PROVING PROGRAM CORRECTNESS

PROJECT REPORT

Submitted in partial fulfillment of the
Requirements for the degree of

BACHELOR IN ENGINEERING
In
COMPUTER ENGINEERING
By

Apurva Mehta

Department of Computer Engineering
Thadomal Shahani Engineering College
University of Mumbai
2006-2007

Table of Contents

0. Introduction

What this project is about

This project is about the design of algorithms. It is intended to be a demonstration of how one may achieve precision, clarity, discipline, and brevity, both in the process of designing algorithms, and in writing about the algorithm design process.

In general, attaining precision, clarity, discipline, and brevity in our thinking and in our communication is challenging. In order to meet this challenge, we often need to create a special purpose terminology, the use of which allows us to focus on the relevant aspects of the subject matter while suppressing irrelevant detail. This special purpose terminology is what we call a 'conceptual interface' to the subject. We believe that in meeting the aforementioned challenge, the choice of conceptual interface plays a crucial role.

The challenge of designing algorithms, and of writing about the design process, is particularly acute. This is because algorithms are very compact deposits of intellectual labor. Hence, the choice of conceptual interface is all the more crucial. For example, it is possible for even a small, but un-annotated, algorithm to be completely inscrutable to a seasoned programmer. However, such an algorithm can be made intelligible by providing appropriate annotation and by using a suitably defined programming language. These are part of our conceptual interface to algorithm design.

In this project, we choose as the base of our interface the Predicate Calculus and Predicate Transformer Semantics, as presented in [DS90]. The Predicate Transformer Semantics provides us with a method of annotation, a set of concepts, and a definition of a programming language, which form a part of our vocabulary for reasoning about algorithms. The other part is the Predicate Calculus, which is a vocabulary designed to facilitate crisp logical reasoning.

Over the years, the theory presented in [DS90] has been molded into a methodology of programming which may be used by the working programmer (in [Kal90] for instance). This includes the formulation of basic programming techniques using the theory, the introduction of additional terminology, the identification and exploitation of basic program schemes, etc.. We incorporate the elements of this methodology into our interface for designing algorithms.

Thus, this project may just as well be seen as a demonstration of the use of the above-mentioned interface to solve problems in algorithm design.

On our criteria for selecting problems

Since this project is intended to be a demonstration of a method of designing algorithms, it is desirable to focus on the design process as much as possible. Thus, the problems we we would like to solve are those which allow us to focus on the problem-solving process, rather than those which distract us with unnecessary details. It is for this reason that we choose to work with small and simple problems: they are the ones which allow us to focus on the design process without getting bogged down by unnecessary details.

On the relevance of this project to the working programmer

One may well question the relevance of such a project to the working programmer. Why, one may ask, does one spend so much time and energy on such simple problems when the problems faced during the design of sophisticated digital systems are orders of magnitude more complex? Here is our answer:

Firstly, we believe that the adoption of the algorithm design methodology presented in this project has the potential to greatly improve the quality of the algorithms a programmer is able to design, whatever the level of sophistication.

Secondly, we believe that the lessons learnt by solving simple problems well are the key to being able to solve more sophisticated problems at all. In particular, when we solve problems in this project, we are precise and detailed about the opportunities available, and the decisions made, at every turn. In addition, we try to specify the heuristics that motivate our choices wherever possible. By paying attention to these details, we develop a sensitivity to the design process that is crucial when we confront more sophisticated problems.

Finally, sophisticated digital systems are so complex that, without an adequate conceptual interface to them, we find ourselves unable to reason about them in a satisfactory manner. Thus we believe that the skill of creating and using appropriate conceptual interfaces is vital to the working programmer. This skill can be developed with practice. By presenting the careful and disciplined use of a conceptual interface to algorithm design, we have presented an example of this sort of practice.

Organization

Chapter 1 briefly introduces the conceptual interface we use for the design of our algorithms. It provides pointers into the literature where we may find more extensive coverage of the various theories employed.

Subsequently, the project consists of a series of chapters, each of which tackles a single programming problem. The reader may find it beneficial to read the chapters pairwise, since pairs of expositions are related. For instance, Chapter 2 deals with an exposition of a simple algorithm known as the Linear Search and Chapter 3 solves a problem which involves an application of the Linear Search . However, it is possible to read all the chapters independently should the reader wish to do so.

Acknowledgments

This project takes its roots in the lessons I learnt during my stay in The Netherlands during the academic year 2005/2006 . Thus I deem this to be the appropriate place to thank the people who made that trip what it was.

Thanks to Sridhar Gantimahapatruni and Mita Raval for providing the necessary financial impetus.

Thanks to Wim Feijen , Ria Dijkstra , and Hanneke Driever for making all the necessary arrangements and, perhaps more importantly, for making me feel at home.

Thanks to Tom Verhoeff who always took the time to comment on my work. His comments have been instructive.

Thanks to Wim Feijen in his capacity as my supervisor during my stay in The Netherlands . Towards the end of my stay, he repeatedly said to me: "Do simple things, but do them beautifully. Then you will find that they are not so simple." . Those words motivated me to undertake this project. The work I have put into this project has driven home their full import. I feel that I am richer for the experience.

Apart from the people involved in my stay in The Netherlands , there also others I wish to acknowledge.

I wish to thank my parents in Mumbai , Rupa and Rajiv Mehta , who have taken care of all my needs while I worked on this project. It would have been considerably harder without their contribution.

Thanks are due to the the faculty and management of Thadomal Shahani Engineering College for cooperating with me to make this project possible as part of my Bachelors degree. I would also like to thank my project guide Prof. Archana Kale for her cooperation.

Last, but certainly not the least, there is Jeremy Weissmann . He has taken the time and effort to scrutinize every page of this project, and share feedback whenever necessary. In the pursuit of simplicity and perfection, he has pushed me further than I would have dared to go myself. It has been in the process of stretching myself as much as I did that

I have learnt the most valuable lessons. This project would not be what it is without his contribution. The effort that has gone into making it would not have been as rewarding without his influence. I owe him my most sincere gratitude, and thus it is to him that I dedicate this project.

1. A Review of the Literature

In this chapter we include a brief overview of the interface we use to design our algorithms. The key components of this interface are the Predicate Calculus , the Predicate Transformer Semantics , the General Programming Techniques , and the Canonical Program Schemes .

The Predicate Calculus

We use the Predicate Calculus as presented in the first half of [DS90] to carry our proofs of correctness. This presentation of the Predicate Calculus is designed to facilitate human calculation. As a result of our use of this calculus, the proofs we present are calculational, i.e. each step of a proof is the result of uninterpreted symbol manipulation according to the rules of the calculus.

The Predicate Transformer Semantics

The theory of Predicate Transformer Semantics as presented in the second half of [DS90] is used to define a simple programming language called the Guarded Command Language . This theory is built upon the Predicate Calculus and associates each construct of the language with proof rules. We use these proof rules to guide the design of our programs.

We restrict ourselves to the use of simple constructs like the assignment, the composition, the selection, and the repetition. As we shall see, these constructs are rich enough to describe all the algorithms we wish to design.

The General Programming Techniques

In [Kal90] , we find some basic program design techniques which are phrased in terms of the Predicate Transformer Semantics . These techniques enable us to derive programs or program fragments purely by calculation and we appeal to them whenever possible.

We shall use two of these techniques in this project, viz. taking conjuncts of the post-condition as invariant and replacing a constant by a variable.

The Canonical Program Schemes

There are certain general algorithms, or algorithm schemes, which may be suitably instantiated to solve a wide class of problems. In [Kal90] these schemes are presented in terms of general functions or predicates and we exploit them by instantiating these functions according to the requirements of a particular programming problem. By using these algorithm schemes we get complete and correct programs or program fragments with very little effort.

Some of the algorithm schemes we use in this project are: the Linear Search, the Bounded Linear Search, and the Binary Search.

2. The Linear Search

Introduction

The Linear Search is a very simple algorithm for finding the first occurrence of a value in a sequence, which satisfies a particular property, given that such a value exists. We shall derive a version of the Linear Search where we find the first occurrence of the value **true** in a boolean sequence b, given that this value exists. A formal specification of the algorithm is given in the following section.

A formal specification

For a boolean sequence $b.i$, $0 \leq i$, defined on the naturals, the Linear Search may be specified as follows:

$|[$ **var** $x :$ **int** ;
$\{ Q \}$
Linear Search
$\{ R \}$
$]|$.

The precondition is:

$Q :$ $\langle \exists i : 0 \leq i : b.i \rangle$.

The postcondition is:

$R :$ $R0 \quad \wedge \quad R1 \quad \wedge \quad R2$.

Where,

$R0 :$ $0 \leq x$

$R1 :$ $b.x$

$R2 :$ $\langle \forall i : 0 \leq i \ \wedge \ i < x : \neg b.i \rangle$.

Analyzing the postcondition

Given the shape of the postcondition, and since b is arbitrary, we will need a repetitive construct to establish R. Our first task is to design an invariant and a guard for this construct, and we do so in the next section.

Designing an invariant and a guard

One common technique for designing an invariant and a guard for a repetitive construct is to split the conjuncts of the postcondition of the construct between the invariant and the negation of the guard. We will employ this technique and shall now decide where to place each of the conjuncts of the postcondition.

The conjunct $0 \leq x$ is placed in the invariant for two reasons. First, it is extremely easy to initialize: any natural value will do. Second, observing that x is an index variable of sequence b, placing the negation, ie. $x < 0$, in the guard will create out-of-bound conditions on b. This is undesirable to say the least.

We shall place the conjunct $b.x$ in the guard because, were we to place it in the invariant, initialization would be a problem.

Finally, the conjunct $\langle \forall i : \ 0 \leq i \ \wedge \ i < x : \ \neg\, b.i \ \rangle$ is placed in the invariant for two reasons. First, we may easily initialize it with $x := 0$. Second, it is generally preferable not to place quantified expressions in the guard because they are difficult to evaluate, and a guard must be evaluated at every step of the repetition.

Thus we have chosen as invariant :

$P :$ $\quad P0 \quad \wedge \quad P1$

$P0 :$ $\quad 0 \leq x$

$P1 :$ $\quad \langle \forall i : \ 0 \leq i \ \wedge \ i < x : \ \neg\, b.i \ \rangle \qquad .$

The guard is $\neg\, b.x$.

The first version of the program

And so we have derived the following program:

$|[$ **var** $x :$ **int** ;
$\quad x := 0$
$\quad \{$ Inv: $\ P \ \}$
; **do** $\ \neg\, b.x \quad \rightarrow \quad \{ \ P \ \wedge \ \neg\, b.x \ \} \ldots \{ \ P \ \}$ **od**
$\quad \{ \ P \ \wedge \ b.x \ ,$ hence $\ R \ \}$
$]|$ $\qquad .$

The next task is to fill in the dots, ie. to refine the loop body.

Refining the loop body

The statement we choose for the loop body must satisfy two properties. First, it should maintain the invariant. Second, it should modify the loop variables in some way so that the loop has a chance of terminating. Keeping these requirements in mind, we investigate a suitable refinement for the loop body.

Since x is the only loop variable, we investigate modifications to it. Given the conjunct $0 \leq x$ of the invariant, and the initial value of x, we must restrict ourselves only to increments of x. The simplest increment to x which we can think of is $x := x + 1$, and so we will explore whether this statement maintains the invariant.

The statement $x := x + 1$ straightforwardly maintains the conjunct $0 \leq x$. As for the conjunct $P1$ of the invariant, we calculate in the context of the precondition of the loop body:

$[\![$ Context: $0 \leq x \quad \wedge \quad \langle \forall i : \ 0 \leq i \ \wedge \ i < x : \ \neg\, b.i \rangle \quad \wedge \quad \neg\, b.x$

$wp.(x := x + 1).(\ \langle \forall i : \ 0 \leq i \ \wedge \ i < x : \ \neg\, b.i \rangle \ )$

$\equiv$ { Axiom of Assignment }

$\langle \forall i : \ 0 \leq i \ \wedge \ i < x + 1 : \ \neg\, b.i \ \rangle$

$\equiv$ { Split off $i = x$, $0 \leq x$ from the Context }

$\langle \forall i : \ 0 \leq i \ \wedge \ i < x : \ \neg\, b.i \ \rangle \quad \wedge \quad \neg\, b.x$

$\equiv$ { Context }

true

$]\!]$

Thus we see that the statement $x := x + 1$ satisfies both the necessary properties of the loop body: it modifies x, and maintains the invariant. Hence we choose it as our loop body.

The final program

And so we have derived the following program:

$|[$ **var** $x :$ **int** ;

$x := 0$

$\{$ Inv: P $\}$

; **do** $\neg b.x \quad \rightarrow \quad x := x + 1$ **od**

$\{$ R $\}$

$]|$.

Termination

So far so good! The program above is partially correct. Our last obligation is to prove that the loop terminates.

In order to prove that a loop terminates, we need to show that the loop variables are bounded, and that they approach their bounds with each step of the repetition. The loop invariant and guard are the context for these proofs.

Our only loop variable is x , and it is incremented at every step of the repetition. Hence we need to prove that x is bounded from above. The context for this proof is invariant P and guard $\neg b.x$.

The conjuncts $0 \leq x$ and $\neg b.x$ of the context are not useful in revealing an upper bound for x . Thus we must derive the upper bound from the conjunct $\langle \forall i : 0 \leq i \wedge i < x : \neg b.i \rangle$. Hence, we shall try to calculate the expression $x \leq X$, which denotes that X is an upper bound for x , as follows :

$\langle \forall i : 0 \leq i \ \wedge \ i < x : \neg b.i \rangle$.

$\equiv$ $\{$ Rearranging to introduce the term $x \leq$ $\}$

$\langle \forall i : 0 \leq i \ \wedge \ b.i : x \leq i \rangle$

$\Rightarrow$ $\{$ Instantiation, assuming $0 \leq X \ \wedge \ b.X$ $\}$

$x \leq X$

Thus we have proved that the arbitrary value X is the upper bound for x , provided that we can assume $0 \leq X \ \wedge \ b.X$. Fortunately, we can assume the existence of an arbitrary value satisfying this property on account of the precondition $\langle \exists i : 0 \leq i : b.i \rangle$ of the program.

And so we have proved that our program terminates and this completes our derivation of the Linear Search .

Acknowledgment

This chapter is modeled on Wim Feijen's exposition of the Linear Search as presented in [WF116] .

3. The Tale of an Eventually Periodic Sequence

Introduction

This note is devoted to a programming problem Wim Feijen shared with us about a year ago while we were in Eindhoven. Wim promised us that this problem admits a very beautiful solution. Here we present a solution which we feel lives up to that qualification.

Some terminology

Before presenting the problem we introduce some terminology.

To say that a sequence is **iteratively defined** means that each of its elements (after the first) is some fixed function of the preceding element. In symbols, to say that a sequence g is iteratively defined means that there exists some function f such that:

$$g.(n+1) \;=\; f.(g.n) \qquad \text{for all } n \qquad .$$

To say that a sequence is **periodic** means that the sequence is composed of a finite pattern (the **period**) which repeats indefinitely. In symbols, to say that g is periodic means that there exists some number p such that:

$$g.(n+p) \;=\; g.n \qquad \text{for all } n \qquad .$$

The smallest such p is called the **period length**.

To say that a sequence is **eventually periodic** means that the sequence is periodic from some point onwards. In symbols, to say that g is eventually periodic means that there exist numbers p and N such that:

$$g.(n+p) \;=\; g.n \qquad \text{for all } n \;,\quad N \le n \qquad .$$

Please note that the particular formalizations we have chosen above do not represent design decisions; they are not necessarily the properties we will use in our solution to the problem. We have included them only to sharpen and clarify the English descriptions.

The problem

Given an iteratively defined, eventually periodic sequence, we are asked to compute the number of elements in its initial non-periodic segment. In what follows, we call the given sequence g .

Observations about the iterative definition of g

We begin with two observations about the iterative definition of g .

First, since an iteratively defined sequence is generated by a function, ultimately we wish to implement g in terms of its generating function. However, while designing the program, for simplicity's sake we prefer to reason about g in the usual way, with indices. In order to reconcile these conflicting concerns, we observe that g can be straightforwardly implemented in terms of its generating function, provided that our program only makes small increments to indices. We therefore adopt this constraint, so that we can design our program entirely in terms of g , and make the desired refinements afterwards.

Second, though we are given that g is iteratively defined and eventually periodic, the postcondition mentions only periodicity. This leads us to wonder whether we can solve the problem without using the iterative definition of g . However, we cannot: Given an arbitrary eventually periodic sequence, we have to inspect infinitely many of its elements in order to determine whether a particular element is in the periodic segment or not. Therefore the iterative definition of g is a necessary precondition.

The above consideration also suggests that the iterative definition of g will allow us to draw conclusions about periodicity by inspecting only finitely many elements of g . Indeed, this is the only use we make of the iterative definition of g , and in order to prepare the reader for this use, we present in advance the only two proof steps which rely on this property. They are:

" n is in the periodic segment of g "

$\Leftarrow$ { g is iteratively defined }

" $g.n$ occurs twice in g "

and:

" m is a multiple of the period length "

$\Leftarrow$ { g is iteratively defined }

" some element of g repeats at a distance of m " .

We hope that the reader finds these proof steps unobjectionable.

The postcondition, and a coarse-grained solution

Recall that we wish to compute the number of elements in the non-periodic segment of g , so let us introduce a natural variable n for this purpose. How might we formalize the postcondition? Rather than commit ourselves prematurely to a particular formalization of periodicity, we simply let T be the number we wish to compute, and formalize the postcondition as $n = T$.

Since T is unknown, the final version of our program may not include T in any guards or assignments, and thus it is preferable to put assertions containing T in the invariant, whenever possible. For this reason, we disentangle the postcondition $n = T$, rewriting it as $n \le T \ \wedge \ T \le n$. The conjunct $n \le T$ is an ideal candidate for the invariant: it is easily initialized with $n := 0$, and its maintenance under $n := n+1$ requires preassertion $n < T$, equivalently $\neg(T \le n)$. This yields the first version of our program:

$|[$ **var** n : **nat** ;
$n := 0$
{ inv: $n \le T$ }
; **do** $\neg(T \le n)$ $\rightarrow$ $n := n+1$ **od**
{ $n \le T \ \wedge \ T \le n$, hence $n = T$ }
$]|$

This program terminates on account of the invariant $n \le T$, and is therefore totally correct. However, because T appears in the guard $\neg(T \le n)$, this program is not an acceptable solution to the problem. Thus our next aim is to refine the guard.

Refining the guard, and a more fine-grained solution

We wish to refine the guard into an expression that is easily evaluated, for instance an expression in g and the program variables. A little pondering reveals the simple equivalence:

$T \le n$
$\equiv$ { the non-periodic segment is $[0..T)$ }
" n is in the periodic segment of g " .

This rewrite eliminates T and introduces g, but the expression it yields is still not easily evaluated, and hence we refine further. The only other property left to use is that g is iteratively defined:

" n is in the periodic segment of g "

$\Leftarrow$ { g is iteratively defined }

" $g.n$ occurs twice in g "

$\Leftarrow$ { one possible formalization, assuming $1 \leq m$ }

$g.n \;=\; g.(n+m)$.

Thus we have proved:

(0) $\quad (\; T \leq n \;\Leftarrow\; g.n = g.(n+m) \;) \;\Leftarrow\; 1 \leq m$.

Expression $g.n = g.(n+m)$ is indeed easily evaluated, which suggests that we introduce a new program variable m satisfying $1 \leq m$, and consider rewriting guard $\neg(T \leq n)$ as $\neg(\, g.n = g.(n+m) \,)$.

However, this rewrite would weaken the guard, forcing us to revisit all correctness proof obligations. In the interest of simplicity we wish to rewrite the guard equivalently, and hence we investigate sufficient conditions for:

$$T \leq n \;\Rightarrow\; g.n = g.(n+m) \quad .$$

Writing this equivalently as:

" n is in the periodic segment of g "

$\Rightarrow$ { ??? }

$g.n \;=\; g.(n+m)$,

we see that a simple sufficient condition is that m is a multiple of the period length. Thus we conclude:

(1) $\quad (\; T \leq n \;\Rightarrow\; g.n = g.(n+m) \;) \;\Leftarrow\; P \sqsubseteq m$,

where P is the length of the period and $\sqsubseteq$ is the 'divides' relation.

From (0) and (1) we conclude:

(2) $\quad (\; T \leq n \;\equiv\; g.n = g.(n+m) \;) \;\Leftarrow\; 1 \leq m \;\wedge\; P \sqsubseteq m$.

Having obtained a suitable refinement for the guard, we arrive at the next version of our program:

$$
\begin{array}{l}
|[\ \textbf{var}\ n, m : \textbf{nat}\ ; \\
\quad \text{“Establish ...”} \\
\quad \{\ 1 \leq m \quad \wedge \quad P \sqsubseteq m\ \} \\
\ ; n := 0 \\
\quad \{\ \text{inv:}\ \ n \leq T\ \} \\
\ ;\ \textbf{do}\ \ \neg(\ g.n = g.(n+m)\ ) \quad \rightarrow \quad n := n+1\ \ \textbf{od} \\
\quad \{\ n = T\ \} \\
]| \qquad .
\end{array}
$$

Our next task is to refine “ Establish ... ” .

Establishing $1 \leq m \ \wedge \ P \sqsubseteq m$

A very standard way to establish a conjunction like $1 \leq m \ \wedge \ P \sqsubseteq m$ is to split the conjuncts between the invariant and the negation of the guard of a loop. The conjunct $1 \leq m$ makes an ideal invariant, leaving $\neg(P \sqsubseteq m)$ for the guard:

$$
\begin{array}{l}
\quad m := 1 \\
\quad \{\ \text{inv:}\ \ 1 \leq m\ \} \\
\ ;\ \textbf{do}\ \ \neg(P \sqsubseteq m) \quad \rightarrow \quad m := m+1\ \ \textbf{od} \\
\quad \{\ 1 \leq m \quad \wedge \quad P \sqsubseteq m\ \}
\end{array}
$$

This program fragment terminates because $m \leq P$ is a loop invariant. However, the guard $\neg(P \sqsubseteq m)$ is not easily evaluated, and hence we aim to refine it.

Refining the guard, again

As before, we wish to rewrite the guard $\neg(P \sqsubseteq m)$ in terms of g and the program variables. Towards this end we calculate:

$$
\begin{array}{ll}
 & P \sqsubseteq m \\
\equiv & \quad \{\ \text{translation}\ \}
\end{array}
$$

" m is a multiple of the period length "

$\Leftarrow$ { g is iteratively defined }

" some element of g repeats at a distance of m "

$\Leftarrow$ { one possible formalization }

$g.n \;=\; g.(n+m)$.

Thus we conclude:

(3) $P \sqsubseteq m \quad \Leftarrow \quad g.n \;=\; g.(n+m)$.

As before, we would like an equivalent guard, and so we investigate sufficient conditions for the converse of (3) :

$P \sqsubseteq m \quad \Rightarrow \quad g.n \;=\; g.(n+m)$.

But here the work has already been done for us, since (1) can be written equivalently as:

(1) $(\; P \sqsubseteq m \;\; \Rightarrow \;\; g.n \;=\; g.(n+m) \;) \quad \Leftarrow \quad T \leq n$.

Thus from (1) and (3) we may conclude:

(4) $(\; P \sqsubseteq m \;\; \equiv \;\; g.n \;=\; g.(n+m) \;) \quad \Leftarrow \quad T \leq n$.

Via (4) , we can rewrite guard $\neg(P \sqsubseteq m)$ as $\neg(\, g.n \;=\; g.(n+m)\,)$, provided we can place $P \sqsubseteq m$ in the context of $T \leq n$. We deal with this obligation in the next section.

Putting $P \sqsubseteq m$ in the context of $T \leq n$

Were we to follow the approach we took when we refined a guard earlier, we would precede " Establish ... " with assertion $T \leq n$, and then set our sights on establishing $T \leq n$. However, this would send us in circles! For a similar reason, we cannot hope to maintain $T \leq n$ as invariant, because we would still have to establish it initially.

The only other way to put $P \sqsubseteq m$ in the context of $T \leq n$ is to put $T \leq n$ into the guard. Thus the guard would become either:

$T \leq n \;\; \wedge \;\; \neg(P \sqsubseteq m)$

or:

$$\neg(\ T \le n \ \ \wedge \ \ P \sqsubseteq m\ ) \qquad .$$

We choose the latter as our guard, because, on account of invariant $1 \le m$, we have:

$$\begin{array}{ll} & T \le n \ \ \wedge \ \ P \sqsubseteq m \\ \equiv & \quad \{ \ \ 1 \le m \ , \ \ (0) \ , \ \ (1) \ , \ \ (3) \ , \text{ and predicate calculus } \} \\ & g.n \ = \ g.(n+m) \qquad . \end{array}$$

(This might be considered the fundamental theorem of iteratively defined, eventually periodic sequences.)

By changing the guard to $\neg(\ T \le n \ \ \wedge \ \ P \sqsubseteq m\ )$, we have weakened it. Thus the correctness of the postcondition is maintained —the guard was not used to prove the invariance of $1 \le m$ —, but termination is no longer guaranteed. This brings us to the final part of our argument.

Termination

We would have proven that the program terminates if we can prove that the guard $\neg(\ T \le n \ \ \wedge \ \ P \sqsubseteq m\ )$ is eventually **false** . We prove the eventual falsification of the guard as follows:

Because of assignment $m := m+1$ in the loop body, predicate $P \sqsubseteq m$ is periodically **true** , and thus we can guarantee termination provided $T \le n$ becomes stably **true** . The heuristics of Widening/Weakening suggest that this can be accomplished simply by initializing n with any natural number, and putting any increment to n in the loop; indeed, then $T \le n$ will become stably **true** after at most T iterations. (For convenience, we initialize n with $n := 0$, and increment it with $n := n+1$.)

This completes the correctness argument, and we are now ready to present the solution.

A solution in terms of g

$$
\begin{array}{l}
|[\ \textbf{var}\ n, m : \textbf{nat}\ ; \\
\quad n, m\ :=\ 0, 1 \\
\quad \{\ \text{inv:}\ \ 1 \leq m\ \} \\
;\ \textbf{do}\ \ \neg(\ g.n = g.(n+m)\ )\quad \rightarrow\quad n, m\ :=\ n+1, m+1\ \ \textbf{od} \\
\quad \{\ 1 \leq m\quad \wedge\quad P \sqsubseteq m\ \} \\
;\ n := 0 \\
;\ \textbf{do}\ \ \neg(\ g.n = g.(n+m)\ )\quad \rightarrow\quad n := n+1\ \ \textbf{od} \\
\quad \{\ n = T\ \} \\
]|\qquad .
\end{array}
$$

An implementation of g

As promised, we implement g in terms of its generating function, which we call f. For convenience we let $c = g.0$. By choosing variables x and y to take on the roles of $g.n$ and $g.(n+m)$ respectively, and by choosing the appropriate invariants for x and y, we obtain the following program:

$$
\begin{array}{l}
|[\ \textbf{var}\ x, y\ ;\ \textbf{var}\ n, m : \textbf{nat}\ ; \\
\quad n, m\ :=\ 0, 1;\ x, y\ := c, f.c \\
\quad \{\ \text{inv:}\ \ x = g.n\quad \wedge\quad y = g.(n+m)\quad \wedge\quad n+1 = m\ \} \\
;\ \textbf{do}\ \ \neg(\ x = y\ )\quad \rightarrow\quad n,\ m,\ x,\ y\ :=\ n+1,\ m+1,\ f.x,\ f.(f.y)\ \ \textbf{od} \\
\quad \{\ x = g.n\quad \wedge\quad n+1 = m\ \} \\
;\ n, x, y\ :=\ 0, c, f.x \\
\quad \{\ \text{inv:}\ \ x = g.n\ \wedge\ y = g.(n+m)\ \} \\
;\ \textbf{do}\ \ \neg(\ x = y\ )\quad \rightarrow\quad n,\ x,\ y := n+1,\ f.x,\ f.y\ \ \textbf{od} \\
\quad \{\ n = T\ \} \\
]|\qquad .
\end{array}
$$

The correctness proofs for the new assertions are left as an exercise for the industrious reader. They do not involve much more than the Axiom of Assignment.

The final program

Finally, we project this program onto the variables and statements which are relevant for the computation of the desired postcondition:

$$
\begin{array}{l}
|[\ \mathbf{var}\ x, y\ ;\ \mathbf{var}\ n : \mathbf{nat}\ ; \\
\quad x, y\ := c, f.c \\
;\ \mathbf{do}\ \ \neg(\, x = y\,) \quad \rightarrow \quad x, y\ :=\ f.x, f.(f.y)\ \ \mathbf{od} \\
;\, n, x, y\ :=\ 0, c, f.x \\
;\ \mathbf{do}\ \ \neg(\, x = y\,) \quad \rightarrow \quad n, x, y := n+1, f.x, f.y\ \ \mathbf{od} \\
\quad \{\ n = T\ \} \\
]| \qquad .
\end{array}
$$

Acknowledgment

This chapter was co-authored with Jeremy Weissmann .

4. The Bounded Linear Search

Introduction

The Bounded Linear Search is one of the most simple and widely used algorithms that we know of. One informal specification of it is "Set n to be the index of the first **true** value in the boolean array $b[0..N)$. If all values in b are **false**, set n to N". Our first step toward a solution is, as always, to provide a formal specification of the problem. A literal translation of our informal specification leads to the following.

A first formal specification

We must design a program S that satisfies:

$|[$ **con** b : **array** $[0..N)$ of **bool** ;
var n : **int** ;
$\{ Q \}$
S
$\{ R \}$
$]|$.

Where

Q : $0 \leq N$

R : $R0 \ \wedge \ R1 \ \wedge \ R2$

$R0$: $0 \leq n \leq N$

$R1$: $b.n \ \vee \ n = N$

$R2$: $\langle \forall i : \ 0 \leq i < n : \ \neg \, b.i \ \rangle$.

Analyzing the postcondition

The postcondition above reveals a potential problem: Since n is an index variable of the array $b[0..N)$ and $n = N$ is a permissible value for n, it is possible that our program may evaluate the undefined expression $b.N$.

Our solution to this problem is to design the program so that it never evaluates $b.N$.

Thus the value of $b.N$ is irrelevant to the program in execution but still remains a part of our considerations. In the parlance, we call this value a 'thought value' , and we should choose thought values in order to simplify our reasoning. Thus we investigate a suitable choice of value for $b.N$.

Choosing our thought value to yield a simplified postcondition

Assuming $b.N \equiv \textbf{true}$, we have $n = N \Rightarrow b.n$. Thus we may simplify the conjunct $R1$ of our postcondition to $b.n$, which is nice. On the other hand, choosing $b.N \equiv \textbf{false}$ does not yield a readily apparent simplification. Hence we choose **true** as our value for $b.N$.

We express this choice of thought value by strengthening our precondition with the conjunct $b.N$. Since $b.N$ is a thought value, such a move does not compromise the generality of our program. Thus our precondition becomes

$Q:$ $\quad 0 \leq N \ \wedge \ b.N \qquad .$

Our postcondition becomes

$R:$ $\quad R0 \ \wedge \ R1 \ \wedge \ R2$

$R0:$ $\quad 0 \leq n \leq N$

$R1:$ $\quad b.n$

$R2:$ $\quad \langle \forall i : \ 0 \leq i < n : \ \neg \, b.i \ \rangle \qquad .$

So far so good. We are now ready to design a program meeting this specification. As is normal, we begin our design by manipulating the postcondition.

Analyzing the new postcondition

Since the array b is arbitrary we will need a loop to establish postcondition R . We need an invariant and guard for this loop, and we shall now look to design an invariant and guard.

Designing an invariant and a guard

A standard technique for designing the invariant and guard of a loop which establishes a postcondition having the shape of R is to split the conjuncts of the postcondition over the invariant and the negation of the guard. We shall employ this technique, and shall now choose where to place each of the conjuncts of the postcondition R .

We shall select $R0$ as part of the invariant because it is easy to ensure that it is maintained by modifications to n .

Since $\neg R2$ is difficult to evaluate, its is best not chosen as our guard. Thus we are tempted to choose it as part of the invariant.

If we choose choose $\neg R1$ as the guard with $R0$ and $R2$ as the invariant, we cannot exclude the undefined expression $b.N$ from being evaluated. Thus we shall choose to place $R1$ in the invariant as well. But this causes another problem: $R0$ and $R2$ are initially established with $n := 0$ whereas $R1$ is initially established with $n := N$. There is an initialization conflict.

This is a common problem which has an equally common solution: We introduce another variable m and thus choose as invariant

$P :$ $\quad P0 \ \wedge \ P1 \ \wedge \ P2$

with

$P0 :$ $\quad 0 \leq n \leq N$

$P1 :$ $\quad b.m \quad .$

$P2 :$ $\quad \langle \forall i : \ 0 \leq i < n : \ \neg b.n \ \rangle$

With this choice of invariant, our guard is $n \neq m$.

Thus our program S takes the following form.

Program version 0

$|[$ **var** $m :$ **int** ;
$n, m \ := \ 0, N$
$\{$ Inv : $P \}$
; **do** $n \neq m \quad \rightarrow$
$\{ \ P \ \wedge \ n \neq m \ \}$
"Modify n and m "
$\{ \ P \ \}$
od
$\{ \ P \ \wedge \ n = m,$ hence $R \ \}$
$]|$

Refining "Modify n and m"

It is inevitable that the elements of b must be inspected in the body of the loop. Keeping in mind of our obligation to ensure that our program never inspects $b.N$, we now turn our attention to the two candidate elements for inspection, viz. $b.n$ and $b.m$.

We immediately discount inspecting $b.m$ on account of the initial value of m.

As for $b.n$, we must investigate the conditions under which the precondition of the loop body —$P \wedge n \neq m$— implies $n \neq N$. From the conjunct $P0$ of P we have $0 \leq n \leq N$. Thus strengthening $P0$ to

$P0: \quad 0 \leq n \wedge n \leq m \wedge m \leq N$

ensures $n \neq N$ on account of $n \neq m$.

Finally, we turn our attention to modifications of n and m. Noting that $n := n+1$ is the only reasonable modification to n, we choose to investigate it.

All the conjuncts of the invariant except $P2$ are maintained by $n := n+1$ thanks, in part, to the guard $n \neq m$. The maintenance of $P2$ requires the additional condition $\neg\, b.n$. Thus we introduce a selection command and choose $\neg\, b.n$ as the guard of $n := n+1$.

To ensure that our selection does not abort, we introduce another guarded statement with $b.n$ as the guard. On account of the guard $b.n$, we have an opportunity to decrease m to n while maintaining $P0 \wedge P1$. Thus we choose $m := n$ as our other guarded statement.

Thus we arrive at the following refinement for "Modify n and m":

```
|[
  { P ∧ n ≠ m }
  if  ¬ b.n   →   n := n + 1
  []  b.n     →   m := n
  fi
  { P }
]|
```

The program

Our final program is

$$
\begin{array}{l}
|[\ \textbf{con}\ \ b :\ \textbf{array}\ [0..N)\ \ \text{of}\ \ \textbf{bool}\ \ ; \\
\quad \textbf{var}\ \ n, m :\ \textbf{int}\ \ ; \\
\quad \{\ 0 \leq N\ \wedge\ b.N\ \} \\
\quad n, m\ :=\ 0, N \\
\quad ;\ \textbf{do}\ \ n \neq m\ \ \ \rightarrow \\
\qquad \textbf{if}\ \ \neg\, b.n\ \ \ \rightarrow\ \ \ n := n + 1 \\
\qquad [\!]\ \ b.n\ \ \ \rightarrow\ \ \ m := n \\
\qquad \textbf{fi} \\
\quad \textbf{od} \\
\quad \{\ R\ \} \\
]|\ \ \ .
\end{array}
$$

A word on termination

To prove that our program terminates, it suffices to show that the loop variables are bounded and at least one of them approaches its bound at each each step of the repetition.

From the invariant we have $0 \leq n \leq m \leq N$. The assignment $n := n + 1$ properly increases n toward its upper bound N thanks to $n \leq N$. The assignment $m := n$ properly decreases m toward its lower bound 0 thanks to $0 \leq n \leq m$ and the guard $n \neq m$. Since one of these assignments is executed at each step of the repetition, the program is guaranteed to terminate.

Acknowledgment

This chapter is modeled on Wim Feijen's exposition of the Bounded Linear Search as presented in [WF118] .

5. The Lexicographic Relationship Between Two Arrays

Introduction

This chapter is a demonstration of an application of the Bounded Linear Search .

Our task is to design a program which determines the lexicographic relationship between two integer arrays $f[0..N)$ and $g[0..N)$. In particular, using $f \prec g$ to denote that f lexicographically precedes g , our program should assign to boolean variables a , b , and c such values that:

$$(a \equiv f \prec g) \quad \wedge \quad (b \equiv f = g) \quad \wedge \quad (c \equiv g \prec f) \qquad .$$

We are free to choose a refinement for $\prec$ as we see fit.

A formal specification

We may formally specify the problem as follows:

$|[$ **con** N : **int** ; f, g : **array** $[0..N)$ of **int** ;
var a, b, c : **bool** ;
{ Q }
Lexicographic order
{ R }
$]|$.

The precondition is:

Q : **true** .

The postcondition is:

R : $(a \equiv f \prec g) \quad \wedge \quad (b \equiv f = g) \quad \wedge \quad (c \equiv g \prec f) \qquad .$

Analyzing the postcondition

Our task is to design a program fragment with postcondition R . Since we are required to establish a relationship between arbitrary arrays, we might try to design a repetitive construct having R as its postcondition. We would then look to use R to design

an invariant and guard for this repetitive construct. However, the usual techniques for designing an invariant and guard from a given postcondition do not work very well with R .

Thus we are led to try and design another type of program construct which establishes R . Since there is little to guide us in the design of such a construct, we will try to formulate some simple properties of $\prec$ in the hope that they will serve as the required guide.

A simple property of $\prec$

One simple observation related to the lexicographic order $\prec$ is: for any f and g of the same size, exactly one of $f \prec g$, $f = g$, and $g \prec f$ will be **true** . Consequently, exactly one of a , b , and c will be **true** upon termination.

Selecting a suitable construct

The observation above leads us to consider a selection statement having the following shape:

```
|[
  { ¬a ∧ ¬b ∧ ¬c }
  if  ?   →   a := true   { R }
  []  ?   →   b := true   { R }
  []  ?   →   c := true   { R }
  fi
  { R }
]|
```

We must now design the guards of this selection statement. This involves identifying the precise conditions under which each of the assignments establish R . We shall derive these conditions in the next section.

Deriving the conditions under which each of the assignments establish R

We calculate the conditions in the context of the precondition of the selection statement:

$[\![$ Context: $\quad \neg\, a \quad \wedge \quad \neg\, b \quad \wedge \quad \neg\, c$

$wp.(a := \mathbf{true}).R$

$\equiv$ { Axiom of Assignment }

$\mathbf{true} \;\equiv\; f \prec g \qquad \wedge \qquad b \;\equiv\; f = g \qquad \wedge \qquad c \;\equiv\; g \prec f$

$\equiv$ { Property of $\prec$; Predicate Calculus }

$f \prec g \quad \wedge \quad \neg\, b \quad \wedge \quad \neg\, c$

$\equiv$ { Context }

$f \prec g$

$\Leftarrow$ { Using the following —standard— specification of $\prec$ }

$0 \leq x < N \quad \wedge \quad f.x < g.x \quad \wedge \quad \langle \forall i : 0 \leq i < x : \; f.i = g.i \rangle \qquad .$

$]\!]$

In the above, we have used the following specification for $\prec$:

$$f \prec g \quad \equiv \quad \langle \exists n : \; 0 \leq n < N : \; f.n < g.n \; \wedge \; \langle \forall i : \; 0 \leq i < n : \; f.i = g.i \; \rangle \; \rangle \quad .$$

Additionally, we introduced a fresh integer program variable x . In the interests of homogeneity, we shall use this same variable in the following calculations.

In a similar vein, we calculate:

$[\![$ Context: $\quad \neg\, a \quad \wedge \quad \neg\, b \quad \wedge \quad \neg\, c$

$wp.(c := \mathbf{true}).R$

$\Leftarrow$ { As above, details left to the reader }

$0 \leq x < N \quad \wedge \quad g.x < f.x \quad \wedge \quad \langle \forall i : 0 \leq i < x : \; f.i = g.i \rangle \qquad .$

$]\!]$

Finally,

$[\![$ Context: $\quad \neg\, a \quad \wedge \quad \neg\, b \quad \wedge \quad \neg\, c$

$wp.(b := \mathbf{true}).R$

$\equiv$ { Axiom of Assignment; Predicate Calculus; Context. (As in the first calculation) }

$f = g$

$\equiv$ { Refining $f = g$ }

$\langle \forall i : 0 \leq i < N : f.i = g.i \rangle$

$\Leftarrow$ { Predicate Calculus }

$0 \leq x = N \quad \wedge \quad \langle \forall i : 0 \leq i < x : f.i = g.i \rangle$.

$\rrbracket$

Thus we have identified the conditions under which assigning the value **true** to each of a , b , and c will establish the postcondition R .

Choosing the guards

The above conditions may be satisfied by placing some of the conjuncts in the precondition of the selection statement, while choosing others as guards.

In general, it is advisable to choose the guards to be as weak as possible. Thus, we shall choose to place all similar conjuncts in the precondition of the selection statement, since this will generally lead to weaker guards. Furthermore, we will try to choose our guards in such a way that the proof of non-abortion of the selection statement is straightforward.

We now proceed to place the expressions we have derived above in either the precondition of the selection or in the guards.

The conjuncts $0 \leq x$ and $\langle \forall i : 0 \leq i < x : f.i = g.i \rangle$ appear in all the three cases, and so we choose to place them in the precondition of the selection statement.

The conjunct $x = N$ appears only once and hence is chosen as a guard for $b := \mathbf{true}$.

The expression $x < N$ is a required precondition for both $a := \mathbf{true}$ and $c := \mathbf{true}$. In the interests of choosing the guards to be as weak as possible, we shall satisfy $x < N$ by choosing the weaker $x \neq N$ as a conjunct of the guards, and by simultaneously strengthening the precondition with $x \leq N$.

Finally, $f.x < g.x$ is added as a conjunct of the guard of $a := \mathbf{true}$, and $g.x < f.x$ is added as a conjunct of the guard of $c := \mathbf{true}$.

Thus we have derived the following guarded commands:

- $x \neq N \ \wedge \ f.x < g.x \quad \rightarrow \quad a := \mathbf{true}$
- $x = N \quad \rightarrow \quad b := \mathbf{true}$
- $x \neq N \ \wedge \ g.x < f.x \quad \rightarrow \quad c := \mathbf{true}$

In order to prove that the selection statement with these guarded commands does not abort, we have to prove the disjunction of the guards. In order to make this proof, we need $x = N \ \vee \ f.x \neq g.x$ as part of the context, as the reader may verify. Thus we shall strengthen the precondition of the selection with $x = N \ \vee \ f.x \neq g.x$. Hence we arrive at the first version of our program:

Program version 0

$$
\begin{array}{l}
|[\ \textbf{var}\ x : \ \textbf{int} \\
\quad \{\ 0 \leq x \leq N \quad \wedge \quad (x = N \ \vee \ f.x \neq g.x) \quad \wedge \quad \langle \forall i : 0 \leq i < x : f.i = g.i \rangle\ \} \\
\quad \{\ \neg\, a \ \wedge \ \neg\, b \ \wedge \ \neg\, c\ \} \\
\quad \textbf{if} \ \ x \neq N \ \wedge \ f.x < g.x \quad \rightarrow \quad a := \textbf{true} \\
\quad [] \ \ x = N \quad \rightarrow \quad b := \textbf{true} \\
\quad [] \ \ x \neq N \ \wedge \ g.x < f.x \quad \rightarrow \quad c := \textbf{true} \\
\quad \textbf{fi} \\
\quad \{\ R\ \} \\
]|
\end{array}
$$

Satisfying the precondition

Our remaining task is to design program fragments meeting the precondition of the selection statement.

The conjunct $\neg\, a \ \wedge \ \neg\, b \ \wedge \ \neg\, c$ of the precondition is easily satisfied by the statement $a,\ b,\ c \ := \ \textbf{false},\ \textbf{false},\ \textbf{false}$.

Thus we turn our attention to:

$$0 \leq x \leq N \quad \wedge \quad (x = N \ \vee \ f.x \neq g.x) \quad \wedge \quad \langle \forall i : 0 \leq i < x : f.i = g.i \rangle \qquad .$$

A moments reflection reveals that this is the precise postcondition of the Bounded Linear Search ! The Bounded Linear Search has the following specification:

$|[$ **con** $b :$ **array** $[0..N)$ of **bool** ;

var $x :$ **int** ;

{ **true** }

Bounded Linear Search

$\{\ 0 \leq x \leq N \quad \wedge \quad (x = N \ \vee \ b.x) \quad \wedge \quad \langle \forall i : 0 \leq i < x : \ \neg\, b.i \rangle\ \}$

$]|$.

for a boolean function b defined on the domain $[0..N)$. For the problem at hand, we define b as

$$b.i \quad \equiv \quad f.i \neq g.i \qquad .$$

Instantiating the standard refinement of the Bounded Linear Search with b as defined above, we arrive at the final version of our program.

The final program

The final version of our program is:

$|[$ **var** $x, y :$ **int** ;

$x, y \ := \ 0, N$

; **do** $x \neq y \quad \rightarrow$

if $f.x = g.x \quad \rightarrow \quad x := x + 1$

$[]$ $f.x \neq g.x \quad \rightarrow \quad y := x$

fi

od

; $a,\ b,\ c \ :=$ **false**, **false**, **false**

if $x \neq N \ \wedge \ f.x < g.x \quad \rightarrow \quad a :=$ **true**

$[]$ $x = N \quad \rightarrow \quad b :=$ **true**

$[]$ $x \neq N \ \wedge \ g.x < f.x \quad \rightarrow \quad c :=$ **true**

fi

$]|$

6. The Binary Search

Introduction

The Binary Search is another simple, efficient, and widely used algorithm for searching. In its most common form, it is used for determining whether or not a particular value exists in a sorted array.

We will design a generalized version of the binary search, which we specify as follows: Given a function or array $f[a..b]$ of some type, with $a < b$, and such that $f.a$ and $f.b$ are in some relation to each other — which we shall denote by $a \mathcal{Z} b$ —, our algorithm must find a pair of neighboring elements of f which are in relation $\mathcal{Z}$.

Later, we will show a popular instantiation of our general solution, viz. with $x \mathcal{Z} y$ instantiated with $f.x \leq C \ \wedge \ C < f.y$, for some constant C. Coupled with the constraint that the array is ascending, this version yields the common form of the Binary Search which determines the existence of a particular value in a sorted array.

A formal specification

The Binary Search may be formally specified as follows:

$|[$ **con** $a, b :$ **int** ; **con** $f[a..b]$;
var $x :$ **int** ;
$\{ \ Q \ \}$
Binary Search
$\{ \ R \ \}$
$]|$

where

$Q :$ $\quad a < b \quad \wedge \quad a \mathcal{Z} b$

$R :$ $\quad a \leq x \quad \wedge \quad x < b \quad \wedge \quad x \mathcal{Z} (x + 1)$.

Analyzing the postcondition

Given the shape of the postcondition, and given that f is arbitrary, we will need to use a repetitive construct. The first step toward designing a repetitive construct is to design

an invariant and a guard for the construct, and that is where we turn our attention in the next section.

Designing an invariant and a guard

A common technique for designing an invariant and guard given a postcondition like ours is to divide the conjuncts of the postcondition between the invariant and the negation of the guard. We employ this technique and now consider how to divide the conjuncts.

The conjuncts $a \leq x$ and $x < b$ are preferably placed in the invariant for two reasons. First, it is easy to ensure their maintenance under modifications to x . Second, since x is an index variable of f , choosing either of their negations as the guard — $x < a$ and $b \leq x$ respectively — creates out-of-bounds conditions, which is unattractive to say the least.

The conjunct $x \mathcal{Z} (x + 1)$ is also preferably chosen as a part of the invariant. This is because choosing its negation as the guard would require us to prove that the pair $x \mathcal{Z} (x + 1)$ exists as part of our termination argument. It is more attractive to design a more refined program —with a simpler termination argument— which computes such a pair. This program will then serve as the necessary existence proof.

If we choose all the three conjuncts of the postcondition as part of the invariant, we are confronted with another problem, viz. that of initialization. In particular, the conjunct $x \mathcal{Z} (x + 1)$ is impossible to establish initially: we are designing a program to compute precisely such an x !

Fortunately, the conjunct $a \mathcal{Z} b$ of the precondition allows for a straightforward solution to this dilemma: we may simply introduce another variable y to take on the role of $x + 1$ and thus select as invariant

$P :$ $\quad P0 \quad \wedge \quad P1$

$P0 :$ $\quad a \leq x \quad \wedge \quad x < y \quad \wedge \quad y \leq b$

$P1 :$ $\quad x \mathcal{Z} y \quad .$

This invariant is initially established with $x, y := a, b$. For the guard, we choose $x + 1 \neq y$.

Program version 0

We are thus headed for a program of the following form:

$|[$ **var** $x, y :$ **int** ;
$x, y \ := \ a, b$
{ Inv: P }
; **do** $x + 1 \neq y \quad \rightarrow$
{ $P \quad \wedge \quad x + 1 \neq y$ }
Modify x *and* y
{ P }
od
{ $P \quad \wedge \quad x + 1 = y$, hence R }
$]|$

Refining the loop body

In view of the initial values of x and y , and the conjunct $P0$ of the invariant, it is reasonable to investigate increments to x and decrements to y as possible refinements for our loop body. We choose to do so here.

Assuming m to be a value such that $x < m$, the statement $x := m$ properly increases x . Thus we are led to investigate the conditions under which this statement maintains the invariant. Noting that $P \ \wedge \ x < m$ is a part of the precondition to $x := m$, we calculate

$[\![$ Context: $P \ \wedge \ x < m$
$(x := m).P0$
$\equiv$ { Substitution }
$a \leq m \quad \wedge \quad m < y \quad \wedge \quad y \leq b$
$\equiv$ { $a \leq x \ \wedge \ x < m$ from the Context, hence $a \leq m$; $y \leq b$ from the Context }
$m < y$
$]\!]$

Thus we require m to satisfy $m < y$ as well. As for conjunct $P1$, we calculate again :

$(x := m).P1$

$\equiv$ { Substitution }

$m \mathcal{Z}\ y$

And so we choose to guard $x := m$ with $m \mathcal{Z}\ y$ as there seems to be no other readily apparent way to establish $m \mathcal{Z}\ y$ as a precondition of $x := m$.

As for a decrement to y, we note that m must satisfy $m < y$ as well, so $y := m$ properly decreases y . By symmetry, we are thus lead to the following refinement of our loop body

$|[$

"Compute m "

$\{\ x < m < y\ \}$

if $m \mathcal{Z}\ y \quad \rightarrow \quad x := m$

$[]\ \ x \mathcal{Z}\ m \quad \rightarrow \quad y := m$

fi

$]|$

Refining "Compute m"

Thanks to the conjunct $x < y$ of our invariant, and the guard $x + 1 \neq y$, we have $2 \leq y - x$ as a precondition to "Compute m ". Thus it is always possible to compute an m satisfying $x < m < y$. A popular choice for m which leads to very fast termination is $m := (x + y)$ **div** 2 . We choose this refinement for our program.

Termination

Thus we have derived a program which is partially correct. We now have to show that this program terminates. This involves two proof obligations, viz. to show that the selection statement does not abort, and to show that the loop terminates. We deal with these obligations in turn.

In general, to prove that a selection statement does not abort, we have to prove that the disjunction of its guards follows from its precondition. Thus in our case we have to prove

$$(x \mathcal{Z} m \quad \vee \quad m \mathcal{Z} y) \qquad \Leftarrow \qquad (P \quad \wedge \quad x+1 \neq y \quad \wedge \quad x < m < y) \qquad .$$

Thanks to the conjunct $P1$ of P, this obligation may be met by assuming the following property of $\mathcal{Z}$ (for all x, y, and m of appropriate type) :

(0) $$(x \mathcal{Z} m \quad \vee \quad m \mathcal{Z} y) \qquad \Leftarrow \qquad x \mathcal{Z} y \qquad .$$

(An Aside) This property is not as remarkable as it may first appear. It is equivalent to assuming that the complement of $\mathcal{Z}$ — denoted by $\neg \mathcal{Z}$ — is transitive. ie.

$$x \ (\neg \mathcal{Z}) \ m \quad \wedge \quad m \ (\neg \mathcal{Z}) \ y \qquad \Rightarrow \qquad x \ (\neg \mathcal{Z}) \ y \qquad .$$

(End of An Aside.)

Our last obligation is to show that the loop terminates. To meet this obligation, it suffices to show that the loop variables are bounded and at least one of them approaches its bound at each each step of the repetition.

From the invariant we have $a \leq x < y \leq b$, and so x and y are both bounded. The assignment $x := m$ properly increases x toward its upper bound b thanks to its precondition $x < m$. The assignment $y := m$ properly decreases y toward its lower bound a thanks to its precondition $m < y$. Since one of these assignments is executed at each step of the repetition, the loop is guaranteed to terminate.

The Program

Thus we have derived the following program for the Binary Search:

```
|[ var x, y, m : int ;
   x, y := a, b
 ; do x + 1 ≠ y  →
        m := (x + y) div 2
    ; if m Z y  →  x := m
      [] x Z m  →  y := m
      fi
   od
]|
```

One popular instantiation for $\mathcal{Z}$

Our program above works for any relation $\mathcal{Z}$ satisfying (0) . The most popular instantiation for $x \mathcal{Z} y$ is $f.x \leq C \ \wedge \ C < f.y$, for some constant C . This relation indeed satisfies (0) , as the reader may verify. With this instantiation of $\mathcal{Z}$, we obtain the following program:

$$
\begin{array}{l}
|[\ \textbf{var}\ x, y, m : \ \textbf{int}\ ; \\
\quad \{\ Q : \quad a < b \quad \wedge \quad f.a \leq C \quad \wedge \quad C < f.b\ \} \\
\quad x, y \ := \ a, b \\
\quad \{\ \text{Inv:} \quad P0 \quad \wedge \quad P1\ \} \\
\quad \{\ P0 : \quad a \leq x \quad \wedge \quad x < y \quad \wedge \quad y \leq b\ \} \\
\quad \{\ P1 : \quad f.x \leq C \quad \wedge \quad C < f.y\ \} \\
\quad ;\ \textbf{do} \quad x + 1 \neq y \quad \rightarrow \\
\qquad m := (x + y)\ \textbf{div}\ 2 \\
\qquad ;\ \textbf{if} \quad f.m \leq C \quad \rightarrow \quad x := m \\
\qquad\ \ [] \quad C < f.m \quad \rightarrow \quad y := m \\
\qquad\ \ \textbf{fi} \\
\quad \textbf{od} \\
\quad \{\ R : \quad a \leq x \quad \wedge \quad x < b \quad \wedge \quad f.x \leq C \quad \wedge \quad C < f.(x + 1)\ \} \\
]|
\end{array}
$$

Furthermore, if we are given that the function f is ascending, then we may record the presence of the value C in $f[a..b]$ by appending the following statement to the program above :

$$present \ := \ f.x = C \qquad .$$

If $f.x = C$, then $present$ will be assigned the value **true** . If $f.x \neq C$, then we have $f.x < C \ \wedge \ C < f.(x + 1)$ thanks to the postcondition of the repetitive construct. And since f is ascending, we may thus conclude that C will not occur anywhere else in f . Hence $present$ is correctly assigned the value **false** .

And this indeed is the Binary Search on ascending functions which we are all so familiar with. Since the ascendingness of the function comes into play only at the last possible

moment, we believe that the Binary Search is eminently applicable outside the realm of ascending functions, with which it has conventionally been so closely associated.

A note on applying the Binary Search

There are two important things to keep in mind when applying the Binary Search .

First, we noted that (0) is a sufficient condition to ensure that the Binary Search terminates. However, in applications of the Binary Search , we often calculate directly with the disjunction of the guards rather than trying to prove (0) . This is because a more specific problem often provides more properties with which to manipulate, and sometimes it is not possible to prove termination without appealing to these properties.

Second, it is often possible to use the Binary Search even when we have a weaker precondition than $a < b$. For instance, if we have as precondition $a - 1 < b + 1$, we may instantiate the Binary Search with $a, b := a - 1, b + 1$ by defining thought values such that we have $(a - 1)\ \mathcal{Z}\ (b + 1)$. However, in these cases, we have the additional obligation to ensure that f values outside the range $[a..b]$ are not evaluated.

We shall demonstrate how these concerns come into play in the next chapter.

Acknowledgment

This chapter is modeled on Wim Feijen's exposition of the Binary Search as presented in [FG96] .

7. The Top of the Hill

Introduction

In this chapter, we demonstrate an application of the Binary Search algorithm, as well as an application of the design technique known as stepwise refinement.

The problem we are about to tackle may be informally specified as follows: Given that array $f[a..b]$, $a \leq b$, is the concatenation of an increasing and a decreasing sequence, design a program to compute the index of the maximal element of f .

As always, we begin by getting a better grip on the problem through a formal specification.

A formal specification

We specify our program as follows :

|[**con** a, b, N : **int** ; **con** f : **array** $[a..b]$ of **int** ;
var x : **int** ;
{ Q }
Top of the hill
{ R }
]| .

Our postcondition is:

R : $a \leq x \leq b \quad \wedge \quad C.x$.

Our precondition is:

Q : $Q0 \quad \wedge \quad Q1$

$Q0$: $a \leq b$

$Q1$: $a \leq N \leq b \quad \wedge \quad C.N$.

The predicate C used above is specified as follows:

$C.x$: $I.a.x \quad \wedge \quad D.x.b$,

for x an integer. The expression $I.a.b$ denotes "$f[a..b]$ is increasing" , and $D.a.b$ denotes "$f[a..b]$ is decreasing" .

Applying the Binary Search

Since the problem involves searching for an element in an array, we may employ one of our searching algorithms to solve it. In this case, we choose to use the Binary Search . We recall that the specification of the Binary Search is :

$|[$ **con** $a, b :$ **int** ; **con** $f :$ **array** $[a..b]$ of **int** ;

var $x :$ **int** ;

$\{\ a < b \quad \wedge \quad a\ \mathcal{Z}\ b\ \}$

Binary Search

$\{\ a \leq x < b \quad \wedge \quad x\ \mathcal{Z}\ (x+1)\ \}$

$]|$.

The first step toward applying the Binary Search is to design a suitable relation $\mathcal{Z}$ for the problem at hand. What properties will this relation need to satisfy? If we rewrite the sub-expression '$\leq b$' in our specification as —the equivalent— '$< b+1$' , and compare the specification of the Binary Search with that of our program, we see that our $\mathcal{Z}$ needs to satisfy:

- $a\ \mathcal{Z}\ (b+1)$.
- $x\ \mathcal{Z}\ (x+1) \quad \equiv \quad I.a.x \ \wedge\ D.x.b \quad (\equiv \quad C.x)$.

In order to satisfy the second requirement, we shall define $\mathcal{Z}$ as:

(0) $\qquad x\ \mathcal{Z}\ y \qquad \equiv \qquad I.a.x \quad \wedge \quad D.(y-1).b \qquad .$

This specification of $\mathcal{Z}$ satisfies the first requirement provided we have

$$I.a.a \quad \wedge \quad D.b.b$$

as properties of I and D . We shall assume these properties.

Thus our precondition becomes:

$Q :$ $\quad Q0 \quad \wedge \quad Q1$

$Q0 :$ $\quad a < b+1 \quad \wedge \quad a\ \mathcal{Z}\ (b+1)$

$Q1 :$ $\quad a \leq N \leq b \quad \wedge \quad C.N \qquad .$

Our postcondition becomes:

$R:\quad a \le x < b+1 \quad \wedge \quad x \mathrel{\mathcal{Z}} (x+1) \quad .$

Thus we may instantiate the Binary Search with $a, b := a, b+1$. And so we arrive at the following program.

Program

$$
\begin{array}{l}
|[\ \textbf{var}\ \ x, y, m : \ \textbf{int} \\
\quad x, y \ := \ a, b+1 \\
\quad \{\ \text{Inv:}\ \ P \quad \equiv \quad P0 \ \wedge \ P1\ \} \\
\quad \{\ P0 : \ a \le x \ \wedge \ x < y \ \wedge \ y \le b+1\ \} \\
\quad \{\ P1 : \ x \mathrel{\mathcal{Z}} y\ \} \\
\quad ;\ \textbf{do}\ \ x+1 \neq y \quad \rightarrow \\
\qquad\quad m := (x+y)\ \textbf{div}\ 2 \\
\qquad\quad \{\ x < m < y\ \} \\
\qquad\quad ;\ \textbf{if}\ \ m \mathrel{\mathcal{Z}} y \quad \rightarrow \quad x := m \\
\qquad\quad\ \ []\ \ x \mathrel{\mathcal{Z}} m \quad \rightarrow \quad y := m \\
\qquad\quad\ \ \textbf{fi} \\
\qquad\quad \{\ P\ \} \\
\quad\ \ \textbf{od} \\
\quad \{\ P \ \wedge \ x+1 = y\ ,\ \text{hence}\ \ R\ \} \\
|]\qquad .
\end{array}
$$

By refining $m \mathrel{\mathcal{Z}} y$ and $x \mathrel{\mathcal{Z}} m$ using (0) , and assuming that $I.a.b$ and $D.a.b$ do not evaluate f values outside the range $[a..b]$, we may conclude from $P0$ and $x < m < y$ that this program does not inspect f values outside the range $[a..b]$. Thus our only remaining obligation is to prove the non-abortion of the selection statement.

Proving non-abortion

To prove non-abortion, we must prove the disjunction of the guards. Thus we calculate:

$m \mathcal{Z} y \quad \vee \quad x \mathcal{Z} m$

$\equiv \quad \{ \quad (0) \quad \}$

$(I.a.m \ \wedge \ D.(y-1).b) \quad \vee \quad (I.a.x \ \wedge \ D.(m-1).b)$

$\equiv \quad \{ \quad x \mathcal{Z} y \quad (\equiv \quad I.a.x \ \wedge \ D.(y-1).b) \quad \text{from the invariant} \ \}$

$I.a.m \quad \vee \quad D.(m-1).b$

$\equiv \quad \{ \quad (1) \ \text{and} \ (2) \ ; \ \ a < m < b \quad \}$

$m \leq N \quad \vee \quad N \leq (m-1)$

$\equiv \quad \{ \ \text{Arithmetic} \ \}$

$\mathbf{true} \qquad .$

In the above, we have used the following properties of predicates I and D. For $a \leq p < q \leq b$, we have :

(1) $\qquad I.p.q \quad \equiv \quad q \leq N$

(2) $\qquad D.p.q \quad \equiv \quad N \leq p \qquad .$

Properties (1) and (2) follow from $C.N$ in $Q1$, as well as properties of I and D, as the interested reader may verify.

Refining the guards

Finally, it would be nice to refine the guards into easily executable code. Toward this end we calculate with guard $m \mathcal{Z} y$:

$m \mathcal{Z} y$

$\equiv \quad \{ \quad (0) \quad \}$

$I.a.m \quad \wedge \quad D.(y-1).b$

$\equiv \quad \{ \quad x \mathcal{Z} y \quad (\equiv \quad I.a.x \ \wedge \ D.(y-1).b) \quad \}$

$I.a.m$

$\equiv \quad \{ \quad (1) \ ; \ \ a < m < b \quad \}$

$m \leq N$

$\equiv \quad \{ \quad (1) \ ; \ \ a \leq m-1 < m < b \quad \}$

$$
\begin{array}{ll}
& I.(m-1).m \\
\equiv & \quad \{ \text{ Property of } \ I \ \} \\
& f.(m-1) < f.m \qquad .
\end{array}
$$

We can calculate

$$x \ \mathcal{Z} \ m \quad \equiv \quad f.m < f.(m-1)$$

in a similar manner, as the interested reader may verify.

The final program

And so we arrive at our final program:

$$
\begin{array}{l}
|[\ \textbf{var} \ \ x, y, m : \ \textbf{int} \\
\quad x, y := a, b+1 \\
\quad ; \textbf{do} \ \ x+1 \neq y \quad \rightarrow \\
\qquad\quad m := (x+y) \ \textbf{div} \ 2 \\
\qquad\quad ; \textbf{if} \ \ f.(m-1) < f.m \quad \rightarrow \quad x := m \\
\qquad\quad \ [] \ \ f.m < f.(m-1) \quad \rightarrow \quad y := m \\
\qquad\quad \ \textbf{fi} \\
\quad \textbf{od} \\
\quad]|
\end{array}
$$

Postscript: An observation

Finally, in addition to (1) , we have used the following properties of I :

- $I.p.p$
- $I.(p-1).p \quad \equiv \quad f.(p-1) < f.p$

We wish to note that these properties are both satisfied by the conventional specification of an ascending sequence:

$$I.p.q \quad \equiv \quad \langle \forall i : \ p \leq i < q : \ f.i < f.(i+1) \ \rangle \qquad .$$

The same observation applies to predicate D .

Acknowledgment

I am indebted to the Eindhoven Tuesday Afternoon Sessions for the original version of this design. In addition, Tom Verhoeff and Jeremy Weissmann provided extensive comments on earlier versions of this document. Their feedback has been invaluable.

8. Concluding Remarks

On what we have achieved

We have presented the derivation of several algorithms in this project. In our presentations, we have articulated the choices faced, and the decisions made, at every turn. We have articulated the heuristics guiding our decisions wherever possible. All through this process, we have managed to maintain brevity: each exposition is on the average only six pages long, inclusive of program text, etc. .

Thus we feel that we have achieved what we set out to do, viz. to show how one may achieve precision, clarity, brevity, and discipline in the process of designing algorithms. In other words, we have achieved our objective of showing how one may be methodical in the algorithm design activity.

On what we have ignored

In this project, we have by-and-large ignored the task of designing the specifications for our algorithms. We have always begun our designs by analyzing a specification of the algorithm, which was presented in the terminology of our chosen conceptual interface. We shall now highlight the importance of the specification design process, so as to gain a better appreciation of the context in which we use our method.

The specification acts as an interface between the algorithm and the larger system. It specifies what both the algorithm, and the system, require to perform their function, as well as what they provide to each other.

From the point of view of the algorithm designer, the specification is the only information available from which to proceed with the design. Thus, if the specification is poorly designed, it is likely that the algorithm designer will find it difficult, if not impossible, to methodically design an algorithm meeting it.

From the point of view of the system designer, the algorithm specification acts like a contract which the algorithm must fulfill. If this contract incorrectly specifies what is required of the algorithm, the algorithm will behave incorrectly, resulting in an error in the system. Thus for our algorithms to be useful to the larger system, the specification must correctly express what is required of it.

Hence, if we wish to use our method to efficiently design algorithms which are a part of a larger system, we must pay attention to the process of designing good specifications.

However, we feel that the design of specifications is a separate activity, and hence is better dealt with separately. This is why we chose to ignore the specification process in this project, even though we consider it of great importance.

The journey ahead

What can we do to improve our method for designing algorithms?

Firstly, we could try to solve more programming problems using this method. These solutions may reveal oversights, suggest new heuristics, inspire the formulation of new program schemes, etc. . All of this would make the methodology more robust and useful.

Secondly, as we mentioned in the previous section, the ability to design good specifications is crucial to our ability to efficiently use our method to design algorithms which are part of a larger system. However, we have not yet been able to develop a methodical approach to specification design. Bringing method to the specification process would greatly enhance the utility of our methods for designing algorithms. Thus we feel that investigating the specification process is an important part of the journey ahead of us.

* *
*

And so we have reached the end of this little monograph. I hope that the reader has enjoyed reading it as much as I have enjoyed writing it. I would consider the whole enterprise successful if the reader, after carefully studying the material, gains an appreciation of the subtlety of the algorithm design process, resulting in his taking greater care while designing his own algorithms.

9. References

[DS90] Dijkstra, E.W. and Scholten, C.S. , *Predicate Calculus and Program Semantics* , Springer-Verlag New York , 1990 .

[Kal90] Kaldewaij, A. , *Programming: The Derivation of Algorithms* , Prentice Hall , 1990 .

[WF116] Feijen, W.H.J. , *WF116: The Linear Search, and some more* , 1990 .

[WF118] Feijen, W.H.J. , *WF118: The Bounded Linear Search* , 1990 .

[FG96] Feijen, W.H.J. and van Gasteren, A.J.M. , *The Binary Search revisited* , Educational Issues of Formal Methods , Academic Press , 1996 .