Université de Montréal

Programmation Orientée Objet

Sous-typage

Méthodes

Interfaces

Dispatch dynamique

Héritage

Sous-typage

Aussi parfois appelé polymorphisme *ad-hoc*

Possibilité de passer des arguments de plusieurs types

Distinction entre type statique et type dynamique:

- Type dynamique = type d'une valeur
- Type statique = type d'une variable

Cohérence: *Type-Dynamique* $<$ *Type-Statique*

Type = ensemble de contraintes

Sous-typage correspond à inclusion de contraintes

Sous-typage sur les structs

```
struct parent { shape *a; t2 b; }
```

Sous-typage en *largeur*, lorsqu'un *struct* est un préfixe d'un autre:

```
struct fille  { shape *a; t2 b; t3 c; }
```

Sous-typage en *profondeur*, lorsqu'un champ est un sous-type:

```
struct fils   { square *a; t2 b; }
```

où on présume que `square` est un sous-type de `shape`

Pourquoi le sous-typage

Le sous-typage peut être vu comme résultat pragmatique!

Accès à un object de type `parent` se traduisent par

```
LOAD   Rd <- 8(Rs)        ;; x->a
```

et

```
LOAD   Rd <- 16(Rs)       ;; x->b
```

N'importe quel type où `x->a` et `x->b` se traduisent de la même manière peut être considéré un sous-type de `parent`

$\Longrightarrow$ ¡¡ Une autre traduction mène à une autre notion de sous-typage !!

[Ou vice-versa! en tout cas: *ad-hoc*!]

Covariance / Contravariance

Pour que $f_1 : A_1 \to B_1$ puisse s'utiliser
n'importe où où $f_2 : A_2 \to B_2$ est acceptée elle doit:

- accepter n'importe quel argument accepté par f_2
- renvoyer des valeurs acceptées par n'importe quel appelant de f_2

Les arguments de fonctions sont donc naturellement *contravariants*:

$$\frac{A_2 \leq A_1 \quad B_1 \leq B_2}{A_1 \to B_1 \;\leq\; A_2 \to B_2}$$

Sous-typage en profondeur et mutabilité

Le sous-typage en profondeur est incompatible avec la mutabilité:

```
struct parent { shape  *s; }
struct enfant { square *s; }

void set_s (parent *p, shape *s)
{ p->s = s; }

void problème (enfant *e, shape *s)
{ set_s (e, s); }
```

L'affectation `e->s = s;` serait (correctement) rejetée!

On dit que les champs mutables doivent être *invariants*

Interface

Une interface spécifie les contraintes d'un type

- Typiquement: liste de *méthodes* avec leurs signatures
- parfois inclus: des *champs* que l'objet doit avoir, avec leurs types

Donc, une *interface* est en fait une sorte de *type*

On dit qu'un type *implémente* une interface

- Peut se faire après-coup, séparément de la définition du type

Héritage

Souvent confondu avec le sous-typage

Technique de déclaration de nouveau type

- Type décrit par des ajouts à un type existant
- Réutilisation des champs et méthodes du type parent

Parfois la seule manière de définir un sous-type

- Ne défini pas forcément un sous-type

Controversé: Dépendance forte entre le sur-type et le sous-type

Classes

Une classe décrit un type ainsi qu'un ensemble de méthodes

- La classe et son type sont souvent utilisés indistinctement
- La classe existe aussi à l'exécution

À l'exécution, une classe est un objet qui peut contenir:

- La table des méthodes
- Un ensemble de champs: les "attributs de classe"

Étant un objet, on peut lui associer une classe: la *métaclasse*

Dispatch dynamique

Un appel de méthode choisi la méthode selon l'objet

Chaque appel peut exécuter un code différent à chaque fois

Aussi parfois appelé polymorphisme *ad-hoc*

Méthodes

Une méthode est une sorte de fonction associée à un objet:

```
x.method (arg1, arg2);
```

Exécute la méthode `method` associée à l'objet `x`

Se traduit généralement en un appel de fonction:

```
funcall (lookup (x, method),
         x, arg1, arg2)
```

Où la fonction qui implémente `method` a un argument supplémentaire nommé `self` ou `this`

Table des méthodes

La magie est donc dans: `lookup (x, method)`

Implémentation "classique":

```
lookup (x, method)  =  x->class->method
```

Chaque objet a un champ caché `class` à une position constante

`class` contient la *table des méthodes* (parfois appelée *vtable*)

Chaque méthode a une position connue d'avance dans la *vtable*

```
LOAD Rc <- 0(Rs)   ;; 0 = offset de "class"
LOAD Rf <- M(Rc)   ;; M = offset de "method"
```

Types des classes et instances

```
class A { a : int; method m1 (int); }
class B { a : int; b : float;
          method m1 (int); method m2 (string); }
```

Ces déclarations créent 4 types d'objets:

$$A\_class = (class : metaclass, m1 : int \rightarrow void)$$
$$B\_class = (class : metaclass, m1 : int \rightarrow void, m2 : string \rightarrow void)$$

$$A\_inst = (class : A\_class, a : int)$$
$$B\_inst = (class : B\_class, a : int, b : float)$$

$B\_class \leq A\_class$ (par *width subtyping*)

$B\_inst \leq A\_inst$ (par *width & depth subtyping*)

Types des classes et instances (corr.)

```
class A { a : int; method m1 (int); }
class B { a : int; b : float;
          method m1 (int); method m2 (string); }
```

Ces déclarations créent 4 types d'objets:

$$A\_class = (class : metaclass, m1 : A\_inst \rightarrow int \rightarrow void)$$
$$B\_class = (class : metaclass, m1 : B\_inst \rightarrow int \rightarrow void, \ldots)$$

$$A\_inst = (class : A\_class, a : int)$$
$$B\_inst = (class : B\_class, a : int, b : float)$$

$B\_class \not\leq A\_class$ (pour raisons de contravariance)

$B\_inst \leq A\_inst$ (par ¿magie?)

Classes de type

Programmation orientée objet en Haskell

Cousin des *interfaces* de Java:

```
class Drawable a where
  draw :: Display → a → IO ()
```

Spécifie les méthodes que les instances doivent fournir

Peut fournir des implémentation par défaut

```
draw :: Drawable a ⇒ Display → a → IO ()
```

Instances de classes de type

Les instances sont définies séparément du type correspondant:

```
instance Drawable Square where
  draw :: Display → Square → IO ()
  draw s d = ...

instance Drawable Circle where
  draw :: Display → Circle → IO ()
  draw c d = ...
```

On peut donc les définir longtemps après avoir défini *Square*

Exemple: listes numériques

```
instance Num a => Num [a] where
  x + y = zipWith (+) x y
  x * y = zipWith (*) x y
  negate x = map negate x
  signum x = map signum x
  abs    x = map abs x
  fromInteger n = nlist
    where nlist = fromInteger n : nlist
```

et ainsi:

$$[1,2,3] + 6 \implies [7,8,9]$$

$$6 - [[1,2,3],[42,43]] \implies [[5,4,3],[-36,-37]]$$

Contraintes sur les types

Les classes sont des contraintes sur les types

```
dist :: (Ord a, Num a) => a -> a -> a
dist x y = if x > y then x - y else y - x
```

Une classe de type peut hériter d'un (ou plusieurs) autres

```
classe (Ord a, Num a) => OrdNum a where
  ...
```

alors

```
dist :: OrdNum a => a -> a -> a
dist x y = if x > y then x - y else y - x
```

Contraintes sur des types “impropres”

`Int` et `Maybe Int` sont des *proper types*

`Maybe` est un *constructeur de type*: sorte de fonction sur les types

Les classes de types se généralisent aux *constructor classes*

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor [] where
  fmap = map

instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

Délégation

Champs dans les instances, méthodes dans la classe $\Rightarrow$ trop restrictif!

Remplace le champ caché `class` par le champ caché `parent`

```
lookup (x, method) =
  if (method existe dans x)
      return x.method;
  else
      lookup (x->parent, method);
```

Et ce même `lookup` est utilisé pour les *champs*!

$\Rightarrow$ Champs et méthodes: même combat!

Unifie aussi classes et instances!

Introduit par Self, popularisé par Javascript!

Table dans la méthode: Multi-méthodes

Autre définition possible de `lookup`:

```
lookup (x, method) =
   gethash (method->table, typeof (x))
```

Utilisé en Common Lisp: `(method x arg1 arg2)`

Se généralise à plusieurs arguments:

```
(defmethod add ((x int)    (y int))    ...)
(defmethod add ((x string) (y string)) ...)
(defmethod add ((x string) y)          ...)
```

Plus besoin d'un champ *class* ou *parent*: compatible avec `Int`!

Table de méthodes flottante: Type classes

Haskell manipules ses vtables (*dictionnaires*) séparément des objets!

```
min :: Ord a => a -> a -> a
min x y | x < y = x
        | otherwise = y
```

Est transformé par le compilateur en

```
min :: OrdDict a -> a -> a -> a
min dict x y | dict.< x y = x
             | otherwise = y
```

Note: le dictionnaire n'appartient ni à `x` ni à `y`

```
sum :: Num a => [a] -> a
```

Un seul dictionnaire!

Avantages/inconvénients des types classes

Pas besoin de champs *class* dans les objets

- Applicable aux *Int*, *Bool*, aux types fonctions, ...

Bonne interaction avec l'inférence de types

Dispatch sans valeur: `read :: Read a => String -> a`

Impossible d'avoir deux méthodes *différentes* avec le même nom

`Drawable a => [a]` est une liste homogène!

- Pour obtenir des structures hétérogènes, il faut passer par

```
data DrawableElem =
    | Elem forall a . Drawable a => a
```

qui crée des paquets qui combinent un *a* avec un *DictDrawable a*