

# Programming Paradigms

©Andrey Kruglyak, 2010

Wednesday, April 21, 2010

1

# Programming paradigms

- ◁ We will briefly look at:
  - ◁ imperative programming (C, C++, Objective-C, Java, ...)
  - ◁ functional programming (Haskell, Standard ML, Scheme, Scala, Erlang, ...)
  - ◁ parallel programming (CCS,  $\pi$ -calculus, actors, Ada, ...)
  - ◁ logic programming (Prolog)
  - ◁ neural network programming
- ◁ Note that OO (object-oriented) programming is not on this list!

Wednesday, April 21, 2010

2

# Imperative programming

- ◁ In its purest form, an imperative program is a sequence of instructions that change state variables (inside the program or outside it = output) - i.e. only side-effects
- ◁ Large programs easily become difficult to write and understand (though they can be very effective)
- ◁ Extensions to imperative programming decreasing this complexity:
  - ◁ procedures
  - ◁ object-orientation

Wednesday, April 21, 2010

3

# Imperative programming

- ◁ The programmer has full control over which instructions are executed and when, as well as over how memory is used
- ◁ Even in high-level languages, with a higher abstraction level (e.g. Java), it is always known to the programmer how high-level instructions will be translated into low-level instructions

Wednesday, April 21, 2010

4

## Functional programming

- ◀ Main idea: a program is seen as a function from input values to an output value; a function performs a calculation and returns a value
- ◀ Typically, functions are values, i.e.
  - ◀ a function is a valid result of a computation
  - ◀ a function can be sent around in the program
  - ◀ a function of several arguments can be applied partially, creating a new function

Wednesday, April 21, 2010

5

## Functional programming

- ◀ In a functional program, we define the result of a computation but abstract from the details of how the computation is performed
- ◀ Compared to an imperative program, a functional program is typically shorter, it is easier to map an algorithm to code, but:
  - ◀ implementation is often less effective (but how to compare effectiveness of a quick program that sometimes calculates a wrong value to that of a slow program that always calculates a correct value?)
  - ◀ it is not very good for programming interaction with the environment or the user

Wednesday, April 21, 2010

6

## Haskell

- ◀ Actually, the only "pure" functional language is Haskell (the others mix functional and imperative paradigms, as they allow side-effects)
- ◀ Even Haskell employs special data structures - monads - to encode interaction with the environment
  - ◀ A monad can be used to define a sequence of operations, where the result of one operation is piped to the next operation
  - ◀ IO monad allows to view a program as a function from an infinite stream of inputs to an infinite stream of outputs
  - ◀ Monads are quite difficult to work with!

Wednesday, April 21, 2010

7

## More about Haskell...

- ◀ In Haskell, we have lazy evaluation - call-by-need
- ◀ This allows us to define large data structures but if a program only uses a small part of these, only that part will be created

```
takeFirstFiveElements  
  (map aTediousFunction [1..10000])
```

Wednesday, April 21, 2010

8

## Example

### ◀ imperative style

```
int[] v = new int[5];
int index = 0;
int i = 0;
while (i<10) {
  if (i%2 == 1) {
    v[index] = i;
    index++;
  }
}
```

the result is in memory location denoted by "v"

### ◀ functional style

```
takeOdd [] = []
takeOdd (x:xs)
  | isOdd x
    = x:takeOdd xs
  | otherwise
    = takeOdd xs

takeOdd [1..9]
```

to trigger the computation, takeOdd [1..9] should be invoked from within an IO monad

## Parallel programming

- ◀ A program is a set of processes that send messages to each other (no side-effects, no functions)
- ◀ There is no programming language that has message sending as its main operation, but some include this feature (Ada), and many offer libraries implementing it
- ◀ It is well defined in several calculi, e.g. Calculus of Communication Systems and  $\pi$ -calculus

## $\pi$ -calculus

- ◀ Let  $P$  and  $Q$  denote processes. Then
  - $P \mid Q$  denotes a process composed of  $P$  and  $Q$  running in parallel
  - $a(x).P$  denotes a process that waits to read a value  $x$  from the channel  $a$  and then, having received it, behaves like  $P$
  - $\bar{a}\langle x \rangle.P$  denotes a process that first waits to send the value  $x$  along the channel  $a$  and then, after  $x$  has been accepted by some input process, behaves like  $P$
  - $(\nu a)P$  ensures that  $a$  is a fresh channel in  $P$  (read the Greek letter "nu" as "new")
  - $!P$  denotes an infinite number of copies of  $P$ , all running in parallel
  - $P + Q$  denotes a process that behaves like either  $P$  or  $Q$
  - $0$  denotes the inert process that does nothing

## $\pi$ -calculus example

- ◀ Suppose you want to model a remote procedure call between a client and a server.
- ◀ Consider the following function, `incr`, running on the server. `incr` returns the integer one greater than its argument, `x`:

```
int incr(int x) { return x+1; }
```
- ◀ First, we model the "incr" server as a process in  $\pi$ -calculus as follows:

```
!incr(a, x). $\bar{a}\langle x+1 \rangle$ 
```

## $\pi$ -calculus example

◀ `!incr(a, x).ā<x+1>`

Ignoring the ! for now, this process expression says that the `incr` channel accepts two inputs: one is the name of the channel, `a`, which we will use to return the result of calling `incr`, and the other is the argument, `x`, which will be instantiated with an integer value upon a client call. After the call, the process will send back the result of incrementing its argument, `x`, on the channel `a`. The use of the replication operator, !, in the above process expression means that the "incr" server will happily make multiple copies of itself, one for each client interaction.

## $\pi$ -calculus example

◀ Now let's model a client call to the "incr" server. In the following assignment statement, the result of calling `incr` with 17 gets bound to the integer variable `y`:

```
y := incr(17)
```

◀ and would look like this in  $\pi$ -calculus:

```
(va) (incr<a,17> | a(y))
```

## $\pi$ -calculus example

◀ `(va) (incr ā, 17) | a(y)`

This says in parallel: (1) send on the `incr` channel both the channel `a` (for passing back the result value) and the integer value 17, and (2) receive on the channel `a` the result `y`. The use of the `v` operator guarantees that a private channel of communication is set up for each client interaction with the "incr" server.

## $\pi$ -calculus example

◀ Putting the client and server processes in parallel together we get the final process expression:

```
!incr(a, x).ā<x+1> | (va) (incr<a,17> | a(y))
```

This expresses the client call to the "incr" server with the argument 17 and the assignment of the returned value 18 to `y`.

## Logic programming

- ◀ Historically appeared as a part of research on AI
- ◀ The solution to a problem is defined in terms of predicates.
- ◀ Predicates are statements about some variables and/or constants, each variable ranging over a certain set of values. For each combination of values, a predicate containing it can be either true or false.
- ◀ When the program is run, one (or all) combination of values that satisfy the requirements on the solution is found. This is basically done by "brute force", trying out all combinations.

Wednesday, April 21, 2010

17

## Logic programming

- ◀ The choices of values for predicates and variables are made in a certain sequence, defined by the order of the predicates in the query and their internal form. This makes up a tree, with predicates or variables at its nodes.
- ◀ Once one branch is followed to the end (and no solution has been found or multiple solutions are sought), the solver "backtracks" to the nearest node where another choice can be made.
- ◀ In this way, all combinations of values satisfying the query can be found.
- ◀ For efficiency, a special predicate, called "cut", can be used to specify that no backtracking should cross that node.

Wednesday, April 21, 2010

18

## Logic programming

- ◀ Clearly, this approach is most suitable for a specific class of problems – when you can formulate the requirements on the solution rather than the search algorithm.
- ◀ Unfortunately, for large problems this will be grossly inefficient unless the query is well-written, which requires intimate knowledge of the solver.

Wednesday, April 21, 2010

19

## Prolog example

```
sibling(X,Y) :-
    parents(F,M,X),
    parents(F,M,Y),
    X \= Y.
parents(Ann, Mark, John).
parents(Ann, Mark, Kate).
parents(Ann, David, Justin).

? sibling(John,X).

? sibling(X,Y).
```

Wednesday, April 21, 2010

20

## Neural network programming

- ◀ Another example of an approach suitable for a particular class of problems – when the solution algorithm is unknown, but we have access to a large set of “correct” data (pairs of input and output values) and are seeking to create a machine that will provide output for inputs outside this set.
- ◀ A number of parameters are defined in the input. These parameters are assigned “weights” inside the program, and then the program “learns” (adjust the weights) on a training set of known input/output pairs (calculations are followed by feedback from the environment, specifying whether the answer was correct).
- ◀ Once the weights have been adjusted, the program is ready to be used on new input.

## Programming paradigms

- ◀ Imperative programming:  
programming with side-effects (modifying state variables)
- ◀ Functional programming:  
programming with functions (functions return a value)
- ◀ Parallel programming:  
programming with communicating processes
- ◀ Logic programming:  
finding all solutions for a problem formulated using predicates
- ◀ Neural network programming  
learning from a known set of answers before calculating new ones

## In reality...

- ◀ Programming languages typically combine several paradigms, but one is normally central to their operation
- ◀ This determines the class of problems that a language is particularly suitable for