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

- Eliminate recursion by a loop; in general, requires a stack
- Transforming a while loop into (tail) recursion
- Eliminate tail recursion by a loop
- Generating combinatorial objects: backtracking
- Operating on recursive data structures (lists, trees, . . .)
- Parsing structured texts
- Generic type definitions, type parameters

Definition and Classification of Recursion

In preceding lecture:

- Static call graph concerns program *text*
- Direct, indirect, and mutual recursion (syntactic notions)
- Dynamic call graph concerns program *execution*

  Conforms to static call graph.

  Is a tree (no cycles).

  Can differ from run to run, depending on input/parameters.
- Bound function is used to argue about termination

Degenerate Recursion

```
1  /** @bound  0 */
2  void bogus() {
3      if (false) {
4          bogus(); // never called
5      }
6  }
7
8  /** @bound  0 if 0 <= n else 1 */
9  void printAbs(int n) {
10     if (0 <= n) { // What if condition would be  0 < n ?
11         System.out.println(n);
12     } else { // n < 0
13         printAbs(-n);
14     }
15 }
```

Further Classification of Recursion

Given a recursively defined function f:

- Linear recursion: dynamic call graph of f is linear w.r.t. f

  Each call of f gives rise to *at most one* direct other call of f.

- Branching recursion: dynamic call graph of f branches w.r.t. f

  Some calls of f may give rise to *multiple* direct other calls of f.

Independent of number of syntactic occurrences of recursive calls.

Multiple syntactic occurrences could be linear recusion (**`if else`**).

Single syntactic occurrence could be branching recursion (inside loop).

Tail Recursion

Tail recursive call: A recursive call that occurs as last action before the method body returns.

N.B. `fac` of preceding lecture is *not* tail recursive, because in

```
return n * fac(n - 1);
```

the part `n *` is executed *after* `fac(n - 1)` returns.

Tail recursion: when each recursive call is tail recursive.
This is a form of linear recursion.

Tail recursion is special, because it can easily be transformed into a loop without using a stack.

Eliminating a `while` loop by recursion

```
while (Cond) {
    Body;
}
Fini;
```

can be transformed into the call `whileRec();` where

```
void whileRec() {
    if (! Cond) {
        Fini;
    } else {
        Body;
        whileRec();
    }
}
```

This is tail recursion: the recursive call is the last action.
N.B. It is often useful to introduce suitable parameters.

Trade-off between Loop and Recursion

- Loops and recursion are 'equally expressive'.

  You can program the same things with them.

- Some problems are easier to solve and implement by recursion.

  For others, a loop is more 'natural'.

- Recursion incurs overhead:

  Method call with parameter/return-value passing costs time.

  Stack frames cost memory; depends on maximum recursion depth.

- In (pure) Functional Programming there are no variables and loops, only parameters and recursion.

Eliminating `void` tail recursion by a `while` loop

```
void tailRec() {
    if (Cond) {
        Base; // does not call tailRec()
    } else {
        Step; // does not call tailRec()
        tailRec();
    }
}
```

The work done looks like: `Step; Step; ...; Step; Base;`
The call `tailRec();` can be transformed into the loop

```
while (! Cond) {
    Step;
}
Base;
```

Eliminating parameterized `void` tail recursion by a `while` loop

```
void tailRec(T t) {
    if (Cond(t)) {
        Base(t);
    } else {
        Step(t);
        if (B(t)) tailRec(E); else tailRec(F);
    }
}
```

The call `tailRec(v);` can be transformed into call `nonRec(v)`, where:

```
void nonRec(T t);
    while (! Cond(t)) {
        Step(t);
        if (B(t)) t = E; else t = F;
    }
    Base(t);
}
```

Eliminating non-`void` tail recursion by a `while` loop

```
T tailRec(T t) {
    if (Cond(t)) {
        return Base(t);
    } else {
        Step(t);
        if (B(t)) return tailRec(E); else return tailRec(F);
    }
}
```

The call `tailRec(v)` can be transformed into call `nonRec(v)`, where

```
T nonRec(T t) {
    while (! Cond(t)) {
        Step(t);
        if (B(t)) t = E; else t = F;
    }
    return Base(t);
}
```

Example of transforming non-tail recursion into tail recursion

```
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 }
```

Also see `Recursion2.java`.

Transforming non-tail recursion into tail recursion

Generalize the expression `n * fac(n - 1)` as `b * fac(a)`

That is, the subexpressions `n` and `n - 1` are *decoupled*.

Give it a name (*overloading*): `fac(a, b) = b * fac(a)`

The original problem is a *special case*: `fac(n) = fac(n, 1)`

Derive recurrence relation: Base case

```
    fac(0, b)
=     { Def. of generalized fac(int, int) }
    b * fac(0)
=     { Def. of original fac(int), base case }
    b * 1
=     { Unit of * }
    b
```

Derive recurrence relation: Inductive step

For `0 < n`:

```
    fac(n, b)
=     { Def. of generalized fac(int, int) }
    b * fac(n)
=     { Def. of original fac(int), inductive step, using 0 < n }
    b * (n * fac(n - 1))
=     { Associativity of * }
    (b * n) * fac(n - 1)
=     { Def. of generalized fac(int, int) }
    fac(n - 1, b * n)
```

The last equality is where we see that the generalization 'works'.

Tail recursive version of `fac(n, b)`

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

Parameter `b` is called an accumulation parameter.

Iterative version of `fac(n, b)`

```
1  /**
2   * @pre      0 <= n
3   * @return   b times (n factorial)
4   * @bound    n
5   */
6  public static long facIter(int n, long b) {
7      while (n != 0) {
8          b = b * n; // Mind the order!
9          n = n - 1; // No multi-assignment in Java
10     }
11     return b;
12 }
```

Iterative version of `fac(n) == fac(n, 1)`

Turn accumulation parameter into a local variable:

```
1  /**
2   * @pre      0 <= n
3   * @return   n factorial
4   * @bound    n
5   */
6  public static long facIter(int n) {
7      long b = 1;
8      while (n != 0) {
9          b = b * n;
10         n = n - 1;
11     }
12     return b;
13 }
```

Recursion: Some Observations

- Recursion is a special case of Divide & Conquer:

  Decompose a problem into simpler version(s) of itself

- Recursion cannot be mastered by operational reasoning

  about what happens in the machine during execution

- Recursion can be mastered by contractual reasoning

  about what the contract requires (`@pre`) and ensures (`@post`)

Combinatorial Objects: Bit Patterns

All bit patterns of length 3:

```
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

How to generate with a program?

Bit Patterns of Length 3: Nested `for` loops

```
1  /**
2   * @pre   true
3   * @post  each 3-bit pattern has been printed exactly once
4   */
5  public static void generateAll3BitPatterns() { // Not recursive
6      for (int b2 = 0; b2 <= 1; ++ b2) {
7          for (int b1 = 0; b1 <= 1; ++ b1) {
8              for (int b0 = 0; b0 <= 1; ++ b0) {
9                  System.out.println ( "" + b2 + b1 + b0 );
10             }
11         }
12     }
13 }
```

How to generate all bit patterns of length N (parameter)?

Variable number of nested **`for`** loops?

Recursive Nature of Bit Patterns of Length n

0	0	0
0	0	1
0	1	0
0	1	1

1	0	0
1	0	1
1	1	0
1	1	1

Base case: When $n = 0$, there is one such pattern (empty string)

Step: When $0 < n$, all n-bit patterns are the union of:

- a `0` followed by each of the $(n-1)$-bit patterns
- a `1` followed by each of the $(n-1)$-bit patterns

Generalized problem: print `String s` followed by each n-bit pattern. (`s` is an accumulation parameter)

Bit Patterns of Length n: Recursive Routine

```
1  /**
2   * @pre   0 <= n
3   * @post  each bit pattern with prefix s and n additional bits
4   *   (that is, s.length()+n bits) has been printed exactly once
5   * @bound n
6   */
7  public static void generate(String s, int n) {
8      if (n == 0) {  // base case
9          System.out.println(s);
10     } else {  // 0 < n, inductive step
11         for (int b = 0; b <= 1; ++ b) {
12             // pre: 0 <= n - 1; bound: n - 1 < n
13             generate(s + b, n - 1);
14         }
15     }
16 }
```

Branching recursion with single syntactic occurrence of recursive call

Bit Patterns of Length n: Initial Call

```
generate ( "", 5 )
```

In a way, this involves a variable number of nested **`for`** loops.

Running time grows exponentially with recursion depth.
Memory overhead (in stack frames) grows linearly.

Note that the string and integer parameter are repeatedly passed on.

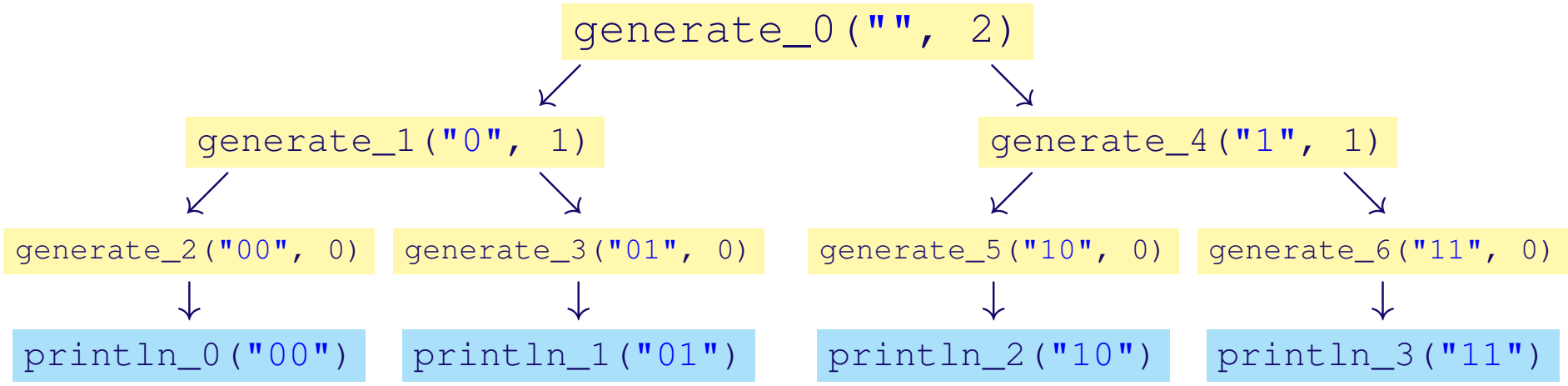
This involves a lot of superfluous copying, which can be suppressed (via global variables, but that falls outside the scope of this course).

Dynamic Call Tree

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

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

Example for bit patterns with $n = 2$:



Bit Patterns of Length n with $k \leq n$ 1's by Filtering

Generate all n-bit patterns and *filter* out unwanted patterns.

```
1  /**
2   * @pre   0 <= n && 0 <= k && m == (\num_of i; s.has(i); s.charAt(i) == '1')
3   * @post  each bit pattern with prefix s and n additional bits,
4   *   having k 1's, has been printed exactly once
5   * @bound n
6   */
7  public static void generate(String s, int n, int k, int m) { // Inefficient
8      if (n == 0) {  // base case: all bits are there
9          if (m == k) {  // keep the desired patterns
10             System.out.println(s);
11         } else { /* m != k: drop the others */ }
12     } else {  // 0 < n, inductive step: extend by one bit b
13         for (int b = 0; b <= 1; ++ b) {
14             generate(s + b, n - 1, k, m + b);
15         }
16     }
17 }
```

Bit Patterns of Length n with $k \leq n$ 1's by Filtering

In general, filtering is inefficient:

- The number of n-bit patterns is 2^n.

  This is exponential in n.

- The number of n-bit patterns with k 1's is $\binom{n}{k}$.

  This is polynomial in n for fixed k.

  E.g. $\binom{n}{2} = n(n-1)/2$ is quadratic; much less than exponential

Better: filter earlier, and generate only useful prefixes.
Cf. Branch & Bound

Bit Patterns of Length n with $k \leq n$ 1's: Filter Earlier

```
1  /**
2   * @pre   0 <= k <= n
3   * @post  each bit pattern with prefix s and n additional bits,
4   *   of which k 1's, has been printed exactly once
5   * @bound n
6   */
7  public static void generateA(String s, int n, int k) {
8      if (n == 0) {  // k == 0, base case
9          System.out.println(s);
10     } else {  // 0 < n, inductive step
11         if (k <= n - 1) {  // can afford a 0
12             generate(s + 0, n - 1, k);
13         }
14         if (1 <= k) {  // can afford a 1
15             generate(s + 1, n - 1, k - 1);
16         }
17     }
18 }
```

Bit Patterns of Length n with $k \leq n$ 1's: Filter Earlier (2)

```
 1 /**
 2  * @pre   0 <= k <= n
 3  * @post  each bit pattern with prefix s and n additional bits,
 4  *   of which k 1's, has been printed exactly once
 5  * @bound n
 6  */
 7 public static void generate(String s, int n, int k) {
 8     if (n == 0) {  // k == 0, base case
 9         System.out.println(s);
10     } else {  // 0 < n, inductive step
11         for (int b = 0; b <= 1; ++ b) {
12             if (0 <= k - b && k - b <= n - 1) {
13                 generate(s + b, n - 1, k - b);
14             }
15         }
16     }
17 }
```

Recursive methods on recursive data structures

A natural use of recursive methods is on recursive data structures.

List : a list is either empty, or a pair of a value and a list

Tree : a tree is either empty, or a value and a tuple of trees

Java allows recursive data type definitions:

```
1 public class BinaryTree<E> {
2     BinaryTree<E> left;
3     BinaryTree<E> right;
4     E value;
5 }
```

null can serve as base case, representing the empty tree Rep invariant should exclude cycles and sharing (aliasing)

Backtracking

- Speculative technique to search through a state space.
- Recursive implementation for Binary Puzzle Assistant:

  1. Find an open cell c. If not found: solved (base case).
  2. For each possible cell state s, set c to s, apply "strategies", and if state still valid, solve remainder recursively.
     Bound: number of open cells

- Implementation variations:

  – Copy entire state (onto the stack) for each recursive call.

    Undo is "automatic" by falling back to earlier stored state.

  – Maintain state in global variable, and modify it: do-undo.

    Here, the Command Pattern can be used.

Parsing nested parentheses

(( ))

(( ) ( )) ( )

) (

(( ) ( )

Iterative solution: count number of unclosed parentheses read so far

This count may never be *negative* and should end at *zero*.

The counter can be viewed as a very primitive stack (with *tallies*).

Parsing nested parentheses recursively

A well-nested parenthesis expression E is

- either *empty* ,
- or (E) E

See `Recursion2.java`

```
1 /**
2   * @pre    true
3   * @return whether s is a well-nested () expression
4   */
5 public static boolean isWellNested(String s)
```

Parsing nested parentheses: Generalized specification

```
 1 /**
 2  * @pre    0 <= i <= s.length()
 3  * @post   i <= \result <= s.length() &&
 4  *     s.substring(i, \result) is a maximal well-nested () expression,
 5  *     appearing from index i onward in s, if no exception
 6  * @throw  ParseException  if maximal parsed prefix cannot be completed
 7  * @bound  s.length() - i
 8  */
 9 public static int parseNested(String s, int i)
10         throws ParseException
```

Parsing nested parentheses: Using the Generalization

```
1  public static boolean isWellNested(String s)
2  {
3      try {
4          int i = parseNested(s, 0);
5          System.out.println("Correct up to index " + i);
6          return i == s.length();
7      } catch (ParseException e) {
8          System.out.println(e);
9          return false;
10     }
11 }
```

Recursive implementation of `parseNested(String s, int i)`

```
1     if (s.length() == i) { // nothing remains beyond i
2         return i;
3     }
4     // 0 <= i < s.length()
5     if (s.charAt(i) == '(') {
6         char exp = ')'; // next character expected
7         // bound: s.length() - (i + 1) < \old(s.length() - i)
8         i = parseNested(s, i + 1);
9         if (i == s.length() || s.charAt(i) != exp) {
10            throw new ParseException("Expected " + exp + " at index " + i);
11        }
12        // i < s.length() && s.charAt(i) == exp, skip and parse remainder
13        // bound: s.length() - (i + 1) < \old(s.length() - i)
14        return parseNested(s, i + 1);
15    } else { // s.charAt(i) != '('
16        return i;
17    }
```

Parsing multiple kinds of nested parentheses

```
( [ ) ]
```

```
[ ( ) ]
```

```
( [ ] )
```

A well-nested multi-parenthesis expression E is one of

- *empty*
- (E) E
- [E] E
- { E } E

Generic Type Definitions: Overview

- Example in JCF: `List<E>` is a generic type definition

  `E` is a formal type parameter

- Usage: Substitute concrete type for formal type parameter

  Examples: `List<String>`, `List<Set<Integer>>`

  Auto boxing, unboxing: **`int`** $\leftrightarrow$ `Integer`

- Definition: formal type parameter is used in definition as a type

```
public class Pair<A, B> {
    public A a;
    public B b;
}
```

Generic Type Definitions: Motivation

- Pre-Java 5: `List` involves list of `Object`

  User responsible for proper typing

```
List list = new ArrayList(); // intended as list of integers
list.add( "okay" ); // no complaint, but unintended
list.add( 42 );
int i = (Integer)list.get(1); // cast needed
```

  Java 5 and beyond: User can indicate intention to compiler

-
```
List<Integer> list = new ArrayList<Integer>();
list.add( "okay" ); // compile error
list.add( 42 );
int i = list.get(1); // no cast needed
```

Generic Type Definitions: Complications

- If `U` is a subtype of `T`, then `C<U>` is *not* a subtype of `C<T>`

  Method **`void`** `m(List<Object> list)`
  cannot be called as `m(`**`new`** `ArrayList<String>)`

- Wildcards:

  - `C<?>`: any `C<U>` is a subtype
  - `C<?` **`extends`** `T>`: `C<U>` is subtype if `U` **`extends`** `T`
  - `C<?` **`super`** `T>`: `C<S>` is subtype if `T` **`extends`** `S`

See: `docs.oracle.com/javase/tutorial/extra/generics/morefun.html`

Assignment: Define Generic `Relation<A, B>`

Not required, see peach[3]

Summary

- Linear, Tail, Branching recursion
- Transformations between tail recursion and loops
- Generate combinatorial objects: backtracking
- Parse structured text
- Generic type definitions, type parameters