## Langages et types

- typage statique : Ada, O'Caml, Eiffel, Haskell
- typage dynamique : Lisp, Scheme, CLOS, Smalltalk
- typage statique ET dynamique :
  - Java 1.4, C# 1.0
  - Java 1.5, C# 2.0

tendance typage statique : sûreté + généricité

## Polymorphismes

Dans le cadre du typage statique :

- paramétrique (à la ML)
- *ad hoc* ou de surcharge
- objets ou d'inclusion

volonté d'intégration des différents polymorphismes dans les langages statiquement typés

## Inférence de types

- totale en O'Caml, Haskell
  - sauf dans les interfaces .mli
  - peut être nécessaire de l'aider
    - extension objet, ...
- partielle pour la résolution des wildcard (<?>) en Java 1.5

# Surcharge et inférence (1)

## en O'Caml : difficultés liées à l'inférence

- polymorphisme paramétrique

```
# let g o = o#m();;
val g : < m : unit -> 'a; .. > -> 'a = <fun>
# let f x y = x#m y;;
val f : < m : 'a -> 'b; .. > -> 'a -> 'b = <fun>
```

# Surcharge et inférence (2)

- application partielle :
  Soit une classe c ayant 2 méthodes :
  - `m : int -> int -> int`
  - `m : int -> float`

```
let g (o:c) (i:int) = o#m i;;
```

Quelle méthode `m` choisir?

# Surcharge et inférence - Haskell (3)

En Haskell :

- définir des types de classes regroupant des ensembles de fonctions surchargées.
- Une déclaration de classe définit une nouvelle classe et les opérateurs que celle-ci autorise.
- Une déclaration d'instance (d'une classe) indique qu'un certain type est une instance d'une classe. Cela inclue la définition des opérateurs surchargés de sa classe pour ce type.

```
class Num a where
  (+)    :: a -> a -> a
  negate :: a -> a
```

# Surcharge et inférence - Haskell (4)

On peut maintenant déclarer une instance `Int` de la classe `Num` de cette manière :

```
instance Num Int where
  x + y    =  addInt x y
  negate x = negateInt x
```

Et l'instance `Float`

```
instance Num Float where
  x + y    =  addFloat x y
  negate x = negateFloat x
```

L'application de `negate` `Num` aura un comportement différent si l'argument est de l'instance `Int` ou `Float`.

# Surcharge et inférence - Haskell (5)

type de `negate`: `Num a => a -> a`

$\Rightarrow$ : un argument supplémentaire (class restriction) à l'exécution pour effectuer le dispatch.

```
fib :: Num a, Num b => a -> b
fib n = fibGen 0 1 n

fibGen :: Num a, Num b => b -> b -> a -> b
fibGen a b n = case n of
        0 -> a
        n -> fibGen b (a + b) (n - 1)
```

# Surcharge et inférence (6)

Extension du système de types en O'Caml

- Langage reFLect (Intel) et surcharche :

```
overload print : IO.output -> 'a -> unit
overload _+_ : 'a -> 'a -> 'a

instance print Int.print Float.print
instance _+_ Int._+_ Float._+_
```

voir : `http://gallium.inria.fr/~pouillar`

## Génériques (1)

classes paramétrées :

- C++ : code spécialisé pour chaque instance de template
- O'Caml : reste dans le cadre du polymorphisme paramétrique de la couche fonctionnelle
- Java : même machine virtuelle, compatibilité ascendante, code engendré compatible avec la JVM, polymorphisme borné
- C# 2.0 : machine virtuelle avec instructions génériques
  - Design and Implementation of Generics for the .NET Common Language Runtime Andrew Kennedy and Don Syme. PLDI 2001
  - http://research.microsoft.com/ akenn/generics/index.html

## Génériques (2)

- même code pour tout type :
  - représentation uniforme des données ($\alpha$)
  - paramètre supplémentaire (class rectriction) puis dispatch
- monomorphisation : code spécialisé pour chaque instance :
  $\Rightarrow$ code plus important
- casts sûrs dans le code :
  $\Rightarrow$ code moins rapide

$\Rightarrow$ conséquences sur les performances (GC, ...)

# Intégration (1)

- Styles de programmation
  - fonctionnel(paramétrique)/objet (à la O'Caml)
  - fonctionnel(paramétrique)/*ad hoc* : fonctions génériques (à la CLOS ou à la Haskell)
  - objet/*ad hoc* : Java 1.4, C# 1.0
  - objet/*ad hoc*/paramétrique : Java 1.5, C# 2.0
- Mélange
  - F# : C# pour l'objet + caml-light pour le fonctionnel/impératif
  - C# 3.0 : $\lambda$-expressions, inférence de types (var locales)

```
x => x + 1                      // Implicitly typed, expression body
x => { return x + 1; }          // Implicitly typed, statement body
(int x) => x + 1                // Explicitly typed, expression body
(int x) => { return x + 1; }    // Explicitly typed, statement body
(x, y) => x * y                // Multiple parameters
() => Console.WriteLine()       // No parameters
```

# Intégration (2)

- Comparaison :
  - Narbel Ph : Programmation fonctionnelle, générique et objet. Vuibert. 2005.
    comparaison des structurations modules et classes pour la généricité.
- Autres langages
  - Nemerle : langage statiquement typé, fonctionnel, impératif, objet pour .NET, syntaxe à la C#, filtrage de motifs, ...
  - Scala :langage statiquement typé, fonctionnel, impératif, objet pour Java et .NET, en Java permet d'avoir accès aux bibliothèques Java directement; syntaxe à la Java, simplifie les génériques, filtrage de motifs, modularité à la mixin `http://www.scala-lang.org`

## Interopérabilité

- FFI : foreign function interface (external O'Caml, JNI Java)
- IDL : fonctions, données (COM)
- IDL : objets (exemple O'Jacaré)

*runtime* commun : facilite l'interopérabilité ( O'Jacaré.NET) pour la gestion mémoire (GC), la concurrence (threading), ...

## Typage statique ou dynamique

Tendance :

- vers le typage statique
- avec du typage dynamique pour les valeurs/programmes venant de l'extérieur (sérialisation, réseau)

Passage facilité par des nouveaux systèmes de types :

- Typed Scheme : The Design and Implementation of Typed Scheme (POPL 2008)
- Thorn : intégration of typed and untyped code in a scripting language (POPL2010)