Langages de Programmation, Interprétation, Compilation

Christine Paulin
Christine.Paulin@universite-paris-saclay.fr

Département Informatique, Faculté des Sciences d'Orsay, Université Paris-Saclay

LDD3 Informatique, Mathématiques - Magistère Informatique

2025–26

Sémantique d'un langage orienté objet

Paradigme de programmation objet

- le code est organisé dans des classes
- l'exécution d'un programme est centrée sur l'interaction d'un certains nombre d'objets
- un objet est une petite structure de données, liée à une classe.
- paradigme présent dans de nombreux langages de programmation, parfois central comme en C++ et java.
- ici on étudiera les principes de la programmation objet en prenant le modèle de java.

Partie 4 : Programmation Objet

Classes, objets : définitions

- En programmation, un objet est une petite structure de données
- à laquelle sont associées des fonctions spécifiques : ses méthodes.
- La forme et les méthodes d'un objet sont définies par une classe.

Exemple

La déclaration d'une classe introduit un nouveau type (comme un enregistrement).

```
class Point {
  int x;
  int y;
}
```

Ici, x et y sont les deux champs de la classe Point (attributs).

Classes, attributs, instances

- Chaque champ est identifié par un nom, et est associé à un type.
- Une instance d'une classe C, appelée un objet, est une structure possédant une valeur pour chaque champ de C.
- Cette structure est stockée en mémoire (sur le tas) et accessible par un pointeur.
- Son type est précisément C, le type défini par la classe C.

Création d'un objet

Opérateur de construction associé au nom de la classe

```
Point p = new Point();
```

- Ce code définit une nouvelle variable p, de type Point,
- la valeur de p est une nouvelle instance de la classe Point.
- Les champs de cette nouvelle instance reçoivent une valeur par défaut (en java, 0).

Accès aux champs

Opérateur . (point), syntaxe `o.f` avec `o` un objet et `f` un nom de champ.

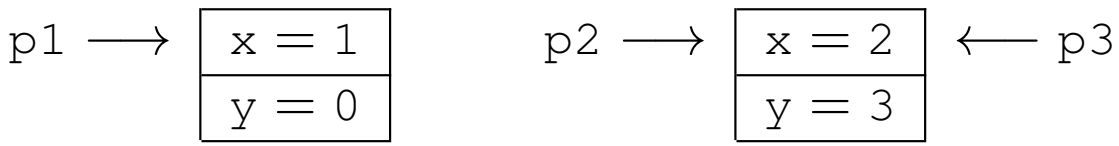
```
p.x = 1 + p.y;
```

- `o.f` est qualifiée d'expression gauche,
- peut apparaître à gauche d'une affectation (représente alors une adresse)
- utilisée dans une expression ordinaire : valeur trouvée à l'adresse
- une variable, un accès à un tableau `t[3]` sont aussi des expressions gauches

Accès aux champs

- Deux instances d'une même classe sont des structures indépendantes.
- Chacune a ses propres valeurs pour chaque champ et évolue indépendamment de l'autre.
- On peut aussi avoir plusieurs pointeurs vers un même objet !

```
Point p1 = new Point();
Point p2 = new Point();
p1.x = 1;
p2.x = 2;
Point p3 = p2;
p3.y = 3;
```



Constructeur

- Une classe définit des champs + code applicable aux instances.
- constructeur : fonction pour “initialiser” l’objet

```
class Circle {
  Point center;
  int radius;

  Circle(Point c, int r) {
    if (r < 0) throw new Error("Circle: negative radius");
    center = c;
    radius = r;
  }
}
```

- ajout de paramètres, permettant de préciser les valeurs initiales
- le code peut vérifier la validité des valeurs initiales.
- la fonction de construction est appelée à la création d’un nouvel objet

Constructeurs : exemple

- on suppose avoir un constructeur pour la classe Point

```
Circle c = new Circle(new Point(1, 2), 3);
```

- introduit une nouvelle variable c dont la valeur est une instance de Circle,
- le premier champ est lui-même une nouvelle instance de Point.

Méthodes

- La classe contient des définitions de méthodes,
- fonctions applicables aux instances de cette classe.
- les méthodes avoir des paramètres et/ou renvoyer un résultat.

```
class Point {
  ...
  void move(int dx, int dy) {
    x += dx;
    y += dy;
  }
  Point copy() {
    return new Point(x, y);
  }
  int sqDistTo(Point q) {
    return (x-q.x)*(x-q.x) + (y-q.y)*(y-q.y);
  }
```

Appel de méthodes

- o.m appelle la méthode m à l'objet o de la classe correspondante
- avec p une instance de la classe Point

```
Point q = p.copy();
p.move(2, -1);
int d = p.sqDistTo(q);
```

- l'appel de méthode `p.move(2, -1)` est analogue à l'appel `mv(p, 2, -1)` d'une fonction ordinaire `mv` définie par

```
void mv(Point p,int dx,int dy)
    { p.x += dx; p.y += dy; }
```

- l'instance p est appelée le paramètre implicite de l'appel.
- x et y font référence aux champs de cette instance
- un appel de méthode p.f (...) peut consulter et/ou de modifier les champs de l'objet p.

Paramètres

- le mot-clé `this` dans le code d'une méthode ou d'un constructeur désigne le paramètre implicite.

```
void move(int dx, int dy) {
  this.x = this.x + dx;
  this.y += dy;
}
```

- les paramètres « ordinaires » 2 et –1 sont les paramètres explicites.
- sqDistTo :
  - simple accès x, sous-entendu `this.x`, au champ x du paramètre implicite,
  - accès q.x au champ x du paramètre explicite q.
- l'emploi de `this` peut parfois être obligatoire (ambiguïté avec une autre variable nommée x).

```
Point(int x, int y) {
  this.x = x;
  this.y = y;
}
```

Champs et méthodes statiques

- certains champs ne sont pas liés aux instances mais à la classe elle-même.
- une seule valeur, partagée à l'échelle de la classe (variable globale).
- ```
  class Point {
    int x, y;
    static int max = 1024;
  ```
- accès depuis l'intérieur de la classe avec le nom (max),
- accès depuis l'extérieur en ajoutant le nom de la classe (Point.max).
- Les méthodes peuvent aussi être statiques.
  - fonctions ordinaire
  - pas de paramètre implicite
  - pas de référence à this ou à des champs ordinaires.
  - peut accéder aux champs statiques

```
static boolean inBound(int k)
        { return 0 <= k && k < max; }
```
```

Encapsulation

- Une classe peut être associée à une invariant
- Exemple : radius positif ou nul dans Circle
- Garantir l'invariant :
  - toutes les méthodes laissent des objets qui vérifient l'invariant
  - aucun code externe ne peut modifier directement radius sans passer par les méthodes de la classe

```
class Circle {
    Point center;
    private int radius;
    Circle(Point c, int r) { ... }
```

- introduire si besoin une méthode de consultation

```
    int getRadius() { return this.radius; }
```

Espaces de noms

Une classe délimite un espace de noms :

- les noms des attributs et méthodes d'une classe sont indépendants de ceux utilisés dans les autres classes.
- exemple : classe GraphicElt désignant un élément graphique ancré à un point donné, et déplaçable par une méthode move.

```
class GraphicElt {
    Point anchor;
    void move(int dx, int dy) { anchor.move(dx, dy); }
```

- deux méthodes move
- pas d'ambiguïté : le paramètre implicite de l'appel à move est associé à une classe qui détermine l'unique méthode pertinente.

Génie logiciel

encapsulation et d'espace de noms

- séparent le fonctionnement interne d'une structure de données et son utilisation par un code tiers
- éléments centraux de génie logiciel
- réalisés de différentes manières dans d'autres langages.
- En caml par exemple, on utiliserait le système de modules et les types abstraits

Une classe trop générale ?

- ajouter à la classe GraphicElt une méthode déterminant si l'élément contient un point donné.
- Sans connaissance sur la forme des éléments, on ne sait pas écrire une telle méthode dans la classe GraphicElt.

```
boolean contains(Point p)
        { throw new Error("Passe à Kevin"); }
```

Partie 4 : Programmation Objet

Object-oriented features

- Dériver de la classe C une nouvelle classe D, qui hérite de tous les champs de C et de toutes les méthodes définies dans C.
- La classe D peut introduire de nouveaux champs et de nouvelles méthodes.

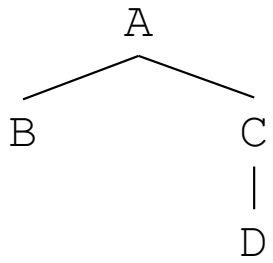
```
class D extends C { ... }
```

- La classe dérivée D décrit un cas particulier de C (tout ce qui existe pour C+éléments spécifiques.)
- D est une sous-classe, ou classe fille, ou encore spécialisation de la classe C,
- la classe C est symétriquement la super-classe, classe mère, ou généralisation de D.

Hiérarchie de classes

- en java, chaque classe hérite au maximum d'une autre classe (héritage simple).
- C++ et python autorisent l'héritage multiple
- un ensemble de classes et leur relation d'héritage est un arbre (ou forêt) (hiérarchie de classes).

```
class A { ... }
class B extends A { ... }
class C extends A { ... }
class D extends C { ... }
```



Spécialisation

- redéfinir un cercle comme un cas particulier d'élément graphique, dont l'ancre est le centre.
- Cette nouvelle classe hérite du champ anchor et de la méthode move de GraphicElt.
- on ajoute le champ spécifique radius ainsi qu'un constructeur.

```
class Circle extends GraphicElt {
    int radius;
    Circle(Point c, int r) {
      if (r < 0) throw new Error("Circle: negative radius");
      anchor = c.copy();
      radius = r;
    }
      ...
```

Redéfinition

- on peut coder la méthode contains
- on redéfinit la méthode (override),
- pour les instances de Circle, ce code remplace celui hérité de GraphicElt.
- annotation (optionnelle) `@Override` : le compilateur vérifie que la signature correspond à une méthode de la classe mère.

```
    boolean contains(Point p) {
      return anchor.sqDistTo(p) <= radius * radius;
    }
}
```

Exemple : suite

- rectangle ancré par son coin inférieur gauche et caractérisé par une largeur et une hauteur.
- définition

```
class Rectangle extends GraphicElt {
  int width, height;
  Rectangle(Point p1, int w, int h) {
    anchor = new Point(p1.x,p1.y);
    width  = w;
    height = h;
  }
  Rectangle(Point p1, Point p2) {
    anchor = new Point(Math.min(p1.x, p2.x), Math.min(p1
    width  = Math.abs(p1.x-p2.x);
    height = Math.abs(p1.y-p2.y);
  }
  boolean contains(Point p) {
    return anchor.x <= p.x && p.x <= anchor.x + width
    && anchor.y <= p.y && p.y <= anchor.y + height;
  }
}
```

Sous-typage

- tout objet de Circle ou Rectangle peut être considéré comme un objet de la classe mère GraphicElt.
- ces objets possèdent les caractéristiques d'un élément graphique (champ anchor, méthodes move, contains),
- ils peuvent être utilisés sans risque là où on attend une instance de GraphicElt.

```
Point p = new Point(1, 0);
Point q = new Point(0, 0);
GraphicElt elt1 = new Circle(p, 1);
GraphicElt elt2 = new Rectangle(p, 1, 1);
if (elt1.contains(q)) { ... }
if (elt2.contains(q)) { ... }
```

- Les types définis par les classes Circle et Rectangle sont des sous-types du type défini par la classe GraphicElt.

Type statique, type dynamique

- la construction **new** Circle(p, 1) construit un objet de la classe Circle.
  - cette classe détermine la forme concrète de l'objet en mémoire
  - la classe est fixée définitivement à sa création.
  - c'est le type type réel de l'objet, appelé type dynamique (connu à l'exécution !)
- la déclaration GraphicElt elt1 introduit une variable elt1 de type GraphicElt.
  - c'est le type apparent de elt1, appelé type statique, seule information connue du compilateur.
  - un objet de type Circle peut être considéré comme de type GraphicElt.
  - l'objet produit par **new** Circle(p, 1) est une valeur légitime pour la variable elt1, de type statique GraphicElt.
- L'appel elt1.contains(...) est accepté car GraphicElt, le type statique de elt1, déclare une méthode contains.
- la méthode contains appelée est celle de Circle, classe réelle (type dynamique) de l'objet.

Polymorphisme, liaison dynamique

- nouvel élément graphique Group : plusieurs éléments graphiques.
- classe GList : listes chaînées d'éléments graphiques.

```
class GList {
    GraphicElt elt;
    GList next;
    GList(GraphicElt g, GList n) { elt = g; next = n; }
      }
```

- constructeur polymorphe : traite des entrées de plusieurs types différents

```
Point p = new Point(0, 0);
Point q = new Point(1, 1);
Circle c = new Circle(q, 1);
Rectangle r = new Rectangle(p, 2, 2);
GList quadrature = new GList(c, new GList(r, null));
```

Polymorphisme, liaison dynamique

Définition de la classe Group

```
class Group extends GraphicElt {
    private GList group;
    Group(Point a) { anchor = a; group = null; }

    void add(GraphicElt g) {
       group = new GList(g, group);
     }
```

Redéfinition de la méthode contains

```
boolean contains(Point p) {
       for (GList l = group; l != null; l = l.next)
         if (l.elt.contains(p)) return true;
       return false;
     }
```

- le compilateur ne peut pas connaître le type dynamique de l'élément `l.elt` (donc quel code utiliser)
- le choix du code se fait à l'exécution : liaison dynamique.

Redéfinir, tout en réutilisant

- Group hérite de la méthode move de GraphicElt.
- move déplace l'ancre associée au groupe, on voudrait déplacer chaque élément du groupe.
- redéfinir une méthode move pour Group qui reste polymorphe sur les éléments
- utilise la méthode move de la classe mère, notée super.move, avec une boucle sur la liste qui appelle la méthode move de chaque élément.

```
void move(int dx, int dy) {
    super.move(dx, dy);
    for (GList l = group; l != null; l = l.next)
        l.elt.move(dx, dy);
  }
```

Abstraction

- la classe GraphicElt n'est pas instanciée directement
- cadre général pour des éléments concrets : Circle, Rectangle et Group
- classe abstraite, pas de code pour les méthodes

```
abstract class GraphicElt {
      Point anchor;
      void move(int dx, int dy) { anchor.move(dx, dy); }
      abstract boolean contains(Point p);
      }
```

chaque sous-classe concrète de GraphicElt doit redéfinir la méthode contains.

Et la surcharge ?

- Java permet une surcharge statique (indépendant du paradigme objet).
- Utilisé pour les constantes et opérateurs arithmétiques
- plusieurs méthodes de même nom dans un même espace de noms (classe).
- distinction par la signature (nombre et types des arguments, type de retour)

```
boolean contains(Point p) { ... }
boolean contains(int x, int y) { ... }
```

- Le nom partagé contains est une facilité d'écriture, en interne deux noms différents
- Résolution à la compilation si le choix est sans ambiguïté.
- Difficulté en présence de sous-typage : la classe D étend la classe C, les deux méthodes suivantes ne peuvent pas cohabiter :

```
C combine(C a, D b) { ... }
C combine(D a, C b) { ... }
```

Partie 4 : Programmation Objet

1. Sémantique d'un langage orienté objet
   - Classes, objets, méthodes
   - Héritage, sous-typage, liaison dynamique
   - Formalisation du sous-typage
   - Vérification des types en présence de sous-typage
   - Sous-typage au-delà des objets
   - Comparaison avec la programmation fonctionnelle
   - Comprenez-vous la liaison dynamique ?

Formalisation du sous-typage

- Relation de sous-typage
- interaction avec les règles de typage

Types en présence d'objets

- Deux formes de types :
  - types de base : `int` ou `boolean`,
  - types définis par des classes, notés par un nom C.

$$\tau \quad ::= \quad \texttt{int} \mid \texttt{boolean} \mid C$$

- un type de classe C est lié à une classe `c` particulière, possédant des attributs et des méthodes.
  - association entre des identifiants (noms d'attributs ou de méthodes) et des informations de typage.
  - attribut `x` de C : $C(\texttt{x})$ est le type de `x`.
  - méthode `f` de C : $C(\texttt{f})$ la signature de `f`.
- signature d'une méthode avec n paramètres de types τ_1 à τ_n, et un résultat de type τ : $(\tau_1 \times \ldots \times \tau_n) \to \tau$.
- méthode ne renvoyant pas de résultat : signature $(\tau_1 \times \ldots \times \tau_n) \to \texttt{void}$.

Exemple de la classe `Point`

Type $C_{\mathtt{p}}$, avec les équations :

$$
\begin{array}{rcl}
C_{\mathtt{p}}(\mathtt{x}) & = & \mathtt{int} \\
C_{\mathtt{p}}(\mathtt{y}) & = & \mathtt{int}
\end{array}
\qquad
\begin{array}{rcl}
C_{\mathtt{p}}(\mathtt{move}) & = & (\mathtt{int} \times \mathtt{int}) \to \mathtt{void} \\
C_{\mathtt{p}}(\mathtt{copy}) & = & () \to C_{\mathtt{p}} \\
C_{\mathtt{p}}(\mathtt{sqDistTo}) & = & (C_{\mathtt{p}}) \to \mathtt{int}
\end{array}
$$

Héritage

- fonction (partielle) `parent` sur les types de classes.
  - Si une classe [de nom `d` et de type] D hérite d'une classe [de nom `c` et de type] C, alors $\texttt{parent}(D) = C$.
- propriété de cohérence sur les types de classes.
  - Si $\texttt{parent}(D) = C$ et si $C(\texttt{id})$ est défini, alors $D(\texttt{id})$ est défini et $D(\texttt{id}) = C(\texttt{id})$.
- La classe GraphicElt associée à un type $C_\texttt{g}$, et les classes Circle et Rectangle à des types $C_\texttt{c}$ et $C_\texttt{r}$ tels que, par exemple :

$$\begin{aligned} \texttt{parent}(C_\texttt{c}) = \texttt{parent}(C_\texttt{r}) &= C_\texttt{g} \\ C_\texttt{g}(\texttt{anchor}) = C_\texttt{c}(\texttt{anchor}) = C_\texttt{r}(\texttt{anchor}) &= C_\texttt{p} \\ C_\texttt{g}(\texttt{contains}) = C_\texttt{c}(\texttt{contains}) = C_\texttt{r}(\texttt{contains}) &= (C_\texttt{p}) \to \texttt{boolean} \\ C_\texttt{c}(\texttt{radius}) &= \texttt{int} \\ C_\texttt{r}(\texttt{width}) &= \texttt{int} \end{aligned}$$

Relation de sous-typage

- Le jugement de sous-typage $\vdash \tau_1 <: \tau_2$ exprime que le type τ_1 est un sous-type de τ_2
- Il est valide si les deux types sont égaux, ou si τ_1 est une classe héritant directement ou indirectement de τ_2.
- Vocabulaire alternatif
  - τ_1 est subsumé par τ_2, c'est-à-dire inclus dans τ_2
  - τ_2 subsume τ_1.
  - τ_2 est un sur-type (ou super-type) de τ_1,

Règles d'inférence

clôture réflexive-transitive de la relation d'héritage.

$$\frac{}{\vdash \tau <: \tau} \qquad \frac{\texttt{parent}(D) = C}{\vdash D <: C} \qquad \frac{\vdash \tau_1 <: \tau_2 \qquad \vdash \tau_2 <: \tau_3}{\vdash \tau_1 <: \tau_3}$$

Règles de typage des expressions

- jugement de typage $\Gamma \vdash e : \tau$
- l'expression e est cohérente et de type τ dans le contexte Γ.
- le contexte Γ associe les identifiants de variables aux types déclarés pour ces variables
- règles déjà vues :

$$\frac{\Gamma(\texttt{x}) = \tau}{\Gamma \vdash \texttt{x} : \tau} \qquad \frac{}{\Gamma \vdash \texttt{n} : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \ \texttt{-} \ e_2 : \texttt{int}}$$

Règles pour le sous-typage et les objets

- règle liant typage et sous-typage : subsomption

$$\frac{\Gamma \vdash e : \sigma \qquad \vdash \sigma <: \tau}{\Gamma \vdash e : \tau}$$

- le contexte Γ associe également chaque nom de classe à son type
- opérateur **new** de construction d'un nouvel objet avec un constructeur sans argument

$$\frac{\Gamma(\mathtt{c}) = C}{\Gamma \vdash \mathtt{new\ c()} : C}$$

- accès $e.\mathtt{x}$ à un attribut est cohérent si l'expression e a un type de classe possédant un attribut $\mathtt{x}$.

$$\frac{\Gamma \vdash e : C \qquad C(\mathtt{x}) = \tau}{\Gamma \vdash e.\mathtt{x} : \tau}$$

Règles pour les méthodes

Appel de méthode (avec résultat $\neq$void) $e.\texttt{f}(e_1, \ldots, e_n)$ cohérent si la classe de e a une méthode `f`, et si les paramètres $e_1, \ldots, e_n$ ont les types attendus.

$$\frac{\Gamma \vdash e : C \qquad C(\texttt{f}) = (\tau_1 \times \ldots \times \tau_n) \to \tau \qquad \forall i, (\Gamma \vdash e_i : \tau_i)}{\Gamma \vdash \texttt{e.f(}e_1\texttt{, ..., }e_n\texttt{)} : \tau}$$

Règles de typage des instructions

- affectation d'une variable

$$\frac{\Gamma(x) = \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x \texttt{ = } e\texttt{;}}$$

- affectation d'un champ $e_1\texttt{.x = } e_2\texttt{;}$

$$\frac{\Gamma \vdash e_1 : C \qquad C(x) = \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\texttt{.x = } e_2\texttt{;}}$$

- méthode sans résultat

$$\frac{\Gamma \vdash e : C \qquad C(\texttt{f}) = (\tau_1 \times \ldots \times \tau_n) \to \texttt{void} \qquad \forall i, (\Gamma \vdash e_i : \tau_i)}{\Gamma \vdash \texttt{e.f(}e_1\texttt{, ..., }e_n\texttt{);}}$$

Exemple de dérivation

```
GraphicElt elt = new Circle();
```

Environnement Γ tel que $\Gamma(\texttt{elt}) = C_g$,

$$\dfrac{\Gamma(\texttt{elt}) = C_g \qquad \dfrac{\dfrac{}{\Gamma \vdash \texttt{new Circle()} : C_c} \qquad \dfrac{\texttt{parent}(C_c) = C_g}{\vdash C_c <: C_g}}{\Gamma \vdash \texttt{new Circle()} : C_g}}{\Gamma \vdash \texttt{elt = new Circle();}}$$

Partie 4 : Programmation Objet

1. Sémantique d'un langage orienté objet
   - Classes, objets, méthodes
   - Héritage, sous-typage, liaison dynamique
   - Formalisation du sous-typage
   - Vérification des types en présence de sous-typage
   - Sous-typage au-delà des objets
   - Comparaison avec la programmation fonctionnelle
   - Comprenez-vous la liaison dynamique ?

Vers un algorithme de typage

Quelle règle appliquer dans quelle situation ?

- La règle de transitivité

$$\frac{\vdash \tau_1 <: \tau_2 \qquad \vdash \tau_2 <: \tau_3}{\vdash \tau_1 <: \tau_3}$$

  quel intermédiaire τ_2 ?
- La règle de subsomption

$$\frac{\Gamma \vdash e : \sigma \qquad \vdash \sigma <: \tau}{\Gamma \vdash e : \tau}$$

  ne dépend pas de la forme de e analysée, introduit un type σ
- Comment décider quand utiliser ces règles ? avec quels types ?

La liberté autorise la diversité

Plusieurs dérivations possibles du même jugement

- 3 types de classes $C_{\mathtt{a}}$, $C_{\mathtt{b}}$ et $C_{\mathtt{c}}$,
- $\mathtt{parent}(C_{\mathtt{a}}) = C_{\mathtt{b}}$, $\mathtt{parent}(C_{\mathtt{b}}) = C_{\mathtt{c}}$
- $C_{\mathtt{a}}(\mathtt{f}) = C_{\mathtt{b}}(\mathtt{f}) = (C_{\mathtt{c}}) \to \mathtt{void}$,
- expression e telle que $\vdash e : C_{\mathtt{a}}$,
- validité d'un appel $e.\mathtt{f}(e)$

$$\dfrac{\Gamma \vdash e : C_{\mathtt{a}} \qquad \dfrac{\dfrac{\mathtt{parent}(C_{\mathtt{a}}) = C_{\mathtt{b}}}{\vdash C_{\mathtt{a}} <: C_{\mathtt{b}}} \qquad \dfrac{\mathtt{parent}(C_{\mathtt{b}}) = C_{\mathtt{c}}}{\vdash C_{\mathtt{b}} <: C_{\mathtt{c}}}}{\vdash C_{\mathtt{a}} <: C_{\mathtt{c}}}}{\Gamma \vdash e : C_{\mathtt{c}}} \qquad e : C_{\mathtt{a}} \qquad C_{\mathtt{a}}(\mathtt{f}) = (C_{\mathtt{c}}) \to \mathtt{void}}{\Gamma \vdash \mathtt{e.f(e);}}$$

Autre dérivation

$$\dfrac{\dfrac{(1)}{\Gamma \vdash e : C_b} \quad C_b(\texttt{f}) = (C_c) \to \texttt{void} \quad \dfrac{(2)}{\Gamma \vdash e : C_c}}{\Gamma \vdash \texttt{e.f(e);}}$$

(1)
$$\dfrac{\Gamma \vdash e : C_a \qquad \dfrac{\texttt{parent}(C_a) = C_b}{\vdash C_a <: C_b}}{\Gamma \vdash e : C_b}$$

(2)
$$\dfrac{\dfrac{\Gamma \vdash e : C_a \qquad \dfrac{\texttt{parent}(C_a) = C_b}{\vdash C_a <: C_b}}{\Gamma \vdash e : C_b} \qquad \dfrac{\dfrac{}{\vdash C_b <: C_b} \qquad \dfrac{\texttt{parent}(C_b) = C_c}{\vdash C_b <: C_c}}{\vdash C_b <: C_c}}{\Gamma \vdash e : C_c}$$

Stratégie de typage

- encadrer l'utilisation de la transitivité et de la subsomption
- variante plus contrainte des règles d'inférence, appelée système de typage algorithmique
- preuve d'équivalence des deux systèmes (on accepte les mêmes programmes)
- algorithme déduit du système contraint

Sous-typage algorithmique

Pour distinguer cette version de la précédente, on note $\vdash_a \tau_1 <: \tau_2$ le jugement de sous-typage obtenu en appliquant le système algorithmique.

$$\frac{}{\vdash_a \tau <: \tau} \qquad \frac{\texttt{parent}(D) = C \qquad \vdash_a C <: \tau}{\vdash_a D <: \tau}$$

- Le système de sous-typage de référence et le système de sous-typage algorithmique sont en ce sens équivalents : dérivent les mêmes jugements de sous-typage
- référence : spécification, algorithmique : implémentation
  - le système algorithmique est correct : tout jugement dérivé est correct
  - le système algorithmique est complet : il permet de dériver tous les jugements corrects

Correction

toute relation de sous-typage dérivable dans le système restreint (algorithmique) est également dérivable dans le système de référence.

Lemme de correction.

$$\text{Si} \qquad \vdash_a \sigma <: \tau \qquad \text{alors} \qquad \vdash \sigma <: \tau$$

Preuve par induction sur la dérivation de $\vdash_a \sigma <: \tau$.

- Cas $\vdash_a \tau <: \tau$. Alors on a bien $\vdash \tau <: \tau$.
- Cas $\vdash_a D <: \tau$ avec $\texttt{parent}(D) = C$ et $\vdash_a C <: \tau$. Par hypothèse de récurrence on a $\vdash C <: \tau$. On construit alors la dérivation suivante :

$$\dfrac{\dfrac{\texttt{parent}(D) = C}{\vdash D <: C} \qquad \vdash C <: \tau}{\vdash D <: \tau}$$

□

Complétude

- le système algorithmique suffit à dériver toute relation de sous-typage valide
- la propriété de transitivité est admissible dans le système algorithmique

Lemme de transitivité.

Si $\vdash_a \tau_1 <: \tau_2$ et $\vdash_a \tau_2 <: \tau_3$ alors $\vdash_a \tau_1 <: \tau_3$

Preuve par induction sur la dérivation de $\vdash_a \tau_1 <: \tau_2$.

- Si $\vdash_a \tau_1 <: \tau_2$ car $\tau_1 = \tau_2$, alors en effet $\vdash_a \tau_1 <: \tau_3$.
- Si $\vdash_a \tau_1 <: \tau_2$ avec $\tau_1 = C$ et $\vdash_a \texttt{parent}(C) <: \tau_2$, alors par hypothèse d'induction $\vdash_a \texttt{parent}(C) <: \tau_3$, et donc $\vdash_a C <: \tau_3$. □

Preuve de la complétude

Lemme de complétude.

$$\text{Si} \qquad \vdash \sigma <: \tau \qquad \text{alors} \qquad \vdash_a \sigma <: \tau$$

Preuve par induction sur la dérivation de $\vdash \sigma <: \tau$.

- Cas $\vdash \tau <: \tau$. Alors en effet $\vdash_a \tau <: \tau$.
- Cas $\vdash D <: C$ avec $\texttt{parent}(D) = C$. Alors on construit la dérivation

$$\frac{\texttt{parent}(D) = C \qquad \overline{\vdash_a C <: C}}{\vdash_a D <: C}$$

- Cas $\vdash \tau_1 <: \tau_3$ avec $\vdash \tau_1 <: \tau_2$ et $\vdash \tau_2 <: \tau_3$ pour un certain type τ_2. Alors par hypothèses d'induction on a $\vdash_a \tau_1 <: \tau_2$ et $\vdash_a \tau_2 <: \tau_3$, et par le lemme de transitivité précédent on conclut $\vdash_a \tau_1 <: \tau_3$. □

Typage algorithmique

- On note $\Gamma \vdash_a e : \tau$ et $\Gamma \vdash_a i$ les jugements de typage obtenus avec ces règles restreintes.
- Restreindre l'utilisation de la subsomption immédiatement au-dessus d'une règle imposant un type précis à l'une des expressions analysées.
  - typage des appels de méthode
  - règles d'affectation
- la règle de subsomption disparait
- les autres règles sont identiques mais portent sur les nouveausx jugements

Nouvelles règles

- affectations `x = e;` et e_1`.x = `e_2`;`

$$\frac{\Gamma(x) = \tau \qquad \Gamma \vdash_a e : \sigma \qquad \vdash_a \sigma <: \tau}{\Gamma \vdash_a x = e;}$$

$$\frac{\Gamma \vdash_a e_1 : C \qquad C(x) = \tau \qquad \Gamma \vdash_a e_2 : \sigma \qquad \vdash_a \sigma <: \tau}{\Gamma \vdash_a e_1\texttt{.x = } e_2\texttt{;}}$$

- appel de méthode

$$\frac{\Gamma \vdash_a e : C \qquad C(\texttt{f}) = (\tau_1 \times \ldots \times \tau_n) \to \tau \qquad \forall i, (\Gamma \vdash_a e_i : \sigma_i \wedge \vdash_a \sigma_i <: \tau_i)}{\Gamma \vdash_a \texttt{e.f(}e_1\texttt{, ..., }e_n\texttt{)} : \tau}$$

$$\frac{\Gamma \vdash_a e : C \qquad C(\texttt{f}) = (\tau_1 \times \ldots \times \tau_n) \to \texttt{void} \qquad \forall i, (\Gamma \vdash_a e_i : \sigma_i \wedge \vdash_a \sigma_i <: \tau_i)}{\Gamma \vdash_a \texttt{e.f(}e_1\texttt{, ..., }e_n\texttt{)}}$$

Équivalence entre les deux systèmes de types

Lemme de correction.

Si	$\Gamma \vdash_a e : \tau$	alors	$\Gamma \vdash e : \tau$
Si	$\Gamma \vdash_a i$	alors	$\Gamma \vdash i$

La preuve, similairement à la précédente, se fait sans grande difficulté par induction sur la dérivation de $\Gamma \vdash_a e : \tau$ ou de $\Gamma \vdash_a i$.

Complétude

- Il est faux que si $\Gamma \vdash e : \tau$ alors $\Gamma \vdash_a e : \tau$
- décalage possible entre le type obtenu algorithmiquement et le type τ.

Lemme de complétude.

Si	$\Gamma \vdash e : \tau$	alors	il existe σ sous-type de τ tel que $\Gamma \vdash_a e : \sigma$
Si	$\Gamma \vdash i$	alors	$\Gamma \vdash_a i$

Preuve par induction sur les dérivations de typage.

Deux cas

- Si $\Gamma \vdash e : \tau$ par la règle de subsomption avec prémisses $\Gamma \vdash e : \sigma$ et $\vdash \sigma <: \tau$.
  - Par HR on a $\Gamma \vdash_a e : \sigma'$ avec $\vdash \sigma' <: \sigma$.
  - Par transitivité du sous-typage on a $\vdash \sigma' <: \tau$,
  - Par équivalence $\vdash_a \sigma' <: \tau$.

  On a bien trouvé un sous-type σ' de τ tel que $\Gamma \vdash_a e : \sigma'$.
- Si $\Gamma \vdash e_1.\mathrm{x} = e_2;$ avec les prémisses $\Gamma \vdash e_1 : C$ et $C(\mathrm{x}) = \tau$ et $\Gamma \vdash e_2 : \tau$.
  - par HR, on a $\Gamma \vdash_a e_1 : \sigma_1$ avec σ_1 un sous-type de C.
  - nécessairement σ_1 est un type de classe (noté D)
  - par la propriété de cohérence on a également $D(\mathrm{x}) = \tau$.
  - par HR, on a $\Gamma \vdash_a e_2 : \sigma_2$ avec σ_2 un sous-type de τ.
  - on conclut avec la règle

$$\frac{\Gamma \vdash_a e_1 : D \qquad D(\mathrm{x}) = \tau \qquad \Gamma \vdash_a e_2 : \sigma_2 \qquad \vdash_a \sigma_2 <: \tau}{\Gamma \vdash_a e_1.\mathrm{x} = e_2;}$$

Partie 4 : Programmation Objet

Introduction

Les notions de sous-typage et de subsomption sont générales

- un type σ est sous-type d'un type τ dès lors qu'une valeur de type σ peut être utilisée sans problème partout où une valeur de type τ est attendue.
- Le sous-type σ peut être considéré dans ce cadre comme n'importe quel type « plus précis » que τ.
- L'opération de subsomption, qui consiste à considérer qu'une expression e pour laquelle on connaît un type précis σ comme étant du type moins précis τ correspond en un certain sens à « oublier » des informations sur e.
- L'idée recèle cependant un certain nombre de subtilités.

Covariance

Type $\tau_1 \times \tau_2$ des paires (supposées immuables, comme en caml ou python).
Plusieurs formes de types subsumés par $\tau_1 \times \tau_2$.

- Types de la forme $\tau_1 \times \tau_2 \times \ldots \times \tau_n$
  Sous-typage en largeur.
- Types de la forme $\sigma_1 \times \sigma_2$ où chaque σ_i est un sous-type du τ_i correspondant.
  Sous-typage en profondeur.
- Combinaison des deux cas précédents.

Sous-typage covariant : la relation de sous-typage sur les composantes est transférée aux composés.

$$\frac{\vdash \sigma_1 <: \tau_1 \qquad \vdash \sigma_2 <: \tau_2}{\vdash (\sigma_1 \times \sigma_2) <: \tau_1 \times \tau_2}$$

Contravariance

Type $\tau_1 \to \tau_2$ des fonctions avec un argument de type τ_1 et un résultat de type τ_2.

- `f` de type $\sigma_1 \to \sigma_2$.
- Quelles conditions doivent respecter σ_1 et σ_2 pour que `f` puisse être utilisée à la place d'une fonction $\tau_1 \to \tau_2$?
- le résultat `f(a)` de type σ_2 compatible avec le type τ_2 attendu.
  σ_2 est un sous-type de τ_2.
- l'argument `a` de type τ_1 est compatible avec le type σ_1 attendu par `f`.
  Autrement dit, le type τ_1 doit être un sous-type de σ_1.
- condition de sous-typage covariante pour le type du résultat, mais contravariante (c'est-à-dire d'orientation contraire) pour le type du paramètre.

$$\frac{\vdash \tau_1 <: \sigma_1 \qquad \vdash \sigma_2 <: \tau_2}{\vdash (\sigma_1 \to \sigma_2) <: \tau_1 \to \tau_2}$$

Invariance

- Pour un type de données mutables (tableau en java ou référence en caml) : ni covariance ni contravariance.
- Contraintes contraires en lecture et écriture.
- Avec $\vdash \sigma[] <: \tau[]$: un tableau de type $\sigma[]$ peut être vu comme un tableau de type $\tau[]$
  - un élément du tableau est de type σ mais peut être considéré comme de type τ
    $\longrightarrow \sigma$ doit être un sous-type de τ.
  - on peut placer dans le tableau de type $\sigma\,[\,]$ un élément de type τ.
    $\longrightarrow \tau$ doit être un sous-type de σ.
  - on obtient une règle qui demande que σ et τ soient équivalents du point de vue du sous-typage .

$$\frac{\vdash \sigma <: \tau \qquad \vdash \tau <: \sigma}{\vdash \sigma\,[\,] <: \tau\,[\,]}$$

Partie 4 : Programmation Objet

1. Sémantique d'un langage orienté objet
   - Classes, objets, méthodes
   - Héritage, sous-typage, liaison dynamique
   - Formalisation du sous-typage
   - Vérification des types en présence de sous-typage
   - Sous-typage au-delà des objets
   - Comparaison avec la programmation fonctionnelle
   - Comprenez-vous la liaison dynamique ?

Espaces de noms : classes ou modules.

Equivalent de la classe Point en caml

- définir un type d'enregistrement avec deux champs (mutables) x et y,
- fonction correspondant à chaque méthode.
- encapsuler le tout dans un module : espace de noms spécifiques.

```
module Point = struct
    type t = { mutable x: int; mutable y: int }

    let move p dx dy =
      p.x <- p.x + dx;
      p.y <- p.y + dy

    let sqDist p1 p2 =
      (p1.x-p2.x)*(p1.x-p2.x) + (p1.y-p2.y)*(p1.y-p2.y)

    let copy p = { x = p.x; y = p.y }
  end
```

Espaces de noms : classes ou modules.

- type t des points (Point.t de l'extérieur)
- trois fonctions avec un premier paramètre de type Point.t
- on peut ajouter une fonction de construction

```
let mk x y = { x; y }
```

- la signature du module peut restreindre la visibilité des champs

Hiérarchies de classes ou types algébriques

- classe abstraite GraphicElt, instanciée via l'un des trois cas particuliers concrets que sont les cercles, les rectangles et les groupes.
- en caml, type algébrique graphic_elt doté de trois constructeurs : un pour chaque forme concrète.

```
type graphic_elt =
| Circle    of Point.t * int
| Rectangle of Point.t * int * int
| Group     of Point.t * graphic_elt list ref
```

Liaison dynamique ou filtrage

- Chaque méthode, concrète ou abtraite, de la classe mère GraphicElt est traduite par une fonction caml qui prend un paramètre du type graphic_elt, et qui est définie par filtrage sur la forme de ce paramètre. (paramètre implicite plutôt en dernier argument)
- un cas par constructeur (sous-classe concrète de GraphicElt).

```
let rec contains p = function
  | Circle(a, r)          -> Point.sqDist a p <= r*r
  | Rectangle(a, w, h) -> a.x <= p.x && p.x <= a.x+w &&
                          a.y <= p.y && p.y <= a.y+h
  | Group(_, l)           -> List.exists (contains p) !l

let rec move dx dy = function
  | Circle(p, _) | Rectangle(p, _, _) ->
      Point.move p dx dy
  | Group(p, l) ->
      Point.move p dx dy;
      List.iter (move dx dy) !l
```

Une différence pratique

- Les hiérarchies de classes et la liaison dynamique en java, d'une part, et les types algébriques et le filtrage en caml, d'autre part, permettent de réaliser des effets similaires.
- différence « pratique » entre les deux.
  - En java, tout ce qui est relatif à une forme donnée d'objet est regroupé à un seul endroit (une classe), mais la définition de chaque fonction est séparée en plusieurs morceaux (les méthodes redéfinies par chaque classe).
  - En caml, chaque fonction est entièrement définie à un endroit, mais les éléments relatifs à un constructeur donné sont disséminés (dans le cas correspondant de chaque fonction).
- pas les mêmes choses qui « faciles » dans ces deux approches.
  - En java, il est facile d'ajouter un cas : il suffit de définir une nouvelle classe. Il est en revanche plus laborieux d'ajouter une fonction : il faut ajouter des méthodes dans plusieurs classes.
  - En caml, il est facile d'ajouter une fonction : il suffit de la définir à un endroit. Il est en revanche plus laborieux d'ajouter un constructeur : il faut ajouter un cas dans chaque fonction.

Factorisations

On peut factoriser la déclaration de l'attribut anchor et de la méthode move dans la classe GraphicElt

```
type graphic_elt = { anchor: Point.t; shape: shape }
and shape =
  | Circle    of int
  | Rectangle of int * int
  | Group     of graphic_elt list ref
```

Les fonctions s'y adaptent naturellement.

```
let rec contains p g = match g.shape with
  | Circle r         -> Point.sqDist g.anchor p <= r*r
  | Rectangle(w, h) -> g.anchor.x <= p.x && p.x <= g.anchor.x+w &&
                       g.anchor.y <= p.y && p.y <= g.anchor.y+h
  | Group l          -> List.exists (contains p) !l

let rec move dx dy g = match g.shape with
  | Group l -> Point.move g.anchor dx dy; List.iter (move dx dy) !l
  | _       -> Point.move g.anchor dx dy
```

Partie 4 : Programmation Objet

1. Sémantique d'un langage orienté objet
   - Classes, objets, méthodes
   - Héritage, sous-typage, liaison dynamique
   - Formalisation du sous-typage
   - Vérification des types en présence de sous-typage
   - Sous-typage au-delà des objets
   - Comparaison avec la programmation fonctionnelle
   - Comprenez-vous la liaison dynamique ?

Exemple d'utilisation de la liaison dynamique

Classe Fib avec une méthode fib calculant récursivement et naïvement un terme de la suite de Fibonacci.

```
class Fib {
  int fib(int n) {
    if (n < 2) { return n; }
    else       { return fib(n-1) + fib(n-2); }
  }
}
```

Sa complexité est exponentielle, et rend inimaginable le simple calcul suivant.

```
Fib f = new Fib();
System.out.println(f.fib(1000));
```

Mémoïsation

Classe héritant de Fib, et réutilisant sa méthode naïve.

```
class FibM extends Fib {
  HashMap<Integer, Integer> mem = new HashMap<>();
  int fib(int n) {
    if (!mem.containsKey(n)) { mem.put(n, super.fib(n)); }
    return mem.get(n);
  }
}
```

Le résultat est maintenant instantané.

```
Fib f = new FibM();
System.out.println(f.fib(1000));
```

Pourquoi ?

Synthèse

A retenir

- Les principaux mécanismes de la programmation objet dans Java
- Leur traduction dans des mécanismes plus bas niveau
- La structure des jugements de typage
  - contexte étendu par des informations sur les types définis
  - jugement de sous-typage
- La problématique d'une vérification de type algorithmique en présence de sous-typage

Savoir faire

- Simuler la vérification de type et l'exécution sur une programme Java simple