Langages de programmation, interprétation, compilation

Thibaut Balabonski @ Faculté des Sciences d'Orsay. V3, automne 2023

Partie 4

Table des matières

11 Sémantique d'un langage orienté objet

Dans le paradigme de programmation objet, le code est organisé dans des *classes* et l'exécution d'un programme est centrée sur l'interaction d'un certains nombre d'*objets* : de petites structures de données chacune liée à une classe. De nombreux langages de programmation intègrent ce paradigme chacun à sa manière, voire en font leur mode principal, dont notamment C++ et java. Dans ce chapitre, on présente les principes de la programmation objet en prenant le modèle de java.

11.1 Classes, objets

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***.

Classes, attributs, instances. La déclaration d'une classe introduit ainsi un nouveau ***type***. Dans sa forme la plus simple, une telle déclaration peut être vue comme la description d'un enregistrement :

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

Ici, `x` et `y` sont les deux ***champs*** de la classe `Point` (on les appelle aussi ses ***attributs***). 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`.

On crée un nouvel objet à l'aide de l'opérateur de construction `new`, en précisant au minimum le nom de la classe dont on veut créer une instance.

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

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

On accède aux champs d'un objet à l'aide de l'opérateur `.` (point), dont les deux opérandes sont un objet et un nom de champ.

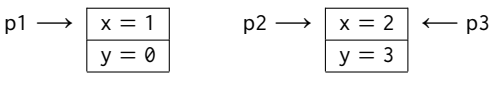
```
p.x = 1 + p.y;
```

Un simple nom de variable `z` ou un accès à un tableau `t[3]` sont aussi des expressions gauches.

L'expression obtenue, qualifiée d'***expression gauche***, peut être utilisée comme une expression ordinaire pour consulter la valeur d'un champ d'un objet (`p.y`) mais aussi à gauche de l'opérateur d'affectation pour affecter une nouvelle valeur à ce champ (`p.x = ...;`).

Deux instances d'une même classe sont des structures indépendantes. Chacune a ses propres valeurs pour chaque champ et évolue indépendamment de l'autre. En revanche, on peut 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;
```



Constructeurs. Une classe ne définit pas seulement des champs mais également du code applicable aux instances de cette classe, à commencer généralement par un ***constructeur***, c'est-à-dire une fonction destinée à initialiser les champs d'une nouvelle instance.

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

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

À noter : le constructeur est également l'occasion d'intégrer du code s'assurant de la validité des valeurs utilisées pour l'initialisation.

On peut alors fournir des paramètres lors de la création d'une instance pour que la création de l'objet lui-même soit immédiatement suivie de l'initialisation de ses champs à l'aide du code du constructeur. En supposant qu'on ait également donné un constructeur à la classe `Point` on pourrait ainsi écrire

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

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

Méthodes. Outre le constructeur, la définition d'une classe contient des définitions de ***méthodes***, c'est-à-dire de fonctions applicables aux instances de cette classe. Comme des fonctions ordinaires, les méthodes peuvent s'appliquer à 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);
  }
```

On appelle toujours une méthode pour un objet donné de la classe correspondante, en utilisant à nouveau la notation `.` (point). Partant d'une instance `p` de la classe `Point` on pourrait donc écrire

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

Dans ces trois exemples, l'instance `p` est appelée le ***paramètre implicite*** de l'appel. C'est aux champs de cette instance que font référence les mentions de `x` et `y` dans le code des méthodes. Ainsi, un appel de méthode `p.f(...)` est susceptible de consulter et/ou de modifier les champs de l'objet `p`. On peut expliciter cet aspect avec le mot-clé `this` qui, dans le code d'une méthode ou d'un constructeur, désigne le paramètre implicite de l'appel, c'est-à-dire l'instance à laquelle on applique ce code.

En première approximation, on peut voir l'appel de méthode `p.move(2, -1)` comme l'appel `mv(p, 2, -1)` d'une fonction ordinaire `mv` qui aurait été définie par

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

Cela peut vous rappeler un idiome python, avec un paramètre généralement nommé `self`.

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

Par opposition au paramètre implicite, les paramètres « ordinaires » `2` et `-1` sont les ***paramètres explicites*** de l'appel. De même, dans le code de `sqDistTo`, on a une distinction entre les simples accès `x`, sous-entendu `this.x`, au champ `x` du paramètre implicite, et les accès `q.x` au champ `x` du paramètre explicite `q`. Dans certaines situations, l'emploi de `this` peut même être obligatoire, pour lever une ambiguïté avec une autre variable nommée `x`. Ainsi, on pourrait écrire

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

le constructeur de la classe `Point`, où `x` désigne le paramètre et `this.x` le champ.

Champs et méthodes statiques. Dans une définition de classe, on peut introduire des champs qui ne sont pas liés aux instances mais à la classe elle-même. On a alors une valeur unique, partagée à l'échelle de la classe, qui est assimilable à une variable globale.

```
class Point {
  int x, y;
  static int max = 1024;
```

On accède à un tel champ depuis l'intérieur de la classe avec seulement son nom (`max`), ou depuis l'extérieur en y ajoutant le nom de la classe (`Point.max`). Attention : comme ce champ est global, toute modification est visible à l'échelle de la classe.

Les méthodes peuvent également être statiques. Elles deviennent alors similaires à des fonctions ordinaires. En particulier, elles n'ont pas de paramètre implicite et ne peuvent donc ni faire référence à `this` ni accéder à des champs ordinaires. En revanche, une telle méthode peut accéder aux champs statiques, qui sont justement définis au niveau de la classe.

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

Encapsulation. Chaque classe peut être associée à des ***invariants***, c'est-à-dire des propriétés de cohérence des valeurs des attributs d'une instance. Une instance de la classe `Circle` par exemple n'est bien formée que si son attribut `radius` contient une valeur positive ou nulle. On maintient de tels invariants en combinant deux moyens :

— S'assurer que le code de la classe `Circle` elle-même respecte bien l'invariant voulu. Le constructeur, par exemple, empêche toute mauvaise initialisation.

— Bloquer l'accès à `radius` à tout code externe à `Circle`. Pour cela on déclare le champ `radius` comme étant ***privé***, c'est-à-dire invisible en dehors de la classe `Circle`.

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

Alors, l'accès `c.radius` devient impossible depuis le code d'une autre classe. Pour permettre néanmoins la consultation de la valeur de `radius` (mais pas sa modification) par une autre classe, on peut ajouter une méthode dédiée à `Circle`.

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

Espaces de noms. Une classe délimite un ***espace de noms*** : les noms utilisés pour les attributs et méthodes d'une classe sont indépendants de ceux utilisés dans les autres classes. On peut donc tout à fait définir à côté de notre classe `Point`, une classe indépendante `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); }
```

Nos deux classes `Point` et `GraphicElt` possèdent toutes deux une méthode `move`, avec un contenu différent : celle de `Point` met à jour les champs `x` et `y` de l'instance à laquelle elle s'applique, celle de `GraphicElt` fait appel à une méthode de son champ `anchor`. Il n'y a cependant pas d'ambiguïté, chaque appel à une méthode `move` se faisant avec un paramètre implicite dont la classe détermine l'unique méthode pertinente.

Les notions d'*encapsulation* et d'*espace de noms*, qui séparent le fonctionnement interne d'une structure de données et son utilisation par un code tiers, sont des éléments centraux de génie logiciel, réalisés de différentes manières dans d'autres langages. En caml par exemple, le système de *modules* et les *types abstraits* permettent de réaliser quasiment à l'identique tout ce que nous avons vu dans cette section.

Une classe trop générale ? On pourrait imaginer qu'un élément graphique dispose d'une méthode déterminant si l'élément contient un point donné. Mais sans plus de précision 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"); }
```

11.2 Héritage, sous-typage, liaison dynamique

Une caractéristique distinctive de la programmation objet est la possibilité, étant donnée une classe `C` déjà définie, d'en dériver une nouvelle classe `D`, qui ***hérite*** de tous les champs de `C` et de toutes les méthodes définies dans `C`. La classe `D` peut en outre introduire de nouveaux champs et de nouvelles méthodes qui lui sont propres.

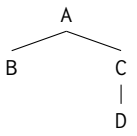
```
class D extends C { ... }
```

La classe dérivée `D` peut être vue comme la description d'un *cas particulier* de `C` : elle contient tout ce qui caractérisait `C`, et y ajoute des éléments spécifiques. On appelle `D` une ***sous-classe***, ou ***classe fille***, ou encore ***spécialisation*** de la classe `C`, cette dernière étant symétriquement la ***super-classe***, ***classe mère***, ou ***généralisation*** de `D`.

Hiérarchie de classes. En java, chaque classe hérite au maximum d'une autre classe (on parle d'héritage ***simple***). On peut donc visualiser un ensemble de classes et leur relation d'héritage comme un arbre ou une forêt. On parle de ***hiérarchie*** de classes.

La limitation à l'héritage simple est un choix de java. Les langages C++ et python autorisent à l'inverse l'héritage ***multiple***, mais cela implique quelques subtilités.

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



Spécialisation, redéfinition. On peut ainsi redéfinir un cercle comme un cas particulier d'élément graphique, dont l'ancre est le centre. Cette nouvelle classe possède donc en particulier un champ `anchor` et une méthode `move`, hérités de `GraphicElt`, sans que ceux-ci soient explicités dans le code. On y ajoute la déclaration du champ `radius`, qui est spécifique au cercle, 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;
  }
  ...
```

En outre, on a maintenant suffisamment d'informations sur la forme de notre élément pour donner une définition sensée à la mathode `contains`. On ***redéfinit*** donc cette méthode (*override*), c'est-à-dire qu'on lui donne une nouvelle définition qui va, pour les instances de `Circle` seulement, *remplacer* la définition héritée de `GraphicElt`.

On utilise souvent l'annotation `@Override` pour expliciter une redéfinition. C'est facultatif mais utile : le compilateur vérifie alors que la signature correspond bien à une méthode de la classe mère.

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

Similairement, on peut définir un rectangle ancré par son coin inférieur gauche et caractérisé par une largeur et une hauteur. On propose un constructeur prenant en paramètres deux coins opposés quelconques du rectangle, et une redéfinition de la méthode `contains` adaptée à cette nouvelle forme.

```
class Rectangle extends GraphicElt {
  int width, height;
  Rectangle(Point p1, Point p2) {
    anchor = new Point(Math.min(p1.x, p2.x), Math.min(p1.y, p2.y));
    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;
  }
}
```

On a ainsi deux cas particuliers d'éléments graphiques, avec chacun sa méthode `contains`.

Sous-typage. Cette notion de spécialisation a un impact sur le typage : toute instance des classes `Circle` et `Rectangle` peut être considérée comme étant également une instance de la classe mère `GraphicElt`. Par définition, ces instances possèdent en effet toutes les caractéristiques d'un élément graphique (champ `anchor`, méthode `move`, méthode `contains`), et peuvent être utilisées sans risque dans tout contexte où l'on s'attend à travailler avec une instance de `GraphicElt`.

Les valeurs de type `Rectangle` et `Circle` forment un *sous-ensemble* de celles de type `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, qui détermine notamment la forme concrète de l'objet en mémoire, est fixée définitivement à sa création. Il s'agit du *type réel* de l'objet, appelé ***type dynamique***.

Type « dynamique » ne signifie pas qu'il est susceptible de changer, mais qu'il est le type que l'on peut observer à l'exécution !

La déclaration `GraphicElt elt1` introduit une variable `elt1` dont le type est `GraphicElt`. Cette information est le *type apparent* de `elt1`, appelé ***type statique***, et est la seule information que connaisse le compilateur.

Un objet de type `Circle` peut également être considéré comme étant du type `GraphicElt`. Ainsi, 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 également accepté par le compilateur car `GraphicElt`, le type statique de `elt1`, déclare bien une méthode `contains`.

En exécutant l'exemple précédent on peut cependant observer que la méthode `contains` appelée est celle de la classe `Circle`, c'est-à-dire la classe réelle (type dynamique) de l'objet. La méthode `contains` de la classe mère `GraphicElt` a été *redéfinie* dans ses classes filles, et à l'exécution la méthode choisie est celle donnée par le type dynamique de l'objet (***liaison dynamique***).

Polymorphisme, liaison dynamique. On peut enrichir encore notre hiérarchie, et préciser la notion de redéfinition, en introduisant un nouvel élément graphique `Group`, qui contient plusieurs éléments graphiques. Comme première étape, on crée une classe `GList` représentant les listes chaînées d'éléments graphiques.

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

Le constructeur de cette classe est ***polymorphe***, c'est-à-dire capable de traiter des entrées de plusieurs types différents. En effet, bien qu'il déclare son premier paramètre comme de l'unique type (apparent, statique) `GraphicElt`, nous savons que ce type recouvre en réalité plusieurs types (réels, dynamiques) possibles. On peut ainsi construire une liste mélangeant des formes différentes.

```
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));
```

Nous pouvons maintenant définir un nouvel élément graphique ayant une telle liste pour attribut. À nouveau, la méthode permettant d'ajouter un nouvel élément à un groupe est polymorphe.

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

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

On peut définir une méthode `contains` adaptée à cette nouvelle forme, en convenant qu'un groupe contient un point `p` dès lors que l'un de ses éléments contient `p`.

Note de style : voici un bon idiome pour l'itération sur une liste chaînée, *lorsque l'on n'a pas d'itérateurs* sous la main.

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

Ici, le compilateur *ne peut pas connaître* le type dynamique de l'élément `l.elt` récupéré dans la liste polymorphe `l`. Un tel code ne peut donc fonctionner que grâce à la liaison dynamique.

Redéfinir, tout en réutilisant. Avec ceci notre classe `Group` est complète : elle redéfinit sa propre méthode `contains`, et hérite de la méthode `move` de sa classe mère `GraphicElt`. Cependant, cette méthode `move` est incomplète : elle ne déplace que l'ancre associée au groupe, alors que l'on voudrait déplacer également chaque élément du groupe. Il faut donc redéfinir une méthode `move` qui déplace l'ancre du groupe comme le fait déjà la méthode héritée, *et* qui déplace aussi l'ancre de chaque élément du groupe. Pour cela, on combine un appel à la méthode `move` telle qu'héritée 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. Notre classe `GraphicElt` n'a pas vocation à être instanciée directement : elle ne décrit qu'un cadre général pour des éléments concrets comme `Circle` et `Rectangle` et leurs groupes, et on ne construit des instances que via ces éléments concrets. On peut le formaliser en déclarant cette classe comme ***abstraite***, ce qui permet par ailleurs de ne pas fournir de contenu à la méthode `contains`.

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

Cela impose en revanche que chaque sous-classe concrète de `GraphicElt` redéfinisse bien la méthode `contains`.

Et la surcharge ? Des langages comme java proposent un mécanisme additionnel qui *ressemble* à du polymorphisme sans en être réellement : la ***surcharge statique***, qui permet de définir plusieurs méthodes ou opérateurs de même nom dans un même espace de noms (ici : dans une même classe). Ces différentes versions sont alors distinguées par leur signature, et plus particulièrement par les types des paramètres attendus. Lors d'un appel, le compilateur choisit la version à utiliser en fonction des types statiques des paramètres effectifs.

Ce mécanisme de surcharge statique est indépendant de tout concept objet, et *n'est pas* une caractéristique du paradigme de programmation objet.

En revanche, la conjonction du sous-typage et de la surcharge statique apporte quelques subtilités : parmi les signatures disponibles pour un identifiant donné, il faut choisir celle qui est « la plus précise » vis-à-vis du type *statique* des paramètres effectifs.

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

Le nom partagé `contains` n'est qu'une facilité d'écriture offerte au programmeur. En interne, le compilateur leur affecte des noms différents, forgés en combinant l'identifiant `contains` avec les types des paramètres.

Ce mécanisme ne fonctionne que si le choix entre plusieurs signature est sans ambiguïtés. Si la classe `D` étend la classe `C`, les deux méthodes suivantes ne peuvent pas cohabiter : il y a des paires de paramètres auxquelles elles s'appliquent toutes deux, sans qu'aucune soit prioritaire sur l'autre.

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

11.3 Formalisation du sous-typage

Pour décrire précisément les programmes bien typés dans un langage orienté objet avec héritage simple, nous devons formaliser à la fois une relation de « sous-typage », et son interaction avec les règles de typage.

Types. On considère deux formes de types : des types de base comme `int` ou `boolean`, et les types définis par des classes, que l'on notera C.

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

Chaque type de classe C est lié à une classe `c` particulière, possédant des attributs et des méthodes. Pour modéliser cela, on voit le type de classe C comme une association entre des identifiants (noms d'attributs ou de méthodes) et des informations de typage.

— Pour chaque attribut `x` de C, on note $C(\texttt{x})$ le type de `x`.
— Pour chaque méthode `f` de C, on note $C(\texttt{f})$ la signature de `f`.

La signature d'une méthode attendant n paramètres de types respectifs τ_1 à τ_n, et renvoyant un résultat de type τ, est notée $(\tau_1 \times \ldots \times \tau_n) \to \tau$. Dans le cas d'une méthode ne renvoyant pas de résultat, on notera la signature $(\tau_1 \times \ldots \times \tau_n) \to \texttt{void}$. Pour tout identifiant `id` ne désignant ni un attribut ni une méthode de la classe concernée, $C(\texttt{id})$ n'est pas défini.

À noter : `void` n'est pas à proprement parler un type, et a plusieurs significations. Dans une signature de fonction il désigne une absence de valeur. Dans un pointeur `void*` (en C) il désigne une valeur de type inconnu.

Pour la classe `Point` prise en exemple en début de chapitre, nous aurions donc un type $C_\texttt{p}$, avec les équations :

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

Héritage. On va modéliser les relations d'héritage entre classes par une 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 $\texttt{parent}(D) = C$.

Rappelons que lorsqu'une classe `d` hérite d'une classe `c`, alors `d` contient tous les attributs et méthodes de `c`, avec les mêmes types et mêmes signatures. On traduit cela par une 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` serait ainsi 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 :

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

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 (on dit aussi que τ_1 est ***subsumé*** par τ_2, c'est-à-dire inclus dans 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. On dit aussi dans ce cas que τ_2 est un ***sur-type*** (ou *super-type*) de τ_1, ou que τ_2 ***subsume*** τ_1.

On utilise encore le symbole $\vdash$, qui rappelle qu'un tel jugement nécessite une dérivation.

On peut définir des règles d'inférence décrivant cette relation comme la 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. Le ***jugement de typage*** $\Gamma \ \vdash \ e \ : \ \tau$ exprime que l'expression e est cohérente et de type τ dans le contexte Γ. On peut reprendre un certain nombre des règles déjà vues, comme

Rappel : le contexte associe les identifiants de variables aux types déclarés pour ces variables.

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

et y ajouter des règles spécifiques aux objets et au sous-typage.

La règle fondamentale liant typage et sous-typage est la règle de ***subsomption***, qui exprime qu'une expression e cohérente et de type σ peut également être considérée comme étant de n'importe quel type τ supérieur à σ.

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

L'opérateur `new` de construction d'un nouvel objet crée un objet de la classe demandée. On suppose ici que le contexte Γ associe également chaque nom de classe à son type. On considère dans cette version que la construction ne prend pas d'argument.

Pour ajouter des arguments au constructeur, on peut s'inspirer de la règle d'appel de méthode présentée un peu plus bas.

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

On définit que l'accès $e.\texttt{x}$ à un attribut est cohérent si l'expression e a un type de classe possédant un attribut `x`.

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

Similairement, l'appel de méthode $e.\texttt{f}(e_1, \ldots, e_n)$ est cohérent si l'expression e a un type de classe possédant une méthode `f`, *et si les paramètres e_1 à e_n ont bien les types attendus par cette méthode.*

Cette règle ne permet pas de typer l'appel d'une méthode qui ne renvoie pas de résultat (retour `void`). Une telle méthode ne peut en effet être appelée qu'en position d'instruction.

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

Règles de typage des instructions. Une instruction ne produit pas de valeur. On utilise donc pour les instructions un ***jugement de typage*** simplifié $\Gamma \vdash i$, exprimant que l'instruction i est cohérente, sans mentionner de type de résultat. Un tel jugement de typage vérifie en revanche que les expressions présentes dans l'instruction i sont utilisées à bon escient. Ainsi une instruction d'affectation `x = e;` sera bien typée dès lors que l'expression e est bien typée, et d'un type cohérent avec la variables x.

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

L'affectation $e_1\texttt{.x = }e_2\texttt{;}$ d'une nouvelle valeur à un attribut d'un objet étend l'idée précédente en utilisant la classe de e_1 pour déterminer le type attendu pour e_2.

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

Enfin, on peut donner ici une règle pour la seule utilisation cohérente possible d'une méthode ne renvoyant pas de résultat : que cet appel joue le rôle d'une instruction.

On pourrait également autoriser à cette position un appel de méthode produisant un résultat (résultat qui serait ignoré).

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

Exemple de dérivation. On considère la ligne de code suivante, qui combine la déclaration d'une variable `elt` et une instruction d'affectation.

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

On se place donc dans un environnement Γ tel que $\Gamma(\texttt{elt}) = C_\text{g}$, et on peut écrire la dérivation de typage suivante pour l'instruction d'affectation. Observez que cette seule dérivation combine tous les jugements de cette section : typage d'une instruction, typage d'une expression, et sous-typage.

$$\dfrac{\Gamma(\texttt{elt}) = C_\text{g} \qquad \dfrac{\dfrac{}{\Gamma \vdash \texttt{new Circle()} : C_\text{c}} \qquad \dfrac{\texttt{parent}(C_\text{c}) = C_\text{g}}{\vdash C_\text{c} <: C_\text{g}}}{\Gamma \vdash \texttt{new Circle()} : C_\text{g}}}{\Gamma \vdash \texttt{elt = new Circle();}}$$

11.4 Vérification des types en présence de sous-typage

Une formalisation telle que la précédente spécifie parfaitement les jugements de typage ***valides***. Ces règles sont cependant beaucoup moins guidées que ne l'étaient les règles de typage de ImpScript.

— La règle de ***transitivité***

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

utilise un type intermédiaire τ_2 qu'on ne peut pas extraire directement de l'objectif $\vdash \tau_1 <: \tau_3$. *Comment décider quel type intermédiaire utiliser pour poursuivre l'analyse efficacement ?*

— La règle de ***subsomption***

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

est susceptible d'être utilisée à tout moment, indépendamment de la forme syntaxique de l'expression e analysée, et introduit un type σ qui n'est pas donné par l'objectif $\Gamma \vdash e : \tau$. *Comment décider quand l'utiliser, et avec quels types ?*

La liberté autorise la diversité. Les degrés de liberté que laissent les règles précédentes permettent de justifier qu'un même jugement est valide avec plusieurs arbres de dérivation, de formes différentes. En supposant par exemple trois types de classes C_a, C_b et C_c, avec $\mathtt{parent}(C_\mathsf{a}) = C_\mathsf{b}$, $\mathtt{parent}(C_\mathsf{b}) = C_\mathsf{c}$ et $C_\mathsf{a}(\mathtt{f}) = C_\mathsf{b}(\mathtt{f}) = (C_\mathsf{c}) \to \mathtt{void}$, et en prenant une expression e telle que $\vdash e : C_\mathsf{a}$, on peut justifier la validité d'un appel $e.\mathtt{f}(e)$; avec la dérivation

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

ou avec la dérivation

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

et, bien que certains puissent considérer que l'une des deux est incongrue, ces deux dérivations sont aussi valables l'une que l'autre.

Stratégie de typage. Un algorithme de vérification des types sera plus efficace s'il suit une stratégie déterministe, qui ne nécessite pas d'explorer toutes ces possibilités. On va donner dans la suite les principes d'un tel algorithme, qui encadre l'utilisation de la transitivité et de la subsomption.

L'algorithme va prendre la forme d'une variante plus contrainte des règles d'inférence, appelée ***système de typage algorithmique***. Une propriété de ce système algorithmique est qu'il ne laisse plus qu'une forme d'arbre de dérivation possible pour chaque jugement valide. En conséquence, on pourra déduire des règles de typage un programme de vérification déterministe et efficace de la même manière que nous l'avions fait pour ImpScript.

En revanche, notez bien que l'objectif est de restreindre la forme des dérivations de typage, et pas l'ensemble des programmes acceptés. Nous démontrerons que le système algorithmique est équivalent au système d'origine, au sens où il permet de typer les mêmes programmes. Pour distinguer le distinguer du système algorithmique, on appellera ***système de référence*** le système d'origine défini à la section précédente.

Sous-typage algorithmique. Pour obtenir une version algorithmique des règles de sous-typage, on restreint l'application de la règle de transitivité au cas où le type intermédiaire τ_2 est immédiatement le parent de τ_1. Concrètement, cela revient à combiner la règle de transitivité et la règle du parent en une seule. 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.

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

Équivalence entre les deux systèmes de sous-typage. Le système de sous-typage de référence et le système de sous-typage algorithmique permettent de dériver les mêmes jugements de sous-typage, et sont en ce sens ***équivalents***. Comme les deux systèmes ont des vocations différentes (le premier définit les jugements valides, et le second décrit une stratégie de recherche), on exprime l'équivalence avec un vocabulaire asymétrique. On dit ainsi que le système algorithmique est correct (il ne permet par de justifier de jugement invalide) et complet (il permet bien de justifier tous les jugements valides) par rapport au système de référence.

Une première direction de l'équivalence, la correction, est censée être facile : toute relation de sous-typage dérivable dans le système restreint (algorithmique) est également dérivable dans le système de référence.

Lemme de correction.
Si $\vdash_a \sigma <: \tau$ alors $\vdash \sigma <: \tau$

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

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

□

La réciproque, la complétude, exprimant que le système algorithmique suffit à dériver toute relation de sous-typage valide, est un peu moins évidente. On démontre d'abord un lemme énonçant que la propriété de transitivité est ***admissible*** dans le système algorithmique, avec d'en déduire le résultat principal.

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 $\vdash_a \mathtt{parent}(C) <: \tau_2$, alors par hypothèse d'induction $\vdash_a \mathtt{parent}(C) <: \tau_3$, et donc $\vdash_a C <: \tau_3$. □

Lemme de complétude.
Si $\vdash \sigma <: \tau$ alors $\vdash_a \sigma <: \tau$

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 $\mathtt{parent}(D) = C$. Alors on construit la dérivation

$$\frac{\mathtt{parent}(D) = C \qquad \dfrac{}{\vdash_a C <: C}}{\vdash_a D <: C}$$

- 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 $\vdash_a \tau_1 <: \tau_3$. □

On pourrait aussi réaliser cette preuve en une seule induction, sans lemme de transitivité, à condition de généraliser un peu l'énoncé à démontrer.

Typage algorithmique. Nous allons maintenant restreindre l'utilisation de la subsomption, de sorte à ce qu'elle ne puisse s'appliquer qu'immédiatement au-dessus d'une règle imposant un type précis à l'une des expressions analysées. Il s'agit d'une part des règles de typage des appels de méthode, qui demandent que les paramètres effectifs aient exactement le type déclaré dans la signature de la méthode, et d'autre part des règles d'affectation, qui demandent que la nouvelle valeur ait exactement le type déclaré pour la variable ou pour l'attribut cible. On note $\Gamma \vdash_a e : \tau$ et $\Gamma \vdash_a i$ les jugements de typage obtenus avec ces règles restreintes.

On a ainsi une nouvelle règle pour l'affectation `x = e;` à une variable, qui demande que le type de l'expression e soit subsumé par le type de la variable `x`, ce qui est équivalent à donner la possibilité d'appliquer d'appliquer la règle de subsomption à e pour lui donner le type attendu. On a également une nouvelle règle similaire pour la modification d'un attribut d'un objet.

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

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

On a de même de nouvelles règles pour l'appel de méthode, qui demandent que le type de chaque paramètre effectif soit subsumé par le type attendu.

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

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

Les autres règles définissant $\Gamma \vdash e : \tau$ et $\Gamma \vdash i$ sont directement portées à $\Gamma \vdash_a e : \tau$ et $\Gamma \vdash_a i$, à l'exception de la règle de subsomption elle-même, qui disparaît.

Équivalence entre les deux systèmes de types. Comme déjà vu pour les jugements de sous-typage, l'un des sens de l'équivalence est simple : tout jugement de typage qui peut être dérivé dans le système algorithmique peut également être dérivé dans le système de référence.

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 ferait sans grande difficulté par induction sur la dérivation de $\Gamma \vdash_a e : \tau$ ou de $\Gamma \vdash_a i$.

L'énoncé même de la réciproque est un peu plus subtil. Comme le système algorithmique ne permet pas de finir une dérivation par une application de la règle de subsomption, on peut avoir un léger décalage dans le type obtenu à la fin d'une dérivation.

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	$\Gamma \vdash_a i$

La preuve se ferait à nouveau par induction sur les dérivations de typage. Voici deux cas concentrant les principaux arguments.

- Si $\Gamma \vdash e : \tau$ par la règle de subsomption avec prémisses $\Gamma \vdash e : \sigma$ et $\vdash \sigma <: \tau$. Alors par hypothèse d'induction on a $\Gamma \vdash_a e : \sigma'$ avec $\vdash \sigma' <: \sigma$. En outre, par transitivité du sous-typage on a donc $\vdash \sigma' <: \tau$, et par équivalence $\vdash_a \sigma' <: \tau$. On a donc bien trouvé un sous-type σ' de τ tel que $\Gamma \vdash_a e : \sigma'$.
- Si $\Gamma \vdash e_1.\mathtt{x} = e_2;$ avec les prémisses $\Gamma \vdash e_1 : C$ et $C(\mathtt{x}) = \tau$ et $\Gamma \vdash e_2 : \tau$. Par hypothèse d'induction sur la première prémisse, on a $\Gamma \vdash_a e_1 : \sigma_1$ avec σ_1 un sous-type de C. Alors nécessairement σ_1 est un type de classe D, et par la propriété de cohérence on a également $D(\mathtt{x}) = \tau$. Par hypothèse d'induction sur la deuxième

prémisse, on a $\Gamma \vdash_a e_2 : \sigma_2$ avec σ_2 un sous-type de τ. On peut donc conclure avec l'application de règle

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

11.5 Approfondissement : sous-typage au-delà des objets

Les notions de sous-typage et de subsomption peuvent s'étendre au-delà du simple cadre d'un ensemble de classes liées par des relations d'héritage. On peut ainsi considérer qu'un type σ est sous-type d'un type τ dès lors qu'une valeur de type σ peut être utilisée sans problèmes 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. Commençons par un exemple simple, avec un type $\tau_1 \times \tau_2$ des paires (supposées immuables, comme en caml ou python où de telles paires existent). On peut imaginer plusieurs formes de types subsumés par $\tau_1 \times \tau_2$.

- Des types de la forme $\tau_1 \times \tau_2 \times \ldots \times \tau_n$ comportant des composantes supplémentaires, après les deux premières de types τ_1 et τ_2. La subsomption consiste alors à « oublier » les composantes à partir de la troisième. On parle de sous-typage *en largeur*.
- Des types de la forme $\sigma_1 \times \sigma_2$ où chaque σ_i est un sous-type du τ_i correspondant. La subsomption consiste alors à potentiellement oublier des informations sur chaque composante. On parle de sous-typage *en profondeur*.
- N'importe quelle combinaison des deux cas précédents.

Dans le sous-typage en profondeur de nos paires, une relation de sous-typage sur les composantes est transférée aux composés. Le sous-typage est ici qualifié de ***covariant***.

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

Contravariance. Considérons maintenant un type $\tau_1 \to \tau_2$ représentant des fonctions attendant un argument de type τ_1 et renvoyant un résultat de type τ_2. Soit `f` une fonction 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 $\tau_1 \to \tau_2$?

D'une part, il faut que le résultat `f(a)` d'un appel à `f` soit d'un type σ_2 compatible avec le type τ_2 attendu. Il faut donc que σ_2 soit un sous-type de τ_2. D'autre part, il faut que l'argument `a` de type τ_1 donné à `f` soit compatible avec le type σ_1 attendu par `f`. Autrement dit, le type τ_1 doit être un sous-type de σ_1.

D'où finalement une condition de sous-typage qui est *covariante* pour le type du résultat, mais ***contravariante*** (c'est-à-dire d'orientation contraire) pour le type du paramètre. On dit aussi que le type τ_1 du paramètre a une position ***négative*** dans le type de fonction $\tau_1 \to \tau_2$.

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

Invariance. Pour un type de données mutable en revanche, comme un type τ`[]` de tableau en java ou un type τ `ref` de référence en caml, on n'admet ni covariance ni contravariance. En effet, une telle donnée pouvant être à la fois lue et écrite, nous avons des contraintes contraires.

- Si $\vdash \sigma[] <: \tau[]$, alors le type σ d'un élément lu dans un tableau σ`[]` doit être un sous-type de τ.
- Si $\vdash \sigma[] <: \tau[]$, alors on doit pouvoir placer un élément de type τ dans un tableau de type σ`[]`. Autrement dit, τ doit être un sous-type de σ.

On obtiendrait donc une règle qui demande que σ et τ soient équivalents du point de vue du sous-typage (avec nos exemples précédents cela ne peut être réalisé que pour des types égaux, mais dans certaines situations on pourrait admettre de légères variations).

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

En réalité, java autorise le sous-typage covariant des tableaux, pour des raisons historiques. Ce choix qui corrompt la sûreté du système de types impose un test à l'exécution lors de chaque écriture dans un tableau, susceptible de produire une `ArrayStoreException`.

On peut supposer que cette règle serait retirée aujourd'hui si cela n'invalidait pas les bases de code existantes.

11.6 Approfondissement : comparaison avec la programmation fonctionnelle

La définition d'une classe abstraite et d'un certain nombre de sous-classes concrètes, puis l'appel d'une méthode commune à toutes mais dont chaque sous-classe est susceptible d'avoir une variante propre sélectionnée par liaison dynamique, est un idiome caractéristique de la programmation orientée objet qui n'existe pas en tant que tel dans les autres paradigmes. Cependant, certains mécanismes centraux de caml et des langages fonctionnels n'en sont pas si éloignés.

Espaces de noms : classes ou modules. On peut facilement écrire en caml un équivalent de la classe `Point`. Il suffit de définir un type d'enregistrement avec deux champs (mutables) `x` et `y`, puis une fonction correspondant à chaque méthode. En encapsulant le tout dans un module, on s'assure de plus que le tout vit dans son propre espace de noms.

Ce code crée explicitement un module au sein d'un fichier. On peut aussi simplement placer toutes ces définition dans un fichier séparé `point.ml`.

```
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
```

Dans ce module on définit un type `t` des points, qui sera connu à l'extérieur sous le nom `Point.t`. Les trois fonctions sont ensuite similaires aux trois méthodes de la classe `Point`, à ceci près qu'elle prennent explicitement un premier paramètre de type `Point.t` qui correspond au paramètre implicite. Pour un parallèle complet, on pourrait également y ajouter une fonction équivalente au constructeur (nécessaire si on ne rend pas public la structure du type `Point.t`).

Raccourci d'écriture : `{x; y}` est l'enregistrement `{x = x; y = y}` dont le champ `x` est initialisé avec la valeur de la variable `x`, et similairement pour `y`.

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

Hiérarchies de classes ou types algébriques. La classe abstraite `GraphicElt` n'est jamais instanciée directement, mais seulement via l'un des trois cas particuliers concrets que sont les cercles, les rectangles et les groupes. On peut donc en caml résumer cette hiérarchie par un 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
```

Les paramètres de chaque constructeur reprennent les attributs de la classe correspondante. Ainsi chacun a une ancre (type `Point.t`), les cercles ont en plus un rayon (type `int`), les rectangles une largeur et une hauteur (type `int`), et les groupes une liste d'éléments graphiques (type `graphic_elt list`, placé ici dans une référence pour pouvoir agrandir la liste).

Liaison dynamique ou filtrage. Alors 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.

```
let rec contains g p = match g with
  ...
```

On a un cas par constructeur, c'est-à-dire un cas par sous-classe concrète de `GraphicElt`. Le contenu de chacun de ces cas est celui de la méthode héritée ou redéfinie par la classe concrète correspondante.

```
  | 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 (fun g -> contains g p) !l
```

À noter : en plaçant le paramètre g, qui correspond au paramètre implicite, en dernier argument, on simplifie légèrement l'écriture du code en profitant des mécanismes d'application partielle.

```
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
```

Et de même pour la fonction `move`, qui fait appel à la fonction de même nom définie dans le module `Point`.

```
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
```

Bilan : la liaison dynamique est d'une certaine manière équivalente à une opération de filtrage !

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. Il reste cependant une 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).

Ce ne sont donc pas les mêmes choses qui sont « 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. Vous avez peut-être remarqué que la factorisation de la déclaration de l'attribut `anchor` et de la méthode `move` dans la classe `GraphicElt` n'était pas reproduite dans notre type `graphic_elt`. En effet, chaque constructeur déclare indépendamment des autres un paramètre du type `Point.t` pour son ancre. On corrige cela avec cette nouvelle définition de type, qui définit chaque élément graphique par une ancre et une forme.

```
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
```

N'hésitez pas à comparer les deux versions !

Un interprète en java. Le lien entre hiérarchies de classes et types algébriques peut également être exploité dans l'autre sens. Voici une traduction directe en java de l'interprète IMP donné au chapitre 2. On y trouve deux classes concrètes pour les programmes et pour les séquences d'instructions, chacune avec sa méthode exec, la seconde prenant en paramètre une table de hachage créée par la première et représentant l'environnement.

On omet dans ce code un constructeur routinier pour chaque classe.

```
class Program {
  Sequence code;
  void exec() { code.exec(new HashMap<String, Integer>()); }
}

class Sequence {
  List<Instruction> seq;
  void exec(HashMap<String, Integer> env) {
    for (Instruction i: seq) { i.exec(env); }
  }
}
```

Puis une classe abstraite pour les instructions, qui déclare une méthode abstraite exec,

```
abstract class Instruction {
  abstract void exec(HashMap<String, Integer> env);
}
```

et une sous-classe concrète pour chaque constructeur, avec la définition adaptée de exec.

```
class Set extends Instruction {
  String x;
  Expression e;
  void exec(HashMap<String, Integer> env) { env.put(x, e.eval(env)); }
}
class If extends Instruction {
  Expression e;
  Sequence s1, s2;
  void exec(HashMap<String, Integer> env) {
    if (e.eval(env) != 0) { s1.exec(env); }
    else                  { s2.exec(env); }
  }
}
class While extends Instruction {
  Expression e;
  Sequence s;
  void exec(HashMap<String, Integer> env) {
    if (e.eval(env) != 0) { s.exec(env); this.exec(env); }
  }
}
class Print extends Instruction {
  Expression e;
  void exec(HashMap<String, Integer> env) {
    System.out.print((char)e.eval(env));
  }
}
```

Et de même une classe abstraite pour les expressions, qui déclare une méthode abstraite `eval`, suivie d'une classe concrète par constructeur avec chacune une définition adaptée pour `eval`.

```
abstract class Expression {
  abstract int eval(HashMap<String, Integer> env);
}
class Int extends Expression {
  int n;
  int eval(HashMap<String, Integer> env) { return n; }
}
class Var extends Expression {
  String x;
  int eval(HashMap<String, Integer> env) { return env.get(x); }
}
enum Bop { ADD, MUL, LT, AND }
```

```
class Binop extends Expression {
  Bop op;
  Expression e1, e2;
  int eval(HashMap<String, Integer> env) {
    switch (op) {
      case ADD: return e1.eval(env) + e2.eval(env);
      case MUL: return e1.eval(env) * e2.eval(env);
      case LT:  return (e1.eval(env)<e2.eval(env))?1:0;
      case AND: return (e1.eval(env)!=0)?e2.eval(env):0;
      default: throw new Error("unknown binary operator");
    }
  }
}
```

Un vérificateur de types en java ? On peut ajouter à l'interprète précédent un vérificateur de types. Il « suffit » pour cela d'ajouter une méthode `typecheck` à chaque classe d'instruction et une méthode `type` à chaque classe d'expression, chacune traduisant la règle de typage correspondante.

11.7 Comprenez-vous la liaison dynamique ? le test ultime

Voici une 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));
```

Voici maintenant une classe héritant de `Fib`, et réutilisant sa méthode naïve (notez la présence de `super`).

```
  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);
    }
  }
```

Si maintenant nous exécutons les instructions suivantes, le résultat est instantané.

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

Pourquoi ?

12 Compilation d'un langage orienté objet

Dans ce chapitre, nous allons réaliser les principaux mécanismes de la programmation objet, à l'aide des mécanismes plus élémentaires de tableaux et de fonctions du langage ImpScript. Il suffirait donc ensuite de composer cette transformation avec le compilateur déjà connu pour ImpScript pour obtenir un compilateur complet d'un langage objet vers Mips.

12.1 Représentation des objets.

On représente un objet par une structure allouée sur le tas, qui doit contenir au minimum :
- une identification de la classe,
- les valeurs des différents attributs.

La valeur d'un objet est un pointeur vers cette structure.

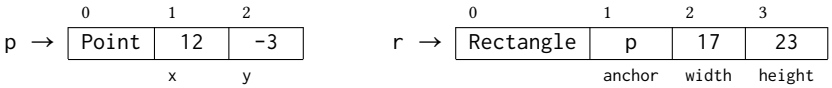
En ImpScript, on peut le réaliser simplement à l'aide d'un tableau, avec un élément par case : l'identification de la classe à l'indice `0`, puis les valeurs des k champs aux indices `1` à k, dans l'ordre dans lequel ils ont été déclarés. Ainsi, les objets créés avec les déclarations

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

class Rectangle {
  Point anchor;
  int width;
  int height;
  ...
}
```

```
Point p = new Point(12, -3);
Rectangle r = new Rectangle(p, 17, 23);
```

peuvent être représentés par les tableaux



On précisera un peu plus tard la nature de l'identification de la classe stockée à l'indice `0`.

Accès aux attributs. Avec une telle représentation, l'accès à l'un des champs se fait simplement à l'aide d'un accès au tableau, à l'indice correspondant au champ. Pour récupérer la valeur du champ `x` du point `p`, il suffit en ImpScript d'aller chercher la valeur à l'indice `1` dans le tableau corrrespondant.

Kawa	ImpScript
`p.x`	`p[1]`

Ce code ImpScript pourrait ensuite être traduit en Mips par des opérations de lecture ou d'écriture en mémoire, mais ce n'est pas notre travail ici : le compilateur ImpScript le fait déjà.

De même, pour définir la hauteur du rectangle `r` au double de sa largeur, il suffit de manipuler les cases d'indices `2` et `3`. on utilise l'instruction ImpScript suivante.

Kawa	ImpScript
`r.height = 2 * r.width;`	`r[3] = 2 * r[2];`

Nouveauté importante par rapport au compilateur ImpScript : *le typage de Kawa est nécessaire à la production des bonnes instructions ImpScript !*

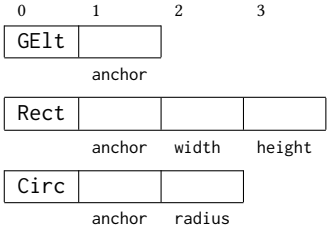
Pour produire un tel accès, un compilateur de Kawa vers ImpScript doit savoir déterminer (statiquement) l'indice correspondant à un nom de champ. Pour cela, il se base sur le type (statique) des expressions `p` et `r`, et sur la position du nom de champ cherché dans la liste des déclarations.

Héritage et attributs. Lors de la traduction d'un accès `e.x` au champ `x` de l'objet o désigné par l'expression `e`, le compilateur se sert du type statique de `e` pour localiser la valeur dans la structure représentant o. Il est donc nécessaire que, quelque soit le type dynamique de o, la position déterminée avec le type statique soit correcte. Autrement dit, si une classe `D` étend une classe `C`, on va s'assurer que les champs déclarés par `C`, qui sont tous présents dans `D` par héritage, soient placés aux même indices dans les structures représentant chaque type d'objet. Il suffit pour cela de placer en premier tous les champs hérités, et ensuite seulement les champs ajoutés. Avec le code ci-contre où deux classes `Rect` et `Circ` héritent d'une même classe `GElt`, le champ hérité `anchor` est placé en première position pour tout le monde, puis chaque sous-classe ajoute ses propres champs à partir de l'indice suivant.

```
class GElt {
  Point anchor;
  ...
}

class Rect extends GElt {
  int width;
  int height;
  ...
}

class Circ extends GElt {
  int radius;
  ...
}
```



12.2 Classes, méthodes et liaison dynamique

Une méthode `m(`x_1`, ..., `x_n`)` à n arguments définie dans une classe `C` peut être vue comme fonction ordinaire à $n+1$ arguments : les n paramètres explicites x_1 à x_n, plus le paramètre implicite utilisé pour appeler la méthode (désigné dans le corps de la méthode par `this`). C'est précisément ce que l'on va faire ici pour traduire une méthode Kawa en une fonction ImpScript, en forgeant en plus pour la fonction un nouveau nom qui combine le nom de la classe et le nom de la méthode d'origine.

Kawa (dans la classe `Point`)

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

ImpScript

```
function Point_move(this, dx, dy) {
  this[1] = this[1] + dx;
  this[2] = this[2] + dy;
}
```

Rappel : `this` désigne ici une instance de la classe `Point`, c'est-à-dire dans la traduction ImpScript un tableau contenant en particulier les valeurs des champs `x` et `y` aux indices `1` et `2`.

Une fois cette traduction fixée pour les méthodes, on souhaite qu'un appel de méthode `p.move(...)`, avec paramètre implicite `p` de la classe `Point`, soit traduit en ImpScript par un appel à la fonction `Point_move` prenant le paramètre implicite `p` en premier argument.

Kawa

```
p.move(2, 3)
```

ImpScript (tentative)

```
Point_move(p, 2, 3)
```

Les choses ne sont toutefois pas si directes, du fait de la liaison dynamique : lors d'un appel de méthode `e.m(`e_1`, ..., `e_n`)`, la méthode à exécuter est déteminée par le type *dynamique* de e, c'est-à-dire par la classe réelle de l'objet désigné par e. Cependant, cette information n'est pas toujours disponible pour le compilateur, qui doit se débrouiller pour générer le bon appel à partir des informations auxquelles il a accès, à savoir :

— le nom `m` de la méthode, et
— le type *statique* de l'expression e.

Ce mécanisme est la subtilité principale de la compilation des langages orientés objet.

Descripteurs de classes. Lors de la compilation d'une classe `C`, on produit donc un ensemble de fonctions ImpScript : une pour chaque méthode de `C`. Chacune de ces fonctions fait partie du programme, et est stockée en mémoire. Rappelons aussi un détail du langage ImpScript : lors d'une définition

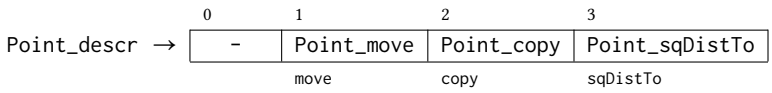
```
function Point_move(...) { ... }
```

on déclare une *variable globale* `Point_move`, dont la valeur est la fonction correspondante, représentée par son adresse en mémoire.

Voir p.132 la traduction d'une expression `Call(f, params)` en Mips, qui évalue `f` et saute à l'adresse résultant de cette évaluation.

Pour permettre l'accès dynamique (c'est-à-dire : déterminé à l'exécution) aux méthodes des différentes classes, on va stocker en mémoire une structure de données pour chaque classe `C`, contenant les fonctions ImpScript obtenues par traduction des méthodes de `C`. La structure, appelée ***descripteur de classe***, sera ici concrètement un tableau contenant l'adresse de chaque méthode de la classe décrite. Comme on l'avait déjà fait pour représenter les objets, on place les méthodes dans les cases successives, dans l'ordre de leur apparition dans le programme source. On conserve en outre la case d'indice `0` pour une information additionnelle que l'on ajoutera un peu plus tard. Le descripteur de classe de la classe `Point` ci-contre a donc la forme

```
class Point {
 ...
 void move(int dx, int dy) {...}
 Point copy() { ... }
 int sqDistTo(Point q) { ... }
}
```

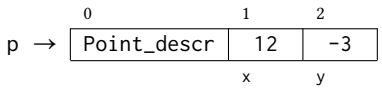


Les descripteurs de classe sont initialisés au début du programme, avant l'exécution du code principal. Ici on peut le faire avec la simple déclaration ImpScript d'un tableau initialisé.

```
let Point_descr = [null, Point_move, Point_copy, Point_sqDistTo];
```

Ces descripteurs sont aussi précisément l'information d'identification de classe à placer au début de chaque objet. Voici donc le tableau représentant l'instance de `Point` déjà prise en exemple, avec pour premier élément (un pointeur vers) le descripteur de la classe *réelle* de cet objet.

```
p = new Point(12, -3);
```



Appel de méthode avec liaison dynamique. Rassemblons les pièces du puzzle. Pour réaliser un appel $e.\texttt{m}(e_1, \ldots, e_n)$, nous avons à notre disposition :

- l'objet o obtenu en évaluant le paramètre implicite e, dont la première donnée est un pointeur vers le descripteur de classe de la classe réelle (type dynamique) de o,
- le type statique de e et le nom `m` de la méthode, qui nous permettent de déterminer (statiquement) la position dans le descripteur de classe à laquelle se trouve la fonction à appeler.

On réalise alors l'appel en combinant l'indice de la méthode, déterminé statiquement, avec le descripteur de classe, obtenu dynamiquement. Pour l'appel de la méthode `move` d'une instance `p` de la classe `Point`, on accède donc au descripteur de classe de `p` avec l'accès `p[0]`, puis à la méthode `Point_move` en prenant la valeur stockée à l'indice `1` dans ce descripteur.

Kawa

```
p.move(2, 3)
```

ImpScript

```
p[0][1](p, 2, 3)
```

C'est précisément cela que permet l'instruction `jalr` en Mips.

Cette mécanique dépend crucialement du fait que l'on puisse appeler une fonction, pas seulement directement par son nom, mais également par un pointeur ! On parle d'***appel indirect***.

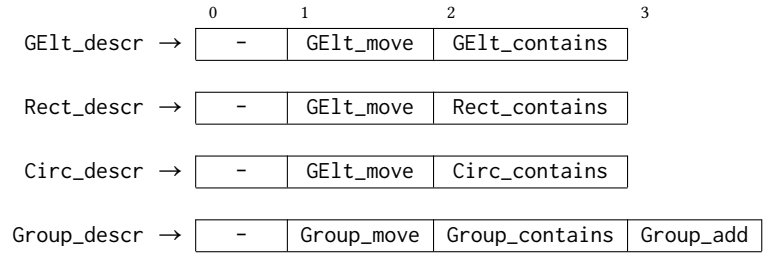
Héritage et descripteurs de classes. Le descripteur d'une classe doit contenir l'ensemble des méthodes à appeler pour un objet appartenant à cette classe. Similairement à ce qu'on a déjà fait pour les attributs dans la représentation des objets, lorsqu'une classe `D` étend une classe `C`, les méthodes héritées de `C` vont avoir dans le descripteur de la classe `D` les mêmes positions qu'elles avaient dans le descripteur de la classe `C`. En outre, dans le cas d'une *redéfinition*, la nouvelle méthode va prendre la position de la méthode qu'elle remplace. Pour le code ci-contre nous avons donc quatre descripteurs, dont trois partagent un pointeur vers la même version de la méthode `move`, mais dont chacun a sa propre version de `contains`. En outre, seule la classe `Group` possède une méthode `add`.

```
class GElt {
 void move(int dx, ind dy) ..
 boolean contains(Point p) ..
}

class Rect extends GElt {
 boolean contains(Point p) ..
}

class Circ extends GElt {
 boolean contains(Point p) ..
}

class Group extends GElt {
 void add(GElt g) ..
 boolean contains(Point p) ..
 void move(int dx, int dy) ..
}
```



Vous pouvez également remarquer que, dans le descripteur de la classe `Group`, les méthodes ne respectent plus l'ordre du code source de cette classe. En effet, les positions des méthodes `move` et `contains` ont été fixées par la classe mère `GElt`.

Avec de tels descripteurs, si `e` est une expression de type statique `GElt`, on peut donc simplement traduire les appels `e.move(...)` ou `e.contains(...)` en utilisant l'indice `1` ou `2` dans le descripteur de classe de l'objet `e`, et on sait que la version effectivement appelée sera celle correspondant au type dynamique de `e`.

Kawa

```
e.move(2, 3)
e.contains(p)
```

ImpScript

```
e[0][1](e, 2, 3)
e[0][2](e, p)
```

En revanche, un appel `e.add(...)` ne doit être accepté et traduit en `e[0][3](e, ...)` que si `e` a statiquement le type `Group` (on y reviendra à la prochaine section).

Optimisations pour les méthodes statiques. L'appel de méthode dynamique que nous venons de voir est puissant mais a un certain coût, puisqu'il faut aller lire consécutivement dans deux structures de données (l'objet, puis le descripteur de classe) pour déterminer la méthode à appeler. Dans le cas d'une méthode *statique* il est judicieux de remplacer cet appel indirect par un appel direct, où on saute directement à une adresse calculée statiquement.

On peut également généraliser ce critère à toute situation où, connaissant le type statique du paramètre implicite et le nom de la méthode appelée, on sait déterminer qu'il existe une seule méthode possible.

Construction d'un objet et constructeurs. La création d'un objet comporte plusieurs aspects :

1. allocation de la structure, c'est-à-dire création du tableau qui servira de support à l'objet,
2. initialisation de l'entête avec un pointeur vers le bon descripteur de classe,
3. initialisation des autres champs de l'objet par un ***constructeur***.

Vous pouvez constater que les constructeurs sont fort mal nommés : ce ne sont pas eux qui créent les objets. Il ne sont en réalité que des méthodes un tout petit peu particulières, qui s'appliquent à un objet déjà construit pour en initialiser les champs.

L'une des particularités des constructeurs, par rapport aux méthodes ordinaires, est qu'ils ne sont appelés que lors de la création d'un objet. À ce moment, on connaît donc précisément le type réel de l'objet : nous sommes dans la situation évoquée au paragraphe précédent où il est possible de connaître statiquement la bonne fonction à appeler, et donc de réaliser un appel direct. Il faut en revanche bien au préalable avoir construit l'objet lui-même, ce qu'on peut faire avec une unique ligne ImpScript qui crée un tableau de la bonne taille, avec un pointeur vers le descripteur de classe dans la première case et des valeurs nulles ailleurs.

Kawa

```
Point p = new Point(12, -3);
```

ImpScript

```
let p = [Point_descr, null, null];
Point_constructor(p, 12, -3);
```

En conséquence également, il n'est pas nécessaire d'ajouter les constructeurs dans les descripteurs de classes.

Constructeur `super`. Dans le cas où une classe `D` étend une classe `C`, le constructeur de `D` peut se reposer sur le constructeur de `C` pour initialiser les champs hérités de `C`. Cela se fait en java par exemple en utilisant le mot-clé `super` comme une méthode. On pourrait ainsi avoir pour nos classes `GElt` et `Rect` une mise en place comme ci-contre. Alors la création d'un objet de la classe `Rect` se fait comme précédemment, par création de la structure d'abord puis appel du constructeur `Rect_constructor`, avec la seule subtilité supplémentaire que le code de `Rect_constructor` contient un appel à `GElt_constructor`.

```
class GElt {
 Point anchor;
 GraphicElt(Point p) {
   this.anchor = p;
 }
}

class Rect extends GElt {
 int width, height;
 Rectangle(Point p,
           int w, int h) {
  super(p);
  this.width = w;
  this.height = h;
 }
}
```

Kawa

```
Rect r = new Rect(p, 4, 8);
```

ImpScript

```
let p = [Rect_descr,
         null, null, null];
Rect_constructor(r, p, 4, 8);
```

```
Rect(Point p,
     int w, int h) {
  super(p);
  this.width = w;
  this.height = h;
}
```

```
function Rect_constructor(r, p,
                          w, h) {
  GElt_constructor(r, p);
  r[2] = w;
  r[3] = h;
}
```

À nouveau, le choix de ce constructeur parent est fait statiquement.

En java l'appel au constructeur `super` est systématique : le compilateur insère un appel `super()` s'il n'y a pas d'appel explicite. Avec des hiérarchies profondes, cela peut faire un long enchaînement d'appels pour initialiser certains objets.

Appel de méthode avec `this` et `super`. Le mot-clé `this` est une référence vers l'objet courant (le paramètre implicite de l'appel de méthode en cours). Son type statique est la classe courante, mais son type dynamique peut être différent, et dans l'appel `this.m(`e_1`, ..., `e_n`)` le choix de la version de `m` se fait bien en fonction du type dynamique de l'objet. C'est ce phénomène qui était à l'œuvre dans le code de la section 11.7.

Le mot-clé `super` est également une référence vers l'objet courant, le même que désigné par `this`, mais considère cette fois cet objet comme étant *réellement* de la classe mère de la classe courante. Utilisé pour appeler une méthode, il contourne le mécanisme de liaison dynamique : il applique un appel de méthode à l'objet courant `this`, mais en sélectionnant statiquement la version valable dans la classe mère de la classe courante.

12.3 Hiérarchie des classes.

On a vu que le type statique et le type dynamique d'une expression pouvaient différer. En conséquence, le compilateur est susceptible de ne pas connaître toutes les méthodes d'un objet donné, et de refuser (lors de son analyse statique) l'emploi d'une méthode qui pourtant existe (dynamiquement). Ainsi, dans un apppel comme le suivant où g désigne une instance de `Group` mais est vue avec le type statique `GElt`, le compilateur rejette tout appel `g.add(...)` au motif que `add` n'est pas une méthode de `GElt`.

```
class GElt {
 void move(int dx, ind dy) ..
 boolean contains(Point p) ..
}

class Group extends GElt {
 void add(GElt g) ..
 boolean contains(Point p) ..
 void move(int dx, int dy) ..
}
```

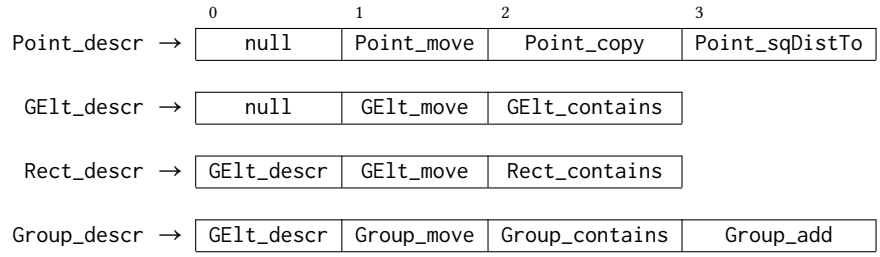
```
GElt g = new Group();
g.add(...);
```

Des langages comme java permettent de forcer la main au compilateur dans ce genre de situations. On peut par exemple écrire ainsi une fonction qui prend en paramètre deux éléments graphiques, tente d'ajouter le deuxième au premier, et renvoie un booléen indiquant si l'ajout était possible.

```
static boolean add(GElt g, GElt e) {
  if (g instanceof Group) {
    ((Group)g).add(e);
    return true;
  } else {
    return false;
  }
}
```

De telles opérations demandent, à l'exécution, d'explorer la hiérarchie des classes, pour déterminer si le type dynamique d'un objet est compatible avec un type donné.

Représentation de la hiérarchie à l'exécution. Pour conserver à l'exécution des informations sur les relations d'héritage, il suffit d'utiliser le champ que nous avions laissé libre dans les descripteurs de classes pour indiquer l'éventuel parent de la classe décrite. On identifie cette éventuelle classe parent par un pointeur vers son descripteur. Dans le cas d'une classe qui n'a pas de parent, il suffit de laisser à la place un pointeur nul. On complète donc ainsi les descripteurs de classes pris en exemples dans les pages précédentes.



Test du type dynamique avec `instanceof`. Un test `e instanceof C` produit `true` si le type dynamique de `e` est un sous-type de `C`, et `false` sinon. Pour réaliser un tel test à l'exécution, on consulte d'abord le descripteur de classe de l'objet désigné par `e`, puis le descripteur parent, et ainsi de suite jusqu'à trouver le descripteur de `C` (réponse `true`) ou jusqu'à arriver au bout de la hiérarchie (réponse `false`).

Dans cet exemple on utilise un tel test pour définir la valeur d'une variable booléenne `b`. Côté ImpScript, une variable `d` désigne le descripteur en cours d'exploration. On fait évoluer son contenu jusqu'à arriver soit au bout de la hiérarchie, soit au descripteur de la classe cherchée.

Kawa

```
b = g instanceof Group;
```

ImpScript

```
d = g[0];
while (d != null && d != Group_descr)
  { d = d[0]; }
if (d == Group_descr) { b = true;  }
else                  { b = false; }
```

Transtypage. Une expression de conversion `(C)e` (ou ***transtypage***, ou ***cast***) désigne simplement l'expression `e`, mais force le compilateur à la considérer comme étant du type `C`. Ainsi :

— la valeur de `(C)e` est la valeur de `e`,
— le type dynamique de `(C)e` est le type dynamique de (l'objet désigné par) `e`,
— le type statique de `(C)e` est `C`.

Au moment de traiter une opération `(C)e`, le compilateur connaît deux types avec certitude : le type statique `D` de l'expression `e` et le type `C` fourni par le programme. Le type dynamique `E` de l'objet désigné par `e` est a priori inconnu : on sait seulement qu'il s'agit d'un sous-type du type statique `D`. On a alors trois cas de figure selon la relation entre les deux types `C` et `D`.

1. Si le type cible `C` est une super-classe du type statique `D` (***upcast***), alors la conversion est toujours légitime : on sait que le type dynamique `E` est un sous-type de `D`, et donc aussi par transitivité un sous-type de `C`, et donc que l'objet peut bien être considéré comme une instance de `C`. Le code produit pour `(C)e` est exactement le code de `e` : la conversion n'a aucun impact sur l'exécution.
2. Si le type cible `C` est une sous-classe du type statique `D` (***downcast***), alors la conversion est parfois légitime : le type dynamique `E` est susceptible de bien être un sous-type de `C`, mais ce n'est pas systématique. On ne peut cependant le savoir que dynamiquement. On inclut donc à l'exécution, en plus du code de `e`, un test vérifiant que `E` est bien une sous-classe de `C`.
3. Si le type cible `C` n'est ni une sous-classe ni une super-classe du type statique `D`, alors il est certain que le type dynamique `E` n'est pas un sous-type de `C`, et le compilateur peut rejeter directement cette expression.

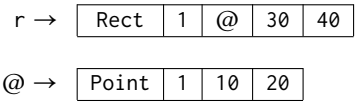
12.4 Approfondissement : gestion de la mémoire

La traduction proposée jusqu'ici de Kawa vers ImpScript suppose que la gestion de la mémoire est effectuée intégralement (et automatiquement) au niveau du programme ImpScript. C'est d'ailleurs ce qui arrivera si notre code ImpScript est exécuté par un moteur javascript ordinaire. Supposons maintenant disposer en ImpScript de deux fonctions `malloc` et `free`. Nous allons voir une technique simple pour détruire automatiquement *certains* des objets qui ne sont plus utilisés.

Références. Rappel : un objet est une structure allouée en mémoire, à laquelle on accède par un pointeur. Conséquence : un objet vers lequel il n'existe plus de pointeur est définitivement inaccessible, et peut être détruit. Ce critère peut directement être traduit en une technique de gestion automatique de la mémoire appelée ***reference counting*** : on mémorise dans un champ spécial de chaque objet le nombre de copies existantes du pointeur dont il est la cible, et on détruit l'objet lorsque ce compte tombe à zéro. Exemple : la ligne

```
Rect r = new Rect(new Point(10, 20), 30, 40);
```

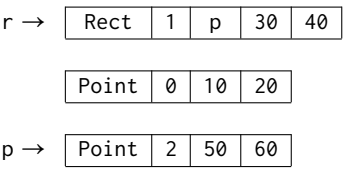
crée deux objets, vers chacun desquels il existe précisément un pointeur : la variable `r` vers l'instance de `Rect`, et le champ `r.anchor` vers l'instance de `Point`. Voici les structures correspondantes, où le compteur de références est placé dans la case d'indice `1` (et les attributs décalés pour en tenir compte), et @ est l'adresse de l'instance anonyme de `Point`.



Après création d'une nouvelle instance de `Point` et modification de l'ancre de `r` avec le code

```
Point p = new Point(50, 60);
r.anchor = p;
```

on obtient une nouvelle situation avec trois objets. Le compteur de références du nouveau `Point` vaut `2`, puisqu'on peut y accéder à la fois par le pointeur `p` et par le champ `r.anchor`. Le compteur du premier `Point` est en revanche tombé à `0` : il peut être détruit.



Gestion d'un compteur de références. Pour gérer un compteur de références, il faut déjà le prévoir et l'initialiser au moment de la création d'un objet.

Kawa

```
Point p = new Point(10, 20);
```

ImpScript

```
p = [Point_descr, 1, null, null];
Point_constructor(p, 10, 20);
```

Ensuite, chaque affectation d'une variable ou d'un champ contenant une référence vers un objet doit, en plus de l'affectation elle-même, réaliser deux opérations : incrémenter le compteur du nouvel objet (c'est-à-dire ajouter `1` au champ dédié), et décrémenter le compteur de l'ancien objet.

Kawa

```
r.anchor = p;
```

ImpScript

```
p[1] += 1;
r[2] = p;
decr_ref(r[2]);
```

Le décrément est un peu subtil, car il doit notamment déclencher la destruction de l'objet lorsque le compteur atteint 0. On fait cette destruction en deux parties : d'abord décrémenter les compteurs de référence de tous les objets pointés par les champs de `obj`, ce qui peut déclencher d'autres destructions en cascade, et enfin détruire l'objet lui-même.

Il faut vérifier que l'objet cible existe bien. En revanche, on peut supposer que le compteur est strictement positif.

```
function decr_ref(obj) {
  if (obj == null) { return; }
  obj[1] -= 1;
  if (obj[1] == 0) {
    obj[0][1](obj);   // méthode spéciale ''finalize''
    free(obj);
  }
}
```

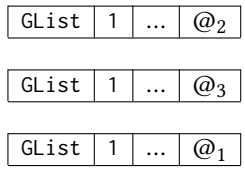
Pour décrémenter les compteurs des autres objets vers lesquels `obj` possède une référence, on propose ici d'avoir pour chaque classe `C` une méthode spéciale `C_finalize` placée à l'indice `1` de son descripteur. Cette méthode est générée par le compilateur sans intervention du programmeur. Elle est spécifique à chaque classe, pour désigner précisément quels champs doivent être concernés. Pour notre classe des rectangles nous aurions ainsi à déclencher un décrément sur l'objet référencé par le champ `anchor`.

```
function Rect_finalize(r) {
  decr_ref(r.anchor);
}
```

Objets inaccessibles et insuffisance du comptage de références. Le critère précédent ne capte cependant qu'une partie des objets effectivement inaccessibles qui peuvent être détruits. Par exemple, avec la classe `GList` ci-contre il est possible de créer la situation suivante, où trois objets à trois adresse anonymes $@_1$, $@_2$ et $@_3$ forment un cycle en mémoire. Chaque objet a un compteur de références valant précisément `1`, en raison du champ `next` de l'objet qui le précède dans le cycle, mais aucun n'est réellement accessible.

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

GList l3 = new GList(.., null);
GList l2 = new GList(.., l3);
GList l1 = new GList(.., l2);
l3.next = l1;
l1 = null;
l2 = null;
l3 = null;
```



Ces trois objets sont *de facto* inaccessibles au programme et devraient être éliminés par un gestionnaire de mémoire complet.

Gestion automatique de la mémoire. Un système de complet de gestion automatique de la mémoire doit donc, *en plus ou la place* d'un suivi des références, régulièrement explorer le tas pour déterminer quelles structures sont encore ou non accessibles par une suite de pointeurs à partir d'éléments ***racines***, c'est-à-dire d'éléments de base du programme comme les variables, qui sont directement accessibles.

12.5 Approfondissement : héritage multiple

De nombreux langages orientés objet, comme C++ ou python, permettent l'héritage multiple, c'est-à-dire permettent de définir une classe étendant plusieurs autres classes.

Représentation des objets. Dans un cas d'héritage multiple il faut affiner la représentation des objets et des descripteurs de classe. En effet, les super-types du type réel d'un objet o ne correspondent plus systématiquement à des préfixes du tableau représentant o.

Considérons l'exemple suivant, écrit cette fois avec une syntaxe à la C++, où on définit une classe `FlexArray` héritant de deux classes `Array` et `List`.

```
class Array {
  int nth(int i) { ... }
};
class List {
  void push(int v) { ... }
  int pop() { ... }
};
class FlexArray : public Array, public List {};
```

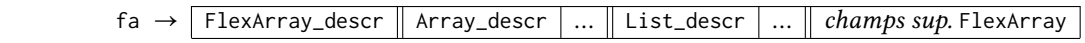
La représentation d'une instance de `FlexArray`

```
FlexArray fa = new FlexArray();
```

va contenir le descripteur de cette classe, les champs hérités de chacune des deux classes `Array` et `List`, et enfin ses champs propres.

Comme précédemment, la référence `fa` sur un objet de type dynamique `FlexArray` peut également être considérée comme une référence vers un `Array`, car les champs de `Array` sont placés de manière adéquate. Ce n'est en revanche pas le cas pour `List` : les champs hérités de la classe `List` sont placés plus loin dans la structure, et ne sont même pas attenants à un descripteur.

On propose donc la variante suivante, qui intègre dans la représentation des instances de `FlexArray` de véritables instances de `Array` et de `List`.



Alors une conversion de types, qu'il s'agisse d'*upcast* ou de *downcast* se traduit par un peu d'arithmétique de pointeur. Par exemple ici, on convertit de `FlexArray` vers `Array` (*upcast*) en considérant le sous-tableau qui commence à la deuxième case (`Array_descr`) et vers `List` (*upcast* encore) en considérant le sous-tableau qui commence à la case `List_descr`.

Le diamant maléfique. Supposons maintenant que les classes `Array` et `List` précédentes héritent toutes deux d'une même classe `Collection`.

```
class Collection {
  int cardinal() { ... }
};
```

La classe `FlexArray` hérite de la méthode `cardinal` de deux manières différentes : une via `Array` et une via `List`. Tant que toutes ces classes partagent une même version `Collection_cardinal`, la situation reste claire. Mais que penser de la situation où `Array` ou `List` redéfinit la méthode `cardinal()` ? Quelle version doit alors être appelée en priorité par `FlexArray` ?

On a ici une vraie ambiguïté sémantique, appelée le ***problème du diamant***. Chaque langage avec héritage multiple fixe ses propres règles pour clarifier cette question, avec des stratégies très variées :

- imposer que la classe `FlexArray` redéfinisse explicitement les méthodes ambiguës,
- imposer que les accès aux méthodes ambiguës mentionnent explicitement le nom de la classe à utiliser,
- fixer un ordre de priorité, selon l'ordre dans lequel les héritages sont déclarés,
- etc.

Bilan : si vous utilisez un langage avec héritage multiple, vous n'avez d'autre choix que de vous renseigner sur la manière dont ce langage traite le problème du diamant.

Et si vous pensiez être à l'abri du problème en java, c'est raté. Java permet un héritage multiple au niveau des interfaces. Sans code dans les interfaces le problème du diamant n'existe pas. Mais depuis java 8 les interfaces peuvent fournir des implémentations par défaut aux méthodes déclarées.

13 Et ensuite

Revenons sur ce que nous avons parcouru pendant ce semestre sur la compilation et la théorie des langages de programmation, et en particulier sur :

— ce qu'il faut en retenir, même pour ceux qui ne voudraient plus jamais y toucher,
— ce qui reste, pour ceux qui voudraient aller plus loin.

13.1 Bilan du semestre

Dans ce cours, j'ai fait le choix de vous présenter ensemble deux aspects des langages de programmation : leur *sémantique*, qui définit la signification et les effets des programmes, et leur *compilation*, c'est-à-dire la décomposition de leur comportement en mécanismes plus élémentaires et exécutables. L'objectif de ce mélange est d'élargir votre compréhension de l'acte même de programmer.

Définition d'un langage de programmation. Un langage de programmation est défini par *des* syntaxes et *des* sémantiques. Côté syntaxe :

— la syntaxe concrète définit ce que l'on écrit lorsque l'on programme, et attise parfois les passions et les trolls,
— la syntaxe abstraite définit les structures qui sont représentées par un texte concret, et est également ce sur quoi on peut raisonner.

Suis-je en train de dire que les trolls résonnent plus qu'ils ne raisonnent ?

Côté sémantique :

— la sémantique statique, en particulier avec les types, distingue les programmes qui ont du sens de ceux qui n'en ont pas,
— la sémantique dynamique décrit les comportements à attendre d'un programme, et spécifie notamment les résultats et les effets.

Exécution d'un programme. Pour permettre l'exécution d'un programme donné dans un langage source, l'interprétation et la compilation sont deux voies de natures différentes. On peut observer cette différence par les entrées et sorties produites dans chaque cas.

— Interpréter un programme, c'est exécuter son comportement. Dans ce contexte on a :
  — en entrée : un programme *et* des entrées,
  — en sortie : le résultat.
— Compiler un programme, c'est le traduire vers un autre langage (qui pourra ensuite être exécuté à l'aide de ses mécanismes propres). Dans ce contexte on a :
  — en entrée : un programme,
  — en sortie : un programme *équivalent*.

Au-delà de leurs différences fondamentales, ces deux voies ont de nombreux aspects techniques communs. Elles mettent en œuvre l'analyse d'un texte source (analyse lexicale, analyse grammaticale), la manipulation d'une représentation symbolique du programme (AST), et imposent le respect d'une sémantique (correction).

Un domaine tentaculaire. Étudier la compilation, c'est approfondir sa culture et sa compréhension des langages de programmation. On y décortique les langages et leurs mécanismes. On en ressort meilleur programmeur, mieux apte à écrire des programmes sûrs et efficaces, et on améliore sa capacité à comprendre et corriger les erreurs dans ses programmes.

C'est aussi une occasion d'explorer et de connecter des champs multiples de l'informatique :

— l'algorithmique, avec des algorithmes avancés d'analyse et d'optimisation,
— l'informatique théorique, avec la sémantique, les langages formels, la calculabilité,
— la technologie informatique, lors de l'interface avec l'architecture et le système,
— la programmation elle-même : un compilateur est un programme d'envergure, dont l'écriture regroupe tous les aspects précédents.

Ouvertures. Certains des chapitres de ce cours ouvrent sur des domaines que vous pourrez explorer dans la suite de votre cursus.

À la suite des chapitres consacrés aux langages formels, aux grammaires et aux automates, s'étendent les domaines de la *calculabilité* et de la *complexité*, qui donnent les *fondements théoriques de l'algorithmique*. On y cherche notamment à caractériser les dispositifs de calcul (automates, machines de Turing, machines RAM, etc) et les problèmes qu'ils peuvent résoudre.

On y découvre avec la notion d'indécidabilité que certains problèmes ne peuvent pas être résolus par des algorithmes. On y caractérise aussi certains problèmes algorithmiques qui, bien que solubles, sont intrinséquement difficiles. On ne sait en revanche toujours pas si P est égal ou non à NP.

À la suite des chapitres traitant de syntaxe abstraite, de sémantique et de types, on arrive à la *science des langages de programmation*. Outre la modélisation du comportement des programmes, on y vise à créer de nouveaux outils d'analyse des programmes et de leurs comportements, de nouveaux systèmes de types et de nouveaux langages de programmation permettant d'améliorer la sûreté des programmes futurs. C'est ici que l'on traite de preuve de programmes. C'est de là que viennent les assistants à la preuve, et les plus intéressants des nouveaux langages de programmation. À ce propos, connaissez-vous Rust ?

Les chapitres présentant des traductions d'un langage donné vers un autre de plus bas niveau forment le début de l'*implémentation des langages de programmation*. On y veut faire le lien entre des mécanismes de programmation de haut niveau, agréables au programmeur, et les opérations plus élémentaires accessibles à la machine. On y cherche également à ce que l'exécution des programmes soit ensuite la plus efficace possible. Cela peut passer par des analyses fines de la structure des programmes et de leurs opérations pour détecter des endroits où des optimisations sont possibles, ou encore par la meilleure utilisation des capacités matérielles de la machine cible.

13.2 Suite du cours

Avec le cours de compilation de M1, vous pourrez compléter ce cours de deux nouvelles parties, toujours à cheval entre sémantique des langages et compilation.

Sémantique, typage et compilation des langages fonctionnels. Cette partie inclut les constructions classiques de la programmation fonctionnelle (fermetures, récursion, types algébriques, filtrage), de la gestion automatique de la mémoire, mais aussi du typage avancé (polymorphisme, inférence). Et toujours sur un large spectre allant de la théorie au code.

Analyse de programme et génération de code optimisé. Transformer un programme pour le rendre plus efficace, faire le meilleur usage possibles des instructions et des registres de la machine, et tout ça automatiquement. Cela demandera de nouveaux algorithmes pour analyser finement le comportement des programmes.