2IP15 Programming Methods

From Small to Large Programs

Tom Verhoeff
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

www.win.tue.nl/~wstomv/edu/2ip15

Overview

- Inductive definitions in logic
- Recursion

Definitions in Mathematical Logic

A **definition** introduces an abbreviation for a valid expression.

Example: We abbreviate $(\forall\, x : x \in A : x \in B)$ as $A \subseteq B$

Definitions often are 'parameterized' (as above), and can be 'nested':

Example: We define relation $\supseteq$ on sets by $A \supseteq B \Leftrightarrow B \subseteq A$

Such definitions can always be eliminated via (repeated) substitutions:

$$
\begin{array}{ll}
 & X \cup Y \supseteq X \cap Y \\
\overset{\text{val}}{=\!=} & \quad \{ \text{ definition of } \supseteq \ \} \\
 & X \cap Y \subseteq X \cup Y \\
\overset{\text{val}}{=\!=} & \quad \{ \text{ definition of } \subseteq \ \} \\
 & (\forall\, x : x \in X \cap Y : x \in X \cup Y)
\end{array}
$$

Inductive Definitions in Logic

Compare these definitions of the factorial function $fac : \mathbb{N} \to \mathbb{N}$:

- $fac(n) = 1 * 2 * \cdots * n$

  ellipses $(\ldots)$ are not formal: what if $n = 0, 1, 2$?

- $fac(n) = (\prod\, k : 1 \leq k \leq n : k)$

  relies on a *quantifier*

- $$fac(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * fac(n-1) & \text{if } 1 \leq n \end{cases}$$

  inductive definition (a.k.a. recursive definition)

  looks *circular*, but is not

Inductive Definition: Terminology

- $$fac(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * fac(n-1) & \text{if } 1 \leq n \end{cases}$$

There is a case distinction:

base (case): case that does *not* refer to the entity being defined

(inductive) step: case that *does* refer to the entity being defined

Alternative form of inductive definition:

- $$\begin{cases} fac(0) & = & 1 \\ fac(n+1) & = & (n+1) * fac(n) \quad \text{if } 0 \leq n \end{cases}$$

Inductive Definition: Elimination

- $$fac(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * fac(n-1) & \text{if } 1 \leq n \end{cases}$$

An inductive definition cannot be eliminated in *general*: $fac(a) = ?$

But in every *specific* application, it can be eliminated by rewriting:

$$\begin{array}{ll} & fac(2) \\ = & \quad \{ \text{ definition of } fac(2) \text{ (step, because } 1 \leq 2 \text{ } \} \\ & 2 * fac(2-1) \\ = & \quad \{ \text{ } 2-1=1, \text{ definition of } fac(1) \text{ (step, because } 1 \leq 1 \text{ } \} \\ & 2 * (1 * fac(1-1)) \\ = & \quad \{ \text{ } 1-1=0, \text{ definition of } fac(0) \text{ (base) } \} \\ & 2 * (1 * 1) \end{array}$$

Inductive Definition: Termination

Not every inductive definition is valid: $fac(n) := fac(n+1)/(n+1)$

To be valid, the induction must be **well-founded**:

> Every sequence of rewrites *terminates* (on a base case).

One practical way of guaranteeing termination is to provide a **bound (function)** bf (a.k.a. **variant function**), such that

- if $f : \mathcal{D} \to \mathcal{R}$ is defined inductively, then $bf : \mathcal{D} \to \mathbb{N}$
- for every $f(E)$ occurring in definition of $f(p)$, $bf(E) < bf(p)$

Hence, $f(p)$ can be rewritten to a base case in at most $bf(p)$ steps.

N.B. $bf(p)$ is an *upper bound* on number of steps; could be very crude.

Examples for termination of induction

- $$fac(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * fac(n-1) & \text{if } 1 \leq n \end{cases}$$

Take $bf : \mathbb{N} \to \mathbb{N}$ with $bf(n) = n$.

Verification conditions:

- $bf : \mathbb{N} \to \mathbb{N}$
- $bf(n-1) < bf(n)$

This bound bf is tight (exact).

Examples for termination of induction

$f : \mathbb{N} \to \mathbb{N}$ with $f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 + f(\lfloor n/2 \rfloor) & \text{if } 0 < n \end{cases}$

Take $bf : \mathbb{N} \to \mathbb{N}$ with $bf(n) = n$.

Verify: $bf(\lfloor n/2 \rfloor) < bf(n)$ if $0 < n$:

$$
\begin{array}{ll}
 & \lfloor n/2 \rfloor < n \\
\overset{\text{val}}{\Longleftarrow} & \quad \{\ \lfloor n/2 \rfloor \leq n/2,\ \text{transitivity}\ \} \\
 & n/2 < n \\
\overset{\text{val}}{\Longleftarrow} & \quad \{\ \text{algebra}\ \} \\
 & 0 < n
\end{array}
$$

This bound bf is very crude.
Tight (but not easier to prove): $bf(n) = \lfloor \log_2(n+1) \rfloor$

Termination of induction can be difficult to prove

$f : \mathbb{N} \to \mathbb{N}$ with

$$f(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 1 + f(n/2) & \text{if } 1 < n \ \wedge \ n \text{ is even} \\ 1 + f(3n+1) & \text{if } 1 < n \ \wedge \ n \text{ is odd} \end{cases}$$

It is unknown whether this inductive definition is valid (terminates).

For all the many values that have been tried, it terminates.

Recursive Method Definitions in Java

- In some older programming languages, recursion was forbidden.

  Reason: Function body used fixed memory locations for parameter values, return value, and return address.

  Modern processors use a *stack*.

- Java allows recursive method definitions.

- Compiler will not enforce proper termination: `StackOverflowError`

- In annotation, use `@bound` to document a bound function.

  `@bound` is not standard javadoc

  Available via `PrePostDoclet.jar`

Factorial Recursively in Java

```
1  /**
2   * @pre      0 <= n
3   * @return   n factorial
4   * @bound    n
5   */
6  public static long fac(int n) {
7      if (n == 0) {
8          return 1;
9      } else { // 0 < n
10         return n * fac(n - 1);
11     }
12 }
```

Recursive Method Definitions: Terminology

Static call graph of program P:

- Nodes: each method definition in P is a node
- Arrows: arrow $q \rightarrow r$ when body of q contains a call to r

Recursion: Directed cycle in call graph
(*syntactic* phenomenon, i.e. does not depend on *execution*)

Direct recursion: Cycle of length 1: body contains call to itself

Indirect recursion: Cycle of length > 1

Mutual recursion: Cycle of length 2

Sum Digits Recursively

```
1  /**
2   * @pre      0 <= n && 2 <= r
3   * @return   sum of digits of n in radix-r notation
4   * @bound    n
5   */
6  public static int sumDigits(int n, int r) {
7      if (n == 0) {
8          return 0;
9      } else { // 0 < n && 2 <= r, hence n / r < n
10         return n % r + sumDigits(n / r, r);
11     }
12 }
```

N.B. Bound is crude: amount of work is actually *logarithmic* in `n`.

Sum Array Elements Recursively (Defective! Why?)

```
1  /**
2   * @pre      a != null && 0 <= i <= j <= a.length
3   * @return   (\sum k; i <= k < j; a[k])
4   * @bound    j - i
5   */
6  public static int sumArrayBad(int[] a, int i, int j) {
7      if (i == j) { // empty interval
8          return 0;
9      } else { // i < j
10         int h = (i + j) / 2; // h between i and j
11         return sumArrayBad(a, i, h) + sumArrayBad(a, h, j);
12     }
13 }
```

Sum Array Elements Recursively (Correct)

```
1  /**
2   * @pre      a != null && 0 <= i <= j <= a.length
3   * @return   (\sum k; i <= k < j; a[k])
4   * @bound    j - i
5   */
6  public static int sumArray(int[] a, int i, int j) {
7      if (i == j) { // j - i == 0: empty interval
8          return 0;
9      } else if (i + 1 == j) { // j - i = 1: singleton
10         return a[i];
11     } else { // i + 1 < j, hence  j - i >= 2
12         int h = (i + j) / 2;
13         // i < h < j, hence  h - i < j - i  and  j - h < j - i
14         return sumArray(a, i, h) + sumArray(a, h, j);
15     }
16 }
```

Could be faster by parallel execution on multiprocessor system.

Proof of Termination for `sumArray`

$h - i < j - i \;\wedge\; j - h < j - i$ [to be proved, if $i + 1 < j$]

$\overset{\text{val}}{=\!=}$ { algebra }

$i < h < j$

$\overset{\text{val}}{=\!=}$ { definition of `h` }

$i < \lfloor (i + j)/2 \rfloor < j$

$\overset{\text{val}}{=\!=}$ { algebra }

$2i < 2\lfloor (i + j)/2 \rfloor < 2j$

$\overset{\text{val}}{=\!=}$ { property of integer division (mod $\rightarrow$ remainder after division) }

$2i < i + j - (i + j) \bmod 2 < 2j$

$\overset{\text{val}}{=\!=\!|}$ { property of mod: $0 \leq a \bmod 2 < 2$ }

$2i < i + j - 1 \;\wedge\; i + j < 2j$

$\overset{\text{val}}{=\!=\!|}$ { algebra }

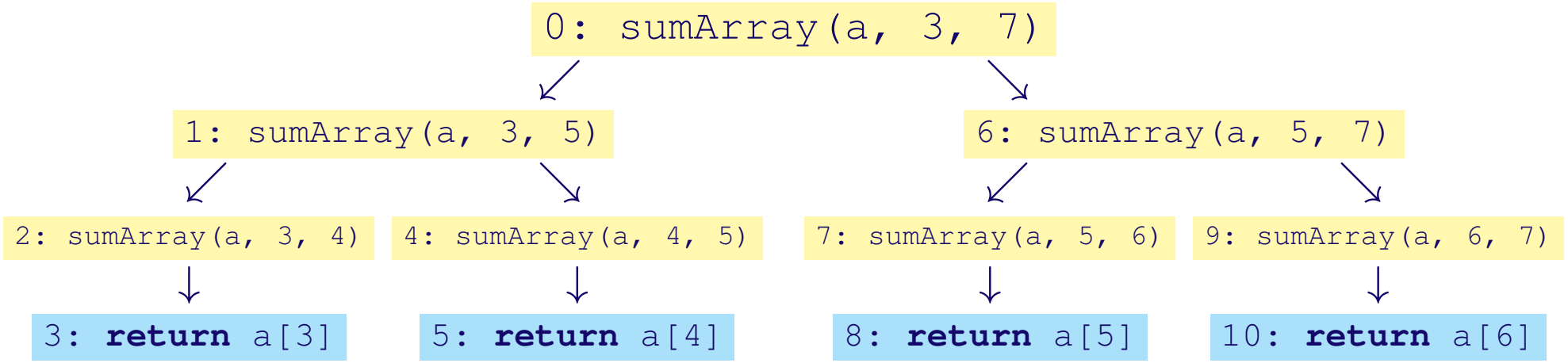
$i < j - 1 \;\wedge\; i < j$ [given by case condition]

Dynamic Call Tree

Dynamic call tree of an *execution* of program P:

- Nodes: specific routine *calls* in execution of P
- Arrows: $q \rightarrow r$ when execution of call q results in call r

Example for `sumArray` with `j - i` $= 4$:



Recursion depth is logarithmic in `j - i`; amount of work is linear.

Summary

- Inductive definitions: base case, inductive step

  induction must be well founded; bound function

- Static call graph: recursion $\Leftrightarrow$ directed cycle

  direct recursion, indirect recursion, mutual recursion

- Dynamic call graph: is a tree (no cycles, not even undirected)