let $P$ be a predicate and $s$ a state:

$$P s = s \in P$$

Notice that implication corresponds to set inclusion. That is, $\vdash P \Rightarrow Q$ is equivalent with the fact that $P$, interpreted as a set, is a subset of (included in) $Q$. Hence, a weaker predicate is larger, and a stronger one is smaller. Analogously, the operator $\lor$ and $\land$ correspond to respectively set union and set intersection.

### 6.1.4 The Formal Meaning of Hoare Triples

Now we can give a meaning to specifications in terms of our models. However, since our models assume termination, with them we can only express the meaning of specifications under the partial correctness interpretation. Recall that under partial correctness, this:

$$\{* P *\} S \{* Q *\}$$

means that if $S$ is executed on a state in $P$ then, if it terminates, its final state will be in $Q$. Recall that we model $S$ as a relation on states. In terms of our models, the above specification means this:

**Definition 6.1.4**: Hoare Triple in Partial Correctness

$$\{* P *\} S \{* Q *\} = (\forall s : s \in P : S s \subseteq Q)$$

The programming logic that we will introduce later will be in total correctness. To give you a deeper understanding of some of the logic’s inference rules sometimes we also show their proofs. However, to make these proofs easy to follow they will be given in terms of partial correctness using the models discussed in this section. So, keep in mind that formally those proofs are not complete (we have yet to prove termination).

From the above definition, we can for example already infer the following ‘principles’:

1. Suppose $\{* P *\} S \{* Q *\}$. This implies that for any state $s$ in $P$ we have $S s \subseteq Q$. If $Q'$ is weaker than $Q$, then $Q \subseteq Q'$. It follows that $S s \subseteq Q'$. Hence, $S$ also satisfies $Q'$ as a post-condition. This principle is also called post-condition weakening.

2. Suppose $\{* P' *\} S \{* Q *\}$. This implies that any state $s' \in P'$ we have $S s' \subseteq Q$. If $P$ is stronger than $P'$, then $P \subseteq P'$. So, any state $s \in P$ will also be in $P'$, and thus we also have $S s \subseteq Q$. Therefore $S$ also satisfies $P$ as a pre-condition. This principle is also called pre-condition strengthening.

3. A bit more sophisticated, suppose $S$ satisfies these two specifications:

   (a) $\{* P_1 *\} S \{* Q_1 *\}$
6.2. **WEAKEST PRE-CONDITION**

\[ \vdash Q \Rightarrow Q' \]
\[ \{* P *\} S \{* Q *\} \]
\[ \{* P *\} S \{* Q' *\} \]

**Rule 6.1.5 : POST-condition Weakening**

\[ \vdash P \Rightarrow P' \]
\[ \{* P' *\} S \{* Q *\} \]
\[ \{* P *\} S \{* Q *\} \]

**Rule 6.1.6 : PRE-condition Strengthening**

\[ \vdash P \Rightarrow P' \]
\[ \{* P' *\} S \{* Q *\} \]
\[ \{* P *\} S \{* Q *\} \]

**Rule 6.1.7 : HOARE TRIPLE DISJUNCTION**

\[ \{* P_1 *\} S \{* Q_1 *\} \]
\[ \{* P_2 *\} S \{* Q_2 *\} \]
\[ \{* P_1 \lor P_2 *\} S \{* Q_1 \lor Q_2 *\} \]

**Rule 6.1.8 : HOARE TRIPLE CONJUNCTION**

\[ \{* P_1 *\} S \{* Q_1 *\} \]
\[ \{* P_2 *\} S \{* Q_2 *\} \]
\[ \{* P_1 \land P_2 *\} S \{* Q_1 \land Q_2 *\} \]

---

Figure 6.1: Some basic rules for Hoare triples.

(b) \( \{* P_2 *\} S \{* Q_2 *\} \).

You can also prove the following:

(a) From any state \( s \) in \( P_1 \cup P_2 \), the set of possible final states, \( S s \), is a subset of \( Q_1 \cup Q_2 \).
   In other words, \( S \) satisfies \( \{* P_1 \lor P_2 *\} S \{* Q_1 \lor Q_2 *\} \).

(b) From any state \( s \) in \( P_1 \cap P_2 \), the set of possible final states, \( S s \), is a subset of \( Q_1 \cap Q_2 \).
   In other words, \( S \) satisfies \( \{* P_1 \land P_2 *\} S \{* Q_1 \land Q_2 *\} \).

The principles are called *Hoare triples disjunction* respectively conjunction.

Though these are simple principles, we can already use them to formally reason about programs and specifications. However, as you can see in the above explanation, we have to keep translating predicates to sets. While this is useful when trying to explain and justify a 'principle' or an inference rule, we do not want to keep doing it when reasoning about programs. For the latter, we will keep our proofs at the predicate level; it is more abstract that way.

The above 'principles' are formally expressed by the inference rules shown in Figure 6.1.

### 6.1.5 Expressing Non-termination

Just for completeness, we will briefly discuss here how to extend the models above so that we can express total correctness. This discussion is theoretical in nature. We prefer not to, and we are not going to, write the correctness proofs of our programs in terms of those models.

The models have to be as such that we can distinguish between terminating and non-terminating programs. We can do this by extending \( \sum \) with a special element, let’s denote it by \( \bot \), to denote non-termination. Let us denote this extended \( \sum \) with \( \sum_\bot \). If \( S \) is a statement, it is now modelled by a function from \( \sum_\bot \) to \( \text{Pow}(\sum_\bot) \). For any \( t \), \( S t \) is never empty (either it terminates in some state, or it yields \( \bot \)). Furthermore, for any \( S, S \bot = \{ \bot \} \). So, the models of some statements, in particular \( \text{skip} \) and assignment given before will have to be fixed:

\[ \text{skip} \ s \ = \ s=\bot \to \{ \bot \} \ | \ \{s\} \]
\[ (x:=e) \ s \ = \ s=\bot \to \{ \bot \} \ | \ (\lambda v. v = x \to e \ s \ | \ s \ x) \]

The definitions of sequential composition and if-then-else do not have to be changed. For the latter we do assume that the evaluation of the guard always terminate.

### 6.2 Weakest Pre-condition

We will show you two complementary verification approaches. This section discusses the first one, which is called \( \text{wp} \) reduction. Given a specification \( \{* P *\} S \{* Q *\} \), the idea is to somehow
calculate the *weakest pre-condition* $P'$ that guarantees that $S$ will always terminate in $Q$. After obtaining $P'$, it is then sufficient to prove that $P$ implies $P'$. Unfortunately, calculating the weakest pre-condition is not always possible, e.g. when $S$ contains a loop or recursion. This is where the second verification approach comes in; we will discuss this later.

Take a look at the second rule in Figure 6.1; it is called the *Pre-condition Strengthening Rule*. It says that in order to prove $\{\ast P \ast\} S \{\ast Q \ast\}$ we can instead try to prove another specification, say $\{\ast P' \ast\} S \{\ast Q \ast\}$. This can be useful if the second specification is easier to prove. The rule allows this provided $P$ is stronger than $P'$. The disadvantage of this approach is that if $\{\ast P' \ast\} S \{\ast Q \ast\}$ turns out to be invalid, we do not know whether it is because the chosen $P'$ is too weak, or because the original specification is simply invalid (and hence there is no $P'$ such that $\{\ast P' \ast\} S \{\ast Q \ast\}$).

A more clever strategy is to construct the weakest $P'$. We will denote this by $wp \ S \ Q$ which is pronounced: "the weakest pre-condition of $S$ with respect to (the post-condition) $Q". The meaning of this is somewhat different for partial and total correctness. Under the *partial correctness* interpretation it can be seen as the set of all initial states of $S$, such that the corresponding final states are all in $Q$. We can express this in terms of our set models (Section 6.1):

**Definition 6.2.1 : wp under Partial Correctness**

\[
wp \ S \ Q = \{s \mid S \ s \subseteq Q\}
\]

An equivalent way to express this, which we will use several times later, is the following:

**Corollary 6.2.2 :**

\[
s \in wp \ S \ Q = S \ s \subseteq Q
\]

Under the *total correctness* interpretation $wp \ S \ Q$ specifies the set of all initial states from which $S$ will terminate in $Q$. Rather than giving a direct formal definition of this we will instead provide you with the following characterization for $wp$. This characterization is good for both partial and total correctness:

**Definition 6.2.3 : Characterization of wp**

\[
\vdash \ P \Rightarrow wp \ S \ Q = \{\ast P \ast\} S \{\ast Q \ast\}
\]

Firstly, the definition implies that $wp \ S \ Q$ itself, whatever it is, is at least also a 'good' pre-condition for realizing $Q$. This is formally expressed by the following theorem (we leave out the proof for you; it’s really trivial):

**Corollary 6.2.4 :**

\[
\{\ast wp \ S \ Q \ast\} \ S \ \{\ast Q \ast\}
\]

Secondly, the above definition of $wp$ implies that if $P' = wp \ S \ Q$, then it will be *weaker* than any pre-condition $P$ from which $S$ can realize the post-condition $Q$. Since this holds for any $P$, this indirectly implies that $P'$ must be the weakest one.

Notice also that we have an equality relation above, which is bi-directional. This implies, that if $\vdash \ P \Rightarrow wp \ S \ Q$ turns out to be *not* valid (there exists a counter example against its validity), the above definition says that the specification $\{\ast P \ast\} S \{\ast Q \ast\}$ is thus invalid as well.
6.2. WEAKEST PRE-CONDITION

The above definition is also very powerful: it allows you to reduce the problem of proving that a program satisfies a specification into a problem of proving an ordinary predicate logic formula. Of course this is only useful if we can actually calculate \( \text{wp} \ S \ Q \). In some cases it is possible, even trivial, e.g:

\[
\text{wp \ skip} \ Q = Q
\]  

(6.1)

Unfortunately, there are also cases where no general algorithm is known for calculating the weakest pre-condition—we will return to this issue later.

The following theorem shows how to calculate the \( \text{wp} \) of if-then-else.

**Theorem 6.2.5 : wp of if-then-else**

\[
\text{wp} \ (\text{if } g \text{ then } S_1 \text{ else } S_2) \ Q = g \rightarrow \text{wp} \ S_1 \ Q \ | \ \text{wp} \ S_2 \ Q
\]

\( \square \)

By the way, the \( \text{wp} \) is also equivalent with the following:

\[(g \Rightarrow \text{wp} \ S_1 \ Q) \land (\neg g \Rightarrow \text{wp} \ S_2 \ Q)\]

or this one:

\[(g \land \text{wp} \ S_1 \ Q) \lor (\neg g \land \text{wp} \ S_2 \ Q)\]

**Proof**

We will only prove this for partial correctness. If we interpret the predicates at the left and right sides of '=' above as sets, then it is sufficient to show that for any state \( s \):

\[s \in \text{wp} \ (\text{if } g \text{ then } S_1 \text{ else } S_2) \ Q\]

if and only if:

\[g \ s \rightarrow s \in \text{wp} \ S_1 \ Q \ | \ s \in \text{wp} \ S_2 \ Q\]

This is quite easy to prove:

\[s \in \text{wp} \ (\text{if } g \text{ then } S_1 \text{ else } S_2) \ Q\]

\[= \{ \text{Corollary 6.2.2} \}\]

\[(\text{if } g \text{ then } S_1 \text{ else } S_2) \ s \subseteq Q\]

\[= \{ \text{Def. 6.1.1 of if-statement} \}\]

\[(g \ s \rightarrow S_1 \ s \ | \ S_2 \ s) \subseteq Q\]

\[= \{ \text{using COND Conversion} \}\]

\[g \ s \rightarrow S_1 \ s \subseteq Q \ | \ S_2 \ s \subseteq Q\]

\[= \{ \text{Corollary 6.2.2} \}\]

\[g \ s \rightarrow s \in \text{wp} \ S_1 \ Q \ | \ s \in \text{wp} \ S_2 \ Q\]

\( \square \)

From the theorem we can prove this inference rule. We leave out the proof for you.
Rule 6.2.6 : IF-THEN-ELSE

\[
\begin{align*}
\{ * P \land g \} & S_1 \{ * Q \} \\
\{ * P \land \neg g \} & S_2 \{ * Q \}
\end{align*}
\]

\[
\{ * P \} \quad \text{if } g \text{ then } S_1 \text{ else } S_2 \quad \{ * Q \}
\]

\[\Box\]

And here is how to calculate the \(wp\) of a sequence of statements:

**Theorem 6.2.7 : \(wp\) OF STATEMENTS SEQUENCE**

\[
wp (S_1;S_2) Q = wp S_1 (wp S_2 Q)
\]

\[\Box\]

**PROOF**

We will only prove this for partial correctness:

\[
s \in (wp (S_1;S_2) Q)
\]

\[
= \{ \text{Corollary 6.2.2} \}
\]

\[
(S_1;S_2) s \subseteq Q
\]

\[
= \{ \text{Def. 6.1.1 of sequence} \}
\]

\[
\cup \{ S_2 t \mid t \in S_1 \} \subseteq Q
\]

\[
= \{ \text{set theory} \}
\]

\[
(\forall t : t \in S_1 s : S_2 t \subseteq Q)
\]

\[
= \{ \text{Corollary 6.2.2} \}
\]

\[
(\forall t : t \in S_1 s : t \in wp S_2 Q)
\]

\[
= \{ \text{set theory} \}
\]

\[
S_1 s \subseteq wp S_2 Q
\]

\[
= \{ \text{Corollary 6.2.2} \}
\]

\[
s \in wp S_1 (wp S_2 Q)
\]

\[\Box\]

We also have the analogous of the if-the-else Rule for sequence of statements, but to prove that we first need the following result. Recall Rule 6.1.5: it states that you can weaken a post-condition. The rule below rephrases it in terms of \(wp\):

**Rule 6.2.8 : \(wp\) POST-CONDITION WEAKENING**

\[
\models Q \Rightarrow Q' \quad \models wp S Q \Rightarrow wp S Q'
\]

\[\Box\]
6.2. WEAKEST PRE-CONDITION

PROOF
Interpreting predicates as sets, it is sufficient to prove that \( s \in \text{wp} \ S \ Q \) implies \( s \in \text{wp} \ S \ Q' \). By Corollary 6.2.2, \( s \in \text{wp} \ S \ Q \) is equal to:

\[ S \ s \subseteq Q \]

Now in terms of set, \( Q \Rightarrow Q' \) means \( Q \subseteq Q' \). So with the above this implies:

\[ S \ s \subseteq Q' \]

Applying Corollary 6.2.2 again, then we have \( s \in \text{wp} \ S \ Q' \)

\( \square \)

Now we can have the following inference rule about sequence of statements:

**Rule 6.2.9 : SEQ**

\[
\begin{array}{l}
\{\ast \ P \ast\} \ S_1 \ \{\ast \ P' \ast\} \\
\{\ast \ P' \ast\} \ S_2 \ \{\ast \ Q \ast\}
\end{array}
\]

\[
\{\ast \ P \ast\} \ (S_1; \ S_2) \ \{\ast \ Q \ast\}
\]

\( \square \)

PROOF
The second assumption implies \( P' \Rightarrow \text{wp} \ S_2 \ Q \). Now we use Rule 6.2.8 (instantiating the \( S \) in the rule with \( S_1 \), and \( Q \) with \( P' \)), and conclude:

\[ \text{wp} \ S_1 \ P' \Rightarrow \text{wp} \ S_1 \ (\text{wp} \ S_2 \ Q) \]

The first assumption implies \( P \Rightarrow \text{wp} \ S_1 \ P' \). So, together with the above we have:

\[ P \Rightarrow \text{wp} \ S_1 \ (\text{wp} \ S_2 \ Q) \]

Now applying Theorem 6.2.7 on the right hand side of \( \Rightarrow \) above we obtain:

\[ P \Rightarrow \text{wp} \ (S_1; \ S_2) \ Q \]

which is equivalent to:

\[ \{\ast \ P \ast\} \ S_1 ; \ S_2 \ \{\ast \ Q \ast\} \]

\( \square \)

**Assignment**

The weakest pre-condition of an assignment requires some explanation. Consider the following simple example.

\[ \{\ast \ ? \ast\} \ x := x+10 \quad \{\ast \ x > 10 \ast\} \]

For example, \( x = 1 \) is a sufficient pre-condition for the assignment above to realize \( x > 10 \). But it is not the weakest one, because \( x = 2 \) will also do.

Since the value of \( x \) after the assignment is equal to the value of \( x + 10 \) before the assignment, the condition \( x > 10 \) after the assignment is equivalent with the condition \( x + 10 > 10 \) before the assignment. In other words, the requirement \( x > 10 \) after the assignment is equivalent to the requirement \( x + 10 > 10 \) as a pre-condition. The fact that they are equivalent implies that the pre-condition we just came up with is also the weakest one!
By the same argument, if we now have an arbitrary expression $E$ at the right hand side of the assignment:

$$\{? \} \quad x := E \quad \{? x > 10 \}$$

the weakest pre-condition for this assignment to realize the post-condition $x > 10$ would be $E > 10$.

Note that the argument above suggests we should replace $x$ in the post-condition by $E$. So, we obtain $E > 10$ by the substitution $(x > 10)[E/x]$.

In general, consider a post-condition $Q$ of the assignment $x := E$. Any $x$ (occurring free) in $Q$ refers to the final value of $x$. That is, the value of $x$ in the state after the assignment. However since $x$ gets its value from $E$, its final value is equal to the value of $E$, but latter is interpreted on the state before the assignment. So, it follows that the value of $Q$ interpreted on the state after the assignment is equal to the value of $Q[E/x]$ interpreted on the state before the assignment. Because of the equilance this implies that $Q[E/x]$ would be a good pre-condition for $Q$, and furthermore it must also be the weakest one. Formally:

**Theorem 6.2.10**: wp of Assignment

$$\text{wp} \ (v := E) \ Q = Q[E/v]$$

In the theorem above, the target of the assignment ($v$) should be a program variable. Assignments to an array element or a record field requires a slightly different treatment —this will be discussed later in Section 6.12.

There is an important assumption which is implicit in the above theorem, namely that changing the value of a variable named $x$ only affects $x$. This is the case in uPL because uPL does not allow aliases —see also the discussion about alias in Section 5.1. This assumption is made to simplify reasoning. For languages like Java where aliasing is pervasive it is not realistic; a more powerful logic will be needed.

As an example, consider again the following (a bit contrived) example. We want to prove:

$$\{? x > 0 \} \quad y := -1-x \ ; \ z := x*y \quad \{? z < 0 \}$$

The validity of this specification is not intuitively clear. So a formal proof would be helpful here. We can do it by first calculating the wp of the statement above with respect to its post-condition:

$$\text{wp} \ (y := -1-x; \ z := x*y) \ (z < 0)$$

$$= \{ \text{Theorem 6.2.7} \}$$

$$\text{wp} \ (y := -1-x) \ (\text{wp} \ (z := x*y) \ (z < 0))$$

$$= \{ \text{Theorem 6.2.10} \}$$

$$\text{wp} \ (y := -1-x) \ ((z < 0)[x*y/z])$$

$$= \{ \text{applying the substitution} \}$$

$$\text{wp} \ (y := -1-x) \ (x*y < 0)$$

$$= \{ \text{Theorem 6.2.10} \}$$

$$(x*y < 0)[{-1-x/y}]$$

$$= \{ \text{applying the substitution} \}$$

$$x * (-1-x) < 0$$
6.2. WEAKEST PRE-CONDITION

{∗ x>0, justified by Definition 6.2.3 and PROOF top ∗}
y := −1 − x ;
z := x*y ;
{∗ z<0 ∗}

PROOF top
[A:] x > 0
[G:] wp (y := −1 − x; z := x*y) (z < 0)
BEGIN

1. { see the wp calculation in page 68 }
wp (y := −1 − x; z := x*y) (z < 0) = x * (−1 − x) < 0
2. { follows from A }
x^2 > 0
3. { follows from A }
−x < 0
4. { follows from 2 and 3 }
−x < x^2
5. { follows from 4 }
−x − x^2 < 0
6. { follows from 5 }
x * (−1 − x) < 0
7. { rewrite 6 with 1 }
wp (y := −1 − x; z := x*y) (z < 0)

END

Figure 6.2: An example of a formal proof of a specification.

So, we obtain x * (−1 − x) < 0 as the wp. This is equal to −x < x^2, which is indeed implied by the pre-condition x > 0. This proves the validity of the specification given above. The last step in the proof (that the pre-condition implies the calculated wp) is however given informally. In this example it is quite harmless, since it is a quite simple argumentation. But as an example of a full formal proof see Figure 6.2. The pre-condition is augmented with a comment, saying that the specification is proven through the Definition 6.2.3 of wp; recall that the latter implies that it is sufficient to prove that the given pre-condition implies the wp. The proof of the latter is given in proof top.

Looking at top, it sets the wp of the corresponding statement as its goal, and the pre-condition as an assumption (A). You see that the first step is to prove that:

wp (y := −1 − x; z := x*y) (z < 0) = x * (−1 − x) < 0

The right hand side of this equation is just the result of calculating the wp. In other words, the equation above tells you what the resulting wp is, which is what you must show to be implied by the assumption A. The proof then proceeds by trying to derive the right hand side of the equation above. Step 6 finally succeeds in deriving it, and hence, based on the equation above the goal is proven.

In the above example we have shown the sf wp calculation. However, such a calculation is usually quite straightforward (you only have to apply the corresponding wp theorems, such as
PROOF
[A:] \( x > 0 \)
[D:] \( P = \text{wp} \ (y := -1-x; \ z := x \times y) \ (z < 0) \)
[G:] \( P \)

BEGIN

1. { calculating \( \text{wp} \) }
   \( P = x \times (-1-x) < 0 \)
2. { follows from \( \text{A} \) }
   \( x^2 > 0 \)
3. { follows from \( \text{A} \) }
   \( -x < 0 \)
4. { follows from 2 and 3 }
   \( -x < x^2 \)
5. { follows from 4 }
   \( x \times (-1-x) < 0 \)
6. { rewrite 5 with 1 }
   \( P \)

END

Figure 6.3: Using abbreviation to improve the readability of a proof.

Theorems 6.2.10, 6.2.7, and 6.2.5). For this reason in most proofs in this book we usually omit the \( \text{wp} \) calculation.

You can use abbreviations to improve the readability of a proof. You see that in the proof in Figure 6.2 the expression \( \text{wp} \ (y := -1-x; \ z := x \times y) \ (z < 0) \) is repeated three times. Figure 6.3 shows a bit more readable version of \( \text{top} \). It introduces a new name \( P \) to abbreviate \( \text{wp} \ (y := -1-x; \ z := x \times y) \ (z < 0) \), and hence avoiding its annoying duplications.

The return statement

Recall that when specifying a program, we can use the keyword \textit{return} in the post-condition to refer to the value returned by the program. This is also the only place where \textit{return} can appear in the specification. For the purpose of reasoning we can treat the \textit{return} in the post-condition as if it is a variable and the statement \textit{return} \( e \) as if it is an assignment \textit{return} := \( e \). Then we can handle it as explained above.

6.3 Loop Reduction Strategy

The proof strategy in the previous section is to convert the problem of proving a specification \( \{ * P \} \ S \ { * Q \} \) to the problem of showing that \( P \) implies \( \text{wp} \ S \ Q \). This works as long as we can calculate the \( \text{wp} \). Unfortunately, there is no general algorithm known for calculating the \( \text{wp} \) of a loop and of a recursive program. However, if you recall the Pre-condition Strengthening Rule (Rule 6.1.6), to prove \( \{ * P \} \ S \ { * Q \} \) it is sufficient to find a \( P' \) which is \textit{weaker} than \( P \) such that \( \{ * P' \} \ S \ { * Q \} \). So, it is not necessary to find the weakest \( P' \)! However you have to keep in mind that if we later discover that \( \{ * P' \} \ S \ { * Q \} \) is invalid, we do not know whether it is because the original specification \( \{ * P \} \ S \ { * Q \} \) is invalid, or because the choice of \( P' \) is too weak. A closer analysis will be needed to determine which of these two is the problem and how to deal with it.
Consider now a specification:

\[ \{ * P * \} \text{ while } g \text{ do } S \{ * Q * \} \]

Using the idea above, we can prove it by finding a weaker pre-condition \( I \) such that:

\[ \{ * I * \} \text{ while } g \text{ do } S \{ * Q * \} \]

(6.2)

The next question is: what kind of \( I \) would satisfy the above specification? A trivial candidate \textit{false} will do. However, we will then have a problem in proving that \( P \) implies it.

To find a more reasonable \( I \), consider an \( I \) such that it holds at the end of every iteration. This is nice, because then \( I \) will also hold after the last iteration. So, if \( I \) also implies \( Q \) then it follows that from the pre-condition \( I \) the loop, if it terminates, will terminate in \( Q \). We can express this with a rule as follows:

\[
\begin{array}{c}
\models I \Rightarrow Q \\
I \text{ holds at the end of every iteration}
\end{array}
\implies \{ * I * \} \\text{ while } g \text{ do } S \{ * Q * \}
\]

The rule is still a bit informal; we will make it formal in a minute. But the important thing here is that this is a sufficient rule. It tells us how we can prove that a loop, if it terminates, will terminate in \( Q \). The rule says thus that we have to find an \( I \) satisfying the above two premises.

Another important thing to note here is that the argument that leads to the above rule has assumed termination. The premise "\( I \) holds at the end of every iteration" does not by itself implies that the loop above will actually terminate. So, the rule is only good for the partial correctness interpretation.

The first premise, \( I \Rightarrow Q \) can actually be weakened (thus easier to prove). When the loop terminate we know something else, namely that \( g \) must be false. So, it is actually sufficient to prove \( I \land \neg g \Rightarrow Q \).

The second premise, namely that \( I \) holds at the end of every iteration, can be formally expressed by:

\[ \{ * \text{true} * \} \ S \{ * I * \} \]

However this is actually a stronger requirement. Requiring \( S \) to establish \( I \) from \textit{true} as the pre-condition means that \( S \) has to establish \( I \) from \textit{any} initial state. But in reality, \( S \) will not start from an arbitrary state. At the \( i \)-th iteration, it will start from whatever state, which is the result of the previous iteration. Now, if \( I \) is to holds at the end of every iteration, we actually also know that it therefore also hold and the end of the previous iteration. And therefore we can assume it as the pre-condition of \( S \). In other words, it is sufficient to require this instead:

\[ \{ * I * \} \ S \{ * I * \} \]

Furthermore, since an iteration can only execute if \( g \) is true, we can further weaken the above to this requirement:

\[ \{ * I \land g * \} \ S \{ * I * \} \]

which may be easier to prove because we have a stronger pre-condition.

With all the refinements above we now formally express the inference rule to prove the correctness of a loop (under the partial correctness interpretation):

\textbf{Rule 6.3.1: Loop Reduction in Partial Correctness}

\[
\begin{array}{c}
\models I \land \neg g \Rightarrow Q \\
\{ * I \land g * \} \ S \{ * I * \}
\end{array}
\implies \{ * I * \} \text{ while } g \text{ do } S \{ * Q * \}
\]

\[ \square \]
A predicate \( I \) satisfying the second premise above is also called \textit{invariant}, because it is preserved by each iteration. Note that an iteration may temporarily destroy an invariant (cause it to become false), but in the end it must re-establish it. An invariant can also be thought to specify the goal of every iteration, since it should hold at the end of all iterations.

If you have an arbitrary specification \( \{∗P∗\} \text{ while } g \text{ do } S \{∗Q∗\} \) to prove, the above rule seems to require you prove that \( P \) is invariant. In most cases the pre-condition that you have will actually not be an invariant. However, recall that we can weaken a pre-condition. It is sufficient to find a \textit{weaker} predicate than \( P \), which is an invaraint. This is expressed by the following version of the \textit{Loop Reduction} Rule:

\begin{center}
\textbf{Rule 6.3.2} : \textsc{Loop Reduction in Partial Correctness}
\end{center}

\[
\begin{array}{c}
\models P \Rightarrow I \\
\models I \land \neg g \Rightarrow Q \\
\{∗I \land g∗\} \ S \ \{∗I∗\} \\
\{∗P∗\} \ \text{while } g \text{ do } S \ \{∗Q∗\}
\end{array}
\]

\[\square\]

Let us now take a look at what would be needed to also prove the termination of a loop. This is particularly important for programs whose termination are not obvious, e.g. as in the Euclides’ program for computing the greatest common divisor you saw in Chapter 1. One way to show termination is to first come up with an integer expression, say, \( m \) whose value is decreased by every iteration. Subsequently, we show that at the beginning of every iteration \( m \) has a fixed lower bound, say 0. In other words, at beginning of every iteration \( m > 0 \) holds. Suppose the loop does not terminate. Every iteration decreases the value of \( m \). As the loop keeps iterating, at some point \( m \) will become negative. However, this is a contradiction, since we have shown that at the beginning of every iteration \( m > 0 \) holds. So, the assumption that the loop does not terminate must be false. In other words, in this way we have shown that the loop terminates.

Since \( g \) holds at the beginning of every iteration, showing \( g \Rightarrow m > 0 \) will imply that \( m > 0 \) holds at the beginning of every iteration. If you have shown that \( I \) is invariant (that is, it is preserved by every iteration), then you know that \( I \) also holds at the beginning of every iteration. Hence it is sufficient to show \( I \land g \Rightarrow m > 0 \). The requirement that every iteration decreases \( m \) can be captured by:

\[\{∗I \land g∗\} \ (C := m; S) \ \{∗ m < C ∗\}\]

where \( C \) is a fresh auxiliary variable introduced to record the value of \( m \) at the beginning of \( S \). Recall that an auxiliary variable is a variable introduced just for the purpose of specifying a property of a program. It can be added to a program but only in a way that does not influence the behavior of the original program.

This is formalized in the following inference rules. This one is for total correctness of course, because it contains premises implying termination.

\begin{center}
\textbf{Rule 6.3.3} : \textsc{Loop Reduction for Total Correctness}
\end{center}

\[
\begin{array}{c}
\textbf{TC1} : \{∗I \land g∗\} \ (C := m; S) \ \{∗ m < C ∗\} \\
\textbf{TC2} : \models I \land g \Rightarrow m > 0 \\
\textbf{EC} : \models I \land \neg g \Rightarrow Q \\
\textbf{IC} : \{∗I \land g∗\} \ S \ \{∗I∗\} \\
\{∗I∗\} \ \text{while } g \text{ do } S \ \{∗Q∗\}
\end{array}
\]

In \textbf{TC1} \( C \) should be a fresh variable.

\[\square\]
The integer expression \( m \) used to argue termination is also called *termination metric*. In a more general setup you may need a more sophisticated termination metric, e.g. a lexicographically ordered tuple of integers. The general requirement remains the same: it has to be strictly decreasing, and has a lower bound. In this book, we will only use integer-typed termination metrics.

For completeness below we also show the variant of the rule above that allows you to directly prove a specification with an arbitrary pre-condition \( P \):

**Rule 6.3.4 : Loop Reduction for Total Correctness**

\[
\begin{align*}
\text{InitC} & : \models P \Rightarrow I \\
\text{TC1} & : \{* I \land g \} \; (C := m; S) \; \{* m < C *\} \\
\text{TC2} & : \models I \land g \Rightarrow m > 0 \\
\text{EC} & : \models I \land \neg g \Rightarrow Q \\
\text{IC} & : \{* I \land g \} \; S \; \{* I *\} \\
\end{align*}
\]

\{* P *\} while \( g \) do \( S \) \{* Q *\}

In TC1 \( C \) should be a fresh variable.

There is unfortunately no algorithm to (automatically) construct a right \( I \). In fact, finding one can be a challenging puzzle. E.g. choosing a weak \( I \) may cause EC, TC1, or TC2 to become invalid. For these you would prefer a strong \( I \). However, if \( I \) is too strong InitC may become invalid. Proving IC poses a similar dilemma because \( I \) appears in both pre- and post-conditions.

### 6.4 Organizing Your Proof

Before we show you examples involving concrete programs with loops, let us first discuss how to organize your top level proof so that it gives you a better overview.

If a program is too large to be proven in a single step we can split its specification by adding assertions. Recall that adding assertions is safe. That is, it cannot turn an invalid specification to valid. For example, suppose we have this specification to prove:

\[
\{* P *\} S_1 ; S_2 \{* R *\}
\]

(6.3)

By inserting a new assertion \( Q \), we can instead try to prove:

\[
\{* P *\} S_1 \{* Q *\} ; S_2 \{* R *\}
\]

(6.4)

The new specification is more helpful as it provides you with more information, though in exchange now you have two goals to prove:

\[
\{* P *\} S_1 \{* Q *\} \text{ and } \{* Q *\} S_2 \{* R *\}
\]

A frequently appearing program shapes are what we call *IL* and *ILF* shapes: IL stands for 'init+loop', and ILF for 'init+loop+finalization'. The first is a program with a single loop preceded by an initialization code, like this:

\[
S_i ; \text{while } \! g \text{ do } S
\]

A ILF-shaped program is an IL program followed by another piece of code that we call the finalization code. Let us first consider IL-shaped programs. Consider this specification:

\[
\{* P *\} S_i ; \text{while } \! g \text{ do } S \{* Q *\}
\]

Inspired by the previous discussion about splitting up a specification, we add an intermediate assertion \( I \), obtaining this:

\[
\{* P *\} S_i ; \{* I *\} \text{ while } \! g \text{ do } S \{* Q *\}
\]

which requires you to prove two things:
C1: prove that the loop initialization part $S_i$ can establish $I$.

C2: prove $\{ * I * \} \textbf{while} \ g \ \textbf{do} \ { * Q *}$

C2 can be proven using one of the Loop Reduction rules from the previous section. You can choose $I$ such that it is an invariant of the loop.

Figure 6.4 shows the global structure of the correctness proof of a specification of an IL shaped program. It consists of five separate proofs. Init proves the condition C1 above. PTC1, PTC2, PEC, and PIC prove the conditions required by Rule 6.3.3, which in turns implies C2 above. The rule requires a termination metric. This is specified as a comment next to the assertion specifying the loop’s invariant.

A variant of the IL shape is the ILF shape. An ILF-shaped program has an IL shape followed by a piece of so-called finalization code, like this:

$\{ * P * \} \ S_i; \ \textbf{while} \ g \ \textbf{do} \ S_f \ { * R *}$

$S_f$ is the finalization code. If it does not contain any loop then we can calculate its wp with respect to $R$, and thus reducing the above specification to the following:

$\{ * P * \} \ S_i; \ \textbf{while} \ g \ \textbf{do} \ { * \text{wp} \ S_f \ R *}$

which has an IL shape and can thus be handled in the way we discussed above.
6.4. ORGANIZING YOUR PROOF

\{
\begin{verbatim}
\{ * P, see PROOF Init * \}
S_i;
\{ * I, see PROOF PTC1, PTC2, PEC, PIC. Termination metric: ... * \}
while g do S
\{ * Q * \}
ASSUMING
I = ... 
\end{verbatim}\}

PROOF Init

Here you prove the initialization part: \{ * P * \} S_i \{ * I * \}. If S_i does not contain any
loop, then you can do this by showing that P implies wp S_i I.

PROOF PTC1

Here you prove the TC1 premise (see the Loop Reduction Rule). This premise, together
with TC2, is required to prove that the loop terminates. Basically, what you prove
here is that every iteration will decrease the value of the termination metric.

PROOF PTC2

Here you prove the TC2 premise, namely that at the beginning of every iteration, the
value of the termination metric is positive.

PROOF PEC

Here you prove the EC premise, namely that at the end of the loop, the invariant and
the negation of the guard imply the specified post-condition of the loop.

PROOF PIC

Here you prove the IC premise, namely that body of the loop re-establishes the invari-
ant I.

Figure 6.4: The global structure of the proof of an IL shaped program.
6.5 Example: Fibonacci Problem

You have now been introduced to all the basic inference rules to verify the correctness of imperative sequential programs. You have also seen one example, but it was a rather trivial example. Let us now consider a slightly larger example.

This is a classical example. In 1202 Leonardo Fibonacci investigated a model to calculate how fast a rabbit population grows. The model is perhaps overly simple, but it still makes an interesting programming example. In the model the population starts with a newly born pair of rabbits (one male and one female). The rabbits are assumed to live forever. Every pair will become mature after two months and will then produce a new pair every month. The environment in which the rabbits live is assumed to be sufficient and stable. The table below shows how many pairs of rabbits this model will produce at the end of the \( n \)\textsuperscript{th} month:

<table>
<thead>
<tr>
<th>month ((n):)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>#rabbits (\text{in pairs})</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
<td>144</td>
</tr>
</tbody>
</table>

The sequence of numbers in the second row is popularly called the Fibonacci sequence. Let us use the notation \( \text{fib} \ n \) to denote the \( n \)-th number in the sequence. Looking at the sequence above, you may observe that it seems that it can produced by the following recursive function:

\[
\begin{align*}
\text{fib} 0 &= 0 \\
\text{fib} 1 &= 1 \\
\text{fib} (n + 2) &= \text{fib} (n + 1) + \text{fib} n
\end{align*}
\]

for all \( n \geq 0 \). This (recursive) function can be directly implemented as a program. Unfortunately the program would be very inefficient. For example, \( \text{fib} 5 \) recursively calls \( \text{fib} 3 \) and \( \text{fib} 4 \). The latter also calls \( \text{fib} 3 \). So, \( \text{fib} 3 \) is unnecessarily called twice. Each call to \( \text{fib} 3 \) in turn calls \( \text{fib} 1 \) and \( \text{fib} 2 \). A call to \( \text{fib} 2 \) also calls \( \text{fib} 1 \), so again we have duplicated calls; and so on.

Alternatively, we can also construct a program that implements the rabbit model more directly. An example is shown Figure 6.5. As a correctness criterion we can require that it should produce the same result as the function \( \text{fib} \) above.

The program reads an integer \( n \), and will return the number of rabbits (in pairs) after \( n \) months. The variable \( t \) models time in months. The program starts with \( t=1 \). The variable \( n\text{Now} \) maintains
the number of rabbits (in pairs) that we currently have after \( t \) months. The variable \texttt{nBefore} maintains the number of rabbits we have the month before.

It starts of course with \texttt{nNow}=1, since the model starts with one pair of rabbits. The value of \texttt{nBefore} is set to 0, because before we have no rabbit at all. As we move from time \( t \) to \( t+1 \), we will get new pairs of rabbits. A pair become mature after two months. So, as we enter \( t+1 \)-th month, the number of mature pairs is equal to \texttt{nBefore}, and these will give an equal number of new pairs. So, at the end of every iteration, the new number of pairs is equal to the current number of pairs (\texttt{nNow}) plus \texttt{nBefore}. Compare this explanation with the program code.

The function \texttt{fib} in (6.5) provides an abstract and elegant way to characterize the Fibonacci sequence. However, to actually produce the numbers in the sequence, \texttt{FIB} (the imperative version in Figure 6.5) is more efficient. The loop will only iterate \( n \) times to compute \( \texttt{fib} n \), whereas the function \texttt{fib} itself requires in the order of \( 2^n \) recursive calls. However, we have yet to show that \texttt{FIB}, as its specification claims, is indeed a consistent implementation of \texttt{fib}. Let us now see how the theory you have seen so far can be used to do so.

Although it is quite obvious that \texttt{FIB} terminates, for the sake of example we will still prove its termination. So, the given specification in Figure 6.5 is to be interpreted under the total correctness interpretation.
6.5.1 Step 1: Proof Plan

The program has an ILF shape. Following the strategy discussed in the previous section we draw an outline for our proof —see Figure 6.6. The outline is not the complete proof. Its purpose is to line out the global structure of our proof and to give an overview of the formulas we have to prove (also called proof obligations). We will also call such an outline a proof plan. We will gradually fill it in with details until it becomes a complete proof.

Notice that a new assertion \( n \text{Now} = \text{fib} \ n \) is inserted between the loop and the final assignment in the program. However, the proof plan in Figure 6.6 does not include a proof for:

\[
\{ \ast n \text{Now} = \text{fib} \ n \ast \} \quad \text{return} \ n \text{Now} \quad \{ \ast \text{return} = \text{fib} \ n \ast \}
\]

As commented in the proof plan, the assertion \( n \text{Now} = \text{fib} \ n \) is obtained by calculating the wp of \( \text{return} \ n \text{Now} \) with respect to the post-condition \( \text{return} = \text{fib} \ n \). Being the weakest pre-condition it automatically satisfies the above specification.

For the purpose of computing wp, the statement \( \text{return} \ n \text{Now} \) is treated as if it is an assignment \( \text{return} := n \text{Now} \). This yields indeed \( n \text{Now} = \text{fib} \ n \) as the weakest pre-condition.

Step 2: Constructing I

As said before, there is no general algorithm to construct I. There are however some strategies. For example we can try to construct I by incrementally adding details to it. The final I has this form: \( I_1 \land \ldots \land I_n \). We start by giving a guess for \( I_1 \) and assume that \( I = I_1 \land h_1 \) where \( h_1 \) is unspecified and is used as a place holder for further refinement. As we work out the proofs we may find out that \( I_1 \) alone is not sufficient to close all the proofs. We fix I by instantiating (refining) \( h_1 \) to \( I_2 \land h_2 \) for some \( I_2 \). So, now we have \( I = I_1 \land I_2 \land h_2 \), with \( h_2 \) as a new place holder for further refinement. The process is repeated until you can close all the proofs.

Note that the strategy above fails if you start with an \( I_1 \) which is already too strong, and similarly if at some point you choose an \( I_i \) which is overly strong. So, it requires some wisdom in making the choices.

As you gain experience, you will become more skilled in guessing a right I. Many programs are also quite similar to each other. Similar programs tend to have similar sorts of invariants.

For the first guess usually PEC provides a useful clue. Recall that PEC concerns the situation when the loop terminates. We have to show that I and the guard being false (implying the end of the loop) implies the specified post-condition of the loop. This is the only condition in the Loop Reduction rules that directly relates I with the loop’s specified goal, which is why it often gives useful insight for the choice of I. Now, this is what we have to prove in PEC:

\[
I \land (t = n) \Rightarrow n \text{Now} = \text{fib} n
\]  

(6.6)

A trivial choice of I is \textit{false}. It would satisfy the above implication, but unfortunately Init, for example, will then become impossible to prove. So \textit{false} is a useless choice.

Another trivial I that would satisfy the above implication is \( n \text{Now} = \text{fib} \ t \). Is this a good choice? The only way to really verify it is of course by proving it. But we can first give it some further informal consideration. Recall that \( n \text{Now} \) is supposed to contain the current number of the rabbits (in pairs). That is, the number of rabbits at time \( t \). This is supposedly equal to \( \text{fib} \ t \), suggesting that \( n \text{Now} = \text{fib} \ t \) is an invariant property, and hence indeed a part of I. So, let us indeed put it in our I:

\[
I = (n \text{Now} = \text{fib} \ t) \land h_1
\]  

(6.7)

where \( h_1 \) is still unknown and is a place holder for further refinement. We add this new information in the proof plan. The changes are shown in below:
6.5. EXAMPLE: FIBONACCI PROBLEM

\{ \ast n > 0 \ast \}\nFIB's body
\{ \ast \text{return } = \text{fib } n \ast \}\n
ASSUMING

I = (n\text{Now } = \text{fib } t) \land h_1

PROOF Init ...
...
PROOF PEC ...
...

As said PEC is trivial, but for the sake of completeness it is shown below:

PROOF PEC
[\text{A:}] I \land (t = n)
[\text{G:}] n\text{Now } = \text{fib } n

BEGIN

1. { follows from A and the def. of I } n\text{Now } = \text{fib } t
2. { follows from A and the def. of I } t = n
3. { rewrite 1 with 2 } n\text{Now } = \text{fib } n

END

Note that the proof above does not depend on what \( h_1 \) is. Consequently, instantiating it will not invalidate the conclusion of PEC.

6.5.2 Step 3: Cross Checking with Init

Init is where we prove that the initialization code can setup I for the loop as the loop is entered for the first time. Obviously an I which is not realizable in this sense is not very useful. Checking Init is usually easy. Each time you refine I, you may want to cross check it against Init before you start working on PIC, which is usually much more involved. Let us do so with our example. We cannot complete Init yet, because I is only partially known. However we can make this calculation:

\[
\text{wp} (t := 1; \text{nBefore } := 0; \text{nNow } := 1) \ I
= \{ \text{definition of I } \} \\
\text{wp} (t := 1; \text{nBefore } := 0; \text{nNow } := 1) ((\text{nNow } = \text{fib } t) \land \ldots) \\
= \{ \text{calculating the wp } \} \\
(1 = \text{fib } 1) \land \ldots \\
= \{ \text{definition of fib } \} \\
(1 = 1) \land \ldots
\]

This shows that at least the known part of I will not cause a problem in Init, because after the wp calculation it is reduced to \( 1 = 1 \) which is trivially true.
6.5.3 Step 4: Verifying Invariance

So now we know that our choice of $I$ so far is strong enough to prove PEC’s goal and does not seem to cause a problem in $\text{Init}$. The next thing to do is to verify the invariance of $I$. This is done in PIC (see the proof plan). More precisely, you have to show:

$$\{ * I \land (i \neq n) \} "\text{loop's body}" \{ * I \}$$  \hspace{1cm} (6.8)

Notice that the definition of $I$ specifies a property of $\text{nNow}$. Each iteration updates $\text{nNow}$ using $\text{nBefore}$. So, in order to prove the property about $\text{nNow}$ you can expect that you need to know something about $\text{nBefore}$. However, the pre-condition of the loop’s body, which is $I \land i < n$, does not contain any information about $\text{nBefore}$. So you can expect that the proof will fail. The calculation below will show you. To prove (6.8) we can equivalently prove:

$$I \land (i \neq n) \Rightarrow \wp "\text{loop's body}" I$$  \hspace{1cm} (6.9)

We calculate the $\wp$:

$$\wp "\text{loop's body}" I$$

$$= \{ \text{definition of } I \}$$

$$\wp "\text{loop's body}" ((\text{nNow} = \text{fib } t) \land ...)$$

$$= \{ \text{calculating } \wp \}$$

$$(\text{nNow} + \text{nBefore} = \text{fib } (t + 1)) \land ...$$

There is no way to prove the last formula above from $I \land (t \neq n)$ because $I$ does not say anything about $\text{nBefore}$.

If you recall the explanation given for the code of $\text{FIB}$ (page 77), the variable $\text{nBefore}$ is used to hold the number of rabbits (in pairs) at time $t - 1$. The loop seems to maintain this relation. In other words, it seems to be an invariant property —perhaps we can use this fact. To do so, we add it to our $I$. This is done by instantiating $h_1$ in the previous definition of $I$ (6.7) to:

$$h_1 = (\text{nBefore} = \text{fib } (t - 1)) \land h_2$$  \hspace{1cm} (6.10)

where $h_2$ is unknown and serves as a new place holder. So, $I$ is now:

$$I = (\text{nNow} = \text{fib } t) \land (\text{nBefore} = \text{fib } (t - 1)) \land h_2$$  \hspace{1cm} (6.11)

With this new information now we redo the calculation:

$$\wp "\text{loop's body}" I$$

$$= \{ \text{definition of } I \}$$

$$\wp "\text{loop's body}" ((\text{nNow} = \text{fib } t) \land (\text{nBefore} = \text{fib } (t - 1)) \land h_2)$$

$$= \{ \text{calculating } \wp \}$$

$$(\text{nNow} + \text{nBefore} = \text{fib } (t + 1)) \land (\text{nNow} = \text{fib } t) \land ...$$

As before, we have to prove that $I \land (t \neq n)$ implies the last formula above. Assuming $I \land (t \neq n)$ we continue the calculation we did above:

$$(\text{nNow} + \text{nBefore} = \text{fib } (t + 1)) \land (\text{nNow} = \text{fib } t) \land ...$$

$$= \{ I \text{ is assumed, and it says that } \text{nNow} = \text{fib } t \}$$

$$(\text{nNow} + \text{nBefore} = \text{fib } (t + 1)) \land \text{true} \land ...$$

$$= \{ \text{using the assumed } I \text{ again; it says } \text{nBefore} = \text{fib } (t - 1) \}$$

$$(\text{fib } t + \text{fib } (t - 1) = \text{fib } (t + 1)) \land \text{true} \land ...$$
6.5. EXAMPLE: FIBONACCI PROBLEM

\[
\{ \text{simplifying} \} \\
(fib \ t + fib \ (t - 1) = fib \ (t + 1)) \land \ldots \\
= \{ \text{definition of fib} \} \\
(fib \ (t + 1) = fib \ (t + 1)) \land \ldots
\]

The equality in the last expression is trivially true. Are we done with \text{PIC}? Well, not really. The last step above is not entirely correct.

In the last step we use the definition of \text{fib} to rewrite \( fib \ t + fib \ (t - 1) \) to \( fib \ (t + 1) \). However, this two are only equal if \( t > 0 \), which cannot be inferred from \( I \land (k \neq n) \) using the above choice of \( I \). So, we still need to extend \( I \) with further detail. An easy solution is to simply extend \( I \) with \( t > 0 \) itself. This is reasonable since the loop starts with \( t = 1 \) and increases it at each iteration. So, \( t > 0 \) is an invariant property. It is also not too difficult to see that the iterations will also maintain \( t \leq n \). We do not need this knowledge now, but we will need it later as we prove the loop’s termination. So, we patch the definition of \( h_1 \) (6.10) to include the above details:

\[
h_1 = (n_{\text{Before}} = fib \ (t - 1)) \land 0 < t \leq n \land h_2
\]

(6.12)

So, the new \( I \) is:

\[
I = (n_{\text{Now}} = fib \ t) \land (n_{\text{Before}} = fib \ (t - 1)) \\
\land 0 < t \leq n \\
\land h_2
\]

(6.13)

For later it is convenient to introduce \( J \) and write \( I \) like this:

\[
J = (n_{\text{Now}} = fib \ t) \land (n_{\text{Before}} = fib \ (t - 1)) \land 0 < t \leq n
\]

The new proof plan should now look like this (leaving out the parts which are not changed):

\[
\{ * n > 0 * \} \\
\text{FIB’s body} \\
\{ * \text{return} = \text{fib} \ n * \}
\]

\text{ASSUMING}

\[
I = J \land h_2 \\
J = (n_{\text{Now}} = fib \ t) \land (n_{\text{Before}} = fib \ (t - 1)) \land 0 < t \leq n
\]

\ldots

\text{PROOF PIC} \ldots \\
\ldots
\]

\text{PIC} is shown in Figure 6.7. It is not complete yet: \( h_2 \) is unknown, so we cannot prove its invariance yet (the second goal in Figure 6.7).

The proof uses the following useful property of \text{wp} saying that the \text{wp} of some statement \( S \) with respect to \( Q_1 \land Q_2 \) can be constructed by \emph{separately} calculating the \text{wp} with respect to \( Q_1 \) and \( Q_2 \), and then taking the conjunction of the results. A similar property holds for disjunction, provided \( S \) is deterministic. The proof can be found in Section 6.9.

\textbf{Theorem 6.5.1 : wp Distributivity}

1. \( \vdash \text{wp} \ S \ (Q_1 \land Q_2) = (\text{wp} \ S \ Q_1) \land (\text{wp} \ S \ Q_2) \)

2. If \( S \) is deterministic, we have:

 \[
\vdash \text{wp} \ S \ (Q_1 \lor Q_2) = (\text{wp} \ S \ Q_1) \lor (\text{wp} \ S \ Q_2)
\]

\( \square \)
PROOF PIC

[A :
I ∧ (t ≠ n)
[D1:] Q₁ = wp "loop’s body" J
[D2:] Q₂ = wp "loop’s body" h₂
[G :] wp "loop’s body" (J ∧ h₂)
BEGIN

1. { follows from A and def. of I }
   (nNow = fib t) ∧ (nBefore = fib (t − 1)) ∧ 0 < t ≤ n

2. { wp calculation and def. of J }
   Q₁ = (nNow + nBefore = fib (t + 1)) ∧ (nNow = fib t) ∧ 0 < t + 1 ≤ n

3. { see the equational proof below }
   nNow + nBefore = fib (t + 1)

   EQUATIONAL PROOF
   nNow + nBefore
   = { PIC.1 implies nNow = fib t and nBefore = fib (t − 1) }
   fib t + fib (t − 1)
   = { PIC.1 implies t > 0, use the def. of fib }
   fib (t + 1)

END

4. { ∧ elimination on 1 } nNow = fib t

5. { follows trivially from 0 < t, which is implied by 1 } 0 < t + 1

6. { follows from 1 and A } t ≤ n ∧ t ≠ n

7. { follows from 6 } t + 1 ≤ n

8. { follows from 3,4,5,7, and 2 } Q₁

9. { proof is delayed } Q₂

10. { follows from 8,9, and wp Distributivity (Theorem 6.5.1) }
    wp "loop’s body" (J ∧ h₂)

END

Figure 6.7: The PIC part of the proof of FIB. The justification of step 9 is still missing.
6.5.4 Step 5: Loop Termination

Intuitively, it is quite clear that FIB’s loop will terminate. Nevertheless for the sake of example, we will still show the formal proof of this. Recall the discussion about the Loop Reduction rules, in particular Rule 6.3.4: to prove termination we need to find a termination metric $m$. This is an integer expression whose value is decreased by every iteration (proven in PTC1), but bounded below by some constant (proven in PTC2).

Now, each iteration in FIB’s loop increases the value $t$. So, each iteration decreases $-t$. This is a good candidate for $m$. However, our Loop Reduction rule requires an $m$ with 0 as its lower bound; $-t$ does not have this property. This is quite easy to fix. We take $n-t$ instead, which also decreases over the iterations. The invariant $I$ (6.13) says that $t \leq n$. An iteration can only start if $t \neq n$. The last two facts imply that $t < n$, hence $n-t > 0$.

So, to summarize this $m$ will be our chosen termination metric:

$$m = n - t$$

This new information should be added to the proof plan. PTC1 and PTC2 are quite trivial, but for the completeness are shown below.

**PROOF PTC1**

$$[A:] I \land t \neq n$$

$$[G:] n - t > 0$$

BEGIN

1. { follows from the 1-st conjunct of $A$ and definition of $I$ } $t \geq n$

2. { follows from 1 and 2-nd conjunct of $A$ } $t < n$

3. { follows from 2 } $n - t > 0$

END

**PROOF PTC2**

$$[A:] I \land t \neq n$$

$$[D:] Q = \ wp \ (C := n - t; \ loop's body) \ (n - t < C)$$

$$[G:] Q$$

BEGIN

1. { calculating wp } $Q = n - (t + 1) < n - t$

2. { trivial } $n - t - 1 < n - t$

3. { follows from 1 and 2 } $Q$

END

Are we done now? Well, not quite. We still have to do Init and PIC is also incomplete because the proof of the invariance of $h_2$ is still missing; and $h_2$ itself is still unspecified. What should we use for it? Well, in all the proofs we do so far we actually never use it. In other words, the proofs so far do not depend on what $h_2$ is. So, take $h_2 = \text{true}$. With respect to the definition of $I$, this has the same effect as simply dropping $h_2$ from the definition. It also means the still missing part of PIC in which we have to prove $h_2$ can also be dropped.

So, we remove $h_2$. The proof plan should be adjusted accordingly. This leaves us with one remaining proof obligation, namely Init.
6.5.5 Step 6: Initialization

In $\text{Init}$ we have to prove that the initialization code can establish the invariant $I$ as the program enters the loop for the first time. In our case, this is what we have to prove:

$$\{\ast n > 0 \ast\} (t := 1; \text{nBefore} := 0; \text{nNow} := 1) I \quad (6.14)$$

The proof is below.

**PROOF Init**

[A:] $n > 0$

[G:] $\text{wp} (t := 1; \text{nBefore} := 0; \text{nNow} := 1) I$

BEGIN

1. { see the equational proof below } $\text{wp} (t := 1; \text{nBefore} := 0; \text{nNow} := 1) I = 1 \leq n$

   **EQUATIONAL PROOF**

   $\text{wp} (t := 1; \text{nBefore} := 0; \text{nNow} := 1) I$

   = { definition $I$ }

   $\text{wp} (t := 1; \text{nBefore} := 0; \text{nNow} := 1) ((\text{nNow} = \text{fib} t) \land (\text{nBefore} = \text{fib} (t - 1)) \land 0 < t \leq n)$

   = { calculating $\text{wp}$ }

   $(1 = \text{fib} 1) \land (0 = \text{fib} 0) \land 0 < 1 \leq n$

   = { def. of $\text{fib}$ }

   $(1 = 1) \land (0 = 0) \land 0 < 1 \leq n$

   = { simplification }

   $1 \leq n$

   END

2. { follows from A } $1 \leq n$

3. { rewrite 2 with 1 } $\text{wp} (t := 1; \text{nBefore} := 0; \text{nNow} := 1) I$

END

This is also a typical situation where program verification is often successful in flushing out errors. Since a program with a weaker a pre-condition is more versatile (the program accepts a larger set of inputs), a less experienced programmer may be tempted to weaken the pre-condition of $\text{FIB}$, for example to $0 \leq n$. Unfortunately this will cause a problem if the program is called with $n = 0$ (it does not terminate). Such an error may slip unnoticed through the testing phase, because it only happens to one input combination out of infinitely many of them. There is however no way the error can slip through program verification. If you set the pre-condition to $0 \leq n$, there is simply no way you can prove $I$ in $\text{Init}$.

6.5.6 Summarizing

Now we have completed all the proofs, and hence the program $\text{FIB}$ is correct with respect to its specification. We have shown to you how to incrementally construct an invariant. During the process we extended $I$ several times. For your overview Figure 6.8 shows the final proof plan. It shows the final $I$ and the termination metric $m$. The proofs are not repeated in the Figure—we have shown them all.

The incremental approach is typically used when you are dealing with programs whose structures or algorithms may be new to you. As you gain in experience you will be getting better in 'guessing' invariants. You may even be able to immediately guess a correct invariant when given a simple program.
6.6. ON ITERATIVELY CONSTRUCTING $I$

Consider again the steps we took to construct the invariant in the Fibonacci example. We started with a simple $I$ (6.7), namely:

$$I = (n_{\text{Now}} = \text{fib} t) \land (n_{\text{Before}} = \text{fib} (t - 1)) \land 0 < t \leq n$$

where $h_1$ is unspecified. As we proceeded we discovered that with this $I$ we could not close PIC. After some calculation, we ended up having to prove that $I$ implies:

$$n_{\text{Now}} + n_{\text{Before}} = \text{fib} (t + 1)$$

(6.15)

We could not prove this because $I$ contains no information about $n_{\text{Before}}$. So, in (6.10) and (6.11) we decided to extend the candidate invariant $I$ with:

$$n_{\text{Before}} = \text{fib} (t - 1)$$

as a new conjunct. The discovery of this fact about $n_{\text{Before}}$ was motivated by appealing to your programmer insight on the program’s code. It worked. However programming insight or intuition will be much less effective when we have to deal with complicated programs.

Rather than relying on insight we can just extend $I$ with the formula on which either PIC, PTC1, or PTC2 fail. We then rework the proofs, and iterate this process until we can close all the proofs. So, in the above case we would just add (6.15) to $I$. It is intuitively less obvious that it is an invariant property (it is!), but on the other hand discovering it does not require any thinking: if we cannot prove it from $I$, we add it.
This process will always drive you towards an \( I \) satisfying all the conditions required by Rule 6.3.3. But it is a costly procedure. It may also not terminate, and if it does you may still end up with an \( I \) which is too strong to be setup by your loop initialization code and its given pre-condition.

We will try this strategy in the next example. After you have studied it, as an exercise you can redo the proof of \( \text{FIB} \) using the same strategy.

6.7 Linear Scanning

In the following example we have a program that works on an array. Its specification is slightly more complicated than that of the previous example. So is its proof. Below is a simple program to sum all positive integers in the array \( a \), in the domain \( 0 \leq i < n \).

Example 6.7.1 :

\[
\{ \ast \ n \geq 0 \ast \} \\
\text{SUMP (READ } a : \text{int[]}, n : \text{int}, \text{OUT s : int) : ()}
\]

\[
\{ \text{ var i : int ; i := 0 ; s := 0 ; while } i \neq n \text{ do} \\
\{ \text{ if } a[i] > 0 \text{ then } s := s + a[i] \text{ else skip ;} \\
\{ i := i + 1 \} \\
\}
\]

\[
\{ \ast \ s \ = \ \text{SUM}[x | x \text{ from } a[0\ldots n], x > 0] \ast \}
\]

The program has an IL shape. Following the strategy in Subsection 6.4 we extend the specification as shown in Figure 6.9. The figure also shows our initial proof plan. To complete the proof you need to come up with a right \( I \) and \( m \).

6.7.1 Constructing Invariant

Arbitrarily trying out different candidates for \( I \) is not a successful strategy. We will construct it incrementally. We first take a look at the post-condition for clues. In \( \text{PEC} \) (see the proof plan) you have to prove that \( I \land (i = n) \), which is assumed to hold when the loop in \( \text{SUMP} \) terminates, implies the loop’s specified post-condition. That is, we have to prove:

\[
I \land (i = n) \Rightarrow (s = \text{SUM}[x | x \text{ from } a[0\ldots n], x > 0]) \quad (6.16)
\]

Let us first introduce the following abbreviation:

\[
\text{sum a n} = \text{SUM}[x | x \text{ from } a[0\ldots n], x > 0] \quad (6.17)
\]

So, now we can write (6.16) as:

\[
I \land (i = n) \Rightarrow (s = \text{sum a n}) \quad (6.18)
\]

This is shorter. Moreover, it is now more obvious that a quite trivial choice for \( I \) is \( s = \text{sum a i} \). Is this a reasonable choice? Well, at least initialization will not be a problem:
6.7. LINEAR SCANNING

\{ * n ≥ 0 , see proof Init * \}

\[ i := 0 ; s := 0 ; \]
\{ * I , see proofs PTC1, PTC2, PEC, PIC. Termination metric: m = ... * \}

while \( i \neq n \) do
\{ if \( a[i] > 0 \) then \( s := s + a[i] \) else skip \;
\[ i := i + 1 \]
\{ \( s = \sum [x \mid x \text{ from } a[0 \ldots n], x > 0] \) \}

ASSUMING

I = ...

PROOF Init
... (here you prove \{ * n ≥ 0 * \} \( i := 0 ; s := 0 \) \{ * I \})

PROOF PTC1
... (here you prove \{ * I \land i \neq n * \} \( C := m; " \text{loop's body}" \) \{ * m < C * \})

PROOF PTC2
... (here you prove \( I \land i \neq n \Rightarrow m > 0 \))

PROOF PEC
... (here you prove \( I \land (i = n) \Rightarrow (s = \sum [x \mid x \text{ from } a[0 \ldots n], x > 0]) \))

PROOF PIC
... (here you prove \{ * I \land i \neq n * \} "loop's body" \{ * I \})

Figure 6.9: An initial proof plan of SUMP

\[ wp (i:=0; s:=0)(s = \text{sum } a i) \]
\[ = \]
\[ 0 = \text{sum } a 0 \]
\[ = \{ \text{def. sum } \} \]
\[ 0 = 0 \]
\[ = \]
\[ \text{true} \]

We can further rationalize the choice by appealing to our programming insight on how SUMP’s loop works. However, let us not do this now. We pretend that the loop is complicated, and try, this time, to just rely on our calculation. So, to summarize, for now we take this as \( I \):

\[ I = (s = \text{sum } a i) \land h_1 \]  \hspace{1cm} (6.19)

where \( h_1 \) is a place holder for further refinement of \( I \). PEC, as already motivated above, is trivial and is not shown.

6.7.2 Loop Termination

To formally prove termination we need a termination metric \( m \), which is an integer expression whose value is decreased by every iteration (proven in PTC1), but bounded below by 0 (proven in PTC2). Now, each iteration increases \( i \), but leaves \( n \) unchanged. Therefore, it decreases \( n - i \) —let us take this as our \( m \). It has to be bounded below by 0. In this case, it means:

\[ I \land (i \neq n) \Rightarrow 0 < n - i \]  \hspace{1cm} (6.20)
In terms of implication from $I$ we can equivalently express the above as:

$$ I \Rightarrow ((i \neq n) \Rightarrow 0 < n-i) $$

If we simplify this a bit, it is equivalent to:

$$ I \Rightarrow i \leq n $$  \hspace{1cm} (6.21)

Unfortunately, we cannot prove this $i \leq n$ from the information we have in $I$ so far. So, according to the strategy in Section 6.6, we simply add $i \leq n$ to $I$. We can do so by instantiating the place holder $h_1$ to $i \leq n \land h_2$, where $h_2$ is a new place holder for further refinement. So, now $I$ is:

$$ I = (s = \text{sum } a \ i) \land i \leq n \land h_2 $$  \hspace{1cm} (6.22)

With this choice of $I$ PTC2 is of course trivial. We leave it out for you.

That $n$ is decreasing has been argued informally above. For completeness we show PTC1 below—it is quite trivial. Notice that since we have an if-then-else in the loop’s body, the resulting wp is of the form $g \rightarrow f \mid g$. In this case it is actually of the form $g \rightarrow f \mid f$, which is of course equal to $f$ (see the step marked with (*)). The assumption $A$ in is not used in the proof, but is there for formality.

**PROOF PTC1**

[A:] $I \land i \neq n$

[D:] $Q = \text{wp } (C = n - i; "loop's body") (n - i < C)$

[G:] $Q$

See the equational proof below:

**EQUATIONAL PROOF**

$$ Q $$

= \{ \text{wp calculation} \}

a[i] > 0 \rightarrow n - (i+1) < n - i \mid n - (i+1) < n - i

= \{ (*) \text{ rewrite with COND Conversion (T A.4.4) } \}

n - (i+1) < n - i

= \{ \text{trivial} \}

true

**END**

We can add the new information we gain over $I$ and $m$ to the the proof plan—the changes to the plan are shown below:

{∗ ... ∗}

* initialization *

{∗ I , see proofs PTC1,PTC2,PEC,PIC. Termination metric: n−i ∗}

while $i \neq n$ do { loop's body }

{∗ $s = \text{sum a n}$}

**ASSUMING**

$I = (s = \text{sum } a \ i) \land i \leq n \land h_2$

**PROOF ...**

Notice that PTC1 and PTC2 do not depend on what $h_2$ is. So further extension of $I$ obtained through instantiating $h_2$ will not invalidate those proofs.
6.7. LINEAR SCANNING

6.7.3 Proving Invariance

We have proven that the loop in SUMP terminates and that the state upon termination indeed satisfies SUMP's post-condition. However, the proofs assume the invariance of $I$. Now, in PIC, we are now going to verify this assumption. Given the definition of $I$, this is what we have to prove in PIC:

\[
\{^* \text{I} \land i \neq n \} \text{ "loop body" } \{^* (s = \text{sum a i}) \land i \leq n \land h_2 \}
\]

Since the body does not contain further loop, we can prove this by showing that the pre-condition implies the wp of the body over the specified post-condition. Theorem 6.5.1 allows us to prove the implication separately for the wp of each of the conjunct in the post-condition. See the top level of PIC below:

PROOF PIC

\[A:\] $I \land i \neq n$
\[G:\] wp "loop's body" \((s = \text{sum a i}) \land i \leq n \land h_2\)

BEGIN

1. \{ justified by proof PI1 \} wp "loop's body" \((s = \text{sum a i})\)
2. \{ justified by PI2, left out for you \} wp "loop's body" \(i \leq n\)
3. \{ justified by PI3, postponed \} wp "loop's body" \(h_2\)
4. \{ wp Distributivity (Theorem 6.5.1) on 1.2.3 \}
   wp "loop's body" \((s = \text{sum a i}) \land i \leq n \land h_2\)

END

PI2 proves wp body \(i \leq n\). It is quite easy, and is left out for you. PI3 is supposed to prove wp body \(h_2\), but it cannot be given yet, since \(h_2\) is still unspecified. We will return to this later.

The proof of wp body \((s = \text{sum a i})\) is less trivial. It is shown below, but it is still incomplete. It proceeds as follows. In step 1 we show the result of calculating the wp. The resulting formula is of the form $P \rightarrow R_1 \mid R_2$, which can be proven by proving $P \Rightarrow R_1$ and $\neg P \Rightarrow R_2$ separately (formally, based on Theorem COND Split; see page 31). Finally, step 6 derives the goal.

PROOF PI1

\[D:\] $Q = \text{wp loop body } (s = \text{sum a i})$
\[G:\] $Q$

BEGIN

1. \{ calculating wp \}
   \(Q = a[i] > 0 \rightarrow (s + a[i] = \text{sum a } (i + 1)) \mid (s = \text{sum a } (i + 1))\)
2. \{ rewrite 1 with \(s = \text{sum a i}\), which follows from PIC.A and def. (6.22) of I \}
   \(Q = a[i] > 0 \rightarrow (\text{sum a } i + a[i] = \text{sum a } (i + 1)) \mid (\text{sum a } i = \text{sum a } (i + 1))\)
3. \{ justified by proof PCASE1, see Figure 6.10 \}
   \(a[i] > 0 \Rightarrow (\text{sum a } i + a[i] = \text{sum a } (i + 1))\)
4. \{ justified by PCASE2, see Figure 6.11 \}
   \((a[i] > 0) \Rightarrow (\text{sum a } i = \text{sum a } (i + 1))\)
5. \{ rewrite with COND Split (T 3.8.6) on the conjunction of 3 and 4 \}
   \(a[i] > 0 \rightarrow (\text{sum a } i + a[i] = \text{sum a } (i + 1)) \mid (\text{sum a } i = \text{sum a } (i + 1))\)
6. \{ rewrite 5 with 2 \} \(Q\)

END
CHAPTER 6. THE BASIC PROOF SYSTEM

Let us now take a closer look at step 3 in the proof PI1. Assuming $a[i] > 0$ you have to prove that \( \text{sum } a[i] + a[i] = \text{sum } a(i+1) \). This is a bit tricky. Unfolding the definition of \text{sum}, you have to prove that:

\[
\text{SUM } [x|x \text{ from } a[0...i], x>0] + a[i] = \text{SUM } [x|x \text{ from } a[0...i+1], x>0]
\]  

(6.23)

is equal to:

\[
\text{SUM } [x|x \text{ from } a[0...i+1], x>0]
\]  

(6.24)

We expect: $a[0...i+1] = a[0...i] + [a[i]]$. Hence (6.24) is equal to:

\[
\text{SUM } [x|x \text{ from } (a[0...i] ++ [a[i]]), x>0]
\]  

(6.25)

which is equal to:

\[
\text{SUM } [x|x \text{ from } a[0...i], x>0] + (a[i]>0 \rightarrow a[i] \mid 0)
\]  

(6.26)

And since $a[i]>0$ is assumed, the above is equal to (6.23), and hence we have shown what we want, namely that (6.23) is equal to (6.24).

There is however a detail overlooked in the above argument. We have said that $a[0...i+1]$ is equal to $a[0...i] ++ [a[i]]$. However, this is only the case if $0 \leq i$ —with a negative $i$, $a[0...i+1]$ is empty whereas $a[0...i] ++ [a[i]]$ is not, so they are not equal. See also Theorem 4.1.11. Splitting an array and a domain expression (e.g. $a \leq i < b$) has the same pitfall. In terms of bug, the above unsound reasoning can lead to the conclusion that it is safe to initialize $i$ with for example $-1$, which of course leads to an incorrect result.

So, expression 3 in PI1 cannot be proven without first having the condition $0 \leq i$ derived. Unfortunately, none of the assumptions of PI1 or its ancestors. Essentially, we cannot prove $0 \leq i$ because $I$ does not contain the needed information. So again, we apply the strategy in Section 6.6. We add $0 \leq i$ to $I$ by instantiating $h_2$:

\[
h_2 = 0 \leq i \land h_3
\]  

(6.27)

where $h_3$ is a new place holder. So, $I$ is now:

\[
I = (s = \text{sum } a n) \land i \leq n \land 0 \leq i \land h_3
\]  

(6.28)

Now we can prove expression 3 of PI1. The proof is shown in Figure 6.10. The step marked with (*) is where we use the new information $0 \leq i$. Expression 4 (of PI1) can be proven in much the same way; the proof is shown in Figure 6.11.

We are almost done. PIC is still incomplete. PI3, where we have to prove \text{wp body } h_2 is still missing. We now know more about $h_2$, see (6.27), so perhaps now we can close the proof. See below:

PROOF PI3

\[
[G:] \text{wp body } (0 \leq i \land h_3)
\]  

BEGIN

1. \{ \text{wp calculation} \} \text{ wp body } 0 \leq i = 0 \leq i+1
2. \{ \text{follows from } I \text{ in } PIC.A \} \text{ 0 } \leq i
3. \{ \text{follows from } 2 \} \text{ 0 } \leq i+1
4. \{ \text{rewrite } 3 \text{ with } 1 \} \text{ wp body } 0 \leq i
5. \{ \text{see the explanation below} \} \text{ wp body } h_3
6. \{ \text{Theorem 6.5.1} \} \text{ wp body } (0 \leq i \land h_3)

END
6.7. LINEAR SCANNING

**PROOF PCASE1**

[A:] \( a[i] > 0 \)

[G:] \( \text{sum a i + a[i]} = \text{sum a (i + 1)} \)

BEGIN

1. { follows from PIC.A, def. (6.28) of I }
   \( 0 \leq i \)

2. { see the equational proof below }
   \( \text{sum a i + a[i]} = \text{sum a (i + 1)} \)

**EQUATIONAL PROOF**

\[ \text{sum a (i + 1)} \]

\[ = \{ \text{P11.D} \} \]

\[ \text{SUM} [x|x \text{ from } a[0...i + 1], x > 0] \]

\[ = \{ \text{Enumeration Split (T A.5.8), justified by 1} \} \]

\[ \text{SUM} [x|x \text{ from } (a[0...i] ++ [a[i]]), x > 0] \]

\[ = \{ \text{Comprehension Split (T A.5.12)} \} \]

\[ \text{SUM} ([x|x \text{ from } a[0...i], x > 0] ++ [x|x \text{ from } [a[i]], x > 0]) \]

\[ = \{ \text{Singleton Comprehension (T A.5.10)} \} \]

\[ \text{SUM} ([x|x \text{ from } a[0...i], x > 0] ++ (a[i] > 0 \rightarrow [a[i]] | [])) \]

\[ = \{ \text{homomorphism of SUM (T A.5.16)} \} \]

\[ \text{SUM} [x|x \text{ from } a[0...i], x > 0] + \text{SUM} (a[i] > 0 \rightarrow [a[i]] | []) \]

\[ = \{ \text{def. sum and A} \} \]

\[ \text{sum a i + SUM (true \rightarrow [a[i]] | []))} \]

\[ = \{ \text{COND conversion (T 3.8.5)} \} \]

\[ \text{sum a i + SUM [a[i]]} \]

\[ = \{ \text{def. of SUM} \} \]

\[ \text{sum a i + a[i]} \]

END

END

Figure 6.10: Sub-proof PCASE1.
PROOF PCASE2

[A:] ¬(a[i] > 0)

[G:] \(\sum a_i = \sum a(i+1)\)

BEGIN

1. \{ follows from PIC.A, def. (6.28) of I \}
   \(0 \leq i\)

2. \{ see the equational proof below \}
   \(\sum a_i = \sum a(i+1)\)

   EQUATIONAL PROOF
   \(\sum a(i+1)\)
   
   \(= \{\) same steps as in PCASE1 \(\}\)

   \(\sum [x|x from a[0...i), x > 0] + \sum (a[i] > 0 \rightarrow [a[i] | []])\)
   
   \(= \{\) def. sum and A \(\}\)

   \(\sum a + \sum (false \rightarrow [a[i] | []])\)
   
   \(= \{\) COND conversion (T 3.8.5) \(\}\)

   \(\sum a + \sum []\)
   
   \(= \) def.ofSUM

   \(\sum a\)

   END

END

Figure 6.11: Sub-proof PCASE2.
The justification of \(\text{wp body } h_3\) in step 5 is still missing. We cannot prove it because \(h_3\) is still unknown. However, this and \(\text{Init}\) are the only proofs left to close. Taking the trivial solution \(\text{true}\) for \(h_3\) will do: \(\text{wp body true } = \text{true}\); so step 5 becomes trivial. In \(\text{Init}\) we prefer an \(I\) which is as weak as possible; taking \(\text{true}\) for \(h_3\) will not cause us to fail \(\text{Init}\). By taking \(\text{true}\), we can just as well remove \(h_3\) from the definition of \(I\).

Below is the structure of the new proof plan. It now includes the full invariant, and also an overview of the proof obligations which have been fulfilled.

\[
\{ \ast n \geq 0 \ast \}, \ \text{see proof Init} \\
\text{initialization}; \\
\{ \ast I \ast, \text{see proofs PTC1,PTC2,PEC,PIC.} \ \text{Termination metric: } n-i \ast \}
\]

\[
\text{while } i \neq n \text{ do } \{ \text{loop's body } \}
\]

\[
\{ \ast s = \text{sum a n } \ast \}
\]

\[
\text{ASSUMING}
\]

\[
I = (s = \text{sum a i } \land i \leq n \land 0 \leq i)
\]

\[
\text{PROOF Init to do}
\]

\[
\text{PROOF PTC1 ... done}
\]

\[
\text{PROOF PTC2 ... done}
\]

\[
\text{PROOF PIC ... done}
\]

\[
\text{PROOF PEC ... done}
\]

### 6.7.4 Initialization

It now remains to show, in \(\text{Init}\), that the invariant that we have constructed can be established by the initialization code. More specifically, you have to prove:

\[
\{ \ast n \geq 0 \ast \} \ (i := 0; s := 0) \ \{ \ast I \ast \}
\]

The proof is quite simple:

\[
\text{PROOF Init} \\
[A:] n \geq 0 \\
[G:] \text{wp} (i := 0; s := 0) I
\]

\[
\text{BEGIN}
\]

1. (see the equational proof below) \(\text{wp} (i := 0; s := 0) I = n \geq 0 \)

   \[
   \text{EQUATIONAL PROOF}
   \]

   \[
   \text{wp} (i := 0; s := 0) I
   = \{ \text{def. of } I, \text{calculating } \text{wp} \}
   \]

   \[
   (0 = \text{SUM } [x | x \text{ from a}[0...0], x > 0]) \land 0 \leq n \land 0 \leq 0
   = \{ \text{Empty Enumeration (T A.5.5)} \}
   \]

   \[
   (0 = \text{SUM } [x | x \text{ from } [], x > 0]) \land 0 \leq n \land 0 \leq 0
   = \{ \text{Empty Comprehension (T A.5.9)} \}
   \]

   \[
   (0 = \text{SUM } []) \land 0 \leq n \land 0 \leq 0
   = \{ \text{def. of SUM, simplification } \}
   \]

   \[
   0 \leq n
   \]

   END

2. (rewrite A with 1) \(\text{wp} (i := 0; s := 0) I\)

END

This completes the proof of SUMP.
6.8 Invariants

Quite often, the goal of a loop is to implement the calculation of a certain expression, e.g. to calculate \( \sqrt{x} \), \( \text{fib } t \), \( \sum i : 0 \leq i < n : a[i] \). A loop is needed either because the expression cannot be directly expressed in the given programming language, or we simply want to have our own implementation of it, e.g. to improve performance. Let \( f n \) be the expression we want to calculate, and the post-condition to realize is:

\[
\text{while } g \text{ do } \{ \ast x = f n \ast \} \tag{6.29}
\]

for some result-variable \( x \). E.g. in our previous examples \textsc{fib} and \textsc{sump} in Figures 6.5 and 6.7.1, the post-conditions are \( \text{nNow} = \text{fib } n \) and \( s = \text{sum } a \ n \). The presence of some parameters like \( n \) is typical; it simply reflects the fact that our program itself takes parameters.

What we have seen in the previous examples is that the key part of our invariants have the form of:

\[
x = f i
\]

where \( i \) is a variable or a sub-expression that will progress towards \( n \), so that when the loop terminates with \( i = n \), we then have the desired post-condition in (6.29). For example, in the program \textsc{fib} the post-condition to realize is \( \text{nNow} = \text{fib } n \), and the key part of its invariant is \( \text{nNow} = \text{fib } t \), where \( t \) is its loop counter. In \textsc{sump} the post-condition is \( s = \text{sum } a \ n \), and the key part of its invariant is \( s = \text{sum } a \ i \). Both formulate the invariant in terms of what has been calculated so far. For example:

\[
s = \text{sum } a \ i
\]

expresses that so far we have calculated \( \text{sum } a \ i \): the value of this expression is stored in \( \text{nNow} \).

6.8.1 Head and Tail Invariant

Alternatively, we can also try to formulate an invariant in terms of what still have to be computed. For example, let us define \( \text{sumr } a \ i \ n = (\sum k : i \leq k < n : a[k]) \). The following is also a good invariant for \textsc{sump}:

\[
0 \leq i \leq n \land (s + \text{sumr } a \ i \ n = \text{sum } a \ n) \tag{6.30}
\]

Remember that \textsc{sump}’s loop terminates with \( i = n \). This implies that \( \text{sumr } a \ i \ n = 0 \). Hence \( s = \text{sum } a \ n \).

Notice that the \( \text{sumr } a \ i \ n \) express the part that still has to be computed: its value is not stored in any variable. Termination should imply that this part would then become 'empty', and thus \( s \) will hold the desired goal.

Consider again the general problem in (6.29), with \( x = f n \) as the post-condition. The two forms of invariants have the following general shapes:

\[
I : x = f i \tag{6.31}
\]

\[
I : x \oplus \psi i = f n \tag{6.32}
\]

The latter is also called a tail invariant (and thus some people call the first one a head invariant). In the first case, when the loop terminates with \( i = n \), the post-condition is realized. In the second one, when the loop terminates it should imply that \( \psi i \) would be equal to the unit element of \( \oplus \); thus implying \( x = f n \).

In the case of \textsc{sump} both head and tail invariants are equally natural. For \textsc{fib} it is not straightforward to come up with a tail invariant. Then you can also imagine that there are situations where it is not straightforward to come up with a head invariant. An example of that is the program \textsc{euclid} program you saw in the Introduction —see again Figure 1.1.
Euclid calculates the greatest common divisor of two positive integers \(x\) and \(y\). Its loop’s post-condition is \(x = \gcd(X, Y)\), where \(X\) and \(Y\) represent the initial values of \(x\) and \(y\). Its invariant is:

\[
0 < x \land 0 < y \land (\gcd(x, y) = \gcd(X, Y))
\]  
(6.33)

When its loop terminates we have \(x = y\). This implies \(\gcd(x, y) = x\). Hence, by the invariant above, we now have the post-condition.

The key part of the above invariant is the \(\gcd(x, y) = \gcd(X, Y)\) part. It is not a head invariant, because we have not calculated \(\gcd(x, y)\). We do have some intermediate results in \(x\) and \(y\), but they are not \(\gcd(x, y)\). The invariant does not look to fit the pattern in (6.32) either. At least, not at the first glance. However, the \(\gcd(x, y)\) does represent the remaining work to do to get the value of \(x\) and \(y\) towards the goal \(\gcd(X, Y)\). If we do a little trick to rewrite the invariant to:

\[
x \oplus y = \gcd(X, Y)\text{ where } x \oplus y = \gcd(x, y)
\]

Now it does fit to (6.32). It is just a ‘cosmetical trick’ of course; the important thing is to distinguish between expressing your invariant in terms of what you did or in terms of what is still to be done. The choice depends on which one can be expressed.

Generalizing

In the two examples discussed above, the programs SUMP and Euclid, the post-conditions take the form of \(v = f n\) where \(f\) is some function whose value is to be calculated, and \(v\) is the variable that holds the result of this calculation —see the pattern in (6.29). Let us re-express the problem in a slightly more general way. Consider a variation of such a post-condition where goal is to calculate the value of \(x\) such that \(\phi x = f n\). That is:

\[
\text{while } g 
\text{ do } S \{ \star \phi x = f n \star \}
\]  
(6.34)

If \(\phi\) is invertible then we can indeed re-express the post-condition as \(x = \phi^{-1}(f n)\), which is an instance of the previous problem of "calculating the value of \(f n\)". If \(\phi\) is not invertible then obviously we cannot formulate the problem in that way.

A candidate head invariant for the loop would have this form as its key part:

\[
I : \quad \phi x = f i \text{ with } \phi x = f n \text{ as the post-condition}
\]  
(6.35)

with termination, \(I \land \neg g\), should be as such that it implies \(i = n\).

A candidate tail invariant has this form as its key part:

\[
I : \quad \phi x \otimes \psi i = f n
\]  
(6.36)

For this to work, we need termination to imply that \(\psi i\) is equal to the unit element of \(\otimes\). Note that the above invariant is an instance of (6.32): we can write it as \(x \oplus \psi i\) with \(\oplus\) defined as \(x \oplus y = \phi x \otimes y\).

As an example, consider a program called digits to destruct a non-negative integer \(x\) into a list of its digits. The list will be stored in the array \(a\). If we have such a list, we can easily express what the integer value it represents, namely:

\[
\text{ival a n} = \left( \text{SUM } i: 0 \leq i < n : a[i] \times 10^i \right)
\]

With this we can specify digits as shown in Figure 6.12. The post-condition of its loop is also annotated there.

Here is a good invariant for the loop:

\[
0 \leq x \leq X \land (\text{ival a i} + x \times 10^i = X)
\]  
(6.37)

where \(X\) is the initial value of \(x\) (when it was passed to digits). When the loop terminates, we have \(x = 0\). So, the term \(x \times 10^i\) is also 0. Hence we realize the post-condition \(\text{ival a i} = X\). Notice that the invariant is a tail invariant.
\{ \ast x = X \land X \geq 0 \ast \}\

digits(x:int, a:int[]) : int { 
  var i:int ;
  i := 0 ;
  while x>0 do {
    a[i] = x mod 10 ;
    x = x div 10 
  }
  \{ \ast \text{ival a i = X} \ast \}
  return i 
}
\{ \ast \text{ival a return} = x \ast \}

Figure 6.12: A program to destruct an integer.

6.8.2 Loop for Searching a Solution

Another typical use of a loop is to find a solution of some predicate \( Q \), e.g. to find a minimum of a function (it may have multiple minimums), or a crossing point between two curves. The specification typically has this form:

\[
\{ \ast (\exists x : x \in \Sigma : Qx) \ast \} \quad \text{search} (\Sigma) \quad \{ \ast Q \text{return} \ast \}
\]

where \( Q \) is a parameterized predicate, whose solution is wanted. So, \( Q \text{return} \) means that the return value of the program above satisfies \( Q \); so, it is a solution. Note that \( Q \) itself may have multiple solutions, but here we just want to have one solution.

The \( \Sigma \) in the pre-condition expresses the space within which we will do our search. The pre-condition furthermore says that a solution exists within this \( \Sigma \). In practice you may want drop this condition and have the program to return a special value to signal if a solution cannot be found. For simplicity we will not do this here.

When searching, what you do is essentially to repeatedly and systematically shrink the search space, which you can do with a loop. So, if \( V \) is represents the part of \( \Sigma \) which we have visited, this will make a reasonable candidate of the invariant of that loop:

\[
(\exists x : x \in \Sigma/V : Qx) = (\exists x : x \in \Sigma : Qx)
\]

You can terminate the loop once you can trivially find an element in \( \Sigma/V \) that satisfies \( Q \), e.g. if the set only contains one element. Returning this element will realize the post-condition in (6.38).

Notice that the above invariant is a tail invariant: \( \Sigma/V \) is the part of the search space which still have to be searched.

6.8.3 Iterative Implementation of Tail Recursion

Consider the following two functions to recursively calculate the sum of the elements in a list. You have seen the definition of the first one, \( \text{SUM} \), before, in Chapter 4. The second function \( \text{summacc} \) works differently. It accumulates the result of the summing in its first parameter \( r \), which it passes down through the recursion until it reaches the base case \([],\); at that point \( r \) would contain the final result. You can prove that \( \text{summacc} 0s = \text{SUM} s \), and hence you can use the first to calculate the latter.

\[
\text{SUM} [] = 0 \\
\text{SUM} (x : s) = x + \text{SUM} s
\]
6.8. INVARINANTS

\[
\text{sumacc } r \ [ ] = r \\
\text{sumacc } r \ (x : s) = \text{sumacc } (r + x) \ s
\]  

(6.41)

Notice that when \text{sumacc} recurses, after returning from the recursion the result of the recursion does not need further processing. It is immediately returned. In contrast, in \text{SUM} the result of a recursion does need further processing: it needs to be combined with \(x\) before being returned. The recursion pattern of \text{sumacc} is also called \textit{tail recursion}.

Let us first consider a general form of tail recursive functions:

\[
F \ x = g \ x \rightarrow \text{base } x \mid F (\Delta \ x)
\]  

(6.42)

Notice that when the recursion hits an \(x\) such that \(g \ x\) is true, it stops recursing. In other words, \(g \ x\) is the recursion’s base case. The function \text{base} specifies the final processing upon reaching the base case. The function \(\Delta\) specifies what calculation is to be accumulated into the parameter \(x\) at every recursion.

Such a function can be directly translated to a more efficient iterative solution:

\[
\{ \ast \ \text{true} \ast \} \\
\ x := \alpha \ ; \ \text{while} \neg g \ x \ \text{do} \ x := \Delta \ x \ ; \\
\ r := \text{base } x \\
\{ \ast \ r = F \alpha \ast \}
\]

The program above will calculate \(F \alpha\), for some \(\alpha\) that was given at the start of the program. We can prove the correctness of this loop through the invariant:

\[
I : \ F \ x = F \alpha
\]  

(6.43)

Notice that it has the form of the tail invariant in (6.36).

When the loop terminate, we would have \(I \land g \ x\). When \(g \ x\) holds, the definition of \(F\) (6.42) says that \(F \ x = \text{base } x\), and therefore \(I \land g \ x\) would then imply \(\text{base } x = F \alpha\). And therefore, after the final assignment \(r := \text{base } x\), we will establish the post-condition \(r = F \alpha\).

To prove that the invariant can be maintained we need to prove:

\[
\{ \ast \ F \ x = F \alpha \land \neg g \ x \ast \} \ x := \Delta \ x \ \{ \ast \ F \ x = F \alpha \ast \}
\]

By calculating the \text{wp} of \(x := \Delta \ x\) the above is equivalent to:

\[
F \ x = F \alpha \land \neg g \ x \ \Rightarrow \ F (\Delta \ x) = F \alpha
\]

But this follows from the definition of \(F\) that says that when \(g \ x\) does not hold then \(F \ x\) is equal to \(F (\Delta \ x)\).

If we assume \(F\) to be of type \(A \rightarrow A\), we can show that the above loop terminates by finding a termination metric in the form of a function \(m : A \rightarrow \text{int}\), and showing that \(\neg g \ x\) implies \(m \ x > 0\) and \(m (F \ x) < m \ x\).

Let us now return to the example \text{sumacc} in (6.41). Notice that it can be rewritten as shown below:

\[
\text{sumacc } (r, s) = s = [ ] \rightarrow r \mid \text{sumacc } (r + \text{head } s, \ \text{tail } s)
\]  

(6.44)

where for a non-empty list \(s\), \text{head } \(s\) and \text{tail } \(s\) return respectively the first element of \(s\), and the rest of \(s\) after its first element.

We can easily see that \text{sumacc} is an instance of the pattern in (6.42), which then gives us the corresponding iterative implementation, which in this case would be:

\[
\{ \ast \ \text{true} \ast \} \\
\ r, s := 0, \sigma \ ; \ \text{while} \ s \neq [ ] \ \text{do} \ x, s := \Delta \ x, \text{tail } s \\
\{ \ast \ r = \text{sumacc } (0, \sigma) \ast \}
\]

The general pattern above also specifies what the needed invariant for the loop, namely \(I : \text{sumacc} (r, s) = \text{sumacc} (0, \sigma)\).
6.9 Proof of the Distributivity of wp

This section shows the proofs that (under some conditions) wp 'distributes' over conjunction and disjunction of post-conditions. What this means is that we can split the calculation of the wp of \( Q_1 \land Q_2 \) or \( Q_1 \lor Q_2 \) as the post-conditions to calculation of the wp of \( Q_1 \) and \( Q_2 \) separately. We have used these results in e.g. the proof of the Fibonacci program.

**Theorem 6.9.1**: \( \wp \) Distributivity over \( \land \)

\[ \models \wp S (Q_1 \land Q_2) = (\wp S Q_1) \land (\wp S Q_2) \]

\[ \Box \]

**PROOF**

Since \( Q_1 \land Q_2 \Rightarrow Q_1 \), by Rule 6.2.8 we have:

\[ \wp S (Q_1 \land Q_2) \Rightarrow \wp S Q_1 \]

By a similar argument we have:

\[ \wp S (Q_1 \land Q_2) \Rightarrow \wp S Q_2 \]

So:

\[ \wp S (Q_1 \land Q_2) \Rightarrow (\wp S Q_1) \land (\wp S Q_2) \]

It remains to show that the implication in the reversed direction also holds. By Corollary 6.2.4 we have:

\[ \{ * \wp S Q_1 * \} S \{ * Q_1 * \} \]

\[ \{ * \wp S Q_2 * \} S \{ * Q_2 * \} \]

Now, by applying Hoare triple conjunction on the two above we obtain:

\[ \{ * (\wp S Q_1) \land (\wp S Q_2) * \} S \{ * Q_1 \land Q_2 * \} \]

So, the pre-condition above must imply \( \wp S (Q_1 \land Q_2) \).

\[ \Box \]

**Theorem 6.9.2**: \( \wp \) Distributivity over \( \lor \)

\[ \models \wp S (Q_1 \lor Q_2) = (\wp S Q_1) \lor (\wp S Q_2) \]

provided \( S \) is deterministic.

\[ \Box \]

**PROOF**

The full proof of this is more complicated. We will only show it for partial correctness. Note first that it is sufficient to show that for all state \( s \):

\[ s \in \wp S (Q_1 \lor Q_2) = s \in \wp S Q_1 \lor s \in \wp S Q_2 \]

We derive:

\[ s \in \wp S (Q_1 \lor Q_2) \]

\[ = \{ \text{Corollary 6.2.2} \} \]

\[ S s \subseteq Q_1 \lor Q_2 \]

\[ = \{ \text{in set interpretation } \lor \text{corresponds to set union} \} \]

\[ S s \subseteq Q_1 \cup Q_2 \]
Since $S$ is deterministic, $S$ contains exactly one state; let’s call it $t$. So, $S = \{ t \}$. We continue the derivation:

$$S s \subseteq Q_1 \cup Q_2$$

$$= \{ S s = \{ t \} \}$$

$$t \in Q_1 \cup Q_2$$

$$= \{ \text{set theory} \}$$

$$t \in Q_1 \lor t \in Q_2$$

$$= \{ S s = \{ t \} \}$$

$$S s \subseteq Q_1 \lor S s \subseteq Q_2$$

$$= \{ \text{Corollary 6.2.2} \}$$

$$s \in \text{wp} S Q_1 \lor s \in \text{wp} S Q_2$$

\[\square\]

### 6.10 Simultant Assignment

A simultant assignment is an assignment that simultaneously assigns values to multiple variables. We add it to uPL just for convenience. Here is an example:

$$x, y := 0, x+y$$

This is executed as follows. First all expressions at the right hand side are evaluated, then the results are assigned to the corresponding variables. Notice the difference with e.g. this:

$$x := 0; y := x+y$$

This will two assignments in sequence. First $0$ is evaluated then assigned to $x$, then $x + y$ is evaluated and then assigned to $y$. See the table below:

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 1, y = 2$</td>
<td>$x, y := 0, x + y$</td>
</tr>
<tr>
<td>$x = 1, y = 2$</td>
<td>$x := 0; y := x + y$</td>
</tr>
<tr>
<td>$x = 0, y = 3$</td>
<td>$x = 0, y = 2$</td>
</tr>
</tbody>
</table>

As said, simultant assignment is just for convenience. It can be implemented using ordinary assignments. For example, the assignment $x, y := 0, x+y$ can be implemented by:

```plaintext
{var @x, @y ; @x := x ; @y := y ;
 x := 0 ;
 y := @x + @y }
```

To calculate the \text{wp} of a simultant assignment we will need the corresponding notion of simultant substitution: the notation $Q[e_1/x, e_2/y]$ means the expression obtained by simultaneously replacing all free occurrences of $x$ and $y$ in $Q$ with respectively $e_1$ and $e_2$. In particular, you should not perform the substitutions in this order $Q[e_1/x][e_2/y]$, nor $Q[e_2/y][e_1/x]$. For example:

$$(x - y)[y/x, x + 1/y] = (y - (x + 1))$$

whereas:

$$(x - y)[y/x][x + 1/y] = (x + 1) - (x + 1)$$

$$(x - y)[x + 1/y][y/x] = y - (y + 1)$$

A simultant assignment implicitly requires that the target variables are all distinct. So, $x, x := 0, 1$ is illegal. Similar requirement applies to simultant substitution: $Q[0/x, 1/x]$ is illegal.
The Bar Notation

The notation \( \overline{e} \) (pronounced "e bar") denotes a list of expressions: \( e_1, \ldots, e_k \). The same notation is used to denote a list of variables.

Two lists of expressions or variables, \( \overline{d} \) and \( \overline{e} \), are compatible if they are of the same length and the type of \( d_i \) matches with the type of \( e_i \) for every \( 0 \leq i < \text{length of } \overline{d} \).

We will use the notation \( x := e \) to abbreviate \( x_0, \ldots, x_k := e_1, \ldots, e_k \), and \( P[ e/x ] \) to abbreviate \( P[ e_1/x_1, \ldots, e_k/x_k ] \).

**Theorem 6.10.1**: \( \wp ( \overline{\varphi} ) Q = Q[ \overline{\varphi} ] \)

\[ \square \]

6.11 Reducing Program’s Specification

The logic given so far actually only deals with statements. When given the specification of a full program, what we did in the previous examples was to implicitly reduce it to a slightly different specification in terms of the program’s body. Subsequently, we can handle it with the logic from the previous sections. So far we do the reduction from program to its body rather informally, but there are actually some technical details which you have to be aware of. Consider first this simple program without parameters:

\[
\text{trivial}() \{ \text{var } z; \; z := 0; \; \text{return } z \}
\]

You may expect that a specification like:

\[
\{ \ast P \ast \} \text{trivial()} \{ \ast Q \ast \}
\]

(6.45)

can be reduced to:

\[
\{ \ast P \ast \} \{ z := 0; \; \text{return } := z \} \{ \ast Q \ast \}
\]

(6.46)

That is, proving the latter is sufficient to prove the first. Although seemingly trivial, there is already one complication that you have to consider here. The reduction is not correct if either \( P \) or \( Q \) in (6.45) contains a free occurrence of \( z \). This \( z \) would refer to the \( z \) in the context of \( \text{trivial} \)'s caller, and not to \( \text{trivial} \)'s local \( z \). In (6.46) the \( z \) would refer to \( \text{trivial} \)'s local \( z \).

Fortunately we have required that a program’s specification must be closed (page 53). That is, it is not allowed to mention any variable other than the program’s own formal parameters or the auxiliary variables used to remember the parameters’ initial values. So, mentioning \( z \) in the pre- or post-condition in (6.45) is not allowed. Under this assumption (6.45) and (6.46) are equivalent.

Suppose we now have a program with parameters. Consider this specification:

\[
\{ \ast \text{true} \ast \}
\]

\[
\text{Y := y; test(x, OUT y) \{ x := x+y; y := 0; \text{return } x \}}
\]

(6.47)

\[
\{ \ast (\text{return } = x+Y) \land (y = 0) \ast \}
\]

Recall the agreement we made in page 53: \( x \) is a pass-by-value parameter; so \( x \) in the post-condition refers to its initial value rather than its final value just before the program \( \text{test} \) returns. Suppose we now naively translate the above specification to the one below, in terms of \( \text{test} \)'s body:

\[
\{ \ast \text{true} \ast \} \{ Y := y; \; x := x+y; \; y := 0; \; \text{return } := x \} \{ \ast (\text{return } = x+Y) \land (y = 0) \ast \}
\]
The \( x \) in the post-condition now has a very different interpretation: it now refers to its final value. So, the above translation is not correct. We should have replaced \( x \) with \( X \) as follows:

\[
\{ \ast \text{true} \ast \} \{ X, Y = x, y; x := x+y; y := 0; \text{return} := x \} \quad \{ \ast (\text{return} = X + Y) \land (y = 0) \ast \}
\]

which is equivalent to \((6.47)\). Formally, the reduction is captured by the following theorem.

The case when the program have more or less parameters and local variables can be handled analogously.

**Theorem 6.11.1: Program to Statement Reduction**

Let \( Pr \) be defined as below; \( x, y \) are its formal parameters; \( x \) is passed by-value and \( y \) by copy-restore; \( X, Y \) are auxiliary parameters used to record \( x, y \)'s initial values. Let this be a closed specification:

\[
\{ \ast P \ast \} \quad X, Y := x, y; \quad Pr(x, \text{OUT} \ y) \quad \{ \text{var} \ z; \ S; \ \text{return} := e \} \quad \{ \ast Q \ast \}
\]

It is valid if and only if the following statement is valid:

\[
\{ \ast P \ast \} \quad X, Y := x, y; \quad S; \ \text{return} := e \quad \{ \ast Q[X/x] \ast \}
\]

\[\square\]

### 6.12 Updating Compound Structures

Assignments to compound structures, in particular arrays, have a little complication. Our method to calculate the wp of an assignment (Theorem 6.2.10) is:

\[
\text{wp} \ (v := E) \ Q = Q[E/v]
\]

Recall that we have specifically require that \( v \) should be a program variable. So, it does not apply for an assignment like \( a[i] := 0 \). Let us see what will go wrong if we do that. Consider the post-condition \( a[i] = 0 \). We now calculate the wp simply by replacing occurrences of the the assignment’s left hand side in the post-condition with the assignment’s right hand side:

\[
\text{wp} \ (a[i] := 0) \ (a[i] = 0) \quad = \ { \text{assume we calculate the wp like this} } \quad (a[i] = 0)[0/a[i]]
\]

\[
\quad = \quad 0 = 0
\]

\[
\quad = \quad \text{true}
\]

This is as expected, since regardless the initial state \( a[i] := 0 \) will of course establish \( a[i] = 0 \) as the post-condition. Now consider a different post-condition:

\[
a[j] = 0
\]

We do the calculation again:

\[
\text{wp} \ (a[i] := 0) \ (a[j] = 0) \quad = \ { \text{assume we calculate the wp like this} } \quad (a[j] = 0)[0/a[i]]
\]

\[
\quad = \quad a[j] = 0
\]
CHAPTER 6. THE BASIC PROOF SYSTEM

However, this is not correct! The post-condition can also be realized if initially \( i = j \). Since above we have concluded that \( a[j] = 0 \) is the weakest pre-condition, then it must be the case that:

\[(i = j) \Rightarrow (a[j] = 0)\]

is valid. However, it is not.

What we have here is an aliasing problem at the array level. If \( i \) and \( j \) turns out to point to the same index in the array \( a \), then changing the value of \( a[i] \) will of course impact \( a[j] \). In the above derivation, the substitution \([0/a[i]]\) that we did essentially ignores this fact. Although we have said that, for simplicity, aliasing is not allowed in uPL (see page 5.1), we will still allow aliasing of array indices. Our logic will need a little extension though.

Let us introduce some notation first. Let \( a(i \text{ repby} e) \) denote a new array whose content is the same as that of array \( a \), except that at index \( i \) its value is \( e \). Formally this can be characterized by the following equality:

**Definition 6.12.1 : repby**
Let \( a \) be an array:

\[a(i \text{ repby} e)[j] = (i = j) \rightarrow e \mid a[j]\]

Notice that assignments \( a[i] := e \) and \( a := a(i \text{ repby} e) \) have the same effect. Of course the latter is very inefficient, but for the purpose of reasoning we equivalently convert \( a[i] := e \) to:

\[a := a(i \text{ repby} e)\]

Subsequently, we can calculate the wp in the usual way (as in Theorem 6.2.10). So:

**Theorem 6.12.2 : wp of Assignment to Array**

\[wp (a[i] := e) Q = Q[a(i \text{ repby} e)/a]\]

Notice that in effect the substitution:

\[Q[a(i \text{ repby} e)/a]\]

replaces not only \( a[i] \) but also every sub-expression of the form \( a[j] \) in \( Q \) with \( i = j \rightarrow e \mid a[j] \). The is done because \( j \) may point to the same element as \( i \) in the array \( a \). However, at this point we do not know this for sure. The formula \( i = j \rightarrow e \mid a[j] \) takes both possibilities into account.

If we later can prove that \( i = j \), hence they are aliases, then \( i = j \rightarrow e \mid a[j] \) can be reduced to \( e \). Similarly, if we can prove \( i \neq j \), so they are not aliases, we can reduce the formula to \( a[j] \). Here is an example:

\[wp (a[i] := 0) (a[j] = 0)\]

\[= \{ \text{Theorem 6.12.2} \} \]

\[(a(i \text{ repby} x))[j] = 0\]

\[= \{ \text{Definition 6.12.1} \} \]

\[(i = j \rightarrow 0 \mid a[j]) = 0\]

The last formula can be proven if the pre-condition implies either \( i = j \) or \( a[j] = 0 \).
6.12.1 Updating Record

Assignments to record fields are in some sense similar to assignments to array elements. We can see a record as an array, and its fields are the array’s indices. We will not get the aliasing problem with records however, because the field names of a record are static (they do not change during the execution). So if two field names \( fn_1 \) and \( fn_2 \) are distinct, they will refer to different fields.

Still, it is convenient to handle assignments to a record field in the same way as we handle assignments to an array element. We will extend the repby notation to records:

**Definition 6.12.3 : repby**

Let \( r \) be a record. Let \( fn_1 \) and \( fn_2 \) be distinct field names:

\[
\begin{align*}
    r(fn_1 \, repby \, e).fn_1 &= e \\
    r(fn_1 \, repby \, e).fn_2 &= r.fn_2
\end{align*}
\]

\( \Box \)

Logically, the effect of \( r.fn := e \) is the same as \( r := r(fn \, repby \, e) \). The wp of the latter is calculated as standard. Formally:

**Theorem 6.12.4 : wp of Assignment to Record**

\[
\begin{align*}
    wp (r.fn:=e) \, Q &= Q[r(fn \, repby \, e)/r] \\
    \Box
\end{align*}
\]

As an example consider the record type **person** defined as follows:

```
record person {key:int, name:string}
```

Suppose \( a \) is a variable of type **person**. The wp of the assignment \( a.key:=0 \) with respect to the post-condition \( a.key = b.key \) is calculated as follows:

\[
\begin{align*}
    wp (a.key := 0) & \, (a.key = b.key) \\
    &= \{ \text{Theorem 6.12.4} \} \\
    (a(key repby 0)).key &= b.key \\
    &= \{ \text{Definition 6.12.3} \} \\
    0 &= b.key
\end{align*}
\]

A post-condition that contains an expression on a record as a whole, e.g. a post-condition like \( a = b \), causes a little complication. For example:

\[
\begin{align*}
    wp (a.key := 0) & \, (a = b) \\
    &= \{ \text{Theorem 6.12.4} \} \\
    (a(key repby 0)) &= b
\end{align*}
\]

The resulting **repby** expression cannot be further simplified. However, we do know that two records \( r_1 \) and \( r_2 \) of type **person** are equal if and only if \( r_1.key = r_2.key \) and \( r_1.name = r_2.name \). With this information we can simplify further:

\[
\begin{align*}
    (a(key repby 0)) &= b \\
    &= (a(key repby 0)).key = b.key \wedge ((a(key repby 0)).name = b.name) \\
    &= \{ \text{Definition 6.12.3} \} \\
    (0 = b.key) \wedge (a.name = b.name)
\end{align*}
\]
6.13 Advanced Features

We have kept uPL simple because of its introductory purpose. Below, we will shortly discuss some more advanced features which are not covered in uPL. A good source for further reading is [3, 4, 19].

6.13.1 assert and assume

Many languages e.g. Java allow us to write a statement like \texttt{assert } P. It checks if \( P \) holds. If so, then everything is fine. It will just proceed to the next statement. If \( P \) does not hold it will throw an exception. So, it is just a syntactic sugar of

\[
\text{if } P \text{ then skip else throw } an\text{-exception}
\]

In practice people use \texttt{assert} to help them during testing, namely to express general properties that are expected to hold at certain points in the execution. But for verification it is also has another use. Let us define it such that it satisfies this property:

\begin{definition}
\text{wp of assert}
\end{definition}

\[
\text{wp (assert } P) Q = P \land Q \text{, if } Q \neq \text{true}
\]

The definition implies that to prove \( \{* O *\} \text{ assert } P \{* Q *\} \), it is sufficient to prove that the pre-condition \( O \) implies both \( P \) and \( Q \). So, the logic wants us to prove that we will not violate \( P \) in the first place. In theory this is incomplete. It may still be ok to violate \( P \) if you catch the exception elsewhere. However, this is not captured by the above definition. So, as said, the definition above is incomplete. But in return, we can use it without having to deal with its exception effect.

We can exploit \texttt{assert} to deal with e.g. normal arrays without having to change our logic. Unlike uPL arrays, normal arrays have finite size. For example, our inference rules for assignment do not apply on \( a[i] := 0 \) if \( i \) turns out to be a value outside the allowed range of \( a \)'s indices. However, we can convert the assignment to:

\[
\text{assert } 0 \leq i < a\text{.length } ; a[i] := 0
\]

This can be treated with our uPL logic as is, without any extension.

Interestingly, we can also introduce the dual of \texttt{assert}, namely \texttt{assume } P. As the name suggests, this assumes that \( P \) holds when the execution reaches the statement. If \( P \) does not hold, then we do not care anymore. Its semantic can be characterized as follows:

\begin{definition}
\text{wp of assume}
\end{definition}

\[
\text{wp (assume } P) Q = P \Rightarrow Q
\]

So, \( \{* O *\} \text{ assume } P \{* Q *\} \) is valid only if \( O \Rightarrow (P \Rightarrow Q) \). Notice that the implication is valid if \( O \land P \) implies \( Q \). But it is also valid if \( O \) implies \( \lnot P \), in which case it does not matter what \( Q \) is.

This statement \texttt{assume} is just a theoretical concept. It cannot be implemented, because \texttt{assume false} would then be a magic statement that can realize any post-condition. However, it is useful for inserting assumptions in our verification. Imagine we have a program that contains \( a[i] := 0 \) and \( a[z] := 0 \). For the first one we are not sure if \( i \) will be within the allowed range, so we want to verify that. This means that we then translate \( a[i] := 0 \) to the \texttt{assert}-construct as discussed above.
But suppose for the second one, \(a[z] := 0\), we are sure that \(z\) will be in the allowed range. Well, then we can just leave the assignment \(a[z] := 0\) as it is. However, we can also convert it to:

\[
\text{assume } 0 \leq z < a.\text{length} ; \ a[z] := 0
\]

This will now explicitly insert the assumption \(0 \leq z < a.\text{length}\). If at some point later you have another statement that exploits this assumption, then having it explicitly in the proof does matter.

### 6.13.2 Non-deterministic Assignment

A non-deterministic assignment is an assignment of this form: \(x, y \leftarrow P\). This will magically set the variables \(x\) and \(y\) to some values satisfying \(P\). Since there are typically multiple values that satisfy a given \(P\), this statement is non-deterministic. Its semantic can be characterized as follows:

**Definition 6.13.3**: ND Assignment

\[
\begin{align*}
\text{wp} \ (x \leftarrow P) \ Q &= (\forall x :: P \Rightarrow Q)
\end{align*}
\]

Such an assignment is not in general implementable, but can be useful to be used as an equivalent abstraction to replace a concrete statement. Such an abstraction can be useful for verification.

For example, consider a statement \(S\). Its write frame is the set of variables it writes. Suppose that this is just \(\{x, y\}\), and suppose we know that \(S\) satisfies \(\{* P *\} S \{* Q *\}\). Then for the purpose of verification we can just replace the entire occurrence(s) of \(S\) in the program by the statement:

\[
\text{assert } P; \ x, y \leftarrow Q
\]

### 6.13.3 Non-deterministic Choice

Let \(S \sqcup T\) be a statement that non-deterministically chooses to either execute \(S\) or \(T\). It can be defined as follows:

**Definition 6.13.4**: ND Choice

\[
\begin{align*}
\text{wp} \ (S \sqcup T) \ Q &= (\text{wp } S \ Q) \land (\text{wp } T \ Q)
\end{align*}
\]

This can be used to define non-deterministic if-then-else and non-deterministic loop, as shown below. This also means that we get their inference rules as well.

\[
\begin{align*}
\text{if } g_1 & \rightarrow S_1 \ \ \ \ g_2 \rightarrow S_2 \\
\text{while } g_1 & \rightarrow S_1 \ \ \ \ g_2 \rightarrow S_2
\end{align*}
\]

= \text{assert } g_1 \lor g_2 ; ((\text{assume } g_1 ; S_1) \Box (\text{assume } g_2 ; S_2))

\[
\begin{align*}
\text{while } g_1 & \rightarrow S_1 \ \ \ \ g_2 \rightarrow S_2
\end{align*}
\]

= \text{while } g_1 \lor g_2 \text{ do } ((\text{assume } g_1 ; S_1) \Box (\text{assume } g_2 ; S_2))

### 6.13.4 goto

Our uPL logic is nicely syntax driven. That is, it constructs a proof by following the syntactical structure of the given program. This relies on the assumption that a program’s control flow follows its syntactical structure. For example, in a statement \(S_1; S_2\) the program will really proceed with \(S_2\) after \(S_1\). A syntax driven logic can be straightforwardly automated. A goto statement as in BASIC allows a jump to an arbitrary control location in a program. If goto is allowed, the above assumption no longer holds and the entire uPL logic collapses. A more powerful logic is needed, e.g. a proof-outline [9] based logic. Unfortunately, such a logic is no longer syntax driven.
6.13.5  Inner block

So far we only introduce local variables at the beginning of a program’s body. In general, we can have an inner statement block that declares its own local variables. Here is how you can calculate the \text{wp} of such a block:

**Theorem 6.13.5 : wp of Block**

\[
\text{wp}\{\text{var } x; S\} Q = (\forall x' :: (\text{wp } S (Q[@x/x]))[x'/x])[x/@x]
\]

where @x and x' are fresh variables.

The first substitution \(Q[@x/x]\) replaces \(x\) in \(Q\) with a fresh variable. This is only temporary. The last substitution \([x/@x]\) will replace @x back with \(x\). Recall that this \(x\) does not refer to the locally declared \(x\) in the block \(\{\text{var } x; S\}\). We (temporarily) replace \(x\) with @x to prevent it from being affected by assignments to the local \(x\) in \(S\).

So far, we have not said anything about the initial value of a variable when it is declared. In e.g. Java newly declared variables will assume pre-defined default values. To make it simple, we will assume no such mechanism in uPL. So, when declared a variable may be in any state.

Now, \(\text{wp } S (Q[@x/x])\) gives the weakest pre-condition just before \(S\) starts. However, since initially the \(x\) declared in \(\text{var } x\) can take any value, the weakest pre-condition just before the block is entered is \((\forall x :: \text{wp } S (Q[@x/x]))\). To allow us to restore @x back to \(x\) we rename this quantification so that it quantifies over \(x'\). This explains the second substitution \([x'/x]\).

If we add a syntax to immediately initialize a local variable at its declaration, the formula for calculating \(\text{wp}\) can be simplified:

**Theorem 6.13.6 : wp of Block with Initialization**

\[
\text{wp}\{\text{var } x = e; S\} Q = (\text{wp } (x:=e; S) (Q[@x/x]))[x/@x]
\]

where @x is a fresh variable.

\[\square\]

6.13.6  Expression with side effect

Expression like \(x++\) in C has a side effect. Allowing program call inside an expression may also cause a side effect. The logic of uPL will incorrectly calculate the \(\text{wp}\) of, for example, \(y := x++\), as it assumes that only \(y\) is affected. We can however transform such an expression to an equivalent code that is side effect free. For example, \(y := x++\) is logically equivalent to \(y := x; x := x + 1\).

Once translated to a side effect free statement, we can handle it with the ordinary uPL logic. To write such a translation we have to know the precise semantics of expressions. For many real imperative languages, e.g. C, it can be quite tricky —see for example [22].

6.13.7  Exception

Exception is a form of \texttt{goto} jump. However, the jump made by exception is never cyclic. This makes a big difference. A more powerful logic is still needed, but we can still keep it syntax driven, e.g. as in [19, 5].

Let’s add this statement: \texttt{raise} throws an exception. To make it simple, let us just one sort of exception, and it does not carry any data as exceptions in e.g. Java do. We also add a \texttt{try S catch T} construct. It executes \(S\); if it throws an exception the control will jump to the handler \(T\).

We extend Hoare triples to look like this:

\[
\{* P *\} S \{* Q, R *\}
\]
Q specifies the expected final state of S if it terminates normally, whereas R specifies it if S terminates by exception. So, Q corresponds to the normal post-condition that we have before. We will have to fix our inference rules; we will show some of them below.

1. A skip does not any side effect, nor does it throw any exception. So:

   \[
   \emptyset \vdash \text{skip} \vdash \{ \ast \}
   \]

Notice that any exceptional post-condition R is good, because skip will not terminates exceptionally anyway.

2. If an assignment never throws an exception, then its inference rule is simply:

   \[
   \{ \ast \} \ x := e \ \vdash \{ \ast \}
   \]

Like in skip, the exceptional post-condition does not matter.

However, what if the assignment can throw an exception, e.g. \(x := x/y\) if \(y\) turns out to be 0? A solution is to first convert this to:

\[
\text{if } y = 0 \ \text{then throw else } x = x/y
\]

This can be handled by our logic.

3. The only effect of throw is to throw an exception. So, if R is the required exceptional post-condition, this has to hold as well before the throw:

   \[
   \emptyset \vdash \text{throw} \vdash \{ \ast \}
   \]

4. The rule for catch is somewhat complicated:

   \[
   \{ \ast \} \ S \ \vdash \{ \ast, Q', R \}
   \]

Note that if S terminates normally, then T is not executed. So, S must realize the normal post-condition Q. This is captured by the first premise. However, S may also terminate exceptionally. Suppose when this happens the state satisfies some intermediate predicate Q'. The execution then proceeds to the exception handler T. So, from this pre-condition Q' T will have to realize the specified goal \{ \ast, Q, R \}. This is reflected in by the second premise.

5. And the rule for sequential composition:

   \[
   \{ \ast \} \ S \ \vdash \{ \ast, Q', R \}
   \]

Notice that the rule looks very similar to that of try – catch. This is because in a way their behavior is also similar. In both, the execution of T is conditional. In try S catch T the control continues to T if S terminates exceptionally, and else not. In S; T is this the other way around. So, it is not so surprising the their inference rules look the same, except for some inversions, namely the position of the intermediate Q'.
6.13.8 Alias

If aliasing is allowed, an assignment to $x$ will also affects all its aliases. The logic of uPL will again incorrectly calculates the wp of $x := e$ with respect to a post-condition $Q$, because it assumes that the assignment only affects $x$. A more powerful way of calculating wp is needed. See for example [17].
New Vocabulary

alias; deterministic; EC; exceptional post-condition; Hoare triples disjunction/conjunction; IC; IL/ILF; invariant; loop reduction; non-deterministic; non-deterministic assignment; non-deterministic choice; post-condition weakening; pre-condition strengthening; proof plan; repby; simultaneous assignment; state; state: initial, terminal; tail invariant; TC1/TC2; termination metric; weakest pre-condition; wp


6.14 Exercise

1. Let \( P \) and \( Q \) be some predicates. What do the following specifications say if we interpret them with the partial correctness interpretation? What do they say under total correctness? What kind of statements \( S \) satisfy each of them?

   (a) \{ \ast P \ast \} \ S \ \{ \ast \text{true} \ast \}
   (b) \{ \ast P \ast \} \ S \ \{ \ast \text{false} \ast \}
   (c) \{ \ast \text{false} \ast \} \ S \ \{ \ast Q \ast \},

Prove your conclusions.

2. A statement \( S \) 'always' terminates if it terminates from any state. Express the property "\( S \) always terminates" using a Hoare triple. Prove that if \( S; T \) always terminates, so does \( S \). Does it imply that \( T \) always terminate?

3. In partial correctness, the characterization of \( \text{wp} \) we give in Definition 6.2.3 is actually a theorem. It follows from Definition 6.2.1 of \( \text{wp} \) and Definition 6.1.4 of Hoare triples in partial correctness. Prove this.

4. We want to add a new statement to \( \text{uPL} \). If \( S_2 \) and \( S_2 \) are statements, \( S_1 \sqcup S_2 \) is a statement that non-deterministically chooses \( S_1 \) or \( S_2 \) and then executes the choice.

   (a) Define the meaning of this statement in terms of models discussed in Section 6.1 (so this would be a definition that assumes termination).
   (b) Give a formula for calculating its \( \text{wp} \).
   (c) Prove that your formula for calculating its \( \text{wp} \) is correct with respect to your definition. That is, that you can show that it follows from your definition above.

5. Let \( \text{even} \ x \) and \( \text{odd} \ x \) mean that \( x \) is an even respectively odd integer. Prove that the validity of this specification:

   \{ \ast \text{odd} \ x \land \text{odd} \ y \ast \}
   y := y+1 ; \text{if} \ x>y \ \text{then} \ x:=x-y \ \text{else} \ \text{skip}
   \{ \ast \text{odd} \ (x+y) \ast \}

6. Let \( \text{cntEven} \ a \ i = \text{COUNT} [x|x \ \text{from} \ a[0..i], \ \text{even} \ x] \). Prove this:

   \{ \ast \ 0 \leq \ i \land (c = \text{cntEven} \ a \ i) \ast \}
   \text{if even} \ (a[i]) \ \text{then} \ c:=c+1 \ \text{else} \ \text{skip} ;
   \ i:=i+1
   \{ \ast c = \text{cntEven} \ a \ i \ast \}

7. If two programs satisfy exactly the same set of Hoare triple specifications then they will behave the same way and thus implement the same functionality. We can say that they are equivalent (but keep in mind that we do not pay attention to aspects like performance in Hoare triple specifications). Prove that the following two statements are equivalent in the above sense:
6.14. EXERCISE

(a) \( \text{if } g \land h \text{ then } S \text{ else } T \)
(b) \( \text{if } g \text{ then } \{ \text{if } h \text{ then } S \text{ else } T \} \text{ else } T \)

8. Give a sketch of a proof showing the validity of each of the specifications below (with total correctness interpretation). Come up with a minimalistic invariant.

(a) \( \{ \ast \text{ true } \ast \} \text{ while } i > 0 \text{ do } i := i - 1 \} \{ \ast \text{ true } \ast \}
(b) \{ \ast \text{ reds } = 100 \land \text{ blues } = 100 \ast \}

\text{while reds } > 0 \lor \text{ blues } > 0 \text{ do } \{
\quad \text{if reds } > 0 \text{ then } \{ \text{ reds } := \text{ reds } - 1 ; \text{ blues } := \text{ blues } + 2 \} \}
\quad \text{else blues } := \text{ blues } - 1
\} \{ \ast \text{ true } \ast \}

(c) \{ \ast x = 10 \ast \}

\text{while } 0 < x \text{ do }
\quad \text{if } x = 10 \text{ then } x := x + 1 \text{ else } x := x - 2
\} \{ \ast x = -1 \ast \}

9. We want to extend uPL with a for-loop:

\text{for var } := e_1 \text{ to } e_2 \text{ step } d \text{ do } S

This statement initializes var to \( e_1 \) and repeatedly executes \( S \) while \( var < e_2 \). At the end of every iteration the value of \( var \) is increased by \( d \).

(a) Give an inference rule to prove the correctness of a for-loop.
   \text{Hint: express the for-loop in terms of while-loop.}
(b) Arguing the termination of a for-loop is often easier. Give a variant of your inference rule whose conditions are easier to prove.
   \text{Motivate the correctness of your rule.}

10. We want to extend the while-loop in uPL to allow multiple bodies, which can be chosen non-deterministically:

\text{while } g_1 \text{ do } S_1
\quad \mid \ldots
\quad \mid g_n \text{ do } S_n

This loop will iterate if there is at least one guard which is true. If multiple guards are true, one will be chosen non-deterministically, and the corresponding body will be executed. The loop terminates when all its guards are false.

Extend the inference rule for while so that it can deal with this non-deterministic version of while.

Consider the program below:

\text{while } i > j \text{ do } i := i - 1
\quad \mid i > j \text{ do } j := j + 1

If initially \( i \geq j \), the program will terminate with \( i = j \). Give an invariant and a termination metric that can prove this. Motivate your answer.

11. Consider the specification of the program test below. Is the specification open or closed? Use the Program to Statement Reduction rule (page 101) to transform it to an equivalent specification in terms of its body.
CHAPTER 6. THE BASIC PROOF SYSTEM

\texttt\{\texttt\{* true \}\

\texttt\{test (b:[] bool, i:int) : bool 
\{ var r:bool ;
\quad r:=true ;
\quad while r \land i>0 do \{ i:=i-1; r:=b[i] \} ;
\quad return \neg r
\}
\texttt\{\texttt\{* return = (\exists j: 0\leq j<i: \neg b[j]) *\}\}

12. Prove the following specification:

\texttt\{\texttt\{* 0\leq n *\}\}

\texttt\{i:=0 ; c:=0 ;
while i < n do
\{ if x[i]=0 then c:=c+1 else skip ;
\quad i:=i+1 \}
\texttt\{\texttt\{* c = COUNT [i | i from [0...n], x[i] = 0] *\}\}

13. Section 6.5 discusses the correctness of a program to compute Fibonacci numbers. It discussed an incremental approach to construct an invariant. Section 6.6 sketches a variant of this approach. The approach is less intelligent, but on the other hand it can be automated. Redo the Fibonacci example, this time use the latter strategy to construct your invariant.

14. We want a program \texttt\{SUMPOWER\} with the following specification:

\texttt\{\texttt\{* n \geq 0 *\}\}

\texttt\{SUMPOWER(x,n:int) : int
\texttt\{\texttt\{* return = \sum [x^i | i from [0...n]] *\}\}

Write \texttt\{SUMPOWER\}. You can avoid using the exponentiation operator, and thus getting a much faster program. Prove the correctness of your program.

15. Prove the following specification:

\texttt\{\texttt\{* n > 1 *\}\}

\texttt\{i:=0 ; s:=0 ;
while i < n do \{ s:=s+i; i:=i+1 \}
\texttt\{\texttt\{* s = n*(n-1) div 2 *\}\}

The program simply sums all integers from 0 up to \texttt\{n-1\}. So, if the specification above is valid in a sense it also proves that \texttt\{\texttt\{SUM [0...n] = n*(n-1) div 2\}\}.
16. Below is an algorithm to efficiently multiply two integers. It is a very old algorithm, written by an Egyptian A’h-mose, back in 1700 BC! It is the same algorithm used in CPU. The $\ast 2$ and $\div 2$ operations used in the algorithm can be done efficiently by, essentially, shifting the bit strings representing the target integer. Specify the algorithm, and prove its correctness. The version below only works on non-negative $x$.

```plaintext
MUL (x,y:int) : int
{ var z ;
  z:=0 ;
  while x ≠ 0 do
    if odd x then { z:=z+y ; x:=x-1 }
    else { y:=y*2 ; x := x div 2 } ;
  return z
}
```

17. Let the record type `Item` is defined by:

```plaintext
record Item = {key:int, name:string}
```

Calculate the weakest pre-condition of the statements below with respect to the given post-conditions; $r$ and $s$ are of the `Item` type.

(a) $j:=a[i]$; $a[a[i]]:=a[i]$ ; $a[i] = 0$ { $a[j] = 0$ }

(b) $a[j]:=0$; $r$.key := $a[i]$ { $(r$.key = 0 ) $\lor$ $(r$.item = 0 ) }

(c) $r$.key := 0; $s$.item := 0 { $r = s$ }

18. The program below swaps the contents of $a[i]$ and $a[j]$. Specify it, and prove its correctness.

```plaintext
SWAP (a:[]int, i,j:int) : ()
{ var temp:int ;
  temp := a[i] ;
  a[i] := a[j] ;
  a[j] := temp }
```

19. Write a program to initialize an integer array by setting all elements in the domain $0 \leq i < n$ to 0. Give a sufficient invariant; motivate your choice and prove its invariance.

20. Give a translation scheme so that we can handle assignments like:

```
a[e_1].fname := e_2 \quad \text{or} \quad r.fname[e_1] := e_2
```

Your translation scheme should translate these assignments to ordinary assignments that target variables. Can your scheme handle an assignment targeting an arbitrarily nested arrays and records?

21. The program below sorts an array. It is not the fastest algorithm. Its worst case run time is linear to $n^2$, where $n$ is the assumed size of the array, compared to $n \ast \log n$ of some other algorithms. However, its algorithm is very simple.

The outer loop gradually increases the region where the array is already sorted (elements with indices less than $i$). The inner loop keep swapping elements in the unsorted region (elements with indices at least $i$) in a certain direction, so that in the end a smallest element of the unsorted region can be found at the index $i$; then the counter $i$ is advanced by 1. The algorithm is also known as bubble sort (from the way the inner loop 'bubbles' smaller elements to the 'top' of the array).
BUBBLESORT (a:[int], n:int) : ()
{ var i,j:int ;
  i:=0 ;
  while i<n do
  { j:=n-1 ;
    while i<j do
    { j:=j-1
      if a[j+1]<a[j]
      then { tmp := a[j] ;
          a[j] := a[j+1] ;
          a[j+1] := tmp }
      else skip
    } ;
    i:=i+1 } }

Write a specification for the program. Give invariants that would prove its correctness; motivate your choice (are they sufficient? are they really invariant?)

22. Here is another sorting program. It takes an array a and implements an algorithm called counting sort. It is simple but very fast: linear to a's size! However, it requires that the range of a's elements to be known and that this range is narrow. An array with the latter property typically contains many duplicates.

For simplicity, we will assume a's elements to range from 0 to max. The program is also given two help arrays. The array count has been prepared such that count[x] is equal to the number of elements of a whose values are less than x. This array can be built in linear time.

The array b contains the sorted elements. We still have to copy it back to a, but we leave this step out from the program. It can be done in linear time too.

COUNTSORT (a,b,count:[], n,max:int) : ()
{ var i,x:int ;
  i:=0 ;
  while i<n do
  { x:=a[i] ;
    b[count[x]] := x ;
    count[x]:=count[x]+1 ;
    i:=i+1 } }

Write a specification for the program; to make it simple it is sufficient if your specification implies that upon termination b will be sorted. Give an invariant that would prove the program's correctness and motivate your choice.
6.15 Solution

1. (a) \( \{\ast P \ast\} S \{\ast \text{true} \ast\} \)
   In partial correctness the above specification is always valid. In this interpretation, \( S \) is assumed to terminate. Intuitively we can argue that when a statement terminates, it will terminate in some state, and since any state satisfies \( \text{true} \), the above specification is thus valid.

In terms of the model in Section 6.1 it can be argued as follows. Recall that in this model \( S \) is viewed as a relation on states. The above specification is valid if \( S s \subseteq \text{true} \) for any state \( s \in P \). However, since \( \text{true} \) represents the set of all states, it will also include \( S s \).

In any case, \( \text{true} \) as a post-condition does not give any useful information about the state upon termination. However under total correctness the above specification does say that \( S \) terminates, which is a useful fact to know.

(b) \( \{\ast P \ast\} S \{\ast \text{false} \ast\} \)
   If \( P \) itself is non-empty (it is not \( \text{false} \)), no real program can establish \( \text{false} \) as the post-condition, because no (end) state can satisfy \( \text{false} \). Another way to look at it: if such an \( S \) exists, it would be a magical program that can establish any post-condition from \( P \) (since \( \text{false} \) implies any post-condition).

(c) \( \{\ast \text{false} \ast\} S \{\ast Q \ast\} \)
   This is a trivial specification which is always valid (for any statement and any \( Q \)). The fact may be theoretically interesting, but is practically useless, since no initial state satisfies \( \text{false} \).

   The proof is simple. By Definition 6.2.3 it is equivalent to \( \text{false} \Rightarrow \wp S Q \), which is trivially valid.

2. (a) "\( S \) always terminates", in the sense that it terminates on all possible initial state, can be expressed by \( \{\ast \text{true} \ast\} S \{\ast \text{true} \ast\} \) in total correctness.

(b) PROOF

   \[ \begin{align*}
   & [A:] \{\ast \text{true} \ast\} S; T \{\ast \text{true} \ast\} \\
   & [G:] \{\ast \text{true} \ast\} S \{\ast \text{true} \ast\} \\
   \end{align*} \]

   \[ \begin{align*}
   & 1 \{ \text{rewrite A with Definition 6.2.3} \} \quad \text{true} \Rightarrow \wp (S; T) \text{true} \\
   & 2 \{ \text{rewrite 1 with wp of sequence} \} \quad \text{true} \Rightarrow \wp S (\wp T \text{true}) \\
   & 3 \{ \text{rewrite 2 with Definition 6.2.3} \} \quad \{\ast \text{true} \ast\} S \{\ast \wp T \text{true} \ast\} \\
   & 4 \{ \text{post condition weakening on 3} \} \quad \{\ast \text{true} \ast\} S \{\ast \text{true} \ast\} \\
   \end{align*} \]

3. We have to prove in partial correctness Definition 6.2.3 is actually a theorem:

   EQUATIONAL PROOF

   \[ \begin{align*}
   P & \Rightarrow \wp S Q \\
   = & \{ \Rightarrow \text{corresponds to } \subseteq \text{ in set interpretation} \} \\
   P & \subseteq \wp S Q \\
   = & \{ \text{set theory} \} \\
   (\forall s : s \in P : s \in \wp S Q) & \Rightarrow \{ \text{Corollary 6.2.2} \} \\
   (\forall s : s \in P : S s \subseteq Q) & \Rightarrow \\
   \end{align*} \]
4. (a) Define: \((S_1 \sqcup S_2) s = S_1 s \cup S_2 s\).

(b) To calculate \(wp\): \(wp (S_1 \sqcup S_2) \ Q = (wp \ S_1 Q) \land (wp \ S_2 Q)\)

(c) Proof of (b):

\[
\text{EQUATIONAL PROOF}
\]
\[
s \in wp (S_1 \sqcup S_2) \ Q = \{ \text{Corollary 6.2.2} \} (S_1 \sqcup S_2) s \subseteq Q = \{ \text{def. of } \sqcup \text{ above} \} S_1 s \cup S_2 s \subseteq Q = \{ \text{set theory} \} S_1 s \subseteq Q \land S_1 s \subseteq Q = \{ \text{Corollary 6.2.2} \} s \in \wp S_1 Q \land s \in \wp S_2 Q = \{ \text{set theory} \} s \in (wp S_1 Q) \cap (wp S_2 Q) = \{ \text{in set interpretation } \cap \text{ corresponds to } \land \} s \in (wp S_1 Q) \land (wp S_2 Q)
\]

END

5. It is sufficient to show that the pre-condition implies the \(wp\) with respect to the specified post-condition.

\[
\text{PROOF main [A:]} \ \text{odd x} \land \text{odd y}
\]

\[
[D: ] \ Q = \ wp (y := y + 1; \text{if } x > y \text{ then } x := x - y \text{ else skip}) (\text{odd } (x+y))
\]

\[
[G: ] \ Q
\]

BEGIN

1 \{ \ wp calculation \} \quad Q = x > y + 1 \rightarrow \text{odd } (x - (y + 1) + y + 1) | \text{odd } (x + y + 1)

2 \{ \ follows from A \} \quad \text{even } (x+y)

3 \{ \ follows from 2 \} \quad \text{odd } (x+y+1)

4 \{ \ see the equational proof below \} \quad Q = \text{true}

\[
\text{EQUATIONAL PROOF}
\]
\[
Q = \{ \text{main.1} \} x > y + 1 \rightarrow \text{odd } (x - (y + 1) + y + 1) | \text{odd } (x + y + 1) = \{ \text{simplication} \} x > y + 1 \rightarrow \text{odd } x | \text{odd } (x + y + 1) = \{ \text{main.A says odd } x \} x > y + 1 \rightarrow \text{true} | \text{odd } (x+y+1) = \{ \text{rewrite with main.3} \} x > y + 1 \rightarrow \text{true} | \text{true} = \{ \text{Theorem A.4.4} \} \text{true}
\]

true

END

5 \{ \ True Consequence rule on 4 (Rule A.1.4) \} \quad Q

END
6.15. SOLUTION

6. PROOF main

[A1:] 0 ≤ i

[A2:] c = \text{cntEven a i}

[D1:] Q = wp
  (\text{if even}(a[i]) \text{ then } c := c + 1 \text{ else skip; } i := i + 1)
  (c = \text{cntEven a i})

[G :] Q

BEGIN

1 { wp calculation } 
Q = \text{even}(a[i]) → c + 1 = \text{cntEven a}(i + 1) | c = \text{cntEven a}(i + 1)

2 { justified by CaseA below }
\text{even}(a[i]) ⇒ (c + 1 = \text{cntEven a}(i + 1))

EQUATIONAL PROOF CaseA

[A:] \text{even}(a[i])
  \text{cntEven a}(i + 1)
  = \{ \text{def. cntEven} \}
  \text{COUNT}[x | x \text{ from } a[0...i + 1], \text{even } x]
  = \{ \text{List +1 Split, justified by main.A1} \}
  \text{COUNT}[x | x \text{ from } (a[0...i] ++ [a[i]]), \text{even } x]
  = \{ \text{Distributivity of comprehension} \}
  \text{COUNT}(\text{x | x from a[0...i], even x})
  ++
  \text{COUNT}[x | x \text{ from } [a[i]], \text{even } x]
  = \{ \text{homomorphism of COUNT} \}
  \text{COUNT}[x | x \text{ from } a[0...i], \text{even } x]
  +
  \text{COUNT}[x | x \text{ from } [a[i]], \text{even } x]
  = \{ \text{def. cntEven} \}
  \text{cntEven a i + \text{COUNT}[x | x \text{ from } [a[i]], \text{even } x]}
  = \{ \text{comprehension over singleton} \}
  \text{cntEven a i + \text{COUNT}(\text{even}(a[i]) → [a[i]] | [])}
  = \{ \text{assumption A} \}
  \text{cntEven a i + \text{COUNT}(\text{true → } [a[i]] | [])}
  = \{ \text{COND conversion} \}
  \text{cntEven a i + \text{COUNT}[a[i]]}
  = \{ \text{Main.A2 and and def. COUNT} \}
  c + 1

END

3 { justified by CaseB } 
\neg(\text{even}(a[i])) ⇒ (c = \text{cntEven a}(i + 1))

... CaseB goes in much the same way as CaseA

4 { COND Split Rule on 2 and 3 }
\text{even}(a[i]) → c + 1 = \text{cntEven a}(i + 1) | c = \text{cntEven a}(i + 1)

5 { rewrite 4 with 1 } Q

END

7. It is sufficient to show that the wp of the two statements are equivalent. The proof below uses the following variant of Case Split rule, which you can prove yourself:

\[ P_1 \land P_2 \quad \Rightarrow \quad Q \]
\[ P_1 \land \neg P_2 \quad \Rightarrow \quad Q \]
\[ \neg P_1 \land P_2 \quad \Rightarrow \quad Q \]
\[ \neg P_1 \land \neg P_2 \quad \Rightarrow \quad Q \]
\[ Q \]
The proof:

\[
\text{PROOF main}
\]

\[
[D1:] \quad A = \wp (if \ g \ then \ \{ \ if \ h \ then \ S \ else \ T \} \ else \ T) \ Q
\]

\[
[D2:] \quad B = \wp (if \ g \land h \ then \ S \ else \ T) \ Q
\]

\[
[G :] \quad A = B
\]

BEGIN

1 \{ rewrite D1 with def. \wp \} \quad A = g \to (h \to \wp S \ Q \ | \ wp T \ Q) \ | \ wp T \ Q

2 \{ rewrite D2 with def. \wp \} \quad B = g \land h \to \wp S \ Q \ | \ wp T \ Q

3 \{ justified by CaseA \} \quad g \land h \Rightarrow (A = B)

\text{EQUATIONAL PROOF CaseA}

\[
[A:] \quad g \land h
\]

\begin{align*}
A &= \{ \text{main.1} \} \\
g &\to (h \to \wp S \ Q \ | \ wp T \ Q) \ | \ wp T \ Q \\
&= \{ \text{assumption A} \} \\
\text{true} &\to (T \to \wp S \ Q \ | \ wp T \ Q) \ | \ wp T \ Q \\
&= \{ \text{COND conversion} \} \\
\wp S \ Q \\
&= \{ \text{COND conversion} \} \\
\text{true} &\to \wp S \ Q \ | \ wp T \ Q \\
&= \{ \text{assumption A} \} \\
g \land h &\to \wp S \ Q \ | \ wp T \ Q \\
&= \{ \text{main.2} \} \\
B
\end{align*}

END

4 \{ justified by CaseB \} \quad g \land \neg h \Rightarrow (A = B)

\text{PROOF CaseB \ldots \ quite similar to CaseA}

5 \{ justified by CaseC \} \quad \neg g \land h \Rightarrow (A = B)

\text{PROOF CaseC \ldots \ quite similar to CaseA}

6 \{ justified by CaseD \} \quad \neg g \land \neg h \Rightarrow (A = B)

\text{PROOF CaseD \ldots \ quite similar to CaseA}

7 \{ Case Split on 3,4,5,6 \} \quad A = B

END

8. (a) For the specification:

\{ *true* \} \ while \ i>0 \ do \ i:=i-1 \ { *true* \}

Essentially, the specification only says that the loop terminates on any initial state. With respect to the post-condition true, taking true itself as the invariant will do. Init and PIC becomes trivial with this invariant.

As for the termination part, i is an obvious choice for the termination metric. Its value obviously decreases in each iteration. The loop’s guard already implies that it is bounded below by 0. These two facts can be proven without knowing what I is, hence true as I will not pose a problem.
(b) Since the post-condition is just true, using true as invariant will do. For termination metric we can use \( m \) defined as 3 * reds + blues. However, just the guard of loop does not imply that this \( m \) would be >0. To prove this we need to include reds \( \geq 0 \) \& blues \( \geq 0 \) in the invariant.

(c) Use \( x=10 \rightarrow 12 \mid x \geq -1 \) as the termination metric and \( x \geq -1 \) as a part of the invariant. These will do to prove that the program terminates. The post-condition is however more specific: it demands that the program terminates with \( x=-1 \). You will need to strengthen the invariant to get this; I will leave this to you.

9. (a) The for-loop:

\[
\text{for } i:=e_1 \text{ to } e_2 \text{ step } d \text{ do } S
\]

Can be simulated by:

\[
i:=e_1; \text{ while } i < e_2 \text{ do } \{S; i:=i + d\}
\]

You can obtain a rule for for-loop by adjusting the rule for while, according to the above translation:

\[
\begin{array}{l}
\{\ast P \ast\} \ i:=e_1\ \\{\ast I \ast\} \\
\{\ast I \land i<e_2 \ast\} \ S; \ i:=i + d \ \{\ast I \ast\} \\
\models I \land i<e_2 \Rightarrow m > 0 \\
\{\ast I \land i<e_2 \ast\} \ C:=m; \ S; \ i:=i + d \ \{\ast m < C \ast\} \\
\models I \land i<e_2 \Rightarrow Q \\
\{\ast P \ast\} \ \text{for } i:=e_1 \text{ to } e_2 \text{ step } d \text{ do } S \ \{\ast Q \ast\}
\end{array}
\]

(b) We can obtain a specialized variant of the rule above where you get termination almost for free.

If \( S \) does not modify \( i \) and any variable in \( e_1, e_2, \) and \( d \), and if \( i \) does not appear in \( e_2 \), then \( d > 0 \) is a sufficient condition for the termination of the loop. This implies namely that each iteration decreases \( e_2 - i \). The guard \( i<e_2 \) implies that \( e_2 - i > 0 \). So, we have our termination metric.

The inference rule can thus be simpler:

\[
\begin{array}{l}
\{\ast P \ast\} \ i:=e_1\ \\{\ast I \ast\} \\
\{\ast I \land i<e_2 \ast\} \ S; \ i:=i + d \ \{\ast I \ast\} \\
\models I \Rightarrow d > 0 \\
I \land i\geq e_2 \Rightarrow Q \\
\{\ast P \ast\} \ \text{for } i:=e_1 \text{ to } e_2 \text{ step } d \text{ do } S \ \{\ast Q \ast\}
\end{array}
\]

10. We will show the rule for while with only two non-deterministic guards (and the corresponding bodies). You can generalize it yourself for loops with \( N \)-guards.

\[
\begin{array}{l}
\text{TC1 – 1 : } \{\ast I \land g_1 \ast\} \ (C := m; S_1) \ \{\ast m < C \ast\} \\
\text{TC1 – 2 : } \{\ast I \land g_2 \ast\} \ (C := m; S_2) \ \{\ast m < C \ast\} \\
\text{TC2 : } \models I \land (g_1 \lor g_2) \Rightarrow m > 0 \\
\text{EC : } \models I \land \neg g_1 \land \neg g_2 \Rightarrow Q \\
\text{IC – 1 : } \{\ast I \land g_1 \ast\} \ S_1 \ \{\ast I \ast\} \\
\text{IC – 1 : } \{\ast I \land g_2 \ast\} \ S_2 \ \{\ast I \ast\} \\
\{\ast I \ast\} \ \text{while } g_1 \text{ do } S_1 \\
\{\ast Q \ast\} \ | \ g_2 \text{ do } S_2
\end{array}
\]

In the TC1’s \( c \) should be a fresh variable.

For the example program, take \( 1 \geq j \) as the invariant, and \( 1 - j \) as the termination metric.
11. It is a closed specification. Expressed as an equivalent specification of its body:

\{
\begin{align*}
&\text{* true * } \\
i0:=i ; \\
r:=\text{true} ; \\
\text{while } r \land i > 0 \text{ do } \{ i:=i-1 ; r:=b[i] \} ; \\
\text{return } : = \neg r \\
\end{align*}
\}

\{\text{\text{return } = (\exists j: 0 \leq i0 < \neg b[j])}\}

where \(i0\) is introduced as an auxiliary variable.

12. We first extend the specification:

\{
\begin{align*}
&\text{* 0 \leq n , see PROOF Pinit * } \\
i:=0 ; c:=0 ; \\
\end{align*}
\}

\{\* I , see PROOF PTC1, PTC2, PEC, PIC. Termination metric: n-i *\}

while \(i < n\) do

\{ if \(x[i]=0\) then \(c:=c+1\) else skip ; \}

\{ \(i:=i+1\) \} 

\{ \* \(c = \text{cnt x n}\) *\}

\text{ASSUMING}

\begin{align*}
&I = (c = \text{cnt x i}) \land 0 \leq i \leq n \\
&\text{cnt x k} = \text{COUNT} [j \mid j \text{ from } [0\ldots k], \ x[j] = 0]
\end{align*}

The abbreviation \(\text{cnt}\) is introduced for convenience. Now the proofs:

\text{PROOF PEC}

\begin{align*}
&\text{[A1:] } I \\
&\text{[A2:] } \neg (i < n) \\
&\text{[G :] } c = \text{cnt x n}
\end{align*}

\text{BEGIN}

\begin{align*}
1 & \{ \text{A1 and def. of } I \} \quad i \leq n \\
2 & \{ \text{follows from 1 and A2} \} \quad i = n \\
3 & \{ \text{follows from A1 and def. I} \} \quad c = \text{cnt x i} \\
4 & \{ \text{rewrite 3 with 2} \} \quad c = \text{cnt x n}
\end{align*}

\text{END}

In PTC2 you must prove \(I \land i < n \Rightarrow n - i > 0\). This is of course trivial!

\text{PROOF PTC1}

\begin{align*}
&\text{[A1:] } I \\
&\text{[A2:] } i < n \\
&\text{[G :] } \text{wp (C := } n - i; \text{ loop's body) (n - i < C) }
\end{align*}

\text{BEGIN}
1 \{ see the equational proof below \}
G = \text{true}

EQUATIONAL PROOF

G
= wp (C := n - i; \text{loop's body}) (n - i < C)
= \{ wp calculation \}
x[i] = 0 \rightarrow n - (i + 1) < n - i \mid n - (i + 1) < n - i
= \{ \text{COND conversion} \}
n - (i + 1) < n - i
= \{ \text{the above is always true} \}
true

END

2 \{ \text{True Consequence (Rule A.1.4) on 1} \}
G

PROOF PinvA

[G:] wp \text{loop's body} (c = cnt x i)
BEGIN

1 \{ justified by the proof below \}
wp \text{loop's body} (c = cnt x i)

PROOF PIC

[A1:] I
[A2:] i < n
[G :] wp \text{loop's body} I
BEGIN

1 \{ justified by the proof below \}
wp \text{loop's body} (c = cnt x i)

EQUATIONAL PROOF

[A:] x[i] = 0
cnt x (i + 1)
= \{ \text{def. cnt} \}
COUNT [j \mid j from [0...i+1], x[j] = 0]
= \{ \text{Enumeration Split, justified by } 0 \leq i, \text{ which is implied by I in PIC.A1} \}
COUNT [j \mid j from [0...i+1], x[j] = 0]
= \{ \text{Comprehension Split and Singleton Comprehension} \}
COUNT ([j \mid j from [0...i], x[j] = 0] + COUNT (x[i] = 0 \rightarrow [x[i]])]
= \{ \text{I in PIC.A1 says the left expression is } c \}
c + COUNT (x[i] = 0 \rightarrow [x[i]])]
= \{ \text{the assumption A} \}
c + COUNT (true \rightarrow [x[i]])]
= \{ \text{COND Conversion} \}
c + COUNT [x[i]]
= \{ \text{definition of COUNT} \}
c + 1
3 \{ the proof is quite similar to above; left out for you \}
\[ x[i] \neq 0 \Rightarrow (c = \text{cnt} \times (i + 1)) \]
4 \{ COND Split Rule on 2 and 3 \}
\[ x[i] = 0 \rightarrow (c + 1 = \text{cnt} \times (i + 1)) \] \( \land \) \( (c = \text{cnt} \times (i + 1)) \)
5 \{ rewrite 4 with 1 \} \quad G

2 \{ prove this yourself \} \quad \text{wp loop’s body} 0 \leq i < n
3 \{ use wp Distributivity on the conjunction of 1 and 2 \}
\text{wp loop’s body} ((c = \text{cnt} \times i) \land 0 \leq i < n)
4 \{ definition of I \} \quad \text{wp loop’s body} I

PROOF Pinit
[A:] 0 \leq n
[G:] \text{wp} (i := 0; c := 0) I

BEGIN

1 \{ wp calculation \} \quad G = (0 = \text{cnt} \times 0) \land 0 \leq 0 < n
2 \{ see the equational proof below \} \quad 0 = \text{cnt} \times 0

\text{cnt} \times 0
\quad = \{ \text{definition cnt} \}
\quad \text{COUNT} [j | j \text{from} [0\ldots 0], x[j] = 0]
\quad = \{ \text{Empty Enumeration} \}
\quad \text{COUNT} [j | j \text{from} [], x[j] = 0]
\quad = \{ \text{Empty Comprehension} \}
\quad \text{COUNT} []
\quad = \{ \text{def. of COUNT} \}
\quad 0

3 \{ follows from A \} \quad 0 \leq 0 < n
4 \{ rewrite the conjunction of 2 and 3 with 1 \} \quad G

END

13. Just a hint: applying the strategy you will end up with this invariant:

\[ (\text{nNow} = \text{fib} \ t) \land (\text{nNow} + \text{nBefore} = \text{fib} (t + 1)) \land 0 < t \leq n \]

14. A possible solution is by computing \( x^2 \) at each iteration, and accumulates this in some variable. We can do better, as shown below. It avoids the use of exponentiation.
The choice of the termination metric is obvious: \( n - i \). We also introduce this abbreviation:

\[
\text{sum } x \ n = \text{SUM } x^j \mid j \text{ from } [0...n])
\]

so that we can write the post-condition above as \( \text{return } = \text{sum } x \ n \). We take this as invariant:

\[
(r = \text{sum } x \ i) \land (y = x^i) \land 0 \leq i \leq n
\]

The extended specification:

\[
\{ \ast \ n \geq 0 \ \ast \}, \text{see PROOF Init }\ast
\]

\[
i := 0 ; \ y := 1 ; \ r := 0;
\]

\[
\{ \ast \ I \ \ast \}, \text{see PROOF PTC1, PTC2, PEC, PIC. Termination metric: } n-i \ast
\]

\[
\text{while } i < n \ \text{do}
\]

\[
\{ \ast \ r = \text{sum } x \ n \ \ast \}, \text{justified by wp calculation}
\]

\[
\text{return } r
\]

\[
\{ \ast \text{return } = \text{sum } x \ n \ \ast \}
\]

**ASSUMING**

\[
I = (r = \text{sum } x \ i) \land (y = x^i) \land 0 \leq i \leq n
\]

Now the proofs:

**PROOF PEC**

[A1:] I

[A2:] \( \neg(i < n) \)

[G :] \( r = \text{sum } x \ n \)

BEGIN

1 \{ follows from A1 and def. I \} \ i \leq n

2 \{ follows from A1 and def. I \} \ r = \text{sum } x \ i
PTC1 and PTC2 are quite trivial and are not shown.
In PIC we will exploit the distributivity of \( \text{wp} \) so that we can prove each conjunct of \( I \) in the post-condition separately.

**PROOF PIC**

\[ \begin{align*}
\text{[A1:]} & \quad I \\
\text{[A2:]} & \quad i < n \\
\text{[D1:]} & \quad J_1 = (r = \text{sum } x \cdot i) \\
\text{[D2:]} & \quad J_2 = (y = x^i) \\
\text{[D3:]} & \quad J_3 = (0 \leq i \leq n) \\
\text{[G :]} & \quad \text{wp loop's body} I \\
\end{align*} \]

**BEGIN**

\[ \begin{align*}
1 & \quad \{ \text{wp calculation} \} \quad \text{wp loop's body } J_1 = (r + y = \text{sum } (i + 1)) \\
2 & \quad \{ \text{see the proof below} \} \quad r + y = \text{sum } (i + 1) \\
\end{align*} \]

**EQUATIONAL PROOF**

\[ \begin{align*}
\text{sum } x \cdot (i + 1) \\
& = \{ \text{def. } \text{sum} \} \\
\text{SUM } [x^i \mid 0 \ldots i + 1] \\
& = \{ \text{Enumeration Split, justified by } 0 \leq i, \text{which is implied by PIC.A1 and def. of } I \} \\
\text{SUM } [x^i \mid 0 \ldots i] + [x^i] \\
& = \{ \text{Comprehension Split} \} \\
\text{SUM } ([x^i \mid 0 \ldots i] + [x^i]) \\
& = \{ \text{Singleton Comprehension} \} \\
\text{SUM } ([x^i \mid 0 \ldots i] + [x^i]) \\
& = \{ \text{Homomorphism of SUM} \} \\
\text{SUM } x^i \cdot [0 \ldots i] + \text{SUM } x^i \\
& = \{ \text{def. sum and SUM} \} \\
\text{sum } x^i + x^i \\
& = \{ I \text{ in PIC.A1 says } x^i = y \} \\
r + y \\
\end{align*} \]

**END**

\[ \begin{align*}
3 & \quad \{ \text{rewrite 2 with 1} \} \quad \text{wp loop's body } J_1 \\
4 & \quad \{ \text{prove this yourself} \} \quad \text{wp loop's body } J_2 \\
5 & \quad \{ \text{prove this yourself} \} \quad \text{wp loop's body } J_3 \\
6 & \quad \{ \text{use wp Distributivity on the conjunction of 3,4,5} \} \\
\text{wp loop's body } (J_1 \wedge J_2 \wedge J_3) \\
7 & \quad \{ \text{D1, D2, D3, def. of } I \} \quad \text{wp loop's body } I \\
\end{align*} \]

**END**

**PROOF Init**

\[ \begin{align*}
\text{[A:]} & \quad n \geq 0 \\
\text{[G:]} & \quad \text{wp } (i := 0; y := 1;r := 0) I \\
\text{BEGIN} \\
\end{align*} \]
6.15. **SOLUTION**

1. \{ see the proof below \} \quad G = n \geq 0

EQUATIONAL PROOF

\[
\begin{align*}
G &= \{ \text{def. of } G \} \\
\wp (i := 0; y := 1; r := 0) I &= \{ \text{wp calculation} \} \\
(0 = \text{sum } x 0) \land (1 = x^0) \land 0 \leq 0 \leq n &= \{ \text{simplification} \} \\
(0 = \text{sum } x 0) \land 0 \leq n &= \{ \text{def. } \text{sum} \} \\
(0 = \text{SUM}[x^j \mid j \text{from } 0 \ldots 0]) \land 0 \leq n &= \{ \text{Empty Enumeration} \} \\
(0 = \text{SUM}[x^j \mid j \text{from } ]]) \land 0 \leq n &= \{ \text{Empty Comprehension} \} \\
0 \leq n &= \{ \text{def. } \text{SUM, simplification} \} \\
\END
\end{align*}
\]

2. \{ rewrite A with 1 \} \quad G

END

15. We will only give hints and proof sketch. The choice for the termination metric is obvious: 
\( n - i \). Use this as invariant:

\[
(s = i \ast (i - 1) \div 2) \land 0 \leq i \leq n
\]

Proving termination, initialization, and exit condition would be quite easy. \textbf{PIC} is more complicated, due to the use of the \texttt{div} operator. After calculating \( \wp \), we obtain:

\[
(s + i = (i + 1) \ast i \div 2) \land 0 \leq i + 1 \leq n
\]

We have to prove that it follows from \( I \land i < n \). The second conjuct is easy. The first one is more difficult. We can split the proof in two cases: if \( i \) is even, and otherwise. For the first case, assume \( i \) is even. It follows that \( i = 2k \), for some \( k \). We can now derive:

\[
\begin{align*}
(i + 1) \ast i \div 2 &= \{ i = 2k \} \\
(i + 1) \ast 2k \div 2 &= \{ \text{the div of an even integer} \} \\
(i + 1) \ast k &= \{ \text{simple arithmetics} \} \\
(i - 1) \ast k + 2k &= \{ \text{the div of an even integer} \} \\
((i - 1) \ast 2k \div 2) + 2k &= \{ i = 2k \} \\
((i - 1) \ast 2k \div 2) + i &= \{ I \} \\
s + i
\end{align*}
\]
The case when \( i \) is odd is left to you.

16. A specification for `MUL` is below. It is expressed in terms of `MUL`’s body; \( X \) and \( Y \) below are auxiliary variables.

\[
\{ \ast 0 \leq x \ast \} \\
X:=x; \ Y:=y; \\
z:=0; \\
\text{while } x \neq 0 \text{ do} \\
\quad \text{if } \text{odd } \ x \ \text{then} \ \{ \ z:=z+y; \ x:=x-1 \} \\
\qquad \text{else} \ \{ \ y:=y*2; \ x:=x \div 2 \} ; \\
\{ * z = XY * \}
\]

You can use \( x \) as the termination metric. (Tail) invariant:

\[
(z + xy = XY) \land 0 \leq x
\]

Notice that \( xy = 0 \) when the loop terminates, hence the invariant would imply \( z = XY \) that we want.

17. (a) \[ \text{wp } (j := a[i]; \ a[a[i]] := a[i]; \ a[i] = 0) \ (a[j] = 0) \]
\[ = \{ \text{wp of array assignment } \} \\
\text{wp } (j := a[i]; \ a[a[i]] := a[i]) \ ((a(i \ repby 0))[j] = 0) \]
\[ = \{ \text{def. repby } \} \\
\text{wp } (j := a[i]) \ ((i = j \rightarrow 0 \land a[i] = 0) \land (j = a[i] \rightarrow a[i] \land a[i])) = 0 \]
\[ = \{ \text{wp of assignment } \} \\
(i = a[i] \rightarrow 0 \land (a[i] = a[i] \rightarrow a[i] \land a[a[i]])) = 0 \]
\[ = \{ \text{prove this yourself } \} \\
(i = a[i]) \lor (a[i] = 0)
\]

(b) \[ \text{wp } (a[j] := 0; \ r.key := a[i]) \ ((r.key = 0) \lor (r.item = 0)) \]
\[ = \{ \text{wp of array assignment, then use def. repby } \} \\
\text{wp } (a[j] := 0) \ ((a[i] = 0) \lor (r.item = 0)) \]
\[ = \{ \text{wp of array assignment, then use def. repby } \} \\
((i = j \rightarrow 0 \land a[i] = 0) \lor (r.item = 0)) \]
\[ = \{ \text{after some simplification } \} \\
(i = j) \lor (a[i] = 0) \lor (r.item = 0)
\]

(c) You can either first expand equalities on records occurring in the post-condition, like \( r = s \) to \( (r.key = s.key) \land (r.item = s.item) \), before you apply the `wp` rule for an assignment to a record field, or you can do the record equality expansion after. The first method is easier, we leave it to you. The second method is shown below:

\[ \text{wp } (r.key := 0; \ s.item := 0) \ (r = s) \]
\[ = \{ \text{wp of record assignment } \} \\
\text{wp } (r.key := 0; \ s.item := 0) \ (r = s(item repby 0)) \]
\[ = \{ \text{wp of record assignment } \} \\
r(key repby 0) = s(item repby 0) \]
\[ = \{ \text{expand record equality } \} \\
(r(key repby 0).key = s(item repby 0).key) \land \\
(r(key repby 0).item = s(item repby 0).item) \]
\[ = \{ \text{def. of repby } \} \\
(0 = s.key) \land (r.item = 0) \]
18. Here is a seemingly reasonable specification:

\[
\{ \text{true} \} \\
X:=a[i]; \ Y:=a[j]; \ \text{SWAP}(a,i,j) \\
\{ \text{true} \} \\
\]

The specification specifies the most important aspect of the program, but it is not complete: we cannot infer from it that it does not mess with other elements of \(a\). Whether this incompleteness is acceptable depends on how you use the specification in practice. For the purpose of verifying \(\text{SWAP}\) itself it is quite adequate; you can argue that the program quite obviously does not tamper with elements other than \(a[i]\) and \(a[j]\). If you are going to use the program as a black box, then the specification above is not adequate. You would need a more complete specification, for example:

\[
\{ \text{true} \} \\
A:=a; \ \text{SWAP}(a,i,j) \\
\{ \text{true} \} \\
\]

We will only show the proof of the first specification. We first convert the specification to express it in terms of \(\text{SWAP}'s body:

\[
\{ \text{true} \} \\
X:=a[i]; \ Y:=a[j]; \ \text{temp}:=a[i]; \ a[i]:=a[j]; \ a[j]:=\text{temp} \\
\{ \text{true} \} \\
\]

Let us now calculate the wp:

\[
\begin{align*}
\text{wp} \ (X &:= a[i]; \ Y := a[j]; \ \text{temp} := a[i]; \ a[i] := a[j]; \ a[j] := \text{temp}) \\
&= \{ \ \text{wp of array assignment, def. repby} \} \\
\text{wp} \ (X &:= a[i]; \ Y := a[j]; \ \text{temp} := a[i]; \ a[i] := a[j]) \\
&= \{ \ \text{wp of array assignment, def. repby} \} \\
\text{wp} \ (X &:= a[i]; \ Y := a[j]) \\
&= \{ \ \text{wp assignment} \} \\
\text{wp} \ (X &:= a[i]; \ Y := a[j]) \\
&= \{ \ \text{wp assignment, of the remaining assignments} \} \\
&= \text{true}
\end{align*}
\]

So, we obtain true as the weakest pre-condition, which is obviously implied by the given pre-condition.
19. This will do:

\[
\{ \ast \; 0 \leq n \; \ast \}
\]

\[i := 0 \; ; \; \text{while } i \neq n \; \text{do } \{ \; a[i] := 0; \; i := i + 1 \; \}\]

\[
\{ \ast \; \text{zero a n} \; \ast \}
\]

**ASSUMING**

\[
\text{zero a n} \; = \; (\forall k : 0 \leq k < n : \; a[k] = 0)
\]

This invariant will do:

\[
I \; = \; \text{zero a i} \; \land \; 0 \leq i \leq n
\]

We are not going to give the complete PIC, but only its most important part. We have to prove:

\[
\wp (a[i] := 0; \; i := i + 1) \; (\text{zero a i} \; \land \; 0 \leq i \leq n)
\]

We can prove this for each conjunct separately. The second conjunct is left out for you. For the first conjuct, we first calculate the \(\wp:\)

\[
\wp (a[i] := 0; \; i := i + 1) \; (\text{zero a i})
\]

\[
= \{ \; \wp \; \text{the assignments} \; \}
\]

\[
\text{zero (a(i repby 0)) (i + 1)}
\]

\[
= \{ \; \text{def. zero} \; \}
\]

\[
(\forall k : 0 \leq k < i + 1 : (a(i \; \text{repby} \; 0))[k] = 0)
\]

\[
= \{ \; \text{def. repby} \; \}
\]

\[
(\forall k : 0 \leq k < i + 1 : (i \rightarrow k : 0 \; | \; a[k]) = 0)
\]

\[
= \{ \; \text{domain merging, justified by } 0 \leq i, \; \text{followed by domain split} \; \}
\]

\[
(\forall k : 0 \leq k < i : (i \rightarrow k : 0 \; | \; a[k]) = 0)
\]

\[
\land
\]

\[
(\forall k : k = i : (i \rightarrow k : 0 \; | \; a[k]) = 0)
\]

\[
= \{ \; \text{quantification over singleton} \; \}
\]

\[
(\forall k : 0 \leq k < i : (i \rightarrow k : 0 \; | \; a[k]) = 0)
\]

\[
\land
\]

\[
((i \rightarrow 0 \; | \; a[k]) = 0)
\]

\[
= \{ \; \text{simplification} \; \}
\]

\[
(\forall k : 0 \leq k < i : (i \rightarrow k : 0 \; | \; a[k]) = 0)
\]

\[
= \{ \; k < i, \; \text{so } i \neq k, \; \text{simplification} \; \}
\]

\[
(\forall k : 0 \leq k < i : a[k] = 0)
\]

\[
= \{ \; \text{def. zero} \; \}
\]

\[
\text{zero a i}
\]

The last expression is implied by \(I\).

20. Basically, we can just repeat the translation scheme given in Section 6.12. More precisely, consider this assignment:

\[
e := f
\]
If \( e \) is a variable then we are done. If \( e \) has the form of \( d.fname \) then we translate it to:
\[
d := d(fname \text{ repby } f)
\]
Similarly, if \( e \) has the form of \( d[i] \) we translate it to:
\[
d := d(i \text{ repby } f)
\]
We repeat this translation procedure on \( d \) until we have a variable as the assignment target. This scheme can handle arbitrarily nested arrays or records.

For example, the scheme would translate \( a[e_1].fname := e_2 \) to:
\[
a := e(e_1 \text{ repby } (a[e_1](fname \text{ repby } e_2)))
\]
The resulting assignment can be handled by the standard \( wp \) rule for assignment.

21. Here is a specification, expressed in terms of \textsc{Bubblesort}'s body. The post-condition states that upon termination the array will be sorted. Motivated by the informal explanation on how the program works, additional assertions have been added. One is needed to specify the goal of the inner loop. The other two specify the invariants of the corresponding loops.

\[
\{ \ast 0 \leq n \ast \}
\]

\[
i := 0 ;
\]

\[
\{ \ast \text{sorted } a \ i \ \land \ 0 \leq i \leq n \ast \}
\]

\[
\text{while } i < n \text{ do}
\]
\[
\{ \ast \text{sorted } a \ i \ \land \ 0 \leq i < n \ \land \ i \leq j < n \ \land \ a[j] = \text{MIN}(a[i...n]) \ast \}
\]

\[
\text{while } i < j \text{ do}
\]
\[
\{ j := j - 1
\]
\[
\text{if } a[j+1] < a[j]
\]
\[
\text{then } \{ \text{tmp} := a[j] ; \}
\]
\[
a[j] := a[j+1] ;
\]
\[
a[j+1] := \text{tmp} \}
\]
\[
\text{else skip}
\]
\[
\}
\]

\[
\{ \ast \text{sorted } a \ i \ \land \ 0 \leq i < n \ \land \ a[i] = \text{MIN}(a[i...n]) \ast \}
\]
\[
i := i + 1
\]
\[
\{ \ast \text{sorted } a \ n \ast \}
\]

\textbf{ASSUMING}
\[
\text{sorted } a \ i \ = \ (\forall p, q : 0 \leq p, q < i : p \leq q \Rightarrow a[p] \leq a[q])
\]

The additional assertions seem sufficient, but unfortunately they are not. In particular, we will not be able to re-establish the invariant of the outer loop. Essentially, because the additional information provided by the post-condition of the inner loop, saying that \( a[i] = \text{MIN}(a[i...n]) \), is not sufficient to infer that \( a[0...i+1] \) is sorted.
What is missed is the fact that all elements in the unsorted region are greater (or equal) than the elements in the sorted region. So, the minimum of the unsorted region is also greater or equal to all elements in \(a[0...i]\), and hence we can infer that \(a[0...i+1]\) is sorted. We can express the missed fact as follows:

\[
\text{unsorted } a\ i = (\forall p, q : 0 \leq p < i \land i \leq q < n : a[p] \leq a[q])
\]

\(\text{unsorted } a\ i\) has to be added to the post-condition of the inner loop, and to its invariant.

**22.** The assumption about \(\text{count}\) can be formally captured by \(\text{config count}\) defined by:

\[
\text{config count} = \\
(\forall p : 0 \leq p \leq \text{max} : \text{count}[p] = \text{COUNT}[1 | x \text{ from } a[0...n], x < p])
\land
(\text{count}[0] = 0) \land (\text{count}[\text{max}] = n)
\]

Let \(\text{sort}\) be defined by:

\[
\text{sorted } b\ i\ j = (\forall p, q : i \leq p, q < j : p \leq q \Rightarrow b[p] \leq b[q])
\]

We will specify the post-condition by \(\text{sort } b\ 0\ n\). Below we give a specification, given in terms of \(\text{COUNTSORT}'s body. \ C\) is introduced as an auxilliary variable.

\[
\{* 0 \leq n *\} \land \text{config count}
\]

\[
C := \text{count} ;
\]

\[
i := 0 ;
\]

\[
\{* \text{segSorted } b\ C\ \text{count} \land \ 0 \leq i \leq n *\}
\]

while \(i < n\) do

\[
x := a[i] ;
\]

\[
b[\text{count}[x]] := x ;
\]

\[
\text{count}[x] := \text{count}[x]+1 ;
\]

\[
i := i+1
\]

\[
\{* \text{sorted } b\ 0\ n *\}
\]

The invariant \(\text{segSorted } b\ C\ \text{count}\) still has to be defined. Notice that \(C\) freezes the initial value of \(\text{count}\). The program loops over the array \(a\). It picks its element one by one, and sort each element element \(x\) in a segment in \(b\). From the pre-condition we know \(x\) occurs exactly \(C[x+1] - C[x]\) times in \(a\). Moreover, there are \(C[x]\) elements which are less than \(x\). So, we can store \(x\) somewhere in a still free place in \(b[C[x]...C[x+1]]\). The program fills this segment from left to right, using \(\text{count}[x]\) as a counter pointing to the next free place in the segment. We conclude that the loop maintains each segment up to each \(\text{count}[x]\) sorted. More precisely:

\[
\text{segSorted } b\ C\ \text{count} = \\
(\forall p : 0 \leq p \leq \text{max} : \text{sorted } b\ (C[p]) (\text{count}[p]) \land C[p] \leq \text{count } p \leq C[p+1])
\]

However, this is not sufficient: it only implies that each segment is sorted. In the end we need to infer that \(b\) as a whole is sorted. We also need to specify that the segments are
themselves sorted. If \( s \) and \( t \) are two lists, let \( s < t \) means that every element in \( s \) is less than all elements in \( t \). We should strengthen the definition of \( \text{seqSorted} \) above with:

\[
b[C[p]...count[p]) < b[C[p+1]...count[p+1])
\]

for every \( p \) in \( 0 \leq p < \text{max} \). Furthermore, we need to be able to infer that when the loop terminates (\( i = n \)) the array \( b \) is completely filled. This will do:

\[
\sum[C[p+1] - count[p] | p \text{ from } [0...\text{max}]) = n - i
\]

which should be added to the invariant. When \( i = n \), the sum is 0. From the definition \( \text{segSorted} \) we can infer that each \( C[p+1] - count[p] \) must be non-negative. Because the whole sum is 0, it follows that \( C[p+1] - count[p] \) is also 0. Hence, every segment \( b[C[x]...C[x+1]) \) is completely filled.
A uPL program is comparable to what in other languages are called procedure or function. In particular, a uPL program can be called from another program. A possible way for our logic to handle a program call like \( P(e) \) is to replace the call with a proper instantiation of \( P \)'s body. Subsequently, we can handle it with the logic over ordinary statements from Chapter 6. This does not work if \( P \) is recursive. Moreover, this assumes we have \( P \)'s source code. This is not always the case. Even if we do have the source code, if you recall the discussion about black box in Section 5.2, we may prefer not to rely on it. We will take the black box approach; however, in general it is also more complicated. Let us first show you the case where it is simple. Consider a program \( SUC \) with this specification:

\[
\{\text{true}\} \text{SUC}(x: \text{int}) \{\text{return }= x + 1\}
\]

The program has no side effect; moreover, the specification \textit{fully} specifies what the return value is. Such a specification is said to imply a 'functional behavior'. It allows a nice black box treatment. For example, a call like:

\[
x := \text{SUC}(x)
\]

logically can be treated as:

\[
x := x + 1
\]

which is just a simple assignment; we already know how to handle it. Note that we do not need to know the code of \texttt{SUC} except for its header.

Things are more difficult if we are given a specification that is only partial, such as:

\[
\{\text{true}\} \text{MORE}(x: \text{int}) \{\text{return }> x\}
\]
A call like \( x := \text{MORE}(x) \) cannot be treated as an assignment \( x := e \), because now we cannot infer what \( e \) is from the specification.

Before we proceed, let us first list down some restrictions imposed in order to simplify the logic of program call. Most of them have been discussed before. Keep in mind that these restrictions may be too stringent in practice (hence you would need to find a stronger logic).

### 7.1 Restrictions

#### Syntax of program call

In uPL a program can only be called as a statement, and the syntax is:

\[
\begin{align*}
\text{target} & \ := \ \text{program-name} \ (\text{parameters}) \\
\text{or} & \ \\
\text{program-name} \ (\text{parameters})
\end{align*}
\]

The first will copy the return value to the variable targeted by the assignment; the second will simply ignore the return value. In particular, uPL does not allow a program to be called from an expression. So, if \( P \) is a program, this is not allowed:

\[
x := P(y) + x \times y
\]

If we allow this, the logic will have to take \( P(y) \)'s side effect, and the exact order in which \( P(y) \) within the expression \( P(y) + x \times y \) is executed (is it before or after \( x \times y \)?) into account.

#### Program’s parameters

A parameter can be passed either by value or by copy-restore to a program. A uPL program cannot access a global variable, unless it is passed as a parameter to the program. See also Section 5.1. Parameters named in the definition of a program are usually called formal parameters. For example, in:

\[
\text{SUC}(x:\text{int}) \ {\{ \text{return } x+1 \}}
\]

\( x \) is a formal parameter of \( \text{SUC} \). The parameters which are actually supplied to a program when it is called are called actual parameters. For example, in the call \( \text{SUC}(x+1) \), \( x+1 \) is an actual parameter of \( \text{SUC} \).

If a program has multiple copy-restore parameters, they are restored in the same order as they are listed as the program’s arguments.

#### Alias is excluded

Two variables with distinct names, e.g. \( x \) and \( y \) are aliases of each other if they refer to the same location in the memory. So, changing the content of one will implicitly affect the other. In uPL we cannot have aliases.

Aliasing complicates reasoning, and causes further complication in treating program call. To show you, consider the following program, where \( x \) and \( y \) are passed by reference. Passing parameters by reference is not possible in uPL, but suppose for a moment that it is possible.

\[
\text{TF}(\text{REF } x, y : \text{bool}) \ {\{ x := \text{true}; y := \text{false} \}}
\]  

(7.1)

It may seem reasonable to expect the program to satisfy this specification:

\[
\{* \text{true }*\} \ \text{TF}(x, y) \ {* x \land \neg y *}
\]  

(7.2)

It is not valid however. It is satisfied by most calls, but not if we pass \( x \) twice as in \( \text{TF}(x, x) \). This call would pass the same address in both parameters to \( \text{TF} \). In other words, aliases are passed
to TF. The post-condition would then imply that the final value of x is equal to both true and false, which is a contradiction.

Specification (7.2) is valid if we e.g. strengthen the pre-condition by requiring that x and y are not aliases. But first the programmer will have to be aware of the more subtle situation arising from passing aliases. In practice this is quite easy to overlook; after all x and y are two different names, thus we may be misled into thinking that they are really 'different'.

Anyway, as said in uPL this kind of subtleties will be kept away, since do not have pass-by-reference parameters nor aliases.

7.2 Formal Treatment of Program Call

A call replaces formal parameters with actual parameters. You may expect that specifications can also be inferred in the same way. For example, consider:

\[
\{ \ast \text{true} \ast \} \ x := x; \ \text{INC(out } x \text{ : int)} \ \{ \ast x = x + 1 \ast \} \quad (7.3)
\]

From the above we can infer, simply by replacing x with a:

\[
\{ \ast \text{true} \ast \} \ a := a; \ \text{INC(a)} \ \{ \ast a = a + 1 \ast \}
\]

which at least give us some information to reason about the call INC(a). Another way to look at it is to see a specification like (7.3) as if it is implicitly $\forall$-quantified over all its formal parameters and auxiliary variables. Hence, we can infer a new specification simply by renaming the formal parameters and auxiliary variables. This is formally captured by the following rule.

Rule 7.2.1 : Renaming

Let $\pi$, $\gamma$, $X$ and $Y$ be lists of distinct variables. The latter two are auxiliary variables.

\[
\{ \ast P \ast \} \ X := \pi; \ P_r(\pi) \ \{ \ast Q \ast \}
\]

\[
\{ \ast P[\gamma/\pi] \ast \} \ Y := \gamma; \ P_r(\gamma) \ \{ \ast Q[\gamma/\pi, Y/X] \ast \}
\]

Renaming (Rule 7.2.1) seems to be the natural way to handle program call. However, on its own it is not sufficient. Consider the call INC(a[i]). You may now expect, by replacing the formal parameter x in the INC's specification (7.3) with the actual parameter a[i], to infer:

\[
\{ \ast \text{true} \ast \} \ x := a[i]; \ \text{INC(a[i])} \ \{ \ast a[i] = x + 1 \ast \}
\]

However, Rule 7.2.1 does not actually allow us to infer it. The Rule specifically requires that you can only rename formal parameters with variables. Instantiating the parameter x with a complex expression like a[i] is not covered by the Rule. One may propose to generalize the rule. For the above example it is not a problem. However, in general the rule will be incorrect. To show you the problem, consider the following variation of the program TF you saw before:

\[
\text{ZeroOne(out } x, y \text{ : int)} \ \{ \ x := 0; \ y := 1; \}
\]

The program satisfies this specification:

\[
\{ \ast \text{true} \ast \} \ \text{ZeroOne(out } x, y \text{ : int)} \ \{ \ast (x = 0) \land (y = 1) \ast \}
\]

If Rule 7.2.1 allows formal parameters to be instantiated with arbitrary expressions, it would allow this to be inferred:

\[
\{ \ast \text{true} \ast \} \ \text{ZeroOne(a[i], 1)} \ \{ \ast (a[i] = 0) \land (i = 1) \ast \}
\]
However, this is not valid! Suppose \( i \) is initially 0. As the call exits, the used calling convention of uPL would first restore \( a[0] \), hence setting it to 0, and then restores \( i \), which would set it to 1. There is nothing we can infer about the value of \( a[1] \), and yet the specification above claims that \( a[1] = 0 \).

To get around the above problem, we will first transform a program call to an equivalent code where only variables are passed as actual paremeters. Subsequently, we can safely use Rule 7.2.1. The transformation is only done for our reasoning purpose. In particular, a compiler producing an executable code does not do it, because the transformed code is longer. Subsection 7.2.1 will show you the transformation. Subsection 7.2.2 will show you how to deal with the resulting code.

### 7.2.1 Transforming Program Call

Consider a program with this header:

\[
Pr(x, \text{OUT } y)
\]

Logically we can transform a call like \( Pr(e_1, e_2) \) to the following, which has an equivalent effect:

\[
\@x, \@y := e_1, e_2; \ Pr(\@x, \@y); e_2 := \@y
\]

where \( \@x \) and \( \@y \) are fresh variables. The assignments before the call captures the passing of the actual parameters to the formal parameters. The assignment(s) after the call captures the copy-back action to the copy-restore parameters. Notice that after the transformation \( Pr \) is called in the form that can be handled by the Renaming Rule.

Be careful: if \( Pr \) has multiple copy-restore parameters, they are not restored simultaneously. Consistent to the agreement we made earlier (Section 7.1), these parameters are restored sequentially, and in the same order as they are listed as the program’s arguments. The next example shows this.

Consider again the ZeroOne example; in particular, the specification below:

\[
\{\ast \text{true} \ast\} \ i_0 := i; \ ZeroOne(a[i_0], i) \ \{\ast (a[i_0] = 0) \land (i = 1) \ast\}
\]

Applying the above transformation scheme we obtain:

\[
\{\ast \text{true} \ast\}
\]

\[
i_0 := i;
\]

\[
\@x, \@y := a[i], i; \ ZeroOne(\@x, \@y); \ a[i] := \@x; \ i := \@y
\]

\[
\{\ast (a[i_0] = 0) \land (i = 1) \ast\}
\]

where \( \@x \) and \( \@y \) are fresh variables. Applying the Renaming Rule allows us to infer that \( (\@x = 0) \land (\@y = 1) \) holds after the call \( ZeroOne(\@x, \@y) \). From this we will later be able to prove the specified post-condition, though we will need an additional formal mechanism to do so — we will return to this later.

Recall the example with the program TF (7.1). Below is a call which is similar to the problematic call \( TF(x, x) \): it would pass aliases if the parameters are passed by reference:

\[
ZeroOne(1, 1)
\]

However, it will have this meaning in uPL:

\[
\@x, \@y := 1, i; \ ZeroOne(\@x, \@y); \ i := \@x; \ i := \@y
\]

It will not lead to a contradiction. In fact, we will be able to infer that \( i = 1 \) holds after the call.

A uPL program also returns a value. We can handle return value by treating it as an implicit pass-by-reference parameter. For example, consider:
7.2. FORMAL TREATMENT OF PROGRAM CALL

The program is translated to the equivalent one below:

\[ \text{SUC}'(x: \text{int}, \text{OUT return: int}) \{ \text{return} := x + 1 \} \]

Subsequently a call like \( y := \text{SUC}(x) \) is translated to \( \text{SUC}'(x, y) \). With this translation scheme our logic will require no special treatment to handle the return value. Having said this, in the subsequent discussion we will focus on programs without return value.

Note by the way that a \textit{return} statement can only appear as the last statement of a \textsc{uPL} program.

#### 7.2.2 Black Box Reduction

The \textit{Renaming} Rule, even with the transformation scheme described in the previous subsection, only allows us to, essentially, instantiate a given black box specification. However, this is usually still not sufficient. To illustrate the problem, consider the (black box) specification below of a program \textit{INCR}:

\[
\{\forall x \geq 0 \} \quad X := x; \text{INCR(OUT x : int)} \{\forall x > X \} \quad (7.7)
\]

\( X \) is an auxiliary variable. The specification says that the program increases the value of \( x \) if it is initially non-negative. It does not specify how much \( x \) is increased.

Supposed we have another program that calls \textit{INCR} and requires the following post-condition after the call:

\[
\text{INCR}(x) \{\forall x > 0 \land y > 0 \} \quad (7.8)
\]

What would be a reasonable pre-condition to establish the post-condition?

The \textit{Renaming} Rule cannot help us, at least not directly: there is no way we can obtain the post-condition in (7.8) by applying some renaming on (7.7).

We have to find another way. From (7.7) we conclude, assuming the pre-condition of (7.7) holds, that \( x > X \) will hold after the call \( \text{INCR}(x) \). Since this is not quite the required post-condition \((x > 0 \land y > 0)\), the next best thing we can do is to require that \( x > X \) implies this required post-condition. So, this has to hold in the state after the call:

\[
x > X \Rightarrow x > 0 \land y > 0 \quad (7.9)
\]

Given only (7.7) as our knowledge about \textit{INCR}, the \textit{best} fact that we can infer about the value of \( x \) in the state after the call is that it satisfies \( x > X \); beyond this we do not know the exact value of \( x \). With this observation we can just as well, without loss of information, put a \( \forall \)-quantification over \( x \) in the post-condition (7.8). That is, we replace (7.8) with:

\[
(\forall x :: x > X \Rightarrow x > 0 \land y > 0) \quad (7.10)
\]

The call \( \text{INCR}(x) \) only affects the variable \( x \). But \( x \) no longer occurs free in the post-condition above! Hence, \( \text{INCR}(x) \) cannot change the meaning of (7.10). So, for (7.10) to hold after the call, the same has to hold \textit{before} the call. This is nice, because effectively we have now turned the post-condition requirement (7.9) into a pre-condition!

As further simplification, we replace the \( \forall \)-quantification over \( x \) in (7.10), which is now a pre-condition, with the use of a fresh variable, e.g. \( x' \), obtaining:

\[
x' > X \Rightarrow x' > 0 \land y > 0 \quad (7.11)
\]

With respect to the actual program variables, (7.11) and (7.10) are equivalent, because as a fresh variable we have no constraint on the value of \( x' \); thus, it is implicitly universally quantified.
In addition to (7.11), the call to INCR has to satisfy the pre-condition of (7.7); else we cannot use (7.7). So, the following pre-condition is needed:

\[ x \geq 0 \land (x' > X \Rightarrow x' > 0 \land y > 0) \]  

(7.12)

The \( X \) above comes from the specification of INCR (7.7). Recall that in the specification it refers to the value of \( x \) before the call. The above condition (7.12) is interpreted on the state before the call, where \( X = x \). We can therefore replace \( X \) with \( x \), obtaining:

\[ x \geq 0 \land (x' > x \Rightarrow x' > 0 \land y > 0) \]  

(7.13)

This is the final pre-condition. We conclude that INCR satisfies therefore the following specification:

\[ \{ \ast \ x \geq 0 \land (x' > x \Rightarrow x' > 0 \land y > 0) \ast \} \]

INCR\((x)\)

\[ \{ \ast \ x > 0 \land y > 0 \ast \} \]

By the way it is constructed, the pre-condition above is also the weakest one we can infer from INCR’s black box specification in (7.7).

Let \( Pr \) be a uPL program. Let \( \overline{x}, \overline{X}, \) and \( \overline{x}' \) be compatible lists of variables; the variables in \( \overline{x}' \) are fresh and all distinct. Formally, we can capture the above line of reasoning that we did for INCR in a single inference rule:

\[
\frac{
\{ \ast P \ast \} \ \overline{X} := \overline{x}; \ Pr(\overline{x}) \ \{ \ast Q \ast \}
}{
\{ \ast P \land Q' \ast \} \ \overline{P}:= \overline{x}/\overline{X}; \ Pr(\overline{x}) \ \{ \ast R \ast \}
}\]

(7.15)

where \( Q' = (Q \Rightarrow R)|\overline{x}'|/|\overline{x}|/|\overline{x}/\overline{X}| \).

Essentially \( Q' \) requires that the post-condition from the black box specification has to imply the actual post-condition \( R \). The two successive substitutions in \( Q' \) adapt this after-call requirement to a pre-condition. If you now try to apply the rule to the specification of INCR (7.7) and the post-condition specified in (7.8), you will obtain (7.14).

The rule still has one deficiency however. If \( x \) is a pass-by-value parameter or a read-only parameter of \( Pr \), we have agreed that occurrences of \( x \) in the post-condition of \( Pr \)’s specification refer actually to the value of \( x \) when it is passed to \( Pr \). In particular, it does not refer to \( x \)’s final value just before \( Pr \) returns —see also the discussion in page 53. The rule above does not yet take this fact into account. As an example, consider the following specification:

\[
\{ \ast \ \text{true} \ast \}
\]

test\((\text{READ} \ a : \ \text{int}[], \ x : \ \text{int}, \ \text{OUT return} : \ \text{bool})\)

\[ \{ \ast \ \text{return} = (x = a[0]) \ast \} \]

(7.16)

The \( x \) and \( a \) in the post-condition refer to the values of \( x \) and \( a \) when they are passed to \text{test}. So, we expect to be able to infer this from the above specification:

\[
\{ \ast \ x = a[0] \ast \} \ \text{test}(a,x,b) \ \{ \ast \ b \ast \}
\]

(7.17)

Let us now apply the rule (7.15) on the specification of \text{test} and, as the \( R \), the required post-condition above (7.17). We obtain:

\[
\{ \ast \ (b' = (x' = a'[0])) \Rightarrow b' \ast \}
\]

test\((a,x,b)\)

\[ \{ \ast \ b = (a[0] = 1) \ast \}
\]

(7.18)
7.2. FORMAL TREATMENT OF PROGRAM CALL

where \( x' \) is a fresh variable. The pre-condition above can be simplified to \( x' = a'[0] \). However, there is no way we can prove this from \( x = a[0] \). Hence, we fail to infer (7.17).

In applying rule (7.15) to the specification of \texttt{test}, we should not actually replace \( x \) and \( a \) in the post condition with fresh variables, since they refer to their respective values before the call. The rule below includes this revision. In the literature the rule is also called Adaptation Rule [12].

**Rule 7.2.2 : Black Box Reduction**

Let \( Pr \) be a uPL program. Let \( \pi, X, \) and \( \overline{\pi} \) be compatible lists of variables; variables in \( \overline{\pi} \) are fresh and distinct.

\[
\begin{align*}
\{ * P \} \quad X := \pi; \quad Pr(\pi) & \quad \{ * Q \} \quad \{ * P \land Q' \} \quad Pr(\overline{\pi}) & \quad \{ * R \}
\end{align*}
\]

where

\[
Q' = (Q \Rightarrow R) [\overline{\pi} / \pi] [\pi / X]
\]

and \( \overline{\pi} / \pi \) is a substitution that replaces \( x_i \) with \( x'_i \) only if \( x_i \) is a pass-by-copy-restore parameter of \( Pr \).

If we now use this Black Box Reduction Rule on the specification of \texttt{test} (7.16) and as the \( R \) the post-condition in (7.17), we obtain:

\[
\{ * (b' = (x = a[0])) \Rightarrow b') \}\]

\texttt{test(a,x,b)} \tag{7.19}

\[
\{ * b = (a[0] = 1) \}
\]

The pre-condition can be simplified to \( x = a[0] \) (was \( x' = a'[0] \)), which is exactly the pre-condition given in (7.17).

Notice that Black Box Reduction can only handle calls that pass variables as actual parameters. In the examples so far this is not a problem. In a more general case, we will have to combine its use with the transformation discussed in Subsection 7.2.1. We will show this in the next example.

**Example**

Consider this specification:

\[
\{ * \text{true} \}
\]

\[
X,Y := x,y; \quad \text{UPPER}(x,y : \text{int}, \text{OUT return} : \text{int}) \quad \{ * \text{return} \geq x \text{max} y \}
\]

(7.20)

So, \text{UPPER} returns some integer which is greater or equal to the maximum of \( x \) and \( y \). We want to prove that \text{UPPER} also satisfies the following specification:

\[
\{ * a[i] > 0 \} \quad \text{UPPER}(a[i], 0, 1) \quad \{ * i > 0 \}
\]

(7.21)

Since we pass a complex expression \( (a[i]) \) in the call above, we cannot directly use the Black Box Reduction Rule. So, as we did in Subsection 7.2.1, we first transform the call the an equivalent one below. All variables whose name begin with \( @ \) are assumed to be fresh.

\[
\{ * a[i] > 0 \} \quad \text{UPPER}(@x, @y, @r := a[i], 0, 1) \quad \{ * i > 0 \}
\]

(7.22)
By calculating \( wp \) we can equivalently prove this instead:

\[
\{ \ast a[i] > 0 \ast \} \\
@x, @y, @r := a[i], 0, i \\
\text{UPPER}(@x, @y, @r) \\
\{ \ast @r > 0 \ast \}
\]  

(7.23)

The next step is to use the Black Box Reduction Rule. However, the actual paremeters passed above does not match the specification (7.20), which the Black Box Reduction Rule expects. We first has to adapt (7.20) to match the actual parameters. This is done by applying the Renaming Rule (Rule 7.2.1); we obtain:

\[
\{ \ast \text{true} \ast \}
\]

(7.24)

\[
X, Y := @x, @y; \text{UPPER}(@x, @y, \text{OUT} @r) \\
\{ \ast @r \geq @x \max @y \ast \}
\]

After calculating \( wp \) it comes down to proving this implication:

\[
a[i] > 0 \Rightarrow (r' \geq a[i] \max i \Rightarrow r' > 0)
\]  

(7.26)

which is quite trivial to prove.

### 7.3 Functional Black Boxes

An imperative program that does not have a side effect is said to be functional, as it can be thought to implement a mathematical function. Functions are nice because their behavior are fully determined by the relation between the inputs and their return values, which make reasoning about them simpler.

For simplicity, we will further require that a functional uPL program takes no writable copy-restore parameter. The post-condition in its specification must fully specify what the return value is. So, the post-condition must be of the form:

\[
\{ \ast P \ast \} P(r, x) \{ \ast \text{return} = e \ast \}
\]

for some expression \( e \). Notice that any occurrence of a variable \( x \) from \( x \) in \( e \) refers thus to \( x \)'s initial value. An example of such a specification is the specification of \( \text{SUC} \):

\[
\{ \ast \text{true} \ast \} \text{SUC}(x) \{ \ast \text{return} = x + 1 \ast \}
\]

The Black Box Reduction Rule can be specialized to handle functional programs:
Rule 7.3.1: Functional Box Reduction I
Let $Pr$ be a functional uPL program and $\pi$ be a list of variables.

\[
\begin{align*}
\{ \ast P \ast \} & Pr(\pi) \{ \ast return = e \ast \} \\
\hline
\{ \ast P \land R[e/r] \ast \} & r:=Pr(\pi) \{ \ast R \ast \}
\end{align*}
\]

□

Proof: for simplicity, assume $Pr$ just have one parameter $x$. Treat $Pr$ as if it has $Pr(x, \text{OUT return})$ as header, and treat the call $r:=Pr(x)$ as $Pr(x, r)$. Applying the Black Box Reduction we obtain $P \land ((r' = e) \Rightarrow R[r'/r])$ as a pre-condition. The second conjunct is equivalent to $R[e/r]$.

□

Since $R[e/r]$ is also the wp of $r:=e$ with respect to the post condition $R$, the Rule above essentially allows us to treat a call to $Pr$ as an assignment. We do have to strengthen the resulting wp with $P$, since $P$ specifies when the call is allowed. We can also prove the following corollary, which can handle arbitrary actual parameters (the above version of Functional Box Reduction require all actual parameters to be variables):

Rule 7.3.2: Functional Box Reduction II

Let $Pr$ be a functional uPL program, $\pi$ be a list of variables, and $d$ be a list of expressions.

\[
\begin{align*}
\{ \ast P \ast \} & Pr(\pi) \{ \ast return = e \ast \} \\
\hline
\{ \ast P[d/\pi] \land R[e'/r] \ast \} & r:=Pr(d) \{ \ast R \ast \}
\end{align*}
\]

where $e' = e[d/\pi]$.

□

The proof is left out for you as an exercise.

Example

Recall the program test:

\[
\begin{align*}
\{ \ast true \ast \} \\
\text{test(READ a : int[], x : int, OUT return : bool)} \tag{7.27} \\
\{ \ast return = (x=a[0]) \ast \}
\end{align*}
\]

The program is functional. Moreover the post-condition is of the form $\text{return} = e$. Earlier we have inferred by directly using Black Box Reduction that the call $\text{test}(a,x,b)$, which we can also write as $b := \text{test}(a,x)$, satisfies:

\[
\{ \ast x = a[0] \ast \} \ b := \text{test}(a,x) \{ \ast b \ast \} \tag{7.28}
\]

Using plain Black Box Reduction we first arrive in the specification (7.19) with a complicated looking pre-condition. Using the Functional Box Reduction we can infer (7.28) directly from (7.27).
7.4 Recursion

The logic of uPL so far cannot yet handle recursive programs. Consider the following simple example:

```plaintext
test(OUT n:int) : ()
{
    if n=0 then skip else {
        n:=n-1; test(n)
    }
}
```

Suppose we want to show that it satisfies this:

\{∗ n ≥ 0 ∗\} test(n) \{∗ n = 0 ∗\}

Since a recursion is a form of iteration, we can try to proceed along the same line used to handle the while-loop. To show that the recursion terminates we try to come up with an integer expression which is decreased by each recursion; \( n \) will do in the above case. This expression has to be bounded below by, for example, zero. This can be imposed by the pre-condition. That is, we require that the pre-condition implies \( n ≥ 0 \). In the above case this is trivially satisfied.

Next, we show that if \( \text{test}(n) \) is correct, then so is \( \text{test}(n') \), where \( n' = n + 1 \). Suppose we can show that for the 'base case', namely \( n = 0 \), \( \text{test} \) is correct. It follows that for \( n = 1 \), \( \text{test} \) is correct. Then it follows that it is also correct for \( n = 2 \), and so on. Hence, \( \text{test} \) is correct for all non-negative \( n \). Formally, this is captured by the inference rule below. It is slightly more general, as it allows the termination metric \( n \) to decrease by any positive unit per recursion.

**Rule 7.4.1 : REC Reduction**

Let \( \pi \) be \( Pr \)'s formal parameters. Let \( n \) be an integer expression.

**BC1** : \( P \Rightarrow n \geq 0 \)

**BC2** : \{∗ P ∧ (n = 0) ∗\} \( \overline{X} := \pi; \ Pr(\overline{X})\{∗ Q ∗\} \)

**RC** : \{∗ P ∧ n < K ∗\} \( \overline{X} := \pi; \ Pr(\overline{X})\{∗ Q ∗\} \)

implies

\{∗ P ∧ (n = K) ∧ K > 0 ∗\} \( \overline{X} := \pi; \ Pr(\overline{X})\{∗ Q ∗\} \)

\( \Box \)

We leave the proof of \( \text{test} \) to you as an exercise.

The rule above can be specialized for functional uPL programs that recurse down an integer parameter \( n \) by 1 at a time. The specialized rule is shown below:

**Rule 7.4.2 : REC Reduction II**

Let \( Pr \) be a functional uPL program; \( \pi \) are its formal parameters, and \( n \in \pi \). Let \( e(K) \) denote \( e[K/n] \).

**BC1** : \( P \Rightarrow n \geq 0 \)

**BC2** : \{∗ P ∧ (n = 0) ∗\} \( Pr(\pi)\{∗ \text{return} = e(0) ∗\} \)

**RC** : \{∗ P ∧ (n = K) ∗\} \( Pr(\pi)\{∗ \text{return} = e(K) ∗\} \)

implies

\{∗ P ∧ (n = K+1) ∧ K≥0 ∗\} \( Pr(\pi)\{∗ \text{return} = e(K+1) ∗\} \)

\( \Box \)
Example

Consider the following recursive program to sum all integers from 0 up to \( n \):

```plaintext
SUMN(READ n:int)
{
    var s:int;
    if n=0 then s:=0 else s:=SUMN(n-1);
    return s+n
}
```

Let us prove the following specification of \( SUMN \):

\[
\{ \star n \geq 0 \} \quad SUMN(n) \{ \star return = (n^2 + n)/2 \} \quad (7.29)
\]

Since \( SUMN \) is functional and recurs down \( n \) with step 1, we can use the second \textit{REC Reduction} rule, which says that it is sufficient to prove the following:

1. **BC1**: \( n \geq 0 \Rightarrow n \geq 0 \). Trivial.

2. **BC2**: \( \{ \star n \geq 0 \land (n = 0) \} \quad SUMN(n) \{ \star return = 0 \} \). This is quite easy to prove; we leave it to you.

3. The recursion step (**RC**). Assuming the following recursion hypothesis, call it \( RH \):

\[
\{ \star n \geq 0 \land (n = K) \} \quad SUMN(n) \{ \star return = (K^2+K)/2 \} \]

show that:

\[
\{ \star n \geq 0 \land (n = K+1) \land K \geq 0 \} \quad SUMN(n) \{ \star return = ((K+1)^2+K+1)/2 \} \]

For the last proof obligation, it comes down that we have to prove the following specification of \( SUMN \)'s body:

\[
\{ \star n \geq 0 \land (n = K+1) \land K \geq 0 \} \]

\[
\text{if } n = 0 \text{ then } s := 0 \text{ else } s := SUMN(n-1)
\]

\[
\{ \star s + n = ((K+1)^2+K+1)/2 \}
\]

The pre-condition in (7.30) implies \( n \neq 0 \). So, the \textit{else}-branch will be taken; it is therefore sufficient to prove only the \textit{else}-branch:

\[
\{ \star n \geq 0 \land (n = K+1) \land K \geq 0 \} \]

\[
s := SUMN(n-1)
\]

\[
\{ \star s + n = ((K+1)^2+K+1)/2 \}
\]

By the \textit{Functional Box Reduction} Rule, using the specification assumed in **RC**, we can equivalently prove:

\[
n \geq 0 \land (n = K+1) \land K \geq 0 \quad \Rightarrow \quad ((n-1)^2+n-1)/2 + n = ((K+1)^2+K+1)/2
\]

Since \( n = K+1 \) is assumed, we can simplify the right hand side of the implication to:

\[(K^2+K)/2 + K + 1 = ((K+1)^2+K+1)/2
\]

This is proven below:
\[
\frac{(K + 1)^2 + (K + 1))}{2} = \\
\frac{(K^2 + 2K + 1 + K + 1)}{2} = \\
\frac{(K^2 + K + 2K + 2)}{2} = \\
\frac{(K^2 + K)}{2} + K + 1
\]

We have closed all proof obligations. So we conclude that the specification (7.29) is indeed valid.
7.5 **Exercise**

1. Consider again the ZeroOne program. The specification is in (7.5). In Subsection 7.2.1 we have discussed the following use of ZeroOne, but we have not finished its proof:

\[
\{ \text{true} \} \quad i_0 := i; \text{ZeroOne}(a[i], i) \quad \{ (a[i_0] = 0) \land (i = 1) \}
\]

Complete the proof.

2. Given the following program:

\[
\text{CSWAP}(n, \text{OUT } x, y) : ()
\{
\text{var } \text{tmp:int;}
\text{if } n > 0 \text{ then } \{ \text{tmp:=}x; x:=y; y:=\text{tmp} \} \text{ else skip } \}
\]

(a) Give a formal black box specification for CSWAP.
(b) Prove the following specification. You are not allowed to use CSWAP as a white box.

\[
\{ \text{true} \} \quad x := y + 1; \text{CSWAP}(x, x, y) \quad \{ x \geq 0 \land y > 0 \}
\]

(c) Now try this specification:

\[
\{ \text{true} \} \quad x, y := a[i], i; \text{CSWAP}(i, a[i], i) \quad \{ a[y] = y \}
\]

Is \( a[i] = y \) a valid post-condition for the call above?

3. Given a program \( \text{Pr} \) with the following black box specification:

\[
\{ \text{true} \} \quad \text{EXP2}(n) \quad \{ \text{return} \leq 2^n \}
\]

Prove this specification:

\[
\{ \text{true} \} \quad n := \text{EXP2}(n); m := \text{EXP2}(m) \quad \{ m \land n > 0 \}
\]

Use the Functional Box Reduction.

4. Here is the program test from Section 7.4:

\[
\text{test(OUT } n\text{)} : ()
\{
\text{if } n = 0 \text{ then skip else } \{ n := n-1; \text{test(n)} \}
\}
\]

Prove this specification:

\[
\{ \text{true} \} \quad \text{test}(n) \quad \{ n = 0 \}
\]

5. Given the following recursive program:

\[
\text{allTrue}(n, \text{READ a:bool[]} \) : \text{bool}
\{
\text{var } b\text{;}
\text{if } n = 0 \text{ then } b := \text{true}
\text{else } \{ b := \text{allTrue}(n-1, a) ;
\quad \text{b := a[n-1]}/\b
\}
\}
\]

The program checks whether the array \( a \) contains only true's in the domain \( 0 \leq i < n \). Give a formal specification, and prove its validity.
7.6 Solution

1. We first transform the code $i_0 := i; \text{ZeroOne}(a[i], i)$ into an equivalent one shown below. The pre- and post-conditions are copied, and additional assertions, which are yet to be calculated, are added.

\[
\begin{align*}
&\{\ast \text{true} \ast\} \\
&\{\ast Q_1 \ast\} i_0 := i; \\
&\{\ast Q_2 \ast\} @x, @y := a[i], i; \\
&\{\ast Q_3 \ast\} \text{ZeroOne}(@x, @y); \\
&\{\ast Q_4 \ast\} a[i] := @x; i := @y \\
&\{\ast (a[i_0] = 0) \land (i = 1) \ast\}
\end{align*}
\]

$@x$ and $@y$ are fresh variables.

$Q_4$ can be easily calculated with $wp$. We obtain:

\[
Q_4 : ((i_0 = i \rightarrow @x|a[i_0]) = 0) \land (@y = 1)
\]

$Q_3$ is obtained from $Q_4$ with the Black Box Reduction. However, first we need to properly instantiate the given black box specification of $\text{ZeroOne}$ (7.5) to the one below. This goes via the Renaming rule.

\[
\begin{align*}
&\{\ast \text{true} \ast\} \text{ZeroOne}(\text{OUT} @x, @y : \text{int}) \{\ast (@x = 0) \land (@y = 1) \ast\}
\end{align*}
\]

Now, applying the Black Box Reduction on $Q_4$ using the specification above, we obtain:

\[
Q_3 : (x' = 0) \land (y' = 1) \Rightarrow ((i_0 = i \rightarrow x'|a[i_0]) = 0) \land (y' = 1))
\]

where $x'$ and $y'$ are fresh variables. Notice that in this step $@x$ and $@y$ are replaced by $x'$ and $y'$.

$Q_2$ is obtained by $wp$ calculation on $Q_3$. Notice that neither $@x$ or $@y$ appear in $Q_3$; so $Q_2$ is just the same as $Q_3$.

$Q_1$ can be obtained by $wp$ calculation on $Q_2$. We obtain:

\[
Q_1 : (x' = 0) \land (y' = 1) \Rightarrow ((i = i \rightarrow x'|a[i]) = 0) \land (y' = 1))
\]

which can be simplified to:

\[
(x' = 0) \land (y' = 1) \Rightarrow ((x' = 0) \land (y' = 1))
\]

which is obviously true.

2. (a) A reasonable specification:

\[
\begin{align*}
&\{\ast n > 0 \ast\} \\
&X, Y := x, y; \text{CSWAP}(n, x, y) \\
&\{\ast (x = Y) \land (y = X) \ast\}
\end{align*}
\]
(b) We first transform the code into an equivalent one below. Intermediate assertions $Q_1 \ldots Q_4$ are added —they are yet to be calculated.

\[
\begin{align*}
\{ y \geq 0 \} \\
\{ Q_1 \} & \ x := y + 1; \\
\{ Q_2 \} & \ \mathit{@n}, \mathit{@x}, \mathit{@y} := x, x, y; \\
\{ Q_3 \} & \ \text{CSWAP} (\mathit{@n}, \mathit{@x}, \mathit{@y}); \\
\{ Q_4 \} & \ x := \mathit{@x}; \ y := \mathit{@y}
\end{align*}
\]

\[
\begin{align*}
\{ x \geq 0 \land y > 0 \}
\end{align*}
\]

$\mathit{@n}$, $\mathit{@x}$, and $\mathit{@y}$ are fresh variables.

$Q_4$ can be easily calculated with wp. We obtain:

\[
Q_4 : \ \mathit{@x} \geq 0 \land \mathit{@y} > 0
\]

We first instantiate the specification of CSWAP (see (2a)) to the following. Use the Renaming Rule.

\[
\begin{align*}
\{ \mathit{@n} > 0 \} \\
N, X, Y := \mathit{@n}, \mathit{@x}, \mathit{@y}; \ \text{CSWAP} (\mathit{@n}, \mathit{@x}, \mathit{@y}) \\
\{ (\mathit{@x} = Y) \land (\mathit{@y} = X) \}
\end{align*}
\]

We can obtain $Q_3$ by applying Black Box Reduction rule on $Q_4$, using the above instantiated specification of CSWAP. We obtain:

\[
Q_3 : \ \mathit{@n} > 0 \\
\land \\
((x' = \mathit{@y}) \land (y' = \mathit{@x}) \Rightarrow x' \geq 0 \land y' > 0)
\]

$x'$ and $y'$ are fresh. Notice that in this step we replaced $\mathit{@x}$ in $Q_4$ with $x'$ and $X$ with $\mathit{@x}$. The same goes with $\mathit{@y}$ and $Y$. On the other hand, $\mathit{@n}$ is left unreplaced; do you know why?

$Q_2$ is obtained by calculating the wp on $Q_3$:

\[
Q_2 : \ x > 0 \land ((x' = y) \land (y' = x) \Rightarrow x' \geq 0 \land y' > 0)
\]

$Q_1$ is obtained by calculating the wp on $Q_2$:

\[
Q_1 : \ y + 1 > 0 \\
\land \\
((x' = y) \land (y' = y + 1) \Rightarrow x' \geq 0 \land y' > 0)
\]

To prove the correctness of the original specification it is now sufficient to show that the original pre-condition $y \geq 0$ implies $Q_1$. We leave out the proof for you (quite easy).

(c) This can be proven in the same way as above. What is a bit tricky here is that a call like $\text{CSWAP}(i, a[i], i)$, because of the $i$ in the last two parameters, is sensitive to the restore order of the parameters. To put it differently, $\text{CSWAP}(i, a[i], i)$ and $\text{CSWAP}(i, i, a[i])$ will have different effects on the array $a$. This is one of those subtle difference between pass-by-copy-restore and pass-by-reference. Anyway, the proof should not be a problem once you get the translation correct, which is shown below:
CHAPTER 7. PROGRAM CALL

{∗ i > 0 ∗}
{∗ Q₁ ∗} x, y := a[i], i
{∗ Q₂ ∗} @n, @x, @y := i, a[i], i ;
{∗ Q₃ ∗} CSWAP(@n, @x, @y) ;
{∗ Q₄ ∗} a[i] := @x; i := @y

{∗ a[y] = y ∗}
We leave the calculation of Q₁ ... Qₙ to you.

As for the second question, a[i] = y is not a valid post-condition.

3. We first add some intermediate assertions:

{∗ m ≥ 0 ∧ n ≥ 0 ∗}
{∗ Q₁ ∗} n := EXP2(n);
{∗ Q₂ ∗} m := EXP2(m)
{∗ m ≥ n > 0 ∗}

The program EXP2 is functional, and its given specification matches the form expected by the functional reduction rules. We will use Rule 7.4.2.

Q₂ is computed from the final post-condition using Rule 7.4.2:

Q₂ : 0 ≤ m ∧ 2ⁿ*m > 0

Q₁ is computed from Q₂ using Rule 7.4.2:

Q₁ : 0 ≤ n ∧ 0 ≤ m ∧ 2ⁿ*2ⁿ > 0

It remains to prove that the orginal pre-condition m ≥ 0 ∧ n ≥ 0 implies Q₁. This is quite easy.

4. Applying the REC Reduction (Rule 7.4.1), we have three conditions to prove. Take n as the termination metric. BC₁ is trivial. The remaining two:

(a) BC₂: {∗ n ≥ 0 ∗} test(n) {∗ n = 0 ∗}. This is quite easy to prove.
(b) RC. Assuming {∗ n ≥ 0 ∧ n < K ∗} test(n) {∗ n = 0 ∗}, prove:

{∗ n ≥ 0 ∧ (n = K) ∧ K > 0 ∗} test(n) {∗ n = 0 ∗}

Using the Program to Statement Reduction (Theorem 6.11.1) it is sufficient to prove:

{∗ n ≥ 0 ∧ (n = K) ∧ K > 0 ∗} if n = 0 then skip else n := n - 1; test(n) {∗ n = 0 ∗}

Since the pre-condition above implies n ≠ 0, the else-branch will be taken. So, it is sufficient to prove:

{∗ n ≥ 0 ∧ (n = K) ∧ K > 0 ∗}
{∗ Q₁ ∗} n := n - 1;
{∗ Q₂ ∗} test(n)
{∗ n = 0 ∗}
Q₂ can be obtained with the Black Box Reduction, using the specification given in the above assumption of RC:

\[ Q₂: \ n ≥ 0 ∧ n < K ∧ (n' = 0) \implies (n' = 0) \]

where \( n' \) is a fresh variable. The last conjunct can be dropped; it is trivially equivalent to \text{true}.

Q₁ is obtained by calculating \( \text{wp} \) from Q₂:

\[ Q₁: \ n - 1 ≥ 0 ∧ n - 1 < K \]

It is quite trivial to show that this is implied by the given pre-condition.

5. A reasonable specification for allTrue:

\[
\{ \ast \ n ≥ 0 \} \ \text{allTrue}(n, a) \ \{ \ast \ \text{return} = \text{all} \ a \ n \} \\
\]

where:

\[
\text{all} \ a \ n = (\forall i : 0 ≤ i < n : a[i])
\]

Since allTrue is functional and it recurs down \( n \) with step one, we will use Theorem 7.4.2. The BC₁ condition is trivial. We are left to prove the following:

(a) The BC₂ condition:

\[
\{ \ast \ n = 0 \} \ \text{allTrue}(n, a) \ \{ \ast \ \text{return} = \text{all} \ a \ 0 \} \\
\]

We leave the proof to you.

(b) The RC condition. Assuming:

\[
\{ \ast \ n ≥ 0 ∧ (n = K) \} \ \text{allTrue}(n, a) \ \{ \ast \ \text{return} = \text{all} \ a \ K \} \\
\]

prove:

\[
\{ \ast \ n ≥ 0 ∧ (n = K + 1) ∧ K ≥ 0 \} \ \text{allTrue}(n, a) \ \{ \ast \ \text{return} = \text{all} \ a \ (K+1) \} \\
\]

We continue with the proof of the recursion case. Using the Program to Statement Reduction (Theorem 6.11.1) it is sufficient to prove:

\[
\{ \ast \ n ≥ 0 ∧ (n = K + 1) ∧ K ≥ 0 \} \ A := a; \ \text{body} \ \{ \ast \ \text{b} = \text{all} \ A \ (K+1) \} \\
\]

where \text{body} is allTrue's body, past the declaration of the local variable \( b \).

Since the pre-condition implies that \( n > 0 \), the else-branch of \text{body} will be taken. So, it is sufficient to prove the following:

\[
\{ \ast \ n ≥ 0 ∧ (n = K + 1) ∧ K ≥ 0 \} \\
\{ \ast \ Q₁ \} \ A := a; \\
\{ \ast \ Q₂ \} \ b := \text{allTrue}(n-1, a); \\
\{ \ast \ Q₃ \} \ b := a[n-1] ∧ b \\
\{ \ast \ b = \text{all} \ A \ (K+1) \} \\
\]
Q\(_3\) is obtained by calculating wp:

\[
Q_3 : \quad a[n-1] \land b = \text{all } A \ (K+1)
\]

Q\(_2\) is obtained via Rule 7.4.2 on Q\(_3\):

\[
Q_2 : \quad n-1 \geq 0 \land (n-1 = K) \land (a[n-1] \land \text{all } a K = \text{all } A \ (K+1))
\]

Q\(_1\) is obtained by calculating wp on Q\(_2\):

\[
Q_1 : \quad n-1 \geq 0 \land (n-1 = K) \land (a[n-1] \land \text{all } a K = \text{all } a \ (K+1))
\]

The first two conjuncts are implied by the given pre-condition. What is left to prove is this equality:

\[
a[n-1] \land \text{all } a K = \text{all } a \ (K+1)
\]

This can be proven from \(n-1 = K\) and \(K \geq 0\). The proof is left to you.
Abstract Data Type

ADT is an example of a more advanced program structure that can be built beyond the ordinary statements that we have seen in uPL so far. We will discuss here issues related to its verification and validation.

ADT stands for Abstract Data Type. It is a closed data structure together with a set of operations to access and manipulate the structure. An ADT is closed in two ways. Firstly, the only way to access (and modify) the data stored in an ADT is through the provided operations. This provides some security. For example, implementing a stack as an ADT would force access to the stack to go through its operations. This in turn would prevent the programmer from e.g. accidentally taking out an element from the wrong side of a stack. Secondly, an ADT does not expose the actual implementation of its data structure nor its operations. This is why it is called abstract: it hides implementation details. This allows an ADT to be implemented in different ways. Furthermore, any change in the implementation will not influence its use, since it never relies on any implementation knowledge.

A similar idea of abstraction approach is also found in the Object Oriented (OO) proramming and in Component Based Approach (CBA), e.g. CORBA. An object in OO hides its private fields. Like in ADT, access to them is only possible via the object’s methods. A component in CBA are intended to be highly reusable. To achieve this, a component does not expose its code. Like ADT it only offers a partial information about its functionality, such as the methods it provides and their specifications.

Z [8] and B [2] are examples of industrial strength specification languages that rely a lot on the ADT idea. In Z it is called schema and in B it is called abstract machine. Both have been used for specifying large software.
ADT Queue {
  model Queue = [int]

  operation
    init() : ()
    size() : int
    qin(x:int) : ()
    qout() : int

  spec  // q below is of type Queue
    {∗ true *} q.init() {∗ q=[] *}
    {∗ true *} q.size() {∗ return=COUNT q *}
    {∗ true *} Q:=q; q.qin(x) {∗ q=Q++[x] *}
    {∗ q ≠ [] *} Q:=q; q.qout() {∗ (return=head Q) ∧ (q=tail Q) *}
}

Figure 8.1: An example of an ADT.

8.1 ADT with uPL

Let us extend our language uPL so that we can also specify an ADT. Figure 8.1 shows an example. Such an ADT declaration introduces a new type of data, in this case it is called Queue. As the name suggests, this will be an ADT representing queues.

The model part specifies that we will model the type Queue with the type integer list (denoted by [Int]). This does not say that we will implement a Queue with the said list. It just says that abstractly we will pretend as if it is a list. In particular an ADT's operations will be specified in terms of the ADT’s model.

The operation section lists the operations that are available for this ADT. We see four operations are listed for Queue. It is illegal to access an ADT through any means other than its operations.

The spec section specifies the behavior of these operations with Hoare triple specifications. We use a little OO syntax flavour in the specifications. E.g. the specification of init looks like this:

    {∗ true *} q.init() {∗ q=[] *}

The notation q.init() is new in uPL. What we mean here is that init actually has q as it first parameter, which is furthermore passed by copy-restore. Implicitly, q is assumed to be of type Queue. So, the header of init is actually:

    init(OUT q : Queue): ()

And a call like q.init() is actually the call init(q). Similarly, qin actually has this header:

    qin(OUT q : Queue, x : int): ()

and the call q.qin(x) is actually qin(q,x).
8.2. EXTENDING ADT

The operation *init* is a bit special. It is used to initialize an ADT when it is instantiated. That is, each instantiation of an ADT, e.g. when we create a local variable of its type:

```
{var a : Queue; ...do something with a...}
```

This block creates thus a fresh instance of `Queue` and binds it to the local variable `a`. Immediately after its creation, the operation *init* will be called on it to initialize its content.

From the specifications we can infer that *init* sets the queue to empty; *size* returns the length of the queue; *qin* is used to enter a value into the queue; and *qout* is used to retrieve the oldest value from the queue.

8.2 Extending ADT

In OO a new class can be introduced by extending (subclassing) an existing one. The new class inherits all the fields and methods of the old class. This is a useful construction method. We can also do it with ADTs: a new ADT can be introduced by extending an existing one. We will keep it simple here: an extension only adds new operations (and their specifications). In particular, we do not allow operations to be overriden (as in Java). An example is shown below. It defines an ADT for a queue with an additional operation *dequeue* to retrieve the last element entered to the queue. In the specification, *last* returns the last element of a list, and *lead* returns the entire list except the last element.

ADT DQueue extend Queue {

    operation
        dequeue() : int

    spec
        {∗ q ≠ [] ∗} Q:=q; q.dequeue() {∗ (return=last Q) ∧ (q=lead Q) ∗}

}

The ADT DQueue inherits Queue’s data model (which models a queue as a list) and operations. Any correct implementation of Queue that satisfies the additional specification imposed by DQueue is therefore also an implementation of Queue.

8.3 Implementing ADT

An ADT can be implemented by providing a concrete data structure and code for its operations; this process is also called *data refinement*. We will extend uPL so that we can also express an ADT implementation. An example is shown in Figure 8.2.

The *rep* keyword stands for ‘representation’. The line:

```
Queue = record{a : int[] ; m, n : int}
```

states that this implementation (of `Queue`) represents a queue with a value of the type specified after ‘=’: so, a record of the above structure. If `r` is such a record, the idea here is to store the elements of the queue which `r` represents in the array `r.a`, more specifically in the segment `r.a[m...n]`.

When an instance of `Queue` is created, we actually create a record of the above type. When an operation `p` is applied on a queue, we actually apply the operation on a record of the above type.
IMPL Queue {
  rep Queue = record { a:int[] ; m,n:int }

  absfn α r = r.a[r.m..r.n]

  inv r.m≤r.n // r ranges over the above record

  impl // r below is a record of the above type
  r.init() :() { r.m,r.n:=0,0 }
  r.size() :int { return (r.n-r.m) }
  r.qin(x:int):() { r.a[r.n]:=x; r.n:=r.n+1 }
  r.qout():int
    {var x:int;
      if r.n-r.m>0 then {x:=r.a[r.m]; r.m:=r.m+1 } else skip ;
      return x }
}

Figure 8.2: An example of an ADT implementation.

Remember that in its specification a queue was modelled by a list. We now also have to specify how a record of the above type should be mapped to the Queue’s model. This is specified in the absfn section, which stands for abstraction function. The line:

    α r = r.a[r.m..r.n]

defines a function α that maps records of the above type to Queue’s model, which is [int]. This function is called abstraction function. It defines the meaning of each record of the above type in terms of Queue’s model. As you can see, it says that each record r is to be interpreted as the list r.a[r.m..r.n].

We will explain the inv later. The impl section specifies the implementation of each of Queue operations. However, instead of passing a 'queue' we now of course pass a concrete value representing it, which is a record of the type as specified in the rep section. So for example init would concretely have this type:

    init(OUT r : record[a:int[] ; m,n:int]):() 

This type of init is only known within the implementation. Externally, it would have the type as specified by the ADT to which it belongs. So, externally its type remains:

    init(OUT r : Queue):()

Verifying that the implementation is correct amounts to verifying each specification in Queue’s spec section. However, we have a slight mismatch here. The specification assumes a queue to be represented as a list, whereas the implementation uses a record. Fortunately we have the abstraction function α. Recall that its role was to provide an interpretation of records in terms of lists. Exactly what we need here! However, as the consequence we also have to adapt the original specifications of Queue. For example, this is the original specification of init:

    {*} true *} q.init() { q = [] *}
However q is now actually represented by a record, say r. Given such a record r, α specifies which list it represents, namely: r.a[r.m...r.n]. So, we have to adapt the specification above to:

\{true\} \ r.init() \ {r.a[r.m...r.n] = []}

The init above is now the 'concrete' init. This is the init as defined in the implementation in Figure 8.2. The above is thus the specification we have to verify for this concrete init.

Similarly, this was the original specification for qin:

\{true\} Q := q; q.qin(x) \ {q = Q + [x]}

This has to be adapted to:

\{true\} Q := r.a[r.m...r.n]; r.qin(x) \ {r.a[r.m...r.n] = Q + [x]}

The specification for init can be proven without problem. However that of qin is only valid if \(r.m \leq r.n\) holds in the pre-condition. The idea of the fields m and n is that they specify the left, respectively, right border of the array r.a that actually contains the elements of our queue. So it actually makes sense to expect that if r is not in the middle of being processed by one of the Queue's operations, then \(r.m \leq r.n\) holds. Such a property is also called a data invariant.

We can require \(r.m \leq r.n\) as the data invariant of the implementation in Figure 8.2. This is specified in the inv section. When proving that the implementation is correct, we are allowed to strengthen the pre-conditions with the data invariant. So, in the case of proving the specification of qin, we are allowed to assume \(r.m \leq r.n\):

\{true\} Q := r.a[r.m...r.n]; r.qin(x) \ {r.a[r.m...r.n] = Q + [x]}

Proving this specification poses now no problem.

More precisely, a predicate I is a data invariant of an implementation of ADT A if the implementation makes it so that I holds when an instance of A is created, and is restored by every operation of A. Note that this is not the same notion of 'invariant' as in the context of loops. A data invariant can be thought to express a constraint on the well formedness (integrity) of a data structure. It is analogous to what in JML/Java is called class invariant.

Of course now we also have the obligation to prove that the claimed data invariant is maintained. Firstly, we have to show that it is established when an ADT is instantiated. It means we have to prove that (the implementation of) init satisfies:

\{true\} r.init() \ {r.m \leq r.n}\}

Secondly, we have to show that (the implementation of) each operation maintains the invariant. For example, for qin we have to prove:

\{true\} Q := r.a[r.m...r.n]; r.qin(x) \ {r.a[r.m...r.n] = Q + [x]}

Using ADT in Implementation

We can also use ADTs to implement another ADT. Consider the following stack ADT:
ADT Stack {

model Stack = [int]

operation
    init() : ()
    push(x:int) : ()
    pop() : int

spec // s below is of type Stack

    {* true *} s.init() {s = [] *}
    {* true *} S:=s; s.push(x) {s = x : S *}
    {* s ≠ [] *} S:=s; s.pop() {s = tail S ∧ (return = head S) *}
}

We can implement it using DQueue (from the previous section):

IMPL Stack {
    rep Stack = DQueue
    absfn ...
    inv ...
    impl // q below is of type DQueue
    q.init() {q.init() }
    q.push(x:int) {q.qin(x)}
    q.pop() {var y; y:=q.dequeue() ; return y }
}

The abstraction function and data invariant are left out for exercise. The nice thing about
the implementation above is that it is independent of the chosen implementation of the DQueue.
Though on the other hand, when verifying the correctness of the above implementation we cannot
use the information on how DQueue is implemented, because this may not be known. So, all we
can use is the specification of DQueue itself. So, if it is weak, we may not be able to show the
correctness of the above implementation.

8.4 Validation

So far we usually use a program’s specification for (abstractly) expressing the program’s function-
ality. Such a specification can be used, as discussed in Chapter 7, as a substitute for the program’s
code when reasoning about the program’s effect when it is called by another program.

We can also write specifications purely for the purpose of validating a program. Validation is
often contrasted to verification. These are popular terms, but unfortunately there does not seem
to be a consensus on what they precisely mean, except perhaps that verification is more rigorous
than validation. Here we will define verification as the act of proving that a program satisfies a
functional specification. A functional specification is a specification that describes the program’s
functionality. Validation is the act of proving that the program satisfies a set of specifications.
In particular, these specifications do not have to describe the program’s functionality. However,
failure to satisfy one of them would imply that the program is incorrect. In a sense we can also view
validation as cross-checking a program against a set of partial (weaker) specifications. Verification
would typically give a stronger result; but on the other hand validation may be cheaper —in
practice it is a trade-off that a designer has to decide. We will also call specifications used for
validation: validation properties. For example, consider the program:

SUC(x : int) { return: x+1 }
8.4. VALIDATION

The following would make a good functional specification for SUC:

\[
\{ \ast \text{true} \ast \} \text{SUC}(x) \{ \ast \text{return } = x + 1 \ast \}
\]

But this specification:

\[
\{ \ast \text{true} \ast \} \text{SUC}(x) \{ \ast \text{true} \ast \}
\]

is not very useful for describing SUC’s functionality. However, it still states that SUC terminates; any correct implementation of SUC clearly must satisfy the specification. This is a typical example of a validation property.

It should be noted that the border between a functional specification and a validation property is a grey area. Consider this specification:

\[
\{ \ast \text{true} \ast \} \text{SUC}(x) \{ \ast \text{true} \ast \}
\]

It only partially describes what SUC does. It can be used as a validation property. Still, we can choose to use it as a functional specification. Though it is indeed a rather weak functional specification.

An interesting case of validation is that of ADT. Because an ADT can only be accessed through its operations, it makes the operations themselves suitable for capturing sophisticated validation properties. This leads to a possible application in unit testing and run-time checking. As an example, consider again the Queue ADT (Figure 8.1). The following specification says that the queue’s size will increase by 1 after a \( qin \) operation:

\[
\{ \ast \text{COUNT q = N } \ast \} \text{qin}(x) \{ \ast \text{COUNT q = N+1 } \ast \}
\]

As opposed to \( qin \)’s specification in Queue’s SPEC-section, the above specification only partially describes what \( qin \) does. We may want to have it as a validation property. Another way to express (8.1) is as follows, which relies on \( \text{size} \) to compute the queue’s actual size:

\[
\{ \ast \text{true} \ast \} \text{m} := \text{q.size}(); \text{qin}(x); \text{n} := \text{q.size}() \{ \ast n = m+1 \ast \}
\]

Notice that the assertions in (8.2) are ordinary program expressions, whereas in (8.1) the function \( \text{COUNT} \) is used, which is only available in Form. Suppose we decide to first test a specification. Testing means running a program on some concrete inputs, and checks if it behaves according to its specification. Although testing is only partial, in practice it makes sense to combine testing with validation or verification. For example we may want to delay validation until ordinary testing cannot reveal further error.

During a test, an assertion which is also a program expression can be checked by evaluating the expression and checking if it results in \( \text{true} \). A tool like JUnit [1] can automate testing and assertion checks during the test.

In contrast, checking an arbitrary Form formula as an assertion is more complicated. For example, to check the assertion \( \text{COUNT q = N+1} \) in (8.1) at the run-time would require us to specify an implementation of \( \text{COUNT} \). The same applies for all other operators and functions that are only available in Form, e.g. \( \forall, \in, \text{or } ++ \).

In some specification languages, e.g. JML [18], it is also allowed to have calls to functional programs (programs without any side effect) to appear in assertions. JML also allows a formula of the form \( \text{old e} \) to appear in the post-condition of a program; it refers to the value of \( e \) evaluated on the state when the program is called. With such extensions we can equivalently express (8.2) as follows:

\[
\{ \ast \text{true} \ast \} \text{qin}(x) \{ \ast \text{size}() = \text{old(size)}() + 1 \ast \}
\]

Strictly speaking, (8.2) specifies a call to \( \text{qin} \) when it is preceeded and followed by calls to \( \text{size} \). In contrast, (8.3) above specifies an arbitrary call to \( \text{qin} \). Except for the use of \( \text{old} \) the assertions
ADT Queue {
    model Queue = [int]

    operation ... // as before

    spec ... // as before

    prop // q below is of type Queue

    {* true *} m:=q.size(); q.qin(x); n:=q.size() {* n = m+1 *} ;

    {* true *}
    q.qin(x) ;
    s:=q.size() ;
    while 0<s do { s:=s-1; y:=q.quot() } {* y = x *}
}

Figure 8.3: Queue ADT with some validation properties.

in (8.3) are in principle ordinary program expressions. As before, they can be straightforwardly converted to code fragments that checks at the run-time if the respective assertions are not violated. However now this code can be wrapped around qin so that a call to qin can be refused if the pre-condition is not met, and reported if it violates the post-condition. JML implements this idea of run-time assertion checks.

Most of the specifications you have seen so far are of the form of:

{* P *} X:=x; Pr(x) {* Q *}

In (8.2) you see a different pattern. It uses a more complex statement to specify a property of a single program (qin). This is quite powerful. For example, we may want to have a validation property that says that any element entered into the queue can be retrieved again after sufficient number of calls to qout. This cannot be expressed with a 'plain' Hoare triple. However, in the style of (8.2) we can capture it like this:

{* true *}
q.qin(x) ;
s:=q.size() ;
while 0<s do { s:=s-1; y:=q.quot() } {* y = x *}

Figure 8.3 adds the two validation properties discussed above in the declaration of the Queue ADT; they are listed in the prop section. Like the specification in the spec sections, the specified validation properties do not refer to the implementation details of Queue.

If the specification of an ADT (the spec section) is strong enough, we may be able to prove its validation properties direction from it (the spec section). This gives an extra benefit that any correct implementation of the ADT would then automatically satisfy all its validation properties.

Validation properties can be used for a number of purposes:

1. They can complement the spec section. So they provide additional checks for an implementation.
2. Out of budget consideration, an implementator may decide to only verify validation properties.

3. They can be used to check if the specifications in the *spec* section are making sense. Note that if they contradict each other, then no implementation is possible. There is a contradiction if you can show that a value of the ADT exists that satisfies the *spec* section, but violates one of the validation properties.

4. They can be used to check if the specifications in the *spec* section are sufficient. They are sufficient if all validation properties can be proven from the *spec* section.
8.5 Exercise

1. Sketch how to proceed in order to prove that the implementation of the ADT Queue in Figure 8.2 is correct.

2. Let us write an ADT Set that will provide sets to uPL. To make it simple we will only provide sets of integers. For this exercise we also extend Form with sets, set comprehension, and the standard set operators like ∈ and /. This is just for the purpose of writing specifications. Write now your Set ADT. It should support operations: to insert an element to the set, to delete an element, and to test membership. Write also your own uPL implementation of the ADT. What are your abstraction function and data invariant?

3. Consider again the implementation of the Stack ADT in page 156. The abstraction function and the data invariant are still missing. Specify them. Does the implementation satisfy your data invariant? Give a worked out proof for the correctness of init.

4. Suppose we extend the ADT Queue in Figure 8.1 with operations isEmpty to test if the queue is empty and elem to test if a value is in the queue. Write validation properties capturing the following:
   
   (a) A queue is empty after initialization.
   (b) A queue is empty if and only if size is 0.
   (c) An empty queue does not contain any element.
   (d) A value inserted with qin will be an element of the queue.
   (e) If we insert a value x to a queue that already has n elements, after n times qout x will still be in the queue.

   Are your specifications executable?

5. Consider again the Queue ADT in Figure 8.1 and the validation properties given in Figure 8.3. Are the functional specifications given in the spec section sufficient? Use the validation properties to check it. If you find out that spec is insufficient, give a proposal to strengthen it.

   As a side note: when you strengthen a specification, there is a danger that it becomes overly strong that it cannot be implemented. A trivial example of this is if you replace all the post conditions in spec with false.
8.6 Solution

1. You have to prove that each operation \( m \) in the implementation satisfies its specification \( S \) in the ADT’s \( \text{spec} \) section. Note that \( S \) has to be translated first, so that it is expressed in terms of the concrete data structure provided by the implementation; the translation is specified by the abstraction function. You may also appeal to the specified data invariant when proving \( S \).

Next, you have to prove that the specified data invariant is uphold. It means proving that \( \text{init} \) establishes it, and that it is restored by every operation \( m \).

Finally, you have to prove that the implementation also satisfies the validation properties specified in the ADT’s \( \text{prop} \) section. Alternatively, you can prove for the ADT that \( \text{spec} \Rightarrow \text{prop} \). If the implication is valid, then any implementation that satisfies \( \text{spec} \) will automatically satisfy \( \text{prop} \).

2. See below. Note that we have assumed that \( \text{Form} \) has been extended with sets and set operators so that we can abstractly specify the \( \text{Set} \) ADT. These two things are not the same: by ”sets” we mean the ordinary mathematical notion of sets, and the \( \text{Set} \) ADT is intended to be a specification for any \( \text{uPL} \) implementation of ”sets of integers”.

\[
\text{ADT Set} \{
\text{model Set} = \text{int set}
\text{operation}
\text{init()} :()
\text{ins}(x:\text{int}) :()
\text{del}(x:\text{int}) :()
\text{union}(t:\text{Set}) :()
\text{elem}(x:\text{int}) :\text{bool}
\text{spec} // s below if of type Set
\{\ast \text{true} \ast\} s.\text{init()} \{\ast s = \emptyset \ast\}
\{\ast \text{true} \ast\} S:=s; s.\text{ins}(x) \{\ast s = \{x\} \cup S \ast\}
\{\ast \text{true} \ast\} S:=s; s.\text{del}(x) \{\ast s = S/\{x\} \ast\}
\{\ast \text{true} \ast\} S:=s; s.\text{elem}(x) \{\ast (\text{return} = (x \in s)) \land (s = S) \ast\}
\}
\]

We can implement it with array, as specified below. The code is incomplete. It is chosen to implement insertion simply by extending the array with the inserted element. It does mean that the array may contain duplicates. Deletion (the code is not shown) of \( x \) would be more complicated, as it has to remove all duplicates of \( x \). The corresponding abstraction function and data invariant are given.

\[
\text{IMPL Set} \{
\text{rep Set} = \text{record} \{\ a:\text{int}\[\] ; m,n:\text{int}\}
\text{absfn } \alpha s = \{x \mid x \in a[m...n]\}\}
\text{inv } s.m\leq s.n \ // s ranges over records of the above type
\text{impl } // s below is a record of the above type
\]
3. The abstraction function:

\[
\text{absfn } \alpha s = \text{reverse } s
\]

where \( s \) is the data structure implementing a stack. This is specified in the rep section of the implementation, which says that it \( s \) is thus a DQueue.

**Note:** applying reverse to a DQueue may seem strange. Firstly reverse expects a list, secondly DQueue is an ADT, so it can only be accessed through its own operations. However this restrictions refer to the use of ADT in a program. Abstraction function is a specification (in the sense that we never actually execute it). For a specification it is reasonable to allow a value of an ADT to be treated as if it is represented by its model. The model of a DQueue is a list, and in a specification it is reasonable that we want to refer to the reverse of such a list, as we do as we define the abstraction function above.

No special data invariant is needed. That is, the trivial invariant \( \text{true} \) will do.

For \textit{init} we have to prove this Hoare triple:

\[
\{ \text{true} \} \ s.\text{init}() \ \{ \text{reverse } s = \text{[]} \}
\]

This \textit{init} of our Stack implementation actually calls the \textit{init} of DQueue. Applying Program to Statement Reduction (Theorem 6.11.1) to the above specification reduces it to:

\[
\{ \text{true} \} \ s.\text{init}() \ \{ \text{reverse } s = \text{[]} \}
\]

However, this time the \textit{init} is that of the DQueue. The specification of this ADT provides a black box specification for its \textit{init}. After applying the Black Box Reduction (Theorem 7.2.2), it comes down to proving that the pre-condition \( \text{true} \) (in the specification above) implies:

\[
(s' = \text{[]}) \Rightarrow (\text{reverse } s' = \text{[]})
\]

which is quite trivial.

4. The type signatures:

\[
isEmpty():() \ \text{and} \ \text{elem}(x:\text{int}):\text{bool}
\]

The capturing of the validation properties is below; \( q \) is a variable of type QUEUE. All these specifications are executable in the sense that they can be straight forwardly converted into code fragments that check the corresponding assertions.

(a) \{ \text{true} \} \ q.\text{init}(); \ b := q.\text{isEmpty}() \ \{ \ast \ b \ast \}

(b) \{ \text{true} \} \ b := q.\text{isEmpty}(); \ n := q.\text{size}() \ \{ \ast \ b = (n = 0) \ast \}

(c) \{ \text{true} \} \ b := q.\text{isEmpty}(); \ i := q.\text{elem}(x) \ \{ \ast \ b \Rightarrow \neg i \ast \}

(d) \{ \text{true} \} \ q.\text{qin}(x); \ i := q.\text{elem}(x) \ \{ \ast \ i \ast \}
8.6. SOLUTION

(c) \{\textbf{* true *}\}
\begin{verbatim}
s := q.size() ; // size before insertion
q.qin(x) ;
r := true; // variable to accumulate result
b := true; // just a help variable
while s > 0 do {
    s := s - 1;
    q.quot(); // pop an element out
    b := q.elem(x); // check again if x is still in the queue
    r := r \land b // accumulate; if b ever false, r will remain false
}
\{\textbf{* r *}\}
\end{verbatim}

5. You will find out that proving e.g. the first validation property from spec is problematical. The problem is in the specification of \texttt{size} is incomplete. It fully specifies the returned value, but does not specify the final value of \texttt{q}. By agreement \texttt{q} is passed implicitly as an OUT-parameter, so we have to explicitly add in the post-condition that \texttt{size} does not alter \texttt{q}. So the specification has to be strengthened to:

\{\textbf{* true *}\} \texttt{Q := q; q.size()} \{\textbf{* (return = COUNT q) \land (q = Q) *}\}
At some point we often want to optimize our program, usually by tweaking it at some points. Unfortunately, changing a program invalidates its correctness proof. Since constructing a proof is expensive, we prefer not to redo the proof from scratch. This chapter will discuss an incremental proof approach, where we will try to reuse old results. A potentially interesting application of this is to verify systematic optimization e.g. as done by compilers, or by a code refactoring tool. However that is beyond our scope; here we just introduce you to the basic concepts of incremental proofs.

Here are some other applications that we can mention. It has been found useful for the verification of complicated algorithms [27]. First, one builds a base algorithm, which is simple and easy to prove. Subsequently, it is refined in several steps to get to its final version. Each refinement is justified with an incremental proof. B method [2] uses the same idea to incrementally design programs. One starts with an abstract model of a program, and in several steps incrementally refines it until a model is obtained which is concrete enough to be compiled to an executable code. Incremental proof has also been suggested for speeding-up re-verification of integrated circuits [25].

More formally, we can express an incremental proof strategy as an inference rule. Let $T$ be a transformation that changes a given program and its specification to a new one; we are looking for a rule of this form:

$$
\begin{array}{c}
\{ * \ P * \} \ S \ \{ * \ Q * \} \\
\text{some conditions}
\end{array}
\frac{T(\{ * \ P * \} \ S \ \{ * \ Q * \})}
$$

So, from the validity of the old specification $\{ * \ P * \} \ S \ \{ * \ Q * \}$ we can infer the validity of the new specification just by proving the conditions! In the sequel we will discuss some simple examples of incremental proof strategies. For a more comprehensive discussion you can try [20].
9.1 Splitting Specifications

Rules like the Hoare triples Conjunction and Disjunction (Rules 6.1.8 and 6.1.7) allow us to split a specification into a set $V$ of 'smaller' specifications, such that the validity of the smaller specifications implies the original one. This has the benefit that if we later change one of the specifications in $V$, we do not, as long as the code of the program being specified is unchanged, invalidate the others. So, only the changed specification need to be proven anew. We will discuss some examples.

Multiple Functionalities

Sometimes you have a program with a conjunctive post-condition. That is, a post-condition of the form $Q_1 \land \ldots \land Q_k$. This is typically the case when the program has multiple functionalities, with each $Q_i$ specifying one functionality. We show below a simplified variant of the Hoare triples Conjunction Rule:

$$\{P \} \quad S \quad \{Q_1 \}$$
$$\{P \} \quad S \quad \{Q_2 \}$$
$$\{P \} \quad S \quad \{Q_1 \land Q_2 \}$$

For example, consider a program $\text{reset}(x)$ that resets $x$ to some non-negative value and returns $x$'s old value:

$$\{\text{true} \} \quad \text{X} := x; \quad \text{reset} (\text{OUT x : int}) \quad \{x \geq 0 \land (\text{return} = X) \} \quad (9.1)$$

Suppose that in the implementation $\text{reset}$ just sets $x$ to 0. Now suppose that later the programmer decides that the $x \geq 0$ part of the post-condition is not good enough. He wants to change it to a stronger one below:

$$\{\text{true} \} \quad \text{X} := x; \quad \text{reset} (\text{OUT x : int}) \quad \{(x = 0) \land (\text{return} = X) \} \quad (9.2)$$

Since the new post-condition is stronger, it cannot be inferred from the old one (9.1); so the programmer has to redo the proof.

Alternatively, we could have used the Conjunction Rule to 'split' (9.1) over the two 'functionalties'. That is, we require these two specifications instead:

$$\{\text{true} \} \quad \text{reset} (\text{OUT x : int}) \quad \{x \geq 0 \} \quad (9.3)$$
$$\{\text{true} \} \quad \text{X} := x; \quad \text{reset} (\text{OUT x : int}) \quad \{\text{return} = X \} \quad (9.4)$$

Each of the above specifications is indeed partial. But together they are sufficient, because by the Conjunction Rule they imply the original specification (9.1).

If we later decide to change the post-condition of one partial specification, as long as we do not change the code of $\text{reset}$, it will not invalidate the others. In the case above, the programmer wants to change (9.3) to:

$$\{\text{true} \} \quad \text{X} := x; \quad \text{reset} (\text{OUT x : int}) \quad \{x = 0 \} \quad (9.5)$$

This change does not invalidate (9.4). Of course, (9.5) still has to be proven.

The original Conjunction Rule (Rule 6.1.8) allows you to alter the pre-condition as well. Below we also give another variant of the rule, which is perhaps more convenient to use. It follows from Rule 6.1.8 and Pre-condition Strengthening (Rule 6.1.6).

Rule 9.1.1 : Conjunction of Hoare Triples

$$\vdash P \Rightarrow P_1 \land P_2$$

$$\{P_1 \} \quad S \quad \{Q_1 \}$$

$$\{P_2 \} \quad S \quad \{Q_2 \}$$

$$\{P \} \quad S \quad \{Q_1 \land Q_2 \}$$
9.2. SUPERPOSITION

Multiple Scenarios

A conjunctive post-condition can also be used to capture different 'scenarios' under which a program is expected to operate. The program is to behave differently under different scenarios. For example, consider a program \texttt{xdiv} that returns the result of \( x \div y \) if \( y \) is non-zero, else it just returns a non-positive value. It can be specified as follows:

\[
\{ \ast \text{true} \ast \} \\
\text{xdiv}(x, y) \\
\{ \ast (y \neq 0) \Rightarrow (\text{return} = x \div y) \ast \} \\
\{ \ast (y = 0) \Rightarrow \text{return} \leq 0 \ast \}
\] (9.6)

Using the Conjunction Rule we can again split the specification to get a separate specification for each scenario:

\[
\{ \ast \text{true} \ast \} \ \text{xdiv}(x, y) \ \{ \ast (y \neq 0) \Rightarrow (\text{return} = x \div y) \ast \} \\
\{ \ast \text{true} \ast \} \ \text{xdiv}(x, y) \ \{ \ast (y = 0) \Rightarrow \text{return} \leq 0 \ast \}
\] (9.7) (9.8)

Note that since \( x \) and \( y \) are passed by value, their occurrences in the post-condition refer to their values when \texttt{xdiv} is called. So we can just as well express (9.7) and (9.8) as:

\[
\{ \ast y \neq 0 \ast \} \ \text{xdiv}(x, y) \ \{ \ast \text{return} = x \div y \ast \} \\
\{ \ast y = 0 \ast \} \ \text{xdiv}(x, y) \ \{ \ast \text{return} \leq 0 \ast \}
\] (9.9) (9.10)

If we later decide to change the the pre- or post-condition of one scenario, e.g. we change (9.10) so that 0 is returned, rather than allowing an non-positive integer, it will not affect the correctness of other scenarios.

9.2 Superposition

A simple way to extend a program is by introducing fresh variables and adding assignments to the new variables. A special case of such an extension is called superposition. In superposition we are not allowed to use the new variables in the way that would change the program’s old behavior\(^1\). For example, extending the guard of a loop in the old program with the new variables is not allowed. Superposition obviously preserves the validity of the program’s old specification. To handle it formally we first need to introduce some notions and notations.

Refinement and Equivalence

Let us write \( S \sqsubseteq T \) to mean that \( T \) satisfies all Hoare triples satisfied by \( S \). It implies that \( T \) implements all \( S \)'s functionalities. \( T \) can potentially do 'more': it may terminate when \( S \) does not, and it may be more deterministic than \( S \). In any case, in terms of functionality it follows that \( T \) can be safely used to replace \( S \), so we also say that \( T \) is a refinement of \( S \). The other way around is not always true. That is, we cannot always replace \( T \) with \( S \), since, for example, \( S \) may not terminate when \( T \) does.

Note that the above notion of refinement is only defined with respect to Hoare triples. In particular, the definition does not capture aspects such as performance. So, under the above definition \( S \sqsubseteq T \) does not necessarily mean that \( T \) will be faster than \( S \). Theoretically, you can strengthen the definition to capture more aspects, but here we will keep to the above definition.

\(^1\)The idea is the same as with adding auxiliary variables, except that variables added in superposition are intended to be actual program variables, whereas auxilliary variables are only introduced for the purpose of specifying the program.
If $S$ and $T$ refine each other, they can be used to replace each other. So, in this sense they are equivalent. We will denote it with $S \equiv T$. For example:

$x := x + 1; x := x - 1 \equiv \text{skip}$

The above definition of refinement is too strong however. It does not actually allow us to replace $x := x + 1$ with, for example $x, y := x + 1, 0$, where $y$ is a fresh variable. Let us weaken it a bit. We parameterize it with a set of program variables $V$; we now write $P \sqsubseteq_V Q$ to mean that $Q$ refines $P$, but only relative to what they do to the variables in $V$. Formally this is defined below. If $P$ is a predicate, let $\text{free}(P)$ denote the set of all free variables in $P$.

**Definition 9.2.1 : Refinement**

$$S \sqsubseteq_V T = (\forall P, Q : \text{free}(P) \subseteq V \land \text{free}(Q) \subseteq V : \{\ast P \ast\} S \{\ast Q \ast\} \Rightarrow \{\ast P \ast\} T \{\ast Q \ast\})$$

For example, this is valid:

$x := 0 \sqsubseteq_{\{x\}} x, y := 0, 0$

However, this is not: $x := 0 \sqsubseteq_{\{x, y\}} x, y := 0, 0$. We can now define the corresponding notion of equivalence:

$$S \equiv_V T = S \sqsubseteq_V T \land T \sqsubseteq_V S$$

So, if $S \equiv_V T$ then both statements will behave the same way with respect to the variables in $V$, though they may behave differently with respect to other variables. This can be captured by the following rule, which follows straightforwardly from the definition of $\equiv_V$ and $\sqsubseteq_V$:

**Theorem 9.2.2 :**

If $\text{free}(P) \subseteq V$ and $\text{free}(Q) \subseteq V$ and $S \equiv_V T$, we have:

$$\{\ast P \ast\} S \{\ast Q \ast\} = \{\ast P \ast\} T \{\ast Q \ast\}$$

If two statements are equivalent with respect to some set $V$ of variables, they are also equivalent with respect to any subset of $V$:

**Theorem 9.2.3 :**

$$S \equiv_V T \land U \subseteq V \Rightarrow S \equiv_U T$$

Let $\text{var}(S)$ denote the set of all variables used in $S$. Superpositioning a statement $S$ produces a statement $S'$ which is equivalent to $S$ with respect to $\text{var}(S)$. Formally:

**Theorem 9.2.4 : Equivalence by Superposition**

Let $S'$ be obtained from superpositioning $S$ with assignments to fresh variables. Let $Z$ be the set of these fresh variables. For any $V$ such that $V \cap Z = \emptyset$, we have:

$$S \equiv_{\text{var}(S) \cup V} S'$$
Superpositioning a loop

Loop makes a quite interesting case of superposition. Consider a loop satisfying:

\[ \{ \ast I \ast \} \textbf{while } g \textbf{ do } \{ \ast Q \ast \} \]  \hspace{1cm} (9.11)

Moreover, \(I\) is an invariant and its set of free variables is assumed to be a subset of the set of variables used in the loop. Next, we decide to superposition the body \(S\) to \(S'\), and obtain a new loop:

\[ \textbf{while } g \textbf{ do } S' \]

Note that the new loop is also a superposition of the old one.

Quite trivially, (9.11) implies \( \{ \ast I \ast \} \textbf{while } g \textbf{ do } \{ \ast \text{true} \ast \} \). By Theorem 9.2.2, the new loop satisfies the same specification, which implies that the new loop terminates. Notice that you can thus infer this for free (we do not have to prove the temination of the new loop from the scratch).

Now suppose we also change the post-condition from \(Q\) to \(Q \land Q'\), where \(Q'\) specifies what we want to achieve with the new variables introduced in the superposition. For proving \(Q'\) we may need to strengthen the old invariant \(I\) with some \(I'\); so, taking \(I \land I'\) as the new invariant. For \(Q'\), it is now sufficient to prove that it is implied by \(I \land I' \land \neg g\), which holds when the loop terminates. However, we also have the obligation to show that \(I \land I'\) is indeed an invariant. By Theorem 9.2.2, we know that the new body \(S'\) will maintain old invariant \(I\). So, it is sufficient to prove \(\{ \ast I \land I' \land g \ast \} \textit{body} \{ \ast \text{true} \ast \}\). Formally this is captured by the rule below:

\textbf{Rule 9.2.5 : Loop Superposition}

Let \(S'\) be obtained from superpositioning \(S\) with assignments to fresh variables. Let \(Z\) be the set of these fresh variables. \(Z\) is assumed to have no intersection with \(\text{free}(g) \cup \text{var}(S) \cup \text{free}(I) \cup \text{free}(Q)\). We have:

\[
\begin{align*}
\{ \ast I \ast \} \textbf{while } g \textbf{ do } & \{ \ast Q \ast \} \\
\{ \ast I \land g \ast \} & \\n\{ \ast I \land I' \land g \ast \} & S' \{ \ast I' \ast \} \\
\vdash \: I \land I' \land \neg g \Rightarrow Q' \\
\Box
\end{align*}
\]

Notice that the rule requires the new loop to start from \(I \land I'\), which implies that you may have to adapt the initialization code as well.

\textbf{Example}

Below is a program for counting the number of \textit{true}'s in a Boolean array:

\[
\{ \ast 0 \leq n \ast \}
\]

\[
i,c:=n,0 \\
\textbf{while } 0<i \textbf{ do } \{ \: i:=i-1; \text{ if } b[i] \text{ then } c:=c+1 \text{ else skip } \}
\]

\[
\{ \ast c = \textit{cnt0 b n} \ast \}
\]

where:

\[
\textit{cnt0 b i} = \text{COUNT}\{v \mid v \text{ from } b[0 \ldots i], v\}
\]

In the following discussion we will just focus on the loop part. Suppose we have proven:

\[
\{ \ast I \ast \} \textbf{while } 0<i \textbf{ do } \textit{body} \{ \ast c = \textit{cnt0 b n} \ast \} \]  \hspace{1cm} (9.12)
where:

\[ I = (c = \text{cnt\,0\,b\,i}) \land 0 \leq i \leq n \]

Furthermore, the proof uses \( I \) as as the invariant. So, additionally we also have this information:

\[ \{ \ast \, I \land 0 < i \} \{ i := i - 1; \text{ if } b[i] \text{ then } c := c + 1 \text{ else skip} \} \{ \ast \, I \} \]  

(9.13)

Later, we decide to extend the loop a little bit, so that it also calculate an index to some \text{true} element in \( b \), if one exists. If there are more than one \text{true}'s, it does not matter which one is chosen. We can do this extension with superposition. We introduce a fresh variable \( k \) and extend the code (of the loop) to:

\[
\text{while } 0 < i \text{ do } \{ i := i - 1; \text{ if } b[i] \text{ then } c, k := c + 1, i \text{ else skip} \}
\]

(9.14)

First, Theorem 9.2.2 allows us to infer that the new loop satisfies the same specification as in (9.12). So it confirms that we did not do anything harmful.

Second, we now also have a new functionality. We expect the loop now to satisfy this specification:

\[ \{ \ast \, \ldots \ast \} \]

\[
\text{while } 0 < i \text{ do } \{ i := i - 1; \text{ if } b[i] \text{ then } c, k := c + 1, i \text{ else skip} \}
\]

(9.15)

The new post-condition expresses what we want to compute with the new variable \( k \).

Rather than proving the new specification from the scratch, we invoke Rule 9.2.5. The Rule’s first and second conditions are fulfilled; they are (9.12) and (9.13) respectively. The \( Q' \) in the Rule is now \( c > 0 \Rightarrow b[k] \). Quite obviously, the old invariant \( I \) is not sufficient to get \( c > 0 \Rightarrow b[k] \). We take the naive solution that simply extends \( I \) with \( Q' \) (below, it turns out to be sufficient).

So, the choice of \( I' \) in the Rule is just \( Q' \). This trivially proves the Rule’s 4th condition. So, the only remaining condition to prove is the 3rd one, which is:

\[ \{ \ast \, I \land 0 < i \land (c > 0 \Rightarrow b[k]) \} \ast \]

\[
i := i - 1; \text{ if } b[i] \text{ then } c, k := c + 1, i \text{ else skip}
\]

\[ \{ \ast \, c > 0 \Rightarrow b[k] \} \ast \]

After computing the \( \wp \), we can equivalently prove:

\[ I \land 0 < i \land (c > 0 \Rightarrow b[k]) \]

\[ \Rightarrow \]

\[ (b[i - 1] \rightarrow (c + 1 > 0 \Rightarrow b[i - 1])) \land (c > 0 \Rightarrow b[k]) \]

which is quite trivial (and does not actually depend on what \( I \) is).

So, we conclude that the new code satisfies:

\[ \{ \ast \, I \land (c > 0 \Rightarrow b[k]) \} \ast \]

\[
\text{while } 0 < i \text{ do } \{ i := i - 1; \text{ if } b[i] \text{ then } c := c + 1 \text{ else skip} \}
\]

(9.16)

\[ \{ \ast (c = \text{cnt\,0\,b\,n}) \land (c > 0 \Rightarrow b[k]) \} \ast \]

This does require that the initialization code may have to be extended to realize the extra invariant \( c > 0 \Rightarrow b[k] \). In this case this is not necessary, because \( c \) is initialized to 0, so \( c > 0 \Rightarrow b[k] \) is trivially satisfied.

Notice that in concluding (9.16), Rule 9.2.5 only generates two proof obligations —one of which is trivial.
9.3 Breaking a Loop

A loop can sometimes be optimized by breaking it if further iterations are not needed to get to the loop's specified post-condition. For example, consider this statement for computing the product of all elements in an array over the indices in \([0 \ldots n]\):

\[
\{ \ast 0 \leq n \ast \}
\]

\[
i, r := 0, 1; \text{while } i < n \text{ do } \{ \text{ } r := r \ast a[i]; i := i + 1 \} \}
\]

\[
\{ \ast r = \text{PROD}(a[0 \ldots n]) \ast \}
\]

where for a list \(s\), PROD \(s\) specifies the product of all elements of \(s\). Suppose we have proven the specification's validity. Later on, we decide to optimize the statement: we break the loop as soon as \(r\) becomes 0 (which would happen after encountering a 0 in the array) because further computation will not change the value of \(r\). So we have the following new program now:

\[
i, r := 0, 1; \text{while } i < n \land r \neq 0 \text{ do } \{ \text{ } r := r \ast a[i]; i := i + 1 \} \}
\]

Let us see if it is possible to avoid redoing the entire proof. Consider a specification:

\[
\{ \ast I \ast \} \text{ while } g \text{ do } S \{ \ast Q \ast \}
\]

Suppose we have proven it to be valid using \(I\) as the invariant. Next, we alter the loop by strengthening its guard to:

\[
\text{while } g \land h \text{ do } S
\]

We can make the following observations. Firstly, (9.17) implies that from the pre-condition \(I\) the old loop terminates. The new loop either proceeds as the old one, or stops prematurely because \(h\) becomes false before \(g\) does. In either case we conclude that it terminates.

Secondly, since the body is still the same, any invariant (that is, a predicate preserved by every iteration) of the old loop is also an invariant of the new loop. In particular, \(I\) is also an invariant of the new loop. Consider an execution of the new loop; we have concluded that it will terminate. There are two possibilities:

1. it terminates 'normally', as in the old loop, because \(g\) becomes false. Since \(I\) is also an invariant of the new loop, we know that \(I \land \neg g\) holds upon termination. It suffices now to prove that it implies the required post-condition \(Q\). However, this has been proven in the old loop, so we do not have to do it again.

2. it terminates prematurely because \(h\) becomes false. The state would satisfy \(I \land \neg h\). So, it is sufficient if we can show that \(I \land \neg h \Rightarrow Q\). This is the only additional proof obligation needed to infer:

\[
\{ \ast I \ast \} \text{ while } g \land h \text{ do } S \{ \ast Q \ast \}
\]

Formally, the above reasoning is captured by the following inference rule:

\[
\frac{\{ \ast I \ast \} \text{ while } g \text{ do } S \{ \ast Q \ast \}
\{ \ast I \land g \ast \} S \{ \ast I \ast \}
\ \vdash I \land \neg h \Rightarrow Q}{\{ \ast I \ast \} \text{ while } g \land h \text{ do } S \{ \ast Q \ast \}}
\]

The rule can be improved a bit. The third condition concerns the premature termination of the new loop. Since the first condition already implies that if \(\neg g\) holds upon termination then \(Q\) also holds, it is sufficient to prove the third condition under a further assumption that \(g\) still holds. So, we get this rule:
Rule 9.3.1: Loop Strengthening I

\[
\begin{align*}
\{ \ast I \ast \} \text{ while } g \text{ do } S \{ \ast Q \ast \} \\
\{ \ast I \land g \ast \} S \{ \ast I \ast \} \\
\vdash I \land g \land \neg h \Rightarrow Q \\
\{ \ast I \ast \} \text{ while } g \land h \text{ do } S \{ \ast Q \ast \}
\end{align*}
\]

\[\square\]

Example

Consider again the program to compute the product over an array given in page 171. It can be proven via:

\[
\{ \ast I \ast \} \text{ while } i < n \text{ do } body \{ \ast r = \text{PROD}(a[0...n]) \ast \}
\]

where:

\[
I = (r = \text{PROD}(a[0...i])) \land 0 \leq i \leq n
\]

(9.20)

Suppose we have proven (9.19) using I as an invariant; so, we also have:

\[
\{ \ast I \land i < n \ast \} \text{ body } \{ \ast I \ast \}
\]

(9.21)

We later change the loop to:

\[
\{ \ast I \ast \} \text{ while } i < n \land r \neq 0 \text{ do } body \{ \ast r = \text{PROD}(a[0...n]) \ast \}
\]

(9.22)

Rather than using the standard Loop Reduction Rule, we now use the Loop Strengthening Rule to prove this. The Rule’s first and second conditions have been proven; they are (9.19) and (9.21). So, we only have one proof obligation, namely the Rule’s third condition:

\[
I \land i < n \land (r = 0) \Rightarrow (r = \text{PROD}(a[0...n]))
\]

(9.23)

Informally, it can be argued as follows. The definition of I (9.20) implies that r contains the product up-to the index i. Since this turns out to be 0, so will the product up to the index n. We leave the formal proof to you as an exercise.

Further optimization

Consider again Rule 9.3.1. In the above example we have no problem in proving the third condition, \( I \land g \land \neg h \Rightarrow Q \), which concerns the new loop’s premature termination. In a more general case, the old invariant \( I \) may not be strong enough to prove this implication. We can solve this problem by strengthening the invariant \( I \) to \( I \land I' \) so that now the implication to \( Q \) can be proven. However this does requires that \( I \land I' \) to be an invariant (of the new loop) as well; it is sufficient to prove:

\[
\{ \ast I \land I' \land g \ast \} S \{ \ast I \land I' \ast \}
\]

Moreover, the new loop will have to start from a state satisfying \( I \land I' \). Formally, this is captured by the following rule:

Rule 9.3.2: Loop Strengthening II

\[
\begin{align*}
\{ \ast I \ast \} \text{ while } g \text{ do } S \{ \ast Q \ast \} \\
\{ \ast I \land g \ast \} S \{ \ast I \ast \} \\
\{ \ast I \land I' \land g \ast \} S \{ \ast I' \ast \} \\
\vdash I \land I' \land g \land \neg h \Rightarrow Q \\
\{ \ast I \land I' \ast \} \text{ while } g \land h \text{ do } S \{ \ast Q \ast \}
\end{align*}
\]

\[\square\]

It can be proven from the first Loop Strengthening Rule. The proof is left out as an exercise.
9.4 Exercise

1. The rule below, called Direct Termination Rule, captures the scenario when a loop directly terminates because its guard is already false in the initial state:

\[ \{* Q \land \neg g *\} \text{ while } g \text{ do } S \{* Q *\} \]

Derive the rule from the standard rule for while.

2. Consider the program below:

\[ \{* n \geq 0 *\} \]

\[
i,s:=0,0; \text{ while } i<n \text{ do } \{ s:=s+i; i:=i+1 \}\]

\[\{* s = \text{SUM}[0...n] *\}\]

The specification can be easily proven, and suppose we have done so, using this as an invariant:

\[ I = (s = \text{SUM}[0...i]) \land 0 \leq i \leq n \]

Notice that the \(i \leq n\) part of the invariant is needed so that when the loop terminates (\(i \geq n\)) we can infer that \(i=n\), and hence infer the loop’s post-condition from the \(s = \text{SUM}[0...i]\) part of \(I\).

We now weaken the pre-condition to true. So, a negative \(n\) is allowed. The claim is that the program can still realize its post-condition.

Try to first argue the claim using the standard Loop Reduction rule. Find a suitable invariant, then check if it is sufficient. Note that \(i \leq n\) is now no longer an invariant property.

The second approach is to split the specification. Use the Hoare triple disjunction property to split the specification over two cases: (1) \(0 \leq n\) holds initially, and (2) \(n<0\) holds instead.

How to proceed along this line? What do we gain?

3. The program below checks if a boolean array contains a true:

\[\{* \text{true} *\}\]

\[
i,r:=0,\text{false } ; \text{ while } i<n \text{ do } \{ r:=b[i] \lor r; i:=i+1 \}\]

\[\{* r = (\exists i: 0 \leq i < n : b[i]) *\}\]

Give an invariant that will be sufficient to prove the correctness of the loop above. You may first want to argue the correctness with respect to a stronger pre-condition \(0 \leq n\), and then consider the \(n<0\) case separately as in No. 2.

Next, we optimize the program by stopping the loop as soon as \(r\) becomes true, because further iterations will not change the value of \(r\). The new program:

\[
i,r:=0,\text{false } ; \text{ while } i<n \land \neg r \text{ do } \{ r:=b[i] \lor r; i:=i+1 \}\]

Note that the guard is now strengthened with \(\neg r\). Can we build an incremental proof for the new code? How?
4. We can optimize the second code in No. 3 a bit further by replacing the assignment $r := b[i] \lor r$ with $r := b[i]$. This is safe since $r$ is false anyway when the body is entered. In our theory, refinement is used to formally express code replacement. However, the above kind of replacement, which depends on some assumptions on the states (namely that $r$ is false at that moment), cannot be directly expressed in our notion of refinement. Can you propose an extension to it?

5. Consider again the optimized code in No. 3. We extend it further. We superposition it with a new variable $k$ so that in the end $k$ points to a true element, if such an element can be found. The code and new specification are given below:

\[
\begin{align*}
\{ \ast \text{true} \ast \} \\
i, r := 0, \text{false} ; \\
k := 0 ; \quad \text{--new} \\
\text{while } i < n \land \neg r \text{ do} \\
\{ r := b[i] \lor r ; \\
\quad \text{if } r \text{ then } k := i \text{ else skip } \quad \text{-- new} \\
i := i + 1 \} \\
\{ \ast (r = (\exists i : 0 \leq i < n : b[i]) \land (r \Rightarrow b[k] \land 0 \leq k < n)) \ast \}
\end{align*}
\]

How to prove this incrementally?
9.5. **SOLUTION**

1. It says that if the guard $g$ is false when the loop is entered, the post-condition $Q$ can be established if $Q$ already holds as the loop is entered.

   The rule can be derived from the standard **while** reduction rule. Take $Q \land \neg g$ as $I$. TC1, TC2, PIC, and PEC become trivial.

2. We want to weaken the pre-condition to **true**. To prove the new specification directly using the standard **Loop Reduction** rule, a new invariant is needed. This will do:

   $$(0 \leq n \Rightarrow I_{\text{old}}) \land (n < 0 \Rightarrow (s = 0))$$

   where $I_{\text{old}}$ is the old invariant (the one given in the exercise).

   In the second approach we first split the specification to:

   (a) \{ $0 \leq n$ \} \textit{program} \{ $s = \text{SUM}[0 \ldots n]$ \}

   (b) \{ $n < 0$ \} \textit{program} \{ $s = \text{SUM}[0 \ldots n]$ \}

   By the **Hoare Triple Disjunction** rule (Rule 6.1.7), (a) and (b) implies the correctness of the new code (with pre-condition **true**). (a) has been proven, so we do not have to do it again.

   In the (b)-scenario, the loop will actually terminates directly. The Direct Termination Rule from Exercise No 1 gives a shortcut to handle this. By the rule it is sufficient to prove:

   \{ $n < 0$ \} \textit{i, s := 0, 0} \{ $(s = \text{SUM}[0 \ldots n]) \land i \geq n$ \}

   which is quite easy to prove.

   In the first approach, we have to come up with a new invariant, which may not be obvious. Then we have to prove that it is sufficient. We will redo many steps that we already did in the old proof.

   In the second approach you reuse the old result. It still entails a new proof obligation. We were also able to identify that the new obligation concerns a direct termination scenario, which leads to substantial simplification.

3. (a) **The first code.** Split the specification over $0 \leq n$ and $n < 0$ as pre-conditions. The split can be argued as in the solution of No. 2.

   With respect to the $0 \leq n$ pre-condition, this invariant will do:

   $$I \quad = \quad (r = (3k : 0 \leq k < i)) \land 0 \leq i \leq n$$

(b) **The optimized code.**

   Let $S_i$ be the the program’s initialization code and $S_b$ be the program’s loop’s body. Let $Q$ be the program post-condition. Assume in (a) we have proven:

   (1) \{ $n \geq 0$ \} $S_i$; \textit{while} $i \leq n$ do $S_b$ \{ $Q$ \}

   using the above specified $I$ as an invariant. That means, we also have these results:

   (2) \{ $n \geq 0$ \} $S_i$ \{ $I$ \}

   (3) \{ $I$ \} \textit{while} $i < n$ do $S_b$ \{ $Q$ \}

   (4) \{ $I \land i < n$ \} $S_b$ \{ $I$ \}
This is what we have to prove:

\[ (5) \{ \ast \text{true} \ast \} \text{ } S_i \text{ while } i \lt n \land \neg r \text{ do } S_b \{ \ast Q \ast \} \]

Using the Disjunction Rule we split (5): it is sufficient to prove (6) and (7) below:

\[ (6) \{ \ast \text{n} \lt 0 \ast \} \text{ } S_i \text{ while } i \lt n \land \neg r \text{ do } S_b \{ \ast Q \ast \} \]

\[ (7) \{ \ast \text{n} \geq 0 \ast \} \text{ } S_i \text{ while } i \lt n \land \neg r \text{ do } S_b \{ \ast Q \ast \} \]

(6) is proven with the Direct Termination Rule; see the proof below:

\[
\text{PROOF BEGIN } \\
\begin{align*}
1 \{ & \text{follows from the Direct Termination Rule } \\
& \text{ } \{ \ast Q \land (i \geq n \lor r) \ast \} \text{ while } i \lt n \land \neg r \text{ do } S_b \{ \ast Q \ast \} \\
2 \{ & \text{prove this yourself } \\
& \{ \ast n \lt 0 \ast \} \text{ } S_i \{ \ast Q \land (i \geq n \lor r) \ast \} \\
3 \{ & \text{SEQ Rule on 1 and 2 } \\
& \{ \ast n \lt 0 \ast \} \text{ } S_i \text{ while } i \lt n \land \neg r \text{ do } S_b \{ \ast Q \ast \} \\
\end{align*}
\text{END }
\]

We continue with (7). It follows from (8) and (9) below:

\[ (8) \{ \ast \text{n} \geq 0 \ast \} \text{ } S_i \{ \ast I \ast \} \]

\[ (9) \{ \ast I \ast \} \text{ while } i \lt n \land \neg r \text{ do } S_b \{ \ast Q \ast \} \]

We already have (8): it is just the same as (2). (9) is proven with the Guard Strengthening Rule; see below:

\[
\text{PROOF BEGIN } \\
\begin{align*}
1 \{ & \text{prove this yourself } \\
& \text{ } I \land i \lt n \land r \Rightarrow Q \\
2 \{ & \text{Guard Strengthening Rule on (3), (4), and 1 } \\
& \{ \ast I \ast \} \text{ while } i \lt n \land \neg r \text{ do } S_b \{ \ast Q \ast \} \\
\end{align*}
\text{END }
\]

We are done.

4. One possibility is to extend $\sqsubseteq$ with another parameter. We write $P_0 \vdash S \sqsubseteq_V T$ to mean that when started in a state satisfying $P_0$, $T$ will behave, with respect to variables in $V$, as $S$. Formally:

\[
P_0 \vdash S \sqsubseteq_V T \Rightarrow
(\forall P, Q : \text{free}(P) \subseteq V \land \text{free}(Q) \subseteq V : \\
\{ \ast P_0 \land P \ast \} S \{ \ast Q \ast \} \Rightarrow \{ \ast P_0 \land P \ast \} T \{ \ast Q \ast \})
\]

You can prove this result:

\[
P_0 \Rightarrow (e = e') \quad P_0 \vdash X := e \sqsubseteq_x X := e'
\]

So, under the pre-condition $P_0$ we can safely replace the assignment $x := e$ with $x := e'$. 
5. Let \texttt{Prog}_2 be the optimized program we have in No 3, and \texttt{Prog}_3 be the new program after the superposition. Let \texttt{S}_1, \texttt{S}_b, \texttt{I}, and \texttt{Q} be defined as in solution of No. 3. Let \texttt{T}_i be the initialisation part of \texttt{Prog}_3, and \texttt{T}_b be its loop’s body.

Because of the superposition, we have this equivalence relations:

\begin{enumerate}
  
  \item \( \texttt{S}_i \equiv \texttt{T}_i \)
  
  \item \( \texttt{S}_b \equiv \texttt{T}_b \)
  
  \item \( \texttt{Prog}_2 \equiv \texttt{T}_i \)

\end{enumerate}

Let \( \texttt{Q}' = r \Rightarrow b[k] \land 0 \leq k < n \). This is what we have to prove:

\begin{enumerate}
  
  \item \( \{ \texttt{* true } \} \ \texttt{Prog}_3 \ \{ \texttt{* Q \land Q'} \} \)

\end{enumerate}

We first split it using a combination of the Disjunction and Conjunction Rules: it is sufficient to prove (5), (6), and (7) below:

\begin{enumerate}
  
  \item \( \{ \texttt{* n<0 } \} \ \texttt{Prog}_3 \ \{ \texttt{* Q } \} \)
  
  \item \( \{ \texttt{* n<0 } \} \ \texttt{Prog}_3 \ \{ \texttt{* Q'} \} \)
  
  \item \( \{ \texttt{* n\geq0 } \} \ \texttt{Prog}_3 \ \{ \texttt{* Q \land Q'} \} \)

\end{enumerate}

The same specification as (5) has been proven for \texttt{Prog}_2; see (6) in the solution of No. 3. Since (3) says that \texttt{Prog}_3 is equivalent (over relevant variables) to \texttt{Prog}_2, it follows that (5) is valid.

(6) concerns a new functionality (\( \texttt{Q}' \)); it has to be proven from scratch. Do this part yourself.

To prove (7) we split it to (8) and (9) below:

\begin{enumerate}
  
  \item \( \{ \texttt{* n\geq0 } \} \ \texttt{T}_i \ \{ \texttt{* I \land I' } \} \)
  
  \item \( \{ \texttt{* I \land I' } \} \ \texttt{while i < n \land \neg r do} \ \texttt{T}_b \ \{ \texttt{* Q \land Q'} \} \)

\end{enumerate}

where \( I' \) is an extension to the old invariant \( I \). This is needed to get \( \texttt{Q}' \) in the end. We will take the \( \texttt{Q}' \) itself as \( I' \):

\( I' = \texttt{Q}' \)

We leave the proof of (8) to you.

The proof is of (9) is below:

```
PROOF
BEGIN

1 \{ Pre-condition Strengthening on (4) in solution of No. 3 \}
\{ \texttt{* I \land i < n \land \neg r} \} \ \texttt{S}_b \ \{ \texttt{* I } \} \\

2 \{ prove this yourself \}
\{ \texttt{* I \land i < n \land \neg r \land Q' } \} \ \texttt{T}_b \ \{ \texttt{* Q' } \} \\

3 \{ trivial \}
\texttt{I \land Q' \land \neg (i < n \land \neg r)} \ \Rightarrow \ \texttt{Q'}

5 \{ Loop Superposition Rule on (9) in solution of No. 3, 1,2,3 \}
\{ \texttt{* I \land Q' } \} \ \texttt{while i < n \land \neg r do} \ \texttt{T}_b \ \{ \texttt{* Q \land Q' } \}

END
```

We are done.
1. Consider the following statement to test whether an array consists of only positive element.

\[
i := n; \\
b := T; \\
\text{while } i > 0 \text{ do } \{ i := i - 1; \; b := b \land (a[i] > 0) \}
\]

Suppose \( n \geq 0 \) holds initially. Moreover, the array \( a \) contains at least one zero element in the domain \( 0 \leq j < n \). Prove that the statement will terminate with \( b \) false.

2. A non-deterministic loop is a loop with multiple bodies. The syntax is:

\[
\text{while } g_1 \text{ do } S_1 \\
\ldots \\
\text{g}_n \text{ do } S_n
\]

The loop terminates if none of the guards \((g_1 \ldots g_n)\) is true. At each iteration, if only one guard is true, then the corresponding body will be executed. If multiple guards are true, then one will be selected non-deterministically, and the corresponding body is executed.

As an example, we can use a non-deterministic loop to non-deterministically select a number from a certain range:

\[
\text{NDSELECT(READ n:int) : int} \\
\text{stop:bool ;} \\
\{ \text{i:=0;} \\
\text{stop:=F;} \\
\text{while} \\
\quad \text{¬stop} \land \; \text{i < n do i:=i+1} \\
\quad \text{¬stop do stop:=T} ; \\
\text{return i}
\}
\]
Generalize the reduction rule for the standard while loop to handle non-deterministic loops then prove the correctness of the above program.

3. The following statement is supposed to compute the sum of all elements in the array \( a \) in the domain \( 0 \leq k < n \), if started with \( n \geq 0 \). This sum will be stored in \( s \). Is the program correct (prove it)? If it is not, gives a minimum correction, and proves that it is then correct.

\[
j := n; \ i := 0; \ s := 0;
\]
\[
\text{while} \quad i < j \ \text{do} \quad \{ s := s + a[i]; \ i := i + 1 \} \quad \text{if} \quad j \geq i \ \text{do} \quad \{ j := j - 1; \ s := s + a[j] \} ;
\]

4. Write a program with the following header:

\[
\text{SumLess}(\text{READ} \ a[:int] , n,m[:int]) : \text{bool}
\]

The program checks if the sum of all elements of an array \( a \) in the domain \( 0 \leq i < n \) is less than \( m \). It is known that the array contains only positive integers, so you can stop summing if you already know that the current sum already exceeds \( m \). Give a formal specification for the program, and prove its correctness.

5. Consider the following program to compute the greatest element in a non-empty array:

\[
\{ * \ n > 0 * \} \\
\text{MaxArray(READ} \ a[:int] , n[:int]) : \text{int} \\
\quad \text{i:int;} \\
\quad \{ \quad \text{i:=n-1 ; m:=a[i] ;} \\
\quad \quad \text{while} \ i>0 \ \text{do} \\
\quad \quad \quad \{ \quad \text{i := i-1;} \\
\quad \quad \quad \quad \text{if} \ a[i]>m \ \text{then} \ m:=a[i] \ \text{else} \ \text{skip} \} ; \\
\quad \quad \text{return} \ m \\
\quad \} \\
\{ \text{*return} = \text{MAX(a[0...n])}* \}
\]

Prove the correctness of the program (by now, this should be quite standard for you).

Let us now extend the program, so that not only we know what the greatest element is, but also where it can be found in \( a \). Here is the new program:

\[
\text{MaxArray(READ} \ a[:int] , n[:int], \text{OUT} \ k[:int]) : \text{int} \\
\quad \text{i:int;} \\
\quad \{ \quad \text{i:=n-1 ; k:=i ; m:=a[i] ;} \\
\quad \quad \text{while} \ i>0 \ \text{do} \\
\quad \quad \quad \{ \quad \text{i := i-1;} \\
\quad \quad \quad \quad \text{if} \ a[i]>m \ \text{then} \ m:=a[i]; \ k:=i \} \\
\quad \quad \text{else} \ \text{skip} \\
\quad \} ; \\
\quad \text{return} \ i \\
\}
\]
Give a specification for this new program and the prove its correctness. You may want to investigate first, whether it is possible to reuse some of the proof of the old program so that you don’t have to prove the new one from scratch.

6. Here is another program to compute the greatest element of a non-empty array.

\[
\text{MaxArray} \left( \text{READ} \ a: \text{int}[], \text{READ} \ n: \text{int} \right) : \text{int}
\]

\[
i, j: \text{int};
\]

\{\text{* } n > 0 \text{*} \}

\{
  i := 0; \ j := n - 1;
  \text{while } i \neq j \text{ do }
  \begin{cases}
    \text{if } a[i] \leq a[j] \text{ then } i := i + 1 \text{ else } j := j - 1 \end{cases};
  \text{return } x[i]
\}

\{\text{* return } = \text{MAX}(a[0...n]) \text{*} \}

Prove its correctness. (Hint: it may be easier to use a tail invariant)

7. Write an (efficient) imperative program to compute \text{fusc} n, where \text{fusc} is defined below. Prove the correctness of your program.

\[
\begin{align*}
\text{fusc 0} & \quad = 0 \\
\text{fusc 1} & \quad = 1 \\
\text{fusc 2n} & \quad = \text{fusc n} \\
\text{fusc (2n+1)} & \quad = \text{fusc n} + \text{fusc (n+1)}
\end{align*}
\]

\(n\) is assumed to be \(\geq 0\).

Hint: calculate \text{fusc 78} with hand.

8. Give an \(O(\log n)\) program to compute \(\text{SUM}[x_i \mid i \text{ from } 0...n]\), where \(n \geq 0\), and then prove the correctness of your program.

9. Let us define this:

\[
\text{onlyOne } b \text{ m n } = (\exists i: m \leq i < n : b[i] \land (\forall j: m \leq j < n : b[j] \Rightarrow (i = j)))
\]

So, if \(b\) is a boolean array, \text{onlyOne } b \text{ m n} means that the array contains exactly one true element in the domain \(m \leq i < n\). The following program checks is an array satisfies this property. Prove its correctness.

\[
\text{CheckOnlyOne} \left( \text{READ} \ b: \text{int}[], \text{READ} \ n: \text{int} \right) : \text{bool}
\]

\[
i, j: \text{int};
\]

\{
  i := 0; \ j := n - 1;
  \text{while } i < j \land \neg(b[i]) \text{ do } i := i + 1
  i < j \land \neg(b[j]) \text{ do } j := i - 1 ;
  \text{return } ((i = j) \land b[i])
\}
10. Let us define this:

\[ \text{disected } b \text{ m} \text{ n} = (\exists i : m \leq i < n : \text{leftFalse } b \ 0 \ i \ \land \ \text{rightTrue } b \ i \ n) \]

where:

\[ \text{leftFalse } b \ 0 \ i = (\forall k : 0 \leq k < i : \neg (b[k])) \]

and

\[ \text{rightTrue } b \ i \ n = (\forall k : i \leq k < n : b[k]) \]

So, if \( b \) is a boolean array, \( \text{disected } b \text{ m} \text{ n} \) means that the array, in the domain \( m \leq i < n \), can be split in two parts: the left part that consists only of F’s, and the right part that consists only of T’s. The following program checks is an array satisfies this property. Prove its correctness.

\[
\text{checkDisected(READ } b:\text{int[],READ n:int) : bool}
\]
\[
i,j:\text{int};
\]
\[
\{ i:=0; j:=n-1;
\}
\[
\{ \text{while } i < j \ \land \ \neg (b[i]) \ \text{do } i:=i+1
\]
\[
\{ \text{while } i < j \ \land \ b[j] \ \text{do } j:=i-1 \\
\}
\]
\[
\{ \text{return } (i=j) \}
\]

11. The function \( f \) is defined recursively as follows:

\[
f 0 = 1 \\
f (2n) = 2 \times f n , \text{ provided } n > 0 \\
f (2n + 1) = n + f (2n) , \text{ provided } n \geq 0
\]

Write an imperative implementation of \( f \) with \( O(\log n) \) run time.

12. Here is a program that will delete all 0 from an array. The program does not have to preserve the initial ordering of the elements. Give a formal specification, and prove it.

\[
\text{DEL0(a:int[])} : \text{int}
\]
\[
i:\text{int}
\]
\[
\{ i:=0
\]
\[
\{ \text{while } i < n \ \text{do}
\]
\[
\{ \text{if } a[i]=0 \ \text{then } \{ a[i]:=a[n]; n:=n-1 \} \text{ else } i:=i+1 \}
\]
\[
\{ \text{return } n \}
\]

13. We have assumed that arrays in uPL have unbounded size. In reality this is of course not the case. How can we adjust uPL logic if the arrays are now of bounded size? In particular, an expression like \( a[i] \) may cause a run time error if \( i \) turns out to be outside the domain of the array \( a \). How can we deal with this?

14. Expressions in uPL do not have side effect. In C you have expressions like \( x++ \) and \( x-- \) which will increase (decrease) the value of \( x \) after evaluating its value. These expressions have side effect, because evaluating it also changes the state of the program.

Let us assume the following simplified expression sub-language of uPL:
\[
\text{Expr ::= VariableName | Constant} \\
| \text{Expr + Expr | Expr * Expr} \\
| \text{VariableName++ | VariableName--}
\]

How can we compute the wp of an assignment now? How to handle if-then-else?

15. We extend uPL with exceptions. We introduce two new constructs:

\begin{align*}
\text{abort} \\
\text{try Statement catch Statement}
\end{align*}

If in \textbf{try} \( S_1 \) \textbf{catch} \( S_2 \) the execution of \( S_1 \) executes \textbf{abort} the program jump to the closest enclosing \textbf{catch} statement. So if \( S_1 \) does not contain further \textbf{try} statement, the control will jump to \( S_2 \).

How can we extend the logic for uPL to also handle exception?
Bibliography


∀, 13
∃, 13
\texttt{repby}, 102

Abstract Data Type, 151
abstraction, 59
abstraction function, 154
actual parameter, 134
address, 49
ADT, 151
alias, 49, 68, 134
array, 13
assertion, 9, 51
assignment
  simultant, 99
associativity, 11
auxiliary variable, 52

bar notation, 100
black box specification, 52
bound variable, 14
bug, 51

common divisor, 8
compatible, 100
conclusion, 20
COND Conversion, 31
COND Split, 31
conditional, 13
conditional rewrite, 24
conjunction rule, 166
context, 19
correctness, 51
counter example, 51
critical software, 7
critical system, 7
data invariant, 155
deterministic program, 60
distribute over, 40
distributive, 40
divisor, 8
  common, 8
  greatest common, 8
domain
  empty, 13
  full, 13
domain expression, 13
Domain Merging, 30
domain shift, 14
Domain Split, 30

EBNF, 203
  meta symbol, 203
EC, 72
eliminating ∀, 26
eliminating ∃, 27
expression, 11
  array, 13
  conditional, 13
  domain, 13
  quantified
    without domain, 13
formal parameter, 134
formal proof, 10
formula, 11
  conditional, 13
  quantified, 13
    domain of, 13
free variable, 14
fresh name, 26, 28
fresh variable, 26, 28
functional specification, 156

GCD, 8
gcd, 8
goal, 18
grammar, 203
greatest common divisor, 8
Hoare triple, 9
Hoare triples conjunction, 63
Hoare triples disjunction, 63
IC, 72
IL shape, 73
ILF shape, 73
inference rule, 20
informal proof, 10
instantiating \( \forall \), 20
introducing \( \forall \), 26
introducing \( \exists \), 26
invariant, 72
left associative, 11
list
  comprehension, 39
  condition, 39
  generator, 39
  nested, 39
  split, 39
  transformer, 39
enumeration, 38
listed array, 40
logic, 10, 17
loop reduction rule, 72
model of program, 59
non-deterministic program, 60
parameter
  actual, 134
parameter
  formal, 134
partial correctness, 52
pass by copy-restore, 49
pass by reference, 49
pass by value, 49
pointer, 49
post-condition, 9, 51
post-condition weakening, 62
pre-condition, 9, 51
pre-condition strengthening, 62
predicate, 18
predicate logic, 17
premise, 20
production rule, 203
programming logic, 17
proof
formal, 10
informal, 10
proof plan, 78
proof-context, 19
proving, 18
quantifier, 13
Range Split, 30
reference, 49
reference passing, 49
refine, 167
refinement, 167
relation, 60
rewriting, 22
right associative, 11
satisfy, 51
set interpretation of predicate, 61
shifting domain, 14
simultant assignment, 99
simultant substitution, 99
specializing \( \forall \), 20
specification, 8, 50
  black box, 52
  functional, 140
  formal, 50
  informal, 50
  language, 8
  white box, 52
specification language, 8
state, 60
state:final, 60
state:initial, 60
state:terminal, 60
stronger predicate, 18
subproof, 24
substitution, 21
  simultant, 99
superposition, 167, 168
syntax, 203
TC1, 72
TC2, 72
termination metric, 73
testing, 7
total correctness, 52
uEL, 11
universally true, 18
uPL, 47
valid, 18, 51
validation, 156
validation property, 156
INDEX

value passing, 49
verification, 8, 156

weaker predicate, 18
weakest pre-condition, 63, 64
white box specification, 52
wp, 63
APPENDIX A

Standard Rules and Facts

A.1 Predicate Logic Inference Rules

Rule A.1.1: Excluded Middle

\[ \neg \neg P \rightarrow P \]

Rule A.1.2: Modus Ponens (⇒ Elimination)

\[ P, P \rightarrow Q \rightarrow Q \]

Rule A.1.3: Contradiction

\[ \neg P \rightarrow \text{false} \]

Rule A.1.4: True Consequence

\[ P = \text{true} \rightarrow \text{true} \rightarrow \]

\[ P \rightarrow P \]

Rule A.1.5: ∧ Elimination

\[ P \land Q \rightarrow Q \]

\[ P \land Q \rightarrow P \]

Rule A.1.6: Conjunction (∧ Introduction)

\[ P \rightarrow Q \rightarrow P \land Q \]

Rule A.1.7: ∨ Elimination

\[ \neg P \rightarrow P \lor Q \rightarrow Q \]

Rule A.1.8: ∨ Introduction

1. \[ \neg P \rightarrow Q \rightarrow P \lor Q \]

2. An easier version. But it is weaker, since you may fail to prove Q without the help of \( \neg P \):

\[ Q \rightarrow P \lor Q \]

Rule A.1.9: Case Split

\[ P \rightarrow Q \rightarrow \neg P \rightarrow Q \rightarrow P \lor Q \rightarrow Q \]

Rule A.1.10: Specialization (∃-Elimination)

\[ P e \rightarrow Q e \rightarrow (\forall i : P i \rightarrow Q i) \rightarrow (\exists i : P i \rightarrow Q i) \rightarrow P e \rightarrow Q e \]

Rule A.1.11: ∃ Introduction

\[ P e \rightarrow Q e \rightarrow (\exists i : P i) \rightarrow (\exists i : P i) \rightarrow P e \land Q e \]
In the last two rules, you should be aware of conditions with regards to the substitution which is implicit in the notation \( P e \) —see Section 3.4.

## A.2  Rewriting

As with substitution, there are also similar conditions for rewriting —see Section 3.4. You should be aware of them when using the rules below.

**Rule A.2.1 : Rewrite**

\[
\begin{align*}
\frac{e_1 = e_2}{P}
\end{align*}
\]

**Rule A.2.2 : Conditional Rewrite**

\[
\begin{align*}
P \Rightarrow (e_1 = e_2) \\
P \\
Q
\end{align*}
\]

**Rule A.2.3 : Rewrite with \( \forall \)**

\[
\begin{align*}
(\forall i : P i : e_1 i = e_2 i) \\
P d \\
Q
\end{align*}
\]

**Rule A.2.4 : Rewrite with T and F**

\[
\begin{align*}
P \\
Q \\
\neg P
\end{align*}
\]

\[
\begin{align*}
Q[\text{true}\!/P] \\
Q\text{false}/P
\end{align*}
\]

### A.2.1  Rewrite with a Valid Equality

A theorem of the form:

\[
\vdash e_1 = e_2
\]

can be treated as if it is universally quantified over its free variable and is already derived in your current proof-context. Then you can use inference rules such as Rules A.2.1 - A.2.4 to perform rewrite.

Almost similarly, if we have a theorem of the form:

\[
\vdash P \Rightarrow (e_1 = e_2)
\]

stating a conditioned equality, it can be treated as if it is universally quantified over its free variable and is already derived in your current proof-context.

## A.3  Inductions

**Rule A.3.1 : Induction on Positive Integers**

\[
\begin{align*}
P 0 \\
(\forall n : n \geq 0 : P n \Rightarrow P (n + 1)) \\
(\forall n : n \geq 0 : P n)
\end{align*}
\]

where \( n \) is assumed to be of type integer.

**Rule A.3.2 : List Induction**

\[
\begin{align*}
P [] \\
(\forall x, s :: P s \Rightarrow P (x : s)) \\
(\forall s :: P s)
\end{align*}
\]
A.4 Predicate Logic Rewrite Theorems

A.4.1 Boolean Connectors

Theorem A.4.1: Basic equalities of Boolean connectors

1. $\vdash \lnot \lnot P = P$
2. $\vdash P \lor Q = Q \lor P$
3. $\vdash \text{true} \lor Q = \text{true}$
4. $\vdash \text{false} \lor Q = Q$
5. $\vdash (P \lor Q) \lor R = P \lor (Q \lor R) = P \lor Q \lor R$
6. $\vdash P \land (Q \lor R) = (P \land Q) \lor (P \land R)$
7. $\vdash P \lor (Q \land R) = (P \lor Q) \land (P \lor R)$
8. $\vdash P \land Q = Q \land P$
9. $\vdash \text{true} \land Q = Q$
10. $\vdash \text{false} \land Q = \text{false}$
11. $\vdash (P \land Q) \land R = P \land (Q \land R) = P \land Q \land R$
12. $\vdash P \Rightarrow Q = \lnot P \lor Q$
13. $\vdash \lnot (P \Rightarrow Q) = P \land \lnot Q$
14. $\vdash P \Rightarrow Q \Rightarrow R = P \Rightarrow (Q \Rightarrow R) = P \land Q \Rightarrow R$
15. $\vdash (P = Q) = (P \Rightarrow Q) \land (Q \Rightarrow P)$

Theorem A.4.2: de Morgan

1. $\vdash \lnot(P \lor Q) = \lnot P \land \lnot Q$
2. $\vdash \lnot(P \land Q) = \lnot P \lor \lnot Q$

Theorem A.4.3: Contra Position

$\vdash P \Rightarrow Q = \lnot Q \Rightarrow \lnot P$

A.4.2 Conditional

Theorem A.4.4: COND conversion

1. $\vdash P \Rightarrow (P \rightarrow e_1 | e_2 = e_1)$
2. $\vdash \lnot P \Rightarrow (P \rightarrow e_1 | e_2 = e_2)$
3. $\vdash P \Rightarrow e | e = e$
4. $\vdash f(P \rightarrow e_1 | e_2) = P \rightarrow f e_1 | f e_2$
5. $\vdash P \rightarrow e_1 | e_2 = \lnot P \rightarrow e_2 | e_1$

Theorem A.4.5: COND Split

$\vdash P \Rightarrow Q \mid R = (P \Rightarrow Q) \land (\lnot P \Rightarrow R)$

$P$, $Q$, and $R$ have to be predicates.

A.4.3 Quantification

Theorem A.4.6: Renaming bound variables

1. $\vdash (\forall i : P : Q) = (\forall i' : P[i'/i] : Q[i'/i])$
2. $\vdash (\exists i : P : Q) = (\exists i' : P[i'/i] : Q[i'/i])$

$i'$ should not occur free in $P$ and $Q$. 
Theorem A.4.7 : Negate $\forall$
\[ \vdash \neg (\forall i : P i \cdot Q i) = (\exists i : P i \cdot \neg (Q i)) \]

Theorem A.4.8 : Negate $\exists$
\[ \vdash \neg (\exists i : P i \cdot Q i) = (\forall i : P i \cdot \neg (Q i)) \]

Theorem A.4.9 : Nested quantifications
1. \[ \vdash (\forall i, j : P i \cdot j) = (\forall i : (\forall j : P i \cdot j)) \]
2. \[ \vdash (\exists i, j : P i \cdot j) = (\exists i : (\exists j : P i \cdot j)) \]

Theorem A.4.10 : Quantification over singleton domain
1. \[ \vdash (\forall i : e : P i) = P \]
2. \[ \vdash (\exists i : e : P i) = P \]

Theorem A.4.11 : Range Split
1. \[ \vdash (\forall i : P i \cdot Q i \cdot R i) = (\forall i : P i \cdot R i) \land (\forall i : Q i \cdot R i) \]
2. \[ \vdash (\exists i : P i \cdot Q i \cdot R i) = (\exists i : P i \cdot R i) \lor (\exists i : Q i \cdot R i) \]

Theorem A.4.12 : Domain split
1. \[ \vdash (\forall i : P i \cdot Q i) = (\forall i : P i) \land (\forall i : Q i) \]
2. \[ \vdash (\exists i : P i \cdot Q i) = (\exists i : P i) \lor (\exists i : Q i) \]

Theorem A.4.13 : Quantification over empty domain
1. \[ \vdash (\forall i : false : P i) = true \]
2. \[ \vdash (\exists i : false : P i) = false \]

Theorem A.4.14 : Domain shift
1. \[ \vdash (\forall i : P i \cdot Q i) = (\forall i : P \cdot Q i) \]
2. \[ \vdash (\exists i : P i \cdot Q i) = (\exists i : P \cdot Q i) \]
3. \[ \vdash (\forall i : P i \cdot Q i) = (\forall i : P i \cdot Q) \]
4. \[ \vdash (\exists i : P i \cdot Q i) = (\exists i : P i \cdot Q) \]

A.4.4 Domains

Theorem A.4.15 : $+1$ Conversion
Let $a$, $b$, and $i$ be integers.
1. \[ i \leq b = i < b + 1 \]
2. \[ a + 1 \leq i = a < i \]

Theorem A.4.16 : Domain merging
Let $a$, $b$, and $i$ be integers.
1. \[ \vdash a \leq b \Rightarrow (a \leq i < b + 1) = a \leq i < b \lor (i = b) \]
2. \[ \vdash a \leq b \Rightarrow (a \leq i < b) = a \leq i < b \lor (i = b) \]
3. \[ \vdash a \leq b \land b \leq c \Rightarrow (a \leq i < b \lor i < c) = a \leq i < c \]

Theorem A.4.17 : Empty Domain Conversion
Let $a$, $b$, and $i$ be integers.
1. \[ \vdash \neg (a \leq i < a) \]
2. \[ \vdash \neg (a \leq i < a) \]
3. \[ \vdash b \leq a \Rightarrow (a \leq i < b \lor i < c) \]
4. \[ \vdash b \leq a \Rightarrow (a \leq i < b = false) \]
A.5 Lists

Definition A.5.1: \(+\) AND \(\in\)
\[
\begin{align*}
[] + t &= t \\
(x : s) + t &= x : (s + t)
\end{align*}
\]
\[
\begin{align*}
x \in [] &= \text{false} \\
x \in (y : s) &= (x = y) \lor x \in s
\end{align*}
\]

Theorem A.5.2: Non-emptiness of List
1. \(\vdash (t \neq []) = (\exists x, s :: t = x : s)\)
2. \(\vdash (t \neq []) = (\exists s, y :: t = s + [y])\)

Definition A.5.3: SUM, COUNT, MAX, and MIN
\[
\begin{align*}
\text{SUM}[] &= 0 \\
\text{SUM} (x : s) &= x + \text{SUM} s \\
\text{COUNT}[] &= 0 \\
\text{COUNT} (x : s) &= 1 + \text{COUNT} s \\
\text{MAX} [x] &= x \\
\text{MAX} (x : s) &= x \text{max} \text{MAX} s, \text{provided } s \text{ is non-empty.} \\
\text{MIN} [x] &= x \\
\text{MIN} (x : s) &= x \text{min} \text{MIN} s, \text{provided } s \text{ is non-empty.}
\end{align*}
\]

Definition A.5.4: map and filter
\[
\begin{align*}
\text{map } f [] &= [] \\
\text{map } f (x : s) &= f \ x : \text{map } f s \\
\text{filter } p [] &= [] \\
\text{filter } p (x : s) &= (p x \rightarrow [x] | []) + + \text{filter } p s
\end{align*}
\]

Theorem A.5.5: Empty Enumeration
1. \(\vdash j \leq i \Rightarrow ([i \ldots j] = [])\)
2. \(\vdash [i \ldots i] = []\)

Theorem A.5.6: Singleton Enumeration
\(\vdash [i \ldots i + 1] = [i]\)

Theorem A.5.7: Enumeration Membership
\(\vdash j \in [i \ldots k] = i \leq j < k\)

Theorem A.5.8: Enumeration Split
1. \(i \leq j \Rightarrow ([i \ldots j + 1] = [i \ldots j] + [])(j)\)
2. \(i \leq j \leq k \Rightarrow ([i \ldots k] = [i \ldots j] + [j \ldots k])\)

Theorem A.5.9: Empty Comprehension
1. \(\vdash [e i | i \text{ from } []], P i = []\)
2. \(\vdash [e i | i \text{ from } s, \text{false}] = []\)
3. \(\vdash ([e i | i \text{ from } s, P i = []] = []) = (\forall i \in s : \neg (P i))\)

Theorem A.5.10: Singleton Comprehension
1. \[ [e_i | i \text{ from } [x]] = [e_i | i \text{ from } x] \]
2. \[ [e_i | i \text{ from } [x], P i] = P x \rightarrow [e x | []] \]

**Theorem A.5.11 : Comprehension Membership**
1. \( x \in [e_i | i \text{ from } s, P i] = (\exists i: i \in s \land P i : x = e_i) \)
2. \( x \in [i | i \text{ from } s, P i] = x \in s \land P x \)

**Theorem A.5.12 : Comprehension Split**
\( [e | i \text{ from } (s + + t), P] = [e | i \text{ from } s, P] + + [e | i \text{ from } t, P] \)

**Theorem A.5.13 : Nested Comprehension**
1. \( [e_1 i | i \text{ from } [e_2 j | j \text{ from } s, P j], Q i] = [e_1 (e_2 j) | j \text{ from } s, P j \land Q (e_2 j)] \)
2. \( [e_1 i | i \text{ from } [e_2 j | j \text{ from } s, P j]] = [e_1 (e_2 j) | j \text{ from } s, P j] \)

**Definition A.5.14 : Listing Array Elements**
Let \( s \) be a list (of integers) and \( a \) be an array.
\[ a \ s = [a[i] | i \text{ from } s] \]

**Theorem A.5.15 : Split over Array**
1. \( i \leq j \Rightarrow (a[i \ldots j + 1]) = a[i \ldots j] + + [a[j]] \)
2. \( i \leq j \leq k \Rightarrow (a[i \ldots k]) = a[i \ldots j] + + a[j \ldots k] \)

**Theorem A.5.16 : Homomorphism of List Functions**
1. \( x \in (s + + t) \quad \Rightarrow \quad x \in s \lor x \in t \)
2. \( \text{SUM } (s + + t) \quad \Rightarrow \quad \text{SUM } s + + \text{SUM } t \)
3. \( \text{COUNT } (s + + t) \quad \Rightarrow \quad \text{COUNT } s + + \text{COUNT } t \)
4. \( \text{map } f (s + + t) \quad \Rightarrow \quad \text{map } f s + + \text{map } f t \)
5. \( \text{filter } p (s + + t) \quad \Rightarrow \quad \text{filter } p s + + \text{filter } p t \)
6. \( s \neq [] \land t \neq [] \Rightarrow (\text{MAX } (s + + t) = \text{MAX } s \land \text{MAX } t) \)
7. \( s \neq [] \land t \neq [] \Rightarrow (\text{MIN } (s + + t) = \text{MIN } s \land \text{MIN } t) \)
B.1 Inference Rules

Rule B.1.1: Post-condition Weakening

\[ \vdash Q \Rightarrow Q' \]
\[ \{ * P * \} S \{ * Q * \} \]
\[ \{ * P * \} S \{ * Q' * \} \]

Rule B.1.2: Pre-condition Strengthening

\[ \vdash P \Rightarrow P' \]
\[ \{ * P' * \} S \{ * Q * \} \]
\[ \{ * P * \} S \{ * Q * \} \]

Rule B.1.3: Hoare Triple Disjunction

\[ \{ * P_1 * \} S \{ * Q_1 * \} \]
\[ \{ * P_2 * \} S \{ * Q_2 * \} \]
\[ \{ * P_1 \lor P_2 * \} S \{ * Q_1 \lor Q_2 * \} \]

Rule B.1.4: Hoare Triple Conjunction

\[ \{ * P_1 * \} S \{ * Q_1 * \} \]
\[ \{ * P_2 * \} S \{ * Q_2 * \} \]
\[ \{ * P_1 \land P_2 * \} S \{ * Q_1 \land Q_2 * \} \]

Rule B.1.5: If-Then-Else

\[ \{ * P \land g * \} S_1 \{ * Q * \} \]
\[ \{ * P \land \neg g * \} S_2 \{ * Q * \} \]
\[ \{ * P * \} \text{ if } g \text{ then } S_1 \text{ else } S_2 \{ * Q * \} \]

Rule B.1.6: SEQ

\[ \{ * P * \} S_1 \{ * P' * \} \]
\[ \{ * P' * \} S_2 \{ * Q * \} \]
\[ \{ * P * \} (S_1; S_2) \{ * Q * \} \]
Rule B.1.7 : Loop Reduction

| TC1 | \{ \* I \land g \* \} (C := m; S) \{ \* m < C \* \} |
| TC2 | \models I \land g \Rightarrow m > 0 |
| IC  | \{ \* I \land g \* \} S \{ \* I \* \} |
| EC  | \models I \land \neg g \Rightarrow Q |

\( \{ \* I \* \} \text{ while } g \text{ do } S \{ \* Q \* \} \)

Rule B.1.8 : Loop Reduction

| TC1 | \models P \Rightarrow I |
| TC2 | \{ \* I \land g \* \} (C := m; S) \{ \* m < C \* \} |
| EC  | \models I \land \neg g \Rightarrow Q |
| IC  | \{ \* I \land g \* \} S \{ \* I \* \} |

\( \{ \* P \* \} \text{ while } g \text{ do } S \{ \* Q \* \} \)

Rule B.1.9 : Direct Termination

\( \{ \* Q \land \neg g \* \} \text{ while } g \text{ do } S \{ \* Q \* \} \)

B.2 Weakest Pre-condition

Definition B.2.1 : Characterization of wp

\( \models P \Rightarrow \text{wp} S Q = \{ \* P \* \} S \{ \* Q \* \} \)

Corollary B.2.2 :

\( \{ \* \text{wp} S Q \* \} S \{ \* Q \* \} \)

Theorem B.2.3 : wp Distributivity

1. \( \models \text{wp} S (Q_1 \land Q_2) = (\text{wp} S Q_1) \land (\text{wp} S Q_2) \)
2. If S is deterministic, we have:

\( \models \text{wp} S (Q_1 \lor Q_2) = (\text{wp} S Q_1) \lor (\text{wp} S Q_2) \)

Theorem B.2.4 : wp of skip

\( \text{wp} \text{skip} Q = Q \)

Theorem B.2.5 : wp of if-then-else

\( \text{wp} (\text{if } g \text{ then } S_1 \text{ else } S_2) Q = g \Rightarrow \text{wp} S_1 Q \text{ or } \text{wp} S_2 Q \)

Theorem B.2.6 : wp of Statements Sequence

\( \text{wp} (S_1; S_2) Q = \text{wp} S_1 \text{ wp} S_2 Q \)

Theorem B.2.7 : wp of Assignment

1. Assignment targeting an identifier (a variable name):

\( \text{wp} (v := E) Q = Q[E/v] \)

2. Assignment targeting an array’s element:

\( \text{wp} (a[i] := e) Q = Q[a(i \text{ repby } e)/a] \)

where \( Q[a(i \text{ repby } e)/a] \) is defined as follows:

\( Q[a(i \text{ repby } e)/a] = (i = j) \Rightarrow e \in a[j] \)
3. Assignment targeting a record's field:

\[
wp (r.fn:=e) Q = Q[r(fn \text{ repby } e)/r]
\]

where \text{repby} is defined as follows:

\[
\begin{align*}
(r(fn \text{ repby } e).fn &= e \\
(r(fn \text{ repby } e).gn &= r.gn & \text{if } fn \neq gn
\end{align*}
\]

**Theorem B.2.8**: Program to Statement Reduction

Let \(Pr\) be defined as below; \(x, y\) are its formal parameters; \(x\) is passed by-value and \(y\) by copy-restore; \(X, Y\) are auxiliary paremeters used to record \(x, y\)'s initial values. Let this be a \textit{closed} specification:

\[
\{ * P * \} \ X, Y := x, y; \ Pr(x, \text{OUT } y) \ {\text{var } z; S; \text{return }= e } \ \{ * Q * \}
\]

It is valid if and only if the following statement is valid:

\[
\{ * P * \} \ { \ { X, Y := x, y; S \} \} \ \text{return} = e \ \{ * Q[X/x] * \}
\]

### B.3 Program Call

**uPL Calling Convention**

An array parameter is passed by copy-restore unless it is marked by a READ keyword. Parameters of other types are passed by value unless they are marked by an OUT keyword. Copy-restore parameters are restored in the same order as they appear in the parameters list.

**Rule B.3.1**: Renaming

Let \(\tau, \gamma, X\) and \(Y\) be lists of distinct variables. The latter two are auxiliary variables.

\[
\begin{align*}
\{ * P \} \ & X := \tau ; \ Pr(\tau) \ \{ * Q \} \\
& \Rightarrow \{ * P(\gamma/\tau) \} \ Y := \gamma ; \ Pr(\gamma) \ \{ * Q(\gamma/\tau, Y/X) \}
\end{align*}
\]

**Rule B.3.2**: Black Box Reduction

Let \(Pr\) be a uPL program. Let \(\tau, X\), and \(\overline{\tau}\) be compatible lists of variables; variables in \(\overline{\tau}\) are fresh and distinct.

\[
\begin{align*}
\{ * P \} \ & X := \tau ; \ Pr(\tau) \ \{ * Q \} \\
& \Rightarrow \{ * P \land Q' \} \ Pr(\tau) \ \{ * R \}
\end{align*}
\]

where

\[
Q' = (Q \Rightarrow R) [\overline{\tau}/\tau] [\tau/X]
\]

and \(\overline{\tau}/\tau\) is a substitution that replaces \(x_i\) with \(x'_i\) only if \(x_i\) is a pass-by-copy-restore parameter of \(Pr\).

**Rule B.3.3**: Functional Box Reduction I

Let \(Pr\) be a functional uPL program and \(\tau\) be a list of variables.

\[
\begin{align*}
\{ * P \} \ & Pr(\tau) \ \{ * \text{return }= c \} \\
& \Rightarrow \{ * P \land R(c/r) \} \ r := Pr(\tau) \ \{ * R \}
\end{align*}
\]

**Rule B.3.4**: Functional Box Reduction II

Let \(Pr\) be a functional uPL program, \(\tau\) be a list of variables, and \(d\) be a list of expressions.

\[
\begin{align*}
\{ * P \} \ & Pr(\tau) \ \{ * \text{return }= c \} \\
& \Rightarrow \{ * P[d/\overline{\tau}] \land R(c'/r) \} \ r := Pr(d) \ \{ * R \}
\end{align*}
\]

where \(e' = e[d/\tau]\).
Rule B.3.5 : REC REDUCTION
Let $\pi$ be $Pr$’s formal parameters. Let $n$ be an integer expression.

**BC1**: $P \Rightarrow n \geq 0$

**BC2**: $\{* P \land (n = 0) *, Pr(\pi)\} \{* Q *, +\}$

**RC**: $\{* P \land n < K *, Pr(\pi)\} \{* Q *, +\}$ implies

$\{* P \land (n = K) \land K > 0 *, \overline{\pi} := \pi, Pr(\pi)\} \{* Q *, +\}$

Rule B.3.6 : REC REDUCTION II
Let $Pr$ be a functional uPL program; $\pi$ are its formal parameters, and $n \in \pi$. Let $e(K)$ denote $e[K/n]$.

**BC1**: $P \Rightarrow n \geq 0$

**BC2**: $\{* P \land (n = 0) *, Pr(\pi)\} \{* return = e(0) *, +\}$

**RC**: $\{* P \land (n = K) *, Pr(\pi)\} \{* return = e(K) *, +\}$ implies

$\{* P \land (n = K + 1) \land K \geq 0 *, Pr(\pi)\} \{* return = e(K + 1) *, +\}$

$\{* P *, +\} Pr(\pi) \{* return = e *, +\}$

B.4 Program Transformation

Definition B.4.1 : REFINEMENT

$S \subseteq V T$

$= \forall P,Q : \text{free}(P) \subseteq V \land \text{free}(Q) \subseteq V :$

$\{* P *, +\} S \{* Q *, +\} \Rightarrow \{* P *, +\} T \{* Q *, +\}$

Definition B.4.2 : EQUIVALENCE

$S \equiv V T = S \subseteq V T \land T \subseteq V S$

Theorem B.4.3 :

If $\text{free}(P) \subseteq V$ and $\text{free}(Q) \subseteq V$ and $S \equiv V T$, we have:

$\{* P *, +\} S \{* Q *, +\} = \{* P *, +\} T \{* Q *, +\}$

Theorem B.4.4 :

$S \equiv V T \land U \subseteq V \Rightarrow S \equiv U T$

Theorem B.4.5 : EQUIVALENCE BY SUPERPOSITION

Let $S'$ be obtained from superpositioning $S$ with assignments to fresh variables. Let $Z$ be the set of these fresh variables. For any $V$ such that $V \cap Z = \emptyset$, we have:

$S \equiv_{\text{var}(S) \cup V} S'$

Rule B.4.6 : LOOP SUPERPOSITION

Let $S'$ be obtained from superpositioning $S$ with assignments to fresh variables. Let $Z$ be the set of these fresh variables. $Z$ is assumed to have no intersection with $\text{free}(g) \cup \text{var}(S) \cup \text{free}(I) \cup \text{free}(Q)$. We have:

$\{* I *, +\} \text{ while } g \text{ do } S \{* Q *, +\}$

$\{* I \land g *, +\} S \{* I *, +\}$

$\{* I \land I' \land g *, +\} S' \{* I' *, +\}$

$\text{if } I \land I' \land \neg g \Rightarrow Q'$

$\{* I \land I' *, +\} \text{ while } g \text{ do } S' \{* Q \land Q' *, +\}$

Rule B.4.7 : LOOP STRENGTHENING I

$\{* I *, +\} \text{ while } g \text{ do } S \{* Q *, +\}$

$\{* I \land g *, +\} S \{* I *, +\}$

$\text{if } I \land g \land \neg h \Rightarrow Q$

$\{* I *, +\} \text{ while } g \land h \text{ do } S \{* Q *, +\}$
Rule B.4.8: Loop Strengthening II

\[
\begin{align*}
\{\ast I \ast\} & \text{ while } g \text{ do } S \{\ast Q \ast\} \\
\{\ast I \land g \ast\} & S \{\ast I \ast\} \\
\{\ast I \land I' \land g \ast\} & S \{\ast I' \ast\} \\
\models I \land I' \land g \land \neg h \Rightarrow Q \\
\{\ast I \land I' \ast\} & \text{ while } g \land h \text{ do } S \{\ast Q \ast\}
\end{align*}
\]
uPL Grammar

The grammar (syntax) of uPL is shown in Figures C.1 and C.2. The grammar is written in the EBNF notation.

EBNF

EBNF stands for Extended Backus-Naur Form. It is a widely used notation invented by John Backus and Peter Naur to precisely describe computer languages. Below is a brief summary of the notation—you can Google “EBNF” to find a more extensive introduction to it.

A language is described by a grammar, which in turns consists of production rules. For example, the following production rule is part of uPL’s grammar:

\[
\langle \text{program} \rangle ::= \langle \text{header} \rangle \langle \text{body} \rangle
\]

It tells us how to write a uPL program. It says that a program consists of a header, followed by a body.

The following production rule describes the syntax of statement in uPL:

\[
\langle \text{statement} \rangle ::= \text{skip} \mid \langle \text{assignment} \rangle \mid \langle \text{if-then-else} \rangle
\]

It says that a statement is either a skip or an assignment, or an if-then-else construct. The above rule is not complete though. In uPL we also have while-loop, program call, etc.

The symbol | above is a meta symbol; it is part of the EBNF notation for describing languages, and is not part of the language being described (in this case uPL). If \( A \) and \( B \) are two EBNF fractions describing some language constructs, then \( A \mid B \) means that in the place where it appears you can either use \( A \) or \( B \).

There are more meta symbols: *, +, and ?. If \( C \) is a construct, \( C^* \) means zero or more repetition of \( C \) is allowed; \( C^+ \) means one or more repetition of \( C \) is allowed; and \( C^? \) means zero or one use of \( C \).

For example the production rules below say that a formal-parameter-list is either empty, or a non-empty list of formal-patameters separated by commas.

\[
\langle \text{formal-parameter-list} \rangle ::= \text{(empty)} \mid \langle \text{formal-parameter} \rangle (',\langle \text{formal-parameter} \rangle)^*
\]
Binary Operators

Available binary operators are shown in the table below, listed in the decreasing order of their priority (so, the top row lists the highest priority operators). The L flag means that the corresponding operator is left-associative; R means that it is right associative; and LR means that the operator is both left and right associative. Operators with no L/R flag is not associative.

\[
\begin{align*}
\&\text{^ (L)} \\
\&\text{* (LR)} \\
\&\text{+ (LR), - (L)} \\
\&\text{mod, div, max (LR), min (LR)} \\
\&\text{<, >, \neq, \leq, \geq} \\
\&\text{\& (LR)} \\
\&\text{\vee (LR)} \\
\&= 
\end{align*}
\]
\begin{itemize}
\item (program) ::= (header) (body)
\item (header) ::= (program-name) '(' (formal-parameter-list) ')' ':' (type)
\item (formal-parameter-list) ::= (empty)
\item (formal-parameter) ::= OUT? READ? (var-decl)
\item (var-decl) ::= (var-name) (',' (var-name))? ':' (type)
\item (body) ::= ']' (locvar-decl-list) (statement-sequence) (return) '}'
\item (locvar-decl) ::= var (var-decl)
\item (locvar-decl-list) ::= (locvar-decl) (',')*
\item (return) ::= (empty) | ']' return (expr)
\item (statement) ::= skip
\item (assignment) ::= (target) := (expr)
\item (target) ::= (variable) | (array-element) | (record-element)
\item (statement-sequence) ::= (statement) (',' (statement))*
\item (if-then-else) ::= if (expr) then (statement) else (statement)
\item (while-loop) ::= while (expr) do (statement)
\item (program-call) ::= (call) | (target) := (call)
\item (call) ::= (program-name) '(' (actual-parameter-list) ')' 
\item (actual-parameter-list) ::= (empty) | (expr) (',' (expr))*
\end{itemize}

Figure C.1: The grammar of uPL: program and statement.
(type) ::= (simple-type) | (record-type) | (array-type) | (type-name)

(simple-type) ::= () | bool | int | char | string

(record-type) ::= record '{' (field-def) (',' (field-def))* '}'

(field-def) ::= (field-name) (',' (field-name))* ':' (simple-type)

(array-type) ::= ((simple-type) | (record-type))[]+

(expr) ::= (constant)
| (variable-name)
| (array-element)
| (record-element)
| ¬(expr)
| (expr) (binary-operator) (expr)
| ( expr ')'

(array-element) ::= (variable-name) ('][ expr ']')+

(record-element) ::= (expr) '.' (field-name)

---

Figure C.2: uPL types and expressions