- Again the new terminating value is dictated by the new postcondition: $x^0 \cdot a = 1 \cdot a = a$.
- The new even recursive branch passes a along unmodified, since it was already a tail call:

  $$[\!] \ \text{even}(p) \wedge (p > 0) \rightarrow \\ \mathit{pow2}(x \cdot x, p \operatorname{div} 2, y, a)$$

  Again this is easily verified:

  – The bound p starts out > 0 and decreases into $p \operatorname{div} 2$ by the guard, so this call is permitted.

  – Its postcondition is

  $$\begin{aligned} y &= (x \cdot x)^{(p \operatorname{div} 2)} \cdot a \\ &= x^p \cdot a \end{aligned}$$

  also by the guard.
- Getting the new odd recursive branch to be a tail call too is why a was added:

  $$[\!] \ \neg \text{even}(p) \rightarrow \\ \mathit{pow2}(x, p - 1, y, u)$$

  Let us verify it to determine the unknown expression u for the new value of a:

  – The bound p starts out > 0 and decreases into $p - 1$ by the guard, so this call is permitted.

  – Its postcondition is

  $$y = x^{(p-1)} \cdot u$$

  but we want it to be

  $$= x^p \cdot a.$$

  We get what we want by solving

  $$u = x \cdot a$$

  from these two equations.
- Putting everything together gives our tail recursive procedure:

```
{ pre: the type for p
  post: y = x^p · a
  bound: p }
proc pow2(value x:ℝ; value p:ℤ; result y:ℝ;
          value a:ℝ);
if p = 0 →
    y := c
[] ¬even(p) →
    pow2(x, p − 1, y, x · a)
[] even(p) ∧ (p > 0) →
    pow2(x · x, p div 2, y, a)
fi
```

- Eliminating its tail recursive calls gives an iterative procedure:

```
{ pre: the type for p
  post: y = x^p · a
  bound: p }
proc pow3(value x:ℝ; value p:ℤ; result y:ℝ;
          value a:ℝ);
do ¬even(p) →
     x, p, a := x, p − 1, x · a
 [] even(p) ∧ (p > 0) →
     x, p, a := x · x, p div 2, a
od;
if p = 0 →
     y := a
fi
```

- This iterative procedure can be further simplified by removing redundant assignments and noting that the **if** guard is always TRUE after the **do** loop:

```
{ pre: the type for p
  post: y = x^p · a
  bound: p }
proc pow3(value x:ℝ; value p:ℤ; result y:ℝ;
          value a:ℝ);
do ¬even(p) →
     p, a := p − 1, x · a
 [] even(p) ∧ (p > 0) →
     x, p := x · x, p div 2
od;
y := a
```

  Now we are very close to our original loop (Figure 10).

- We still need an initial value for our new parameter a such that *pow* and *pow3* produce the same output y for the same inputs x and p. As for u, we can again calculate from their postconditions:

$$\overbrace{x^p \cdot a}^{pow3} = \overbrace{x^p}^{pow}$$
$$a = 1.$$

- Expanding $pow3(x, p, y, 1)$ into its body gives our final loop:

```
a := 1;
do ¬even(p) →
     p, a := p − 1, x · a
 [] even(p) ∧ (p > 0) →
     x, p := x · x, p div 2
od;
y := a
```

- This idea of achieving tail recursion by adding an extra parameter a for the work that

was done when returning from the recursion

can be done when entering it

is called adding an *accumulator* – for accumulating the result as we go on, rather than when we return.

- The idea is applicable when this work is associative, but not in general
- Multiplication is associative:

$$\overbrace{z_0 \cdot (z_1 \cdot (z_2 \cdot (z_3 \cdot \ldots)))}^{\text{done when returning}} = \underbrace{(((z_0 \cdot z_1) \cdot z_2) \cdot z_3) \cdot \ldots}_{\text{done when entering}}.$$

- By such tricks, we can

  develop our initial code recursively, which is natural in many problems — especially inductively defined problems

  transform it later into iterative code, if possible and needed for saving space (and time).

5.6 Well-Founded Bounds

(Some of this material comes from Gries (1981, Chapter 17), but most of it is general mathematics.)

- *Bound*ing the number of still allowed rounds is a natural way to ensure termination of **do** loops, but *bound*ing similarly the allowed recursion depth seems less natural.
- We can in fact show that a given recursive procedure terminates by a more general way than this "counting", but this way requires us to (re)visit what *induction in general* means – as it is a mathematical counterpart to general recursion.
- Let S be an enumerable (= "no bigger than $\mathbb{N}$") set.
- A binary relation $R \subseteq S \times S$ is a *strict partial ordering* of S if it is

  transitive: If both $R(x, y)$ and $R(y, z)$ hold then also $R(x, z)$ holds.

  asymmetric: If $R(x, y)$ holds then $R(y, x)$ cannot hold.

- The ordering relation R can also be

  total: for all distinct $x, y \in S$ we have either $R(x, y)$ or $R(y, x)$

  but it does not have to be.

- Examples:

  $<$ of $S = \mathbb{N}$ is

  transitive: If $p < q < r$ then $p < r$.

  asymmetric: If $p < q$ then $q < p$ is not possible.

  total: We always have exactly one of $p = q$, $p < q$ or $q < p$.

  so it is a strict total ordering.

$\subsetneq$ of $S = \mathcal{P}(\mathbb{N})$, the set of all subsets of $\mathbb{N}$, is on the other hand

transitive: If $A \subsetneq B \subsetneq C$ then $A \subsetneq C$.

asymmetric: If $A \subsetneq B$ then $B \subsetneq A$ is not possible.

partial: Neither $\{1, 2\}$ nor $\{2, 3\}$ is a strict subset of the other.

- A strict (partial or total) ordering relation R of S is *well-founded* if it does not have infinitely long descending chains: No infinite collection of elements $x_0, x_2, x_3, \ldots \in S$ such that
$$\ldots, R(x_3, x_2), R(x_2, x_1), R(x_1, x_0).$$
  - The usual order '$<$' of $S = \mathbb{N}$ is well-founded: any chain
$$n_m < \ldots < n_3 < n_2 < n_1 < n_0$$
cannot descend further if $n_m = 0$.
  - But the usual order '$<$' of $S = \mathbb{Z}$ is *not:* we can always extend any chain
$$z_m < \ldots < z_3 < z_2 < z_1 < z_0$$
of $m + 1$ integers into a longer chain with, say, $z_m - 1 < z_m$.
  - The strict subset relation '$\subsetneq$' of $S = \mathcal{P}(\mathbb{N})$ is again well-founded, since a chain cannot descend below $\emptyset$.

- Sometimes a set is called well-founded, if its usual order is well-founded. Hence in particular $\mathbb{N}$ is well-founded.

- Recall then how some claim $\Phi(n)$ is shown to be true for all natural numbers $n \in \mathbb{N}$ by induction:

  Base case is proving $\Phi(0)$ directly.

  Inductive case is proving $\Phi(n + 1)$ assuming $\Phi(n)$.

- Note that $n < n + 1$ is one link in the "natural" chain of $\mathbb{N}$:
$$0 < 1 < 2 < \ldots < n < n + 1 < \ldots.$$

  Base case proves Φ for the bottom 0 of the chain directly.

  Inductive case proves that Φ stays TRUE when we walk up the links of the chain.

- The general form of induction is the same:

  Background is the set S and its well-founded strict (partial or total) ordering R.

  Base case is to prove $\Phi(z)$ for all those $z \in S$ which are the low endpoints of the chains.

  Inductive case is to prove that for each link $y < x$ of a chain we have $\Phi(x)$ assuming $\Phi(y)$.

- Our *bound*ing expressions for recursion (and iteration as its special tail recursive kind) have been a special case of this general scheme:
  - The set S consists of all the inputs on which the recursive procedure p must work.

- If $bound(z) \leq 0$, then $p(z)$ cannot use recursive calls for this input $z \in S$.
- On the other hand, if

$$0 < bound(x) \tag{62}$$

for an input $x \in S$, then $p(x)$ can use recursive calls, but only into inputs $y \in S$ such that

$$bound(y) < bound(x). \tag{63}$$

- The corresponding (unusual!) well-founded partial ordering $R(y, x)$ of $\mathbb{Z}$ is "Eqs. (62) and (63)". Our proof that p terminates on all the inputs in S is in fact general induction for it.

- Ackermann's function A mentioned previously (Section 2.4.7) was an example whose termination could not be shown using just a *bound*ing expression.
  - Its inputs are pairs $\langle m, n \rangle \in \mathbb{N} \times \mathbb{N}$.
  - The partial well-founded ordering for showing its termination is

$$R(\langle m, n \rangle, \langle p, q \rangle) : \text{"either } m < p \text{ or } m = p \text{ but } n < q\text{"}. \tag{64}$$

  - If we had a fixed upper bound N limiting the second parameter to $0 \leq n \leq N$, then we could write a simple expression

$$bound(\langle m, n \rangle) = (N + 1) \cdot m + n \tag{65}$$

for this R...

  - ...but we do not. Instead, N depends on A itself, and we have no such *bound*ing expression without mentioning A itself. But this would lead into a circular "termination proof".

- Eq. (64) shows that a correct termination proof can be stated in terms of "either variable m is decremented or it stays the same, but variable n is decremented".

  Knowing about well-foundedness allows us to recognize and utilize such nonnumerical termination proofs.

- As an example, let us revisit the *string matching* problem discussed already in the exercises (Exercise 2 of week 5).

- We state the problem here as follows:

  Input: Two arrays, namely the corpus c and the target t.

  To simplify the expressions, let us assume that $\text{lower}(t) = 0$.

  Output: The target t is said to have a *match* starting at index $\text{lower}(c) \leq p \leq \text{upper}(c) - \text{upper}(t)$ of the corpus c if the section $c[p \ldots p + \text{upper}(t)]$ starting at p equals t. In logic, $match(p)$ is

$$\forall 0 \leq i \leq \text{upper}(t) : c[p + i] = t[i]. \tag{66}$$

We are asked to report a starting index p of a match if one exists.

- Now we can state our desired postcondition as

$$(\exists\, \mathrm{lower}(c) \le q \le \mathrm{upper}(c) - \mathrm{upper}(t) : \mathit{match}(q)) \implies \mathit{match}(p)$$

or "if there are *match*es q, then I am reporting one of them as p". Note how it takes into account the possibility of not having any matches: then p can have any value whatsoever.

- It is logically equivalent to

$$(\forall\, \mathrm{lower}(c) \le q \le \mathrm{upper}(c) - \mathrm{upper}(t) : \neg\, \mathit{match}(q)) \lor \mathit{match}(p).$$

- Let us now split these two cases further to "either there are no matches or this p is the first of them":

$$(\forall\, \mathrm{lower}(c) \le q \le \mathrm{upper}(c) - \mathrm{upper}(t) : \neg\, \mathit{match}(q)) \lor (\forall\, \mathrm{lower}(c) \le q < p : \neg\, \mathit{match}(q)) \land \mathit{match}(p).$$

- Another way to differentiate between these two cases is

$$(\forall\, \mathrm{lower}(c) \le q < p : \neg\, \mathit{match}(q)) \land (\underbrace{p = \mathrm{upper}(c) - \mathrm{upper}(t) + 1}_{\text{No matches.}}\, \mathbf{cor}\ \mathit{match}(p)).$$

- Let us then add another free variable k into $\mathit{match}(p)$ with the meaning "the first k characters $t[0 \ldots k-1]$ of the target match the corpus $c[p \ldots p+k-1]$ starting at p". Logically we arrive at our final postcondition

$$(\forall\, \mathrm{lower}(c) \le q < p : \neg\, \mathit{match}(q)) \land (p = \mathrm{upper}(c) - \mathrm{upper}(t) + 1\, \mathbf{cor}\ (k = \mathrm{upper}(t) + 1 \land \forall 0 \le i < k : c[p+i] = t[i])). \quad (67)$$

- Now we allow both free variables to vary within their respective ranges:

$$\mathrm{lower}(c) \le p \le \mathrm{upper}(c) - \mathrm{upper}(t) + 1$$
$$0 \le k \le \mathrm{upper}(t) + 1.$$

We take as our *bound* "either p advances towards the end of its range or it stays the same but k advances towards the end of its range" in the spirit of Eq. (64).

- Athough we could have written an explicit expression like

$$\mathit{bound}(p,k) = (\mathrm{upper}(t) + 2) \cdot (\mathrm{upper}(c) - \mathrm{upper}(t) + 1 - p) + (\mathrm{upper}(t) + 1 - k) \quad (68)$$

following Eq. (65) in this problem, the verbal well-ordering formulation is more suggestive for what to do next.

- Moreover, our problem might instead have specified that the corpus c shall be read from a file f character by character. Then index p would not have been an explicit number but some abstract entity like "the current reading position of file f".

  Then an equation like (68) would be less intuitive than stating "either the next character is read from file f or..." as the measure of progress towards termination.

  (In this case, file f would be read character by character into a ring buffer with the same capacity as the length $\text{upper}(t) + 1$ of the targer t.)

- The development of the matching loop is standard:

  - We select an initialization to get rid of the two quantified parts "$\forall \ldots$" in our final postcondition (67).
  - Our loop invariant is then their conjunction.
  - Our loop guard is similarly the negation of what remains after they have been set to TRUE.
  - Now the two cases of our verbal bound can be directly encoded in the loop body, whereas Eq. (68) would not have suggested them directly.

$$
\begin{array}{l}
\{\ \text{inv: } (\forall\, \text{lower}(c) \le q < p : \neg\, match(q)) \wedge \\
\qquad\quad (\forall 0 \le i < k : c[p+i] = t[i]) \\
\quad \text{bound: either } p \text{ advances or stays put but } k \text{ advances}\ \} \\
p\,,\ k\ :=\ \text{lower}(c)\,,\ 0\,; \\
\textbf{do}\ \ p < \text{upper}(c) - \text{upper}(t) + 1 \wedge k < \text{upper}(t) + 1 \rightarrow \\
\qquad \textbf{if}\ \ \text{we can advance } k \rightarrow \\
\qquad\qquad k\ :=\ k + 1 \\
\qquad\ [\!]\ \ \text{we can advance } p \rightarrow \\
\qquad\qquad p\,,\ k\ :=\ p + 1\,,\ \text{we might have to change } k \\
\qquad \textbf{fi} \\
\textbf{od}
\end{array}
$$

- Then the calculation

$$\text{wp}(k := k + 1, \text{inv}) \quad = \quad \text{inv} \wedge c[p+k] = t[k] \tag{69}$$

  reveals the *guard* needed for advancing k.

- Similarly the calculation

$$\text{wp}((p, k := p + 1, e), \text{inv}) \quad = \ldots = \underbrace{\neg\, match(p)}_{\text{Easy}} \wedge \underbrace{(\forall 0 \le i < e : c[(p+1)+i] = t[i])}_{\text{Hard}}$$

  reveals the condition for advancing p, where e is the possible new value needed for k.

  Easy part can be ensured by the complement of guard (69).

  Hard part has the troublesome '$\forall$'. Fortunately we can make it TRUE by setting $e = 0$.

- We also recall that p is permitted to have any value whatsoever, if there are no *match*es. Hence k is a more reliable answer indicator.

```
proc matching(value c,t:array of τ;result p:ℤ);
{ inv: (∀ lower(c) ≤ q < p : ¬ match(q))∧
       (∀0 ≤ i < k : c[p+i] = t[i])
  bound: either p advances or stays put but k advances }
p, k := lower(c), 0;
do p < upper(c) − upper(t) + 1 ∧ k < upper(t) + 1 →
    if c[p+k] = t[k]
       k := k + 1
    [] c[p+k] ≠ t[k]
       p, k := p + 1, 0
    fi
od;
if k = upper(t) + 1 →
    we have a match starting at position p
 [] k < upper(t) + 1 →
    the target is not in the corpus
fi
```

Figure 16: The String Matching Procedure.

Principle 10 (Well-Founded Bounds)**.** *Express the bound first in words, then formalize it enough to see that it is induction in some well-founded ordering of the inputs. A numeric bounding expression is just one (albeit common and familiar) of the possible ways to see it.*

- This principle is especially useful when dealing with *recursively/inductively* defined data types.
  - E.g. the binary search tree (BST) is defined in algorithmics and data structures books as follows:

    Base case is the *empty* tree.

    Inductive case is a record/structure/object/... which represents the root *node*.

    It consists of the *key* value to compare and inductively of the *left* and *right* *sub*trees containing the smaller and larger *key*s.
  - Then can write the query "Does BST t contain the key k" recursively following its inductive definition (Figure 17).
  - It is "clear" that *found* terminates, because it recurses into either of the two subtrees $t.\,left$ or $t.\,right$ of the node t, and these are *proper subparts of* BST t, and hence strictly smaller than it.
  - This "proper-subparts-of" relation is in fact a well-founded ordering R_{BST} of the set S of all finite BSTs, whose

    bottom element is the empty tree, and

    links are $R_{\mathrm{BST}}(u.\,left, u)$ and $R_{\mathrm{BST}}(u.\,right, u)$ for every node u.

    This *structural induction* is the mathematically solid foundation underlying our "clear" termination proof.
  - But note that inadequate *programming* languages and habits can undermine it:

```
proc found(value t:BST; value k:key;
           result f:𝔹);
if t is the empty tree →
   f := FALSE
[] t is a node ∧ k = t.key →
   f := TRUE
[] t is a node ∧ k < t.key →
   found(t.left, k, f)
[] t is a node ∧ k > t.key →
   found(t.right, k, f)
fi
```

Figure 17: Finding a Key in a Binary Search Tree.

  - Suppose that the *left* and *right* subtrees are represented as pointers to other nodes, as is commonly done.
  - Suppose also that these pointer can be reassigned, as is common.
  - Then what would stop us from creating a node v such that its $v.\,right$ points back to v itself?
  - Then $R_{\mathrm{BST}}(v.\,right, v)$ fails, because $v.\,right$ is not a proper subpart of v!
  - And also $found(v, k)$ might loop forever.
  - Mathematically, v is a *conceptually infinite* tree.

- Reasoning about conceptually infinite data structures can be carried out in *co*induction, another general proof principle. It can also handle perpetual processes, since they can be seen as *co*recursive computations over infinite data. But we do not go there.

6 On User-Defined Types

This material is adapted from Morgan (1994, Chapter 15). Similar constructs are available in some programming languages such as Standard ML (Milner et al., 1997) and Haskell (Peyton Jones, 2003).

- Let us now add to GCL tools for defining new data types such as BSTs (Section 5.6).
- We declare a new type with

  ```
  type Typename
       = TAG1
       | ⋮
       | TAGm
  ```

  where these different TAGs allow us to tell which kind of value we are currently dealing with.

- A simple example would be

  ```
  type Boolean
       = FALSE
       | TRUE
  ```

which could be used to define $\mathbb{B}$ if we did not have it already.

- We can also declare TAGs to have *field*s:

  $$\mid \ \textsc{Tag}_i \ \ \mathit{fieldname}_{i,1} : \mathit{type}_{i,1} \ \ \ldots \ \ \mathit{fieldname}_{i,n_i} : \mathit{type}_{i,n_i}$$

  - The idea is that if a value v (whose type is *Typename*) has the tag $\textsc{Tag}_i$, then this value v has also each field $v.\mathit{fieldname}_{i,j}$ (whose type is $\mathit{type}_{i,j}$).
  - But if v has some *other* tag $\textsc{Tag}_k \neq \textsc{Tag}_i$ instead, then these fields do *not* exist, and trying to access them **abort**s!

    Instead, v then has the fields $v.\mathit{fieldname}_{k,j}$ declared for $\textsc{Tag}_k$.
  - These fields and tags are *read-only* to avoid the previously discussed problems with aliasing conceptually infinite values.
  - All this could be made precise similarly to our treatment of arrays (Section 2.4.8) but we skip the details.

- E.g.

  type *MaybeInt*
  $= \textsc{Just} \ \ \mathit{here} : \mathbb{Z}$
  $\mid \textsc{Nothing}$

  declares the type *MaybeInt* for "either some integer or nothing at all".

- The new GCL *expression*

  $$\textsc{Tag}_i \ \ \mathit{expression}_{i,1} \ \ \ldots \ \ \mathit{expression}_{i,n_i}$$

  constructs a new value v of type *Typename* whose tag is $\textsc{Tag}_i$ and each field $v.\mathit{fieldname}_{i,j}$ has the value of $\mathit{expression}_{i,j}$ (whose type must therefore be $\mathit{type}_{i,1}$).

- Conversely, the new GCL *statement*

  if x **is**
  $\textsc{Tag}_1 \rightarrow$
  the branch where each $x.\mathit{fieldname}_{1,j}$ can be used
  [] $\vdots$
  [] $\textsc{Tag}_m \rightarrow$
  the branch where each $x.\mathit{fieldname}_{m,j}$ can be used
  fi

  deconstructs the current value v in the variable x of type *Typename* based on its TAG. Similarly for **do** as repeated **if**.

- Using these $x.\mathit{fieldname}_{i,j}$ cannot **abort** since "the GCL compiler" can check that each branch mentions only the correspoding $x.\mathit{fieldname}$s. This improves correctness.

- We also add these "x **is** $\textsc{Tag}_i$" as another atomic formula for our specifications, so that we can test TAGs there as well.

- Then e.g. the string matching procedure (Figure 16) can be written as

```
{ pre: ...
  post: r is JUST cand match(r.here)∨
        r is NOTHING cand
        ∀ lower(c) ≤ q ≤ upper(c) − upper(t) : ¬ match(q) }
proc matching2 ( value c,t : array of τ ;
                 result r : MaybeInt ) ;
do ... od ;
if k = upper(t) + 1 →
   r := JUST p
[] k < upper(t) + 1 →
   r := NOTHING
fi
```

where the new postcondition states more explicitly that "either r is JUST, in which case $r.\,here$ is a *match*, or r is NOTHING, in which case there were no *match*es".

- Then its caller must explicitly react to these two cases:

```
matching2(d, u, s) ;
if s is
   JUST →
   handle the match starting at position s.here
[] NOTHING →
   handle the case without any matches
fi
```

  - The caller must first explicitly test that there was indeed a match before it can get its starting position $s.\,here$.
  - Conversely, the caller cannot process a fake "starting position" when there were no matches, because $s.\,here$ is not available in that branch.
  - This is much more type-safe than just giving an index p as the **result**, and relying on the caller to test whether the index it received is a valid starting position or not – which the caller might forget to do!

- We can also allow *inductive* types such as lists and trees by allowing the *Typename* being declared to appear also within its declaration: E.g.

```
type BST
     = EMPTY
     | NODE left : BST  key : ℤ  right : BST
```

declares a binary search tree (BST) with integer *key*s.

- Note that this declaration matches the textbook structure of the BST – only the relation between its *key*s is missing.

- The corresponding query (Figure 17) can also be expressed directly:

```
proc found ( value t : BST ; value k : key ;
             result f : 𝔹 ) ;
if t is
   EMPTY →
```

```
        f := FALSE
    [] NODE →
        if k = t.key →
            f := TRUE
         [] k < t.key →
            found(t.left, k, f)
         [] k > t.key →
            found(t.right, k, f)
        fi
fi
```

Its termination is straightforward to see by structural induction, since the recursion is on $t.\mathit{left}$ and $t.\mathit{right}$.