The Full Assignment Command

- The general definition of $\text{wp}(\vec{x} := \vec{e}, \phi)$ for assignments including arrays uses all the pieces described earlier.
- Let $y_1, y_2, y_3, \ldots, y_m$ be the names of the variables being assigned to in the left side $\vec{x}$.
  - That is, each y_i occurs as `...,`$y_i\lambda$`,...` on the top level of $\vec{x}$ — outside any of the indexings `[...]`.
  - Here the λ denotes the sequence of indexings `[...]` attached to this occurrence of variable name y_i.
    If y_i occurs here without any indexings, then $\lambda = \varepsilon$.
  - Each of these y_i can be treated *separately* from the other y_j...
  - ...but all the different occurrences of the same y_i must be treated *together* and in their *original order*.
- Let the occurrences of y_i in $\vec{x}$ be

$$y_i\lambda_{(i,1)}, y_i\lambda_{(i,2)}, y_i\lambda_{(i,3)}, \ldots, y_i\lambda_{(i,n_i)}$$

  in the same left-to-right order.
- Let the expressions in the right side $\vec{e}$ corresponding to them be

$$e_{(i,1)}, e_{(i,2)}, e_{(i,3)}, \ldots, e_{(i,n_i)}.$$

- The construct of Eq. (26) for y_i combines these occurrences:

$$\sigma_i = \langle y_i; \lambda_{(i,1)} \leftarrow e_{(i,1)}; \lambda_{(i,2)} \leftarrow e_{(i,2)}; \lambda_{(i,3)} \leftarrow e_{(i,3)}; \ldots$$
$$\ldots; \lambda_{(i,n_i)} \leftarrow e_{(i,n_i)}\rangle.$$

- Then the general definition is the familiar simultaneous textual substitution of each y_i with its construct σ_i as in Eq. (9):

$$\phi[y_1 \leftarrow \sigma_1, y_2 \leftarrow \sigma_2, y_3 \leftarrow \sigma_3, \ldots, y_m \leftarrow \sigma_m].$$

- The result is then simplified using the rules given earlier to eliminate these constructs σ_i to get the final formula for $\text{wp}(\vec{x} := \vec{e}, \phi)$.
- This full generality will be needed later for subroutines. We will model passing parameter values into and from them as assignments into and from the formal parameter variables. Then we need to be prepared for all the possible combinations that might be passed.

3 Logic-Guided Development of Algorithms

The material in this section is from Gries (1981)[Chapters 13–17] and Dromey (1989)[Part 2].

- Algorithm/program development is conceptually:

{ Given *pre*condition(s):
    What can be assumed about the input etc.? }
`Invent here` ***`code`*** `which achieves this!`
{ Given *post*condition(s):
    What is wanted as the result? }

- The weakest precondition approach (Section 2.3) is *goal-oriented:*

Principle 1 (Postcondition First)**.** *The postcondition guides code invention. The precondition tells when we can stop inventing more code – when it guarantrees that our current code works.*

- The questions to be studied are thus: How can we...

  "see" what code we should try from the given postcondition?

  massage it to "see" this more clearly?

  develop the details for the kind of code we decide to try?

- One step of development can in turn be conceptually stated as:

  { Given *pre*condition(s):
      What can be assumed about the input etc.? }
  `Invent` ***`more code`*** `here if necessary!`
  { wp(`already invented` ***`code`***, *post*condition) }
  `The already invented` ***`code`***`.`
  { Given *post*condition(s):
      What is wanted as the result? }

  – More code is still necessary, if the *pre*condition does not yet guarantee that the `already invented` ***`code`*** works.
  – That is, if the *pre*condition does not yet imply the weakest precondition under which executing the `already invented` ***`code`*** guarantees the *post*condition.
  – If the *pre*condition already implies it, then strengthening the *pre*condition (Theorem 11) is enough, and no more code is necessary.

- Here we focus on "programming in the small": we mostly invent loops and their initializations.

3.1 Developing an "if" Command

- Suppose that we have "seen" to try an **if** command:

  $$\{ \text{ precondition } \phi \ \}\textbf{if } \mathcal{B} \textbf{ fi}\{ \text{ postcondition } \psi \ \}$$

  How should we invent the branches $1, 2, 3, \ldots$ into the initially empty body $\mathcal{B}$?

- The conceptual principle can be read from the **if** theorem (Theorem 13):

  1. Find a *command* which ensures ψ in at least some of the situations described by ϕ.
  2. Determine these exact situations by computing $\varphi = \text{wp}(\textit{command}, \psi)$.

3. Form a *guard* such that $\phi \wedge \mathit{guard} \implies \varphi$, as in Condition 2.

   That is, "What more than ϕ do we need to assume, before we can prove φ?"

4. Add this branch $\mathit{guard} \to \mathit{command}$ into $\mathcal{B}$.

5. Continue this until ϕ implies the disjunction of all the added *guards*, as in Condition 1.

   To get ahead, try to handle new situations not already handled by the already added branches.

- Steps 2–4 can be summarized as "invent and add a branch $\mathit{guard} \to \mathit{command}$ such that $\{\ \phi \wedge \mathit{guard}\ \}\ \mathit{command}\{\ \psi\ \}$".

- E.g. suppose we have two variables $j, k \in \mathbb{N}$ whose values depend on each other:

$$j = k \bmod 10.$$

  How can we increment k (by 1) while keeping j "in sync", if using 'mod' is impossible for some reason?

1. The initial empty **if** is:

```
{ $j = k \bmod 10 \wedge k = K$ }
if
fi
{ $j = k \bmod 10 \wedge k = (K+1)$ }
```

2. One *command* is clearly to increment both variables:

```
{ $j = k \bmod 10 \wedge k = K$ }
if  $\mathit{guard} \to$
    $k , j := k+1 , j+1$
fi
{ $j = k \bmod 10 \wedge k = (K+1)$ }
```

   Calculating shows the situations handled by this *command*:

$$\begin{aligned} &\mathrm{wp}((k,j := k+1, j+1), (j = k \bmod 10 \wedge k = (K+1))) \\ =&((j+1) = (k+1) \bmod 10 \wedge (k+1) = (K+1)) \\ \Leftrightarrow&(j < 9 \wedge \underbrace{j = k \bmod 10 \wedge k = K}_{\text{our precondition}}). \end{aligned}$$

   Hence we need $j < 9$ more than the precondition, so that is our *guard* for this *command*.

3. The missing situation is $j = 9$ which can be handled by resetting j to 0. Because this is such an exact branch, we omit its formalities, and add it directly:

```
{ $j = k \bmod 10 \wedge k = K$ }
if  $j < 9 \to$
    $k , j := k+1 , j+1$
 [] $j = 9 \to$
    $k , j := k+1 , 0$
fi
{ $j = k \bmod 10 \wedge k = (K+1)$ }
```

But wouldn't $j \geq 9$ be an even better *guard*, since this code would then reset j properly even if it has not been properly initialized? *No:*

Principle 2 (Strong **if** Guards)**.** *Resist the temptation to weaken an **if** guard past what was actually needed during code invention. Debugging-wise, it is better to **abort** as soon as a precondition violation is detected!*

3.2 Developing a "do" Command

- Suppose next that we have "seen" to use a **do** command when its *invariant and bound are already known:*

```
{ invariant: $\phi$
  bound: $bound$ }
do $\mathcal{B}$
od
{ $\phi \wedge \neg\, guards$ chosen to imply $\psi$ }
```

  How should we invent the branches $1, 2, 3, \ldots$ into the initially empty body $\mathcal{B}$?

- Inventing an invariant ϕ and *bound* from a given postcondition ψ is a separate (and very rich!) subject to be discussed later.
- Again, the conceptual principle can be read from the **do** theorem (Theorem 14) and the corresponding 5-point checklist.
- We can approach the task from two directions.

3.2.1 Inventing the Guard First

- One approach is to start with the *guard* and derive its *command* later.
- The *guard* is sought by checkpoint 3: we must have

$$\phi \wedge \neg\, guard \Longrightarrow \psi$$

  at the end of the loop. Hence

  1. first we ask "What *more* information in addition to the invariant ϕ do we need to prove the desired postcondition ψ?"
  2. then our *guard* is its complement $\neg\, more$ since the aim of the loop is to achieve $\phi \wedge more$ when it finishes to ensure ψ
  3. and we need to invent a corresponding *command* which
     – decrements the *bound* by checkpoint 5, but
     – maintains the *invariant* true by checkpoint 2.

- This approach requires that *more* is
  – "easy" to see by massaging ϕ and ψ
  – simple enough to for $\neg\, more$ to be permitted as a *guard*.
- This yields the following *single*-branch loop:

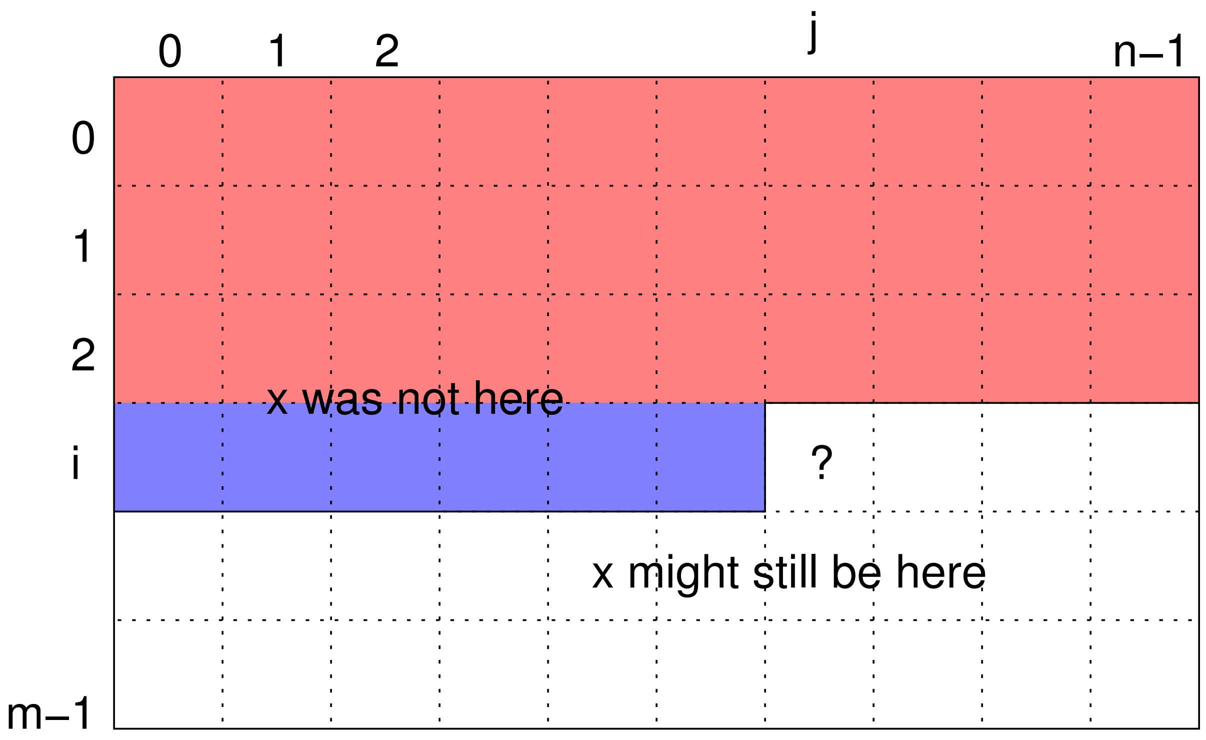

Figure 12: The Invariant for Matrix Search.

```
{ invariant: $\phi$
  bound: $\mathit{bound}$ }
do  $\neg\,\mathit{more} \rightarrow$
      $\mathit{command}$
od
{ $\phi \land \mathit{more}$ }
```

- E.g. consider the following problem:

  Inputs: A two-dimensional array b. To keep the notation simple, assume that it has rows $0, 1, 2, \ldots, m-1$ and each row has columns $0, 1, 2, \ldots, n-1$.
  An element x to find in this b.

  Output: If x occurs in b, then some indices i, j such that $b[i][j] = x$; otherwise $i = m$.

  Invariant: Informally, x was not “above or to the left” of the current element $b[i][j]$ (Figure 12).

  Bound: The number $(m-i) \cdot n - j$ of elements still to be tested.

- Note that these pictures (Figure 12) can be very illustrative, but they can also omit details: e.g. can the last row be full or empty?

- Let us formalize the

  invariant ϕ e.g. as

$$\begin{aligned} &0 \le i \land 0 \le j \land \\ &\qquad (\forall 0 \le p < i, 0 \le q < n \colon b[p][q] \neq x) \land \\ &\qquad\qquad (i < m \Longrightarrow \forall 0 \le r < j \colon b[i][r] \neq x) \end{aligned} \tag{30}$$

postcondition ψ e.g. as

$$\overbrace{0 \le i \land i < m \land 0 \le j \land j < n \land b[i][j] = x}^{\text{“\textit{Yes}, we found it here.”}} \lor \\ \underbrace{i = m \land \forall 0 \le p < i, 0 \le q < n \colon b[p][q] \ne x}_{\text{“\textit{No}, it is nowhere.”}} . \quad (31)$$

- We must add *more* to our invariant ϕ to ensure both parts of our postcondition ψ.

  Yes part needs $i < m$ and $b[i][j] = x$.

  No part needs $i = m$.

- Hence we get our *guard* as

$$\neg((i < m \,\textbf{cand}\, b[i][j] = x) \lor i = m). \quad (32)$$

- Our code is now

```
$i$ , $j$ := 0 , 0;
{ invariant: $\phi$
  bound: $(m - i) \cdot n - j$ }
**do** $i \ne m$ **cand** $b[i][j] \ne x \rightarrow$
    { $\phi$ and our *guard* }
    *our still missing command*
    { $\phi$ still holds but the *bound* has decreased }
**od**
{ $\psi$ }
```

  where we have

  – simplified our *guard* using ϕ

  – initialized the already tested area to be empty (Figure 12), which clearly establishes ϕ.

- Again: Wouldn't $i < m$ be a better guard?

  Again: *No.*

Principle 3 (Weak **do** Guards)**.** *Resist the temptation to strengthen an* ***do*** *guard past what was actually needed during code invention. Debugging-wise, it is better to get stuck to where the error appeared than going past that spot.*

- Our next task is to invent a suitable *command*.

3.2.2 Making Progress towards Termination

- When we need to invent a suitable *command* which must

  decrease the given *bound*, but

  maintain the given **do** loop invariant ϕ true

  our main focus is (perhaps surprisingly) the former requirement.

- The reason is that we can check the latter requirement by calculating and verifying that
$$\phi \wedge guard \Longrightarrow \mathrm{wp}(proposed\ command, \phi).$$
- Moreover, we can even take our *proposed command* to be an empty **if fi** and start adding branches to its body until this latter requirement is met (Section 3.1).
- This permits us to have more than one way to meet the former requirement working together, if one way is not enough.
- In our matrix search example (Section 3.2.1), this idea is applied as follows:
  - Now $\phi \wedge guard$ entails $i < m$, $j < n$ and $b[i][j] \neq x$, so $b[i][j]$ can be moved from the still untested into the already tested area (Figure 12).

    This decrements the *bound* as desired.
  - One way to do this is to increment j (by 1).

    However, computing its $\mathrm{wp}(j := j + 1, \phi)$ reveals that it requires also that $j < n - 1$.
  - Since incrementing j works in some but not all situations, we use it conditionally:

```
if $j < n - 1 \rightarrow$
    $j := j + 1$
fi
```

  - Keep filling this **if** by considering the still unhandled situations $j \geq n - 1$, which is $j = n - 1$ by ϕ. Hence the

    question is "How can we move $b[i][j]$ from the untested into the tested area when it is the last element on row i?"

    answer is "Move i to the next row and j to its left end!"
  - This expands our **if** into

```
if $j < n - 1 \rightarrow$
    $j := j + 1$
 [] $j = n - 1 \rightarrow$
    $i, j := i + 1, 0$
fi
```

    which covers all the situations in $\phi \wedge guard$, so it is complete.
  - Our final code is thus:

```
$i, j := 0, 0;$
{ invariant: $\phi$
  bound: $(m - i) \cdot n - j$ }
do $i \neq m$ cand $b[i][j] \neq x \rightarrow$
    if $j < n - 1 \rightarrow$
        $j := j + 1$
     [] $j = n - 1 \rightarrow$
        $i, j := i + 1, 0$
    fi
od
{ $\psi$ }
```

3.2.3 Inventing the Commands First

- The opposite approach is to start with the *command*s and derive their *guard*s later.
- This can lead to *multi*-branch loops, instead of a single **do**-branch.
- The conceptual principle is similar to the one for **if** (Section 3.1) — after all, **do** is "repeated **if**" (Section 2.4.7):
  1. Find a *command* which decreases the *bound* in at least some of the situations described by ϕ while keeping it true.
  2. Determine these exact situations by computing $\varphi = \text{wp}(\textit{command}, \phi)$.
  3. Form a *guard* such that $\phi \wedge \textit{guard} \Longrightarrow \varphi$.
     Often we can just choose φ itself as our *guard*, or something weaker.
  4. Add this branch *guard* $\rightarrow$ *command* into the initially empty body of the **do ... od** loop.
  5. Continue this until ϕ and the negations of the added *guard*s together imply the desired postcondition ψ.
- If we take this approach in the matrix search example (Section 3.2.2), it goes as follows (details omitted):
  - First, we modify the invariant ϕ to permit also the value $j = n$: $0 \leq i \leq m \wedge 0 \leq j \leq n \wedge \ldots$ — but still *not* trying to index b with it!
  - The simplest way to decrease the similarly modified *bound* $(m-i)\cdot n-j+m-i$ (to take into account $j = n$) is to increment j (by 1).
  - A $\textit{guard}_j$ which ensures that this $\textit{command}_j$ preserves the invariant ϕ is $i \neq m \wedge j \neq n$ **cand** $b[i][j] \neq x$ which can be either found formally through calculating $\text{wp}(j := j+1, \phi)$ or "seen" informally.
  - This is more complicated than the corresponding **if** guard, since we no longer have a separate **do** guard in the background.
  - Another way to decrease the *bound* is to increment i (by 1) instead — to move to the next row.
  - This *command* in turn can preserve ϕ only if we are still on a row ($i < m$) and it contains no x. Hence we must also have $j = n$ so that the whole current row i has already been checked. This is the corresponding $\textit{guard}_i$.
  - But this is not yet enough to preserve ϕ: we must also move j from the end of the current to the beginning of the next row. Hence the corresponding $\textit{command}_i$ must be $i, j := i+1, 0$.

    Only their combination $\textit{guard}_i \rightarrow \textit{command}_i$ preserves ϕ.
  - Adding the already familiar initialization yields the following code:

```
i , j := 0 , 0;
{ invariant: the slightly modified $\phi$
  the modified bound: $(m-i)\cdot n-j+m-i$ }
do $i \neq m \wedge j \neq n$ cand $b[i][j] \neq x \rightarrow$
     j := j + 1
 [] $i \neq m \wedge j = n \rightarrow$
     i , j := i + 1 , 0 .
```

```
od
{ $\psi$ }
```

It is also the final code, since (it can be verified that) the invariant ϕ, $\neg\, guard_j$ and $\neg\, guard_i$ together do imply the desired postcondition ψ.

- Comparing this final code with the final code obtained by developing the single guard first instead (Section 3.2.1) reveals that this essentially “flattened” its **do-if** structure into one **do**.
  - Recall that here we permit and use also the value $j = n$, there we did not, which causes minor differences:

    Here j becoming n signals that the algorithm must move to the next row on its next loop round.

    There the algorithm moved to the next row already on this round instead of j becoming n.

    Such flattening often involves such signal values.
  - These codes are similar because their *command*s were developed similarly from the *bound* (Section 3.2.2).
- Loops developed like this can be read as *rule collections:*

```
do the current situation is like this →
     decrease the bound like this
  ▯ the current situation is like that →
     decrease the bound like that
  ▯ ⋮
od
```

- One structured way to implement such multi-branch **do** loops in a conventional programming language is with an explicit flag for the disjunction of all its *guard*s:

```
guards ← TRUE
repeat  if $guard_1$
           then $command_1$
        elseif $guard_2$
           then $command_2$
        elseif $guard_3$
           then $command_3$
                ⋮
           else  guards ← FALSE
  until ¬ guards
```

Here we use a **repeat** (and not the more common **while**) loop to make it explicitly clear that the *guard*s do get tested at least once.

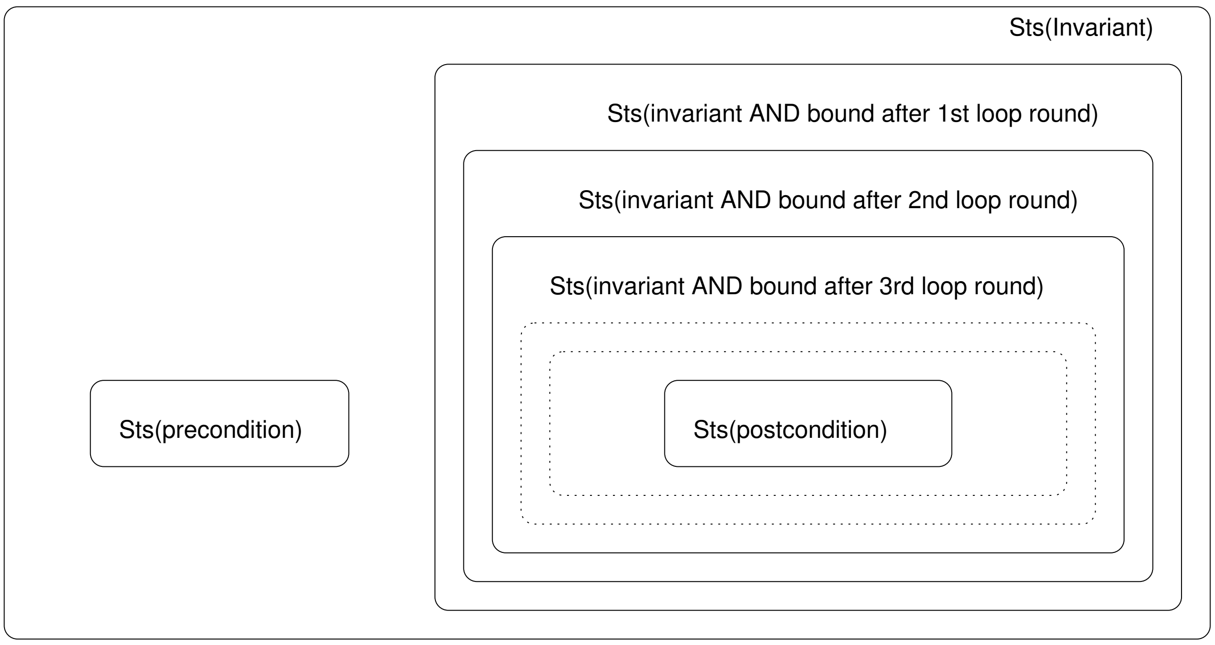
Sts(Invariant)
Sts(invariant AND bound after 1st loop round)
Sts(invariant AND bound after 2nd loop round)
Sts(invariant AND bound after 3rd loop round)
Sts(precondition)
Sts(postcondition)

Figure 13: The Deflating Balloon.

3.3 The Balloon Theory

- One metaphor for understanding the interplay between

  { *precondition*, *invariant* and *bound* }
  do ⋮
  od
  { *postcondition* }

  is as a balloon with states as the air molecules inside it (Figure 13):

  Postcondition is the deflated state — the molecules that still remain in the balloon after the loop has finished executing.

  Precondition consists of those molecules which must be in the balloon when the loop starts executing.

  Invariant is the inflated state — which must contain both the pre- and postcondition molecules.

  Bound is letting the air molecules gradually out of the balloon — the states which are no longer possible as the loop execution progresses.

Principle 4 (Correct Loop Design)**.** *Every loop must be designed so that its invariant and the negation of its guards together entail its postcondition! Only this guarantees that the loop has done what it promised to do when it terminates!*

- With this metaphor, designing the loop *invariant* involves *weakening* its desired *postcondition* — that is, inflating the balloon — until it encompasses also the *precondition*.
- This inflating part of the balloon moves from the *postcondition* into the loop *guards*:

  The *guards* must entail this inflating part, so that we can read the loop termination condition

$$\mathit{invariant} \land \neg\, \mathit{guards} \quad \text{as} \quad \text{“the inflated balloon } \textit{but not} \text{ the inflating part”}$$

which in turn entails the *postcondition* — the deflated balloon.

- Hence the actual *guards* can themselves be a weakening of this inflated part.
- Often the *precondition* is not known exactly. Then the *postcondition* must be weakened enough so that the resulting *invariant* can be established via easy initializations.
- Let us therefore discuss different ways of weakening the *postcondition* so that it becomes more tractable.

3.3.1 Deleting a Conjunct

- Usually the *postcondition* is a conjunction of all the different requirements that the answer must fulfill (or a disjunction of such conjunctions).
- A natural way to weaken a conjunction is to drop one (or more) of its conjuncts. The *negation* of this dropped conjunct is a natural candidate guard — since we inflated with its complement.
- E.g. consider computing the *natural square root* for a given $n \in \mathbb{N}$: the largest $a \in \mathbb{N}$ such that $a \leq \sqrt{n}$.

  This postcondition can be expressed as the conjunction

  $$a^2 \leq n \land \underbrace{n < (a+1)^2}_{\text{Let's drop this!}} \tag{33}$$

  and we get the code skeleton

  ```
  { invariant: $a^2 \leq n$ }
  do $\neg(n < (a+1)^2) \rightarrow$
      what here?
  od
  ```

- Here we have an obvious choice for loop *initialization*:

  ```
  $a$ := 0;
  { invariant: $a^2 \leq n$
    bound: $\lceil\sqrt{n}\rceil - a$ }
  do $(a+1)^2 \leq n \rightarrow$
      $a := a + 1$
  od
  ```

- We also added an obvious choice for the *bound*: the distance how much below the correct answer the current a is.
- Note that this “turn a conjunct into the *guard*” idea fits nicely with the “design the *guard* first, then the *command*” idea for developing the loop further (Section 3.2.1).
- A natural idea for the *command* is then to increment a (by 1):

  $$\mathrm{wp}(a := a + 1, a^2 \leq n) = (a+1)^2 \leq n,$$

  so it is even enough.

- Note in particular that the universal quantifier '$\forall$' is in fact a conceptually infinite conjunction '$\wedge$': $\forall x.\phi$ means intuitively

  $$\phi[x \leftarrow v_1] \wedge \phi[x \leftarrow v_2] \wedge \phi[x \leftarrow v_3] \wedge \ldots$$

  over all the possible values $v_1, v_2, v_3, \ldots$

- Hence *shrinking the range* of '$\forall$' is a valid weakening: e.g.

  $$(\forall\, \mathrm{lower}(b) \leq i \leq \mathrm{upper}(b) : b[i] < 100) \tag{34}$$

  splits into

  $$\begin{gathered}(\forall\, \mathrm{lower}(b) \leq i < j : b[i] < 100) \\ \wedge (\forall\, j \leq i \leq \mathrm{upper}(b) : b[i] < 100)\end{gathered} \tag{35}$$

  for any arbitrary $\mathrm{lower}(b) \leq j \leq \mathrm{upper}(b) + 1$.

3.3.2 Initializing the Loop

Principle 5 (Simplifying Initializers)**.** *Try to find an initializer which sets the variables of the loop into a boundary case where the postcondition has some simpler form.*

- E.g. in our matrix search example (Figure 12) we initialized both i and j to 0 intuitively because it simplified the untested area to nothing.

- More precisely, this initialization divided the conjuncts of the postcondition (31) into two kinds:

  - Those which became certainly TRUE by this initialization: $0 \leq i$, $0 \leq j$ and $(\forall \ldots)$.

    Note that all of them moved into the invariant (30) as conjuncts!

  - The others: $i < m$, $j < n$, $b[i][j] = x$ and $i = m$.

    They stayed behind as the others moved, and this residue formed the guard (32) instead!

    (Except that $j < n$ went all the way into the **if** guard in a subsequent development step.)

- This makes sense:

  - The invariant must be TRUE when the loop starts, so it cannot contain anything which might be FALSE after the initialization.

  - The (negation of the) guard must in turn entail all the extra information which the invariant cannot provide to the postcondition.

- We can calculate (in a straightforward way)

  $$\mathrm{wp}(\underbrace{x_1, x_2, x_3, \ldots, x_k := e_1, e_2, e_3, \ldots, e_k}_{\text{our suggested } \textit{initializer} \text{ code}}, \textit{postcondition})$$

  and consider each *conjunct* separately:

  - If the calculation simplified it to TRUE, then move it(s original unsimplified form) as another conjunct into the *base invariant* being constructed.

    This *base invariant* begins as

    $$x_1 = e_1 \land x_2 = e_2 \land x_3 = e_3 \land \ldots \land x_k = e_k \tag{36}$$

    recording the effects of the *initialization*.
  - Otherwise keep it in the *guard* being constructed instead.

- Then the completed *base invariant* is either suitable "as is" or it can be massaged further.

  E.g. our matrix search example

  added an implication also entailed by the initialization. (Its consequent '∀' is initially empty, making it TRUE, since we set $j = 0$.)

  removed the $i = 0$ and $j = 0$ from (36) since we inted to change them during the loop.

- The completed *guard* should in turn be ready to use (negated) in the code, following the "guard first" approach (Section 3.2.1).

  But if it still contains quantifiers, then we have made only partial progress:

  The *postcondition* should be massaged further into a form where quantifiers go into the *invariant* instead.

  The *initializer* might need to be something more complicated than an assignment. However, calculating its wp gets more tedious.

  After modifying one or both, this same approach can be tried again.

- Or we can just take the intuitive idea and apply it informally:

  - Design an *initializer* which makes as many *conjunct*s of the *postcondition* TRUE — especially all of its quantifications ($\forall \ldots$)!
  - Then these TRUE *conjunct*s move away from the *postcondition* to form the *invariant* — perhaps after some additional massaging.
  - The other unTRUE *conjunct*s stay in the *postcondition* which becomes the *guard* — again perhaps after some additional massaging.

3.3.3 Introducing a New Variable

- How can we *introduce new variables* such as the index j in Eq. (35) into our postconditions and program code?

- In principle we can just add them as new conjuncts:

  $$\begin{aligned}&(\text{the old } \textit{postcondition})\land\\ &\qquad\qquad(\text{the new part defining the value(s) for the new variable } x).\end{aligned}$$

  But this is strictly speaking *strengthening* the old postcondition instead of weakening it: before adding the new part, x was free to have any value whatsoever — but not any more!

- However, we can
  1. first strengthen our original *postcondition* with the variables $x, y, z, \ldots$ we need
  2. then derive our algorithm to satisfy even this new stronger *postcondition*
  3. and finally apply postcondition weakening (Theorem 12) to drop their definitions, and get back our original *postcondition*

  so this is not really an obstacle.

Principle 6 (Variable Introduction)**.**

- *Introduce a new variable only when you can explain precisely what you intend to use it for.*
- *In practical coding, this explanation usually suggests also a good mnemonic name for your variable. (Here we use single-letter names, but just for brevity.)*
- *State also the range of values you intend your new variable to take, since you may need this information later.*
- *Do not recycle the same variable for different uses, because that will get confusing after a while.*

- One certainly good use is to permit subsequent weakening of our current *postcondition* (which was first strengthened to introduce this variable) to find out an appropriate *invariant* for a loop manipulating it.
- There are several common patterns for variable introduction.

Replacing an Expression with a Variable

- One common pattern is to identify some (sub)*expression* occurring in the *postcondition* and name it with our new *variable*.
- Formally, we replace one or more (but not necessary all) occurrences of *expression* in the *postcondition* with our new *variable*, and add $\ldots \land (variable = expression)$ at the end.
- E.g. the *postcondition* (34) tests the whole array, but it does not yet suggest any invariant or its initialization.
- We can add the variable j as

$$(\forall\, \mathrm{lower}(b) \leq i < j \colon b[i] < 100) \land (j = \mathrm{upper}(b) + 1) \land \underbrace{(\mathrm{lower}(b) \leq j) \land (j \leq \mathrm{upper}(b) + 1)}_{\text{its intended value range}} \quad (37)$$

  and start weakening this form instead.

- As a concrete example, let us solve the natural square root problem (Section 3.3.1) differently.

– This time we start by introducing a new variable to the postcondition (33) to get

$$a^2 \leq n \wedge n < b^2 \wedge \underbrace{b = a+1}_{\text{gets deleted}} \wedge \underbrace{a < b \wedge b \leq n+1}_{\text{its range}}.$$

which leads into the following code:

```
a , b := 0 , n+1 ;
{ invariant: the rest of the modified postcondition above
  bound: the distance $b - a - 1$ from $a$ into $b$ }
do $b \neq a+1 \rightarrow$
    what here?
od
{ the original postcondition (33) }
```

– Now that we have both a and b, we can proceed by the same "midpoint trick" as in binary search (Figure 4) and get a faster algorithm:

```
a , b := 0 , n+1 ;
{ invariant: the rest of the modified postcondition above
  bound: the distance $b - a - 1$ from $a$ into $b$ }
do $b \neq a+1 \rightarrow$
    $m := (a+b) \operatorname{div} 2$ ;
    $d := m \cdot m$ ;
    if $d \leq n \rightarrow$
        $a := m$
    [] $d > n \rightarrow$
        $b := m$
    fi
od
{ the original postcondition (33) }
```

These new variables m and d can be considered *local* to the loop body: since they are not in its invariant, proving the rest of the program correct cannot involve them. We can omit introducing such local variables explicitly.

Enlarging the Range of a Variable

- One common way to weaken a *postcondition* is to *allow more values* for some variable x in it than before.
- In particular, we often enlarge the range of the introduced variable immediately after introducing it.
- E.g. we can weaken the range of j in postcondition (37) immediately:

$$(\forall \operatorname{lower}(b) \leq i < j \colon b[i] < 100) \cancel{\wedge (j \neq \operatorname{upper}(b) + 1)}$$
$$\wedge (\operatorname{lower}(b) \leq j) \wedge (j \leq \operatorname{upper}(b) + 1).$$

  We can take this as our *invariant*.

- Our *bound* will be the number of still untested elements:

$$\operatorname{upper}(b) - j + 1.$$

- It decreases by incrementing j, and that requires (Section 3.2.3)

$$\begin{aligned}&\mathrm{wp}(j := j+1, \mathit{invariant}) = \\ &\qquad\qquad \mathit{invariant} \wedge (b[j] < 100) \wedge (\mathrm{lower}(b) \leq j+1) \wedge (j \leq \mathrm{upper}(b)).\end{aligned}$$

- Then our *guard* is those parts which are not yet implied by our *invariant*:

$$(j \leq \mathrm{upper}(b))\,\textbf{cand}\,(b[j] < 100).$$

- At the end of our loop, we thus have *invariant* $\wedge\neg$ *guard* which is

$$\begin{aligned}&(\forall\,\mathrm{lower}(b) \leq i < j : b[i] < 100) \\ &\qquad\qquad \wedge (\mathrm{lower}(b) \leq j) \wedge ((j = \mathrm{upper}(b)+1)\,\textbf{cor}\,(b[j] \geq 100)).\end{aligned}$$

  That is, upon termination j is either (1 position) past the end of the array or the first index where the test failed.

- Or consider the following example ("the Welfare Crook"):

  Input: Three strictly ascending arrays f, g, h which are guaranteed to have common elements.

  Output: Indices i, j, k for such a common element: $f[i] = g[j] = h[k]$.

  - Let $i_{\min}, j_{\min}, k_{\min}$ be the indices for the first common element — the one which is smallest in the ordering between the elements.
  - We strengthen the requested output into the postcondition

    $$i = i_{\min} \wedge j = j_{\min} \wedge k = k_{\min}$$

    so that we promise to find even the first common element.
  - Note that it is OK to mention the desired result $i_{\min}, j_{\min}, k_{\min}$ *in the logic* side — just don't mention it on the code side, but prove that the code variables i, j, k reach the same value.
  - Hence these $i_{\min}, j_{\min}, k_{\min}$ are another kind of ghost variable (Section 2.1) since they appear free in formulae but not in code.
  - Next we enlarge the ranges for the code variables i, j, k on the code side to get our loop invariant:

    $$\begin{aligned}&(\mathrm{lower}(f) \leq i \wedge i \leq i_{\min}) \wedge \\ &\qquad (\mathrm{lower}(g) \leq j \wedge j \leq j_{\min}) \wedge \\ &\qquad\qquad (\mathrm{lower}(h) \leq k \wedge k \leq k_{\min}). \qquad (38)\end{aligned}$$

  - Similarly, we take as our bound the remaining distance:

    $$(i_{\min} - i) + (j_{\min} - j) + (k_{\min} - k).$$

  - Now we can reasonably expect to need a loop incrementing i, j, k so let us write those commands first and seek their guards later (Section 3.2.3):

```
$i\,,\ j\,,\ k\ :=\ \text{lower}(f)\,,\ \text{lower}(g)\,,\ \text{lower}(h)\,;$
do  what here? $\rightarrow$
        $i\ :=\ i+1$
   []  what here? $\rightarrow$
        $j\ :=\ j+1$
   []  what here? $\rightarrow$
        $k\ :=\ k+1$
od
```

– The first guard can be found by calculating

$$\begin{aligned}\text{wp}(i := i+1, \text{invariant (38)}) = {} & \\ & (\text{lower}(f) \leq i+1 \wedge i+1 \leq i_{\text{min}}) \wedge \\ & (\text{lower}(g) \leq j \wedge j \leq j_{\text{min}}) \wedge \\ & (\text{lower}(h) \leq k \wedge k \leq k_{\text{min}})\end{aligned}$$

where only $i+1 \leq i_{\text{min}}$ is not already entailed by invariant (38), but needs to be added by the guard.

* Since i_{min} must not appear in the code, we cannot use $i+1 \leq i_{\text{min}}$ directly as our guard.
* But $i+1 \leq i_{\text{min}}$ means that $f[i]$ is still smaller than the first common element.
* That in turn holds at least when it is smaller than either of the other two:

$$(f[i] < g[j]) \vee (f[i] < h[k]).$$

This entails $i+1 \leq i_{\text{min}}$ and can be used as a guard instead.

– The other two guards can be invented in the same way. So:

```
$i\,,\ j\,,\ k\ :=\ \text{lower}(f)\,,\ \text{lower}(g)\,,\ \text{lower}(h)\,;$
do  $(f[i] < g[j]) \vee (f[i] < h[k]) \rightarrow$
        $i\ :=\ i+1$
   []  $(g[j] < h[k]) \vee (g[j] < f[i]) \rightarrow$
        $j\ :=\ j+1$
   []  $(h[k] < f[i]) \vee (h[k] < g[j]) \rightarrow$
        $k\ :=\ k+1$
od
```

This code terminates and satisfies invariant (38).

– But it is not immediately obvious that it really produces the requested output, so let us check that: Suppose that none of the guards is true (since the loop has terminated). Then in particular none of their first disjuncts are true:

$$f[i] \geq g[j] \qquad g[j] \geq h[k] \qquad h[k] \geq f[i].$$

But then we have

$$f[i] = g[j] \qquad g[j] = h[k] \qquad h[k] = f[i]$$

so the output *is* a common (and the first such) element.

  - It was not also obvious before this check that the second disjuncts of its guards were not needed for correctness after all.
  - Hence they can even be *removed* from the final code!

Principle 7 (Branch First, Prune Later)**.** *The more guards there are, and the weaker these guards are, the easier it may be to develop a correct loop.*

Afterwards the parts which were not needed during the correctness proof can be removed for efficiency.

Image Extension

- We can also do an "inverse" replacement: change a variable or a constant into a more complicated expression defining it.

- Although it complicates the original *postcondition*, its more complicated form can suggest an *invariant* more clearly.

- E.g. the postcondition of our fast exponentiation algorithm (Figure 10) was originally quite unhelpful:

$$\begin{aligned}(a = X^P) &\Leftrightarrow (a \cdot 1 = X^P) && \text{(by mathematics)}\\ &\Leftrightarrow (a \cdot \mathit{anything}^0 = X^P) && \text{(inverting the 1)}\\ &\Leftarrow (a \cdot \mathit{anything}^p = X^P) \wedge (p = 0) && \text{(introducing } p\text{).}\end{aligned}$$

  This last form suggests an idea to try: decrement p from its initial (ghost) value P down to 0.

- Then this initial value of p requires X as the initial value of *anything*. Hence a natural guess for *anything* would be x.

- Taking our more complicated postcondition and deleting its second conjunct (Section 3.3.1) gets us started:

```
a := 1;
{ invariant: $a \cdot x^p = X^P$
  bound: $p$ }
do $p \neq 0$ →
    body here
od
{ $(a \cdot x^p = X^P) \wedge (p = 0)$ }
```

- Then the road to the fast algorithm might start with the body

```
if even($p$) →
    what here?
 [] ¬even($p$) →
    what here?
fi
```

  and continue from there.

- Another classic example is computing the greatest common denominator $\gcd(m, n)$ = the largest number which divides both m and n.

- One version of *Euclid's Algorithm* (from his book *Elementa* around 300 BCE, sometimes called the oldest nontrivial algorithm in history) is

```
    { $m \in \mathbb{N} \setminus \{0\} \wedge n \in \mathbb{N} \setminus \{0\} \wedge M = m \wedge N = n$ }
1    while $m \neq n$
2         do subtract the smaller from the larger
    { $n = \gcd(M, N)$ }
```

  where we have taken the liberty of using a Hoare triple with an informally described algorithm.

- Its verification becomes straightforward, if we first extend its original postcondition in a nonobvious way and using *two* new conjuncts to

$$\underbrace{(\overbrace{\gcd(m,n)}^{\text{change } n \text{ so}} = \gcd(M,N))}_{\text{the invariant maintained by line 2}} \wedge \overbrace{(n = \gcd(m,n)) \wedge \underbrace{(m = n)}_{\text{as line 1}}}^{\text{these } \textit{together} \text{ justify changing } n \text{ so}}.$$

3.3.4 Including the Precondition

- Sometimes our *invariant* needs to mention not only the *postcondition* as usual but also the *precondition* as well.

- This can happen in e.g. an array algorithm, if we must also say that something has *not* been done (yet).

- E.g. consider insertion sorting (Cormen et al., 2001, Chapter 2.1):

```
{ Precondition: the whole array $a$ is *un*modified. }
$i$ := lower($a$) + 1 ;
do  $i \leq$ upper($a$) $\rightarrow$
    $j$ := $i$ ;
    do  ($j >$ lower($a$)) cand ($a[j-1] > a[j]$) $\rightarrow$
        $a[j-1]$ , $a[j]$ , $j$ := $a[j]$ , $a[j-1]$ , $j - 1$
    od
    $i$ := 1 + 1
od
{ Postcondition: the whole array $a$ is sorted. }
```

- The invariant of the outer loop has the form

$$\begin{aligned}&(\text{the prefix } a[\text{lower}(a)\ldots i-1] \text{ is a sorted version of}\\&\qquad\text{the original unmodified prefix } A[\text{lower}(a)\ldots i-1])\\&\qquad\qquad\wedge\\&\qquad\qquad\underbrace{(\text{the suffix } a[i\ldots\text{upper}(a)] \text{ is still unmodified}}_{\text{part of the precondition}}).\end{aligned}$$

- Without this part of the precondition as the second conjunct, we would not be able to establish its first part when i increases (by 1):

  How else would we know that the current $a[i]$ still contains the same value as $A[i]$ in the original unmodified prefix?

- In terms of the balloon theory (Figure 13), both *pre*- and *postcondition* must imply the *invariant*, so their combinations with '$\wedge$' and '$\vee$' must imply it too.
- Here we started from

  $$\text{postcondition} \wedge \text{precondition}$$

  which means that

  "the whole array a is sorted and unmodified"

  —

  that is, the algorithm was given an already sorted input a

  then split both parts at i similarly to Eq. (35) and finally dropped the other conjuncts:

  $$\begin{gathered}(\text{postcondition when } < i)\cancel{\wedge(\text{postcondition when } \geq i)}\\ \wedge\\ \cancel{(\text{precondition when } < i)\wedge}(\text{precondition when } \geq i).\end{gathered}$$

3.4 Example: Edit Distance

- Let us now present an example where it is natural to *nest* loops and initialize them with *other* loops.
- Our example is computing the *edit distance.*
  - Let $A[1 \dots M]$ and $B[1 \dots N]$ be two input strings.
  - Their edit distance is the smallest number of single character insertions and deletions needed to turn A into B.
  - Define the function $\mathrm{ed}(p, q)$ mean intuitively "the smallest number of such operations needed to turn the prefix $A[1 \dots p]$ into the prefix $B[1 \dots q]$".
  - Its formal definition is the following recurrence:

$$\mathrm{ed}(0, q) = q \tag{39}$$

$$\mathrm{ed}(p+1, 0) = p+1 \tag{40}$$

$$\mathrm{ed}(p+1, q+1) = \min \begin{cases} \mathrm{ed}(p+1, q)+1 \\ \mathrm{ed}(p, q+1)+1 \\ \mathrm{ed}(p, q) \text{ but only if } A[p+1] = B[q+1]. \end{cases} \tag{41}$$

Eq. (39) says intuitively that "to turn the empty prefix $A[1 \dots 0]$ into the prefix $B[1 \dots q]$, you must insert each of the characters $B[1], B[2], B[3], \dots B[q]$".

Eq. (40) says in turn that "to make the nonempty $A[1 \dots p+1]$ empty, delete all its characters".

Eq. (41) says that the question can be reduced into a question on what to do to the *last* character:

  - The preceding characters can be solved recursively as a smaller case of the same problem.

- Then we can either insert $B[q+1]$ at the end, delete $A[p+1]$ from the end, or keep the last character as it is if $A[p+1]$ and $B[q+1]$ match.

- Edit distance (which has many variations) is a natural measure of string (dis)similarity, where the changes have happened character by character, independently of each other.

  E.g. it is a basic tool in *computational biology:* strings A and B are DNA molecules, and we ask how close they are, counting pointwise mutations.

- Our task is to fill the matrix $E[0 \dots M][0 \dots N]$ so that

$$\forall 0 \leq p \leq M : \forall 0 \leq q \leq N : E[p][q] = \mathrm{ed}(p,q) \tag{42}$$

  where the values $\mathrm{ed}(p,q)$ would be computed using the already tabulated other values, not by the recursive definition.

  – Because we don't have GCL recursion yet.

  – Also because direct recursion would lead into an *exponential* algorithm.

  – Instead, this *dynamic programming* solution is $\mathcal{O}(M \cdot N)$.

- The intuitive idea is to fill $E[p][q]$ only after the neighbours $E[p-1][q-1]$, $E[p-1][q]$ and $E[p][q-1]$ have already been filled, so they can be used instead of the recursion.

- While a seasoned programmer can "see" and implement the correct filling order, let us derive the algorithm formally instead.

  A practical proramer would use (hopefully) precise natural language statements such as "rows $< i$ have already been filled" instead of these explicit formulas.

- The postcondition (42) certainly needs massaging — it does not even have any free variables, which could be used in the code!

- A natural idea is to fill the table E row by row.

  Hence we introduce (Section 3.3.3) a new variable i meaning "the current row to be filled":

$$(\forall 0 \leq p \leq M : \forall 0 \leq q \leq N : E[p][q] = \mathrm{ed}(p,q)) \\ \wedge (0 \leq i) \wedge (i \leq M+1).$$

  Here we anticipate also the terminating value $i = M+1$ to signal that the table is completely filled.

- Then we replace the row limit M with i:

$$(\forall 0 \leq p \leq i : \forall 0 \leq q \leq N : E[p][q] = \mathrm{ed}(p,q)) \\ \wedge (0 \leq i) \wedge (i \leq M+1) \wedge (i = M).$$

- Then we weaken this postcondition to say just "rows $< i$ have been filled" and to permit incrementing i:

$$(\forall 0 \leq p < i : \forall 0 \leq q \leq N : E[p][q] = \mathrm{ed}(p,q)) \\ \wedge (0 \leq i) \wedge (i \leq M+1). \tag{43}$$

- Now our postcondition (43) is finally *constructive* in the sense that it has a natural initialization $i := 0$ which yields the

  invariant which is the whole (43) since all of it becomes TRUE in the initialization (Section 3.3.2).

  guard $(i \neq M+1)$: Note that its complement and this invariant together do imply the original postcondition (42), as they should in order for our algorithm to do what it promises.

- Our algorithm has reached the shape

```
$i \; := \; 0$ ;
{ outer invariant: Eq. (43)
  outer bound: number of still unfilled rows $M - i + 1$ }
do $i \neq M + 1 \rightarrow$
     fill row $0 \leq i \leq M$
     $i \; := \; i + 1$
od
```

  where we call this invariant and bound "outer" to anticipate an inner loop with its own invariant for filling the row $0 \leq i \leq M$.

- Hence our nested loops are grown from the outside in:

  - We create the outer loop first, then find out that we need an inner loop inside its body to satisfy its invariant.
  - Here the body part "fill row $0 \leq i \leq M$" will need another loop (with its own initializations, invariant, and bound).

- However, since row 0 is treated differently by Eq. (39) than the other rows $i > 0$, it makes sense to lift its initialization from the main loop into a simpler loop of its own:

```
initialize row 0 ;
$i \; := \; 1$ ;
{ outer invariant: Eq. (43)
  outer bound: number of still unfilled rows $M - i + 1$ }
do $i \neq M + 1 \rightarrow$
     fill row $1 \leq i \leq M$ ;
     $i \; := \; i + 1$
od
```

- Logically this means rewriting the invariant (43) into

$$(\forall 0 \leq q \leq N : E[0][q] = \overbrace{\mathrm{ed}(0, q)}^{=q}) \\ \wedge (\forall 1 \leq p < i : \forall 0 \leq q \leq N : E[p][q] = \mathrm{ed}(p, q)) \\ \wedge (0 \leq i) \wedge (i \leq M + 1)$$

  to see the invariant for initializing row 0.

- This invariant needs to be similarly weakened to introduce a variable k for the next column to initialize on row 0. We omit showing the straightforward development steps to get:

```
$k \ := \ 0$;
{ row 0 invariant: $\forall 0 \le q < k : E[0][q] = q$
  row 0 bound: number of still unfilled columns $N - k + 1$ }
do $k \ne N + 1 \rightarrow$
    $E[0][k] \, , \ k \ := \ k \, , \ k + 1$
od;
$i \ := \ 1$;
{ outer invariant: Eq. (43)
  outer bound: number of still unfilled rows $M - i + 1$ }
do $i \ne M + 1 \rightarrow$
    fill row $1 \le i \le M$;
    $i \ := \ i + 1$
od
```