Programming language design and analysis

Functional Programming. Lambda Calculus

Marius Minea

6 October 2010

Functional paradigm [revisited]

Evaluate an expression and use the value for something

Discipline and idea:
Mathematics and the theory of functions

Values produced are non-mutable
Impossible to change part of a composite value
But can make a revised copy of composite value

Atemporal: no matter when done, computation produces same value
pure functional programming is side-effect free

Applicative: all computations done by applying (calling) functions

Natural abstraction: the function
abstracts expression to a function which can be evaluated as an expression

Functions are first class values: full-fledged data just like numbers, lists, ..

Computations driven by needs

after K. Normark, course, Aalborg U.

Key concepts: Binding

binding a name/identifier to an object (expression/value)

static: before running the program (e.g., usual function call)

dynamic: at runtime (e.g., OO virtual method call)

Binding and variable assignment are NOT the same.
Pure functional languages have binding
but do NOT have assignment (mutable values)

Rebinding and mutation are NOT the same.

Scope

= a context to which objects (names, etc.) are associated
an identifier is *visible* within its scope

lexical (static) scoping
rules determined by program text, not by runtime execution sequence
aids modularity, understanding, reasoning (in isolation)

dynamic scoping
scope is remainder of the execution during which binding is in effect
each identifier has stack of bindings (push/pop on enter/exit scope)
meaning of code (e.g. function) depends on past execution (of other code)

Some languages allow choice of static / dynamic scoping (e.g., Perl)

First-class functions

Functions can be:

passed as an argument

returned as a value, and

stored in a data structure.

```
List.map (fun x -> x + 1) [1;2;3]
```

Higher-order functions

= functions that return a function

e.g., `(+): int -> int -> int = <fun>`
`(+) 3: int -> int = <fun>` (same as `fun x -> x + 3`)

A function of several parameters can be rewritten through *currying*
(after *Haskell Curry*)

```
fun x y -> x + y
fun x -> fun y -> x + y
```

Closures

$=$ a function together with an environment, defining its free variables
needed to implement static scoping with first-order functions