4.5.1 Bubblesort

- Also bubblesort is developed using the same outer loop invariant (48) and the new part found in (49) which must be handled by the inner loop.
- Its development deviates by introducing j to *both* occurrences of i in this new part:

$$\underbrace{(a[j] \leq a[j+1 \ldots \mathrm{upper}(a)])}_{\text{Inner loop invariant.}} \wedge (\underbrace{j = i}_{\neg\text{guard.}}).$$

  The first conjunct namely contains the quantifier and is easily established by initializing $j := \mathrm{upper}(a)$.
- When we again ask $\mathrm{wp}(j := j-1, \text{this inner invariant})$ we find that $a[j-1] \leq a[j]$ must be ensured first. And again we can use the appropriate **if** command.
- The end result is:

$$\begin{array}{l}
i\,,\ a\ :=\ \mathrm{lower}(a)\,,\ A\,; \\
\textbf{do}\ \ i \neq \mathrm{upper}(a) \rightarrow \\
\quad j\ :=\ \mathrm{upper}(a)\,; \\
\quad \textbf{do}\ \ j \neq i \rightarrow \\
\qquad \textbf{if}\ \ a[j-1] \leq a[j] \rightarrow \\
\qquad\quad \textbf{skip} \\
\qquad [\!]\ \ a[j-1] > a[j] \rightarrow \\
\qquad\quad a[j-1]\,,\ a[j]\ :=\ a[j]\,,\ a[j-1] \\
\qquad \textbf{fi}\,; \\
\qquad j\ :=\ j-1 \\
\quad \textbf{od}\,; \\
\quad i\ :=\ i+1 \\
\textbf{od}
\end{array}$$

- In practice, bubblesort is one of the least efficient sorting algorithms. It is effective only when the input A is already nearly sorted.

4.5.2 Heapsort

- Rather than developing a separate heap data structure first, we develop it alongside our algorithm.
- This development goes back into the original postcondition

$$\overbrace{\forall\, \mathrm{lower}(a) \leq k < \mathrm{upper}(a) : a[\mathrm{lower}(a) \ldots k] \leq a[k+1 \ldots\ \mathrm{upper}(a)]}^{\text{this is } \mathit{ordered}(a)} \wedge \mathit{perm}(a, A)$$

  but drops the section $a[\mathrm{lower}(a) \ldots k-1]$ to get another sorting specification:

$$\forall\, \mathrm{lower}(a) \leq k < \mathrm{upper}(a) : a[k] \leq a[k+1 \ldots\ \mathrm{upper}(a)] \wedge \mathit{perm}(a, A).$$

- That is, for each k we have

$$a[k] \leq a[k+1] \wedge a[k] \leq a[k+2] \wedge a[k] \leq a[k+3] \wedge \ldots \cdots \wedge a[k] \leq a[\mathrm{upper}(a)].$$

- We weaken this conjunction by keeping only those

$$a[k] \leq a[2 \cdot k] \wedge a[k] \leq a[2 \cdot k + 1] \text{ where } k < 2 \cdot k < 2 \cdot k + 1 \leq \text{upper}(a).$$

  where we assume henceforth that $\text{lower}(a) = \text{lower}(A) = 1$ to make this work and simplify the other expressions too.

- This is where we discovered the *(binary) heap property* (Cormen et al., 2001, Chapter 6.1).

  Noting that this property is faster to maintain than the simpler inner loops of our previous sorting algorithms is again in the eye of the seasoned programmer, not explicitly in its logical specification.

- This weaker form can be written as

$$\begin{aligned} &\mathit{perm}(a, A) \wedge (\forall 1 \leq k < \text{upper}(a) : \\ &\qquad (2 \cdot k \leq \text{upper}(a) \implies a[k] \leq a[2 \cdot k]) \wedge \\ &\qquad\qquad (2 \cdot k + 1 \leq \text{upper}(a) \implies a[k] \leq a[2 \cdot k + 1])). \end{aligned}$$

- Since the only possible initialization $a := A$ does not get us very far, let us add a new variable $j = 1$ which we can manipulate.

- The initialization $j, a := \text{upper}(a) \operatorname{div} 2 + 1, A$ satisfies our invariant:

$$\begin{aligned} &\mathit{perm}(a, A) \wedge (1 \leq j) \wedge (j \leq \text{upper}(a) \operatorname{div} 2 + 1) \wedge \\ &\qquad (\forall j \leq k < \text{upper}(a) \operatorname{div} 2 + 1 : \\ &\qquad (2 \cdot k \leq \text{upper}(a) \implies a[k] \leq a[2 \cdot k]) \\ &\qquad\quad \wedge (2 \cdot k + 1 \leq \text{upper}(a) \implies a[k] \leq a[2 \cdot k + 1])). \end{aligned} \tag{54}$$

- Now our algorithmic skeleton is

$$\begin{array}{l} j\,,\ a\ :=\ \text{upper}(a) \operatorname{div} 2 + 1, A\,; \\ \textbf{do}\ \ j \neq 1 \rightarrow \\ \quad \text{decrement } j \text{ keeping (54) } \textsc{true} \\ \textbf{od} \end{array}$$

  which says that $a[j \ldots \text{upper}(a)]$ satisfies our heap relation, but j needs to be decremented too.

- It terms of the heap data structure, we are now inventing the "heapify" algorithm (Cormen et al., 2001, Chapters 6.2–6.3).

- This extension requires that we swap elements of $a[j - 1 \ldots \text{upper}(a)]$ around until the heap property is restored.

- This swapping can be recast as searching for a suitable index k for $a[j - 1]$ in $a[j \ldots \text{upper}(a)]$ and swapping its elements out of the way while keeping the heap property elsewhere than in $a[k]$.

$k \;:=\; j\,;$
do $(k < n \operatorname{div} 2)$ **cand** $(a[k] > \max(a[2\cdot k], a[2\cdot k+1])) \to$
$\quad m \;:=\; \arg\min(a[2\cdot k], a[2\cdot k+1])\,;$
$\quad k\,,\;\; a[k]\,,\;\; a[m] \;:=\; m\,,\;\; a[m]\,,\;\; a[k]$
od

The mathematical notation 'arg min $f(x)$' means the *arg*ument value x which *min*imizes f. Hence here m is either $x = 2\cdot k$ or $x = 2\cdot k+1$, depending on whose $a[x]$ is smaller.

Figure 15: The "Sifting" Algorithm.

- Examining the result of $\mathrm{wp}(j := j-1, \text{invariant (54)})$ shows an invariant for this search:

$$\begin{aligned} &\mathit{perm}(a, A) \wedge (\forall j \le p < k : \\ &\qquad (2\cdot p \le k \implies a[p] \le a[2\cdot p]) \\ &\quad \wedge (2\cdot p + 1 \le k \implies a[p] \le a[2\cdot p+1])) \\ &\qquad ((k \ge n \operatorname{div} 2)\ \mathbf{cor}\ (a[k] \le \min(a[2\cdot k], a[2\cdot k+1]))). \end{aligned}$$

  The last line says that " the current position $a[k]$ is OK" whereas the preceding lines say "the preceding positions $a[j \ldots k-1]$ are still OK even though our search has already passed over them".

- Here we have added another variable $n = \mathrm{upper}(a)$ whose intuition is that $a[n+1 \ldots \mathrm{upper}(a)]$ is now ready. Formally

$$\mathit{ordered}(a[n+1 \ldots \mathrm{upper}(a)]) \wedge a[1 \ldots n] \le a[n+1 \ldots \mathrm{upper}(a)].$$

- Thus we now know what kind of a search loop we must write (Figure 15).

- Then our "heapify" algorithm has reached the form

$\quad j\,,\;\; a\,,\;\; n \;:=\; \mathrm{upper}(a) \operatorname{div} 2 + 1\,,\;\; A\,,\;\; \mathrm{upper}(a)\,;$
$\quad$**do** $j \ne 1 \to$
$\qquad j \;:=\; j-1\,;$
$\qquad$paste our "sifting" code (Figure 15) here
$\quad$**od**

  where j is decremented *before* sifting because its development assumed that $a[j]$ might violate the heap property, while this loop assumed that $a[j \ldots \mathrm{upper}(a)]$ is OK.

- Our heapsort main algorithm is in turn

$\quad$paste our "heapify" code here
$\quad$**do** $n \ne 1 \to$
$\qquad n\,,\;\; a[1]\,,\;\; a[n] \;:=\; n-1\,,\;\; a[n]\,,\;\; a[1]\,;$
$\qquad$paste our "sifting" code (Figure 15) here with $j = 1$
$\quad$**od**

  or "extend the answer $a[n \ldots \mathrm{upper}(a)]$ by adding $a[1]$ in front and restore the one element shorter heap".

- However, this main algorithm produces the answer in the *opposite* order:

$$\forall 1 \leq k < \mathrm{upper}(a) : a[k] \geq a[k+1 \ldots \mathrm{upper}(a)]!$$

- This can be fixed in two ways:

  either with a postprocessing step to reverse the answer.

  or by moving from this 'min' heap into the 'max' heap where larger elements precede the smaller ones. (Cormen et al., 2001, Chapter 6.4)

  That is, by swapping all the element comparisons around in the appropriate places, so that $a[x] < a[y]$ becomes $a[y] < a[x]$ and so on.

5 Procedures

The treatment for nonrecursive procedures is from Gries (1981, Chapter 12). The treatment for recursive procedures is adapted from Morgan (1994, Chapter 13).

- The heapsort example (Section 4.5.2) showed the need to replicate the same code in different places:

  The "sifting" algorithm was used both in the "heapify" algorithm and in the main loop, and with different values for its variable j.

- It shows the need for *procedures* which can written only once, and called from several places with different parameter values.

- They are central examples of *abstraction:* showing the interesting features of an object while hiding the rest.

- Here the

  shown features are the *pre-* and *post*conditions of this procedure: They together state *what* it does by stating "if the inputs you give me satisfy this precondition, then I will give you an answer which satisfies that postcondition in return".

  hidden feature is its code: *How* it produces this answer.

5.1 Procedure Declaration

- Let us allow **proc**edure declarations

  { pre: ϕ
    post: ψ }
  proc *procname* (*parspec* ; . . . ; *parspec*) ;
  body

  on the *top level* of GCL listings.

- Intuitively, this declares a procedure which

  – is called *procname*

  – communicates with its caller via these *par*ameter *spec*ifiers

– executes the hidden code

$$\{\ \phi\ \}\ body\{\ \psi\ \}$$

when called.

- Allowing only "flat" (unnested) procedures simplifies name visibility.
- Similarly, variables not mentioned in its *par*ameter *spec*ifiers are *local* to its *body* — no global variables!
- Each *par*ameter *spec*ifier is in turn of the form

$$mode \quad parname, \ldots, parname : type$$

where the *mode* is one or both of

value which means that these *par*ameter *name*s denote values given by the caller to this procedure as inputs

result which means that they denote values returned by this procedure as answers to its caller.

- If a *par*ameter *name* appears within the

precondition ϕ then its *mode* must contain **value** since it describes what is given

postcondition ψ then its *mode* must contain **result** since it describes what is returned.

In addition, the pre- and postcondition ϕ and ψ must not contain any of the local variables in the *body*, since their values are hidden from the outside world.

- E.g. our "sifting" algorithm can be abstracted into the following procedure:

$\{$ pre: $perm(a, A) \wedge j = J \wedge j \geq 1 \wedge n = N \wedge n \leq \mathrm{upper}(a)$
$\wedge a[j+1 \ldots n]$ satisfies the heap property
post: $perm(a, A)$
$\wedge a[J \ldots N]$ satisfies the heap property $\}$
proc *sift*(**value result** a: **array of** τ;
value $j, n : \mathbb{Z}$);
our "sifting" code (Figure 15) as the body

- Now its k has become a local variable invisible to the outside.
- Note how ghost variables J and N had to be introduced so that we can mention the **value** parameters j and n in the postcondition.
- Note also that not only the details of the code but also of its correctness *proof* are hidden behind the abstraction. A mathematical analogy is that we have now produced a *lemma*.
- We have left the type τ unspecified, since any ordered type (= any type which has a definition for '$\leq$') will do.

(This is so-called *parametric polymorphism:* This procedure works for any suitable parameter type τ, but all elements of array a must have this same type. This is different from the *ad-hoc* polymorphism in object oriented programming languages, where the interpretation would be instead that all elements of a must belong to subclasses of τ. But is comparing elements from different subclasses meaningful in general?)

5.2 Procedure Call

- The procedure call is a new *command* of the form

$$procname(parvalue_1, \ldots, parvalue_m)$$

such that:

  - The declaration of *procname* had the same number m of *parname*s.
    As expected, kth *parvalue* corresponds to kth *parname*.
  - For each **result** $parname_k$, the corresponding $parvalue_k$ is anything which is allowed on the *left* side of the assignment ':=' and has the correct type.
  - For each **value** $parname_k$, the corresponding $parvalue_k$ is anything which is allowed on the *right* side of the assignment ':=' and has the correct type.

- We also assume that the *parname*s have been chosen distinct from each other and the variables visible at this call site. This keeps the notation simple. (Or it could be easily lifted by renaming them.)

- Assume that our procedure declaration is of the form

$$\begin{array}{l}\textbf{proc}\ p(\textbf{value}\ \vec{x};\ \textbf{value result}\ \vec{y};\ \textbf{result}\ \vec{z});\\ \{\ \phi\ \}\mathcal{B}\{\ \psi\ \}\end{array}$$

where we omit the types from $\vec{x}, \vec{y}, \vec{z}$ by assuming that they do match.

- Then consider the call $p(\vec{a}, \vec{b}, \vec{c})$ where $\vec{a}$ corresponds to $\vec{x}$, $\vec{b}$ to $\vec{y}$ and $\vec{c}$ to $\vec{z}$.

- We define its meaning as

$$\mathrm{wp}(p(\vec{a}, \vec{b}, \vec{c}), \varphi) = \mathrm{wp}((\underbrace{\vec{x}, \vec{y} := \vec{a}, \vec{b}}_{1.}; \underbrace{\mathcal{B}}_{2.}; \underbrace{\vec{b}, \vec{c} := \vec{y}, \vec{z}}_{3.}), \varphi). \quad (55)$$

  1. First assign the input **value**s $\vec{a}, \vec{b}$ of the caller into their formal parameters $\vec{x}, \vec{y}$ of the called procedure p,
  2. then execute the body $\mathcal{B}$ of the called procedure p on these values for $\vec{x}, \vec{y}$ and uninitialized $\vec{z}$, and
  3. finally assign the **result**ing answers from the formal parameters $\vec{y}, \vec{z}$ of the called procedure p into the variables $\vec{b}, \vec{c}$ of its caller.

- The intuition is "replace this call of this *macro* p textually with its body $\mathcal{B}$".

- This works for *non-recursive* procedures p. Let us assume this for the moment, and return to recursion later.

- E.g. our heapsort main algorithm becomes (Section 4.5.2):

$$\begin{array}{l}\{\ \text{pre: } a = A \wedge \mathrm{lower}(a) = 1 \wedge \mathrm{upper}(a) \geq \mathrm{lower}(a)\\ \quad \text{post: reverse } \mathit{ordered}(a) \wedge \mathit{perm}(a, A)\ \}\\ \textbf{proc}\ \mathit{heapsort}(\textbf{value result}\ a\colon\ \textbf{array of}\ \tau);\\ j\ :=\ \mathrm{upper}(a)\ \mathrm{div}\ 2 + 1;\\ \textbf{do}\ j \neq 1 \rightarrow\end{array}$$

$j \;:=\; j-1\,;$
$sift(a, j, \mathrm{upper}(a))$
od;
$n \;:=\; \mathrm{upper}(a)\,;$
do $n \neq 1 \rightarrow$
$n\,,\;\; a[1]\,,\;\; a[n] \;:=\; n-1\,,\;\; a[n]\,,\;\; a[1]\,;$
$sift(a, 1, n)$
od;

- Let us now show some results about such non-recursive calls which are easier to use than the exact definition (55).

Theorem 15 (General Procedure Call)**.** *The following Hoare triple holds:*

$\{$ *1.* $\phi[\vec{x} \leftarrow \vec{a}, \vec{y} \leftarrow \vec{b}] \wedge$
2. $\forall \vec{u}, \vec{v} : \psi[\vec{y} \leftarrow \vec{u}, \vec{z} \leftarrow \vec{v}] \Longrightarrow \varphi[\vec{b} \leftarrow \vec{u}, \vec{c} \leftarrow \vec{v}]$ $\}$
$p\,(\vec{a}\,, \vec{b}\,, \vec{c}\,)$
$\{\; \varphi \;\}$

In other words,

$$1. \wedge 2. \Longrightarrow \mathrm{wp}(p(\vec{a}, \vec{b}, \vec{c}), \varphi)$$

where the two-part antecedent reads that

1. *the input* ***value****s* $\vec{a}, \vec{b}$ *(as* $\vec{x}, \vec{y}$*) satisfy the precondition* ϕ *of* p
2. φ *holds for all possible values* $\vec{u}, \vec{v}$ *which* p *might give on these inputs to the* ***result****s* $\vec{b}, \vec{c}$ *(as* $\vec{y}, \vec{z}$*). These* $\vec{u}, \vec{v}$ *are in turn determined by the postcondition* ψ *of* p*.*

Proof sketch. Suppose for the moment that we know the correct final data values $\vec{u}, \vec{v}$ for the **result**s $\vec{y}, \vec{z}$.

By definition (55) the whole call can then be summarized as the assignment sequence

$$\vec{x}, \vec{y} := \vec{a}, \vec{b} \overbrace{\{\; \phi \;\} \underbrace{\vec{y}, \vec{z} := \vec{u}, \vec{v}}_{\text{its execution}} \{\; \psi \;\}}^{\text{specification for body } \mathcal{B} \text{ of } p} \vec{b}, \vec{c} := \vec{y}, \vec{z} \{\; \varphi \;\} \tag{56}$$

which we must – and can – verify via wp calculations:

$$\mathrm{wp}(\ldots, \phi) = \text{property 1 in this Theorem} \tag{57}$$

$$\begin{aligned} \mathrm{wp}(\ldots, \psi) &= \psi[\vec{y} \leftarrow \vec{u}, \vec{z} \leftarrow \vec{v}][\vec{x} \leftarrow \vec{a}, \vec{y} \leftarrow \vec{b}] \\ &= \psi[\vec{y} \leftarrow \vec{u}, \vec{z} \leftarrow \vec{v}] \qquad \text{since it has no } \vec{x}, \vec{y} \end{aligned} \tag{58}$$

$$\begin{aligned} \mathrm{wp}(\ldots, \varphi) &= \varphi[\vec{b} \leftarrow \vec{y}, \vec{c} \leftarrow \vec{z}][\vec{y} \leftarrow \vec{u}, \vec{z} \leftarrow \vec{v}][\vec{x} \leftarrow \vec{a}, \vec{y} \leftarrow \vec{b}] \\ &= \varphi[\vec{b} \leftarrow \vec{u}, \vec{c} \leftarrow \vec{v}] \qquad \text{since it has no } \vec{x}, \vec{y}. \end{aligned} \tag{59}$$

Next we use our knowledge about procedure p: $\{\; \phi \;\}\mathcal{B}\{\; \psi \;\}$. It and Eq. (57) shows that ψ holds in the designated place of Eq. (56).

Hence, φ is guaranteed to hold after the call, if ψ implies it, no matter what $\vec{u}, \vec{v}$ happened to be. This condition is $\forall \vec{u}, \vec{v}$: Eq. (58) $\Longrightarrow$ Eq. (59), or property 2. □

- How does the information $\vec{x} \leftarrow \vec{a}$ pass from ϕ into ψ? This theorem (Theorem 15) does not show it explicitly!

  It passes through the ghost variables $\vec{X}$ that must be introduced for $\vec{x}$ in ϕ so that they can be mentioned in ψ.

- Let us now use this result (Theorem 15) with a procedure

$$\begin{array}{l}\{\text{ pre: } y1 = X \wedge y2 = Y \\ \quad \text{post: } y1 = Y \wedge y2 = X \ \} \\ \textbf{proc } swap(\textbf{value result } y1, y2 : \tau\,); \\ \mathcal{C}\end{array}$$

  whose body $\mathcal{C}$ no longer interests us, since we can use its specification instead.

- Such a routine might be used for abstracting away the proofs for the basic operation of our sorting algorithms.

- Let us begin with the very basic result:

$$\{\ a = X \wedge b = Y\ \}\, swap(a, b)\{\ a = Y \wedge b = X\ \}. \tag{60}$$

  We can now calculate a precondition for this call according to this theorem:

$$\begin{array}{l}(y1 = X \wedge y2 = Y)[y1 \leftarrow a, y2 \leftarrow b] \wedge \\ \forall\, u1, u2 \colon (y1 = Y \wedge y2 = X)[y1 \leftarrow u1, y2 \leftarrow u2] \Longrightarrow \\ \qquad (a = Y \wedge b = X)[a \leftarrow u1, b \leftarrow u2] \\ \Leftrightarrow a = X \wedge b = Y \wedge \\ \underbrace{\forall\, u1, u2 \colon\ u1 = Y \wedge u2 = X \Longrightarrow u1 = Y \wedge u2 = X}_{\text{TRUE}}\end{array}$$

  Hence the calculated precondition is implied by (indeed, equals) the stated precondition. So the call is indeed valid.

- Consider then proving

$$\{\ a = A \wedge b = Y\ \}\, swap(a, b)\{\ a = Y \wedge b = A\ \} \tag{61}$$

  where we have used X in the specification but A in the call.

  - Recall that free variables have an implicit '$\forall$' so that we can consider our specification to be

$$\begin{array}{l}\forall Z, U : \{\text{ pre: } y1 = Z \wedge y2 = U \\ \qquad\qquad \text{post: } y1 = U \wedge y2 = Z \ \} \\ \qquad\qquad \textbf{proc } swap(\textbf{value result } y1, y2 : \tau\,); \\ \qquad\qquad \mathcal{C}\end{array}$$

    to emphasize that the choices for the actual ghost variable names are immaterial.

  - ... But we have defined quantifiers only in formulas, not in code, so we cannot actually write like this.

  - Nevertheless, we can substitute $[Z \leftarrow A, U \leftarrow Y]$ in our specification, and think that "we could repeat the proof of the *swap* procedure with A in place of Z and Y in place of U if we wanted to" and continue as before.

- Consider then showing that swapping i and $b[i]$ does not affect the rest of the array b (where i is assumed to be its valid index):

$$
\begin{array}{l}
\{\ i = I \wedge \forall\, \mathrm{lower}(b) \le j \le \mathrm{upper}(b) : b[j] = B[j]\ \} \\
\mathit{swap}(i, b[i]) \\
\{\ i = B[I] \wedge b[I] = I \wedge \\
\quad \forall\, \mathrm{lower}(b) \le j \le \mathrm{upper}(b) : I \neq j \Longrightarrow b[j] = B[j]\ \}
\end{array}
$$

Now the theorem gives a precondition

$$
\begin{array}{l}
i = I \wedge b[i] = B[I] \wedge \\
\forall\, \mathit{u1}, \mathit{u2} : \mathit{u1} = B[I] \wedge \mathit{u2} = I \Longrightarrow \\
\qquad \mathit{u1} = B[I] \wedge \langle b; [i] \leftarrow \mathit{u2} \rangle\, [I] = I \wedge \\
\qquad \forall\, \mathrm{lower}(b) \le j \le \mathrm{upper}(b) : I \neq j \Longrightarrow \langle b; [i] \leftarrow \mathit{u2} \rangle\, [j] = B[j]
\end{array}
$$

since passing the **result**s back to the caller is $i, b[i] := \mathit{y1}, \mathit{y2} := \mathit{u1}, \mathit{u2}$ by the assignments in Eq. (56).

Then we can get rid of the outer '∀' by first replacing $\mathit{u1} \leftarrow B[I]$ and $\mathit{u2} \leftarrow I$ in its body, and then simplifying with the first conjunct:

$$
\begin{array}{ll}
\Leftrightarrow & i = I \wedge b[i] = B[I] \wedge \\
& \qquad B[I] = B[I] \wedge \langle b; [i] \leftarrow I \rangle\, [I] = I \wedge \\
& \qquad \forall\, \mathrm{lower}(b) \le j \le \mathrm{upper}(b) : I \neq j \Longrightarrow \langle b; [i] \leftarrow I \rangle\, [j] = B[j] \\
\Leftrightarrow & i = I \wedge b[i] = B[I] \wedge \\
& \qquad \mathrm{TRUE} \wedge I = I \wedge \\
& \qquad \forall\, \mathrm{lower}(b) \le j \le \mathrm{upper}(b) : I \neq j \Longrightarrow b[j] = B[j].
\end{array}
$$

This final form is in turn implied by the given precondition, so the call has been verified.

- Usually, procedure calls are much simpler than the wholly general case treated here. In particular, usually
  - each **result** goes into its own variable, and
  - parts of the pre- and postconditions are independent of them.

  Hence we can specialize this theorem (Theorem 15) accordingly (without proof):

Theorem 16 (Simpler Procedure Call)**.** *Let all the* ***result****s* $\vec{b}, \vec{c}$ *be distinct variable names. Let also* ι *be a formula which does not mention any of them. Then*

$$
\{\ \phi[\vec{x} \leftarrow \vec{a}, \vec{y} \leftarrow \vec{b}] \wedge \iota\ \} p(\vec{a}, \vec{b}, \vec{c}) \{\ \psi[\vec{y} \leftarrow \vec{b}, \vec{z} \leftarrow \vec{c}] \wedge \iota\ \}.
$$

- That is, the call $p(\vec{a}, \vec{b}, \vec{c})$ turns the precondition ϕ with the given **value** parameters $\vec{a}, \vec{b}$ into the postcondition ψ with the given **result** parameters $\vec{b}, \vec{c}$ without affecting any other variables.

- E.g. Eq. (60) becomes very easy with this simpler form (Theorem 16):

$$
\underbrace{\{\ a = X \wedge b = Y\ \}}_{=\mathrm{pre}[y1 \leftarrow a, y2 \leftarrow b]} \mathit{swap}(a, b) \underbrace{\{\ a = Y \wedge b = X\ \}}_{=\mathrm{post}[y1 \leftarrow a, y2 \leftarrow b]}.
$$

- Or consider the call $sift(b, 1, m)$:
  - Here we have used b and m instead of a and n as in our *heapsort* to agree with our simplifying assumption that the parameter names are distinct from the variables at the calling site.
  - We get the precondition

$$\begin{aligned}\text{pre}[a \leftarrow b, j \leftarrow 1, n \leftarrow m] = \\ perm(b, A) \wedge b[1+1 \ldots m] \text{ satisfies the heap property} \wedge \\ m \leq \text{upper}(b) \wedge 1 = J \wedge 1 \geq 1 \wedge m = N\end{aligned}$$

    and the postcondition

$$\begin{aligned}\text{post}[a \leftarrow b] &= perm(b, A) \wedge b[J \ldots N] \text{ satisfies the heap property} \\ &= perm(b, A) \wedge b[1 \ldots m] \text{ satisfies the heap property.}\end{aligned}$$

  - Since the preceding code satisfies this precondition, we conclude that the call restores the heap for the next loop round.
- However, this shows that forbidding **value** parameters from the postcondition leads into unnecessary detours through ghost variables. Let us therefore state (without proof) conditions when such an x_i can be allowed there. Intuitively, when its value remains the same throughout the call.

Theorem 17 (Value Parameter in Postcondition)**.** *Assume that*

1. *the* ***value*** *parameter x_i of the called procedure p does not appear on the left side of ':=' in its body $\mathcal{B}$*
2. *the corresponding expression a_i in the call $p(\vec{a}, \vec{b}, \vec{c})$ does not mention any of the* ***result****s $\vec{b}, \vec{c}$.*

Then we can allow $\psi[x_i \leftarrow a_i, \ldots]$ in the general (Theorem 15) and simpler calls (Theorem 16).

Principle 8 (Read-Only Parameters)**.** *Assign to the procedure parameters only when producing the* ***result****. This guarantees assumption 1 (Theorem 17).*

- E.g. our *sift* procedure adheres to this principle (Principle 8).
- Let us furthermore adopt assumption 2 (Theorem 17) which means that the values for parameters j and n must not mention the parameter a.
- Then we can state its specification more directly without the ghost variables J and N as

$$\begin{aligned}&\{ \text{pre: } perm(a, A) \wedge j \geq 1 \wedge n \leq \text{upper}(a) \\ &\qquad \wedge a[j+1 \ldots n] \text{ satisfies the heap property} \\ &\quad \text{post: } perm(a, A) \\ &\qquad \wedge a[j \ldots n] \text{ satisfies the heap property } \} \\ &\textbf{proc } sift(\textbf{value result } a\colon \textbf{array of } \tau; \\ &\qquad \textbf{value } j, n : \mathbb{Z})\,;\end{aligned}$$

  our "sifting" code (Figure 15) as the body

  which is also easier to use.

5.3 Recursion

- Consider then a *recursive* procedure declaration

  $\{$ pre: ϕ
  post: ψ $\}$
  proc $p($**value** $\vec{x}$; **value result** $\vec{y}$; **result** $\vec{z}$$)$;
  $\mathcal{B}$

  whose body now contains calls

  $$p(\vec{a}, \vec{b}, \vec{c})$$

  to itself.

- How should we treat these calls when verifying

  $$\{\ \phi\ \}\mathcal{B}\{\ \psi\ \}$$

  to see that p indeed agrees with its stated specification?

- We must somehow assume what we are proving: handling such a recursive call already needs its stated pre- and postconditions ϕ and ψ.

- This smells like induction...

- But we cannot do it entirely freely:

  $\{$ pre: ϕ'
  post: ψ' $\}$
  proc $q($**value** $x$$)$;
  $q(x)$

  is incorrect, even though we can "prove"

  $$\{\ \phi'\ \}q(x)\{\ \psi'\ \}$$

  by assuming it!

- This q namely does not terminate.

- This is similar to "proving" an inductive claim $P(n+1)$ by assuming $P(n+1)$ itself, and not $P(n)$ as it should have been done.

- We introduced the *bound*ing expression to ensure termination in the **do** command (Section 2.4.7).

- Let us use the same tool here too, and use the specification

  $\{$ pre: ϕ
  post: ψ
  bound: $f(\vec{x}, \vec{y})$ $\}$
  proc $p($**value** $\vec{x}$; **value result** $\vec{y}$; **result** $\vec{z}$$)$;
  $\mathcal{B}$

  where the *bound*ing expression $f(\vec{x}, \vec{y})$

  – mentions only (some of) the **value** parameters $\vec{x}, \vec{y}$ since they are the only initialized variables in the beginning of the body $\mathcal{B}$

  - produces an integer value $\in \mathbb{Z}$ from them
  - precludes further recursive calls when this value drops ≤ 0, and
  - decreases strictly within each call $p(\vec{a}, \vec{b}, \vec{c})$: $f(\vec{a}, \vec{b})$ $<$ than in the beginning of $\mathcal{B}$.

- Intuitively, f measures how *complicated* the **value** parameters are:
  - They must get simpler at each recursive call, and
  - eventually so simple that recursion is no longer needed.

- We check such a specification as follows:
  - Checking the body

    $$\{\ \phi\ \}\mathcal{B}\{\ \psi\ \}$$

    is extended to checking

    $$\{\ (\mathit{Bound} = f(\vec{x}, \vec{y})) \wedge \phi\ \}\mathcal{B}\{\ \psi\ \}$$

    where *Bound* is a fresh ghost variable for capturing the value of f in the beginning of $\mathcal{B}$.
  - When this checking has reached a recursive call

    $$\ldots\{\ \phi'\ \}p(\vec{a}, \vec{b}, \vec{c})\{\ \psi'\ \}$$

    we add two further requirements to its precondition:

    $$\ldots\{\ (0 < \mathit{Bound}) \wedge (f(\vec{a}, \vec{b}) < \mathit{Bound}) \wedge \phi'\ \}p(\vec{a}, \vec{b}, \vec{c})\{\ \psi'\ \}.$$

    1. This recursive call must be permitted to make others.
    2. The preceding code must have assured simpler $\vec{a}, \vec{b}$.

- E.g. consider the factorial function

  $$0! = 1$$
  $$(n+1)! = (n+1) \cdot (n!)$$

  written recursively as

```
{ pre: n ∈ ℕ
  post: m = n!
  bound: n }
proc fac(value n : ℕ; result m : ℕ);
if n = 0 →
    m := 1
[] n > 0 →
    fac(n − 1, t);
    m := n · t
fi
```

- Checking it starts with:

$$
\begin{array}{l}
\{\ \mathit{Bound} = n \wedge n \in \mathbb{N}\ \} \\
\textbf{if}\ \ n = 0 \rightarrow \\
\quad m\ :=\ 1 \\
[\!]\ \ n > 0 \rightarrow \\
\quad \mathit{fac}(n-1, t)\,; \\
\quad m\ :=\ n \cdot t \\
\textbf{fi} \\
\{\ m = n!\ \}
\end{array}
$$

- Then checking this **if** (Section 2.4.5) means

$$
\begin{array}{l}
\{\ \mathit{Bound} = n \wedge n \in \mathbb{N} \wedge (n = 0 \vee n > 0)\ \} \\
\textbf{if}\ \ n = 0 \rightarrow \\
\quad \{\ \mathit{Bound} = n \wedge n \in \mathbb{N} \wedge n = 0\ \} \\
\quad m\ :=\ 1 \\
\quad \{\ m = n!\ \} \\
[\!]\ \ n > 0 \rightarrow \\
\quad \{\ \mathit{Bound} = n \wedge n \in \mathbb{N} \wedge n > 0\ \} \\
\quad \mathit{fac}(n-1, t)\,; \\
\quad m\ :=\ n \cdot t \\
\quad \{\ m = n!\ \} \\
\textbf{fi}
\end{array}
$$

- Now the Hoare triple in the first branch clearly holds, so that is verified.
- Consider then the second, recursive branch. Handling the **result** assignment and adding the two requirements gives:

$$
\begin{array}{l}
[\!]\ \ n > 0 \rightarrow \\
\quad \{\ (0 < \mathit{Bound}) \wedge (n-1 < \mathit{Bound}) \wedge \\
\quad\ \ \mathit{Bound} = n \wedge n \in \mathbb{N} \wedge n > 0\ \} \\
\quad \mathit{fac}(n-1, t)\,; \\
\quad \{\ t = (n-1)!\ \} \\
\quad m\ :=\ n \cdot t \\
\quad \{\ m = n!\ \}
\end{array}
$$

  - The two added requirements are clearly TRUE, so the recursion is permitted.
  - Hence we are permitted to assume (Theorem 17)

$$
\begin{array}{l}
\{\ \mathrm{pre}[n \leftarrow n-1] \quad = \quad (n-1 \in \mathbb{N})\ \} \\
\mathit{fac}(n-1, t)\,; \\
\{\ \mathrm{post}[n \leftarrow n-1, m \leftarrow t] \quad = \quad (t = (n-1)!)\ \}
\end{array}
$$

  - This precondition is implied by the one in the branch, while this postcondition implies (indeed, equals) the one in the branch. Hence we have verified this branch too.

- A small technical wrinkle is that we got e.g.

$$n \leftarrow n - 1$$

  where the two **value** parameters mix:

left n is the one in the call which is being made at this point

right n is the one in this current call.

This was caused by forgetting our simplification that parameter names and caller variables should be distinct.

- In recursion, how could they be?
- One way would be to take a fresh copy

$$
\begin{array}{l}
\{\ \mathrm{pre}' :\ n' \in \mathbb{N} \\
\quad \mathrm{post}' :\ m' = n'!\ \} \\
\textbf{proc}\ \ \mathit{fac}\,(\,\textbf{value}\ \ n' : \mathbb{N}\,;\ \ \textbf{result}\ \ m' : \mathbb{N}\,)\,; \\
\vdots
\end{array}
$$

of the specification, as in Eq. (61), with fresh names.

- Then in this fresh copy we can separate

$$
\begin{array}{l}
\{\ \mathrm{pre}'[n' \leftarrow n-1] \quad = \quad (n-1 \in \mathbb{N})\ \} \\
\mathit{fac}(n-1, t)\,; \\
\{\ \mathrm{post}'[n' \leftarrow n-1, m' \leftarrow t] \quad = \quad (t = (n-1)!)\ \}
\end{array}
$$

and get the same result without possibility of confusion.

- This is an informal example of the renamings needed to lift our naming simplification. Some mechanism like this is required for exact treatment of recursion.