LECTURE 4:

LOGIC-DRIVEN PROGRAM DEVELOPMENT

Annotated programs revisited

- Annotated programs do not deal with Termination.
- Semi-formal fix: associate with each loop a Variant,
  = a term V such that
  - for some k we have $V \geq k$
    (for all values of the variables in V)
  - V decreases with execution of the loop's body.
- This is restrictive (only poly-time programs would result).
  More generally:
  - There is a well-ordering $\succ$ such that V is decreases w.r.t. $\succ$ with each execution of the loop's body.

- A useful notation: Use Global pre-conditions: formulas $\xi[\vec{z}]$ where $\vec{z}$ are not assigned to in the program.
- Instance of the Consequence Rule have access to Globals:
  $\{\varphi\}\{\psi\}$ provided $\Gamma, \xi \vdash \varphi \rightarrow \psi$

Structural problem solving

- Some sound problem solving strategies are straightforward,
  often driven by the problem structure.

- Sequential decomposition.
  Program construct reflecting decomposition: Composition.

- Example of sequential decomposition: Merge sorted array S and unsorted array U into a sorted list.
  - Partial effect of the task: blended in the merged list will be U sorted.
  - Perform that partial effect first: sort U.
  - Now merging is easy.

- Case decomposition.
  Problem splits into cases, with different solutions.
  Corresponding program construct: Branching.

- Example of case decomposition: Integer square root.
  - Task: Calculate $\sqrt{a^k}$.
  - For k even, $= 2p$, $\sqrt{a^k} = a^p$
  - For k odd, $= 2p + 1$, $\sqrt{a^k} = a^p \cdot \sqrt{a}$

Loop development: the conceptual part

- Deveoping iterative solutions is often trickier.

Conceptual (= preliminary) steps in developing a loop:

1. State the post-condition (= specification)
   " Programming is a goal-oriented activity" (Gries).
2. Identify quantities that change as the task is gradually attained.
   - Think of process with "Snapshots".
   - Changing quantity often represented by
     a single variable appearing in the post-condition
   - But might be more complex,
     e.g. sub-arrays or sub-graphs
3. Identify an invariant:
   - The salient properties remaining constant during the process.
   - Is often is more general than the post-condition.
   - Often needs auxiliary quantities,
     balancing off the change in the principal object.
4. Identify the variant:
   As iterations are performed, what measures progress towards termination?

Loop development: the programming part

5. Initialize the variables in the invariant,
   to guarantee that the invariant is true initially.
6. Develop the guard.
   Two complementary roles:
   (a) Detects successful termination of the loop;
   (b) Invariant + guard's failure implies post-condition.
7. Develop the loop's body, so as to maintain the invariant (and reduce the variant)

On the use of invariants

- The Coins on the Table game
- The Closing the polygon game

Redoing an obvious example: Array summation

- Given array $a[0..n-1]$ of integers.
- Find $s = \sum_{i<n} a[i]$ (= post-condition)

Conceptual part:

- Varying quantity: partial sum $s_k = \sum_{i<k} a[i]$
- Invariant: $s = \sum_{i<k} a[i]$.
  (Note auxiliary quantity k.)
- Variant $= n - k$

Programming part:

- Initialize s and k: $s, k := 0, 0$
  (satisfies the invariant)
- Guard $= k \neq n$:
  1. This places a bound on the variant, AND
  2. guard's failure guarantees the post-condition.
- To maintain the invariant, while reducing the variant, the body of the loop must be

$$s, k := \ s + a[k],\ k + 1$$

In real time: Problem setup

Global: $a[0..k)$ array of integers

```
while      do



od
```

$\{s = \sum_{i<n} a[i]\}$

Defining the invariant

Global: $a[0..k)$ array of integers

$\boxed{\text{Invariant: } \{s = \sum_{i<k} a[i]\}}$

```
while       do




od
```

$\{s = \sum_{i<n} a[i]\}$

Defining the Variant

Global: $a[0..k)$ array of integers

Invariant: $\{s = \sum_{i<k} a[i]\}$
$\boxed{\text{Variant:}}$ $n - k$
```
while       do



od
```

$\{s = \sum_{i<n} a[i]\}$

Initializing

Global: $a[0..k)$ array of integers

$\boxed{s, k := 0, 0}$

Invariant: $\{s = \sum_{i<k} a[i]\}$

Variant: $n - k$

```
while        do




od
```

$\{s = \sum_{i<n} a[i]\}$

Defining the guard

Global: $a[0..k)$ array of integers

$s, k := 0, 0$

Invariant: $\{s = \sum_{i<k} a[i]\}$

Variant: $n - k$

`while` $\boxed{k \neq n}$ `do`

`od`

$\boxed{\{s = \sum_{i<k} a[i] \wedge \neg(k \neq n)\}}$

$\{s = \sum_{i<n} a[i]\}$

The loop body

Global: $a[0..k)$ array of integers

$s, k := 0, 0$

Invariant: $\{s = \sum_{i<k} a[i]\}$

Variant: $n - k$

`while` $k \neq n$ `do`

$\{s = \sum_{i<k} a[i]\}$

$\{s + a[k] = \sum_{i<k+1} a[i]\}$

$s, k := s + a[k],\ k + 1$

$\{s = \sum_{i<k} a[i]\}$

`od`

$\{s = \sum_{i<k} a[i] \wedge \neg(k \neq n)\}$

$\{s = \sum_{i<n} a[i]\}$

The Plateau Problem

- Given array $a[0, n)$ of integers, in non decreasing order:

$$a[i] \leq a[i+1] \qquad (i < n-1)$$

- A plateau is a sub-array $a[i..j-1]$ of constant values: $\quad a[i] = a[j-1]$.
- Task: Find the maximal length of a plateau in a.

Problem setting

Global: $a[0,n)$ of integers $\forall i \in [0,n).\ a[i] \leq a[i+1]$

```
while      do




od
```

$\{p =$ maximal length of a plateau in $a[0..n-1]\}$

The invariant and variant

Global: $a[0,n)$ of integers $\forall i \in [0,n).\ a[i] \leq a[i+1]$

Invariant: $\{p =$ maximal length of a plateau in $a[0..k-1]\}$

Variant: $n-k$

```
while        do



od
```

$\{p =$ maximal length of a plateau in $a[0..n-1]\}$

Initializing

Global: $a[0,n)$ of integers $\forall i \in [0,n).\ a[i] \leq a[i{+}1]$

$\boxed{k,p := 1,1}$

Invariant: $\{p =$ maximal length of a plateau in $a[0..k{-}1]\}$

Variant: $n - k$

```
while      do

od
```

$\{p =$ maximal length of a plateau in $a[0..n{-}1]\}$

The guard

Global: $a[0,n)$ of integers $\forall i \in [0,n).\ a[i] \leq a[i+1]$

$\boxed{k, p := 1, 1}$

Invariant: $\{p = \text{ maximal length of a plateau in } a[0..k-1]\}$

Variant: $n-k$

`while` $\boxed{k \neq n}$ `do`

`od`

$\boxed{\{\text{Invariant} \wedge \neg(k \neq n)\}}$

$\{p =$ maximal length of a plateau in $a[0..n-1]\}$

Insight for maintaining the invariant

- a_k is the last entry of a plateau of length $> p$

  iff

  $a_k = a_{k-p}$
- To maintain the invariant while reducing the variant, the loop body is:

$$\begin{array}{l} \text{if } a[k] = a[k-p] \\ \qquad \text{then } k, p := k{+}1, p{+}1 \\ \qquad \text{else } k := k + 1. \end{array}$$

The complete program

Global: $a[0,n)$ of integers, $\forall i \in [0,n).\ a[i] \leq a[i+1]$

```
k, p := 1, 1
Invariant: {p = maximal length of a plateau in a[0..k−1]}
Variant: n − k
while k ≠ n do
    {Invariant}
    if a[k] = a[k−p]
    then
        {Invariant ∧ a[k] = a[k−p]}
        {p+1 = maximal length of a plateau in a[0..k]}
        k, p := k+1, p+1
    else
        {Invariant ∧ a[k] ≠ a[k−p]}
        {p = maximal length of a plateau in a[0..k]}
        k := k+1
    fi
    {Invariant}
od
{Invariant ∧ ¬(k ≠ n}
{p = maximal length of a plateau in a[0..n−1]}
```

Saddle search

- Given two-dimensional array $a[0..m)[0..n)$ of integers, with all rows and columns increasing:

$$\begin{aligned} a_{ij} &< a_{i,j+1} \\ a_{ij} &< a_{i+1,j} \end{aligned}$$

- Count the number of occurrences of x in a.
- Mathematical insight:
  In searching for entries of a equal to x, the most fruitful inspection of a is the entry $a_{0,n-1}$:
  - if it is $\leq x$,
    then row $a[0, n-2]$ is $< x$, whence removable from search
  - if $a_{0,n-1}$ is $\geq x$,
    then column $a[1..m-1, n-1]$ is $> x$, whence removable

Outline of development

- Changing quantity: internal sub-arrays $a[q..m)[0..r)$
- Take k counter for $\#$ of occurrences of x counted so far.
  Invariant relates k and the changing $a[q..m)[0..r)$:

$$k = \# \text{ of occurrences of } x \text{ in } a \text{ outside } a[q..m)[0..r)$$

- Initialize: $q, r, k := 0, n, 0,$
- Variant = size of the sub-array remaining to search: $(m - q) + r$.
- The guard holds as long as the remaining array has not vanished: $q \neq m \;\vee\; r \neq 0$.
  (Note: failure of either disjunct implies $a[q..m)[0..r) =$ empty, guaranteeing the post-condition.
- The body of the loop uses the insight:
  - If $a_{q,r-1} < x$, eliminate first row: $q := q{+}1$.
    Counter k remains unchanged, since x is absent.
  - If $a_{q,r-1} > x$, eliminate last column : $r := r{-}1$
  - If $a_{q,r-1} = x$, we eliminate both, and increment k:

$$q, r, k := q{+}1, r{-}1, k{+}1$$

The Saddle Search program

{Global: $a[0..m)[0..n)$ array of integers, row/column increasing }

$q, r, k := 0, n, 0$

Invariant: $\{k =$ number of occurrences of x in a outside $a[q..m)[0..r)\}$

Variant: $(m - q) + r$

while $q \neq m \ \vee \ r \neq 0.$ do

  if $a_{q,r-1} < x \ \rightarrow$

    $\{\text{Invariant} \wedge a_{r,q-1} < x\}$ $\{k = \ \# \ x$ in a outside $a[q{+}1..m)[0..r)\}$ (*)

    $q := q{+}1$ $\{\text{Invariant}\}$

  [] $a_{q,r-1} > x \ \rightarrow$

    $\{\text{Invariant} \wedge a_{r,q-1} > x\}$ $\{k = \ \# \ x$ in a outside $a[q..m)[0..r{-}1)\}$ (*)

    $r := r{-}1$ $\{\text{Invariant}\}$

  [] $a_{q,r-1} = x \ \rightarrow$

    $\{\text{Invariant} \wedge a_{r,q-1} = x\}$

    $\{k{+}1 = \ \# \ x$ in a outside $a[q{+}1..m)[0..r{-}1)\}$ (*)

    $q, r, k := q{+}1, r{-}1, k{+}1$ $\{\text{Invariant}\}$

  fi $\{\text{Invariant}\}$

od

$\{\text{Invariant} \wedge \neg(q \neq m \ \vee \ r \neq 0)\}$

$\{k = \ \# \ x$ in a $\}$