- We might define the meaning for the recursive declaration

$$\mathrm{wp}(\textbf{proc}\ p(\vec{x},\vec{y},\vec{z});\mathcal{B},\phi)$$

similarly to Eq. (23) for **do** roughly as follows:

  - We start computing

$$\mathrm{wp}(\mathcal{B},\phi)$$

    for its body $\mathcal{B}$ as usual.
  - When we encounter a recursive call, then we can calculate

$$\begin{aligned}&\mathrm{wp}(p(\vec{a},\vec{b},\vec{c}),\psi)\\ =\ &\mathrm{wp}((\vec{x},\vec{y}:=\vec{a},\vec{b};\mathcal{B};\vec{b},\vec{c}:=\vec{y},\vec{z}),\psi)\\ =\ &\mathrm{wp}((\vec{x},\vec{y}:=\vec{a},\vec{b}),\underbrace{\mathrm{wp}((\mathcal{B};\vec{b},\vec{c}:=\vec{y},\vec{z}),\psi)}_{\text{The repeating part.}})\end{aligned}$$

    but no further: re-entering $\mathcal{B}$ would just lead us into this same situation again and again, and our formula would just get longer and longer.
  - Let us now *assume* (analogously to induction) that we already knew the weakest precondition $\mathcal{X}$ we are currently constructing.
  - Although we do not know what this $\mathcal{X}$ is actually like, we do know that it must be strong enough to validate this call.
  - Hence we replace this repeating part with the implication

$$\mathcal{X}\implies\mathrm{wp}((\vec{b},\vec{c}:=\vec{y},\vec{z}),\psi).$$

  - This way we do get a finite formula φ, but it has placeholders $\mathcal{X}$ which must be filled.
  - They are filled with a new atomic formula $\mathcal{G}^p_\phi(k)$ meaning "calls to p restricted to recursion depth $\leq k$":

$$\begin{aligned}\mathcal{G}^p_\phi(0)&\iff\mathrm{wp}(\mathcal{B}[p(\dots)\leftarrow\textbf{abort}],\phi)\\ \mathcal{G}^p_\phi(k+1)&\iff\mathcal{G}^p_\phi(0)\vee\varphi[\mathcal{X}\leftarrow\mathcal{G}^p_\phi(k)]\end{aligned}$$

Then the desired exact meaning is again

$$\exists k\in\mathbb{N}:\mathcal{G}^p_\phi(k)$$

but it is more involved than just stating and verifying the sufficient precondition we had in mind when designing p.

5.4 A Recursive Example: Quicksort

- Let us verify the well-known *quicksort* (Dromey, 1989, Chapter 9.5) sorting algorithm as another example of a recursive algorithm:

$$\begin{aligned}&\{\ \text{pre: } A=a\wedge\mathrm{lower}(a)\leq l\wedge u\leq\mathrm{upper}(a)\\ &\quad\text{post: }\mathit{perm}(A,a)\wedge\mathit{same}(A,a,l,u)\wedge\mathit{ordered}(a[l\dots u])\\ &\quad\text{bound: } u-l\ \}\\ &\textbf{proc}\ \ \mathit{quicksort}(\textbf{value}\ \ l,\ \ u\colon\ \mathbb{Z};\end{aligned}$$

```
                    value result a: array of τ);
if l < u →
    partition(l, u, a, m);
    quicksort(l, m − 1, a);
    quicksort(m + 1, u, a)
[] l ≥ u →
    skip
fi
```

Recall the use of the ghost variable A to retain the original value of the input-output parameter a.

- Here the formula $same(a, A, l, u)$ means "arrays a and A are the same outside the indices $l \ldots u$". This can be expressed as

$$\text{lower}(a) = \text{lower}(A) \wedge \text{upper}(a) = \text{upper}(A) \wedge \\ \forall\, \text{lower}(a) \le i \le \text{upper}(a) : i < l \vee i > u \Longrightarrow a[i] = A[i].$$

- It relies on the nonrecursive procedure

```
{ pre: B = b ∧ lower(b) ≤ p ∧ p < q ∧ q ≤ upper(b)
  post: perm(B, b) ∧ same(B, b, p, q) ∧
        p ≤ r ∧ r ≤ q ∧ b[p ... r] ≤ b[r ... q] }
proc partition(value p, q: ℤ;
               value result b: array of τ;
               result r : ℤ);
⋮
```

The key to quicksort *efficiency* is to realize that it can be done with only $\mathcal{O}(q - p)$ steps. But as far as its *correctness* goes, these pre- and postconditions suffice.

- Let us first verify the main call

$$quicksort(\text{lower}(x), \text{upper}(x), x)$$

to sort the given array x:

Precondition becomes

$$X = x \wedge \underbrace{\text{lower}(x) \le \text{lower}(x)}_{\text{TRUE}} \wedge \underbrace{\text{upper}(x) \le \text{upper}(x)}_{\text{TRUE}}.$$

Postcondition becomes

$$perm(X, x) \wedge \overbrace{same(X, x, \text{lower}(x), \text{upper}(x))}^{\text{lower}(X)=\text{lower}(x) \wedge \text{upper}(X)=\text{upper}(x)} \wedge \\ ordered(x).$$

Hence the **result**ing value of x is indeed an ordered permutation of its initial value X.

- Let us then verify the nonrecursive branch:

– The postcondition becomes

$$perm(A, a) \wedge \underbrace{same(A, a, l, u)}_{A = a \text{ and}} \wedge \underbrace{ordered(a[l \ldots u])}_{\text{at most one element}}$$

since $l \geq u$ by this **if** guard.

– It is implied by the $A = a$ already in the precondition.

– Hence this **skip** suffices.

- Verifying the recursive branch needs some way of talking about the current contents of array a at each step.

  Let us introduce two new ghost variables A_1 and A_2 for this purpose, where each A_j is array a just before the jth recursive *quicksort* call in the source code.

- Then we take suitable fresh copies of the procedure specifications using these ghost variables for array a.

- The specification copy for the *partition* call is obtained by the following textual replacements:

  $B \leftarrow A$ since the incoming **value** of array b is the incoming **value** of array a.

  $b \leftarrow A_1$ since the **result**ing value of array b becomes the incoming **value** of array a for the 1st recursive call.

  $p \leftarrow l$, $q \leftarrow u$ **and** $r \leftarrow m$ from the call itself.

  We get:

$$\begin{array}{l} \{\ \text{pre: } A = A_1 \wedge \text{lower}(A_1) \leq l \wedge l < u \wedge u \leq \text{upper}(A_1) \\ \quad \text{post: } perm(A, A_1) \wedge same(A, A_1, l, u) \wedge \\ \qquad l \leq m \wedge m \leq u \wedge A_1[l \ldots m] \leq A_1[m \ldots u]\ \} \end{array}$$

  – This precondition is ensured partly by the *quicksort* precondition and partly by the guard of this **if** branch.

  – Hence this call is permitted, and so we can later assume its postcondition.

  – Since we are now separating the different contents of array a from each other (via the ghost variables A, A_1 and A_2), we can *add* this postcondition into the already collected facts.

- That is, our verification of this branch has now proceeded as far as...

$$\begin{array}{l} \{\ \text{the } quicksort \text{ precondition}\ \} \\ \textbf{if}\ \ l < u \rightarrow \\ \quad \{\ \text{the } quicksort \text{ precondition} \wedge \text{the guard } l < u\ \} \\ \quad partition(l, u, a, m)\,; \\ \quad \{\ \ldots \text{here: the } quicksort \text{ precondition} \wedge \text{the guard } l < u \\ \qquad \wedge \text{this } partition \text{ postcondition}\ \} \\ \quad quicksort(l, m-1, a)\,; \\ \quad quicksort(m+1, u, a) \\ \quad \{\ \text{the } quicksort \text{ postcondition, our goal}\ \} \end{array}$$

- Reassigning into the same variable (here array a) can *falsify* formulas that were true before the assignment. Separating its successive values via ghost variables (here A, A_1 and A_2) allows us to keep them true instead, as they speak of its past values.

- The next step is the 1st *quicksort* call. We take a copy of its specification with textual replacements $A \leftarrow A_1$, $a \leftarrow A_2$, $l \leftarrow l$ (since l does not change in this call) and $u \leftarrow m-1$:

$$\begin{array}{l}\{\ \text{pre: } A_1 = A_2 \wedge \text{lower}(A_2) \leq l \wedge m-1 \leq \text{upper}(A_2)\\ \quad \text{post: } \mathit{perm}(A_1, A_2) \wedge \mathit{same}(A_1, A_2, l, m-1) \wedge\\ \qquad \mathit{ordered}(A_2[l \ldots m-1])\\ \quad \text{bound: } (m-1) - l\ \}\end{array}$$

  - The *quicksort* bound must be greater than zero and decrease:

$$\overbrace{u - l > 0}^{\text{the } \textbf{if} \text{ guard}} \wedge (m-1) - l < u - l \Leftrightarrow m - 1 < u \Leftrightarrow m \leq u$$

    which in turn appeared with the addition of the *partition* postcondition.
  - The other claims in this precondition can be seen from this $m \leq u$ and the *quicksort* precondition. Hence this call is permitted, and so we can add its postcondition too.

- The last step is the 2nd *quicksort* call. We take a copy of its specification with textual replacements $A \leftarrow A_2$, $a \leftarrow a$, $l \leftarrow m+1$ and $u \leftarrow u$:

$$\begin{array}{l}\{\ \text{pre: } A_2 = a \wedge \text{lower}(a) \leq m+1 \wedge u \leq \text{upper}(a)\\ \quad \text{post: } \mathit{perm}(A_2, a) \wedge \mathit{same}(A_2, a, m+1, u) \wedge\\ \qquad \mathit{ordered}(a[m+1 \ldots u])\\ \quad \text{bound: } u - (m+1)\ \}\end{array}$$

  The bound

$$\overbrace{u - l > 0}^{\text{the } \textbf{if} \text{ guard}} \wedge u - (m+1) < u - l \Leftrightarrow m + 1 > l \Leftrightarrow m \geq l$$

  and the other claims in this postcondition can be seen as in the preceding step. Hence this call is permitted too, and we can add its postcondition too.

- Let us now use these added conjuncts to verify the three conjuncts which form the *quicksort* postcondition:

  $\mathit{perm}(A, a)$ because we have added $\mathit{perm}(A, A_1)$, $\mathit{perm}(A_1, A_2)$ and $\mathit{perm}(A_2, a)$.

  $\mathit{same}(A, a, l, u)$ because we have added $\mathit{same}(A, A_1, l, u)$, $\mathit{same}(A_1, A_2, l, m-1)$ and $\mathit{same}(A_2, a, m+1, u)$.

  $\mathit{ordered}(a[l \ldots u])$ is a slightly more involved inference:
  - We have $\mathit{ordered}(a[l \ldots m-1])$ by $\mathit{ordered}(A_2[l \ldots m-1])$ and $\mathit{same}(A_2, a, m+1, u)$.
  - Similarly we also have $\mathit{ordered}(a[m+1 \ldots u])$ by the the 2nd *quicksort* call postcondition.
  - Next we note that $\mathit{perm}(A, a)$ and $\mathit{same}(A, a, l, u)$ together imply that $\mathit{perm}(A[l \ldots u], a[l \ldots u])$.
    (We could also have stated this directly in the *quicksort* postcondition at the expense of making the *perm* inference above slightly trickier.)

- In the 1st *quicksort* call this note gives $perm(A_1[l \ldots m-1], A_2[l \ldots m-1])$, from which $same(A_2, a, m+1, u)$ gives $perm(A_1[l \ldots m-1], a[l \ldots m-1])$.
- In the 2nd it gives $perm(A_2[m+1 \ldots u], a[m+1 \ldots u])$ and $same(A_1, A_2, l, m-1)$ gives $perm(A_1[m+1 \ldots u], a[m+1 \ldots u])$ from it.
- We also have $a[m] = A_1[m]$ by $same(A_1, A_2, l, m-1)$ and $same(A_2, a, m+1, u)$
- Hence we can appeal to the $A_1[l \ldots m] \leq A_1[m \ldots u]$ in the *partition* post-condition, which takes care of the element $A_1[m]$ gluing these two *ordered* parts together.

This completes our verification of this remaining *quicksort* recursive **if** branch, and hence of the entire algorithm. □

5.4.1 Separating Value and Result Parameters

- We verified our *quicksort* algorithm as if it had separate **result** parameters:

```
{ pre:  lower(a) ≤ l ∧ u ≤ upper(a)
  post: perm(c, a) ∧ same(c, a, l, u) ∧ ordered(c[l ... u])
  bound: u − l }
proc quicksort2 ( value l , u : ℤ ;
                  value  a : array of τ ;
                  result c : array of τ ) ;
if l < u →
    partition2(l, u, a, m, a1) ;
    quicksort2(l, m − 1, a1, a2) ;
    quicksort2(m + 1, u, a2, c)
 [] l ≥ u →
    skip
fi
```

Then these local code variables a_1 and a_2 concretize the ghost variables A_1 and A_2.

- The practical reason why we kept reassigning into array a instead of introducing these local arrays a_1 and a_2 was execution time and space optimization – eliminating unnecessary copying of unmodified array elements.

  (Except that the call-by-value return parameter passing convention of GCL copies anyway. . .)

- This original *quicksort* is an *in-place* sorting algorithm, and an efficient one at that.

- On the other hand, there was a design time cost:

  - The correctness proof introduced A_1 and A_2 not present in the code, so the proof and code diverged from each other.
  - Note also that they and their concrete counterparts a_1 and a_2 have specific explanations, as suggested in the Variable Introduction Principle (Principle 6), so their explicit introduction would also be justified.

  We should worry about correctness first, and about efficiency only if turns out to be necessary later.

- Using the same memory locations for both input and output parameters can also introduce subtle aliasing bugs:
  - Suppose we have a written a library routine

```
proc mystic(value x: array of τ;
            value result y: array of τ);
⋮
```

    which somehow incorporates into y the information in x.
  - Most practical programming languages (unlike GCL) would pass this y *by reference* so that it can be modified "on the fly" rather than at the end of *mystic*.
  - Suppose then that a user of our library calls $mystic(z, z)$. Is the outcome guaranteed to be correct also in this case? Or will some old input element $z[i]$ get overwritten by the corresponding new output element $z[i]$ before its old value is needed for computing some other output element $z[j]$?
  - The classic example is the low-level routine $copy(p, q, n)$ which copies n bytes from the area starting at address p into an area starting at address q. What happens if these two areas overlap?

Principle 9 (Separate Inputs and Outputs)**.** *Correctness reasoning becomes simpler if procedures allocate and construct their results themselves and "from scratch", instead of reassigning them into the parameters supplied by their callers.*

Hence procedures should be designed with separate input and output parameters. If performance dictates, then they can be later combined – at a risk: What if our procedure accidentally modifies something which the caller (or some other part of the whole program) expects to remain unmodified?

(In GCL design time terms, use separate ***value*** *and* ***result*** *parameters in your first version. If you really need some efficiency-related property such as in-place sorting, then revise your design to combine them – but remember the pitfalls!)*

5.4.2 Pivot Partitioning

- Let us now develop the missing *partition* procedure according to its stated specification.
- Let us choose very naïvely the first element $b[p]$ as our *pivot value* x around which to partition the input $b[p \ldots q]$.
  - A better choice would for x be e.g. the *median of three* values $b[p]$, $b[(p+q) \operatorname{div} 2]$ and $b[q]$, since it is more likely to find a pivot near the "middle" of the input values $b[p], b[p+1], b[p+2], \ldots, b[q]$ (Cormen et al., 2001, Chapter 7.2).
  - Or we could even allow the current pivot x to vary during the partition. This would lead into *interval* partitioning (Dromey, 1989, Chapter 9.5.2).
- Note also that the *partition* specification was intentionally just strong enough for *quicksort* correctness but no stronger: its postcondition just requires us to permute $b[p \ldots q]$ so that
  - $A[r]$ must be equal to our chosen x, and

– prefix $b[p \ldots r] \leq x$ and suffix $b[r \ldots q] \geq x$, but
– the elements equal to x can be in either of these two parts.

This gives us more flexibility is designing our procedure body.

- Let us start massaging its stated postcondition:

$$\overbrace{perm(B,b) \wedge same(B,b,p,q) \wedge p \leq r \wedge r \leq q \wedge}^{\text{We do not repeat this part below.}} \quad b[p \ldots r] \leq b[r \ldots q].$$

- Let us introduce new variables i and j such that

$b[p \ldots i]$ is the prefix $\leq x$
$b[j \ldots q]$ is the suffix $\geq x$
$b[i+1 \ldots j-1]$ is the still unprocessed part between them

this far:

$$b[p \ldots i] \leq x \wedge b[j \ldots q] \geq x \underbrace{\wedge x = b[r] \wedge i = r \wedge j = r}_{\text{Drop these to get the invariant.}}.$$

- Our procedure body gets the following form:

$x\,,\ i\,,\ j\ :=\ b[p]\,,\ p\,,\ q+1\,;$
{ inv: $b[p \ldots i] \leq x \wedge b[j \ldots q] \geq x$
bound: length of unprocessed part $b[i+1 \ldots j-1]$ }
do $i+1 \leq j-1 \rightarrow$
shorten the unprocessed part
od;
$r\,,\ b[p]\,,\ b[i]\ :=\ i\,,\ b[i]\,,\ b[p]$

Note that the loop initialization ensures that the prefix is nonempty. Hence the part after the loop can safely take its last index i as the **return**ed index r.

- One way to shorten the unprocessed part is by incrementing i.
  – Calculating $\text{wp}(\text{inv}, i := i+1)$ yields that $b[i+1] \leq x$ must hold for it to be allowed.
  – This $i+1$ is a valid index by the loop guard and $j-1 \leq q \leq \text{upper}(p)$. Here the first inequality comes from the initialization of j and not incrementing it during the loop. The second comes in turn from the *partition* precondition.

- Similarly we can also decrement j to get:

$x\,,\ i\,,\ j\ :=\ b[p]\,,\ p\,,\ q+1\,;$
{ inv: $b[p \ldots i] \leq x \wedge b[j \ldots q] \geq x$
bound: length of unprocessed part $b[i+1 \ldots j-1]$ }
do $i+1 \leq j-1 \rightarrow$
if $b[i+1] \leq x$
$i\ :=\ i+1$
[] $b[j-1] \geq x$
$j\ :=\ j-1$

```
        [] b[i+1] > x ∧ b[j−1] < x →
           How should we handle this missing case?
      fi
  od;
  r , b[p] , b[i] := i , b[i] , b[p]
```

- This missing third case reduces to the preceding two by swapping $b[i+1]$ and $b[j-1]$.

```
{ pre: B = b ∧ lower(b) ≤ p ∧ p < q ∧ q ≤ upper(b)
  post: perm(B, b) ∧ same(B, b, p, q) ∧
        p ≤ r ∧ r ≤ q ∧ b[p ... r] ≤ b[r ... q] }
proc partition(value p , q : ℤ;
               value result b : array of τ;
               result r : ℤ);
x , i , j := b[p] , p , q + 1;
{ inv: b[p ... i] ≤ x ∧ b[j ... q] ≥ x
  bound: length of uprocessed part b[i + 1 ... j − 1] }
do i + 1 ≤ j − 1 →
    if b[i + 1] ≤ x
       i := i + 1
     [] b[j − 1] ≥ x
       j := j − 1
     [] b[i + 1] > x ∧ b[j − 1] < x →
       i , j , b[i + 1] , b[j − 1] := i + 1 , j − 1 , b[j − 1] , b[i + 1]
    fi
od;
r , b[p] , b[i] := i , b[i] , b[p]
```

(Bentley and McIlroy (1993) provide a detailed account on engineering an efficient *partition*ing algorithm for *quicksort*ing. In particular, they show how the idea of loop invariants is used in practical algorithm design: without expressing them in formal logic, yet reasoning about them systematically.)

5.5 Iteration is Tail Recursion

- Note that the 2nd recursive *quicksort* call can be eliminated:

```
proc quicksort(value l , u : ℤ;
               value result a : array of τ);
if l < u →
    partition(l, u, a, m);
    quicksort(l, m − 1, a);
    quicksort(m + 1, u, a)
 [] l ≥ u → skip
fi
```

can be rewritten as

```
proc quicksort3(value l , u : ℤ;
                value result a : array of τ);
do l < u →
    partition(l, u, a, m);
```

```
        quicksort3(l, m − 1, a);
        l := m + 1
    od
```

- That is, we replaced

  the current parameter values with the new ones in the call — here only l changes, u and a stay as they were

  calling the procedure with repeating its body.

- This was possible because the eliminated call was the *last* thing *quicksort* would do before returning to its caller.

- Such calls are called *tail calls*.

  - A programming language implementation can perform a tail call without increasing its recursion stack depth.
  - It can namely pop the current activation record *before* pushing the activation record of this tail call.
  - Hence a tail call is an elegant "**goto** this procedure and report your results directly back to my caller, not me".

- If a recursive call is tail (like in here) then it is called *tail recursion*.

  - Tail recursion is the kind which permits elimination like this.
  - Or the converse view: loop constructs (such as **do**) are syntactic sugar for tail recursion, since that particular special case of general recursion is so common in practical programming.

    (In fact, some programming languages such as Scheme (Dybvig, 1996) do not even provide separate iteration constructs, just general recursion. Instead the standard specifies that the implementation must eliminate tail calls. Then a programmer specifies iteration as tail recursion. This makes the programming language definition more uniform, and iteration becomes just a space-saving optimization instead of a separate idea.)

- The general tail recursion elimination transformation is from

```
proc p(value x⃗; value result y⃗; result z⃗);
if guard_1 → B_1; p(a⃗_1, y⃗, z⃗)
 [] ⋮
 [] guard_m → B_m; p(a⃗_m, y⃗, z⃗)
 [] guard_{m+1} → B_{m+1} [] ... [] guard_n → B_n
fi
```

  into

```
proc p(value x⃗; value result y⃗; result z⃗);
do guard_1 → B_1; x⃗ := a⃗_1
 [] ⋮
 [] guard_m → B_m; x⃗ := a⃗_m
od;
if guard_{m+1} → B_{m+1} [] ... [] guard_n → B_n fi
```

- That is, if a recursive call
  - is the last thing that this current branch performs before returning back to its caller
  - reports its **result**s $\vec{y}$ and $\vec{z}$ back to the caller unmodified

  then it can be replaced with updating the **value** parameters $\vec{x}$ into new values $\vec{a_i}$ and looping.

  In *quicksort3* we updated l into $m+1$ before looping.

- The other branches are retained after the **do** loop.

  In *quicksort3* this part was

  if $l \geq u \rightarrow$ **skip** **fi**

  which could be further discarded since its guard is guaranteed to be TRUE because it is the opposite of the **do** loop guard.

- Because

  $$\text{wp}(\text{the original } p, \phi)$$

  must ensure ϕ in every possible recursion call order, it must ensure ϕ especially in the orders where a tail recursive branch is chosen whenever possible. Hence

  $$\text{wp}(\text{the original } p, \phi) \Rightarrow \text{wp}(\text{the transformed } p, \phi)$$

  so we can indeed use the transformed p instead wherever the original p is used. But the opposite

  $$\text{wp}(\text{the original } p, \phi) \Leftarrow \text{wp}(\text{the transformed } p, \phi)$$

  is *not* necessarily true: the transformed p may permit more inputs than the original p, namely those for which ϕ is ensured in the "tail recursion first" order but not in general.

- The opposite transformation to introduce tail recursion turns the loop

  do $guard_1 \rightarrow \mathcal{B}_1$
  [] $\vdots$
  [] $guard_n \rightarrow \mathcal{B}_n$
  od

  into the call $q(\vec{u})$, where $\vec{u}$ consists of all the variable names appearing in the loop and

  proc q(**value result** $\vec{u}$);
  if $guard_1 \rightarrow \mathcal{B}_1; q(\vec{u})$
  [] $\vdots$
  [] $guard_n \rightarrow \mathcal{B}_n; q(\vec{u})$
  [] $\neg(guard_1 \vee \ldots \vee guard_n) \rightarrow$ **skip**
  fi

is the definition for the corresponding new tail recursive procedure.

- This connection between iteration and tail recursion explains the similarities between designing and verifying loops and recursive procedures:
  - Both have *bound*ing expressions. In a **do** loop it bounds the number of still remaining iterations, in a recursive procedure the still allowed recursion call depth.

    For tail recursion, these two notions coincide. This explains the former in terms of the latter.
  - Their designs aim at decreasing their *bound*ing expressions to guarantee termination. Again, tail recursion shows that the idea is the same in both.
  - The loop invariant and the precondition of a recursive procedure have the same role: both specify what must hold before each loop round or call.
  - The loop guards and the guards for the nonrecursive branches have also similar roles: For a loop their negation must imply the postcondition together with its invariant. Similarly, a guard of a nonrecursive branch must imply the recursion postcondition together with its precondition.

5.5.1 Exponentiation Recursively

- Designing a recursive procedure can often be *simpler* than a corresponding loop – which may come as a surprise, since usually loops are taught first, and recursion only later, and then as a seemingly complicated, esoteric and scary technique rarely needed and best avoided. . . !
- Let us redo our earlier fast exponentiation algorithm (Figure 10) from this viewpoint.
- That is, we start designing a recursive procedure

  { pre: the type for p
    post: $y = x^p$ }
  proc *pow*(**value** $x : \mathbb{R}$; **value** $p : \mathbb{N}$; **result** $y : \mathbb{R}$);
  $\mathcal{B}$

  Note that we do not need the ghost variables X and P here, since x and p are **value** parameters.
- Next we must choose a *bound* for our procedure, so that we can start developing its body $\mathcal{B}$ to decrease it.
- Parameters of type $\mathbb{N}$ are natural candidates, since they have a value 0 where the recursion must stop, since they have no smaller values.
- Here we can try p:

  { pre: the type for p
    post: $y = x^p$
    bound: p }
  proc *pow*(**value** $x : \mathbb{R}$; **value** $p : \mathbb{N}$; **result** $y : \mathbb{R}$);
  if $p = 0 \rightarrow$
      terminating branch
  [] $p > 0 \rightarrow$

recursive branch
fi

- We could continue with these two branches, and come up with a correct algorithm. Let us however split the recursive branch further into the odd and even branches as in our earlier algorithm:

  { pre: the type for p
    post: $y = x^p$
    bound: p }
  proc *pow* (**value** $x : \mathbb{R}$; **value** $p : \mathbb{N}$; **result** $y : \mathbb{R}$) ;
  if $p = 0 \rightarrow$
      $y := 1$ as the terminating branch
  [] $\neg\,\text{even}(p) \rightarrow$
      odd recursive branch
  [] $\text{even}(p) \wedge (p > 0) \rightarrow$
      even recursive branch
  fi

  The terminating branch is dictated by the postcondition: $x^0 = 1$.

- We develop the two recursive branches as follows:

  { pre: the type for p
    post: $y = x^p$
    bound: p }
  proc *pow* (**value** $x : \mathbb{R}$; **value** $p : \mathbb{N}$; **result** $y : \mathbb{R}$) ;
  if $p = 0 \rightarrow$
      $y := 1$
  [] $\neg\,\text{even}(p) \rightarrow$
      $pow(x, p - 1, z)$;
      $y := z \cdot x$
  [] $\text{even}(p) \wedge (p > 0) \rightarrow$
      $pow(x \cdot x, p \operatorname{div} 2, z)$
  fi

  Note that verifying these branches is straightforward. E.g. in the first branch

  - the bound is $p > 0$ by the guard and decreases to $p - 1$, so the recursive call is permitted, and
  - its postcondition is $z = x^{p-1}$, and hence the assignment makes $y = x^p$ as required.

5.5.2 Exponentiation Tail-Recursively

- However, the branch

  [] $\neg\,\text{even}(p) \rightarrow$
      $pow(x, p - 1, z)$;
      $y := z \cdot x$

  of our procedure *pow* is not tail recursive, and hence it is not quite the same as a **do** loop.

- The intuition of this assignment is "the intermediate **result** z obtained from recursion must be further multiplied by the current x to produce a valid final **result** y from this call".

- This suggests adding a *new* parameter a = "the value with which I want you to multiply your answer, so that I don't have to do it myself after calling you".

  Compare it to a in our original **do** loop (Figure 10).

- This gives:

  { pre: the type for p
    post: $y = x^p \cdot a$
    bound: p }
  proc *pow2* (**value** $x : \mathbb{R}$; **value** $p : \mathbb{N}$; **result** $y : \mathbb{R}$;
  value $a : \mathbb{R}$) ;
  if $p = 0 \rightarrow$
  y := new terminating value
  [] $\neg$ even(p) $\rightarrow$
  new odd recursive branch
  [] even(p) $\wedge$ ($p > 0$) $\rightarrow$
  new even recursive branch
  fi

  Here we are using the idea that modifying existing code is done by changing its specification and redeveloping to get it right.