Advanced (Functional) Programming - *Embedded Domain Specific Languages*

Rinus Plasmeijer, Pieter Koopman and Mart Lubbers

Software Science
Radboud University Nijmegen

Generic Programming

- Some functions are more equal than others:

- equality, unification, mapping, zipping, folding,
- pretty printers, parsers, generators,
- G∀st: automatic test system
- Graphical User Interfaces, Web Pages
- Workflow systems
- Storage and retrieval of information from relational databases
- and many more …

- Generic Programming: define *one* function that works for *any* type !

- One defines a *generic* description of a function
- A *concrete* function is generated by the compiler given the *type*

AFP Course Scheme

I: *Mimic* Generic Programming using the *overloading* mechanism

- *more* work than writing functions by hand:
  additional definitions have to made
- but in this way we understand how it works

II: Generic Programming support offered by Clean

- less work: the compiler will generate generic instances for us

III: iTask: Task Oriented Programming (TOP)
- special flavour of FP
- Tasks as central notion
- Multi-User, Multi-Platform, distributed task coordination system
- uses a lot of Generic Programming techniques
- *example* of an Embedded Domain Specific Language (EDSL)

IV: Embedded Domain Specific Languages

Recap
Functional Programming
in
Clean

Cloogle

cloogle.org
clean.cs.ru.nl

Defining functions using patterns + guards

```
fac :: Int □ Int
fac 0                = 1
fac n
| n > 0              = n * fac (n-1)
| otherwise          = abort "factorial applied to negative
    argument“

id :: a □ a
id x       = x

(o) infixr  9 :: (b □ c) (a □ b) □ (a □ c)
(o) f g         = fog where fog x = f (g x)
```

Polymorphic functions / higher order functions / list comprehensions

```
hd :: [a] □ a
hd [x : _ ]       = x
hd _        = abort “cannot take the head of an empty list…”

map :: (a □ b) [a] □ [b]
map _ [ ]              = [ ]
map f  [x : xs]       = [f x : map f xs]

map2 :: (a □ b) [a] □ [b]
map2 f list            = [ f elem \\ elem □ list ]

myFacs :: [Int]
myFacs = map fac [1 .. 10]

myFacs2 :: [Int]
myFacs2 = take 10 (map fac [1 .. ])
```

Curried use of a function

One may use infinite structures

Polymorphic functions / higher order functions / list comprehensions

```
filter :: (a  Bool) [a]  [a]
filter _      [ ]  = [ ]
filter pred [x : xs]
| pred x               = [x : filter pred xs]
| otherwise            = filter pred xs

filter2 :: (a  Bool) [a]  [a]
filter2 pred list    = [ elem \\ elem  list | pred elem ]
```

Algebraic Data Types (ADT's)

```
:: Bool        = True
               |  False

:: List a      = Nil
               |  Cons a (List a)

:: Tree a b  = Tip a
               |  Bin b (Tree a b) (Tree a b)

:: Rose a    = Rose a (List (Rose a))
```

Overloading: different functions, same name

```
class == infix 4 a :: a a  Bool

instance == Bool
where   (==) True True  = True
        (==) False False = True
        (==) _     _     = False

instance == (List a) | == a
where   (==) Nil         Nil         = True
        (==) (Cons x xs) (Cons y ys) = x == y && xs == ys
        (==) _           _           = False
```

there must be a == for list elements

== on lists

== on list elements

Overloading: different functions, same name

```
class == infix 4 a :: a a  Bool

instance == Bool
where   (==) True True  = True
        (==) False False      = True
        (==) _     _          = False

instance == (List a) | == a
where   (==) Nil            Nil            = True
        (==) (Cons x xs)    (Cons y ys)    = x == y && xs == ys
        (==) _              _              = False

areEqual =    Nil == Nil &&
              (Cons True Nil) == (Cons True Nil) &&
              (Cons (Cons True Nil) Nil) == (Cons (Cons True Nil) Nil)
```

Overloading: different functions, same name

```
class == infix 4 a :: a a □ Bool

instance == (Tree a b) | == a & == b
where   (==) (Tip x)                (Tip y)          = x == y
        (==) (Bin x ltx rtx)   (Bin y lty rty)   = x == y && ltx == lty && rtx == rty
        (==) _                 _                 = False

instance == (Rose a) | == a
where   (==) (Rose x xs) (Rose y ys)    = x == y && xs == ys
```

Overloading allows to assign the *same* name to *different* functions.

All the functions look similar.....

How to define an overloaded *map* function ...

```
mapL :: (a  → b) (List a) → (List b)
mapL   f Nil              = Nil
mapL   f (Cons x xs) = Cons (f x) (mapL f xs)

mapR :: (a  → b) (Rose a) → (Rose b)
mapR f (Rose x xs)   = Rose (f x) (mapL (mapR f) xs)
```

How to define an overloaded *map* function ...

```
class fmap t :: (a □ b) (t a) □ (t b)
```

Curried use of a type

```
instance fmap List where
    fmap f Nil                = Nil
    fmap f (Cons x xs)        = Cons (f x) (fmap f xs)

instance fmap Rose where
    fmap f (Rose x xs)        = Rose (f x) (fmap (fmap f) xs)
```

In category theory this is defined as:

```
class Functor t where fmap t :: (a □ b) (t a) □ (t b)
```

the required laws are:

```
    fmap id            = id
    fmap (f o g)  = fmap f o fmap g
```

How to define an overloaded *map* function …

```
class fmap t :: (a □ b) (t a) □ (t b)

instance fmap List
where  fmap f Nil              = Nil
       fmap f (Cons x xs)      = Cons (f x) (fmap f xs)

instance fmap Rose
where   fmap f (Rose x xs)     = Rose (f x) (fmap (fmap f) xs)
```

A class using a *first order* type is called a type class

A class using a *higher order* type is called a type constructor class

How to define an overloaded *map* function …

```
class fmap t :: (a □ b) (t a) □ (t b)

instance fmap Tree
where  fmap f (Tip x)              = x
       fmap f (Bin x lt rt)        = Bin (f x) (fmap f lt) (fmap f rt)
```

Overloading system must be type-technically sound !

<u>The instance type and the type class variable must be of the same "kind"</u>

```
instance fmap (Tree Int)
where  fmap f (Tip x)              = x
       fmap f (Bin x lt rt)        = Bin (f x) (fmap f lt) (fmap f rt)
```

What kind of types do we have ?

```
:: List a       = Nil
                |  Cons a (List a)
:: Tree a b = Tip a
                |  Bin b (Tree a b) (Tree a b)
:: Rose a   = Rose a (List (Rose a))
```

❑Kinds specify the type of a type

•Any type for which a value exits,   is of kind “*”

Int, List Int, Rose Int, Tree Int Bool, [a] -> a, ..

•Higher order kinds * □ * … □ *

List and Rose	are of kind * □ *
Tree Int	is of kind * □ *
Tree	is of kind * □ * □ *

What *kind* of kinds are there ?

Kind terms are formed according to the grammar: *K* ::= * | (*K* □ *K*)

Kinds are right associative: * □ * □ * = * □ (* □ *)

Ordinairy Types: * Int :: *, Char :: *, Bool :: *

List Int :: *

List (Int □ Int) :: *

List (a □ a) :: *

Rose Int:: *

Tree Int Real :: *

[a] :: *

“Curried” Type Constructors: … * □ *

List :: * □ *

Tree :: * □ * □ *

Tree Int :: * □ *

Rose :: * □ *

Higer Order Types: … (* □ *) … □ *

:: T t = C (t Int) T :: (* □ *) □ *

How to determine the kind given an ADT in Clean ?

:: MyType a_1 a_2 … a_n = C_1 exp_{11} exp_{12} … exp_{1m_1}
| C_2 exp_{21} exp_{22} … exp_{2m_2}
…
| C_k exp_{k1} exp_{k2} … exp_{km_k}

First approach: as many stars as arguments, result is a value, hence is always of kind *

MyType :: $*_1$ □ $*_2$ □ … □ $*_n$ □ *

Next: every argument of any constructor C_j must be a proper type of kind * as well

Check how the arguments a_i are being used, e.g.

C_p … a_i … a_i :: *
C_q … (a_i Int) a_i :: * □ *
C_r … (T a_i) determine how a_i is used in T

Also predefined types can be used in a Curried way

```
class fmap t :: (a □ b) (t a) □ (t b)

instance fmap []
where    fmap f []       []
         fmap f [x:x      [f x : fmap f xs]


map :: (a □ b
map f []
map f [x:xs

is the same a
```

Even functions:

instance fmap ((→)r) where fmap f g = f o g

Tuples: (,), (,,). (,,), …

Arrays: {}, {#}, …

```
map :: (a □ b) [a] □ [b]
map f []            = []
map f [x:xs]        = [f x : map f xs]
```

There are many different type of *map* functions ...

```
class bmap t :: (a □ c) (b □ d) (t a b) □ (t c d)

instance bmap Tree
where    bmap f g (Tip x)             = Tip (f x)
         bmap f g (Bin x l r)   = Bin (g x) (bmap f g l) (bmap f g r)
```

One can imagine *several* maps, depending on the kind of the user type.

Overloading is “just” syntactic sugar

Overloading allows to assign the *same* name to *different* functions.

```
class (+)  infixl 6 a :: a a □ a                // Add arg1 to arg2

instance + Int
where
        (+)  infixl 6 :: Int Int □ Int
        (+)  x y = x +Int y

instance + Real
where
        (+)  infixl 6 :: Real Real □ Real
        (+)  x y = x +Real y
```

Translation of overloaded functions

```
class (+)  infixl 6 a :: a a → a

inci :: Int → Int
inci x = x + 1

incr :: Real → Real
incr x = x + 1.0

plus :: a a → a | + a
plus x y = x + y

Start :: Int
Start = plus 3 4
```

Translation by compiler:

```
inci :: Int → Int
inci x = x +Int 1

incr :: Real → Real
incr x = x +Real 1.0

plus :: (a a → a) a a → a
plus f x y = f x y

Start :: Int
Start = plus +Int 3 4
```

Translation of overloaded functions

```
class (+)  infixl 6 a:: a a □ a

inci :: Int □ Int
inci x = x + 1

incr :: Real □ Real
incr x = x + 1.0

plus :: a a □ a | + a
plus x y = x + y

Start :: Int
Start = plus 3 4
```

```
::Class+  a = { f+ ::a a □ a }

inci :: Int □ Int
inci x = x +Int 1

incr :: Real □ Real
incr x = x +Real 1.0

plus :: (Class+ a) a a □ a
plus c x y = c.f+ x y

Start :: Int
Start = plus { f+ = +Int } 3 4
```

Translation of overloaded functions

```
class (+)  infixl 6 a :: a a □ a

instance (+)  [a] | + a
where
         (+)  infixl 6 :: [a] [a] □ [a] | + a
         (+)  [x:xs] [y:ys]            = [x+y:xs+ys]
         (+)  _      _                 = [ ]


Start :: [Int]
Start = [1..5] + [6..10]
```

```
::Class+  a = { f+ ::a a □ a }



+[ ] ::(Class+ a) [a] [a] □ [a]
+[ ]  c [x:xs] [y:ys] = [c.f+ x y:+[ ] c xs ys]
+[ ]  c _      _      = [ ]


Start :: [Int]
Start = +[ ]  { f+ = +Int } [1..5] [6..10]
```

Translation of overloaded functions

```
class (+)  infixl 6 a :: a a □ a

instance (+)  [a] | + a
where
        (+)  infixl 6 :: [a] [a] □ [a] | + a
        (+)  [x:xs] [y:ys] = [x+y:xs+ys]
        (+)  _      _      = [ ]


Start :: [ [Int] ]
Start = [[1..5],[6..10]] + [[2..6],[7..11]]
```

```
::Class+  a = { f+ ::a a □ a }



+[ ] ::(Class+ a) [a] [a] □ [a]
+[ ]  c [x:xs] [y:ys] = [c.f+ x y:+[ ] c xs ys]
+[ ]  c _      _      = [ ]


Start :: [ [Int] ]
Start = +[ ]  { f+ = +[ ] { f+ = +Int } }
                    [[1..5],[6..10]] [[2..6],[7..11]]
```

Translation of overloaded functions

```
class (+)  infixl 6 a :: a a □ a

instance (+)  [a] | + a
where
        (+)  infixl 6 :: [a] [a] □ [a] | + a
        (+)  [x:xs] [y:ys] = [x+y:xs+ys]
        (+)  _      _      = [ ]


Start :: [ [Int] ]
Start = [[1..5],[6..10]] + [[2..6],[7..11]]
```

```
::Class+  a = { f+ ::a a □ a }



+[ ] ::(Class+ a) [a] [a] □ [a]
+[ ]  c [x:xs] [y:ys] = [c.f+ x y:+[ ] c xs ys]
+[ ]  c _      _      = [ ]


Start :: [ [Int] ]
Start = +[ ]  { f+ = +[ ] { f+ = +Int } }
                   [[1..5],[6..10]] [[2..6],[7..11]]
```