Programming, Proving, and Calculation

W.H.J. Feijen and A.J.M. van Gasteren

April 12, 1999

Preface

It has been recognized for a long time now, that formalisms are indispensable for the design of reliable, i.e. correct, programs. It has been recognized also, however, that formalization in itself is not enough: we must see to it that both the formalisms and the way in which they are used are simple enough and clear enough to be useful and teachable. This, in a nutshell, has been the primary aim of most of our work over the years, and it constitutes the main theme of the present chapter.

Such use of formalism is, for instance, illustrated by the calculational style in which the predicate calculus is used throughout the chapter. After having used the calculational format for many years, both in teaching and otherwise, we are convinced that it is a primary means for combining clarity of exposition with economy of expression.

There are three sections to this chapter. The sections on sequential (imperative) programming and multiprogramming each contain a small collection of specimens of how we teach programming to our students to date. They exemplify how sequential and parallel programs can be derived from their specifications hand in hand with their proofs of correctness. Here, the austerity and simplicity of the formalisms used, viz. predicate calculus, Hoare-triple semantics and the theory of Owicki and Gries, are of decisive importance. The section on calculational mathematics is an illustration of how we think that students can be made familiar with the calculational style.

As a prerequisite we assume the reader to be more or less familiar with the predicate calculus and with Hoare triples and their use.

1

A little on calculational mathematics

What associativity is about

We all know what it means for a binary operator Δ to be associative:

$$(a \,\Delta\, b) \,\Delta\, c \;=\; a \,\Delta\, (b \,\Delta\, c) \quad .$$

Now, if you interview people, asking them what associativity is really about, one of the first answers you get is that it is allowed to omit the parentheses and write

$$a \,\Delta\, b \,\Delta\, c$$

without introducing ambiguity. And this is right!
Next you may get the answer that thanks to the associativity it does not matter whether the value of $a \,\Delta\, b \,\Delta\, c$ is computed from left to right or from right to left. And you may get several other answers, but hardly ever the one to be explained next.

* *
*

For people that calculate, the property of associativity may offer very strong heuristic guidance in proof construction : if, in a calculation, the expression

$$a \,\Delta\, b \,\Delta\, c$$

enters the picture as

$$(a \,\Delta\, b) \,\Delta\, c \quad ,$$

then the advice for the next step(s) is to focus on the subexpression $b \,\Delta\, c$ in

$$a \,\Delta\, (b \,\Delta\, c) \quad .$$

In our experience, this heuristic rule has worked well in a tremendous number of cases. And if you come to think of it: of course, what else could associativity be about?

* * *

The proof of the pudding is in the eating, and we shall now present the proof of a FANTASTIC theorem that our colleague Paul F. Hoogendijk challenged us to prove one day. The theorem is FANTASTIC to the extent that we have absolutely no idea what the mathematics involved "means".

In what follows, constant Γ and dummies x and y are of type Thing. Further, Λ and E map Things to Things, and, last but not least, binary operator $\circ$, which maps two Things on a Thing, is associative. In short, in the ensuing expressions there is no type conflict.

Now, given (0) and (1)

(0) $\quad x = \Lambda.y \quad \equiv \quad \Gamma \circ x = y \qquad (\forall x, y)$

(1) $\quad \mathrm{E}.x \;=\; \Lambda.(x \circ \Gamma) \qquad (\forall x)$

we have to prove

(2) $\quad \mathrm{E}.x \circ \Lambda.y \;=\; \Lambda.(x \circ y) \qquad (\forall x, y)$

Proof Property (0) is a so-called *Galois Connection* and the advice is that we then always write down the corresponding *cancellation rules* as well [**?**]. Here they are (instantiate (0) with $x := \Lambda.y$ and $y := \Gamma \circ x$, respectively, so as to make one side vacuously true) :

(3) $\quad \Gamma \circ \Lambda.y = y \qquad (\forall y)$

(4) $\quad x = \Lambda.(\Gamma \circ x) \qquad (\forall x)$

Now we tackle (2) and, for this occasion, we shall make the steps that appeal to the associativity of $\circ$ explicit.

$$
\begin{array}{ll}
 & \mathrm{E}.x \circ \Lambda.y \;=\; \Lambda.(x \circ y) \\
\equiv & \qquad \{ \text{ (0) from left to right } \} \\
 & \Gamma \circ (\mathrm{E}.x \circ \Lambda.y) \;=\; x \circ y \\
\equiv & \qquad \{ \;\circ\; \text{is associative } \} \\
 & (\Gamma \circ \mathrm{E}.x) \circ \Lambda.y \;=\; x \circ y \\
\equiv & \qquad \{ \text{ (1) to eliminate } \mathrm{E} \; \} \\
 & (\Gamma \circ \Lambda.(x \circ \Gamma)) \circ \Lambda.y \;=\; x \circ y \\
\equiv & \qquad \{ \text{ (3) with } y := x \circ \Gamma \; \} \\
 & (x \circ \Gamma) \circ \Lambda.y \;=\; x \circ y \\
\equiv & \qquad \{ \;\circ\; \text{is associative } \} \\
 & x \circ (\Gamma \circ \Lambda.y) \;=\; x \circ y \\
\equiv & \qquad \{ \text{ (3) } \} \\
 & x \circ y = x \circ y \\
\equiv & \qquad \{ \quad \} \\
 & \mathit{true} \quad .
\end{array}
$$

End Proof .

A very beginning of lattice theory

Let's start at the very beginning.
A very good place to start.

Julie Andrews in *The Sound of Music*

For a large part, mathematics consists of exploring concepts and of investigating and proving their properties. The art of proving plays a major rôle in this game. Since the advent of modern computing science, it has become clear that in many branches of elementary mathematics, proofs can be beneficially rendered in a calculational format. The benefits comprise greater precision and lucidity —without loss of concision—, an enhanced view on how to separate one's concerns, and hence an improved economy of thought. Unfortunately, most textbooks on elementary mathematical issues have not (yet) adopted such a calculational style, so that yet another generation of young people will receive a mathematical education without having experienced the joy and usefulness of calculating. And this is a pity.

The purpose of this note is to transmit some of the flavour of calculation. We have selected a topic from the very beginning of lattice theory and we intend to present a treatment that can be read, understood, and hopefully enjoyed by a reasonable university freshman.

*　　　*
*

Our universe of discourse will be some fixed, anonymous set of things on which a binary relation $\leq$ (" at most") is defined. This relation we postulate to be

- reflexive, i.e. $x \leq x$ $(\forall x)$
- antisymmetric, i.e. $x \leq y \wedge y \leq x \Rightarrow x = y$ $(\forall x, y)$

Remark In the standard literature we usually find the additional postulate that $\leq$ is

transitive, i.e. $x \leq y \wedge y \leq z \Rightarrow x \leq z$ $(\forall x, y, z)$

For the time being though, we do not need the transitivity of $\leq$. Therefore, we do not introduce it now. And apart from that, it will — as we shall see — enter the picture in a totally different way.
End Remark .

Equality of things is a very important concept to have. It is as important as the notion of a function. Equality and functions are at the heart of mathematics, and they are beautifully related by the

Rule of Leibniz

For any function f[1], $x = y \;\Rightarrow\; f.x = f.y$.

End

The two postulates that we have of $\leq$ do not reveal very much about equality; only the antisymmetry mentions it. Therefore, the first thing to do is to collect some more facts concerning equality. The most common one is the

Rule of Mutual Inequality

$$x = y \;\equiv\; x \leq y \wedge y \leq x \quad .$$

End

It is an immediate restatement of $\leq$'s reflexivity and antisymmetry.

A very useful but less common statement about equality is the so-called

Rule of Indirect Equality

$$x = y \;\equiv\; \langle \forall z :: z \leq x \equiv z \leq y \rangle \quad .$$

End

Let us prove it. We prove it by mutual implication. (In our jargon we refer to the implication $LHS \Rightarrow RHS$ from left to right by “ping” and to $RHS \Rightarrow LHS$ by “pong”.)

Proof of ping

$$x = y \;\Rightarrow\; \langle \forall z :: \; z \leq x \equiv z \leq y \rangle$$

[1] We denote function application by an infix dot.

$$
\begin{array}{ll}
\equiv & \quad \{ \ (P\Rightarrow) \text{ distributes over } \forall \ \} \\
 & \langle\, \forall z :: \ x = y \ \Rightarrow \ (z \le x \equiv z \le y) \,\rangle \\
\equiv & \quad \{ \ \text{Rule of Leibniz, see below} \ \} \\
 & \mathit{true}
\end{array}
$$

The function f involved in this application is the boolean function given by $f.a \equiv z \le a$.
End

Proof of pong We have to prove

$$\langle\, \forall z :: z \le x \equiv z \le y \,\rangle \Rightarrow x = y \quad ,$$

and we do so by setting up a weakening chain of predicates that begins with the antecedent and ends with the consequent. Notice that in this chain we will quite likely have to refer to the antisymmetry of $\le$ because this is the only property of $\le$ that mentions the =-symbol; and we have not used it in the ping-part yet. (The latter remark is a very simple example of the kind of bookkeeping that has proven to be very useful in proof design.) Here is the chain

$$
\begin{array}{ll}
 & \langle\, \forall z :: \ z \le x \ \equiv \ z \le y \,\rangle \\
\Rightarrow & \quad \{ \ \text{instantiate with } z := x \text{ and with } z := y \ \} \\
 & (x \le x \ \equiv \ x \le y) \ \wedge \ (y \le x \ \equiv \ y \le y) \\
\equiv & \quad \{ \ \text{reflexivity of } \le \ \} \\
 & x \le y \ \wedge \ y \le x \\
\Rightarrow & \quad \{ \ \text{antisymmetry of } \le \ \} \\
 & x = y
\end{array}
$$

Notice that the first step —the instantiation— is not brilliant at all: the first line contains symbol ∀ and the target line does not, so that somewhere along the way we must eliminate ∀. In fact, about the only rule from the predicate calculus with which one can eliminate the universal quantifier ∀, is the rule of instantiation. Once we are aware of this, the step is no longer a surprise. Furthermore, there is not much

we can instantiate z with, viz. just x and y ; and we did both in order to make the next line as strong as possible, which is beneficial if one has to construct a weakening chain.
End Proof of pong .

The rule of Indirect Equality has a companion, also called the Rule of Indirect Equality. It reads

$$x=y \;\equiv\; \langle \forall z :: \; x\leq z \;\equiv\; y\leq z \rangle \;\; .$$

The difference is the side of the $\leq$ -symbol at which x and y reside. Which of the two is to be used depends on the particular application.

* *
*

So much for $\leq$ and for $=$ in our universe. We now enter lattice theory by postulating that in our universe the equation in p

$$p: \;\; \langle \forall z :: \; p\leq z \;\equiv\; x\leq z \wedge y\leq z \rangle$$

has, for each x and y , at least one solution. (The inexperienced reader should not feel daunted here: in case our universe is just the universe of real numbers with the usual $\leq$ -relation, the maximum of x and y may be recognized as a good candidate for p .)

The first thing we do is to show that the equation has at most one solution. This is done by showing $p=q$, for p and q solutions of the equation. Here, one of the rules of Indirect Equality comes in handy: for any z , we have

$$\begin{array}{ll} & p\leq z \\ \equiv & \quad \{ \; p \text{ is a solution } \} \\ & x\leq z \wedge y\leq z \\ \equiv & \quad \{ \; q \text{ is a solution } \} \\ & q\leq z, \end{array}$$

and, hence, $p=q$. So our equation has exactly one solution for each x and y. Therefore, that solution is a function of x and y , which we propose to denote by $x\uparrow y$ (x “up”y). In summary, we have the beautiful

(0) $$x\uparrow y\leq z \;\equiv\; x\leq z \,\wedge\, y\leq z \qquad (\forall x,y,z)$$

(In the standard literature we find $\uparrow$ under entries like "sup" or "join" or "lub".)

Examples

- A well-known instance of (0) can be found in set theory. If we take set inclusion $\subseteq$ as an instance of $\leq$ —it is reflexive and antisymmetric!— , set union $\cup$ is the corresponding $\uparrow$. Indeed, we have for all sets x , y , and z ,

  $$x\cup y \;\subseteq\; z \;\equiv\; x\subseteq z \,\wedge\, y\subseteq z \quad .$$

- Also, if we take set containment $\supseteq$ for $\leq$, set intersection is the corresponding $\uparrow$. Indeed,

  $$x\cap y \supseteq z \;\equiv\; x\supseteq z \,\wedge\, y\supseteq z \quad .$$

- Another well-known instance is in the predicate calculus where we have

  $$[\; x\vee y \Rightarrow z \;] \;\equiv\; [\; x\Rightarrow z \;] \wedge [\; y\Rightarrow z \;] \quad , \qquad \text{and}$$
  $$[\; x\wedge y \Leftarrow z \;] \;\equiv\; [\; x\Leftarrow z \;] \wedge [\; y\Leftarrow z \;] \;.$$

- From number theory we know the reflexive, antisymmetric relation denoted $|$ ("divides"). Now, the least common multiple of x and y can see the light via

  $$(x \;\mathsf{lcm}\; y) \mid z \;\equiv\; x\mid z \,\wedge\, y\mid z \quad ,$$

  and the greatest common divisor of x and y by

  $$z\mid (x \;\mathsf{gcd}\; y) \;\equiv\; z\mid x \wedge z\mid y \quad .$$

  Both are instances of (0). (How?)

- But probably the best-known instance of (0) is when we take for $\leq$ the usual order between numbers. Then $\uparrow$ is the familiar maximum operator. We will return to this later.

End

Now let us investigate (0). We can rather straightforwardly deduce from it that

- $\uparrow$ is idempotent, i.e. $x \uparrow x \;=\; x$
- $\uparrow$ is symmetric, i.e. $x \uparrow y \;=\; y \uparrow x$
- $\uparrow$ is associative, i.e. $x \uparrow (y \uparrow z) \;=\; (x \uparrow y) \uparrow z$.

Let us prove the symmetry. We appeal to Indirect Equality :

$$\begin{array}{ll} & x \uparrow y \leq z \\ \equiv & \quad \{ \ (0) \ \} \\ & x \leq z \ \wedge \ y \leq z \\ \equiv & \quad \{ \ \wedge \text{ is symmetric } \} \\ & y \leq z \ \wedge \ x \leq z \\ \equiv & \quad \{ \ (0) \text{ with } x , y \ := \ y , x \ \} \\ & y \uparrow x \ \leq \ z, \end{array}$$

and the conclusion follows. From this proof we see that $\uparrow$ inherits its symmetry from $\wedge$. The same holds for $\uparrow$'s idempotence and $\uparrow$'s associativity, as the reader may verify.

The next thing we do with (0) is to study it for some simple instantiations. For instantiation $z := y$ we find

$$\begin{array}{ll} & x \uparrow y \ \leq \ y \\ \equiv & \quad \{ \ (0) \ \} \\ & x \leq y \ \ \wedge \ \ y \leq y \\ \equiv & \quad \{ \ \leq \text{ is reflexive } \} \\ & x \leq y \ \ . \end{array}$$

Thus we have derived the

Rule of Absorption

$$x \uparrow y \ \leq \ y \ \ \equiv \ \ x \leq y$$

End

Next, from (0) with $z := x\uparrow y$, we find the

Rule of Expansion

$$y \leq x\uparrow y$$

End

Using Mutual Inequality, we can combine the rules of Absorption and Expansion into

$$(1) \qquad x\uparrow y = y \;\equiv\; x\leq y \qquad (\forall x,y)$$

Remark Almost every established treatment of lattice theory starts from (1), but that is not nearly as nice as the treatment given here, because the pleasing symmetry exhibited by (0) is completely hidden.
End

So much for some simple instantiations of (0).

* *
*

Now the time has come to prove the beautiful[2]

Theorem For reflexive and antisymmetric $\leq$, and for $\uparrow$ as defined by (0), we have that

$$\leq \text{ is transitive } .$$

Proof We have to prove that for all x , y , and z

$$x\leq y \;\wedge\; y\leq z \;\;\Rightarrow\;\; x\leq z \;\; .$$

Using (1), this we can rewrite as

$$x\uparrow y = y \;\wedge\; y\uparrow z = z \;\;\Rightarrow\;\; x\uparrow z = z \;\; ,$$

and we shall prove this latter by showing the consequent — $x\uparrow z = z$ — thereby using the antecedent — $x\uparrow y = x \;\wedge\; y\uparrow z = z$ — :

$$x\uparrow z$$

[2] We owe this theorem to Edsger W. Dijkstra. It seems to be not generally known to lattice theorists.

$=$ { since $y\uparrow z=z$, from the antecedent }

$x\uparrow(y\uparrow z)$

$=$ { $\uparrow$ is associative }

$(x\uparrow y)\uparrow z$

$=$ { since $x\uparrow y=y$, from the antecedent }

$y\uparrow z$

$=$ { since $y\uparrow z=z$, from the antecedent }

z .

And we are done. (We ask the reader to notice that each individual step in the above calculation is almost forced upon us. This is a very typical characteristic of many a calculation.)
End

Now that we have obtained the transitivity of $\leq$, we shall feel free to use it. For the sake of completeness we mention that a reflexive, antisymmetric, and transitive relation is commonly called a *partial order*, and that a universe equipped with a partial order is called a *partially ordered set* —a "poset" for short—.

* *
*

Definition (0) of $\uparrow$ tells us when $x\uparrow y\leq z$. We now may ask when $z\leq x\uparrow y$. We leave to the reader to verify that

(2) $z \leq x\uparrow y \;\Leftarrow\; z\leq x \;\vee\; z\leq y$,

and we investigate the converse :

$z \leq x\uparrow y \;\Rightarrow\; z\leq x \;\vee\; z\leq y$

$\equiv$ { predicate calculus }

$(z \leq x\uparrow y \;\Rightarrow\; z\leq x) \;\vee\; (z \leq x\uparrow y \;\Rightarrow\; z\leq y)$

$\Leftarrow$ { $\leq$ is transitive }

$x\uparrow y \leq x \;\vee\; x\uparrow y\leq y$

$\equiv$ { Rule of Absorption, twice }

$y \leq x \ \vee\ x \leq y$.

For this last line to be valid for any x and y , we require that $\leq$ be a so-called linear or total order: by definition a total order is a partial order with the additional property that, for all x and y , $x \leq y \ \vee\ y \leq x$.
So, in combination with (2) we find

(3) for $\leq$ a total order,
$z \leq x \uparrow y \ \equiv\ z \leq x \ \vee\ z \leq y$.

Furthermore, we deduce from (1) that

(4) for $\leq$ a total order, operator $\uparrow$ as defined by (0) satisfies
$x \uparrow y = x \ \vee\ x \uparrow y = y$.
(In words : $\uparrow$ is a selector.)

* *
*

From here, we can proceed in many different directions. After all, lattice theory is a huge mathematical terrain, with many ins and outs. We conclude this introduction by confronting $\uparrow$ with other functions.

We consider functions from and to our anonymous universe. For f such a function we have, by definition,

- f is monotonic $\equiv$ $\langle \forall x,y ::\ x \leq y \Rightarrow f.x \leq f.y \rangle$.
- f distributes over $\uparrow$ $\equiv$ $\langle \forall x,y ::\ f.(x \uparrow y) = f.x \uparrow f.y \rangle$.

We can now formulate the well-known, yet beautiful, theorem

(5) f distributes over $\uparrow$ $\Rightarrow$ f is monotonic .

Proof For any x and y , we observe

$f.x \leq f.y$

$\equiv$ { (1) }

$f.x \uparrow f.y = f.y$

$$
\begin{array}{ll}
\equiv & \quad \{\ f \text{ distributes over } \uparrow\ \} \\
& f.(x\uparrow y) = f.y \\
\Leftarrow & \quad \{\ \text{Leibniz's Rule}\ \} \\
& x\uparrow y = y \\
\equiv & \quad \{\ (1)\ \} \\
& x \leq y \quad ,
\end{array}
$$

and the result follows from the outer two lines.
End

Small Intermezzo (on proof design)

We would like to draw the reader's attention to the fact that the above proof —no matter how simple it is— displays a great economy of thought. Let us analyze it in some detail. Given that f distributes over $\uparrow$, we have to construct a calculation of the form

$$f.x \leq f.y \ \ldots \Leftarrow \ldots\ x \leq y \quad .$$

Right at the outset we can argue that such a calculation will require at least four steps, viz.

- a step to introduce symbol $\uparrow$, in order to be able to exploit the given about f
- a step in which the given about f is actually used
- a step to eliminate symbol $\uparrow$ again, because the target line $x \leq y$ does not mention it
- a step to eliminate symbol f , for which Leibniz's Rule is our only means so far.

Our proof contains precisely (these) four steps, so it cannot be shortened. In fact, it was designed with these four considerations in mind. When we wrote above "no matter how simple it is", this may have sounded paradoxical, but it isn't. On the contrary, the proof derives its

simplicity from the consciously considered shapes of the formulae and from the manipulative possibilities available. Nowadays, many more proofs can be and are being designed following such a procedure.
End Small Intermezzo .

A direct consequence of (5) concerns monotonicity properties of $\uparrow$. Because function f defined by $f.x = c\uparrow x$, for whatever c , distributes over $\uparrow$ —as the reader may verify—, theorem (5) tells us that $\uparrow$ is monotonic in its second argument. Since $\uparrow$ is symmetric, we therefore have

(6) $\uparrow$ is monotonic in both arguments.

What about the converse of (5)? Does it hold as well? In order to find out, we try to prove

$$f.(x\uparrow y) \;=\; f.x\uparrow f.y$$

on the assumption that f is monotonic. We do this by Mutual Inequality:

$$\begin{array}{ll}
 & f.x\uparrow f.y \;\le\; f.(x\uparrow y) \\
\equiv & \qquad \{ \text{ definition of } \uparrow \text{, see (0) } \} \\
 & f.x \le f.(x\uparrow y) \quad\wedge\quad f.y \le f.(x\uparrow y) \\
\Leftarrow & \qquad \{ \text{ monotonicity of } f \text{, twice } \} \\
 & x \le x\uparrow y \quad\wedge\quad y \le x\uparrow y \\
\equiv & \qquad \{ \text{ Rule of Expansion, twice } \} \\
 & true \quad ,
\end{array}$$

$$\begin{array}{ll}
 & f.(x\uparrow y) \;\le\; f.x\uparrow f.y \\
\Leftarrow & \qquad \{ \text{ (2) } \} \\
 & f.(x\uparrow y) \le f.x \quad\vee\quad f.(x\uparrow y) \le f.y \\
\Leftarrow & \qquad \{ \text{ monotonicity of } f \text{, twice } \} \\
 & x\uparrow y \le x \quad\vee\quad x\uparrow y \le y \\
\equiv & \qquad \{ \text{ Rule of Absorption, twice } \}
\end{array}$$

$$y \leq x \vee x \leq y \quad ,$$

and the validity of this last line requires $\leq$ to be total. Thus, we have derived, in combination with (5),

(7) for $\leq$ a total order,
f is monotonic $\equiv$ f distributes over $\uparrow$.

* *
*

In lattice theory, one always introduces a companion to $\uparrow$; it is $\downarrow$ ("down"). (In the standard literature we find $\downarrow$ under entries like "inf", or "meet", or "glb".) It sees the light via

(8) $$z \leq x \downarrow y \;\equiv\; z \leq x \wedge z \leq y \qquad (\forall x, y, z),$$

i.e. in a way that is very similar to (0). It has very similar —dual— properties to $\uparrow$. In fact, it has the same properties if we simply flip $\leq$ into $\geq$, and $\uparrow$ into $\downarrow$: just compare (0) and (8). With this symbol dynamics in mind the companion properties for $\downarrow$ come for free. We mention

- $\downarrow$ is idempotent, symmetric, and associative
- $y \leq x \downarrow y \;\equiv\; y \leq x$ Absorption
- $x \downarrow y \leq y$ Contraction (= the dual of Expansion)
- $x \downarrow y = y \;\equiv\; y \leq x$
- $x \downarrow y \leq z \;\Leftarrow\; x \leq z \vee y \leq x$
- $\downarrow$ is monotonic in both arguments
- etcetera .

Of course, we can now also investigate formulae containing both $\uparrow$ and $\downarrow$. We mention

$$x \downarrow (x \uparrow y) = x \quad , \quad x \uparrow (x \downarrow y) = x \quad , \quad \text{and}$$

$$x{\downarrow}y=x \quad\equiv\quad x{\uparrow}y=y \quad .$$

The proofs are left as exercises. We will not continue these investigations now.

In case we take for $\le$ the usual order between real numbers, $\downarrow$ is the familiar minimum operator.

*　　　*
*

Let us, to conclude this story, consider the real numbers with the usual order $\le$. This is a total order. The foregoing little theory now grants us quite a number of useful arithmetical results.

- In order to find out which part of the $(x\,,y)$-plane satisfies $x{\uparrow}y \le x{+}y$, we simply calculate :

  $$\begin{array}{ll} & x{\uparrow}y \;\le\; x{+}y \\ \equiv & \qquad \{\ \text{definition of } f \quad \} \\ & x \le x{+}y \quad\wedge\quad y \le x{+}y \\ \equiv & \qquad \{\ \text{arithmetic}\ \} \\ & 0 \le y \quad\wedge\quad 0 \le x \quad . \end{array}$$

  So the answer is: the first quadrant.
  (Ask one of your colleagues or students to solve this little problem, and observe how he does it. This could be a very instructive experiment.)

- Since function f , defined by $f.x=c{+}x$, is monotonic, we infer from (7):

  addition distributes over the maximum.

- Likewise, multiplication with a nonnegative number distributes over the maximum.

- And also

$$2^{x\uparrow y} = 2^x \uparrow 2^y \quad , \text{ and}$$

$$(x\uparrow y)^2 = x^2\uparrow y^2 \quad (\text{for } x\,,y \geq 0\,,0) \quad , \text{ and}$$

$$z\downarrow(x\uparrow y) = (z\downarrow x)\uparrow(z\downarrow y) \quad .$$

- And now the reader should prove —with a minimal amount of case analysis—

  $$x^2\downarrow y^2 \;\leq\; x{*}y \;\;\Leftarrow\;\; 0 \leq x{*}y \quad .$$

- Perhaps, we can also learn to handle absolute values more readily, because we have $\mid x \mid \; = x\uparrow -x$. Try to use it to prove the triangular inequality

  $$\mid x{+}y \mid \;\leq\; \mid x \mid + \mid y \mid \quad .$$

- Etcetera.

*　　　　*

*

This really was the beginning of lattice theory. Was it difficult? We hope that most of our readers will say: no! We believe that elementary lattice theory —which goes beyond this note— can and should be taught to reasonable freshmen or, in any case, to sophomores, of computing science and mathematics alike. Many of our colleagues, world-wide, especially computing science colleagues, will shudder at the thought, because lattice theory is regarded far too abstract to be useful or to be teachable to the average student. And abstract stands for frightening, doesn't it? We really must disagree with such a point of view, because —as we tried to show— the game is completely under control by the use of a modest repertoire of simple calculational rules. It is the peaceful calculational style which does away with the fear for abstract things. And also, it is the peaceful calculational style which lets the subject matter sink in much more profoundly than would have been the case otherwise. If still in doubt, remember Newton and Leibniz: they took away the deep difficulties attending the notions of limits and derivatives by . . . proposing a symbolism to denote them and a set of formula rewrite rules to manipulate and . . . to master them. By now these notions are high-school topics.

A high-tech calculation

As members of a High-Tech Society, we sometimes feel obliged to embark on high-tech calculations. Here is one.

When we have a tedious calculation of the form

$$\begin{array}{ll} & x = y \\ \Rightarrow & \quad \vdots \\ & P.x.y \quad , \end{array}$$

there is no need to redo all the work if we want to strengthen the last line with conjunct $P.y.x$. We exploit symmetry, and simply continue our calculation with a step indicated

$$\begin{array}{ll} \equiv & \qquad \{ \text{ Symmetry } \} \\ & P.x.y \ \wedge \ P.y.x \quad . \end{array}$$

The simplest application of the above principle is

$$\begin{array}{lll} & x = y & (0) \\ \Rightarrow & \qquad \{ \text{ calculus } \} & \\ & x \leq y & (1) \\ \equiv & \qquad \{ \text{ Symmetry } \} & \\ & x \leq y \ \wedge \ y \leq x & \\ \equiv & \qquad \{ \text{ calculus } \} & \\ & x = y & (2) \end{array}$$

Comparing lines (0), (1), and (2) we have shown

$$x = y \ \equiv \ x \leq y \quad .$$

And that is what we call High Technology.

2

From sequential programming

Around Bresenham

We consider the task of plotting a curve in the Euclidean plane by marking pixels from a grid covering that plane. More specifically, given a real-valued function f on some finite interval, we wish to mark, for each integer x in that interval,, a pixel with integer coordinates $(x\,,y)$ such that y is as close to $f.x$ as possible. This latter requirement can be formulated as

$$|\, y - f.x \,| \ \leq \tfrac{1}{2} \quad ,$$

or, equivalently, as the conjunction of Qa and Qb, given by

Qa: $\qquad y \ \leq \ \tfrac{1}{2} + f.x$

Qb: $\qquad -\tfrac{1}{2} + f.x \ \leq \ y \quad .$

We furthermore assume that f satisfies a kind of "smoothness" property on the interval, to wit

$$0 \ \leq \ f.(x{+}1) - f.x \ \leq \ 1 \quad ,$$

which can also be formulated as the conjunction of Da and Db, given by

Da: $\qquad f.x \leq f.(x{+}1)$

Db: $f.(x{+}1) \leq 1 + f.x$.

It is this property that suggests that the curve be plotted in the order of increasing (or decreasing) value of x , because the pixel to be marked for $x+1$ is not too far away from the pixel to be marked for x (thus yielding the nicer picture). Thus, with the interval given by two integers A and B , $A \leq B$,we will consider a marking program of the form

```
  $x := A$
; do $x \neq B \rightarrow Mark\ (x\,,y)$ ; $x\ :=\ x{+}1$ od
```

and regard it as our task to extend this program with operations on y such that $Qa \wedge Qb$ is an invariant of the repetition.

The reader may recognize the above problem statement as one from the world of graphics. We wish to emphasize, however, that in this essay neither this specific problem nor its origin is of much concern to us here. We are concerned with the development of a program meeting the specification, by paying attention to uninterpreted formulae only, i.e. without further reference —mental or otherwise— to pixels, pictures, or whatever.

*   *
*

The program above requires two adaptations, one for the initialization of y and one for y 's adjustment in the step. As for the initialization, we cannot say much without taking specificities of f into account. For the time being, we may record it as

$$x, y:\ \ x = A \ \wedge\ Qa \ \wedge\ Qb \ \ .$$

For the step, we consider an adaptation of the form $x\,,y\ :=\ x{+}1\,,y{+}\xi$, and try to find out for what integer values of ξ this maintains $Qa \wedge Qb$. We will deal with the invariances of Qa and Qb in turn.

Qa's invariance

The weakest precondition for $x\,,y\ :=\ x{+}1\,,y{+}\xi$ to establish Qa is

G.ξ: $y{+}\xi \ \leq\ \frac{1}{2} + f.(x{+}1)$.

We now investigate for which (integer) values of ξ the required precondition $G.\xi$ is implied by the actual precondition, which is $Qa \wedge Qb \wedge Da \wedge Db$. To that end, we observe

$$y{+}\xi \ \le\ \tfrac{1}{2} + f.(x{+}1)$$

$\Leftarrow$ $\quad$ { using Qa and the transitivity of $\le$ }

$$\tfrac{1}{2} + f.x + \xi \ \le\ \tfrac{1}{2} + f.(x{+}1)$$

$\equiv$ $\quad$ { arithmetic }

$$f.x + \xi \ \le\ f.(x{+}1)$$

$\Leftarrow$ $\quad$ { using Da and the transitivity of $\le$ }

$$f.x + \xi \ \le\ f.x$$

$\equiv$ $\quad$ { arithmetic }

$$\xi \le 0 \ \ .$$

As a result, statement $x\,,y \ := \ x{+}1\,,y{+}\xi$ “automatically” maintains Qa for $\xi \le 0$, but for other valuse of ξ the statement had better be guarded by $G.\xi$.

Now we expect that, in view of the properties D of f , we will never need to consider increments of y —i.e. values for ξ — outside the range $0\,,1$. This expectation will turn out to be true, and, anticipating that, we can summarize the above analysis as

```
{ $Qa$ } { $Da$ }
if $G.1 \rightarrow x\,,y \ := \ x{+}1\,,y{+}1$
[] $true \rightarrow x \ := \ x{+}1$   (i.e. $x\,,y \ := \ x{+}1\,,y{+}0$ )
fi
{ $Qa$ }   .
```

End Qa’s invariance .

$Qb's$ **invariance**

The weakest precondition for $x\,,y \ := \ x{+}1\,,y{+}\xi$ to establish Qb is

$H.\xi$: $\quad -\frac{1}{2} + f.(x{+}1) \le y{+}\xi$.

As before, we investigate for which values of ξ this is implied by the actual precondition:

$$-\tfrac{1}{2} + f.(x{+}1) \le y{+}\xi$$

$\Leftarrow \quad$ { using Qb and the transitivity of $\le$ }

$$-\tfrac{1}{2} + f.(x{+}1) \le -\tfrac{1}{2} + f.x + \xi$$

$\equiv \quad$ { arithmetic }

$$f.(x{+}1) \le f.x + \xi$$

$\Leftarrow \quad$ { using Db and the transitivity of $\le$ }

$$1 + f.x \le f.x + \xi$$

$\equiv \quad$ { arithmetic }

$$1 \le \xi \quad .$$

In summary, we derived

```
{ $Qb$ } { $Db$ }
if $true \rightarrow x\,,y \;:=\; x{+}1\,,y{+}1$
[] $H.0 \rightarrow x \;:=\; x{+}1$
fi
{ $Qb$ }   .
```

End $Qb's$ invariance .

Now we combine the two fragments above into a single program fragment maintaining both Qa and Qb :

```
{ $Qa \wedge Qb$ } { $Da \wedge Db$ }
if $G.1 \rightarrow x\,,y \;:=\; x{+}1\,,y{+}1$
[] $H.0 \rightarrow x \;:=\; x{+}1$
fi
{ $Qa \wedge Qb$ }   .
```

The only remaining proof obligation is to verify that this program fragment does not suffer from the danger of abortion, i.e. we have to check that the disjunction of the guards is implied by the precondition of the if-statement. In order to examine this we expand the guards (while simplifying them):

$G.1$: $y+\frac{1}{2} \le f.(x+1)$

$H.0$: $f.(x+1) \le y+\frac{1}{2}$,

and, lo and behold, we have $G.1 \vee H.0$, so that there is no danger of abortion. (At the same time, this also confirms our expectation that we need not consider increments of y beyond 0 or 1 .)

Collecting the pieces, we have derived that the program depicted in Figure 0 below plots the curve as demanded.

*   *
*

```
  { $Da \wedge Db$ } { $A \le B$ }
  $x,y$ :  $x = A \wedge Qa \wedge Qb$
  { inv. $Qa \wedge Qb$ }
; do  $x \ne B$  →
          $Mark\ (x, y)$
        ; if  $G.1 \rightarrow x, y := x+1, y+1$
          []  $H.0 \rightarrow x := x+1$
          fi
  od  .
```

Figure 0

So much for the development of this algorithm.

*   *
*

There is, however, a little bit more to be said. The evaluation of the guards $G.1$ and $H.0$ will, in general, demand floating-point arithmetic, and for many functions f there is no escaping it. But for some classes of curves —most notoriously the conic sections— there is an opportunity to transform the algorithm so that its execution will require integer arithmetic only. And this is sometimes considered an advantage. We next show such a transformation for the case of a (specific) hyperbola.

Consider the curve given by

$$0 \leq f.x \ \wedge \ (f.x)^2 - x^2 = C \quad ,$$

for some (large) positive integer constant C . It is the "positive branch" of a hyperbola. We wish to plot it on the interval $[A, B)$, with A and B integers satisfying $0 \leq A \leq B$. The reader may verify that, on this interval, f satisfies properties Da and Db .

We first expand guard $G.1$, seeking to express it with integer subexpressions only:

$$
\begin{array}{ll}
 & G.1 \\
\equiv & \quad \{ \text{ definition of } G \ \} \\
 & y+1 \ \leq \ \frac{1}{2} + f.(x+1) \\
\equiv & \quad \{ \text{ arithmetic } \} \\
 & y+\frac{1}{2} \leq f.(x+1) \\
\equiv & \quad \{ \text{ both sides are nonnegative; for } y+\frac{1}{2} \text{ this is so by } Qb \ \} \\
 & (y+\frac{1}{2})^2 \leq (f.(x+1))^2 \\
\equiv & \quad \{ \text{ arithmetic and definition of } f \ \} \\
 & y^2+y+\frac{1}{4} \leq (x+1)^2+C \\
\equiv & \quad \{ \text{ arithmetic } \} \\
 & \frac{1}{4} \ \leq \ x^2 + 2*x - y^2 - y + C + 1 \\
\equiv & \quad \{ \bullet \text{ by } P \text{ given below } \} \\
 & \frac{1}{4} \leq h \quad .
\end{array}
$$

For the "complementary" guard $H.0$, we find

$$H.0 \;\equiv\; \tfrac{1}{4} \geq h \quad .$$

The reason for introducing the additional invariant

$$P: \qquad h \;=\; x^2 + 2{*}x - y^2 - y + C + 1 \quad ,$$

is that the repeated updating of h is assumed to be less costly than the repeated evaluation of P 's right-hand side for the successive values of x and y . Furthermore note that h is an integer —as demanded— .

Now we have almost succeeded in expressing $G.1$ and $H.0$ in terms of integer expressions only, be it for the occurrence of that $\frac{1}{4}$. Because h is an integer, we can eliminate this $\frac{1}{4}$ at a bargain because of

$$\tfrac{1}{4} \leq h \;\equiv\; 1 \leq h$$

and

$$\tfrac{1}{4} \geq h \;\equiv\; 0 \geq h \quad .$$

Thus, we find

$$G.1 \equiv 1 \leq h \quad \text{and} \quad H.0 \equiv 0 \geq h \quad .$$

We now give the final program at once, leaving the standard proof of the invariance of P to the reader. (All that is needed for this is the axiom of assignment.)

```
  $x\,,y \;:=\; A\,, round.(sqrt.(A^2{+}C))$
; $h \;:=\; x^2 \,+\, 2{*}x - y^2 - y \,+\, C \,+\, 1$
; do $x{\neq}B \;\rightarrow$
        $Mark\ (x\,,y)$
      ; if $1{\leq}h \rightarrow x\,,y\,,h \;:=\; x{+}1\,,\, y{+}1\,,\, h + 2{*}x - 2{*}y + 1$
        [] $0{\geq}h \rightarrow x\,,h \;:=\; x{+}1\,,\, h + 2{*}x + 3$
        fi
  od  .
```

The program text can be further embellished, but we leave it at this. For some interesting details concerning the marking we refer to, for instance, [?].

* *
*

The reader that feels like constructing a plotting algorithm himself, may try to do so for, say, a straight line segment in the "first octant" of the plane. The exercise can be quite rewarding, since he will find himself *designing* the famous algorithm that was *invented* by J.E. Bresenham [**?**].

Acknowledgement

We thank Rob Hoogerwoord and Anne Kaldewaij for their comments.

Postscript

This note was written in 1990, because we felt intrigued but not satisfied by [**?**].

How to bound in Branch and Bound

"Branch and Bound", in particular "Bound", refers to a technique to remove some of the inefficiency that almost inevitably attends most "Backtracking Algorithms". Before dealing with that technique, we first spend a few words on backtracking.

Backtracking algorithms usually enter the picture if a huge class of things has to be explored in order to identify a subclass with certain properties. Typical examples of such huge classes are : all subsets of a given set, all permutations of a number of objects, all partitions of a natural number, all walks through a graph, all tic-tac-toe games, etc. .. If certain members of such a class have to be identified, a safe procedure is to generate *all* members and to check, for each of them individually, whether they have the desired properties. The procedure is safe but its time complexity may grow gigantically, and in many cases there is no cure.

* *
*

As a typical example, let us investigate the set V of all bitstrings of length N , and let us assume that each bitstring x —of whatever length— has a cost $c.x$ associated with it. The problem we consider is to compute the minimal cost of a bitstring in V , i.e. the value of

$$\langle \downarrow x : x \in V : c.x \rangle \quad .$$

The backtracking strategy boils down to the introduction of a function f on the set $prefV$ of all bitstrings of length at most N , defined by

$$f.u = \langle \downarrow x :\ u \mathbin{+\!\!+} x \in V :\ c.(u \mathbin{+\!\!+} x) \rangle \qquad , \quad u \in prefV$$

(Symbol $\mathbin{+\!\!+}$ denotes catenation.)
Then the desired answer is the value of $f.[\,]$. (Symbol $[\,]$ denotes the empty string.)

The recursive scheme governing the computation of f is —proof omitted—

- for $\#u = N$, $\qquad f.u = c.u$

- for $\#u < N$, $f.u = f.(u ⧺ [0]) \downarrow f.(u ⧺ [1])$.

($\#u$ stands for the length of u , and $[b]$ is the singleton list containing element b .)

For the actual computation of f we introduce a function $F(u : \text{list}) : \textsf{int}$ with specification

$$\text{precondition :} \quad u \in \mathit{pref}V$$

$$\text{postcondition :} \quad F = f.u \quad .$$

The code for a program to print target value $f.[\,]$ is now straightforward. It reads

```
    func $F(u : \text{list}) : \textsf{int}$ ;
        { pre :    $u \in \mathit{pref}V$ }
        { post :   $F = f.u$ }
        if $\#u = N \rightarrow F := c.u$
        [] $\#u < N \rightarrow F := F(u ⧺ [0]) \downarrow F(u ⧺ [1])$
        fi
    cnuf
  ; print ($F([\,])$)   .
```

As can be seen from the function definition, the recursion only comes to an end when $\#u = N$, i.e. when a bitstring of full length has been generated. In fact, the above program will generate all 2^N elements of V exactly once, and compute the cost of each of them. And if function c is of a whimsical nature, there is nothing we can do to improve on the program's efficiency.

* *
*

In many practical circumstances, however, cost function c is not whimsical at all. A very frequent situation is that we have

(0) $c.u \leq c.(u ⧺ x)$,

a case that we shall now further investigate. The traditional jargon exploits (0) by a remark like : "there is no need to extend string u if $c.u$ is at least as big as the c-value of any bitstring of length N that the program has computed so far". We will illustrate how to exploit (0) in a non-operational manner.

We generalize function f into a function g defined by

$$g.r.u \;=\; r \downarrow f.u \qquad (\; r \in \mathrm{Int} \;,\;\; u \in \mathit{prefV} \;)$$

Here, parameter r is called the "bound". In terms of g , the desired answer —which was $f.[\,]$ — is

$$g.r.[\,] \qquad \text{for any } \; r \text{ satisfying } \; f.[\,] \le r$$

The great advantage of g over f is that now the value of $g.r.u$ can sometimes be computed without recourse to a "laborious" calculation of $f.u$, viz. when $r \downarrow f.u = r$, in which case $g.r.u = r$. Let us investigate this condition :

$$\begin{array}{ll}
 & r \downarrow f.u = r \\
\equiv & \qquad \{ \text{ property of } f \;\; \} \\
 & r \le f.u \\
\equiv & \qquad \{ \text{ definition of } f \;\; \} \\
 & r \;\le\; \langle \downarrow x :\; u \mathbin{+\!\!+} x \in V :\; c.(u \mathbin{+\!\!+} x) \,\rangle \\
\equiv & \qquad \{ \text{ property of } f \;\; \} \\
 & \langle\, \forall x :\; u \mathbin{+\!\!+} x \in V :\; r \le c.(u \mathbin{+\!\!+} x) \,\rangle \\
\Leftarrow & \qquad \{ \;(0)\; \} \\
 & \langle\, \forall x :\; u \mathbin{+\!\!+} x \in V :\; r \le c.u \,\rangle \\
\Leftarrow & \qquad \{ \text{ predicate calculus } \} \\
 & r \le c.u \quad .
\end{array}$$

As a result we find

- for $r \le c.u$, $\quad g.r.u = r$.

For the remaining case, viz. $c.u < r$, we resort to the recursive scheme for f :

- for $c.u<r \;\wedge\; \#u=N$,

$g.r.u$

$= \qquad \{ \text{ definition of } g \; \}$

$r \downarrow f.u$

$= \qquad \{ \; \#u=N \; \} \; \{ \text{ scheme for } \; f \; \}$

$r \downarrow c.u$

$= \qquad \{ \; c.u<r \; \}$

$c.u$.

- for $c.u<r \;\wedge\; \#u<N$,

$g.r.u$

$= \qquad \{ \text{ definition of } g \; \}$

$r \downarrow f.u$

$= \qquad \{ \; \#u<N \; \} \; \{ \text{ scheme for } \; f \; \}$

$r \downarrow (\, f.(u \mathbin{+\!\!+} [0]) \downarrow f.(u \mathbin{+\!\!+} [1]))$

$= \qquad \{ \; \downarrow \text{ is associative } \}$

$(r \downarrow f.(u \mathbin{+\!\!+} [0])) \downarrow f.(u \mathbin{+\!\!+} [1])$

$= \qquad \{ \text{ definition of } g \; \}$

$g\,.\,r\,.\,(u \mathbin{+\!\!+} [0]) \downarrow f.(u \mathbin{+\!\!+} [1])$

$= \qquad \{ \text{ definition of } g \; \}$

$g\;.\;(g\,.\,r\,.\,(u \mathbin{+\!\!+} [0]))\;.\;(u \mathbin{+\!\!+} [1])$

or —equivalently—
with $\quad t \;=\; g\,.\,s\,.\,(u \mathbin{+\!\!+} [1])$
and $\quad s \;=\; g\,.\,r\,.\,(u \mathbin{+\!\!+} [0])$

$g.r.u = t$.

In summary, the recursive scheme governing the computation of g is

- for $r \leq c.u$, $g.r.u = r$
- for $c.u < r \ \wedge \ \#u = N$, $g.r.u = c.u$
- for $c.u < r \ \wedge \ \#u < N$, $g.r.u = t$

where $t = g\,.\,s\,.\,(u +\!\!+ [1])$
and $s = g\,.\,r\,.\,(u +\!\!+ [0])$.

The corresponding program text that prints the target value reads

```
func  G($r$:int, $u$:string) : int ;
      { pre :   $u \in prefV$ }
      { post :  $G = g.r.u$ }
      if  $r \leq c.u \rightarrow G := r$
      []  $c.u < r \ \wedge \ \#u = N \rightarrow G := c.u$
      []  $c.u < r \ \wedge \ \#u < N \ \rightarrow$
             |[  $s, t$:int ;
                 $s :=$ G($r, u +\!\!+ [0]$)
               ; $t :=$ G($s, u +\!\!+ [1]$)
               ; $G := t$
             ]|
      fi
cnuf
```

; "for some r such that $f.[\,] \leq r$, print(G(r, [ ])) " .

Remark The difference between F and G is that the recursion in G also comes to an end when $r \leq c.u$, i.e. when $c.u$ has become too large or r has become small enough. Therefore, it is beneficial —even quite beneficial— for the efficiency of the program, if the initial value for r in the main call can be chosen as small as possible. And in many applications one can indeed without too much effort find an initial estimate for r that is quite close to target value $f.[\,]$.
End Remark .

* *
*

In practice, one usually wishes to compute a witness for the cheapest solution as well. We shall now show how to do this for our program above.

In order to compute a cheapest bitstring in V , we introduce a global variable z of type string, and strengthen the postcondition of G with the conjunct

$$z \in V \ \wedge \ c.z = G \quad .$$

Because $\mathrm{G}(r,[\,])$ as it occurs in the main call, is the desired minimal value, the string z satisfying this additional postcondition is an appropriate witness indeed.

Now we investigate how the three alternatives of function G can establish this new postcondition

- $r \leq c.u \rightarrow G := r$:

$$\begin{array}{ll} & (z \in V \ \wedge \ c.z = G)\ (G := r) \\ \equiv & \quad \{ \text{ substitution } \} \\ & z \in V \ \wedge \ c.z = r \quad , \end{array}$$

and because this does not follow from the guard $r \leq c.u$ nor from G's precondition $u \in prefV$, we decide that

(*) $z \in V \ \wedge \ c.z = r$
be a precondition of G as well.

- $c.u < r \ \wedge \ \#u = N \rightarrow G := c.u$:

$$\begin{array}{ll} & (z \in V \ \wedge \ c.z = G)\ (G := c.u) \\ \equiv & \quad \{ \text{ substitution } \} \\ & z \in V \ \wedge \ c.z = c.u \\ \Leftarrow & \quad \{ \text{ Leibniz } \} \\ & z \in V \ \wedge \ z = u \quad . \end{array}$$

Because $u \in prefV$ —from pre of G — and because $\#u = N$ — from the guard— we conclude that $u \in V$, so that the above calculated precondition $z \in V \wedge z = u$ of $G := c.u$ is readily established by prefixing $G := c.u$ with statement $z := u$.

- $c.u < r \wedge \#u < N \rightarrow$ block :

For the block in the third alternative we need no further adjustments of G , as may follow from the annotation given below. Please observe that the preassertion of each call of G exactly matches the required additional precondition $(*)$ of G .

```
|[  $s,t$: int ;
      { $z \in V \wedge c.z = r$ , from precondition $(*)$ }
      $s :=$ G($r, u ++ [0]$)
    ; { $z \in V \wedge c.z = s$ , from added postcondition of $G$ }
      $t :=$ G($s, u ++ [1]$)
    ; { $z \in V \wedge c.z = t$ , from added postcondition of $G$ }
      $G := t$
      { $z \in V \wedge c.z = G$ , as required }
]|
```

End • .

Thus, our final program which computes a witness as well, has become

```
func G($r$: int, $u$: string) : int ;
      { pre : $u \in prefV \wedge z \in V \wedge c.z = r$ }
      { post : $G = g.r.u \wedge z \in V \wedge c.z = G$ }
      if $r \leq c.u \rightarrow G := r$
      [] $c.u < r \wedge \#u = N \rightarrow z := u$ ; $G := c.u$
      [] $c.u < r \wedge \#u < N \rightarrow$
```

```
                |[  $s,t$:int ;
                    $s := \mathrm{G}(r, u +\!\!+ [0])$
                  ; $t := \mathrm{G}(s, u +\!\!+ [1])$
                  ; $G := t$
                ]|
          fi
      cnuf
    ; “for some $r$ and $z$ such that $z \in V$ , $c.z = r$ , and $f.[\,] \leq r$ ,
      print(G($r$, [ ])) ; print($z$) ” .
```

* *

*

Branch and Bound, and Backtracking, belong to what the field has called “Combinatorial Algorithms. Those algorithms are usually presented in a very operational manner. Backtracking is explained as tree traversal and bounding as some sort of pruning. And the algorithms are presented in terms of pictures rather than with formulae. We do not really like such explanations, in particular if one realizes that the mathematical crux of bounding, viz. the transition from function

$f.u$

to $r \downarrow f.u$

is so beautifully simple.

We learned this technique from our colleagues Anne Kaldewaij and Rob Hoogerwoord. Kaldewaij in his book [?] has considerably raised the standards for dealing with Combinatorial Algorithms, and it should be pointed out —for methodological reasons— that this rise of standard has become possible by the emergence of nice calculational styles of functional programming as, for instance, the one laid down in Hoogerwoord’s [?].

The Binary Search revisited

The Binary search is a beautiful, simple, efficient, and —hence— well-known little algorithm for searching. Most computing scientists and programmers will be familiar with it. Unfortunately, there are two observations that cast some doubt on the level of familiarity. First, many programmers still have a relatively hard time writing down correct program code for it, this in spite of the fact that it concerns an algorithm of just a few lines. Second, most programmers believe that the algorithm only works for searching in sorted arrays, which reveals a misunderstanding. (A long time ago, we ourselves, too, used to sell the binary search to our students by first drawing an analogy with searching for a word in a dictionary. Afterwards, we learned to judge this as an educational blunder.) The purpose of this note is to remedy the situation.

*   *
        *

Our starting point is that we are given a function or an array $f[a..b]$, $a<b$, of elements of some kind, such that the two outer elements $f.a$ and $f.b$ are in some relation Z to each other, a fact that we denote by $a\,Z\,b$. The problem is to find a pair of neighbouring elements of f that are in the relation Z . (Of course, such a pair need not exist; we will address this problem later on.) More precisely, we wish to construct a program with postcondition

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

on the premise that the precondition implies

$$a<b \;\wedge\; a\,Z\,b \quad .$$

The procedure for constructing such a program is quite standard. In view of the shapes of the pre- and the postcondition we try to establish R by means of a repetition with invariants $P0$ and $P1$, given by

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

$P1$: $\quad x\,Z\,y \quad ,$

and with bound function $y-x$. Our program thus gets the form

```
    { $a<b \ \wedge \ a\,Z\,b$ }
    $x\,,y \ := \ a\,,b$
    { inv $P0 \wedge P1$ } { bnd $y-x$ }
  ; do $y \neq x+1 \rightarrow$ shrink $y-x$ under invariance of $P0 \wedge P1$ od
    { $R$ } .
```

And indeed, by construction, $P0 \wedge P1 \wedge y=x+1 \Rightarrow R$.

For the shrinking of $y-x$ in the body of the repetition we investigate an increase of x and a decrease of y. (We do so since our problem is highly symmetric in x and y .) On the assumption that $x<h$, for some h still to be determined, assignment $x := h$ increases x . We now find out under what circumstances this assignment maintains $P0 \wedge P1$. As for the invariance of $P0$, we observe

$P0\ (x := h)$

$\equiv$ { definition of $P0$ }

$a \leq h \ \wedge \ h<y \ \wedge \ y \leq b$

$\equiv$ { $P0$ and $x<h$ are preconditions to $x := h$,
hence $a<h \ \wedge \ y \leq b$ }

$h<y$.

So we require h to satisfy not just $x<h$ but also $h<y$, and hence

$$x<h \ \wedge \ h<y \ .$$

Note Fortunately, such an h exists because from $P0$ and the guard $y \neq x+1$ of the repetition we conclude that $x+2 \leq y$.
End

Having settled the invariance of $P0$ under $x := h$, we now turn our attention to $P1$. We observe

$P1\ (x := h)$

$\equiv$ { definition of $P1$ }

$h\,Z\,y$.

Since this latter condition does not in general follow from $P1$ or $P0$, we plug it in as a guard to $x := h$.

We next address a decrease of y . Because we already have an h such that $h < y$, we propose $y := h$. By the problem's symmetry in x and y , our program now becomes

```
  { $a < b \wedge a Z b$ }
  $x, y := a, b$
  { inv $P0 \wedge P1$ } { bnd $y-x$ }
; do $y \neq x+1$ →
        { $x+2 \leq y$ }
        “ $h$ such that $x < h \wedge h < y$ ”
      ; if $h Z y \rightarrow x := h$
        [] $x Z h \rightarrow y := h$
        fi
  od
  { $R$ }
```

Prog0

Remains the question of termination of this program. It does terminate if we can ensure that at least one of the guards of the if-statement evaluates to *true*. This will in general not be the case, but it is if relation Z satisfies

(0) $\quad x Z y \;\Rightarrow\; h Z y \vee x Z h \qquad (\forall x, y, h)$

(The antecedent is the valid precondition $P1$; the consequent is the disjunction of the guards.)

While adopting this condition on Z , we can even ensure very fast termination by choosing h to be equal to $(x+y)\ \underline{\mathsf{div}}\ 2$, which thanks to precondition $x+2 \leq y$ indeed establishes $x < h \wedge h < y$. And thus, binary search is born!

* *

*

Our program computes an x satisfying postcondition R . Of course, there may be many x's satisfying R . And here we arrive at what we think is a distinguishing feature of the binary search, namely that

> it is beyond our control which x
> satisfying the postcondition
> will be generated.

(This is in sharp contrast to linear searches where a fully specified value is computed.)

In order to substantiate this feature we consider the instantiation $a, b := 0, 100$ and $Z := true$, meeting Prog0's precondition and (0). We then obtain

```
  $x, y := 0, 100$
; do $y \neq x+1$ $\rightarrow$
        $h := (x+y)$ div 2
      ; if $true \rightarrow x := h$
        [] $true \rightarrow y := h$
        fi
  od
  $\{ 0 \leq x \wedge x < 100 \}$ ,
```

which is a highly nondeterministic program. And the reader may check, in whatever way he likes, that, indeed, it can generate *any* value x such that $0 \leq x \wedge x < 100$.

We will return to this distinguishing feature at a later stage.

* *
*

Meanwhile, some readers may have been puzzled by the “weird” condition (0). But it is not weird at all; when taking the contrapositive of (0), we get

(0') $\quad x\,(\neg Z)\,h \ \wedge \ h\,(\neg Z)\,y \ \Rightarrow \ x\,(\neg Z)\,y$,

and this just expresses that Z's complement relation $\neg Z$ is transitive.

Here are some of such relations Z (for array f of the appropriate type) :

$x\,Z\,y \ \equiv$

- $f.x \neq f.y$
- $f.x < f.y$
- $f.x \leq A \ \wedge \ A \leq f.y$
- $f.x * f.y \leq 0$
- $f.x \vee f.y$
- $\neg Q.x \wedge Q.y$, for any Q .

The reader can easily verify that they all satisfy (0) (or (0')).

Hint The first example —$f.x \neq f.y$— is a very appropriate choice when novices are to be introduced to the binary search.
End Hint .

* *
*

The last example in the above list —$\neg Q.x \wedge Q.y$— is a frequently occurring one, and we shall now deal in more detail with its probably best-known instance, viz. $Q.i \ \equiv \ C < f.i$. That is, given

- integer array $f[a..b]$ with $a < b$
- integer constant C satisfying $f.a \leq C \ \wedge \ C < f.b$,

our binary search becomes —see remark below—

```
  { $a<b$ } { $f.a \leq C \ \wedge\ C < f.b$ }
  $x\,,y\ :=\ a\,,b$
  { inv $P0$:   $a \leq x\ \wedge\ x<y\ \wedge\ y \leq b$
        $P1$:   $f.x \leq C\ \wedge\ C<f.y$ } { bnd $y-x$ }
; do  $y \neq x{+}1$  $\rightarrow$
        $h\ :=\ (x{+}y)$ div $2$ { $a \leq x < h < y \leq b$ }
      ; if  $f.h \leq C \rightarrow x := h$
        []  $C < f.h \rightarrow y := h$
        fi
  od
  { $a \leq x\ \wedge\ x<b$ } { $f.x \leq C\ \wedge\ C<f.(x{+}1)$ }
```

Prog1

Remark Actually, we arrived at the above program not by instantiating our original program scheme Prog0 , but by just rederiving it for the current relation $f.x \leq C \wedge C < f.y$. That derivation is so simple that it can be done by heart: guided by the invariants $P0$ and $P1$, the program can be written down at once.
End Remark .

Note that the precondition of Prog0 (and hence of Prog1) is the *only* property of f used in the derivation. Thus we hope to have shown that the binary search has a right of existence outside the realm of sorted arrays.

However, if f *is* sorted in ascending order, Prog1 comes in very handy to record the presence of C in $f[a..b]$, namely by postfixing Prog1 with the assignment

$$present\ :=\ (f.x = C)\ \ .$$

And, in fact, it is only for the conclusion that C is absent from f — $present \equiv false$— that we use f's ascendingness.

*   *
    *

We have derived Prog0 (and Prog1) under a precondition — $a<b\ \wedge$ $a\,Z\,b$ — . In many practical situations, however, this condition need not be satisfied. There are two general ways out. The first way out is to try to solve the problem separately for the case in which the precondition does not hold.

The more elegant way out is based on the observation that Prog1 does not inspect the array elements $f.a$ or $f.b$ at all —see precondition $a<h\ \wedge\ h<b$ to the inspection of $f.h$ — . This implies that their actual values are completely irrelevant to the (outcome of the) computation : they are thought variables mainly.

We exploit this observation in the following way. Suppose all we know about $f[a..b]$ is that it is ascending and that $a\leq b{+}1$ (the array may be empty). We define thought values $f.(a{-}1)$ and $f.(b{+}1)$ in such a way that

- $f[a{-}1\ ..\ b{+}1]$ is ascending, and
- $f.(a{-}1)\leq C\ \ \wedge\ \ C<f.(b{+}1)$, and

since in addition $a{-}1<b{+}1$ holds, we have thus laid down precisely the required precondition of Prog1 , be it with the instantiation $a\,,b\ :=\ a{-}1\,,b{+}1$. The solution to our problem to record the presence of C in array $f[a..b]$ now reads

```
  $x\,,y\ :=\ a{-}1\,,b{+}1$
; do  $y\neq x{+}1$  $\rightarrow$
        $h\ :=\ (x{+}y)$ div $2$
        { $a\leq h\wedge h\leq b$ , hence $f.h$ is defined }
      ; if  $f.h\leq C\rightarrow x:=h$
        []  $C<f.h\rightarrow y:=h$
        fi
  od
  { $a{-}1 \le x \ \wedge\ x < b{+}1$ } { $f.x \le C \ \wedge\ C < f.(x{+}1)$ }
; if $a{-}1 = x \rightarrow$  { $C < f.a$ } $present := false$
  [] $a{-}1 < x \rightarrow$ { $a \le x \ \wedge\ x \le b$ , hence $f.x$ is defined }
                  $present := (f.x = C)$
  fi
```

The technique applied —adding thought values so as to establish a required precondition— is suitable in many other examples as well.

*   *
   *

We conclude this note on the binary search with two exercises and a question. The exercises are

(i) Given that integer array $f[a..b]$, $a \le b$, is the concatenation of an increasing and a decreasing sequence, find the top (maximum). More precisely, design a program that computes a value M such that

$$\begin{array}{ll} & a \le M \wedge M \le b \\ \wedge & f[a\ ..\ M\ ] \quad \text{is increasing} \\ \wedge & f[M\ ..\ b\ ] \quad \text{is decreasing} \end{array}$$

(Note : $f[a\ ..\ M\ ]$ is increasing $\equiv$ $\langle \forall i\colon\ a \le i < M\colon\ f.i < f.(i{+}1) \rangle$)
.

(ii) Given that integer array $f[a\ ..\ b\ ]$, $a < b$,

- is the concatenation of two increasing sequences
- contains a dip, i.e.

$$\langle\ \exists i\colon\ a \le i < b\colon\ f.i > f.(i{+}1)\ \rangle \quad , \quad \text{and}$$

- satisfies $f.a > f.b$,

design a program to compute a dip .

Both exercises can be solved using a binary search.

If, however, in exercise (ii) the given $f.a > f.b$ is dropped, we no longer know of a way to solve this problem with the binary search technique, not even if we introduce a thought value $f.(a-1)$ such that $f.(a-1) > f.b$. The reason why the introduction of such a thought value does not help here is that it may introduce a second dip at $a-1$, and as we said before, it is now beyond our control which dip will be computed by the binary search : it could be the required one, but it could also be the thought dip that we ourselves introduced.

The as yet unsolved problem that remains is to identify the *precise* conditions for a binary search to be applicable.

3

On Multiprogramming

A preamble

If there is one branch of computer programming where the guideline "Keep it Simple, Please" is to be taken highly seriously, it certainly is the branch called "parallel programming" or "multiprogramming". Why is multiprogramming so difficult? There are many reasons, and just one of them is exemplified by the following.

Consider the simple, meaningless, random program

```
  $x := y+1$
; $x := y^2$
; $x := x-y$  .
```

It consists of just three assignment statements. If we start it in an initial state such that $x=7 \wedge y=3$, it will deliver $x=6 \wedge y=3$ as final answer. This is easily checked.

Also consider the equally simple and meaningless program

```
  $y := x+1$
; $y := x^2$
; $y := y-x$  .
```

When started in $x=7 \wedge y=3$, it yields $x=7 \wedge y=42$.

Now let us run these two programs in parallel. Such a parallel execution boils down to selecting an arbitrary interleaving of the three statements of the x-program and the three statements of the y-program, and then executing the six statements in the order selected. Because there are 20 possible interleavings, the parallel execution of the two programs can give rise to 20 different computations. Here are the possible final values for x and y .

x	y
-4032	8128
-3968	4032
-600	1225
-575	600
-72	153
-63	72
-1	2
2	-1
6	30
20	380
56	3080
132	12
240	-224
496	-240
1722	42
2352	-2303
4753	-2352
5112	72
6480	-6399
13041	-6480

The moral of the example is clear: by the parallel composition, the two extremely simple programs have turned into a horrendous monster. The number of possible computations has exploded. If more component programs or less simpler ones join the game, the phenomenon becomes much more pronounced. As a result, any attempt to come to grips with a multiprogram by considering all the individual computations that can be evoked by it, is far beyond what we can imagine or grasp, and is —in practice and in principle— doomed to fail. In short, operational reasoning should be out!

* *
*

If operational reasoning is to be out, we have to resort to formalism. From sequential programming we have learned that the Hoare-triple semantics [?] and the predicate transformer semantics [?] do away with individual computations in a highly effective manner. Moreover, these —very similar— semantics have created eminent opportunity for the formal *derivation* of programs. The formalism that faithfully imports these virtues into the area of multiprogramming is the theory of Owicki and Gries [?] [?] , which we shall explain next.

Remark The theory of Owicki and Gries, said the computing community, has been designed for the a posteriori verification of multiprograms, and is of no use for the *derivation* of such programs. The only thing we can say to this is that this is not true. The main reason for us to write this note is to show how the Owicki-Gries theory can support a method for the formal derivation of multiprograms. Of course, in a short note like this we can show only some of the flavour of such a method. Reasonable introductory accounts can be found in [?] and in [?] , treatises of former students of ours.
End Remark .

The theory of Owicki and Gries

We consider a system of ordinary sequential programs. We call it a multiprogram. The constituent sequential programs we call the (multiprogram's) components. The multiprogram as a whole has a precondition (describing an initial state from which the component programs can start their execution). If all components terminate, the multiprogram has a postcondition as well (describing the combined final states of the components).

We now consider the components to be annotated with assertions, in the way we are used to for sequential programs. The intended operational interpretation of the annotation is the following: if execution of a component has reached an assertion P , then the state of the system as a whole satisfies P . With this in mind, the subsequent proof rules for the correctness of an annotation may sound "sweetly reasonable".

By definition, the annotation is correct whenever for *every* assertion in *every* component, it holds that

(i) that assertion is "locally correct", and

(ii) that assertion is "globally correct" .

Re (i) For proving the local correctness of an assertion P in a component program, we just follow the rules of sequential programming. We can proceed as if this program were to be dealt with in isolation. There are two cases.

- If P is the preassertion of the component program, we have to ensure that it is implied by the precondition of the entire multiprogram.
- If P is the postassertion of a statement S with preassertion Q , we have to ensure the validity of Hoare-triple

$$\{ Q \} S \{ P \} \ .$$

End

Re (ii) For proving the global correctness of an assertion P in a component program, we have to do more work, because now all other components have to be taken into account. Namely, we have to prove for *each* $\{Q\}S$ —i.e. a statement S with preassertion Q — taken from a *different* component, the validity of the Hoare-triple

$$\{ P \wedge Q \} S \{ P \} \ .$$

In words, this boils down to showing that assertion P in the one component cannot be falsified by the statements of the other components.

In the jargon we can often hear P's global correctness phrased as " P is stable".
End

Finally, if the multiprogram has a postcondition, we have to show that all components terminate and that the postcondition is implied by the conjunction of the postassertions of the individual components.

And this, in fact, is all there is to be said about the theory of Owicki and Gries. We can summarize it quite succinctly by

> the annotation is correct
> $\equiv$
> each assertion is established by the component in which it occurs,
> and
> it is maintained by the statements of all other components.

*   *
*

There is another notion that is very important for discussing multiprograms, viz. the notion of a system invariant. A relation P is a system invariant whenever

- it is implied by the precondition of the multiprogram (P holds initially), and
- for each $\{Q\}S$ in each component, we have

$$\{ \ P \wedge Q \ \} \ S \ \{ \ P \ \}$$

(P is not falsified by any statement from any component).

As a result, an invariant can be added as a conjunct to *each* assertion, and therefore we can afford the freedom of writing it *nowhere* in our annotation. This is of great importance for the clarity of exposition and for the economy of reading and writing.

*   *
*

Example and Exercise

We return to the example at the beginning of this section on multiprogramming :

$$\begin{array}{lllll}
\mathit{Pre}: & x=7 \;\wedge\; y=3 & & & \\
 & & & & \\
\mathit{Compx}: & \; x := y+1 & \qquad & \mathit{Compy}: & \; y := x+1 \\
 & ;\; x := y^2 & & & ;\; y := x^2 \\
 & ;\; x := x-y & & & ;\; y := y-x \\
 & & & & \\
\mathit{Post?}: & \langle\, \exists i :: x+y=i^2 \,\rangle & & &
\end{array}$$

Certainly, both components terminate. The exercise is to prove that

Post: $\langle\, \exists i :: x+y=i^2 \,\rangle$

is a correct postcondition. This, most definitely, is not an easy exercise. (Giving a posteriori proofs usually isn't easy; designing programs and their correctness proofs —hand in hand— is much more doable, and far more rewarding.) The problem is that we have to invent, or at best develop, annotation that enables us to draw the desired conclusion. At this point of presentation, we simply reveal an adequate annotation, and then embark on a proof of its correctness. Unfortunately, the exercise is already that complicated that we need to introduce auxiliary variables (thought variables) to carry out the proof. The annotated multiprogram, extended with the thought variables, is —see explanation below—

Pre: $\quad x=7 \wedge y=3 \wedge \neg p \wedge \neg q$

Inv?: $\quad p \wedge q \Rightarrow \langle \exists i :: x+y=i^2 \rangle$

```
Compx:     { $\neg p$ } $x := y+1$              Compy:     $y := x+1$
         ; { $\neg p$ } $x := y^2$                       ; $y := x^2$
         ; { $\langle \exists i :: x=i^2 \rangle$ }                 ; $y, q := y-x, true$
           $x, p := x-y, true$                          { $q$ }
           { $p$ }
```

Post?: $\quad \langle \exists i :: x+y=i^2 \rangle$

For reasons of symmetry between the components we have annotated *Compy* less lavishly than *Compx* . We laid down a postassertion p to *Compx* and a postassertion q to *Compy* , with the intent that their conjunction $p \wedge q$ imply *Post* . That is how the need for invariant *Inv* arose. We leave it as an exercise to the reader to check the correctness of the annotation —which is not difficult—. By way of example, we shall demonstrate the invariance of *Inv* , i.e. of

$$p \wedge q \Rightarrow \langle \exists i :: x+y=i^2 \rangle \quad .$$

We have to show that it holds initially —which it vacuously does— , and that no statement of the multiprogram violates it. By the symmetry in *Inv* and in the components, we can confine our attention to the statements of *Compx* :

Re $\quad \{ \neg p \}\ x := y+1$

Our proof obligation is

$$\{ \mathit{Inv} \wedge \neg p \}\ x := y+\mathit{1}\ \{ \mathit{Inv} \} \quad ,$$

the correctness of which follows from the following calculation:

$$\mathit{Inv}(x := y+1)$$
$$\equiv \qquad \{ \text{ definition of } \mathit{Inv} \ \}$$
$$p \wedge q \Rightarrow \langle \exists i :: y+1+y=i^2 \rangle$$

$$\begin{array}{ll} \equiv & \{ \ \neg p \text{ , from precondition } Inv \wedge \neg p \ \} \\ & true \ . \end{array}$$

End

Re $\{ \neg p \} \ x := y^2$

Similarly.

End

Re $\{ \langle \exists i :: x = i^2 \rangle \} \ x , p \ := \ x - y , true$

Our proof obligation is

$$\{ \ Inv \ \wedge \ \langle \exists i :: x = i^2 \rangle \ \} \ \ x , p \ := \ x - y , true \ \ \{ \ Inv \ \} \ \ ,$$

the correctness of which follows from

$$\begin{array}{ll} & Inv(x , p \ := \ x - y , true) \\ \equiv & \{ \text{ definition of } Inv \ \} \\ & true \wedge q \ \Rightarrow \ \langle \exists i :: \ x - y + y = i^2 \ \rangle \\ \equiv & \{ \text{ simplification } \} \\ & q \Rightarrow \langle \exists i :: x = i^2 \rangle \\ \equiv & \{ \ \langle \exists i :: x = i^2 \rangle \text{ , from the precondition of the statement } \} \\ & true \ . \end{array}$$

End

* * *

Finally, we have to spend a word on our program notation. We write down our sequential programs just the way we are used to, viz. in the Guarded Command Notation. There is one point, though, that needs extra emphasis, namely that an alternative construct like

$$\textsf{if } B \rightarrow \textsf{skip fi}$$

is our main tool for achieving synchronization. To all intents and purposes, it is equivalent to

$$\textsf{do } \neg B \rightarrow \textsf{skip od} \quad ,$$

i.e. "waiting until B is true". In the literature, we often find it denoted as **await** B . Its semantics, in Hoare-triple format, is given by

$$\{\ B \Rightarrow R\ \}\ \textsf{if } B \rightarrow \textsf{skip fi}\ \{\ R\ \} \quad .$$

Total Deadlock and the Multibound

In most multiprogramming problems, the component programs have to cooperate on a task. Correctness of the cooperation usually requires some sort of synchronization among the components. The synchronization becomes manifest by the occurrence of statements like $\textsf{if } B \rightarrow \textsf{skip fi}$ (or $\textsf{await } B$) in the program texts. The effect of their execution is that the normal computational "progress" of a component is blocked until guard B is "found" to be true. However, with the incorporation of these "blocking" statements, a next and very serious complication has entered the game.

While, on the one hand, these statements are necessary for a temporary blocking of a component's progress, they introduce the danger of infinite blocking on the other hand. As a consequence, we are put up with a new kind of proof obligation, namely to show that each individual blocking statement that we introduce into a component program, will not lead to an infinite blocking of that component. This proof obligation is of a completely different nature from the proof obligation we had for the correctness of annotation. In fact, showing what the jargon has called "individual progress" or "liveness", is, in full generality, so difficult that computing science has not yet succeeded in making this problem technically feasible.

However, there are some special circumstances in which individual progress can be shown quite easily. One of these circumstances is when we have the following scenery. Consider, as an example, a three-component multiprogram that, projected on the variables x , y , and z , has the form

$$* [\ x := x{+}1\ ] \qquad\qquad * [\ y := y{+}1\ ] \qquad\qquad * [\ z := z{+}1\ ]$$

($* [\ S\ ]$ is short for do $true \rightarrow S$ od .)

The rest of the code of these components is such that it satisfies some synchronization requirement, the precise nature of which does not concern us here. Now suppose that it so happens that this multiprogram maintains as a system invariant

MB: $\quad x \leq y{+}K \ \wedge\ y \leq z{+}L \ \wedge\ z \leq x{+}M$,

for some constants K , L , and M . We then observe that if, due to the (unknown) synchronization protocol, one of the components comes to a definitive halt in one of its blocking statements, so will the others. As a result, the multiprogram as a whole can exhibit only two scenarios as far as progress is concerned, to wit

> either *all* components get stuck forever
> —this is called "total deadlock"—
>
> or *each* individual component makes progress.

So, in the presence of a "multibound" like MB , individual progress can be demonstrated by showing the absence of the danger of total deadlock. And the nice thing about this is that the latter can be done using the theory of Owicki and Gries. (We do not need to go into this for the purpose of this note.)

*   *
*

So much for these notational and conceptual issues. We now turn our attention to some examples of multiprogram derivation.

The Parallel Linear Search

The problem of the parallel linear search is a nice little paradigm which was first communicated to us in the mid 1980s by Ernst-Rüdiger Olderog. It serves as a running example in [?]. In [?], we can find a first formal derivation of the algorithm, a derivation which is carried out in terms of the UNITY-formalism [?].

The problem is as follows. Given is a boolean function f on the integers, such that

(0) $\langle \exists i :: f.i \rangle$.

Our goal is to design a terminating multiprogram with two components that "finds" an integer with a true f-value. The idea is that one component "searches" through the nonnegative integers and the other component through the nonpositive ones. More precisely, we have to design a terminating multiprogram with the following specification

Pre: $x=0 \wedge y=0$

Inv: $0 \leq x \wedge y \leq 0$

Post: $f.x \vee f.y$

CompA: do $\ldots \rightarrow x := x{+}1$ od $\{ RA \}$

CompB: do $\ldots \rightarrow y := y{+}1$ od $\{ RB \}$.

At this point, there are two immediate concerns. One is that we have to choose —correct!— assertions RA and RB such that

(1) $RA \wedge RB \Rightarrow f.x \vee f.y$.

The other is that we have to ensure termination for both components.

In order to satisfy (1) we choose both RA and RB equal to $f.x \vee f.y$.

Remark In view of the symmetry between the components, this choice for RA and RB is quite reasonable. Besides, it is a good heuristics to choose the annotation as weak as possible, if there is a choice. The reason for this is that the annotation can always be strengthened later on, should the need arise —see [?] or [?]— . In our example, these heuristics rule out the stronger choice $f.x$ for RA , and $f.y$ for RB

.
End Remark .

In order to ensure the local correctness of RA , i.e. of $f.x \vee f.y$, we can choose $\neg f.x$ as a guard of the repetition of $CompA$, but this one is too weak to ensure termination of $CompA$: given (0) does not guarantee the existence of a nonnegative x for which $f.x$ holds. Therefore, we *must* resort to a stronger guard. Let it be $\neg f.x \wedge \neg d$.

We thus arrive at the first approximation of the multiprogram to be designed.

```
Pre:     $x = 0 \wedge y = 0$

Inv:     $0 \leq x \wedge y \leq 0$

Post:    $f.x \vee f.y$

CompA:   do $\neg f.x \wedge \neg d$  $\rightarrow$
                { $\neg f.x$ , see later in Note 1 }
                $x := x+1$
         od
         { $f.x \vee f.y$ }

CompB:   do $\neg f.y \wedge \neg c$  $\rightarrow$
                { $\neg f.y$ }
                $y := y-1$
         od
         { $f.x \vee f.y$  , see Note 0 below }
```

In Note 0 we now examine the —local and global— correctness of assertion $f.x \vee f.y$ in $CompB$. (The correctness of $f.x \vee f.y$ in $CompA$ follows by symmetry.)

Note 0 "$f.x \vee f.y$ in $CompB$ "

L: For $f.x \vee f.y$ in $CompB$ to be locally correct it suffices that

$$\neg(\neg f.y \wedge \neg c) \;\Rightarrow\; f.x \vee f.y$$

which —by predicate calculus— equivales

$$c \;\Rightarrow\; f.x \vee f.y \quad .$$

We shall meet this condition by adopting

PA: $\quad c \Rightarrow f.x \vee f.y$

as a system invariant, and —by symmetry— also

PB: $\quad d \Rightarrow f.x \vee f.y$.

We will address the invariance of *PA* and *PB* in a moment.

G: For $f.x \vee f.y$ in *CompB* to be globally correct we have to show the correctness of the Hoare-triple

$$\{ f.x \vee f.y \}\ x := x{+}1\ \{ f.x \vee f.y \} \quad .$$

To that end we observe

$$\begin{array}{ll} & f.x \vee f.y \Rightarrow (f.x \vee f.y)(x := x{+}1) \\ \equiv & \qquad \{ \text{ substitution } \} \\ & f.x \vee f.y \Rightarrow f.(x{+}1) \vee f.y \\ \Leftarrow & \qquad \{ \text{ nothing is known about } f.(x{+}1) \ \} \\ & f.x \vee f.y \Rightarrow f.y \\ \Leftarrow & \qquad \{ \text{ predicate calculus } \} \\ & \neg f.x \quad . \end{array}$$

And here we encounter the reason why it is pleasant to have $\neg f.x$ as a valid precondition to $x := x{+}1$. That is why we add it .

End Note 0 .

Note 1 “$\neg f.x$ in *CompA* ”

L: The local correctness follows from the guard

G: The global correctness follows, because *CompB* changes neither x nor f .

End Note 1 .

At this point we are left with the care for the invariance of PA and PB , and with the care for termination of both components. The two cares largely coincide. We first observe that

$$\langle\ \forall i\colon 0 \le i < x\colon \neg f.i\ \rangle \quad \wedge \quad \langle\ \forall i\colon y < i \le 0\colon \neg f.i\ \rangle$$

is a system invariant: this is easily checked. With the given (0) , i.e.

$$\langle\ \exists i :: f.i\ \rangle\ ,$$

we therefore conclude that at least one of the two components terminates. If $CompA$ terminates, it has established $f.x \vee f.y$ —see the annotation— and it can now enforce termination of $CompB$ as well by falsifying the latter's guard $\neg f.y \wedge \neg c$, which can be done by —just— $c := true$. Fortunately this does not interfere with the requirement that

PA: $\quad c \Rightarrow f.x \vee f.y$

be an invariant, thanks to the validity of $f.x \vee f.y$. Thus we arrive at our next (and last) approximation of the multiprogram to be designed. We give the fully annotated program text, and leave to the reader the formal proof of the invariance of PA (and of PB).

Pre: $\quad x=0 \ \wedge\ y=0 \ \wedge\ \neg c \ \wedge\ \neg d \ \wedge\ \langle\ \exists i :: f.i\ \rangle$

Inv: $\quad 0 \le x \ \wedge\ y \le 0 \ \wedge\ PA \ \wedge\ PB$

$Post$: $\quad f.x \vee f.y$

$CompA$:

```
      do $\neg f.x \wedge \neg d \rightarrow \{\ \neg f.x\ \}\ x := x+1$ od
      $\{\ f.x \vee f.y\ \}$
    ; $c := true$
      $\{\ c \wedge PA$ , hence $f.x \vee f.y\ \}$
```

$CompB$:

```
      do $\neg f.y \wedge \neg c \rightarrow \{\ \neg f.y\ \}\ y := y-1$ od
      $\{\ f.x \vee f.y\ \}$
    ; $d := true$
      $\{\ f.x \vee f.y\ \}$
```

A problem of Phase Synchronization

The problem of phase synchronization for two machines is one of the simplest problems enabling us to get across some of the flavour of a method for the formal derivation of multiprograms. (For many more examples, we refer to [**?**] and [**?**].) An interesting by-product of the problem is that it can serve to illustrate how "parallelism" can be traded for storage space. Namely, we shall present two solutions, one of which allows "more parallelism" than the other, be it at the expense of more storage space. We also illustrate how one can consciously choose between the two options.

$*$ $*$
$*$

We consider the two-component multiprogram given by

```
 Comp0:  *[ S ]                    Comp1:  *[ T ]
```

(Notation $*[\ S\ ]$ is short for do $true \rightarrow S$ od .)
$Comp0$ and $Comp1$ are given "to run in parallel". Moreover, each individual execution of S and each individual execution of T are guaranteed to terminate. The synchronization task ahead of us is to see to it that the number of completed S's and the number of completed T's are "never too far apart".

In order to make the synchronization requirement precise, we introduce two *fresh* auxiliary integer variables x and y , and adjust the multiprogram proper in the following way:

```
Pre:     $x=0 \wedge y=0$
Comp0:   $*[\ \{\ x \leq y\ ,\ ?\ \}\ S\ ;\ x := x+1\ ]$
Comp1:   $*[\ \{\ y \leq x\ ,\ ?\ \}\ T\ ;\ y := y+1\ ]$
```

This, now, is our formal specification, to the extent that we are now supposed to understand that it is our task to superimpose on these program texts additional code so as to accomplish that

- in $Comp0$, $x \leq y$ is a valid precondition to S , and, similarly,
- in $Comp1$, $y \leq x$ is a valid precondition to T .

Remark Variables x and y , and the operations on them, have been introduced to specify the synchronization problem. It goes without saying that in the further development of the multiprogram, no further changes of x or y are allowed, because they would defeat the purpose for which these variables and the operations on them were introduced in the first place. Inspections of them are, however, harmless.
End Remark .

* *
*

Before we embark on a derivation, we observe that no matter which solution we end up with, relation $x \leq y{+}1$ will be a system invariant. This is so because

- it holds initially
- increments of y in $Comp1$ do not falsify it
- assignment $x := x{+}1$ will have precondition $x \leq y$. (Here we use that no other assignments to x are allowed in the further development.)

By symmetry, also $y \leq x{+}1$ will be an invariant, and hence

MB: $x \leq y{+}1 \wedge y \leq x{+}1$.

This is a perfect multibound for our multiprogram, and as a result our only proof obligation with respect to individual progress will be to show the absence of total deadlock, *no matter* what will be our ultimate solution.

* *
*

Now let us start our derivation. We have to ensure the correctness of assertion $x \leq y$ in $Comp0$. Its global correctness is for free: $y := y+1$ does not falsify it. As for its local correctness, we observe that it is implied by the precondition Pre of the multiprogram. So it suffices to make $x \leq y$ a *loop* invariant of $Comp0$. $Comp1$ is dealt with symmetrically. Thus we arrive at

```
Pre:      $x=0 \wedge y=0$
Comp0:    $*[ \; \{ \; x \leq y \; \} \; S \; ; \; x := x+1 \; \{ \; x \leq y \; , ? \; \} \; ]$
Comp1:    $*[ \; \{ \; y \leq x \; \} \; T \; ; \; y := y+1 \; \{ \; y \leq x \; , ? \; \} \; ]$
```

Approximation 0

Remark By design, this approximation satisfies our original formal specification. But moreover, it acts as the formal specification for what follows. The approximation above is the precise interface between the past and the future in the design process, and it is noteworthy that that interface is quite thin.
End Remark .

From here, there are two essentially different ways in which to proceed. They result in two different solutions.

Solution A
Again, the global correctness of $x \leq y$ in $Comp0$ is for free. We ensure its local correctness by "testing", i.e. by prefixing assertion $x \leq y$ with if $x \leq y \rightarrow$ skip fi . $Comp1$ is dealt with in a symmetric fashion, and thus we arrive at

$$Pre: \qquad x=0 \;\wedge\; y=0$$

```
Comp0:  * [  { $x \leq y$ } $S$                Comp1:  * [  { $y \leq x$ } $T$
           ; $x := x+1$                                 ; $y := y+1$
           ; if $x \leq y \rightarrow$ skip fi            ; if $y \leq x \rightarrow$ skip fi
             { $x \leq y$ }                               { $y \leq x$ }
           ]                                            ] .
```

Approximation 1A

There is no danger of total deadlock because the disjunction of the guards is true. So, in a way we are done. We wish to observe, though, that by the invariance of

MB: $x \leq y+1 \wedge y \leq x+1$,

the difference $y-x$ is just three-valued, so that we can eliminate the ever growing integers x and y at the expense of, say, just two booleans. We will, however, not carry out such a coordinate transformation here.

End Solution A .

Solution B

Starting from Approximation 0 again, we now destroy the symmetry between the components. We ensure the local correctness of assertion $x \leq y$ in $Comp0$ by requiring that the precondition of $x := x+1$ implies $x+1 \leq y$. For $Comp1$ we do *not* carry out such a move. We thus obtain

Pre: $x=0 \wedge y=0$

```
Comp0:  * [  { $x \leq y$ } $S$                Comp1:  * [  { $y \leq x$ } $T$
           ; {  $x+1 \leq y$ , ? }                        ; $y := y+1$
              $x := x+1$                                     {  $y \leq x$ , ? }
              { $x \leq y$ }                               ]
          ]
```

Approximation 1B

Along with the transition from Approximation 0 to Approximation 1B, two things happened.

(*i*) The latter has a *stronger* annotation: preassertion $x+1 \leq y$ to $x := x+1$ has crept in. Now, let us recall the operational interpretation of an assertion. If a component is “at” an assertion, the state of the system as a whole satisfies that assertion. The stronger the assertion, the smaller the space in which the rest of the system can manoeuvre, i.e. the lesser the degree of parallelism that can be exhibited.

(*ii*) No matter how we proceed from Approximation 1B, relation

$$Q: \qquad x \leq y \;\wedge\; y \leq x{+}1$$

will be a system invariant. This is easily checked. As a result, the difference $y{-}x$ is just two-valued, instead of three-valued as in Solution A. We therefore can represent in by a single boolean.

And here we see in a nutshell how “parallelism” can be traded for storage space, a phenomenon that we alluded to before.

Remark We can push the phenomenon to a limit by restoring the symmetry between the components and perform the same transformation for $Comp1$. Then the ensuing invariant will be $x \leq y \wedge y \leq x$, i.e. $x = y$. As a result, total deadlock will become unavoidable. And indeed, total deadlock can be implemented with zero variables, for instance by means of constructs like **if** *false* → **skip fi** .
End Remark .

* * *

Now we proceed from Approximation 1B in a straightforward manner so as to arrive at

$$Pre: \qquad x = 0 \;\wedge\; y = 0$$

$$Inv: \qquad x \leq y \wedge y \leq x{+}1$$

```
$Comp0$:  $*$ [ $\{ x \leq y \}$ $S$
              ; if $x+1 \leq y \rightarrow$ skip fi
                $\{ x+1 \leq y \}$
              ; $x := x+1$
                $\{ x \leq y \}$
              ]
```

```
$Comp1$:  $*$ [ $\{ y \leq x \}$ $T$
              ; $y := y+1$
              ; if $y \leq x \rightarrow$ skip fi
                $\{ y \leq x \}$
              ] .
```

And again, there is no danger of total deadlock, so that individual progress is guaranteed.

As a last step in our development we this time *do* perform the coordinate transformation towards a single boolean. We introduce boolean variable c , coupled to x and y by

$$c \ \equiv \ (x=y) \quad .$$

Then, by the invariant $x \leq y \ \wedge \ y \leq x+1$,

$$x+1 \leq y \ \equiv \ \neg c \qquad \text{and} \qquad y \leq x \equiv c$$

$$\{ \ x+1 \leq y \ \} \ x \ := \ x+1 \qquad \leftrightarrow \qquad \{ \ \neg c \ \} \ c := \mathit{true}$$

$$\{ \ y \leq x \ \} \ y \ := \ y+1 \qquad \leftrightarrow \qquad \{ \ c \ \} \ c := \mathit{false} \quad .$$

And thus we arrive at our final solution, for which the raw code reads

$$\mathit{Pre}: \quad c$$

```
$Comp0$:  $*$ [ $S$
              ; if $\neg c \rightarrow$ skip fi
              ; $c := true$
              ]
```

```
$Comp1$:  $*$ [ $T$
              ; $c := false$
              ; if $c \rightarrow$ skip fi
              ]
```

End Solution B .

A case of intuitive reasoning

Mathematical intuition is a very dangerous and unreliable compass, even more so in the case of multiprogramming. Recently we showed a very small multiprogram to our class, namely the following

```
A:    $y := false$                    B:    $x := false$
    ; if $y \to skip$ fi                  ; if $x \to skip$ fi
```

These are just two straight-line programs, each consisting of just two simple statements. Hardly anything simpler can be conceived, can't it?

Now, for the sake of letting both components terminate, we granted the class the possibility to add statements "$x := true$" to component A , as many as they wanted and wherever they wanted. And similarly, statements "$y := true$" were allowed to be added to B .

The class did not hesitate very long. Because component B is "waiting" for x to become true, termination of B becomes most likely if A performs "$x := true$" as often as possible. And symmetrically so for "$y := true$" . So, here is the solution:

```
A:    $x := true$                     B:    $y := true$
    ; $y := false$                        ; $x := false$
    ; $x := true$                         ; $y := true$
    ; if $y \to skip$ fi                  ; if $x \to skip$ fi
    ; $x := true$                         ; $y := true$
```

But, alas, each effort to give a genuine termination proof failed. And indeed, there is no guarantee that both components terminate. (Let A proceed to its if-statement. Then $x \wedge \neg y$ holds. Next, let B perform its first "$y := true$" . Then $x \wedge y$ holds. Now let A terminate. Then B gets stuck.)

The nice thing is that, if we remove the first line from each component, i.e. if we consider

```
A:     y := false                    B:     x := false
     ; x := true                          ; y := true
     ; if y → skip fi                     ; if x → skip fi
     ; x := true                          ; y := true
```

then everything is okay (proof omitted here [?]).

* *
*

To us, the above is a very nice example to demonstrate the intricacies of multiprogramming to a novice audience, and to warn them to never lean on “intuition”, but on rigorous formal proofs instead.

****

Bibliography