Principes des langages de programmation
INF 321

Eric Goubault

24 mars 2014

Table des matières

Chapitre 1

Introduction

Objectif du cours Ce cours de L3 vise à donner des éléments conceptuels concernant les langages de programmation. Il a été conçu pour être lisible par des élèves de classes préparatoires MPI essentiellement, qui ont déjà l'expérience de la programmation en (O)Caml, et des connaissances en algorithmique élémentaire. On insiste dans ce cours sur les concepts nouveaux pour cette catégorie d'étudiants que sont les notions de calculabilité et complexité, et surtout sur la sémantique mathématique des langages de programmation. Il est illustré par un certain nombre de paradigmes de programmation : la programmation impérative, la programmation fonctionnelle et la programmation réactive synchrone. On n'y traite pas, encore, de programmation logique par exemple.

Contenu du cours Le chapitre 2 met en place les principes des langages impératifs, et permet d'introduire doucement à la syntaxe Java. On en profite pour mettre en place certaines notations de sémantique dénotationnelle (qui sera développée au chapitre 6) que l'on utilise pour clarifier la notion d'adresse (location mémoire, et référence), souvent mal maîtrisée par les jeunes étudiants en informatique, spécialement quand ils n'ont qu'une expérience en programmation fonctionnelle.

L'utilisation de structures de données complexes n'est introduite qu'au chapitre 3, avec une vision très algébrique (somme, produit, équations récursives aux domaines), proche de considérations que l'on peut avoir en sémantique, et en particulier en typage de langages (fonctionnels en général). Cette vision permettra la nécessaire déformation de l'esprit permettant de bien assimiler le chapitre 9.

Le chapitre 4 traite du paradigme orienté objet, à travers Java principalement.

Le chapitre 5 démarre la partie plus théorique du cours : sous le prétexte d'expliquer l'exécution de fonctions récursives, on développe quelques concepts élémentaires de la calculabilité (fonctions récursives primitives, et fonctions récursives partielles), et l'on termine par quelques notions sur les classes de complexité. Ce chapitre introduit à des nombreux concepts développés dans le cours

INF 423, [3].

Le chapitre 6 développe les outils classiques de la sémantique dénotationnelle de langages impératifs. Le problème principal est de définir la sémantique des boucles, et l'on utilise pour ce faire l'artillerie des CPOs, et le théorème de Kleene. On note en passant le lien entre fonctions continues au sens des CPOs (sur les entiers naturels) et les fonctions calculables du chapitre 5.

Le chapitre 7 introduit les concepts de la logique classique du premier ordre nécessaires à la compréhension de la preuve de programmes (chapitre 8) et au typage (chapitre 9). On y introduit, rapidement, le problème de la satisfaction de formules en logique des prédicats du premier ordre, et la théorie de la démonstration, d'un fragment de cette logique (la logique propositionnelle quantifiée du premier ordre). La présentation du problème de satisfaction, dans un modèle quelconque, est traitée de façon légèrement non-standard, dans le sens où on le transcrit en une sémantique dénotationnelle, comme pour les langages de programmation, au chapitre 6. Ceci permet d'attirer l'attention de l'étudiant au parallèle qu'il y a entre logique, et programmes.

On dérive des chapitres 6 et 7 une méthode de validation de programmes (impératifs), par la preuve à la Hoare. Comme au chapitre 7, on montre qu'hélas, tous ces problèmes sont généralement indécidables, c'est-à-dire qu'il n'existe pas de méthode automatique qui peut valider tout programme, en temps fini. Ceci pourrait être une bonne introduction, au théorème de Rice d'une part, et aux méthodes d'approximations en validation, comme l'interprétation abstraite, que l'on ne traite pas ici.

Le chapitre 9 revient sur la programmation fonctionnelle, et introduit un langage jouet, PCF, afin de mettre en lumière certains phénomènes que même des programmeurs Caml n'ont sans doute pas remarqués. Le premier est celui de l'ordre d'évaluation dans les appels de fonctions. On en donne une sémantique opérationnelle, qui est une autre grande famille de sémantiques de langages de programmation possible. En examinant de près ces sémantiques, on comprend sous une autre forme, le paradigme de passage d'argument par valeur et par référence, vu au chapitre 2, dans le cadre de langages impératifs, et on en découvre un autre, le passage par nécessité, à la base du langage fonctionnel Haskell. L'autre point est le typage, qui paraît si naturel au programmeur Caml : on montre en fait qu'il s'agit d'une forme de preuve (au sens de la théorie de la démonstration, chapitre 7) de bon comportement des programmes. Ceci est une bonne introduction à l'isomorphisme de Curry-Howard et à sa descendance nombreuse.

On termine ce cours avec un paradigme sans aucun doute original pour le programmeur classique, les langages réactifs synchrones, tels Lustre. Ce sont des langages non seulement très utiles en pratique, pour la programmation de contrôle-commande par exemple, mais qui ont aussi une sémantique très propre, dont les origines remontent aux réseaux de Kahn. Cela nous permet, une dernière fois, d'utiliser le cadre théorique élégant des CPOs du chapitre 6.

Remarques et remerciements Ce cours amène naturellement à INF 431, cf. [13] et à INF423, cf. [3]. Il pourra être complété par la lecture des polycopiés, plus introductifs à Java, comme [5], et plus algorithmiques, comme [9] (ou encore le livre [6]). En ce qui concerne les langages de programmation, on pourra trouver à la bibliothèque ou sur le web plusieurs livres, dont [7] pour le Java, [11] pour le C, [4] pour le OCaml. Pour avoir une introduction au C++, on pourra se reporter à [10].

Je remercie Sylvie Putot d'avoir bien voulu relire et corriger les deux premières versions de ce polycopié.

Chapitre 2

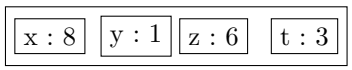
Les fondements de la programmation impérative

La programmation "impérative" est un paradigme essentiel. C'est le premier utilisé dans les langages de programmation, de part son côté naturel. Dans le paradigme impératif, un programme est conçu comme une suite d'ordres donnés à un moteur d'exécution, ce dernier étant lui-même une machine à états. La suite d'ordre modifie ainsi l'état global de la machine (mémoire en particulier), étape par étape. Cela correspond à la vision la plus intuitive que l'on peut avoir d'un algorithme, séquentiel.

Les langages impératifs, comme C et Java, ont tous en commun l'utilisation de cinq constructions, qui constituent ce que l'on appelera le noyau impératif : la déclaration de variables, l'affectation d'une expression à une variable, la séquence, le test et la boucle. Des traits impératifs se retrouvent également dans d'autres langages, reposant sur d'autres paradigmes. Par exemple, il est tout à fait possible de programmer de façon impérative, en Caml.

2.1 Variables et types

Avant de démarrer, il nous faut parler du concept de variable, dans les langages informatiques. C'est une notion un peu différente de la notion mathématique. En mathématiques, les variables sont quantifiées, le nom importe peu ($\forall x, P(x)$, $\exists x, P(x)$), seul le lien avec P importe. En informatique, une variable est une abstraction d'une *location* mémoire (ou *adresse* mémoire)



Les variables permettent en premier lieu de stocker des calculs intermédiaires, et de les réutiliser. Pour effectuer certaines tâches en un temps raisonnable, il est nécessaire d'occuper de la mémoire, il y a une relation entre les classes de

complexité en temps et en espace, que l'on evoquera brièvement au chapitre 5, et qui sera plus traité au cours INF423 [3].

Syntaxiquement, une variable est un mot d'une ou plusieurs lettres (soumis à certaines règles, cf. le mémento [1] Java du cours) par exemple `x`, `y`, `resultat1` etc.

Aux variables sont généralement associés des types, on y reviendra pour la programmation fonctionnelle au chapitre 9.

Un type décrit et structure un ensemble de valeurs possibles (comme, en mathématique, $\mathbb{R}$, $\mathbb{N}$, $\mathbb{R}^2$ etc.). Il existe des types élémentaires (valeurs simples), types composés (tableaux, enregistrements etc.). En fait, les types ont une structure plus intéressante qu'il n'y paraît de prime abord, on verra cela au chapitre 3 pour Java et la plupart des langages impératifs, et au chapitre 9, dans le cas des langages fonctionnels.

Les types élémentaires sont

- `int` : nombres entiers compris entre -2^{31} et $2^{31}-1$; types similaires, `byte`, `short`, `long`, (et en C : `long long`...)
- `boolean` (`false`, `true`) - pas en C
- `float`, type similaire `double`
- `char` : ex. `'g'`

Avant d'utiliser une variable, il faut d'abord la déclarer et déclarer son type, en Java. Puis il faut réserver un emplacement mémoire (« allocation »). Pour les types élémentaires, cette allocation est faite à la déclaration. On peut généralement déclarer, allouer et initialiser en même temps une variable (attention en C néanmoins, il existe quelques règles syntaxiques).

Voici quelques exemples simples :

En Java :

```
int x=3;
```

En C :

```
int x=3;
```

En Caml :

```
let x=ref 3 in p;;
```

Ce dernier code en fait un peu plus, car il y a une notion de portée : `x` est connu uniquement dans `p`.

Les langages de programmation (« haut-niveau ») sont structurés, il existe une notion de bloc ; par exemple dans une fonction, ou le corps d'une boucle etc. Une variable peut être connue dans un bloc, mais pas à l'extérieur. Cela permet d'appliquer une méthodologie saine de codage ; structurer le code, et cloisonner les informations, on verra cela plus en détail au chapitre 3.

Il existe aussi une notion de variable finale en Java : ce sont des variables ne pouvant être affectées qu'une fois (ce sont les équivalents `const` C). L'idée est

1. téléchargeable sur `http://www.enseignement.polytechnique.fr/informatique/INF321/memento.pdf`

que les variables finales ne peuvent être modifiées après une première affectation, elles sont en fait constantes. Donc le code suivant est correct :

```
final int x=4;
y=x+3;
```

Par contre, celui-ci est incorrect :

```
final int x=4;
x=3;
```

Le contraire des variables finales s'appelle les variables mutables. Cela nous permet de donner une explication rapide du ! en Caml. La version de la variable `x` utilisée dans le code suivant est finale :

```
let x=4 in y=x+3
```

Alors que la version mutable est :

```
let x=ref 4 in y=!x+3
```

Le `ref` dans le code plus haut veut dire que `x` contient en fait l'adresse de la location mémoire contenant la valeur 4, et `!x` permet, à partir de l'adresse contenue dans `x`, de rapatrier la valeur de la location mémoire correspondante. On reviendra à l'explication précise de cela à la section 2.6.

2.2 Codage des nombres

Un `int` est codé sur 32 bits, en base 2 (signe codé par complémentation). Donc `x=19` est codé par le mot sur 32 bits :

`00000000000000000000000000010011`

On peut jongler avec la représentation binaire, décalage à gauche : 19 << 1 (`...100110`=38), à droite 19 >> 1 (`...1001`=9), masquage (&, |) etc.

Pour les types `float` et `double`, la différence avec les nombres idéaux (les réels dans ce cas), est encore pire, d'une certaine façon. Il s'agit d'un codage en précision finie : la *mantisse* est codée en base 2, l'*exposant* également, et le tout sur un nombre fini de bits (cf. norme IEEE 754).

Attention, à cause de tout cela, et des erreurs d'arrondi dues au nombre fini de bits utilisés pour le codage des nombres, il n'y a pas associativité de l'addition de la multiplication et de la plupart des opérations qui sont d'habitude associatives dans les nombres réels.

Considérons le programme suivant :

```
float x, y;
x = 1.0f;
y = x+0.00000001f;
```

Alors, `x` et `y` ont la même valeur, après exécution du programme. Ce n'est évidemment pas le cas si on avait une machine qui calculait sur les nombres réels.

Un grand classique (Kahan-Muller) de programme qui donne un résultat surprenant, à cause du calcul sous-jacent en précision finie, est le suivant :

```
float x0, x1, x2;
x0=11/2;
x1=61/11;
for (i=1; i<=N; i++) {
 x2=111 - (1130-3000/x0)/x1;
 x0=x1;
 x1=x2;
}
```

Une exécution produit la suite : 5.5902 ; 5.6333 ; 5.6721 ; 5.6675 ; 4.9412 ; -10.5622 ; 160.5037 ; 102.1900 ; 100.1251 ; 100.0073 ; 100.0004 ; 100.0000 ; 100.0000 ; 100.0000 ; ...(stable) alors que la vraie limite dans les réels est 6 ! La cause de cette différence entre la sémantique en nombres réels et en nombres flottants est que le système dynamique discret dont on calcule l'orbite, dans la boucle, est un système non-linéaire, avec un point fixe attractif (100) et un point fixe répulsif (6). Les conditions initiales $x_0 = \frac{11}{2}$, $x_1 = \frac{61}{11}$, sont pour la première, représentée de façon exacte dans les nombres flottants, ce qui n'est pas le cas pour la deuxième. Cette erreur initiale fait qu'alors qu'on aurait du converger vers 6, on s'en approche, mais ce point fixe étant répulsif, on tend ainsi vers l'autre point fixe, 100, attractif.

Voici un autre exemple de comportement surprenant dans les nombres flottants, beaucoup plus simple, toujours du au même problème :

```
float x0, x1;
x0=7/10;
for (i=1; i<=N; i++) {
  x1 = 11*x0 - 7;
  x0 = x1:
}
```

Ce programme produit la suite 0.7000 ; 0.7000 ; 0.7000 ; 0.6997 ; 0.6972 ; 0.6693 ; 0.3621 ; -3.0169 ; -40.1857 ; -449.0428 ; -4946.4712 ; -54418.1836 ; -598607 ; -6584684 ; qui diverge vers $-\infty$, alors que dans les réels c'est la suite constante égale à 0.7 ! La raison en est que 7/10 n'est pas représentable de façon exacte dans les nombres flottants (par expansion sur les puissances, négatives, de 2). Cette erreur initiale est multipliée par 11 à chaque tour de boucle.

2.3 Expressions arithmétiques et instructions

Une expression arithmétique est un élément de syntaxe décrivant des calculs sur les valeurs des variables. On appelle expression, tout élément de syntaxe formé à partir de constantes et de variables avec des opérations (+, -, *, / etc.).

Voici un exemple d'expression en Java ou C :

```
3+y*5;
```

En C et Caml, cela est très similaire : en C par exemple :

```
3+y*5;
```

En Caml :

```
3+(!y)*5
```

A part la syntaxe, cela est très similaire : en Caml, on a simplement besoin d'utiliser ! pour récupérer la valeur de la variable (mutable) `y`.

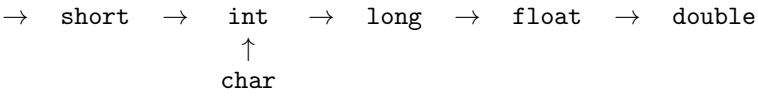
Une instruction est un mot simple dans un langage de programmation, qui fait une action. C'est un concept qui est donc différent d'une expression. On a par exemple comme instructions : l'affectation, le test, la boucle. On en voit la syntaxe et la sémantique dans les sections suivantes.

2.3.1 L'affectation

variable `=` *expression*

Cette expression a pour effet d'évaluer *expression* et d'affecter la valeur obtenue à *variable*. De plus, cette expression possède une valeur, qui est celle d'*expression*. Ainsi, l'expression `i = 5` vaut 5.

L'affectation effectue une *conversion de type implicite* : la valeur de l'expression (terme de droite) est convertie dans le type du terme de gauche. Cette conversion est possible uniquement si elle s'effectue vers un type de taille supérieure à celle du type de départ :



Ces conversions sont sans perte de précision, sauf celles de `int` et `long` en `float` et `double` qui peuvent parfois entraîner des arrondis.

2.3.2 Le branchement conditionnel

La forme la plus générale est :

```
if ( expression )
  {
    instruction-1
  }
else
  {
    instruction-2
  }
```

expression est évaluée. Si elle vaut `true`, *instruction-1* est exécutée, sinon *instruction-2* est exécutée.

Le bloc

```
else
  {
    instruction-2
  }
```

est facultatif. Chaque *instruction* peut ici être un bloc d'instructions.
Par ailleurs, on peut imbriquer un nombre quelconque de tests, ce qui conduit à :

```
if ( expression-1 )
  {
    instruction-1
  }
else if ( expression-2 )
  {
    instruction-2
         ⋮
  }
else if ( expression-n )
  {
    instruction-n
  }
else
  {
    instruction-∞
  }
```

2.3.3 Les boucles

Boucle `while`

La syntaxe de `while` est la suivante :

```
while ( expression )
  {
   instruction
  }
```

Tant que *expression* est vérifiée (*i.e.*, vaut `true`), *instruction* est exécutée.

Boucle `do--while`

Il peut arriver que l'on ne veuille effectuer le test de continuation qu'après avoir exécuté l'instruction. Dans ce cas, on utilise la boucle `do--while`. Sa syntaxe est

```
do
  {
    instruction
  }
while ( expression );
```

Ici, *instruction* sera exécutée tant que *expression* vaut `true`. Cela signifie donc que *instruction* est toujours exécutée au moins une fois.

Boucle `for`

La syntaxe de `for` est :

```
for ( expr-1 ; expr-2 ; expr-3)
  {
    instruction
  }
```

On doit comprendre cette boucle ainsi : *expr-1* est la condition initiale de la boucle, *expr-2* est la condition de poursuite de la boucle et *expr-3* est une expression exécutée après chaque tour de boucle. L'idée est que *expr-1* représente l'état initial d'un compteur dont la valeur sera modifiée après chaque tour par *expr-3*, et la boucle continue tant que la condition *expr-2* est vérifiée.

Par exemple,

```
for (int i=0; i<10; i=i+1)
  {
   System.out.println("Bonjour");
  }
```

affichera à l'écran 10 fois le mot Bonjour suivi d'un retour à la ligne.

Une version équivalente avec la boucle `while` serait :

```
int i = 0;
while (i < 10)
   {
    System.out.println("Bonjour");
    i = i+1;
   }
```

2.4 Sémantique élémentaire

On va commencer à introduire une manière formelle de décrire l'exécution d'instructions dans les langages de programmation. Il s'agit de la *sémantique* des langages de programmation.

A chaque variable $v \in \text{Var}$, on veut associer une valeur. On définit une fonction $\rho \in \text{Env}$:

$$\rho : \text{Var} \to \text{Val}$$

où Val est l'ensemble des valeurs machines ($\mathbb{Z}$ ici pour simplifier) et Env est l'ensemble des environnements. A chaque expression arithmétique $e \in \text{AExpr}$, on veut associer une valeur :

$$[\![e]\!]\rho \in \text{Val}$$

de l'expression e dans l'environnement ρ. A chaque instruction du programme p (et donc au programme lui-même), on veut associer un environnement $[\![p]\!]\rho$, après exécution, dans l'environnement ρ.

2.4.1 Sémantique des expressions

On donne ainsi des règles inductives pour donner la sémantique, tout d'abord, des expressions :

- Constantes $n \in \text{Val}$: $[\![n]\!]\rho = n$
- Variables $v \in \text{Var}$: $[\![v]\!]\rho = \rho(v)$
- Addition de deux sous-expressions $a_0, a_1 \in \text{AExpr}$: $[\![a_0 + a_1]\!]\rho = [\![a_0]\!]\rho + [\![a_1]\!]\rho$
- $[\![a_0 - a_1]\!]\rho = [\![a_0]\!]\rho - [\![a_1]\!]\rho$
- $[\![a_0 * a_1]\!]\rho = ([\![a_0]\!]\rho) * ([\![a_1]\!]\rho)$
- etc.

On l'appelle *sémantique dénotationnelle* (ou *compositionnelle*).

Donnons-en un exemple : dans l'environnement ρ où $\rho(y) = 3$:

$$\begin{aligned} [\![3 * y + 5]\!]\rho &= [\![3 * y]\!]\rho + [\![5]\!]\rho \\ &= [\![3 * y]\!]\rho + 5 \\ &= ([\![3]\!]\rho) * ([\![y]\!]\rho) + 5 \\ &= 3 * 3 + 5 \\ &= 14 \end{aligned}$$

Attention, tout ceci est valide tant que l'on considère des expressions sans *effet de bord*. Les effets de bord sont permis en JAVA :

```
int x = (y=1)+1;
```

Cela renvoie l'environnement : ρ avec $\rho(y) = 1$, $\rho(x) = 2$, mais n'est pas permis dans notre sémantique élémentaire.

2.4.2 Sémantique des instructions élémentaires

Voyons maintenant la sémantique de notre première instruction : l'affectation.

```
x = expr;
```

Par exemple :

```
x = 3+y*5;
```

Cette instruction, en simplifiant un peu les mécanismes sous-jacents, lit les valeurs de variables dont dépend l'expression puis calcule l'expression (avec ou sans *effet de bord*), et remplace la valeur contenue dans `x` par celle calculée.

Remarque importante : L'affectation est aussi une expression dans la plupart des langages impératifs, dont Java, contrairement à ce que l'on a défini dans notre sémantique simplifiée. Donc `x=expr` a une valeur, qui est celle de l'expression.

Dans notre sémantique simplifiée, où les affectations ne sont pas des expressions, la sémantique de l'affectation peut s'écrire de la façon suivante. On associe à l'instruction $v = e$ son interprétation $[\![v = e]\!]$ qui est une fonction qui à chaque environnement ρ va retourner le nouvel environnement, après affectation. On écrit ainsi : pour $v \in \text{Var}$; $e \in \text{AExpr}$, et $\rho \in \text{Env}$:

$$[\![v = e]\!]\rho = \rho[[\![e]\!]\rho/v]$$

"Crochets sémantiques (ou interprétation de) $v = e$ appliqué à ρ est égal à ρ dans lequel l'image de v est remplacée par crochets sémantiques de l'expression e, appliqué à ρ"

L'opération [./.] est décrite plus bas : pour $u, v \in \text{Var}$ et $n \in \text{Val}$:

$$\rho[n/v](u) = \begin{cases} \rho(u) & \text{si } u \neq v \\ n & \text{si } u = v \end{cases}$$

Ce nouvel environnement est le même que ρ, mis à part pour la variable u, dont l'ancienne valeur est « écrasée » par l'affectation à n.

On verra la sémantique d'autres instructions plus tard, en particulier au chapitre 6. Nous introduisons ces premières notations ici afin de pouvoir introduire précisément dans ce chapitre et le suivant, les notions d'adresse et de valeur.

Donnons pour l'instant un exemple simple d'interprétation d'une affectation simple. Considérons $x = 3 * y + 5$ dans l'environnement ρ tel que $\rho(y) = 3$ et $\rho(x) = 1$, on obtient l'environnement σ avec

$$\sigma(u) = \begin{cases} \rho(u) = 3 & \text{si } u = y \\ [\![3 * y + 5]\!]\rho = 14 & \text{si } u = x \end{cases}$$

2.5 Les tableaux

Introduisons maintenant dans la sémantique de l'affectation un type de données un peu plus compliqué, les tableaux. Les tableaux sont un cas particulier de type *produit*, que l'on traite plus généralement au chapitre 3. Un tableau peut être vu comme un ensemble de variables pour chaque indice 1D, 2D, ou multi-dimensionnel. Dans ce chapitre, on va juste considérer les tableaux 1D (c'est-à-dire les vecteurs).

À la déclaration d'un tableau, la taille n'est pas forcément connue, aucun élément n'est encore alloué, contrairement aux types de données « scalaires » tels les entiers, ou les flottants :

```
int[] tab;
```

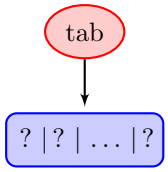
On peut donc montrer ce qui se passe en mémoire, par un petit graphe :

tab est crée, a une location mémoire associée, mais qui n'est pas initialisée, qui ne contient donc pas de référence (c'est-à-dire d'adresse mémoire) à un bloc de mémoire pouvant contenir tous les éléments du tableau.

Il faut maintenant effectuer l'allocation du tableau : pour ce faire, on fixe la taille une fois pour toutes, et les éléments sont alloués :

```
tab = new int[10];
```

On a donc maintenant en mémoire la situation suivante :

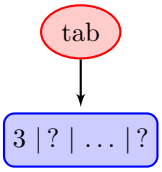


(en fait, il y a des valeurs par défaut en Java, ici ? est égal à 0)

Maintenant, on peut procéder à l'affectation des éléments de tableau. Les *indices* vont ici de 0 à 9 (=`tab.length`-1) :

```
tab[0] = 3;
```

On a donc maintenant en mémoire :



On peut aussi faire les deux en même temps :

```
int[] tab = {3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

Par contre, on n'a pas le droit d'écrire :

```
int[] tab;
tab = {3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

2.6 Sémantique des références

Il va falloir raffiner notre sémantique élémentaire de l'affectation de la section 2.4 pour tenir compte des références, ce aussi bien pour donner une sémantique à l'affectation des tableaux que pour l'affectation des variables mutables.

La variable `tab` est en fait un *pointeur* (ou *référence*, ou *adresse* mémoire) sur le début du bloc mémoire contenant toutes les entrées du tableau. Il nous faut donc un environnement un peu plus compliqué, pour pouvoir parler des variables scalaires, et des tableaux.

Soit Loc l'ensemble des adresses mémoires (numérotation des mots 32 ou 64 bits dans la mémoire de quelques Go de votre ordinateur), sous-ensemble de $\mathbb{N}$. On commence par généraliser les environnements, qui étaient $\rho : \text{Var} \to \text{Val}$ en : $\rho : (\text{Var} \cup \text{Loc}) \to (\text{Val} \cup \text{Loc})$ pour pouvoir associer à des variables (comme `tab`) des adresses, et à des adresses, des valeurs. On verra en section 2.13 que l'on devra encore généraliser un peu plus les environnements, pour pouvoir parler de structures de données dynamiques comme les listes ou les arbres, et de déréférencements plus compliqués. Ici Var représente les adresses allouées statiquement, en *pile* (les déclarations de variables scalaires), Loc représente les adresses allouées dynamiquement, en général sur le *tas*.

On notera adr_t pour l'adresse t, pour qu'il n'y ait pas de confusion avec Val.

En première approximation, pour le code suivant :

```
let x=4 in y=x+3
```

On arrive à l'environnement final : $\rho(x) = 4$, $\rho(y) = 7$. Alors que pour le code (avec la version mutable de `x`) :

```
let x=ref 4 in y=!x+3
```

On arrive à l'environnement final : $\rho(x) = adr_u$, $\rho(adr_u) = 4$, $\rho(y) = 7$.

Dans le cas mutable, x contient une valeur de type « adresse » (dans Loc) ; dans le cas final, x contient une valeur « scalaire ».

Revenons maintenant à la sémantique des tableaux. On considère d'abord la déclaration en Java :

```
int [] tab;
```

L'environnement est après cette déclaration : $\rho(tab) = \bot$. On utilise ici un nouveau symbole, qui n'est égal à aucune valeur, et que l'on lit « bottom », dont nous expliciterons mieux l'origine au chapitre 6. Il nous permet de coder le fait qu'aucune valeur n'est associée encore à `tab`. Venons-en à l'allocation maintenant :

```
tab = new int[10];
```

On obtient l'environnement : $\rho(tab) = adr_t$ où adr_t est l'adresse de la première case mémoire allouée pour `tab`. Sont créées également les adresses adr_{t+1}, adr_{t+2},...,adr_{t+9}, mais pas les valeurs correspondantes : $\rho(adr_t) = \perp$, $\rho(adr_{t+1}) = \perp$, ..., $\rho(adr_{t+9}) = \perp$.

De façon générale, voici les règles définissant la sémantique des tableaux 1D. Pour $tab \in \text{Var}$, $i \in \mathbb{N}$, $e \in \text{AExpr}$, et $\rho \in \text{Env}$:

$$[\![tab[i] = e]\!]\rho = \rho[[\![e]\!]\rho/(\rho(tab) + i)]$$

Et pour le calcul dans une expression $e \in \text{AExpr}$, il faut rajouter dans les expressions arithmétiques le fait de lire une entrée d'un tableau :

$$[\![tab[i]]\!]\rho = \rho(\rho(tab) + i)$$

Cette règle s'interprète de la façon suivante : on lit l'adresse de `tab`, on la décale de `i`, puis on lit la valeur à cette adresse - ici on suppose qu'on n'a que des tableaux 1D de valeurs scalaires.

Donnons-en un exemple. Considérons le code Java suivant :

```
int [] tab = new int[10];
tab[0] = 0; tab[1] = 1; tab[2] = 2; ... tab[9] = 9;
```

Ont été créées donc, dans l'environnement, les adresses adr_t à adr_{t+9} et $\rho(tab) = adr_t$. On termine avec en plus $\rho(adr_{t+i}) = i$ pour $i = 0, \ldots, 9$.

2.7 La séquence d'instructions

La séquence d'instructions consiste juste à mettre en série une instruction puis une deuxième (et éventuellement ainsi de suite). En Java ou C, par exemple, voici deux instructions d'affectation en séquence :

```
x = 1; y = x+1;
```

On exécute `x = 1` puis `y = x+1`.

Et enfin en Caml :

```
let x = 1 in let y = x+1 in p;;
```

Pour P et Q deux programmes composés séquentiellement, la sémantique est facile à écrire :

$$[\![P; Q]\!]\rho = [\![Q]\!]\,([\![P]\!]\rho)$$

"La sémantique de P puis Q, appliquée à ρ est la sémantique de Q appliquée à l'interprétation de P dans l'environnement ρ"

C'est précisément cette règle qui fait qu'on appelle cette sémantique, *compositionnelle.*

2.8 Conditionnelles

Les conditionnelles permettent de choisir entre deux exécutions possibles, selon le résultat d'un test. En Java, et C cela s'écrit :

```
if (b)
  p1
else
  p2
```

On calcule la valeur booléenne de `b` ; si c'est `true` on exécute `p1`, si c'est `false` on exécute `p2`. Par exemple :

```
if (x == 0)
  y = -1;
else
  y = 1;
```

Attention à l'erreur courante : `if (x = true) p1 else p2` fera toujours `p1`, alors qu'on voulait écrire a priori `if (x == true) ...` Attention également à bien mettre dans un bloc toutes les instructions à exécuter dans la branche vraie ou dans la branche fausse, quand il y en a plusieurs, c'est-à-dire de les mettre entre accolades.

En Caml cela est très similaire syntaxiquement :

```
if b then p1 else p2
```

Les expressions pour les tests booléens sont assez simples syntaxiquement, en Java, C et Caml. Cela peut être n'importe quelles conditions séparées par des connecteurs logiques :

```
&&, ||, (expr1 == expr2), ...
```

Nommons BExpr l'ensemble des expressions booléennes permises dans notre langage ; les valeurs Val sont supposées comprendre également les booléens $true$, $false$. Alors on peut écrire une sémantique des expressions booléennes $[\![.]\!]$: BExpr $\to$ Env $\to \{true, false\}$, comme on l'avait fait pour les expressions arithmétiques, inductivement sur la syntaxe :

- $[\![true]\!]\rho = true$
- $[\![false]\!]\rho = false$
- Si $a_0, a_1 \in$ AExpr, $[\![a_0 == a_1]\!]\rho = \begin{cases} true & \text{si } [\![a_0]\!]\rho = [\![a_1]\!]\rho \\ false & \text{sinon} \end{cases}$
- Si $a_0, a_1 \in$ AExpr, $[\![a_0 < a_1]\!]\rho = \begin{cases} true & \text{si } [\![a_0]\!]\rho < [\![a_1]\!]\rho \\ false & \text{sinon} \end{cases}$
- De même pour $a_0 <= a_1$ etc.

On peut maintenant écrire la sémantique de l'instruction conditionnelle :

$$[\![\text{if } b \text{ } p1 \text{ else } p2]\!]\rho = \begin{cases} [\![p1]\!]\rho & \text{si } [\![b]\!]\rho = true \\ [\![p2]\!]\rho & \text{si } [\![b]\!]\rho = false \end{cases}$$

2.9 Itération – la boucle

Dans tous les langages impératifs, on a une façon d'itérer des calculs par une construction de type *boucle*. Par exemple, en Java et C :

```
while (b)
  p
```

En voici un exemple pratique, qui itère l'incrémentation de x de 2, jusqu'à ce que x soit supérieur ou égal à 1000 :

```
while (x<1000)
  x=x+2;
```

En Caml, on a une construction similaire :

```
while b do p;
```

`b` est appelé la *condition de terminaison*. La boucle est un *raccourci* pour tous ses déroulements finis, on verra au chapitre 6 sa sémantique formelle, par *approximations successives* :

```
if (b) {
  p
  if (b) {
    p
    if (b) {
      p
      if (b) {
        p
        ...
      }
    }
  }
}
```

... et ainsi de suite...

Les approximations successives forment une suite finie quand la condition de terminaison devient vraie en un temps fini. Mais toutes les boucles ne sont pas finies, par exemple :

```
int x=3;
while (x<=1000)
  x=2;
```

En général, il est difficile de prouver la terminaison (ce sera formalisé dans un cadre précis au chapitre 8). Par exemple : la terminaison du code suivant reposerait sur une preuve de la conjecture de Syracuse (cf. `http://fr.wikipedia.org/wiki/Conjecture_de_Syracuse`) :

```
while (x > 1) {
  if (x % 2 == 0)
    x = x/2;
  else
    x = 3*x+1; }
```

Toutes les boucles ne terminent pas forcément, contrairement à celle plus haut (ce qui est certes difficile à prouver). Donc dans notre sémantique, quand P contient une boucle par exemple, $[\![P]\!]\rho$ ne peut pas toujours renvoyer un $\sigma \in \text{Env}$. En fait, $[\![P]\!]$ ne peut être qu'une fonction partielle, ou à recodage près, une fonction totale à valeur dans $\text{Env} \cup \{\bot\}$, que l'on note $\text{Env}_\bot$. Par abus de notation, on notera $\bot \in \text{Env}_\bot$, l'environnement tel que $\forall x, \bot(x) = \bot$. On aura alors pour un programme P : $[\![P]\!]\rho = \bot$ si P ne termine pas dans l'environnement ρ.

Une autre syntaxe pour les boucles est donnée par la boucle `for`. Par exemple, le code suivant :

```
x=0;
while (x < 100) {
  y = 2*x;
  x = x+1;
}
```

s'écrit plus naturellement (c'est en particulier, ici, une boucle dont le nombre d'itérations est fini, connu statiquement) par une boucle `for` :

```
for (x=0; x<100; x=x+1)
  y = 2*x;
```

Dans une telle boucle, un *indice de boucle* (ici, `x`) est d'abord initialisé (`x=0`), on effectue un test pour savoir si on continue l'itération ou pas (`x<100`), et enfin, on donne une instruction qui est exécutée à chaque itération, qui consiste assez souvent en une incrémentation de l'indice de boucle (`x=x+1`).

On revient en détail, beaucoup plus formellement, sur la sémantique des boucles, au chapitre 6.

2.10 Fonctions

Imaginons le programme suivant, écrit en Java ou en C :

```
/* Calcul de x */
...
  if (x<0) sgnx=-1;
  if (x==0) sgnx=0; else sgnx=1;
...
/* Calcul de y */
  if (y<0) sgny=-1;
  if (y==0) sgny=0; else sgny=1;
```

Ce code n'est pas pratique, il y a redite pour le calcul du signe, et cela est une cause potentielle d'erreur. Il est de bon principe que de structurer le code par des appels à des fonctions, de plus en plus élémentaires. En fait, l'idée de fonction dans les langages de programmation est plus profond que cela, en particulier dans le cadre de la récursion et des langages fonctionnels (Caml), où le credo est : « Functions as first-class citizens » (Christopher Strachey, vers 1960). On verra cela plus en détail aux chapitres 5 et 9.

Une fonction peut : prendre en compte des données, les *arguments*, retourner une valeur, *le retour de la fonction*, et effectuer une action (afficher quelque chose, modifier la mémoire), l'*effet de bord*. Dans la plupart des langages, on indique le type des arguments et du retour de fonction (sauf en général dans Caml, où il y a inférence de types – cf. chapitre 9), par exemple `int`, `float`.

On appelle généralement *procédure* une fonction sans retour de valeur. Généralement on identifie cela avec le fait de retourner une valeur dans un « type » *vide*. En Java, et C, ce type est nommé par le mot clé `void`. En Caml, c'est le type spécial `unit`.

2.11 Passage d'arguments aux fonctions

Décrivons maintenant le mécanisme de passage d'arguments dans les fonctions Java (ou C). Considérons le code Java, ou C :

```
static int signe(int x) {
  if (x<0) ...
```

Ou encore en Caml :

```
let signe x = if (x<0) then ...
```

Il existe également des fonctions prédéfinies, par exemple dans les *paquetages* (ou « modules », « bibliothèques ») de fonctions mathématiques `Math`, en particulier, par exemple la fonction racine carrée : `Math.sqrt(...)`.

Reprenons le code précédent, où l'on remplace le code de calcul du signe, dupliqué, par un appel à une fonction `signe` :

```
/* Calcul de x */
...
sgnx = signe(x);
/* Calcul de y */
...
sgny = signe(y);
...
```

Les arguments de la fonction `signe` sont évalués avant de démarrer l'exécution de son code même (on aurait pu faire `signe(expr donnant x)`). Ce sont les valeurs des arguments ainsi évalués qui sont transmises à la fonction appelée. C'est ce que l'on appelle le « passage par valeur ».

En Caml, il s'agit de même d'un passage par valeur, dans le code similaire suivant :

```
...
in
let sgnx = signe x in ... ;;
```

Passons maintenant au retour de fonction. En Java, la fonction `signe` est écrite comme suit :

```
static int signe(int x) {
  if (x<0) return -1;
  if (x==0) return 0;
  return 1; }
```

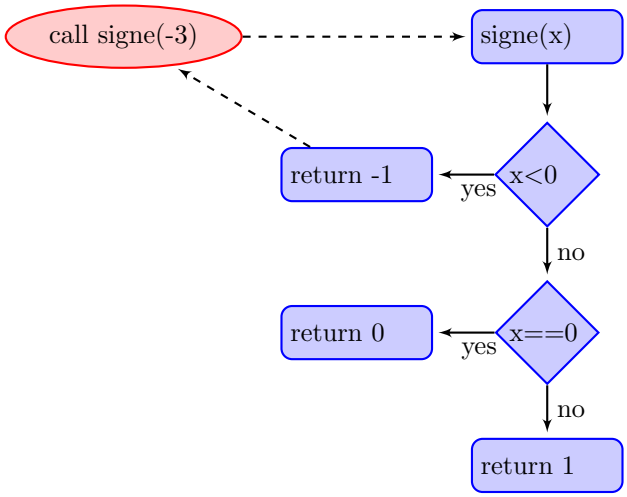
De même en C :

```
int signe(int x) {
  if (x<0) return -1;
  if (x==0) return 0;
return 1; }
```

Et en Caml :

```
let signe x = if (x<0) then -1.0 else
              if (x==0) then 0.0 else 1.0;;
```

La sémantique du retour de fonction est la suivante ; `return` interrompt le déroulement de la fonction :



Prenons maintenant un nouvel exemple de code, qui intervertit les valeurs des variables `a` et `b`, en utilisant la variable intermédiaire `c` :

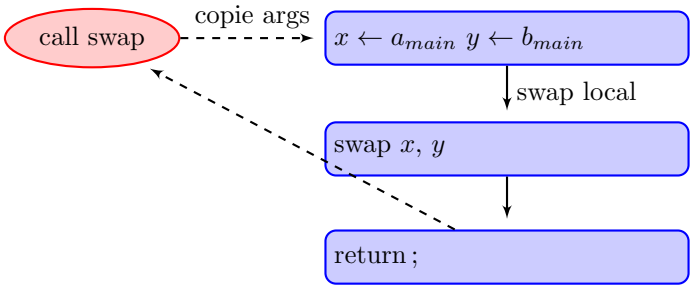
```
class troisVerres {
  public static void main(String[] args) {
    int a=1;
    int b=2;
    int c=a;
    a=b;
    b=c;
    System.out.println("a="+a);
    System.out.println("b="+b);
  }
}
```

L'exécution de `java troisVerres` affiche bien `a=2`, `b=1`.

Ecrivons maintenant un nouveau code par appel de fonction :

```
class troisVerres {
  static void swap(int x, int y) {
    int z=x; x=y; y=z;
}

  public static void main(String[] args) {
    int a=1;
    int b=2;
    swap(a,b);
    System.out.println("a="+a);
    System.out.println("b="+b); }
}
```

L'exécution de `java troisVerres` affiche `a=1`, `b=2`, qui n'est pas le résultat espéré. Ceci est bien sûr du au mécanisme de passage par *valeur*. En Java, C, Caml, on n'a par défaut que le *passage par valeur* (des variables scalaires). La fonction travaille dans ce cas sur une copie des variables passées en paramètre. :



La modification des valeurs des variables x et y n'a aucune « portée » au delà de la fonction : il n'y a aucun mécanisme de recopie des variables x et y, « locales » à la fonction *swap*, vers a et b au retour de fonction.

2.12 Variables locales, variables globales

Ceci mène naturellement au concept de portée des variables. On a par exemple des variables connues du programme entier, ce que l'on appelle des variables globales :

```
...
static int x; x=3; x=0;
...
public static void main(String args) {
  ...
}
```

Elles sont définies hors de toute fonction, y compris la fonction principale `main`. Dans le code plus haut, `x` vaut 0 à la fin de l'exécution de ce programme. Inversement, on peut définir des variables locales à un bloc ou à une fonction :

```
static void reset() {
  int x=0;
}

public static void main(String args[]) {
  int x; x=3; reset(); ...
}
```

Le symbole `x` dans ce cas recouvre deux variables :
- une variable locale à `main`, qui vaut 3 et n'est pas modifiée ;
- une variable locale à la fonction `reset`, qui vaut 0, et qui n'est connue que dans le corps de la fonction `reset`

Cette variable locale *cache* la variable `x` de `main` dans le corps de la fonction `reset`, mais ne l'écrase pas (location mémoire, contenant la valeur correspondante, distincte, nous y reviendrons).

2.12.1 Passages de tableaux en paramètres

Regardons maintenant le cas du passage de tableaux en paramètre, avec ce petit code exemple :

```
static void tab_plus_un(int[] y){
  int n = y.length;
  for (int i = 0; i < n; i=i+1)
    y[i] = y[i]+1;
}

public static void main(String[] args) {
  int[] x = new int[5];
  for (int i = 0; i < 5; i=i+1)
    x[i] = i;
  tab_plus_un(x);
  for (int i = 0; i < 5; i=i+1)
    System.out.println("x["+i+"]="+x[i]);
}
```

Son exécution termine après avoir affiché :

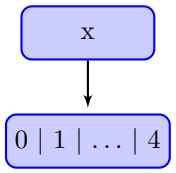
```
x[0]=1 x[1]=2 x[2]=3 x[3]=4 x[4]=5
```

ce qui semble contradictoire avec la sémantique du passage d'arguments par valeur. Cela ne l'est pas : comme on a vu à la Section 2.6, un tableau est en fait un pointeur, c'est-à-dire que la valeur d'une variable de type tableau est (au moins en C, C++ et Java) une adresse mémoire. Ainsi, le passage par valeur d'une variable tableau passe une référence (adresse) sur le bloc mémoire contenant les différentes entrées du tableau. Le passage de tableaux en paramètre se fait donc "par *référence*". Imaginons un instant que la valeur d'une variable de type tableau soit l'ensemble de ses entrées, il serait trop lourd de faire des recopies à l'appel de fonction, et il est naturel de ne passer que sa *référence* (pointeur, c'est-à-dire adresse mémoire) au tableau, à l'appel de la fonction. Cela permet de faire des modifications *en place* comme on l'a fait dans ce programme.

Voici ce qui se passe à chaque étape d'exécution :

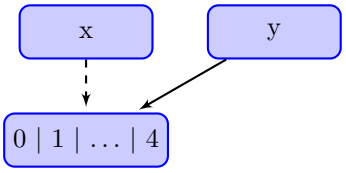
A la fin de la boucle d'initialisation, le tableau est alloué et rempli des valeurs de 0 à 4, dans un bloc contigu de mémoire, la variable `x` contenant en fait l'adresse du début de bloc alloué (ou de façon équivalente, de `x[0]`) :

```
for (i=0;i<5;i=i+1)
  x[i] = i;
```



Puis à l'appel de la fonction `tab_plus_un`, la variable locale `y` à cette fonction va contenir l'adresse mémoire du bloc stockant `x`. La case mémoire `y` est elle-même distincte de la case mémoire `x` (chacune des deux ne contient qu'une valeur scalaire, pointeur, vers le même tableau `x[]`) :

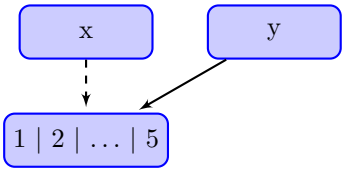
```
tab_plus_un(x) ...
```



Dit autrement, avec notre sémantique formelle : $\rho \in \text{Env}_\perp$ au moment de l'appel puis, $\rho(x) = adr_u$, $\rho(y) = adr_u$, $\rho(adr_{u+i}) = i$ pour $i = 0, \ldots, 4$.

Puis on exécute le corps de la boucle principale de la fonction `tab_plus_un`. La variable `y` pointant vers le tableau `x[]` on peut modifier *en place* les valeurs de `x[]`. On n'a à aucun moment recopié le tableau `x[]` mais seulement son adresse `x` :

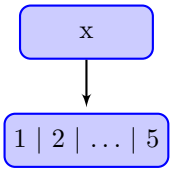
```
...
  y[i] = y[i]+1;
```



Par la sémantique que l'on vient de voir, on en déduit en effet que : $\rho' = [\![y[i] = y[i] + 1]\!]\rho = \rho[[\![y[i] + 1]\!]\rho / (\rho(y) + i)]$, et $[\![y[i] + 1]\!]\rho = \rho(\rho(y) + i) + 1 = \rho(adr_{t+i}) + 1 = i + 1$. D'où à la fin, le seul changement entre ρ et ρ' est $\rho'(adr_{t+i}) = i + 1$.

Et on termine l'exécution avec :

```
...
System.out.println ...
```



Par la sémantique que l'on vient de voir : $[\![x[i]]\!]\rho' = \rho'(\rho'(x) + i)$. Donc, $[\![x[i]]\!]\rho' = \rho'(adr_{t+i}) = i + 1$.

2.13 Références, pointeurs, objets

Reprenons maintenant le code de la classe `troisVerres`, où on utilise une fonction pour faire l'échange de valeurs entre deux variables :

```
class troisVerres {
  static void swap(int x, int y) {
    int z=x; x=y; y=z;
}

  public static void main(String[] args) {
    int a=1;
    int b=2;
    swap(a,b);
    System.out.println("a="+a);
    System.out.println("b="+b);
  }
}
```

On termine toujours avec le résultat erroné : `a=1`, `b=2`.

De même pour le code Caml :

```
let swap x y =
  let z = ref 0 in (z:=!x; x:=!y; y:=!z)

a:=1;
b:=2;
swap a b;
print_int !a;
print_newline();
print_int !b;
print_newline();
```

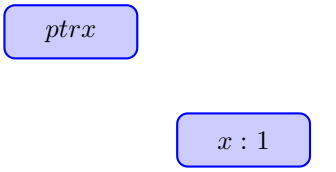
Ce que l'on voudrait, en fait, pour faire marcher ce code, c'est passer la *référence* aux variables `a` et `b` plutôt que simplement leurs valeurs, afin de pouvoir modifier les valeurs *en place*. En Pascal c'était possible :

```
procedure toto(var integer i, integer j);
begin
  ...
end
```

Dans le code plus haut, `i` est passée par référence, `j` est passée par valeur. En Java, C, Caml, il va nous falloir simuler ce passage, par le passage par valeur.

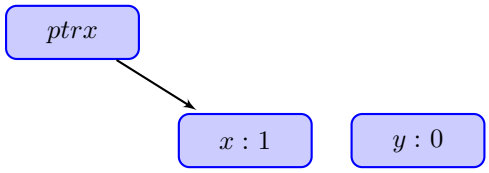
Commençons par du C, où la manipulation d'adresse (pointeur) est explicite. Pour trouver la location d'une variable à partir de son nom, en C, on utilise le mot clé &. Voici les différentes étapes d'exécution d'un programme simple de manipulation d'un pointeur sur un entier `ptrx` :

```
int x, y;
int *ptrx;

x = 1;
```



Puis on récupère l'adresse mémoire de `x` que l'on stocke dans `ptrx` :

```
y = 0;
ptrx = &x;
```

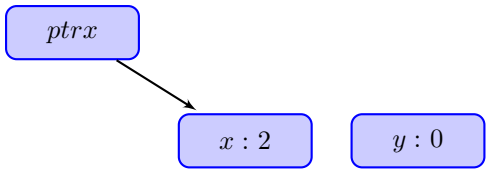


Le pointeur `ptrx` contient ainsi l'*adresse* en mémoire où est stockée la valeur de la variable `x`. Ensuite, on écrase la valeur courante de `y`, par la valeur pointée par la variables `ptrx` :

```
y=*ptrx;
printf("y=%d\n",y);
```

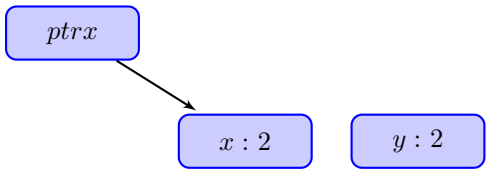
Ceci affiche bien sûr `y=1`.
Si on avait plutôt fait :

```
int x, y;
int *ptrx;

x = 1;
y = 0;
ptrx = &x;

x = 2;
```



Puis :

```
y = *ptrx;
printf("y=%d\n",y);
```



On aurait affiché `y=2`.

Enfin, pour en terminer avec la syntaxe, notons que `*` est l'homologue du `!` de Caml, et `*t=u` l'homologue de `t:=u` de Caml.

En supposant que x est en quelque sorte « bien typé » dans l'environnement ρ : $\rho(x) \in \text{Loc}$, on peut donner une sémantique à l'opération $*$. Dans une expression (AExpr), on aura :

$$[\![*x]\!]\rho = \rho([\![x]\!]\rho)$$

Cette écriture, au lieu d'écrire directement $\rho(x)$ permet de généraliser x à des expressions de certains types, comme $\&y$, ce que l'on verra dans un exemple, plus bas.

Dans une affectation, la sémantique est la suivante :

$$[\![*x = e]\!]\rho = \rho[[\![e]\!]\rho/\rho(x)]$$

Notez le niveau d'indirection, encore, dans cette règle sémantique.

Maintenant, pour être complet, il nous faut généraliser encore une fois notre notion d'environnement. Au lieu de $\rho : (\mathrm{Var} \cup \mathrm{Loc}) \to (\mathrm{Val} \cup \mathrm{Loc})$, il nous faut considérer maintenant que nous pouvoir avoir une variable qui pointe sur une autre (et ainsi de suite), avant, au final, de pointer sur une valeur. Donc il est possible que $\rho(x)$, pour $\rho \in \mathrm{Var}$ par exemple, soit également une variable. On écrit donc maintenant les environnements $\rho : (\mathrm{Var} \cup \mathrm{Loc}) \to (\mathrm{Val} \cup \mathrm{Var} \cup \mathrm{Loc})$. Ainsi pour $x \in \mathrm{Var}$, on aura :

$$[\![\&x]\!]\rho = x \in \mathrm{Var}$$

(remarque : on ne peut utiliser $\&x$ à gauche d'une affectation).

Donnons maintenant un exemple simple de cette nouvelle sémantique :

$$\begin{array}{rcl} [\![*(\&x)]\!]\rho & = & \rho([\![\&x]\!]\rho) \\ & = & \rho(x) \end{array}$$

En Java, un objet est en fait un pointeur (vers la location mémoire contenant toutes les informations sur l'objet). Un scalaire est une valeur, pas un pointeur vers une valeur.

On peut ainsi reprendre, cette fois-ci correctement, le code d'échange de valeurs, en passant en argument à `swap`, par valeur, les références à des objets contenant une unique valeur entière :

```
class Intval {
  public int val;
  ...

class troisVerres {
 public static void main
        (String[] args) {
  Intval a=new Intval();
  a.val=1;
  Intval b=new Intval();
  b.val=2;
  swap(a,b);
  System.out.println("a="+a.val);
  System.out.println("b="+b.val); }
```

Dans ce programme, `int val;` dans `class Intval` n'est pas `static`. Cela veut dire que plusieurs « éléments » de « type » `Intval` peuvent être définis, avec des valeurs entières `val` différentes (on reviendra la-dessus au chapitre 4).

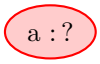
L'instruction `new` (comme sur les tableaux) permet d'allouer un nouvel élément de type `Intval`.

Remarque sur les *classes enveloppantes* : des programmes sont déjà définis en Java, permettant de manipuler des références sur des scalaires (comme `Intval` que l'on a défini plus haut), par exemple `Integer` (mais ceci n'est pas utilisable ici car `Integer` est non-mutable).

Quand on fait `a = new Intval();` dans la fonction `main`, on crée la case mémoire associée à `a` (pouvant contenir `val`, entier). On fait appel en fait à une fonction cachée : le constructeur par défaut `Intval()`.

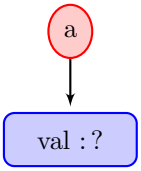
Donc la déclaration d'un objet Java a pour effet de créer un simple emplacement mémoire pour stocker une adresse :

```
Intval a;
```



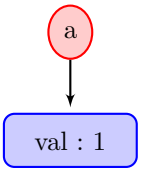
Puis vient l'allocation mémoire, qui permet de trouver une adresse valide en mémoire pour stocker l'objet lui-même :

```
Intval a;
a = new Intval();
```



Et enfin l'affectation d'un valeur au contenu de cet objet :

```
Intval a;
a = new Intval();
a.val = 1;
```



On peut aussi utiliser un raccourci pour l'*allocation et affectation*, en définissant un nouveau *constructeur* :

```
a = new Intval();
a.val=1;
b = new Intval();
b.val=2;
```

$\longrightarrow$

```
public Intval(int u) { val=u; }

a = new Intval(1);
b = new Intval(2);
```

Cette nouvelle fonction (*constructeur*) doit s'appeler du même nom que le programme et ne pas avoir de type de retour (implicitement, cela crée une location mémoire avec le type dont le nom est celui du constructeur) ; on l'appelle par `new ...`. On peut définir autant de constructeurs que l'on veut, pourvu qu'ils prennent des arguments différents, comme par exemple, le constructeur par défaut pour `Intval`, sans argument, et le constructeur défini plus haut `Intval(int u)`. Notons aussi qu'il existe toujours un *constructeur* par défaut sans argument, pas besoin de le définir.

Pour en terminer, voici comment coder l'échange de façon correcte :

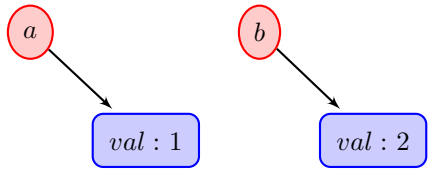
```
static void good_swap(Intval u,
                      Intval v) {
  int w = u.val;
  u.val = v.val; v.val = w;
}
```

Une mauvaise version est :

```
static void wrong_swap(Intval u,
                       Intval v) {
  Intval w = u;
  u = v; v = w;
}
```

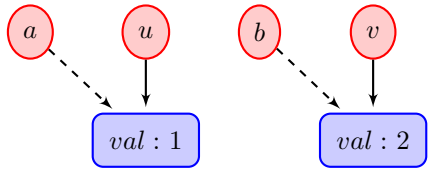
Les différentes étapes de l'exécution de `wrong_swap` sont :

```
a = new Intval();
a.val=1;
b = new Intval();
b.val=2;
```



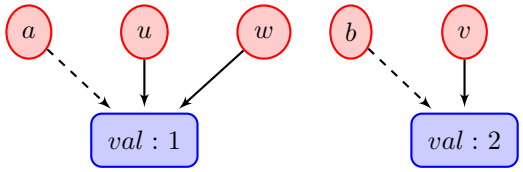
Puis,

```
a = new Intval();
a.val=1;
b = new Intval();
b.val=2;
wrong_swap(a,b);
```



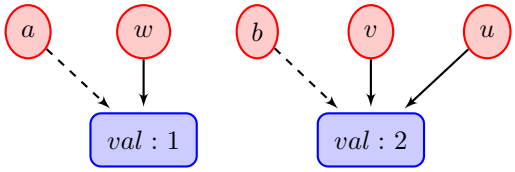
On passe par la variable intermédiaire :

```
Intval w = u;
```



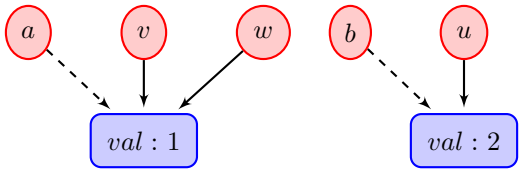
Puis :

```
u = v;
```



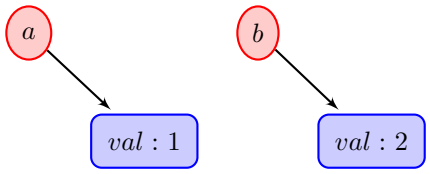
Et on termine par :

```
v = w;
```



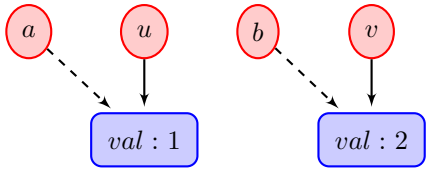
Il est clair que le résultat n'est pas le bon...
Passons maintenant en revue l'exécution de `good_swap` :

```
a = new Intval();
a.val=1;
b = new Intval();
b.val=2;
```



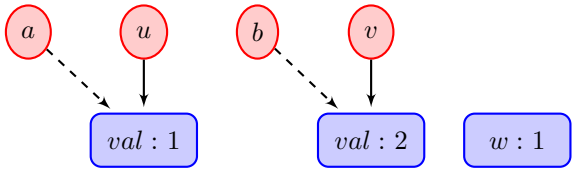
On continue par l'appel à la fonction `swap` :

```
a = new Intval();
a.val=1;
b = new Intval();
b.val=2;
good_swap(a,b);
```



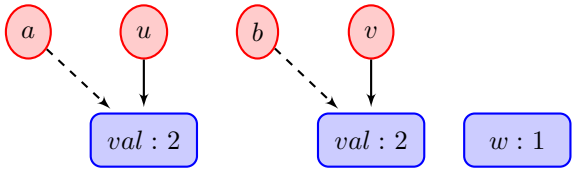
Puis l'utilisation de la variable intermédiaire :

```
int w = u.val;
```



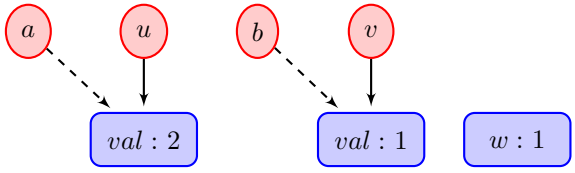
Et :

```
u.val = v.val;
```



Et enfin :

```
v.val = w;
```



Le résultat est celui que l'on souhaite.

2.14 Récapitulation : un peu de syntaxe

Terminons en récapitulant l'articulation d'un fragment purement impératif en Java, C et Caml :

En Java, on écrirait des définitions de fonctions avec leurs types (on reviendra à `static` au chapitre 4) :

```
static T f(T1 x1,...,Tn xn) p
```

En C de même, les lignes suivantes définissent une fonction `f` qui prend des arguments de type `T1`, ..., `Tn`, et renvoie un calcul (effectué par le corps de fonction `p`) de type `T` (on donne un exemple avec des types entiers à la deuxième ligne ci-dessous) :

```
T f(T1 x1,...,Tn xn) p
int f(const int x,int y) return x+1;
```

Et enfin en Caml :

```
let f x1 ... xn = t in p
let hypothenuse x y = sqrt(x*. x +. y *.y)
```

Les arguments sont alors toujours finaux.

Un programme complet Java ressemblera, par sa structure, à :

```
class Prog {
  static T1 x1 = t1;
  ...
  static Tn xn = tn;

  static ... f1(...) {
    ...
  }
  ...
  static ... fp(...) {
    ...
  }
  public static void main(String[] args) {
    ...
  }
}
```

Le programme ci-dessus définit des variables globales `x1` à `xn` puis des fonctions `f1` à `fp`. Enfin, le "programme principal" `main` est défini. C'est le point d'entrée de l'exécution de ce programme Java.

De façon similaire en C, on aurait :

```
T1 x1=t1;
...
Tn xn=tn;
```

```
... f1(...) {
  ...
}

... fp(...) {
  ...
}

int main(int argc, char **argv) {
  ...
}
```

Et en Caml :

```
(* variables globales *)
let x = ...;;
(* variables locales *)
let x = ... in ...;;
(* fonctions - non recursives *)
let f arg1 ... argn = corps de fonction;;
```

Le `main` prend en argument toutes les valeurs passées *à la ligne de commande*.

Examinons un instant la sémantique du `main` à travers un exemple, ici en Java :

```
class Essai {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i=i+1)
      System.out.println("args["+i+"]="+args[i]);
  }
}
```

Ou encore de façon équivalente, en C :

```
int main(int argc, char **argv) {
  int i;
  for (i=0; i<argc; i=i+1)
    printf("argv[%d]=%s\n",i,argv[i]);
  return 0;
}
```

(code d'erreur éventuelle, retournée par la fonction `main`)

La compilation et l'exécution de ce code Java donnent :

```
> javac essai.java
> java Essai premierarg deuxiemearg
args[0]=premierarg
args[1]=deuxiemearg
```

Et en C :

```
> gcc essai.c -o essai
> ./essai premierarg deuxiemearg
```

```
arg[0]=essai
arg[1]=premierarg
arg[2]=deuxiemearg
```

Chapitre 3

Structures de données

Ce chapitre fait le point sur les constructions de types de données composées, que l'on retrouve dans tous les principaux langages de programmation. Dans un premier temps, on définit les types produits, qui permettent de manipuler des couples, ou en général des n-uplets de valeurs. Les tableaux en sont un cas particulier. Ce faisant, on va commencer à allouer *dynamiquement* les données, et à parler de gestion mémoire, et en particulier du ramasse miette présent dans les langages modernes comme Java ou OCaml. Ces considérations nous amènent à parler de nouveaux des pointeurs, et de problèmes de représentation de données, dont le partage et l'égalité structurelle, ou physique. Un type d'algorithme utile, et classique pour gérer l'égalité de structures de données, de façon efficace, en gérant l'égalité structurelle par l'égalité physique, est le hachage, que nous décrivons, ou rappelons pour certains, rapidement. On passe alors à la construction duale des types produits, les types somme, qui permettent d'énumérer différentes possibilités de structures. Enfin, en combinant les deux, et l'allocation dynamique de structures, on en arrive aux structures de données dynamiques, les listes et les arbres en particulier.

3.1 Types produits, ou enregistrements

On a besoin de structurer les programmes, mais aussi les données. Prenons l'exemple d'un codage de points par des coordonnées cartésiennes. Soit p un point de l'espace $\mathbb{N}^3$. On peut le représenter par le triplet de ses coordonnées (x, y, z), x, y et z étant des entiers. En vocable informatique : c'est un cas particulier de *structure* ou *enregistrement*, avec 3 *champs* entiers x, y, et z.

En Java, ce type de données se définirait comme suit :

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

Et en C :

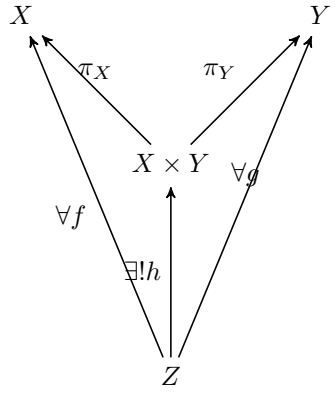
```
struct Point {
  int x;
  int y;
  int z;
};
```

Et enfin en Caml :

```
type Point = {
  mutable x : int;
  mutable y : int;
  mutable z : int;
}
```

Ceci est un cas particulier de type produit, que nous développons maintenant.

Un produit (ou une puissance ici) dans les ensembles, comme $\mathbb{N}^3 = \mathbb{N}\times\mathbb{N}\times\mathbb{N}$, est la donnée d'un ensemble (ici de triplets) et de projections sur chaque coordonnée, ainsi qu'une fonction permettant de construire un triplet à partir des 3 coordonnées. C'est la vision plus algébrique, venant de la théorie des catégories, d'un produit, et plus conforme à la vision que l'on aura des produits, dans les langages de programmation. De fait Caml veut dire « Categorical Abstract Machine Language » et la machine abstraite d'exécution d'origine était basée sur des « combinateurs catégoriques ». Ces propriétés (« universelles ») sont les suivantes :



Propriété universelle : $\forall f, g, \exists! h$, tel que

$$\begin{aligned} \pi_X \circ h &= f \\ \pi_Y \circ h &= g \end{aligned}$$

Dans les ensembles
- $X \times Y = \{(x, y) \mid x \in X, y \in Y\}$
- $\pi_X(x, y) = x$, $\pi_Y(x, y) = y$ (*sélecteurs*)
- $h(z) = (f(z), g(z))$ (*constructeur*)

En Java, on peut construire un nouveau point dans $\mathbb{N}^3$ à partir du constructeur par défaut :

```
Point p = new Point();
p.x = 1;
p.y = 2;
p.z = 3;
```

Les projections sont les suivantes en Java :

```
int xx = p.x;
int yy = p.y;
int zz = p.z;
```

En C et Caml on peut faire de même, avec une syntaxe légèrement différente :

```
struct point p = { 0, 2, 3 };
p.x = 1;
```

(construction)

```
int xx = p.x;
int yy = p.y;
int zz = p.z;
```

(projections)

```
let p = ref { x = 0; y = 2; z = 2; };;
!p.x <- 1;;
```

(construction)

```
let xx = !p.x;;
let yy = !p.y;;
let zz = !p.z;;
```

(projections)

Bien sûr, en Java, on peut définir comme on l'a déjà vu au chapitre 2, un constructeur prenant directement en argument trois entiers, pour créer un point, qui alloue et affecte en même temps un point :

```
class Point {
  int x; int y; int z;
  Point(int a, int b, int c) {
    x = a; // ou this.x = a;
    y = b; // ou this.y = b;
    z = c; // ou this.z = c;
  }
}
```

Pour utiliser le constructeur on écrit par exemple :

```
Point x = new Point(1,2,3);
```

En C et Caml il n'y a pas vraiment d'analogue direct, il y en a par contre dans leur extension orienté objets, C++ et OCaml respectivement, on en verra un peu plus au chapitre 4. Il y a donc ici un certain nombre de différences entre Java, C, Caml, pour le type `Point`. En Java il y a des constructeurs, une valeur vide (`null`) et des valeurs par défaut à la création d'un type produit (ou enregistrement). En C, il n'y a pas de constructeur, mais il y a une valeur vide (`null`) et pas toujours de valeurs par défaut. En Caml, il n'y a pas de définition de constructeur, pas de `null`, et pas de valeur par défaut.

3.2 Enregistrements et appels de fonctions

Intéressons nous au code Java suivant :

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

 Point(int u, int v, int w) {
  x = u; y = v; z = w;
 }
}

class Essai {
 static void show(Point p) {
  System.out.println(p+"=("+p.x+","+p.y+","+p.z+")");
}

 static void incr(Point p, Point q) {
  q.x += p.x; q.y += p.y; q.z += p.z;
 }

 public static void main(String[] args) {
  Point x0 = new Point(1,2,3);
  Point x1 = new Point(-1,-2,-3);
  incr(x0,x1);
  show(x0);
  show(x1);
 }
}
```

Son exécution donne :

```
> java Point
Point@70329f3d=(1.0,2.0,3.0)
Point@59bca5f1=(0.0,0.0,0.0)
```

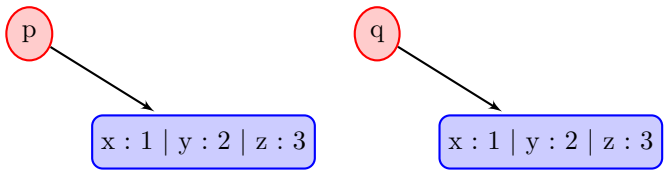
On voit bien qu'il y a là comme un passage par référence. En fait, c'est toujours la même chose, les appels de fonction se font toujours par appel par valeur, mais la valeur d'un « objet » de type `Point` est son adresse en mémoire.

3.3 Egalité physique et égalité structurelle

Reprenons la classe `Point` et $p = (1, 2, 3)$:

```
int xx = p.x;
int yy = p.y;
int zz = p.z;
Point q = new Point(xx, yy, zz);
```

Les objets `p` et `q` dénotent les mêmes points (mêmes coordonnées, i.e. les champs ont même valeur), mais ne sont pas égaux (en tant qu'objets, c'est-à-dire de pointeurs, ou de locations mémoire) :



Le test d'égalité physique `p == q` est donc faux, aussi bien en Java, C qu'en Caml, car `p` et `q` sont deux références distinctes.

Par contre, l'égalité structurelle est elle vraie, c'est-à-dire que `p` et `q` dénotent les mêmes points. On peut programmer cette égalité structurelle de la façon suivante, en Java :

```
static boolean equal(Point p, Point q) {
 return (p.x == q.x) && (p.y == q.y) && (p.z == q.z);
}
```

En C :

```
int equal(struct Point p, struct Point q) {
 return (p.x == q.x) && (p.y == q.y) && (p.z == q.z);
}
```

Et en Caml :

```
let equal p q = (p.x == q.x) && (p.y == q.y) && (p.z == q.z);;
```

En fait, ceci est déjà implémenté en Caml par `p=q`, il n'est pas besoin de le reprogrammer dans ce cas.

3.3.1 Partage

L'égalité structurelle est coûteuse : on peut parfois, par construction, l'obtenir par l'égalité physique. L'idée pour les `Point` est de remplacer le `new` par une fonction de création qui teste si un point de mêmes coordonnées existe déjà, et ne l'alloue pas de nouveau si c'est le cas. Pour ce faire, il faut un test d'égalité structurelle... On l'approxime par un moyen peu coûteux, le *hachage* et les tables de hachage.

Le principe du hachage est de permettre à moindre coût, de distinguer à coup sûr deux objets. Traditionnellement, cela est fait par une *fonction de hachage*, $h : \mathcal{O} \to N$ qui associe à tout objet que l'on veut considérer, un entier, de telle façon que $h(x) \neq h(y) \implies x \neq y$. Généralement, h est à valeur dans $0, \ldots, n$ (pour un certain $n \in N$), on peut ainsi ranger les objets de $\mathcal{O}$ dans une *table de collision* : tableau qui à $i \in 0, \ldots, n$ associe un *tableau de collision* : tous les x tels que $h(x) = i$. On en verra une implémentation un peu meilleure à la section 3.6, quand on aura introduit les types de données récursifs et les listes.

Par exemple, pour `Point` on peut utiliser une fonction de hachage modulaire simple :

```
class Point {
  ... }
```

```
class Hachage {
  final int n = ...; // nombre fixe
  final int m = ...; // taille fixee

  static int hache(Point p) {
    return (p.x+p.y+p.z)%n;
  }
}
```

On va maintenant gérer un ensemble de n points en se constituant une base de données de m (fixé) points distincts au maximum. Mais le remplissage de la base de données peut demander plusieurs fois d'inclure un point avec les mêmes coordonnées qu'un existant dans la base. On veut décider de l'égalité structurelle rapidement (par l'égalité physique)

On va définir un tableau 2D, dans lequel `tab[i][j]` est le `j`ième `Point` de code de hachage égal à `i`, comme suit :

```
class Tablenaive {
  static Point[][] tab;
  static int[] freeindex;
  public static void main(String[] args) {
    tab = new Point[n][m];
    freeindex = new int[n]; // tout a zero
    ...
  }
}
```

Et on peut maintenant écrire le code de création non-redondante d'un `Point` :

```
class Tablenaive {
  ...
  static void newPoint(int x, int y, int z) {
    int k = hache(x, y, z);
    Point p = new Point(x, y, z);
    boolean found = false;
    for (int i=0; i<freeindex[k]; i=i+1)
      if (equal(tab[k][i],p)) {
        i=freeindex[k];
        found=true;
      }
    if (!found) {
      tab[k][freeindex[k]]=p;
      freeindex[k]=freeindex[k]+1;
    }
  }
  ...
```

En voici un exemple, détaillé pas à pas. On rajoute d'abord le point (0,0,0) (hachage=0) :

```
final int n=3;
final int m=4;
...
newPoint(0, 0, 0);
```

(0,0,0)	null	null	null
null	null	null	null
null	null	null	null

Puis le point (1,2,4) (hachage=1) :

```
newPoint(1, 2, 4);
```

(0,0,0)	null	null	null
(1,2,4)	null	null	null
null	null	null	null

Puis le point (1,2,3) (hachage=0) :

```
newPoint(1, 2, 3);
```

(0,0,0)	(1,2,3)	null	null
(1,2,4)	null	null	null
null	null	null	null

Et le point (2,3,6) (hachage=2) :

```
newPoint(2, 3, 6);
```

(0,0,0)	(1,2,3)	null	null
(1,2,4)	null	null	null
(2,3,6)	null	null	null

Puis de nouveau le point (1,2,3) (hachage=0) :

```
newPoint(1, 2, 3);
```

(0,0,0)	(1,2,3)	null	null
(1,2,4)	null	null	null
(2,3,6)	null	null	null

Comme on le voit, les points positifs de cette construction sont qu'il n'y a pas de nouvelle création de point dans la table pour (1,2,3), et ceci en deux comparaisons avec un autre point (au lieu de potentiellement 4). Le point négatif est qu'il y a un coût mémoire incompressible avec le tableau 2D : $m \times n$ éléments alloués pour seulement m points possibles. Pour améliorer cela, il nous faudra utiliser des structures de données *dynamiques* (listes) décrites dans la section 3.6. La recherche dans la table des collisions est également peu optimisée. On pourrait au moins ordonner les points, pour une recherche *dichotomique*.

3.4 Tableaux et types produits

3.4.1 Définition et manipulation des tableaux 1D

Les tableaux sont un cas particulier de type produit, dont tous les champs sont de même type (cela serait plutôt un type « puissance »). C'est en fait un peu l'ancêtre des types produits, qui est utilisé par exemple pour coder les vecteurs et matrices. Il y a la possibilité également, par rapport au chapitre 2, de faire des tableaux 2D (matrices), 3D (tenseurs), etc.

On rappelle qu'en Java la déclaration d'un tableau 1D se fait de la façon suivante :

```
double[] x;
```

En C, un tableau est la même chose qu'un pointeur sur son premier élément :

```
double *x;
```

On rappelle ici que l'allocation (1D) se fait en Java comme suit :

```
x = new double[100];
```

Les indices du tableau ainsi alloué démarrent en 0 ; le dernier indice valide est 99.

En C, on peut allouer le tableau de deux manières, la plus générale :

```
x = (double *) malloc(100*sizeof(double));
```

Ce que fait le code plus haut est qu'il demande au gestionnaire mémoire de l'exécutable de trouver un emplacement mémoire ou mettre 100 variables de type `double` de manière consécutive, à une certaine adresse mémoire, retournée par la fonction `malloc`. Cette fonction retourne un `(void *)`, c'est-à-dire sur un type pointeur sur quelque chose sans type a priori (`void`), mais qui sera considéré par la suite comme du bon type, pointeur sur double (`double *`).

Plus simplement, à la déclaration, quand on connaît la taille du tableau à allouer de maniere statique, on peut écrire :

```
double x[100];
```

Les indices du tableau démarrent également en 0 et le dernier indice valide est aussi 99.

En Caml :

```
let x = Array.make 100 0;;
```

Ici, tous les éléments sont initialisés à zéro. Les projections, et constructions pour les tableaux 1D, vus comme types produit sont : en Java et C :

```
double y = x[50];
x[50] = 1.0;
```

Et en Caml :

```
let y=x.(50);;
x.(50)<-1.0;;
```

3.4.2 Exemple de code en C, Java et OCaml, utilisant des tableaux 1D

Donnons un exemple de code utilisant les tableaux 1D, un simple calcul de produit scalaire dans les trois langages, C, Java et OCaml, pour en illustrer les différences :

En Java :

```
class vectn {
  double[] comp;

  vectn(int n) {
    comp = new double[n];
  }
}

class Essai {
  static double scproduct(vectn p, vectn q) {
    double res = 0;
    for (int i=0; i<p.length; i++)
      res = res+p.comp[i]*q.comp[i]
    return res;
  }
  ...
}
```

Et en C :

```
double u[100], v[100]; ...

  double scproduct(double p[100], double q[100]) {
    double res = 0;
    for (int i=0; i<100; i++)
      res = res+p[i]*q[i]
    return res;
  }

...s = scproduct(u, v);
```

Ou (taille de tableau passé en paramètre) :

```
double *u, *v; int d; ...
u = (double *) malloc(d*sizeof(double));
v = (double *) malloc(d*sizeof(double)); ...

  double scproduct(double *p, double *q, int dim) {
    double res = 0;
    for (int i=0; i<dim; i++)
      res = res+p[i]*q[i]
    return res;
  }
```

```
...s = scproduct(u, v, d);
```

Et finalement, en Caml :

```
let scproduct u v n = let res=ref 0.
                      and n=Array.length u in
  for i=0 to n do
    res := !res +. u.(i) *. v.(i);
  done;
  !res;;
```

Donne une fonction :

```
val scproduct : float array -> float array -> float = <fun>
```

3.4.3 Tableaux de dimension supérieure

Illustrons enfin la construction et la manipulation de tableaux 2D. On considère une matrice de $\mathbb{R}^{N\times M}$, avec $N = 3$ colonnes de $M = 4$ lignes :

En Java, on écrira :

```
double M = new double[4][3];
M[0][0] = 8;
M[0][1] = 1;
...
```

8	1	6
3	5	7
4	9	2
10	12	11

Ou de façon plus compacte :

```
double[][] M={{8,1,6},
              {3,5,7},
              {4,9,2},
              {10,12,11}};
```

8	1	6
3	5	7
4	9	2
10	12	11

On peut aussi être plus explicite sur le choix de l'organisation en mémoire. Ici on fait le choix d'implémenter la matrice dans un bloc contigu en mémoire en remarquant qu'une matrice $N * M$ est donnée par un ensemble de $N \times M$ réels.

On procède alors à l'allocation en faisant, ici en C (c'est simple à écrire en Java également) :

```
float *x = (float *) malloc(N*M*sizeof(float));
```

On récupère $x_{i,j}$ par `x[i*M+j]`.

8	1	6	3	5	7	4	9	2	10	12	11

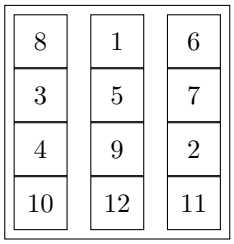
On peut aussi choisir d'organiser la matrice en vecteur de vecteurs en remarquant qu'une matrice $N * M$ est un élément de $\mathbb{R}^{N \times M}$, c'est aussi un vecteur à N composantes chacune dans $\mathbb{R}^M$. L'allocation se fait alors par (en C également, ce que l'on réalise sans peine en Java, de même mais avec `new` à la place de `malloc`) :

```
float **x = (float **) malloc(N*sizeof(float *));
```

Puis :

```
for (i=0; i<N; i++)
  x[i] = (float *) malloc(M*sizeof(float));
```

Le tableau est ainsi implémenté comme `N` pointeurs sur `N` vecteurs de taille `M` (chacun un bloc contigu de mémoire).



Tout ceci se généralise en dimension supérieure bien sûr. En Caml, on peut construire plus simplement les tableaux 2D :

```
let x = Array.make_matrix n m 0;;
x.(1).(2) <- -9.0;;
```

3.5 Types somme

On va maintenant définir un type « dual » du type produit. Prenons l'exemple d'un cas « primitif ». Supposons que l'on veuille représenter un nombre fini de valeurs, par exemple, les couleurs d'une carte à jouer (ici en Caml) :

```
> type couleur = PIQUE | COEUR | CARREAU | TREFLE;;
> let x = PIQUE;;
val x : couleur = PIQUE
```

Ceci est un cas simple de type somme/disjonctif. En Java :

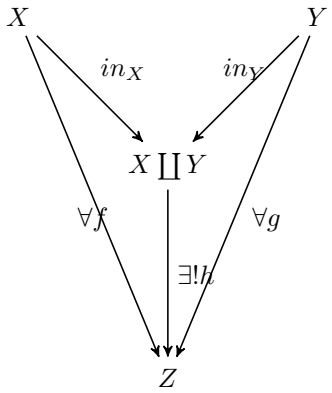
```
enum couleur { PIQUE, COEUR, CARREAU, TREFLE };
couleur c = couleur.PIQUE;
System.out.println(c);
```

(affiche `PIQUE`)

Et en C :

```
enum couleur { PIQUE, COEUR, CARREAU, TREFLE };
enum couleur c = PIQUE;
```

En fait, les types somme, ou disjonctifs, sont le dual des types produits. Une somme, ou une disjonction, une union sont une forme de coproduit (au sens de la théorie des catégories). Elle vérifie la propriété universelle suivante, qui est la même que celle sur les produits, mis à part le fait que toutes les flèches sont dans l'ordre inverse :



Propriété universelle : $\forall f, g, \exists! h$, tel que

$$\begin{aligned} h \circ in_X &= f \\ h \circ in_Y &= g \end{aligned}$$

Dans les ensembles
- $X \coprod Y = \{(0,x) \mid x \in X\} \cup \{(1,y) \mid y \in Y\}$ (union disjointe)
- $in_X(x) = (0,x)$, $in_Y(y) = (1,y)$
- $h(z) = \begin{cases} f(x) & \text{if } z = (0,x) \\ g(y) & \text{if } z = (1,y) \end{cases}$

En fait, en Java (et en Caml) `couleur.PIQUE` (resp. `PIQUE`) est l'injection canonique de l'ensemble à un élément (`PIQUE`) vers le type `enum couleur` (resp. `couleur`). En C, ce n'est qu'une façon de nommer symboliquement des constantes scalaires :

```
enum couleur { PIQUE=0, COEUR=1, CARREAU=2, TREFLE=3 };
enum couleur c = PIQUE; // vaut 0 (int)
```

Passons maintenant au jeu de cartes. En Caml :

```
> type carte = As of couleur
             | Roi of couleur
             | Dame of couleur
             | Valet of couleur
             | Petitecarte of int*couleur;;
> let x = As(PIQUE);;
val x : carte = As PIQUE
> let y = Petitecarte(8, TREFLE);;
val y : carte = Petitecarte (8, TREFLE)
```

`As` représente l'injection canonique d'une copie de `couleur` (« taggé » par `As`) vers le type `carte`. Le *pattern-matching* permet de programmer après, sur ce type, de façon élégante.

On pourrait aussi écrire en Caml quelque chose de théoriquement isomorphe (mais que le compilateur ne verra pas comme isomorphe) :

```
> type valeur = As | Roi | Dame | Valet | Petitecarte of int;;
type valeur = As | Roi | Dame | Valet | Petitecarte of int
> type cartebis = valeur*couleur;;
type cartebis = valeur * couleur
> let z = (As, PIQUE);;
val z : valeur * couleur = (As, PIQUE)
```

(`valeur`×`couleur`∼`carte`∼`cartebis`...)

En Java, le type somme simple pour les valeurs « figures » (roi, dame, valet, as) que l'on a défini en Caml, est implémenté par un type *énuméré*, par contre, on utilise un codage luxueux de la somme comme un produit cartésien à cause des valeurs « numériques » :

```
class Carte {
    enum valeur { As, Roi, Dame, Valet, PetiteCarte };
    enum couleur { PIQUE, COEUR, CARREAU, TREFLE };
    valeur val; int valpetite;
    couleur c;
}
```

On reconnaît bien les enregistrements pour le produit `valeur`×`couleur`, mais `valpetite` n'est utile que quand `val` vaut `PetiteCarte`.

On a le même problème en C :

```
enum couleur { Pique, Coeur, Carreau, Trefle };
enum valeur { As, Roi, Dame, Valet, PetiteCarte };

struct carte {
  enum valeur val; int valpetite;
  enum couleur c;
};
```

Encore qu'en C, il y a un type somme primitif, les unions :

```
enum couleur { Pique, Coeur, Carreau, Trefle };
union valbis {
  enum valeurbis { As, Roi, Dame, Valet } value;
  int valpetitebis;
};

struct carte {
  union valbis val;
  enum couleur c;
};
```

Le problème, qui en est l'avantage également en fait est que les espaces mémoires alloués aux différents champs d'une union sont les mêmes : `x.val.value` et `x.val.valpetitebis` sont les mêmes mots mémoire. C'est une cause d'erreurs fréquentes, ou une source d'inspiration pour des codage abscons.

Comme on l'a déjà signalé, la façon élégante de programmer sur les différents cas d'un type somme se fait en Caml par pattern-matching. Non seulement c'est plus élégant, mais le compilateur fait des vérifications très utiles, comme celle que l'on n'a pas oublié de cas :

```
let f x = function
  | As _ -> ...
  | Roi _ -> ...
  ...
  | Petitecarte (x, c) -> ...
```

```
  | _ -> ...;;
```

Vous reconnaitrez exactement la fonction h de la propriété universelle définissant le type produit.

En Java on a une forme « primitive » du sélecteur :

```
static f(Carte x) {
  switch (x.val) {
    case As : ...; break;
    case Roi : ...; break;
    ...
    default : ...
  }
  ...
}
```

de même en C.

3.6 Types de données dynamiques

Un cas classique où l'on souhaite disposer d'un type de données « dynamique », c'est-à-dire non entièrement défini de façon statique, est celui où l'on souhaite manipuler des matrices dont la taille est inconnue au moment de l'écriture du code, et est paramétrable au moment de l'exécution.

On veut même souvent des types dont la taille évolue au cours du temps : les tableaux ne sont alors plus pertinents. Reprenons l'exemple des tables de hachage, implémentées précédemment par un tableau 2D, où `tab[i][j]` est le `j`ième `Point` de code de hachage égal à `i` :

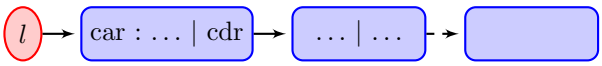
```
class Tablenaive {
  static Point[][] tab;
  static int[] freeindex;

  public static void main(String[] args) {
    tab = new Point[n][m];
    freeindex = new Point[n]; // tout a zero
  }
}
```

3.6.1 Listes

On va définir une liste l (de `int` par exemple) qui est une structure de données telle que :

- son premier élément (*car* de LISP) est un `int` ;
- son deuxième élément (*cdr* de LISP) est une liste d'entiers.

C'est en quelque sorte un type produit, avec un nombre de produits non déterminé à l'avance. L'écriture de ce type est sous la forme d'un type produit, récursif.



On a en effet envie d'écrire, une sorte d'« équation aux domaines » :

$$List_{\mathbb{N}} = \{()\} \cup \mathbb{N} \times List_{\mathbb{N}}$$

pour ne pas oublier la liste vide. Il s'agit donc d'un mélange a priori de définition de type somme et type produit, récursif. Son implémentation en Java est la suivante :

```
class List {
  int hd;
  List tl;
}
```

En C :

```
typedef struct stList {
  int hd;
  struct stList *tl;
} *List;
```

On remarque tout de suite qu'on n'a pas de codage particulier pour la somme. En fait, une liste non vide est allouée quelque part en mémoire (pour conserver la valeur de `hd` et de `tl`), donc est un *pointeur* valide, différent de `null`. On identifie la liste vide avec `null`, en Java et C.

En Caml, on n'a pas de pointeur `null`, donc on écrit explicitement le type somme, pour implémenter les listes :

```
type liste = Nil | Cons of int*liste
```

Dans ce code, `Nil` est l'injection canonique de `unit` vers `liste`, et `Cons` est l'autre injection canonique de `int*liste` vers `liste`.

On peut alors écrire directement :

```
> Cons(1,Nil);;
- : liste = Cons (1, Nil)
```

En Java, on aura un constructeur par défaut, lié à ce type, utilisé par exemple dans le code suivant :

```
List l;
l = new List();
l.hd = 4;
l.tl = new List();
l.tl.hd = 5;
l.tl.tl = null;
```

La construction de la liste se fera par étapes : tout d'abord,

```
List l;
```

Puis,

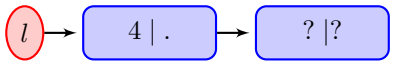
```
l = new List();
```

Puis encore :

```
l.hd = 4;
```

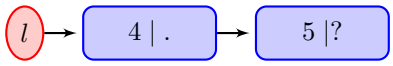
Puis ensuite :

```
l.tl = new List();
```



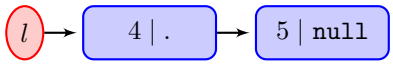
Puis encore :

```
l.tl.hd = 5;
```



Puis enfin :

```
l.tl.tl = null;
```



Il s'agit maintenant de définir l'injection canonique de $\mathbb{N} \times \mathit{List}$ vers List :

```
class List {
  ...
  List(int car, List cdr) {
    hd = car;
    tl = cdr;
}
```

Du coup, on aurait pu écrire (exemple précédent) `List l = new List(4, new List(5, null));`

En C, on aurait pu écrire une fonction assez similaire :

```
List cons(int car, List cdr) {
  List res = (List) malloc(sizeof(struct stList));
  res->hd = car; res->tl = cdr; return res;
}
```

Remarquez le `res->hd`, abbréviation de `(*res).hd`.

3.6.2 Les listes linéaires

On appelle listes linéaires, les listes construites uniquement sans *cycle*. Dans ce cas, les listes ont une notion de longueur : il suffit de définir *length* dans l'un des deux cas liste vide, liste non vide (donc égale à un $cons(hd, l)$) d'après l'équation de domaines $List_{\mathbb{N}} = \{()\} \cup \mathbb{N} \times List_{\mathbb{N}}$:

- $length(()) = 0$,
- $length(cons(hd, l)) = length(l) + 1$

Cela nous interdit bien sûr des listes *circulaires* comme $l = cons(0, l)$, qui « représente » en quelque sorte la liste infinie constituée uniquement de zéros.

En fait en Caml, tout cela est déjà défini, les listes d'objets de type `'a` sont de type `(type 'a list)`. Voici quelques fonctions de manipulation classiques :

```
> [];;
- : 'a list = []
> [ 1 ; 2 ; 3 ];;
> 1 :: 2 :: 3 :: [];;
- : int list = [1; 2; 3]
> [ 1 ; 2 ] @ [ 3 ; 4 ; 5];;
- : int list = [1; 2; 3; 4; 5]
> List.length [ "hello" ; "world" ; "!" ];;
- : int = 3
> List.hd [ 1 ; 2 ; 3 ];;
- : int = 1
> List.tl [ 1 ; 2 ; 3 ];;
- : int list = [2; 3]
```

3.6.3 Application aux tables de hachage

À chaque i entre 0 et n, on va représenter une liste de collisions possibles (au lieu d'un tableau bidimensionnel, figé) :

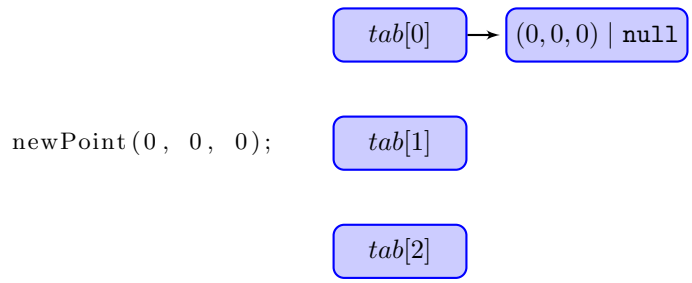
```
class Pointlist {
  Point p;
  Pointlist tl;
  Pointlist(Point q, Pointlist r) {
    p = q; tl = r;
  }
}

class Table {
  static Pointlist[] tab;
  public static void main(String[] args) {
    tab = new Pointlist[n];
    ...
```

```
  }
}
```

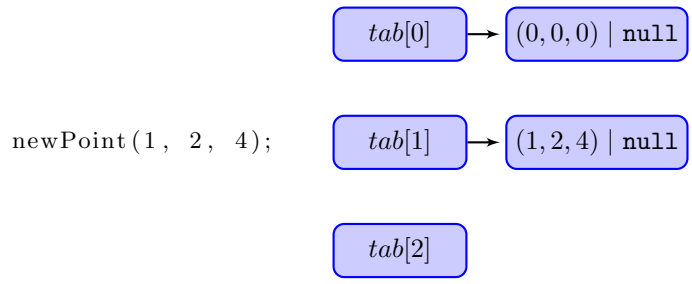
La création d'un éventuel nouveau `Point` se fait comme suit :

```
static Pointlist addPoint(Pointlist l, Point q) {
  if (l == null)
    return new Pointlist(q, null);
  if (equal(l.p, q))
    return l;
  else
    return new Pointlist(l.p, addPoint(l.tl, q));
}

static void newPoint(int x, int y, int z) {
  Point p = new Point(x, y, z);
  int k = hache(p);
  tab[k] = addPoint(tab[k],p);
}
```

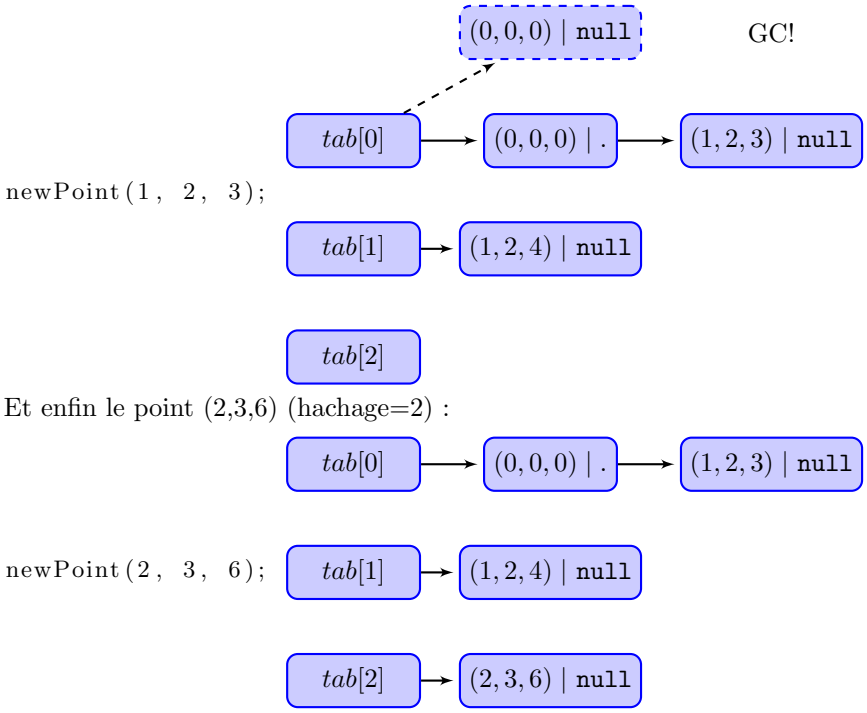
On peut alors dérouler les ajouts suivants. On commence par ajouter le point (0,0,0) (hachage=0) :



Puis le point (1,2,4) (hachage=1) :



Puis encore le point (1,2,3) (hachage=0) :



En fait tout cela existe déjà en Java. On a les classes `List`, `AbstractList`, `Vector`, `HashTable` qui sont déjà définis. De même, en Caml : on a `Hashtbl` etc.

3.6.4 Listes et partage

Les listes en pratique, font du partage, pour des questions d'efficacité (voire du partage maximal quand cela est possible, ou « hash-consing »), et pour représenter des structures de données « infinies ».

Commençons par l'idée de listes infinies (« rationnelles »). Par exemple, on veut représenter toutes les listes infinies, ultimement périodiques, par exemple une liste

$$l = (0, (1, (2, (3, (2, (3, ...$$

(que des *pattern* 2 puis 3 répétés). On pourra écrire par exemple :

```
List l3 = new List();
l3.hd = 2;
l3.tl = new List(3, l3);
List l2 = new List(1, l3);
List l1 = new List(0, l2);
```

Remarque : c'est la construction correcte de `l3` que l'on écrit par « abus de notation » $l3 = cons(2, cons(3, l3))$. Il y a des langages (comme Haskell, voir chapitre 10) qui permettent de manipuler algorithmiquement les listes infinies, par évaluation paresseuse, que l'on peut également simuler en Caml, C ou Java.

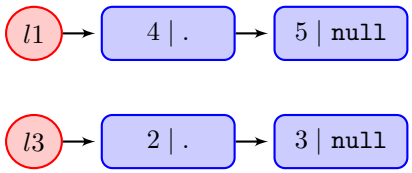
Si on fait du partage de parties de listes, sans *cycle*, on économise juste en mémoire, et cela peut permettre de faire du partage efficace sur les listes. Par exemple, pour la fonction append : on veut concaténer une liste `y` au bout de la liste `x` :

```
static List append(List x, List y) {
  if (x == null) return y;
  List p = x;
  while (p.tl != null) p = p.tl;
  p.tl = y;
  return x;
}
```

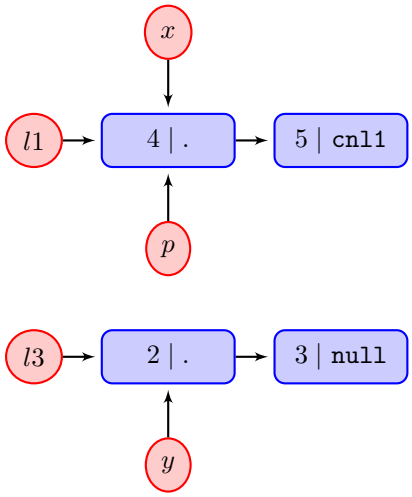
Considérons l'appel :

```
append(l1,l3);
```

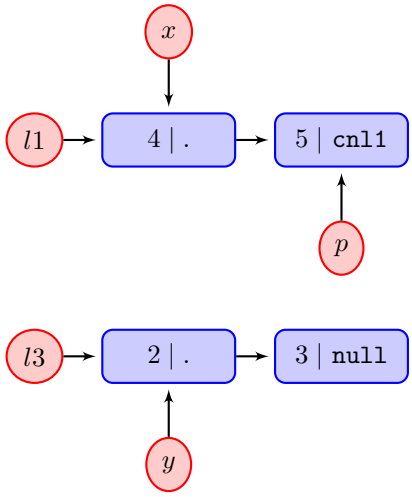
Avec :



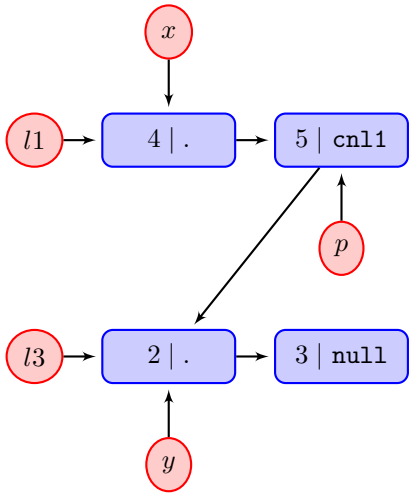
Son exécution procède ainsi :



Puis,



Et encore :



On aurait pu encore écrire cette version de `append` dans laquelle on n'a pas de partage, pour l'argument gauche :

```
static List append(List x, List y) {
  if (x == null) return y;
  else {
    List p = x;
    List q = new List(x.hd, null);
    List r = q;
    while (p.tl != null) {
      q.tl = new List(p.tl.hd,null);
      q = q.tl;
```

```
      p = p.tl;
    }
    q.tl = y;
    return r;
}
```

On a en fait toutes les possibilités d'implémentation : partage possible de l'argument gauche, de l'argument droit, des deux, ou d'aucun des deux. L'intérêt ou l'inconvénient des versions sans partage est que si l'on modifie *en place* les listes `l1`, `l2` ou `l3`, `append(l1,l3)` et `append(l2,l3)` ne sont pas modifiées. On en verra des versions, récursives, au chapitre 5.

3.7 Le ramasse-miette, ou GC

On a vu comment allouer, par `new`, des nouvelles cases mémoires, contenant des données structurées. En Java, comme en OCaml, on n'a pas à se soucier de la libération de la mémoire non utilisée. Il y a un mécanisme de *garbage collector* ou glaneur de cellules, ou encore ramasse-miette, disponible pendant l'exécution de tout programme.

Ce concept a été inventé par John MacCarthy pour le LISP (prix Turing 1971). Il permet de récupérer la mémoire non utilisée, c'est un processus qui s'exécute en parallèle et qui, automatiquement :

- détermine quels objets ne peuvent plus être utilisés par un programme ;
- récupère cet espace mémoire (pour être utilisé lors d'allocations futures).

De nombreux algorithmes ont été étudiés, et implémentés par exemple dans Java, Caml, mais pas dans C.

Les principes de fonctionnement sont les suivants. Pour des algorithmes de type « Mark and Sweep », le GC commence à parcourir les locations mémoires *vivantes* (accessibles depuis les *racines*, i.e. les noms de variables du programme Java). Pendant ce temps, l'exécution du programme Java est suspendue. Il y a alors 2 phases :

- (*mark*) : Les objets alloués et visitables par le GC depuis les racines sont *taggués* : visité et pas visité ;
- (*sweep*) : Le GC parcourt adresse par adresse le *tas* (l'endroit en mémoire où sont alloués les objets) et « efface » les objets non taggués « visité ».

Un autre type d'algorithme couramment utilisé (éventuellement de façon ad hoc par des programmeurs C qui doivent implémenter un tel mécanisme dans leur programme) est le comptage de références. Le GC maintient avec chaque objet, un nombre de références pointant sur chaque objet. Si ce compteur arrive à zéro, l'objet est libéré. D'autres types d'algorithmes existent, que nous ne décrirons pas : « stop and copy », les GC conservatifs, incrémentaux, générationnels (cas de Java) etc.

Comment fait-on alors dans d'autres langages, comme le C, qui n'ont pas de GC ? Il faut procéder à une allocation et désallocation manuelles. Voici un exemple sur les listes :

```
List cons(int car, List cdr) {
```

```
  /* allocation */
  List res = (List) malloc(sizeof(struct stList));
  res->hd = car; res->tl = cdr; return res;
}

void freelist(List l) {
  if (l == null) return;
  freelist(l->tl);
  /* deallocation */
  free(l);
}
```

Le programmeur a ainsi du écrire le code d'une fonction `freelist`, qui va libérer, une par une, les cellules mémoire d'une liste, par l'instruction `free`. Il devra réfléchir précisément à quand appeler cette fonction dans son code, quoi partager en mémoire etc. C'est la cause de nombreuses erreurs, car si on n'appelle pas assez les fonctions de libération mémoire, on court le risque de ne pas avoir assez de mémoire pour exécuter son programme (« fuite mémoire »), et si au contraire on libère trop, on va manipuler des adresses mémoires invalides.

Chapitre 4

Programmation orientée objet, en JAVA

Java est dans la lignée de langages de programmation « orientés objets » nombreux, par exemple Simula 67 (basé sur Algol 60), Smalltalk 71/80, Objective C, C++ 83... L'utilité principale de l'approche orientée objet vient essentiellement de son *style* de programmation, qui permet de bien cloisonner le code, en unités cohérentes, cf. diagrammes de classes et UML (*Unified Modelling Language*). Cela permet aussi de réutiliser du code, plus aisément (« composants »), par héritage en particulier (voir section 4.3).

4.1 Statique versus dynamique

Jusqu'à présent, on n'avait pas pu expliquer le mot clé `class`, que l'on avait utilisé dans deux contextes apparemment très différents. On avait défini des `class` contenant des données (avec constructeurs néanmoins), où les champs n'étaient pas `static`, pour les types produits, au chapitre 3. Ou alors, comme au chapitre 2, on avait défini des `class` ne contenant que du code, et des fonctions déclarées avec le mot clé `static` : les « programmes ». En fait, on peut mêler les deux, et utiliser de façon générale des fonctions non `static`, ou « dynamiques ».

Une philosophie générale de la programmation orientée objet peut être décrite, en première approximation par l'exemple suivant qui implémente une pile, construite à partir d'une liste d'entiers :

```
class Pile {
  List c;
  Pile(List x) { this.c = x; }
}

class Prog {
  static void push(int a, Pile l) {
    l.c = new List(a,l.c);
```

```
  }

  static void pop(Pile l) {
    l.c = l.c.tl;
  }

  static int top(Pile l) {
    return l.c.hd;
  }
}
```

D'une certaine façon, on voudrait mettre toutes les méthodes concernant les piles dans `class Pile`, pour en faire un « module » cohérent et réutilisable. On obtiendrait ainsi :

```
class Pile {
  List c;
  Pile(List x) { this.c = x; }

  static void push(int a, Pile l) {
    l.c = new List(a,l.c);
  }

  static void pop(Pile l) {
    l.c = l.c.tl;
  }

  static int top(Pile l) {
    return l.c.hd;
  }
}

class Prog {
  ... Pile.push(1,p); ...
}
```

Les fonctions s'appliquant à la *collection* ou *classe* des piles sont comme des *champs* fonctionnels d'un type enregistrement, on les appelle des *méthodes*. On lance leur exécution en faisant `Pile.`*méthode*.

Malgré tout, il reste une différence entre ces champs fonctionnels et le champ de données `List c`. Quand on a une `Pile p`, on fait `p.c` pour obtenir son champ de type `List`, pourquoi fait-on ici `Pile.push(1,p)` pour les méthodes ? `c` est un champ non statique (pas de qualificatif `static`) alors que `push` est une méthode statique (qualificatif `static`). Commençons par expliquer les champs statiques/non-statiques avant les méthodes. Expérimentons le code suivant :

```
class Stat {
  static int x = 2;
}

class Prog {
```

```
 public static void main(String[] args) {
  Stat s, t;
  s = new Stat();
  t = new Stat();
  System.out.println(s.x);
  t.x = 3;
  System.out.println(s.x);
 }
}
```

Cela donne :

```
> java Prog
2
3
```

Cela n'a pas l'air très logique... Alors qu'en changeant juste le programme de la façon suivante :

```
class Stat2 {
  int x = 2;
}
```

On obtient bien ce que l'on souhaite :

```
> java Prog
2
2
```

En fait, une `class` (classe) définit un ensemble d'*objets*. `Stat` (version 1, statique) ne contient qu'un singleton alors que `Stat2` (version 2, dynamique) peut contenir n'importe quel nombre d'objets, dont le trait commun est qu'ils contiennent un champ entier `x` initialisé à 2. Plus généralement, les champs `static` sont communs à tous les *objets* de cette classe.

Revenons aux méthodes. D'une certaine manière, l'appel `Pile.push(1,p)` est lourd pour pas grand chose. On voudrait écrire comme pour les champs de données quelque chose comme `p.push(1)`. C'est possible avec une méthode `push` non statique. On obtient alors le code suivant :

```
class Pile {
  List c;
  Pile(List x) { this.c = x; }
...

  void push(int a) {
    this.c = new List(a,this.c);
  }

  void pop() {
    this.c = this.c.tl;
  }
```

```
  int top() {
    return this.c.hd;
  }
}
```

On obtient ainsi un « module » cohérent parlant de piles, avec toutes les opérations « légales » associées.

De même que pour les champs de données statiques, les méthodes `static`, sont toutes communes à leur classe (on parle alors de champ ou de méthode de classe). C'est toujours le cas de `main` par exemple. Quand on programme avec la version dynamique, `p.push` est une fonction différente de `q.push` (pour deux piles `p` et `q` distinctes). La méthode `push` (dynamique) définit une fonction partielle, instanciée par le passage de `p` (appelé `this` – l'objet courant sur lequel s'applique la méthode) lors de son appel par `p.push(...)` (règles de passage d'argument inchangées). Quand on fait `p.push(...)`, la machine regarde le type de `p`, voit `Pile`, trouve l'« enregistrement » (fonctionnel) `push`, passe la référence sur le contenu de `p` à `push` puis exécute son code.

Vous pouvez néanmoins avoir de bonnes raisons pour écrire des méthodes statiques, par exemple, les fonctions de librairie Java `Math.sin`, `Math.cos` etc.

Remarques : `this`... est le plus souvent implicite. On aurait pu écrire :

```
  void push(int a) {
    c = new List(a,l.c);
  }

  void pop() {
    c = l.c.tl;
  }
```

Autre remarque importante : le cas de `null`. Il ne faut jamais faire `p.push(...)` quand `p` vaut `null`, car c'est une valeur indéfinie, qui n'a même pas de type ni donc de pointeurs vers les champs ou méthodes qu'on aimerait y associer.

4.2 Types somme, revisités

Tout cela rend possible une autre implémentation des types somme (voir le chapitre 3).

Considérons le problème de représenter des expressions arithmétiques. On construit un arbre syntaxique. Les expressions qui nous intéressent sont de la forme :

$$expr = Var \mid Cste \mid expr + expr \mid expr * expr \mid -expr$$

Et on écrit une classe Java les implémentant, comme suit :

```
enum Typop {
  plus, minus, times
}

class Expr {
```

```
  int select;
  int Cste;
  String Var;
  Typop Op;
  Expr gauche;
  Expr droite; ...
}
```

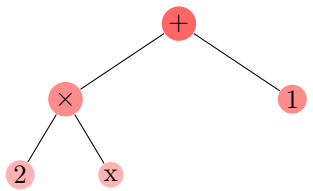
On inclut ici tout dans le type produit. On utilise `select` : si 0, l'expression est une constante (champ `Cste`), si 1, l'expression est une variable (champ `Var`), si 2, l'expression est un opérateur binaire (`Op` est `plus` ou `times`) et les sous-expressions sont `gauche` et `droite`, ou l'expression est unaire (`Op` est `minus`) et `gauche` est la sous-expression.

Néanmoins, on peut obtenir un style de programmation mieux structuré avec les constructeurs. Il est commode en effet de définir plusieurs constructeurs selon les cas (qui « imitent » les injections dans le type somme) :

```
Expr(int constante) {
  select = 0; Cste = constante;
}

Expr(String variable) {
  select = 1; Var = variable;
}

Expr(Typop operateur, Expr argl, Expr argr) {
  select = 2; Op = operateur; gauche = argl; droite = argr;
}

Expr(Expr arg) {
  select = 2; Op = minus; gauche = arg;
}
```

Par exemple, une expression $2 * x + 1$ est représentée comme :

```
Expr e1 = new Expr(1);
Expr e2 = new Expr(2);
Expr e3 = new Expr("x");
Expr e4 = new Expr(times, e2, e3);
Expr e5 = new Expr(plus, e4, e1);
```



Terminons cette section par quelques premiers éléments de vocabulaire communs aux langages *orientés objet* :

- *objet* : structure de donnée composée de :
  - *attributs* : ou champs (cf. types produit, ou enregistrement !), ce sont les données élémentaires composant l'objet ;
  - *méthodes* : fonctions pouvant s'appliquer à ce type d'objet ;
- *classe* : ensemble d'objets du même type ;
- *instance* : on dit qu'un objet est une instance de sa classe.

Un *objet* instance d'une *classe*, par exemple :

```
class Toto {
  int champ1; int champ2;
  int methode1(int parametre) { ...
  }
}

Toto x = new Toto(); ...
```

contient des *champs* de données et des *méthodes* qui sont des fonctions *s'appliquant* sur tout objet de la classe : `x.methode1(parametre)` (en C, non orienté objet, on écrirait `methode1(x,parametre)`).

Les méthodes sont des fonctions associées à une classe d'objets. Elles peuvent avoir plusieurs qualificatifs : méthodes `static`, ou méthodes de classe, on ne peut faire que `methode1(parametre)` ou `Toto.methode1(parametre)`. La méthode appelée n'a alors pas de connaissance d'un objet `x` particulier, mais juste de la classe `Toto`. Les méthodes `public` sont connues de tout le monde. Il existe également les qualificatifs `private` et `protected` qui permettent de restreindre la visibilité des méthodes ou des champs. Par défaut, sans qualificatif, les champs ou méthodes seront visibles de toutes les classes du même paquetage. Il existe certaines méthodes particulières dites *constructeurs* (déjà vues au chapitre 3).

4.3 Héritage

On va voir dans cette section que la méthodologie objet va bien plus loin que cela. L'intérêt principal de l'organisation en classes d'objets est de définir et d'utiliser des relations entre ces classes. Il est courant que l'on ait besoin de structures de données informatiques assez générales (comme un point dans le plan, dans l'exemple qui suit) et d'autres, un peu raffinées (comme les points dans le plan, avec une couleur associée, dans ce qui suit). Le deuxième est une *instance* du premier en ce sens que tout point coloré est en particulier un point. Cette remarque n'est pas du pure esthétique : ayant programmé des fonctions agissant sur des points (comme une translation par un vecteur, par la suite), on remarque qu'elles devraient aussi s'appliquer naturellement aux points colorés, sans avoir à les reprogrammer, source de confusion et d'erreurs. Le mécanisme d'héritage (de code) est fait pour cela. C'est assez similaire à l'utilisation de théorèmes en mathématiques sur des structures algébriques que l'on peut voir de diverses manières : un espace vectoriel est en particulier un groupe abélien, et on peut utiliser n'importe quel théorème applicable aux groupes pour en déduire quelque chose sur les espaces vectoriels.

Reprenons l'exemple de la classe `Point`. on l'a définie ainsi que ses méthodes, au chapitre 3 :

```
class Point {
  int x, y; // coordonnees
  translation(int u, int v) {
    x = x+u; y = y+v; } ... }
```

On veut maintenant des points colorés ; au lieu de redéfinir les méthodes, dont `translation`, qui n'ont pas besoin de la couleur, et qui s'appliquent en quelque sort au `Point` sous-jacent :

```
class ColorPoint {
 int x, y; // coordonnees
 int col; // couleur

 translation(int u, int v) {
  x = x+u; y = y+v;
 }

 ...
}
```

On peut écrire :

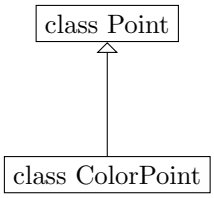
```
class ColorPoint
       extends Point {
  int col; // couleur
  ... }
```

Par le mot clé `extends` : `class A extends B ...` , on dit que `A` *hérite* de `B`. Cela veut dire qu'un `ColorPoint cp` aura des champs `col` (accessible par `cp.col`), mais aussi `x` et `y` (accessibles par `cp.x` et `cp.y`). On aura aussi le fait que `translation` s'applique implicitement sur un objet de la classe `ColorPoint` en s'appliquant à sa « sous-partie » `Point`.

Cela s'appelle l'*héritage*, et permet une économie et une structuration du code meilleure (en Java et Caml, n'existe pas en C « pur »).

Remarque : tous les objets Java héritent d'une classe unique : `Object`.

La structuration par héritage se décrit généralement par des diagrammes de classes, qui sont des graphes décrivant les attributs des classes, et leur relation d'héritage, comme dans l'exemple ci-dessous pour les classes `Point` et `ColorPoint` :



On pourrait imaginer d'hériter de plusieurs classes en même temps, pour pouvoir utiliser des champs et des méthodes de diverses classes : cela s'appelle l'héritage multiple, et est autorisé en C++ mais pas en Java. L'héritage multiple *de code* est compliqué sémantiquement, et le choix a été fait en Java de ne pas l'autoriser, mais d'autoriser l'héritage simple de code, et multiple de signatures (voir section 4.5).

Terminons cette section par un résumé du vocabulaire utile :

- *héritage* : une classe peut être une *sous-classe* d'une autre. La sous-classe l'*étend* en rajoutant des méthodes et des attributs, elle accède aux attributs et aux méthodes de sa sur-classe ;
- *polymorphisme (par sous-typage)* : un objet d'une sous-classe `A` de `B` en accédant aux méthodes de `B` est considéré du type `B` (et `A`).

En fait, on a aussi une forme de polymorphisme pour les méthodes :

Prenons l'exemple de deux implémentations possible de calcul garanti (c'est-à-dire qui permet de « représenter » fidèlement le calcul dans les réels, en utilisant des nombres machine, en précision finie), l'une par arithmétique rationnelle :

```
class Rat {
 int p, q;
 Rat(int x, int y) { p = x; q = y; }

 Rat plus(Rat y) {
   return new Rat(this.p*y.q+this.q*y.p,this.q*y.q);
 }

 void show() {
   System.out.println(p+"/"+q); }
}
```

L'autre par arithmétique d'intervalles :

```
class Doubleint {
  double inf, sup;
  Doubleint(double x, double y) { inf=x; sup=y; };

  Doubleint plus(Doubleint y) {
    return new Doubleint(this.inf+y.inf,
                         this.sup+y.sup);
  }

  void show() {
    System.out.println("["+inf+","+sup+"]"); }
}
```

Les méthodes `plus` et `show` sont *polymorphes* : elles peuvent prendre des `Rat` ou des `Doubleint`. C'est souvent très pratique, cela permet d'utiliser le même programme avec des données de type différent. En voici un exemple d'exécution :

```
class Prog {
 public static void main(String[] args) {
   Rat r = new Rat(1,2);
   Rat s = new Rat(1,3);
   Rat t = r.plus(s); // = s.plus(r);
   Doubleint ri = new Doubleint(0.50,0.50);
   Doubleint si = new Doubleint(0.33,0.34);
   Doubleint ti = ri.plus(si); // = si.plus(ri);
   t.show();
   ti.show();
```

```
 }
}
```

Cela donne :

```
5/6
[0.8300000000000001,0.8400000000000001]
```

4.4 Exceptions

Voici un autre trait « moderne » de langages de programmation comme Java (et qui n'existe pas en C par exemple) : les exceptions, pour traiter des cas d'erreur.

Reprenons le code de `pop()` dans la classe `Pile` :

```
void pop() {
  this.c = this.c.tl;
}
```

Comment traiter le cas `this.c == null` ? :

```
void pop() {
  if (this.c != null)
   this.c = this.c.tl;
}
```

Le problème est que laisser `this.c` à `null` est trompeur. On pourrait aussi changer le type de la méthode `pop()`. Cette méthode pourrait ainsi renvoyer un code d'erreur : `int pop()`). C'est ce que l'on ferait en C, mais cela n'est pas très satisfaisant. En effet, outre le fait d'avoir à changer le type de retour des fonctions, ce qui oblige souvent à changer les types des arguments, pour retourner un résultat, par effet de bord sur un argument (passé par référence). L'autre problème est qu'il est parfois difficile de traiter l'erreur même au niveau de l'appelant direct, il est possible qu'une erreur ne soit traitable qu'à un niveau supérieur.

Le bon mécanisme qui puisse répondre à ces points est le mécanisme d'exceptions. En Java, on les remarque en fait dès que l'on a une erreur à l'exécution. Si l'on fait :

```
Pile p = empty();
p.pop();
```

On obtient :

```
Exception in thread "main" java.lang.NullPointerException
        at Prog.pop(Pile.java:16)
        at Prog.main(Pile.java:23)
```

L'appel du programme a « levé une exception », qui aurait pu être « rattrapée » et traitée par l'appelant. Une exception est en quelque sorte le résultat d'un comportement erroné. Ce n'est pas vraiment une erreur au sens classique

du terme : c'est un *objet* traitable par le programme. Tout calcul peut *lancer* (« throw ») une exception, c'est-à-dire retourner en plus d'une valeur, un type d'erreur, à son appelant, qui pourra la *récupérer* (« catch ») pour la traiter, ou la relancer à son appelant etc.

Ceci est à distinguer d'une *erreur* qui ne peut être traitée et qui arrête le programme dans un état potentiellement incohérent.

Les exceptions en Java forment un certain nombre de classes : `Exception`, avec pour sous-classes, `IOException`, `RuntimeException` etc.

Cela veut dire en particulier que l'on peut créer une nouvelle exception par héritage :

```
class Monexception extends Exception {
  ...
}
```

L'instanciation se fait par :

```
new Monexception();
```

Quand une méthode peut lancer une exception, il faut le déclarer :

```
  void pop() throws Monexception {
    ... }
```

Cela déclare que `pop()` ne renvoie rien comme donnée, mais peut lancer une exception, rattrapable par l'environnement.

Lancer une exception se fait de la manière suivante. On ne doit pas faire `return new Monexception();`, mais plutôt `throw new Monexception();` par exemple :

```
  void pop() throws Monexception {
    if (this.c == null) throw new Monexception();
    this.c = this.c.tl; }
```

Le mot clé `throw` a une sémantique qui s'apparente au `return`, il interrompt le code. *Rattraper* une exception se fait de la façon suivante :

```
try { p.pop(); }
catch (Monexception e) {
 System.out.println("Pile vide!");
 ...
 }
...
```

Les exécutions possibles sont alors :

- Si `p.c` n'est pas `null`, alors `p.pop()` termine normalement, sans lever d'exception ; le code se poursuit normalement au dernier « ... ».
- Si `p.c` est `null`, alors `p.pop()` lance une exception, qui est rattrapée par `catch (Monexception e)`, `e` vaut alors l'objet de type `Monexception` crée par `p.pop()`.
- On peut lancer de nouveau cette exception à l'appelant de l'appelant et ainsi de suite.

4.5 Interfaces

Une interface est un ensemble de déclarations de méthodes sans implémentation. Cela est défini par le mot clé `interface`. Les interfaces permettent de déclarer des variables avec le type de l'interface, mais elles ne sont pas instanciables, il n'y a pas de constructeur en particulier. Une classe peut implémenter une ou plusieurs interfaces, par le mot clé `implements`, cela permet de faire de l'héritage multiple en quelque sorte, mais seulement simple, de code.

Donnons un exemple d'interface, quasi fonctionnel, avec une définition de fonctions de $\mathbb{N}$ dans $\mathbb{N}$:

```
interface Function {
  public int apply(int n);
}
```

Il y aura donc une seule méthode à implémenter pour être une fonction de $\mathbb{N}$ dans $\mathbb{N}$: l'application `apply`. En voici des exemples :

```
public class Carre implements Function {
  public int apply(int n) {
    return n*n;
  }
}

public class Fact implements Function {
  public int apply(int n) {
    ... return fact;
  }
}

public class Exemple {
  public static void main(String[] args) {
    Carre x = new Carre();
    Fact y = new Fact();
    System.out.println("Carre(3)="+x.apply());
    System.out.println("Fact(4)="+y.apply());
}
```

4.6 Héritage et typage

On définit une relation de sous-typage comme suit. On note $T \leftarrow S$ si S est un *sous-type* de T, défini par :

- $T \leftarrow T$
- si la classe S est une sous-classe de T, on a $T \leftarrow S$
- si l'interface S est une sous-interface de I, on a $I \leftarrow S$;
- si la classe C implémente l'interface I, on a $I \leftarrow C$
- si $T \leftarrow S$, alors $T[] \leftarrow S[]$
- si $S \leftarrow SS$ et $T \leftarrow S$, alors $T \leftarrow SS$

La propriété fondamentale de cette relation est que si $T \leftarrow S$, alors toute valeur du type S peut être utilisée en lieu et place d'une valeur de type T (*transtypage* implicite). En voici un exemple. Si S sous-type de T, on peut faire le transtypage (cast) implicite :

```
S x = new S(...);
T y = x;
```

Dans l'autre sens c'est interdit (même quand finalement les types sont assez similaires) :

```
class X { float n; }
class Y extends X {
  public static void main(String[] args) {
    X x = new X();
    Y y = x;
  }
}
```

À la compilation on obtient une erreur :

```
javac Y.java
Y.java:5: incompatible types
found   : X
required: Y
    Y y = x; }
          ^
1 error
```

Il faut faire dans ce cas un transtypage (cast) explicite :

```
class X { float n; }

class Y1 extends X {
  public static void main(String[] args) {
    X x = new X();
    Y1 y = (Y1) x;
  }
}
```

A la compilation tout se passe bien, mais le bon comportement dans ce cadre dépend entièrement de l'utilisateur (ce qui est différent du typage fort à la CAML, cf. chapitre 9).

En C le *cast* est d'emploi très courant, par exemple à l'allocation (cf. chapitre 3, sur l'allocation des listes en C) :

```
List res = (List) malloc(sizeof(struct stList));
```

car `malloc` renvoie le type fourre-tout `void *`.

4.7 Classes abstraites

Les classes abstraites sont une sorte d'intermédiaire entre interfaces et classes (dites *concrètes*). Elles permettent de définir des variables avec le type corres-

pondant et permettent l'héritage (comme les classes concrètes). Elles permettent aussi de spécifier des *méthodes abstraites* avec l'attribut *abstract*, qui sont des spécifications de méthodes, mais pas leur implémentation (comme les interfaces).

En voici un exemple classique (Denis Monasse) qui permet de revenir à une implémentation différente en Java des types somme (cf. chapitre 3) :

```
abstract class Carte {
  public static final int PIQUE = 0,
    COEUR = 1, CARREAU = 3, TREFLE = 4;
  int couleur;

  abstract int valeur(int couleur_atout);
}

class As extends Carte {
  int valeur(int couleur_atout) { return 11; }

class Valet extends Carte {
  int valeur(int couleur_atout) {
    if (couleur==couleur_atout)
      return 20;
    else
      return 1;
}
...
```

4.8 Paquetages

Les paquetages permettent d'organiser un ensemble de classes et d'interfaces en un tout cohérent (sorte de *module* à la CAML). Ils limitent en particulier la portée des identificateurs. Un paquetage peut avoir un nom, et des sous-paquetages (une arborescence, comme des répertoires UNIX).

En voici un exemple : l'API standard de JAVA est organisée en paquetage et sous-paquetages. Le paquetage `java` contient `java.lang`, `java.io`, `java.util` etc.

Voici une façon de définir un paquetage :

```
package monPackage;

public class A {
  public methodeA() {
    ...
  }
}

public class B {
  public methodeB() {
    ...
```

```
  }
}
```

Et d'importer un paquetage :

```
import monPackage;

...
new A().methodeA();
new B().methodeB();
```

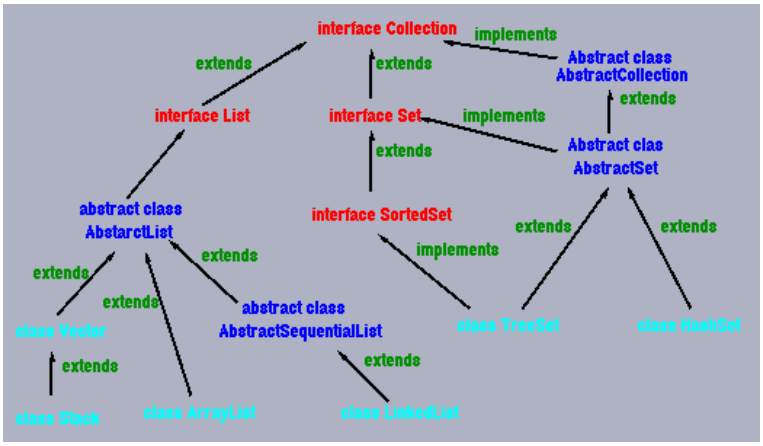
On peut importer tous les classes d'un paquetage par :

```
import java.io.*;
```

Remarques : `java.lang` est toujours importé automatiquement – toutes les classes définies jusqu'à présent étaient dans le paquetage *anonyme*.

4.9 Collections

Pour aller plus loin, sachez qu'il existe un mécanisme de collection Java, qui permet de représenter divers types d'éléments d'objets JAVA :



4.10 Les objets en (O)Caml

Pour être complet, et pour les élèves qui ont déjà une expérience en OCaml, signalons les principales différences avec Java. Les enregistrements et objets sont de même nature en Java, alors que les enregistrements et objets sont différents en Caml (la partie orienté objet est une sur-couche, venue longtemps après). Seules les méthodes peuvent accéder aux champs en Caml (pas visibles autrement). Cela correspond en fait à des méthodes *private* de Java, que l'on ne traite pas dans ce cours. Voici un exemple simple en OCaml :

```
class pile = object
 val mutable c = ([]: int list)
 method get_c () = c
 method testempty () = c=[]
 method push x = c<-x::c
 method pop () = c<-List.tl c
 method top () = List.hd c
end
```

Chapitre 5

Récursivité, calculabilité et complexité

Dans ce chapitre, nous discutons du sens des fonctions récursives, que l'on peut écrire dans tous les langages de programmation raisonnables, tel Java. On en donne tout d'abord le sens, encore informel (il sera rendu plus formel à la fin du chapitre 6). On discute ensuite de son pendant en théorie de la calculabilité : les fonctions récursives partielles. On termine enfin par quelques considérations simples (des rappels pour certains), concernant la complexité algorithmique.

5.1 La récursivité dans les langages de programmation

Le code suivant a t-il un sens ?

```
int ack(int n, int p) {
  if (n==0)
    return p+1;
  if (p==0)
    return ack(n-1,1);
  return ack(n-1, ack(n, p-1));
}
```

Oui, ce code a un sens car la « suite d'appels » définissant `ack` termine toujours. Ainsi, par exemple, ack(1,1) appelle ack(1,0) qui appelle ack(0,1) qui renvoie 2. On trouve alors ack(1,0)=2. On appelle ensuite ack(0,ack(1,0))=ack(0,2) égal à 3. Ce qui nous permet de conclure que ack(1,1) vaut 3 !

De façon générale, pour prouver la terminaison d'un processus récurrent ou récursif, il faut trouver un ordre sur les valeurs successives manipulées par la fonction récursive. Cet ordre est souvent associé à une valeur d'une fonction, décroissante à chaque appel, que l'on appelle *variant*, mais pas toujours. Ici nous introduisons un ordre très classique, l'ordre lexicographique, sur (n, p) les

deux arguments entier de `ack` : l'ordre lexicographique sur $\mathbb{N} \times \mathbb{N}$ est défini comme suit :

$$(x, y) < (x', y') \text{ ssi } \begin{cases} x < x' & \text{ou,} \\ x = x' \text{ et } y < y' \end{cases}$$

Alors il suffit de vérifier qu'à chaque appel de la fonction `ack`, les arguments successifs (n, p) sont de plus en petits, au sens de l'ordre lexicographique que nous venons de définir. Nous examinons les trois cas possibles de la définition de `ack` :

- $ack(0, p) = p + 1$: il n'y a pas d'appel récursif, la terminaison est acquise
- $ack(n + 1, 0) = ack(n, 1)$: $(n, 1) < (n + 1, 0)$ décroît à l'appel suivant
- $ack(n + 1, p + 1) = ack(n, ack(n + 1, p))$: $(n + 1, p) < (n + 1, p + 1)$ et $(n, ack(n + 1, p)) < (n + 1, p + 1)$ décroissent aux appels suivants.

Tout ceci sera défini plus formellement au chapitre 6, pour l'instant, indiquons comment ceci est implémenté et exécuté en machine.

5.2 L'exécution de fonctions récursives, la pile d'appel

5.2.1 Récursion et itération

De façon générale, on dit qu'une fonction est récursive si elle s'appelle elle-même. Plus généralement encore, des fonctions sont dites mutuellement récursives si le *graphe d'appel* des fonctions est cyclique : `f` appelle `g` qui appelle `h`...qui appelle `f` !

On peut se poser naturellement la question de l'utilité de la définition récursive de fonctions. Considérons le code Java suivant :

```
static int iter(int x) {
  while (x<1000)
  x = x+2;
  return x;
}
```

Il itère sur les nombres entiers avec un pas de deux à chaque tour. On peut définir cette fonction de façon récursive, en utilisant un argument supplémentaire qui servira d'« accumulateur », et qui contiendra au final le résultat de la fonction :

```
static int acc_iter(int x,int acc) {
  if (x>=1000)
    return acc;
  return acc_iter(x+2,acc+2);
}

static int iter(int x) {
  return acc_iter(x,x);
}
```

Les boucles simples peuvent de façon générale s'exprimer comme des appels récursifs, parfois de façon plus naturelle, ce qui n'est certainement pas le cas ici. Mais surtout, les fonctions récursives permettent d'écrire des fonctions qu'on ne pourrait pas coder autrement.

Reprenons la fonction `ack` du début de ce chapitre, dite fonction d'Ackermann, du nom du célèbre mathématicien :

```
ackermann(0,p) = p+1
ackermann(n+1,0) = ackermann(n,1)
ackermann(n+1,p+1) = ackermann(n,ackermann(n+1,p))
```

n'est pas implémentable par boucles, par contre, il se programme bien en Java, comme on l'a vu (en C) au début de ce chapitre :

```
static int ackermann(int n, int m) {
 if (n==0)
   return m + 1;
 else
   if (m==0)
     return ackermann(n-1,1);
   else
     return ackermann(n-1,ackermann(n,m-1));
}
```

De même dans d'autres langages, en C ici :

```
int ackermann(const int n, const int m) {
  if (n == 0)
    return m + 1;
  else
    if (m == 0)
      return ackermann(n - 1, 1);
    else
      return ackermann(n - 1, ackermann(n, m - 1));
}
```

Et en Caml :

```
let rec ackermann = function
  | (0, n) -> n + 1
  | (m, 0) -> ackermann (m - 1, 1)
  | (m, n) -> ackermann (m - 1, ackermann (m, n - 1));;
```

Attention néanmoins aux définitions circulaires :

```
static int f(final int x) {
  return f(x);
}
```

Ce code, qui ne termine pas, est repéré à problème par le compilateur JAVA (mais des exemples plus subtils ne le seront pas forcément) :

```
> javac toto.java
toto.java:7: cannot return a value from method whose result
```

```
type is void
  return f(1);
         ^
1 error
```

Comme on le verra dans la sémantique dénotationnelle des boucles et de la récursivité au chapitre 6, une fonction récursive ne terminant pas est codée par une fonction qui n'a pas de valeur de retour (ou la valeur « indéfinie », $\perp$).

Reprenons maintenant une définition de fonction récursive raisonnable, pour la fonction factorielle : `fact(0)=1` et `fact(n+1)=(n+1)*fact(n)`. Cette définition récursive est en fait une simple définition par récurrence, que l'on peut coder dans différents langages comme suit. En Java :

```
static int fact(final int x) {
  if (x == 0) return 1;
  return x*fact(x-1);
}
```

(Attention tout de même, et le compilateur n'est pas assez intelligent pour le voir, on n'a la terminaison que si `x` est positif.)

En C :

```
int fact(const int x) {
  if (x == 0) return 1;
  return x*fact(x-1);
}
```

Et enfin en Caml :

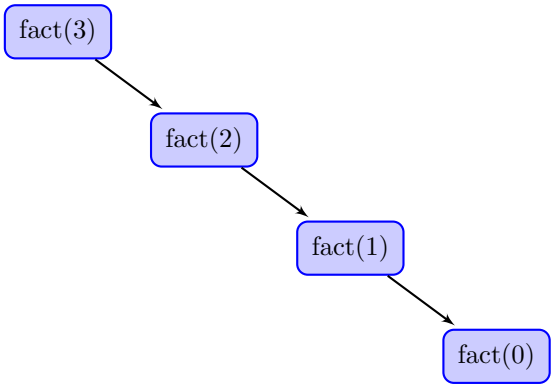
```
let rec fact = function
  0 -> 1
| n -> n*fact(n-1) ;;
```

Avant de définir la sémantique des appels récursifs, donnons en une sémantique informelle, proche de l'exécution réelle du programme, dite sémantique « par déroulement ». L'implémentation d'un code récursif repose sur une *pile d'appel*. Celle-ci permet aux appelants successifs de se souvenir du *site* d'appel (pour pouvoir revenir à l'exécution dans l'appelant apres le `return`), et du *contexte* d'appel, pour pouvoir retrouver les valeurs des variables locales à l'appelant, après `return`.

Dans le code de la factorielle, il s'agit de la valeur de `x` de l'appelant et de la ligne d'appel, ici la ligne 3 :

```
1 static int fact(final int x) {
2   if (x == 0) return 1;
3   return x*fact(x-1);
4 }
```

Ainsi, lors de l'appel de `fact(3)`, sont produit les appels successifs `fact(2)`, `fact(1)` etc. qui empilent à chaque fois la valeur de `x` de l'appelant et l'adresse de retour à l'appelant (la ligne 3) :



La *pile d'appel* contient à l'appel de `fact(0)` :

PC	Ctx
l.3	x=3
l.3	x=2
l.3	x=1

où PC est le « Program Counter », c'est-à-dir le numéro de l'instruction à laquelle il va falloir revenir au retour de l'appel récursif, et Ctx est le « Contexte », donc les valeurs des variables locales à l'appelant, qu'il faudra reprendre au retour de l'appel récursif.

Au retour de l'appel à factorielle le plus profond (`fact(0)`), `return` dépile la dernière valeur empilée, permettant au flot de contrôle de revenir à la bonne ligne de l'appelant (ligne 3) et avec la bonne valeur du contexte (x=1). Le programme termine quand la pile est vide, avec la valeur 6 ici.

Une question naturelle à se poser quand on programme de façon récursive est de savoir si cela est coûteux. En effet, l'exécution repose sur une structure de donnée supplémentaire qui peut prendre potentiellement beaucoup de mémoire. Par exemple ici, une execution de la factorielle avec une valeur initiale trop importante résulte en une erreur (pas assez de place pour pouvoir empiler de nouvelles valeurs sur la pile d'appel) :

```
> java Fact 1000000
Exception in thread "main" java.lang.StackOverflowError
        at Fact.fact(fact.java:5)
        at Fact.fact(fact.java:5)
        ...
```

Bien sûr, cet exemple n'a que peu d'intérêt dans le sens où la valeur qui serait obtenue serait de toutes façons très supérieure à ce qui est représentable dans un type `int`.

Néanmoins, comme on le démontrera sous peu, il est des choses que l'on ne peut calculer qu'au moyen de définitions récursives, et pas calculables avec des boucles simples (si on se limite à des données scalaires, et donc que l'on s'interdit des structures de données supplémentaires).

5.2.2 Dérécursivation

Dans un certain nombre de cas, les définitions récursives peuvent se dérécursiver, c'est-à-dire être transformées en boucles simples. Dans certains cas, cela est fait directement par le compilateur, par exemple dans le cas de la *récursion terminale*.

Par exemple, le code de la factorielle, récursif, peut s'écrire de façon équivalente avec une boucle `while`, par exemple ici en Java :

```
static int fact(final int x) {
  int i;
  int res = 1;
  for (i=x; i>=2; i=i-1)
    res = i*res;
  return res;
}
```

Et bien sûr en C et en Caml :

```
int fact(const int x) {
  int i;
  int res = 1;
  for (i=x; i>=2; i--)
    res = i*res;
  return res;
```

```
let fact n =
let nbr = ref 1 in
for i = 1 to n do
  nbr := (!nbr * i)
done;
!nbr;;
```

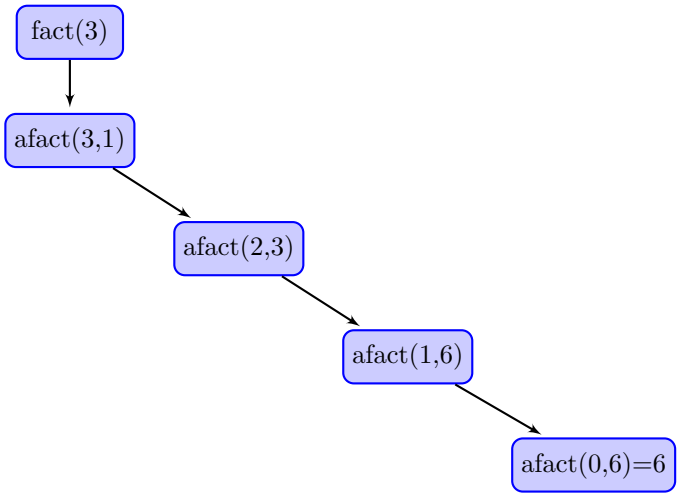
Comment faire cette transformation, de façon automatique ? C'est faisable simplement dans le cas de la récursion terminale. En voici un exemple (toujours pour la factorielle) :

```
static int afact(int n, int acc) {
  if (n == 0)
    return acc;
  return afact(n-1,n*acc);
}

static int terminal_fact(int n) {
  return afact(n,1);
}
```

La propriété caractéristique de la récursion terminale est que dans la suite d'appels effectués il n'y a pas besoin de memoriser ce qu'il reste à faire après les retours de fonctions (grâce ici à l'*accumulateur* `acc`).

Cela permet au compilateur de transformer automatiquement ce programme en code itératif, et donc de ne pas avoir à sauvegarder tous les états intermédiaires sur la pile. Dans ce cas il n'y a évidemment pas d'« explosion » mémoire possible avec ces appels récursifs.

La suite d'appel pour notre code de factorielle, version récursion terminale est ainsi :



`fact(3)` renvoie donc 6 immédiatement, sans aucun besoin de pile. Il y a juste le coût mémoire du scalaire `acc` en plus de ce que l'on a dans la version séquentielle.

5.3 Récursivité et principe de récurrence structurelle

Les codes récursifs sont très naturels quand on manipule... les structures de données récursives. C'est le cas en particulier des fonctions que l'on peut définir par *récurrence structurelle*. Donnons-en un exemple sur les listes linéaires, vues au chapitre 4.

Soit P une propriété qui nous intéresse sur un domaine de valeurs, par exemple pour commencer, les entiers naturels. Le principe de récurrence est le suivant :

– Si P est vraie en 0 (on écrit $P(0)$)
– Et si $P(n) \rightarrow P(n+1)$,

alors P est vraie sur tout N.

Sur les listes linéaires d'entiers, on a de même un principe de récurrence structurelle :

– Si P est vraie en () (la liste vide)

– Et si $P(l)$ vraie pour toutes les listes de longueur n implique $P(cons(car, l))$ est vraie (pour toute valeur de car, et toute liste l de longueur n) ;

alors P est vraie sur tout $List_{\mathbb{N}}$.

Ce principe de récurrence structurelle est la conséquence directe de la récurrence sur les entiers car on a la fonction *totale* $length : List_{\mathbb{N}} \to N$, qui vérifie $length(cons(hd, l)) = length(l) + 1$.

Appliquons maintenant un principe de définition de fonction par récurrence structurelle pour définir la fonction $length$, qui calcule la longueur d'une liste linéaire d'entiers.

En Java, on écrirait naturellement :

```
class List {
  ...
  static int length(List l) {
    if (l == null)
      return 0;
    else
      return length(l.tl)+1;
  }
}
```

Expliquons pourquoi ce code implémente bien le calcul de la longueur d'une liste linéaire. Soit $len(l)$, pour l une liste linéaire, la longueur définie mathématiquement, par récurrence structurelle :

– $len(()) = 0$
– $len(cons(car, l)) = len(l) + 1$

Soit maintenant P le prédicat : $P(l) =$ "$length(l) = len(l)$". Alors par récurrence structurelle on prouve que P est vraie sur tout le domaine des listes linéaires :

– $P(())$ est vraie car $length(()) = 0 = len(())$
– Supposons $P(l)$ vraie pour toute liste de longueur n, on calcule

$$length(cons(car, l)) = length(l) + 1 = n + 1$$

et $len(cons(car, l)) = len(l)+1 = n+1$ par hypothèse de récurrence. Donc on a $P(cons(car, l))$.

Ceci est un peu lourd pour faire la preuve d'une fonction si évidente, mais cela permet au moins de se familiariser un peu avec ce concept.

Remarque : ce principe de récurrence ne fonctionne que sur les listes linéaires et pas quelconques ; par exemple, la fonction `length` ne termine pas si on part de la liste circulaire $l = cons(0, l)$.

Dernière remarque : c'est un cas de récurrence terminale, qui peut donc se transformer aisément en code itératif :

```
static int length(List l) {
  int i = 0;
  while (l != null) {
    i = i+1;
    l = l.tl;
```

```
  }
  return i;
}
```

5.4 Partage en mémoire et récursivité

Notre restriction sur les listes linéaires est un peu plus forte que nécessaire, en fait, on peut partager des bouts de listes communs, si on s'assure que l'on ne crée pas de cycle... Dans ce cas, on peut toujours définir la fonction *length* et raisonner par récurrence structurelle. On économise juste en mémoire, et cela peut permettre de faire du *hash-consing* sur les listes – c'est-à-dire permettre de représenter l'égalité structurelle par l'égalité physique.

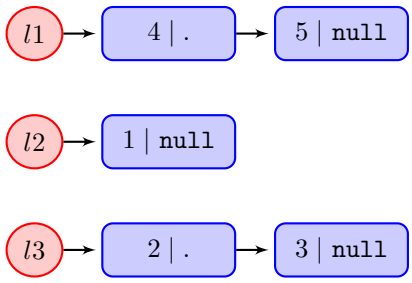
Donnons l'exemple, classique, de la fonction append : on veut concaténer une liste `l2` au bout de la liste `l1` :

```
static List append(List l1, List l2) {
  if (l1 == null) return l2;
  if (l2 == null) return l1;
  l1.tl = append(l1.tl, l2);
  return l1;
}
```

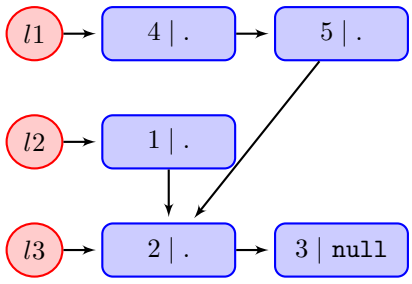
En voici un exemple d'exécution :

```
append(l1, l3);
append(l2, l3);
```

Avec :



A la deuxième et dernière étape d'exécution, on obtient :



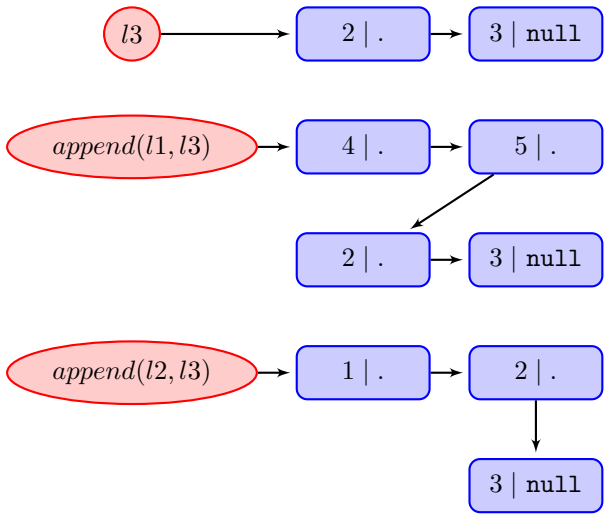
En fait, on peut écrire des codes pour `append` qui permettent à l'opposé, de ne rien partager en mémoire :

```
static List copy(List l) {
  if (l == null) return null;
  return new List(l.hd,copy(l.tl));
}

static List append(List l1, List l2) {
  if (l1 == null) return copy(l2); // return l2;
  return new List(l1.hd,append(l1.tl,l2));
}
```

Que donne dans ce cas ? :

```
append(l1,l3);
append(l2,l3);
```

avec les mêmes listes données en argument, que plus haut ?



De façon plus générale, on peut écrire ce code avec un partage possible de l'argument gauche, de l'argument droit, des deux, ou d'aucun des deux. L'intérêt ou l'inconvénient, selon, des versions sans partage est que si on modifie *en place* les listes `l1`, `l2` ou `l3`, `append(l1,l3);` et `append(l2,l3);` ne sont pas modifiées.

5.5 Les fonctions récursives primitives

Que calcule t-on dans le fragment purement impératif (voir chapitre 2) sans la récursion, et en supposant que l'on n'a que comme type de données, les entiers ? (et bien sûr pas les piles !) On prouve assez facilement que l'on obtient

la classe des fonctions récursives primitives (RP) qui est le *plus petit ensemble* de fonctions de $\mathbb{N}^n$ vers $\mathbb{N}^m$ contenant :

- les 3 fonctions de base : 0, succ (l'incrément de 1), les projections ;
- la composition de fonctions récursives primitives : si h, $g_1, \ldots, g_k$ sont des fonctions RP, $h(g_1, \ldots, g_k)$ est dans RP ;
- les fonctions définies par *récursion primitive* : g et h RP, $g : \mathbb{N}^p \to \mathbb{N}$, $h : \mathbb{N}^{p+2} \to \mathbb{N}$, alors $f : \mathbb{N}^{p+1} \to \mathbb{N}$ définie par :
  - $\forall y \in \mathbb{N}^p$, $f(0, y) = g(y)$;
  - $\forall i \in \mathbb{N}, y \in \mathbb{N}^p$, $f(succ(i), y) = h(i, f(i, y), y)$.

Les fonctions récursives primitives se programment dans tout langage de programmation impératif pur, à l'aide d'une simple instruction itérative `for` :

```
f(x,y) {
 z = g(y);
 for (i=0; i<=x-1; i=i+1) {
  z = h(i,z,y);
 }
return(z);
}
```

A l'inverse, malgré le peu de fonctions de base dans la définition des fonctions récursives primitives, on peut retrouver toutes les fonctions usuelles de l'arithmétique sur les entiers. Par exemple, la fonction prédecesseur est définie par :

```
pred(0) = 0
pred(succ(x)) = x
```

C'est exactement le schéma de récursion primitive avec $p = 0$, $g = 0$, $h(x, y) = x$:

$$\begin{aligned} f(0) &= g \\ &= 0 \\ f(succ(i)) &= h(i, f(i)) \\ &= i \end{aligned}$$

De même, la somme de deux entiers est définissable comme suit :

```
somme(0,y) = y
somme(succ(x),y) = succ(somme(x,y))
```

C'est exactement le schéma de récursion primitive avec $p = 1$, $g(y) = y$, $h(x, z, y) = succ(z)$:

$$\begin{aligned} f(0, y) &= g(y) \\ &= y \\ f(succ(i), y) &= h(i, f(i, y), y) \\ &= succ(f(i, y)) \end{aligned}$$

Le produit de deux entiers est à peine plus complexe :

```
produit(0,y) = 0
produit(succ(x),y) = somme(y,produit(x,y))
```

C'est exactement le schéma de récursion primitive avec $p = 1$, $g = 0$, $h(x, z, y) = somme(y, z)$:

$$\begin{array}{rcl} f(0, y) & = & g \\ & = & 0 \\ f(succ(i), y) & = & h(i, f(i, y), y) \\ & = & somme(f(i, y), y) \end{array}$$

Tout ceci se généralise pour définir la somme de deux fonctions RP, le produit de deux fonctions RP etc.

Une conséquence des axiomes définissant RP est le principe de *minimisation bornée* :

Si $f : \mathbb{N}^{p+1} \to \mathbb{N}$ est RP, et il existe M tel que quel que soit $y \in \mathbb{N}^p$, il existe $x \in \mathbb{N}$, $x \leq M$ tel que $f(x, y) = 0$; alors :

$$g(y) = \min\{x \in \mathbb{N}, f(x, y) = 0\}$$

est une fonction RP de $\mathbb{N}^p$ vers $\mathbb{N}$

On peut alors définir les fonctions RP sont comme étant les fonctions contenant 0, *succ* et les projections, stables par composition, et par minimisation bornée.

5.6 Fonctions récursives partielles

Dans RP, on ne peut définir que des fonctions totales (qui terminent toujours), et l'on sait bien que ce que l'on écrit naturellement dans un langage de programmation peut ne pas toujours terminer, pour de bonnes raisons ; on doit donc étendre la classe de fonctions calculables.

Posons nous également la question de savoir si la fonction d'Ackermann est dans RP. On peut essayez de la définir avec des boucles... :

```
int ack(int n, int p) {
  if (p==0)
    for (i=0;i<=n-1;i=i+1) { ??
  ...
```

Vous verrez, vous n'y arriverez pas, on a une sorte de récurrence emberlificotée sur 2 indices !

On va prouver qu'en fait `ackermann` n'est pas dans RP, mais elle est *calculable* au sens commun du terme et en un certain sens technique que l'on verra en fin de chapitre. On voit donc bien qu'il va falloir trouver une autre définition de fonction calculable. Il faut pouvoir *écrire des fonctions* et les appeler de manière *récursive* comme pour la définition `ackermann`

Montrons tout de même avant cela que `ackermann` n'est pas une fonction récursive primitive, formellement.

Tout d'abord, en calculant quelques valeurs de $A(m, n)$ on s'aperçoit que c'est une fonction qui croît extrêmement vite :

m / n	0	1	2	3
0	1	2	3	4
1	2	3	4	5
2	3	5	7	9
3	5	13	29	61
4	13	65533	265533	A(3,265533)
5	65533	A(4,65533)	A(4,A(5,1))	...
6	A(5,1)	A(5,A(5,1))	A(5,A(6, 1))	...

On peut en fait prouver que Ack croit plus vite que toute fonction RP :

$\forall f : \mathbb{N}^k \to \mathbb{N} \in RP$, $\exists a$ tel que $f(a_1, \ldots, a_k) < A(a, max(a_1, \ldots, a_k))$.

Le schéma de preuve est le suivant, on prouve successivement que :

- la fonction $x \to 0$ est majorée par $A(0, x) = x + 1$;
- la fonction $x \to succ(x)$ est majorée par $A(1, x) = x + 2$;
- les projections $\pi_k^n(x_1, \ldots, x_n) = x_k$ sont majorées par $A(0, max(x_1, \ldots, x_n)) = max(x_1, \ldots, x_n) + 1$.

Pour simplifier les notations, on écrit $\overline{x} = (x_1, \ldots, x_n)$ et $\max_{\overline{x}} = max(x_1, \ldots, x_n)$.

On prouve maintenant que si $g_1, \ldots, g_m$ sont telles que $g_i(\overline{x}) < A(r_{g_i}, max_{\overline{x}})$, et h tel que $h(\overline{x}) < A(r_h, max_{\overline{x}})$ alors $f = h(g_1, \ldots, g_m)$ est telle que $f(\overline{x}) < A(r_f, max_{\overline{x}})$ avec $r_f = r_h + 2 + max_{r_{g_j}}$

Enfin, soit $f \in RP$ définie par récursion primitive (avec g et h). Supposons $g(\overline{x}) < A(r_g, max_{\overline{x}})$, $h(\overline{x}) < A(r_h, max_{\overline{x}})$, alors une récurrence simple montre que

$$f(i, \overline{x}) < A(r_f, max(i, \overline{x}))$$

avec $r_f = 1 + max(r_g, r_h)$.

RP est le *plus petit ensemble* contenant 0, succ, les projections, stable par composition et récursion primitive, donc toute fonction dans RP peut-être majorée par un $A(a, .)$.

On termine par un argument appelé *argument diagonal de Cantor*. On considère $g(a) = A(a, a)$ Supposons g dans RP, alors $\exists r_g$, $\forall a$, $g(a) < A(r_g, a)$. Mais c'est impossible, il suffit pour cela de considérer $g(r_g) = A(r_g, r_g)$!

Que peut-on donc calculer au final avec la récursion ?

Il nous faut tout d'abord définir les fonctions partielles, car certains calculs peuvent ne pas terminer. Une fonction partielle f de $\mathbb{N}^n$ vers $\mathbb{N}^m$ est une fonction définie sur un sous-ensemble de $\mathbb{N}^n$, allant vers $\mathbb{N}^m$. On note alors $f(x) = \perp$ pour les $x \in \mathbb{N}^n$ tels que f n'est pas définie en x (recodage en une fonction totale à valeur dans $\mathbb{N}^n \cup \{\perp\}$)

On appelle maintenant R l'ensemble des fonctions récursives partielles le plus petit ensemble de fonctions *partielles* de $\mathbb{N}^n$ vers $\mathbb{N}^m$ contenant :

- 3 fonctions de base : 0, succ, projections ;
- la composition de fonctions récursives partielles : si h, $g_1, \ldots, g_k$ sont des fonctions R, $h(g_1, \ldots, g_k)$ est dans R ;

– R est clos par minimisation (pas forcément bornée cette fois-ci) : si $f : \mathbb{N}^{p+1} \to \mathbb{N}$ est R, alors

$$g(y) = \min\{x \in \mathbb{N} \mid f(x, y) = 0\}$$

est une fonction R de $\mathbb{N}^p$ vers $\mathbb{N}$.

En particulier si pour un y, il n'existe aucun x tel que $f(x, y) = 0$, alors $g(y)$ est non-définie, i.e. $g(y) = \perp$.

On a alors $RP \subseteq R$ bien évidemment. De fait, R est exactement l'ensemble des fonctions partielles obtenues par au plus une minimisation d'une fonction dans RP. Et la fonction d'Ackermann est dans R !

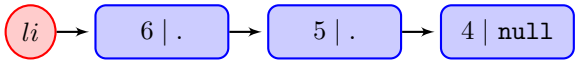
5.7 Pour aller plus loin

Les fonctions de R sont aussi appelées « fonctions calculables ». La thèse de Church concernant la calculabilité dit que « les seules fonctions calculables algorithmiquement, par une machine, sont les fonctions de R ». C'est une thèse vérifiée dans tous les modèles de calcul connus : λ-calcul, tout langage de programmation actuel, machine de Turing, machine de Post, etc. On dit qu'une machine (ou langage) est *Turing complet* si elle permet de calculer toutes les fonctions calculables – Java, C, Caml sont Turing complets.

5.8 Quelques éléments de complexité

Commençons par un cas d'étude simple en complexité algorithmique, l'inversion de liste. Etant donné une liste :

On souhaite retourner la liste contenant les mêmes éléments, mais énumérés dans l'ordre inverse :



Une implémentation possible est la suivante :

```
static List add(final List x, final int y) {
  if (x == null) return new List(y, null);
  return new List(x.hd, add(x.tl, y));
}

static List reverse(final List x) {
  if (x == null) return null;
  return add(reverse(x.tl), x.hd);
}
```

Si la liste est vide, on retourne vide, sinon on appelle récursivement l'inversion de liste sur la queue de la liste, et l'on rajoute sa tête à la fin, grâce à une autre fonction récursive, `add`.

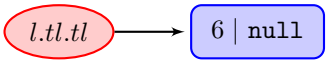
En voici un exemple d'exécution sur la liste donnée au début de cette section :

Etape 1 : on appelle `reverse` de :

Etape 2 : qui appelle `reverse` de :

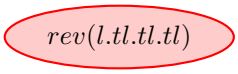
Et reprendra en faisant `add(.,4)`.

Etape 3 : puis qui appelle `reverse` de :



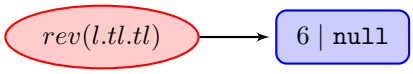
Et reprendra en faisant `add(.,5)`.

Etape 4 : qui appelle `reverse` sur :



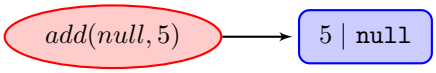
qui renvoie `null`.

A l'étape 4 : cela appelle `add(null,6);` et renvoie :



Etape 5 : puis `add(.,5)` sur le résultat précédent :

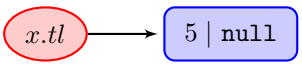
Etape 6 : qui appelle `add(null,5)` et reconstruit une cellule contenant `6` en tête :



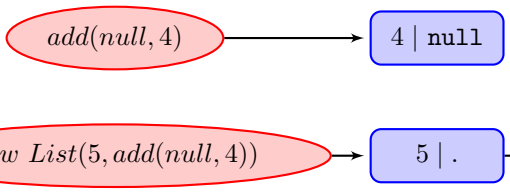
Etape 7 : En faisant `new List(6,.)` sur le résultat précédent (on obtient ainsi `rev(l.tl)`) :

Etape 8 : puis `add(.,4)` sur le résultat précédent :

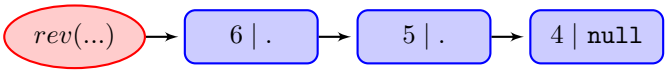
A l'étape 9, cela appelle `new List(6,add(x.tl,4))` donc appelle `add(.,4)` sur :



Et enfin aux étapes 10, 11, cela appelle `new List(5,add(null,4))` qui renvoie donc successivement :



Et enfin le dernier appel `new List(6,.)` s'effectue sur la liste précédente :



On va maintenant compter le nombre d'opérations élémentaires effectuées par `add` (dans un premier temps), en fonction de la *taille* de ses arguments. On suppose un coût unitaire pour chaque opération arithmétique, pour chaque test, pour chaque allocation mémoire etc. Cela n'est pas tout à fait vrai sur une machine moderne, mais cela n'est pas complètement faux quand on raisonne à *ordre de grandeur* près. Enfin, la taille d'une liste l (de scalaires) est comptée comme étant égale à $length(l)$.

Par récurrence, on prouve que `add` a un coût proportionnel à la longueur de `x` (complexité *linéaire*).

Supposons que la taille de `x` soit de n. Le premier appel (dans `reverse`) de `add` se fait sur une taille $n-1$: coût proportionnel à $n-1$. Le deuxième appel de `add`, se fait sur une taille $n-2$: coût proportionnel à $n-2$ etc. Le dernier appel de `add` se fait sur une taille 1 : coût proportionnel à 1.

Donc le coût total est proportionnel à $1+2+\ldots+n-1 \sim n^2$.

Que veut-on dire par « est de l'ordre de » ($\sim$) ?

On va utiliser les notations de Landau et de Hardy ; comme en analyse (ici, pour f, g deux fonctions de $\mathbb{N}$ vers $\mathbb{N}$) :

- $f = O(g)$ ssi $\exists \alpha, N, \forall n \in \mathbb{N}, n \geq N \implies f(n) \leq \alpha g(n)$;
- $f = o(g)$ ssi $\forall \epsilon, \exists N, \forall n \in \mathbb{N}, n \geq N \implies f(n) \leq \epsilon g(n)$;
- $f \sim g$ ssi $lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$.

En fait, ce qui est plus utile pour nous est l'ordre de grandeur :

- $f = \Omega(g)$ ssi $\exists \alpha, N, \forall n \in \mathbb{N}, n \geq N \implies f(n) \geq \alpha g(n)$;
- $f = \Theta(g)$ ssi $\exists \alpha, \beta, N, \forall n \in \mathbb{N}, n \geq N \implies \beta g(n) \leq f(n) \leq \alpha g(n)$.

Donc ici pour notre code `reverse` naif la complexité est en $\Theta(n^2)$.

De façon générale, la complexité d'un algorithme peut être définie en moyenne, ou dans le cas le pire. Etant donné un problème de *taille* n, la complexité d'un algorithme est $f(n)$, le nombre d'opérations *élémentaires* impliquées dans son calcul. Pour la complexité en moyenne, il s'agit de la moyenne arithmétique des nombres d'opérations pour une taille n. Pour la complexité dans le cas le pire, il s'agit du maximum du nombre d'opérations pour une taille n (sur une machine *séquentielle*).

La complexité (dans le cas le pire, et en moyenne) de l'algorithme `reverse` (naif) est en $\Theta(n^2)$. Par contre, la complexité d'un problème est la meilleure complexité d'un algorithme résolvant ce problème. La complexité de l'inversion de liste, en général est en fait de $\Theta(n)$, cf. algorithme suivant. On va utiliser une liste intermédiaire (accumulateur) :

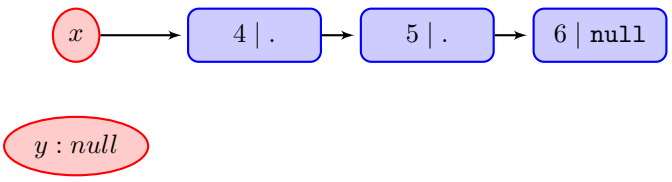
```
static List revappend(final List x, final List y) {
  if (x == null)
    return y;
  return revappend(x.tl, new List(x.hd, y));
}

static List reverse(final List x) {
  return revappend(x,null);
}
```

On peut le coder aussi de façon itérative : on parcourt alors la liste initiale dans un sens, pour construire la liste résultat dans l'autre !

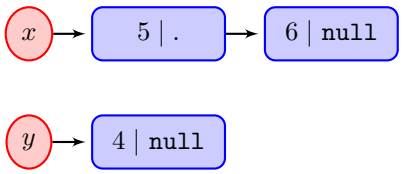
```
static List reverse(List x) {
  List res = null;
  while (x != null) {
    res = new List(x.hd, res);
    x = x.tl;
  }
  return res;
}
```

Donnons en ici un exemple d'exécution.

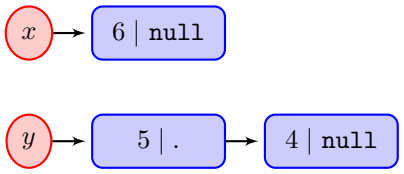
A l'étape 1, on appelle `reverse` (donc `revappend(x,y)`) avec :



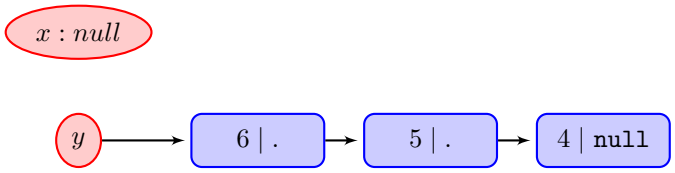
qui appelle à l'étape 2, `revappend(x.tl, new List(4,null));` donc les nouveaux arguments `x` et `y` sont :



Etape 3 : cela appelle `revappend(x.tl, new List(5,y));` donc les nouveaux arguments `x` et `y` sont :



L'étape finale appelle `revappend(null, new List(6,y));` donc les nouveaux arguments `x` et `y` sont :



Cela est bien plus rapide. Etant donnée une liste de taille n, on fait un appel à `revappend` sur une liste de taille n plus une opération élémentaire (création d'une cellule de liste) ; qui fait un appel à `revappend` sur une liste de taille $n-1$ et ainsi de suite ; et se termine par `return y`. En tout, on a de l'ordre de $\Theta(n)$ opérations, et on ne peut faire mieux sur une machine séquentielle.

On peut maintenant définir les classes de complexité algorithmiques.

Donnons quelques exemples de complexité *en temps*. Dans l'ordre (strictement) croissant, les plus classiques sont :

Temps	Classe	Exemple
$\Theta(1)$	temps constant	addition d'entiers (machine)
$\Theta(log(n))$	temps logarithmique	
$\Theta(n)$	temps linéaire	inversion de liste
$\Theta(n\ log(n))$	temps quasi-linéaire	tri
$\Theta(n^2)$	temps quadratique	produit matrice vecteur
$\Theta(n^p)$	temps polynomial	multiplication de matrices
$\Theta(e^{\mathrm{n}})$	temps exponentiel	plus tard...

On note classiquement : L pour le temps logarithmique (*fonctions très peu coûteuses*), P (ou PTIME) pour le temps polynomial (*pas trop coûteuses*), et EXPTIME pour le temps exponentiel (*très coûteuses*).

Donnons maintenant quelques exemple d'ordres de grandeur (en secondes) :

Classe/n	1	2	5	10	...	20	50
L	0...	0.3	0.7	1		1.3	1.7
$\Theta(n)$	1	2	5	10		20	50
$\Theta(n\ log(n))$	0...	0.6	3.5	10		26	85
$\Theta(n^2)$	1	4	25	100		400	2500
$\Theta(n^3)$	1	8	125	1000		8000	125000
$\Theta(e^n)$	2.7	7.4	148	22026		$4.9.10^8$	$5.2.10^{21}$

Ce dernier nombre fait environ $1.6.10^{14}$ années alors que l'âge de l'univers estimé est d'environ 15.10^9 années... notre système solaire aura disparu depuis bien longtemps.

Pour aller plus loin, disons un mot sur les classes NP et surtout la NP-complétude. Sans rentrer dans les détails, et sans définir précisément ce que veut dire NP, les problèmes NP-complets sont les problèmes les « plus durs » dont on peut vérifier la solution en temps polynomial. Hélas beaucoup de problèmes classiques sont NP-complets. Par exemple SAT, qui consiste à déterminer si une formule logique (propositionnelle – cf. chapitre 7) est toujours vraie ou pas. Cela est, en termes de complexité, aussi compliqué qu'énumérer les valeurs booléennes pour toutes les variables (exponentiel) et tester si la formule est vraie avec ces affectations (polynomial) ! On ne connaît à l'heure actuelle que des algorithmes EXPTIME au mieux pour résoudre les problèmes NP-complets. On ne sait pas à l'heure actuelle si $P \neq NP$ (mais bien sûr $P \subseteq NP$), et cela reste un problème à 1 million de dollars ! (Clay Institute)

Il y a également une notion de complexité *en espace*. On compte non pas le temps mais le nombre d'octets nécessaire à résoudre un problème algorithmique. On peut prouver alors la relation suivante entre la complexité en temps, et en espace :

$$L \subseteq P \subseteq PSPACE \subseteq EXPTIME$$

Pour aller plus loin, suivez le cours INF423 !

Exercices

1. Soient f et g deux fonctions récursives primitives. Prouver que $f + g$ et $f \times g$ sont récursives primitives.
2. Est-ce que le prédicat P suivant sur des programmes Java ne prenant pas d'argument en entrée et renvoyant un entier, est décidable ?

$$P(q) = vrai \text{ si le programme } q \text{ renvoie 0, sinon } P(q) = faux$$

3. (++) Est-ce que la fonction (fonction de Sudan) suivante, est récursive primitive ?

$$\begin{array}{rcll} F(0,x,y) & = & x+y & \\ F(n+1,x,0) & = & x & \text{si } n \geq 0 \\ F(n+1,x,y+1) & = & F(n,F(n+1,x,y),F(n+1,x,y)+y+1) & \text{si } n \geq 0 \end{array}$$

4. Si f est une fonction en $\Theta(n)$, et g une fonction en $\Theta(n)$, quel est l'ordre de grandeur (en Θ) de fg ?
5. Quelle est la complexité de l'algorithme récursif permettant de résoudre le problème des tours de Hanoi (TD3-5) ?

Chapitre 6

Sémantique dénotationnelle de langages impératifs

Jusqu'à présent, on a donné des éléments de sémantique dénotationnelle dans des cas simples : les expressions arithmétiques, les affectations et les tests. Le cas de la boucle est bien plus compliqué. En quelque sorte, pour donner un sens mathématique à la boucle, il faut procéder à calcul par approximations finies (déroulements successifs) et montrer qu'il converge toujours, soit vers un environnement bien défini, soit vers $\bot$, indiquant la non-terminaison. Pour ce faire, il va nous falloir introduire des notions de mathématiques discrètes, structures ordonnées et théorèmes de point fixe. Pour le même prix, nous saurons alors donner une sémantique mathématique, donc raisonner, et prouver les programmes comme au chapitre 7, aux programmes récursifs. Le lecteur pourra consulter avec profit [14] pour des détails sur la sémantique dénotationnelle, opérationnelle (non traitée ici) et axiomatique (non traitée ici également) de langages de programmation simples, et [1] pour avoir plus de détails sur les fondements mathématiques (théorie des domaines) de la sémantique mathématique) [1].

6.1 Sémantique élémentaire

On considère ici un langage jouet, sorte de fragment impératif de Java, sans les tableaux ni les références. On appelle Var les variables, Val les valeurs que peuvent prendre les variables ; on nomme également $\mathrm{Val}_\bot$ l'ensemble $\mathrm{Val} \cup \{\bot\}$, qui permet de rajouter une valeur symbolique « bottom » ($\bot$) représentant une valeur indéfinie. On note les environnements $\rho \in \mathrm{Env}_\bot$ qui sont des fonctions partielles de Var vers $\mathrm{Val}_\bot$: $\rho : \mathrm{Var} \to \mathrm{Val}_\bot$. Par abus de notation, on définit l'environnement $\bot$ tel que $\bot(x) = \bot$ pour tout $x \in \mathrm{Var}$.

Ce langage permet de définir des expressions arithmétiques, que l'on note AExpr, des tests, notés BExpr, et définit également des instructions : les affec-

1. Le premier livre se trouve à la BCX, et le deuxième peut se trouver sur le web.

tations, les conditionnelles, et les boucles `while`.

La sémantique pour les expressions arithmétiques est définie par récurrence structurelle sur AExpr, comme au chapitre 2 : pour $a \in \text{AExpr}$, on définit $[\![a]\!] : \text{Env}_\perp \to \text{Val}_\perp$ par :

$$\begin{array}{lcl} [\![n]\!]\rho & = & n \\ [\![v]\!]\rho & = & \rho(v) \\ [\![a_0 + a_1]\!]\rho & = & [\![a_0]\!]\rho + [\![a_1]\!]\rho \\ [\![a_0 - a_1]\!]\rho & = & [\![a_0]\!]\rho - [\![a_1]\!]\rho \\ [\![a_0 * a_1]\!]\rho & = & ([\![a_0]\!]\rho) * ([\![a_1]\!]\rho) \\ \ldots & & \end{array}$$

L'addition, la multiplication etc. sont en fait ici les extensions *strictes* des opérations correspondantes sur Val, c-.à-.d. que $\perp + x = x + \perp = \perp$ pour l'addition, et ainsi de suite.

Pour $b \in \text{BExpr}$ expression booléenne, on définit également par récurrence structurelle $[\![b]\!] : \text{Env}_\perp \to \{true, false\}_\perp$ par :

$$\begin{array}{lcl} [\![true]\!]\rho & = & true \\ [\![false]\!]\rho & = & false \\ [\![a_0 == a_1]\!]\rho & = & \left\{ \begin{array}{ll} true & \text{si } [\![a_0]\!]\rho = [\![a_1]\!]\rho \neq \perp \\ \perp & \text{si } [\![a_0]\!]\rho = \perp \text{ ou } [\![a_1]\!]\rho = \perp \\ false & \text{sinon} \end{array} \right. \\ [\![a_0 < a_1]\!]\rho & = & \left\{ \begin{array}{ll} true & \text{si } [\![a_0]\!]\rho < [\![a_1]\!]\rho \\ \perp & \text{si } [\![a_0]\!]\rho = \perp \text{ ou } [\![a_1]\!]\rho = \perp \\ false & \text{sinon} \end{array} \right. \\ \ldots & & \end{array}$$

L'affectation est interprétée comme suit. Pour $v \in \text{Var}$; $e \in \text{AExpr}$, et $\rho \in \text{Env}_\perp$:

$$[\![v = e]\!]\rho = \rho[[\![e]\!]\rho / v]$$

où pour $u, v \in \text{Var}$ et $n \in \text{Val}$:

$$\rho[n/v](u) = \left\{ \begin{array}{ll} \rho(u) & \text{si } u \neq v \\ n & \text{si } u = v \end{array} \right.$$

Prenons un exemple simple de calcul de la sémantique de l'affectation : Considérons $x = 3 * y + 5$ dans l'environnement ρ tel que $\rho(y) = 3$ et $\rho(x) = 1$, on obtient l'environnement σ avec

$$\sigma(u) = \left\{ \begin{array}{ll} \rho(u) = 3 & \text{si } u = y \\ [\![3 * y + 5]\!]\rho = 14 & \text{si } u = x \end{array} \right.$$

Cette dernière égalité est vraie car dans l'environnement ρ où $\rho(y) = 3$:

$$\begin{array}{lcl} [\![3 * y + 5]\!]\rho & = & [\![3 * y]\!]\rho + [\![5]\!]\rho \\ & = & [\![3 * y]\!]\rho + 5 \\ & = & ([\![3]\!]\rho)\,([\![y]\!]\rho) + 5 \\ & = & 3 * 3 + 5 \\ & = & 14 \end{array}$$

Pour la conditionnelle :

$$[\![\text{if } b \ p1 \text{ else } p2]\!]\rho = \begin{cases} [\![p1]\!]\rho & \text{si } [\![b]\!]\rho = true \\ [\![p2]\!]\rho & \text{si } [\![b]\!]\rho = false \\ \bot & \text{si } [\![b]\!]\rho = \bot \end{cases}$$

On a également la règle pour interpréter la séquence d'instruction :

$$[\![p; q]\!]\rho = [\![q]\!]\,([\![p]\!]\rho)$$

Faire en effet p puis q à partir de l'environnement ρ est équivalent à faire p à partir de l'environnement ρ, puis d'appliquer à ce nouvel environnement la transformation $[\![q]\!]$. La séquence est transformée ainsi en une composition de fonctions. C'est pourquoi la sémantique que l'on donne ici est appelée « compositionnelle » (ou « dénotationnelle » : à chaque élément de syntaxe, on associe une « dénotation », de façon compositionnelle).

Traitons maintenant du cas de la boucle, qui est le point délicat de la sémantique dénotationnelle de notre langage. Observons tout d'abord que toutes les boucles ne terminent pas. Par exemple :

```
while (true) {
  ...
}
```

Donc $[\![P]\!]\rho$ ne peut pas toujours renvoyer un $\sigma \in \text{Env}$, mais seulement une fonction partielle ou une fonction totale à valeur dans $\text{Env}_\bot$, et on notera $[\![P]\!]\rho = \bot$ si P ne termine pas.

Essayons de déterminer quelle devrait être la sémantique d'une boucle, a priori. Pour $w = \text{while } b \ \{ \ c \ \}$, on doit avoir

$$w \sim \text{if } b \text{ then } c; w \text{ else } skip$$

($skip$ est une instruction qui ne fait rien, $[\![skip]\!]\rho = \rho$) Donc on doit avoir :

$$\begin{aligned} [\![w]\!]\rho &= \begin{cases} [\![c; w]\!]\rho & \text{si } [\![b]\!]\rho = true \\ [\![skip]\!]\rho & \text{si } [\![b]\!]\rho = false \end{cases} \\ &= \begin{cases} [\![w]\!]\,([\![c]\!]\rho) & \text{si } [\![b]\!]\rho = true \\ \rho & \text{si } [\![b]\!]\rho = false \end{cases} \end{aligned}$$

C'est une définition circulaire, comment en donner un sens ? En fait, cela peut s'écrire comme une équation aux points fixes. Soit ϕ une *fonction partielle* de Env vers Env – ou de façon équivalente une fonction totale de $\text{Env}_\bot$ vers $\text{Env}_\bot$, et stricte ($\phi(\bot) = \bot$). Soit F la fonctionnelle *stricte* envoyant $\phi : \text{Env}_\bot \to \text{Env}_\bot$ vers une autre fonction du même type $F(\phi) : \text{Env}_\bot \to \text{Env}_\bot$:

$$F(\phi)(\rho) = \begin{cases} \phi\,([\![c]\!]\rho) & \text{si } [\![b]\!]\rho = true \\ \rho & \text{si } [\![b]\!]\rho = false \\ \bot & \text{si } [\![b]\!]\rho = \bot \end{cases}$$

On veut ϕ tel que $\phi = F(\phi)$ point fixe (en fait le *plus petit* en un certain sens, que l'on expliquera plus loin).

6.2 Problèmes de points fixes

Quel est le sens de ce point fixe maintenant ? On n'a hélas pas de continuité et de support compact, de fonctions k-lipschiptziennes et autres, qui nous permettraient d'appliquer un des théorèmes de point fixe de l'analyse mathématique. Mais on a une structure partiellement ordonnée de fonctions partielles, et des théorèmes de points fixes pour ces structures, Tarski et Kleene, que nous allons décrire tout de suite.

Le cadre général est celui des ordres partiels. Un ordre partiel est un couple $(P, \leq)$ d'un ensemble P et d'une relation binaire $\leq$ défini sur P (c'est-à-dire que $\leq \subseteq P \times P$) telle que $\leq$ est réflexive : $\forall p \in P, p \leq p$; transitive : $\forall p, q, r \in P, p \leq q \ \&\ q \leq r \implies p \leq r$; et anti-symétrique : $\forall p, q \in P, p \leq q \ \&\ q \leq p \implies p = q$.

Par exemple $(\mathbb{R}, \leq)$ est un ordre partiel. En fait, on a même pour cet ordre partiel, la propriété que $x \leq y$ ou $y \leq x$ pour tout $x, y \in \mathbb{R}$. C'est ce que l'on appelle un ordre *total*. Un autre exemple est $(\wp(S), \subseteq)$, qui est l'ensemble des sous-ensembles d'un ensemble S, avec pour ordre partiel, l'inclusion ensembliste.

Dans les cas qui nous intéressent, on peut définir un ordre sur $\mathrm{Env}_\perp$ comme suit. Pour $\rho \in \mathrm{Env}_\perp$, $\rho' \in \mathrm{Env}_\perp$,

$$\rho \leq \rho'$$

si

$$\rho'(x) = \rho(x) \text{ si } \rho(x) \neq \perp$$

(ρ *restriction* de ρ' à un sous-domaine de Var ; ou ρ' *extension* de ρ).

Dans un ordre partiel, il y a une notion de majorant, et de minorant. Soit $(P, \leq)$ un ordre partiel. Pour $X \subseteq P$: p est un majorant de X si $\forall q \in X, q \leq p$; p est le plus petit majorant (*sup*) si : p est un majorant de X et pour tous les majorants q de X, $p \leq q$. Le plus petit majorant, s'il existe, est noté : $\bigcup X$. De même (en renversant l'ordre), on définit les minorants, et le plus grand des minorants (*inf*) s'il existe, et que l'on note $\bigcap X$. Un treillis est un ordre partiel ayant un *sup* et un *inf* pour tout X à deux éléments (et par récurrence, pour tout ensemble fini non vide X)

Reprenons nos exemples. $(\mathbb{R}, \leq)$ est non seulement un ordre total, mais en plus, un treillis (le *sup* de deux éléments est leur *max*, l'*inf* est leur *min*). De même, $(\wp(S), \subseteq)$ est un treillis : le *sup* de deux éléments (ensembles) est leur union ensembliste, l'*inf* de deux éléments (ensembles) est leur intersection ensembliste.

Donnons également des exemples d'ordres partiels qui ne sont pas des treillis. $(\mathrm{Env}_\perp, \leq)$ n'est pas un treillis : il suffit de considérer par exemple ρ et σ uniquement définis sur $x \in \mathrm{Var}$, avec $\rho(x) = 1$, $\sigma(x) = 2$. Que pourrait alors valoir $\rho \cup \sigma$? On voit bien qu'on ne peut le définir.

Malgré tout, ce dernier ordre partiel a d'autres bonnes propriétés, de CPO (acronyme de *Complete Partial Order*), comme nous allons le définir dans la suite. Pour P ordre partiel, une chaîne de P est $p_0 \leq p_1 \leq \ldots \leq p_n$. De même, une ω-chaîne de P est $p_0 \leq p_1 \leq \ldots \leq p_n \leq \ldots$, c'est-à-dire est une suite infinie d'éléments de P.

On dit qu'un ordre partiel $(P, \leq)$ est un CPO si toute ω-chaîne $(p_i)_{i\in\mathbb{N}}$ de P admet un *sup* pour $\leq$, et si $(P, \leq)$ admet un plus petit élément encore noté $\bot$, c'est-à-dire $\forall p \in P, \bot \leq p$.

On dit également qu'un treillis est *complet* si tous ses sous-ensembles X admettent un *sup* et un *inf* (condition en fait redondante) ; en particulier, il admet toujours un plus petit élément : $\bot = \bigcup \emptyset$, et un plus grand élément : $\top = \bigcup P$.

Ainsi, $(\mathbb{R}, \leq)$ n'est pas un treillis complet, ni un CPO, mais $\mathbb{R} \cup \{-\infty, \infty\}$ est un treillis complet (donc un CPO). De même, $(\wp(S), \subseteq)$ est un treillis complet, avec $\bot = \emptyset$, $\top = S$. $(\mathrm{Env}_\bot, \leq)$ est un CPO tel que pour toute ω-chaîne $\rho_0 \leq \rho_1 \leq \ldots \leq \rho_n \leq \ldots$ on a

$$\left(\bigcup_{i\in\mathbb{N}} \rho_i\right)(x) = \begin{cases} \rho_j(x) & \text{si } \exists j \in \mathbb{N},\ \rho_j(x) \neq \bot \\ \bot & \text{sinon} \end{cases}$$

A partir d'un CPO et d'un ensemble, on peut aisément construire un autre CPO, comme l'indique le lemme suivant :

Lemme 1. *Supposons que $\mathcal{C}$ est un CPO, A est un ensemble. Alors $\mathcal{C}^A$ (noté aussi $A \to \mathcal{C}$) l'ensemble des fonctions de A vers $\mathcal{C}$, muni de l'ordre : $f \leq g$ si $\forall a \in A$, $f(a) \leq_\mathcal{C} g(a)$ est un CPO.*

Preuve. Soit $f_0 \leq f_1 \leq \ldots \leq$ une ω-chaîne dans $\mathcal{C}^A$, on note $f_\infty : A \to \mathcal{C}$ la fonction définie par : pour tout $a \in A$, $f_\infty(a) = \bigcup_{i\in\mathbb{N}} f_i(a)$ (raisonnable, car pour tout $a \in A$, $f_0(a) \leq f_1(a) \leq \ldots$ est une ω-chaîne dans le CPO $\mathcal{C}$).

Alors $f_\infty \geq f_i$ pour tout i et si on suppose que l'on a $g : A \to \mathcal{C} \geq f_i$ pour tout i, on en déduit :

$$\text{pour tout } a \in A,\ g(a) \geq f_i(a),\ \text{donc } g(a) \geq \bigcup_{i\in\mathbb{N}} f_i(a) = f_\infty(a)$$

. □

Certaines fonctions vont jouer un rôle particulier entre des CPOs : les fonctions continues, et les fonctions croissantes.

On dit qu'une fonction $F : D \to E$ d'un CPO $(D, \sqsubseteq)$ vers un CPO $(E, \subseteq)$ est croissante si $\forall d, d' \in D,\ d \sqsubseteq d' \Rightarrow F(d) \subseteq F(d')$.

Une fonction F croissante est dite continue si pour toutes les ω-chaînes $d_0 \sqsubseteq d_1 \sqsubseteq \ldots \sqsubseteq d_n \sqsubseteq \ldots$ de D, on a :

$$\bigcup_{n\in\mathbb{N}} F(d_n) = F\left(\bigsqcup_{n\in\mathbb{N}} d_n\right)$$

L'appellation de continuité vient de l'analogie avec la topologie, que l'on peut rendre précise au moins partiellement ici. Tout d'abord, remarquons que l'ensemble des ouverts $\mathcal{O}(X)$ d'un espace topologique X, muni de l'inclusion, forme un CPO. Maintenant, une fonction continue, au sens topologique du terme $f : X \to Y$ induit une fonction $\tilde{f} : \mathcal{O}(Y) \to \mathcal{O}(X)$ par $\tilde{f}(o_Y) = f^{-1}(o_Y) \in \mathcal{O}(X)$.

$\tilde{f}$ est ainsi croissante, et continue, au sens des structures ordonnées. Il existe en fait des correspondances exactes entre structures ordonnées et topologies (souvent non Hausdorff). C'est un sujet qui se trouve au coeur de la *dualité de Stone* et de la *théorie des domaines* (fondement de la sémantique dénotationnelle de langages fonctionnels), que les étudiants intéressés pourront poursuivre en M2 [2].

Dans le cas de $\text{Env}_\perp \to \text{Env}_\perp$, on peut se poser la question de caractériser les fonctions croissantes, cela nous sera utile par la suite (ainsi qu'au chapitre 10). Soit $f : \text{Env}_\perp \to \text{Env}_\perp$ croissante. On obtient que si ρ' est une extension de ρ, $f(\rho')$ est une extension de $f(\rho)$.

De même, qu'est-ce qu'une fonction $f : \text{Env}_\perp \to \text{Env}_\perp$ continue ? C'est déjà une fonction f croissante. Elle est en plus telle que pour toute ω-chaîne $\rho_0 \leq \rho_1 \leq \ldots \leq \rho_n \leq \ldots$ les deux calculs suivants sont égaux, pour tout $x \in \text{Var}$:

–
$$\left(\bigcup_{i\in\mathbb{N}} f(\rho_i)\right)(x) \quad = \quad \begin{cases} f(\rho_j)(x) & \exists j \in \mathbb{N},\ f(\rho_j)(x) \neq \perp \\ \perp & \text{sinon} \end{cases}$$

–
$$f\left(\bigcup_{i\in\mathbb{N}} \rho_i\right)(x) \quad = \quad f\left(y \to \begin{cases} \rho_j(y) & \exists j \in \mathbb{N},\ \rho_j(y) \neq \perp \\ \perp & \text{sinon} \end{cases}\right)(x)$$

Remarque : les deuxièmes membres plus haut sont bien définis. Par exemple, pour le premier, si $\exists j, f(\rho_j)(x) \neq \perp$, alors comme $\rho_0 \leq \rho_1 \leq \ldots ...$ et f croissante, $f(\rho_0) \leq f(\rho_1) \leq ...$ donc par définition de notre ordre, si $f(\rho_j)(x) \neq \perp$ tous les $f(\rho_i)(x)$ sont égaux (à $f(\rho_j)(x)$) puisque $f(\rho_i) \leq f(\rho_j)$ et $f(\rho_j)(x)$ défini implique $f(\rho_i)(x) = f(\rho_j)(x)$ (et de même pour $f(\rho_j) \leq f(\rho_i)$).

Pour la sémantique des boucles que l'on essaie de construire, le domaine d'intérêt est $\mathcal{D} = \text{Env}_\perp \to \text{Env}_\perp$. L'ordre partiel sur ce domaine est défini comme suit. Pour $\phi \in \mathcal{D}$, $\psi \in \mathcal{D}$, $\phi \leq \psi$, si pour tout $\rho \in \text{Env}_\perp$, $\phi(\rho) \leq \psi(\rho)$, c'est-à-dire si pour tout $\rho \in \text{Env}_\perp$, $\phi(\rho)$ est une restriction de $\psi(\rho)$ à un sous-domaine de Var. C'est un CPO par le lemme 1.

Définissons maintenant ce que sont les points fixes, les pré-points fixes, et les post-points fixes.

Soit $f : D \to D$ croissante pour un ordre partiel D. Un point fixe de f est un élément d de D tel que $f(d) = d$. Un post-point fixe de f est un élément d de D tel que $f(d) \sqsubseteq d$. Un pré-point fixe de f est un élément d de D tel que $d \sqsubseteq f(d)$.

On a alors deux théorèmes de point fixe très classiques (on se servira surtout du deuxième dans ce cours) :

Théorème 1. *(Tarski) Soit $f : D \to D$ une fonction croissante sur un treillis complet D. Alors f admet au moins un point fixe. De plus, l'ensemble des points fixes de f est un treillis complet, ainsi il existe toujours un unique plus petit point fixe, noté $lfp(f)$ (« least fixed-point ») et un plus grand point fixe, noté $gfp(f)$ (« greatest fixed-point »).*

2. On pourra consulter avec intérêt [1].

Preuve. On considère $m = \bigcap\{x \in D \mid f(x) \leq_D x\}$ et $M = \bigcup\{x \in D \mid x \leq_D f(x)\}$. On montre que m est le plus petit point fixe de f et que M est le plus grand point fixe de f.

Soit $X = \{x \in D \mid f(x) \leq_D x\}$. Soit $x \in X$: on a $m \leq_D x$, donc $f(m) \leq_D f(x)$. Mais $f(x) \leq_D x$ parce-que $x \in X$. Donc $f(m) \leq_D x$ pour tout $x \in X$. Donc $f(m) \leq_D m$.

Ainsi $f(f(m)) \leq_D f(m)$, ce qui implique que $f(m) \in X$, et donc $m \leq_D f(m)$. Enfin, on conclut : $f(m) = m$.

Dernier argument : m est défini comme étant l'*inf* d'un ensemble contenant en particulier tous les points fixes de f, donc m est non seulement un point fixe mais *le plus petit point fixe de* f. □

Théorème 2. *(Kleene) Soit $f : D \to D$ une fonction continue sur un CPO D (avec un plus petit élément $\bot$). Alors,*

$$fix(f) = \bigsqcup_{n \in \mathbb{N}} f^n(\bot)$$

est le plus petit point fixe de f (qui existe ainsi !).

Preuve. Par continuité de f :

$$\begin{array}{rcl} f(fix(f)) & = & f\left(\bigsqcup_{n\in\mathbb{N}} f^n(\bot)\right) \\ & = & \bigsqcup_{n\in\mathbb{N}} f^{n+1}(\bot) \\ & = & \bigsqcup_{n\in\mathbb{N}} f^n(\bot) \\ & = & fix(f) \end{array}$$

Supposons que d est un point fixe de f. On a $\bot \leq_D d$, donc $f(\bot) \leq_D f(d) = d$ par croissance de f, et, par récurrence, $f^n(\bot) \leq_D d$. Ainsi $fix(f) \leq_D d$. □

6.3 Sémantique de la boucle `while`

Revenons à l'interprétation des boucles `while`.

Par récurrence sur les termes c du langage, on suppose $[\![c]\!] \in \mathcal{D}$, alors pour $\phi \in \mathcal{D}$:

$$F(\phi)(\rho) = \begin{cases} \phi\left([\![c]\!]\rho\right) & \text{si } [\![b]\!]\rho = true \\ \rho & \text{si } [\![b]\!]\rho = false \\ \bot & \text{si } [\![b]\!]\rho = \bot \end{cases}$$

Pour pouvoir appliquer le théorème 2 il faut prouver que $F : \mathcal{D} \to \mathcal{D}$ est continue, pour l'ordre sur $\mathcal{D}$.

On commence par en prouver la croissance. Pour $\phi \leq_{\mathcal{D}} \psi$, on vérifie que $F(\phi) \leq_{\mathcal{D}} F(\psi)$; pour tout $\rho \in \mathrm{Env}_\bot$, par exemple dans le cas $[\![b]\!]\rho = true$ (les autres cas sont triviaux) :

$$\begin{array}{rcl} F(\phi)(\rho) & = & \phi([\![c]\!]\rho) \\ & \leq & \psi([\![c]\!]\rho) \\ & = & F(\psi)(\rho) \end{array}$$

Maintenant, pour toute suite $\phi_0 \leq_{\mathcal{D}} \ldots$, et tout $\rho \in \mathrm{Env}_\perp$:

$$\left(\bigcup_i F(\phi_i)\right)(\rho) = F(\bigcup_i \phi_i)(\rho)$$

Mais :

$$\left(\bigcup_i F(\phi_i)\right)(\rho) = \begin{cases} \left(\bigcup_i \phi_i\right)(\llbracket c \rrbracket \rho) & \text{si } \llbracket b \rrbracket \rho = true \\ \rho & \text{si } \llbracket b \rrbracket \rho = false \\ \perp & \text{si } \llbracket b \rrbracket \rho = \perp \end{cases}$$

et

$$F(\bigcup_i \phi_i)(\rho) = \begin{cases} \bigcup_i \phi_i(\llbracket c \rrbracket \rho) & \text{si } \llbracket b \rrbracket \rho = true \\ \rho & \text{si } \llbracket b \rrbracket \rho = false \\ \perp & \text{si } \llbracket b \rrbracket \rho = \perp \end{cases}$$

Le seul cas où il y ait à prouver quelque chose est le premier ($\llbracket b \rrbracket \rho = true$). Ceci est trivial, car $\left(\bigcup_i \Phi_i\right)\sigma = \bigcup_i (\Phi_i \sigma)$ par définition de l'ordre (point à point).

Ceci permet de donner la sémantique de la boucle `while`. En effet, le théorème 2 s'applique à ce F continue sur le CPO $\mathcal{D}$. Montrons en pratique ce que cela veut dire. En fait, le théorème de Kleene appliqué au problème de la sémantique du `while` est exactement une sémantique par approximations finies, où on approxime en déroulant la boucle un peu plus à chaque fois.

On considère par exemple dans la suite, la sémantique de $\llbracket$while (x<10) x=x+1$\rrbracket$. On identifie $\rho \in \mathrm{Env}_\perp$ avec $\rho(x) \in \mathrm{Val}_\perp$ car il y a une seule variable. La fonctionnelle F est, pour $\Phi : \mathrm{Env}_\perp \to \mathrm{Env}_\perp$:

$$F(\Phi)(\rho) = \begin{cases} \Phi(\rho + 1) & \text{si } \rho < 10 \\ \rho & \text{si } \rho \geq 10 \\ \perp & \text{si } \rho = \perp \end{cases}$$

À la première itération, on trouve :

$$F^1(\rho) = F(\perp)(\rho) = \begin{cases} \perp & \text{si } \rho < 10 \\ \rho & \text{si } \rho \geq 10 \\ \perp & \text{si } \rho = \perp \end{cases}$$

C'est ce que l'on obtient si on ne passe pas par le corps de boucle (on sort en 0 itération).

Puis à la deuxième :

$$\begin{aligned} F^2(\rho) = F(F^1)(\rho) &= \begin{cases} F^1(\rho + 1) & \text{si } \rho < 10 \\ \rho & \text{si } \rho \geq 10 \\ \perp & \text{si } \rho = \perp \end{cases} \\ &= \begin{cases} \perp & \text{si } \rho < 9 \\ 10 & \text{si } \rho = 9 \\ \rho & \text{si } \rho \geq 10 \\ \perp & \text{si } \rho = \perp \end{cases} \end{aligned}$$

C'est ce que l'on obtient si on passe zéro ou une fois par le corps de boucle (on sort en 0 ou 1 itération).

Par récurrence, on prouve que l'on trouve, à la $(n+1)$ième itération :

$$F^{n+1}(\rho) = F(F^n)(\rho) = \begin{cases} \bot & \text{si } \rho < 10-n \\ 10 & \text{si } \rho = 10-n, \ldots, 9 \\ \rho & \text{si } \rho \geq 10 \\ \bot & \text{si } \rho = \bot \end{cases}$$

C'est ce que l'on obtient si on passe de zéro à n fois dans le corps de boucle. La limite $\bigcup_{n\in\mathbb{N}} F^n(\bot)$ contient tous les comportements possibles.

6.4 Sémantique des fonctions récursives

Ce procédé nous permet également de donner une sémantique pour des fonctions mutuellement récursives. On suppose que l'on autorise dans notre langage les définitions récursives de ce type par exemple :

$$\begin{array}{lcl} f^1 & = & c_1^1; f^{i_1^1}; c_2^1; f^{i_2^1}; \ldots; f^{i_{j_1}^1} \\ f^2 & = & c_1^2; f^{i_1^2}; c_2^2; f^{i_2^2}; \ldots; f^{i_{j_2}^2} \\ \ldots & & \\ f^k & = & c_1^k; f^{i_1^k}; c_2^k; f^{i_2^k}; \ldots; f^{i_{j_k}^k} \end{array}$$

où les c_j^i sont des suites d'instructions, les f^i sont des fonctions définies mutuellement, qui agissent sur le même environnement (uniquement des variables globales ici).

On peut reprendre comme exemple le calcul de la fonction d'Ackerman : l'entrée est la variable p, le calcul de $ackermann(n,p)$ se fait par $ack_n(p)$, le résultat dans l'environnement est dans la variable q :

$$\begin{array}{lcl} ack_0 & = & (q = p+1) \\ ack_1 & = & \text{if (p==0) } \{p = 1; ack_0\} \\ & & \text{else}\{p = p-1; ack_1; p = q; ack_0\} \\ ack_2 & = & \text{if (p==0) } \{p = 1; ack_1\} \\ & & \text{else}\{p = p-1; ack_2; p = q; ack_1\} \\ \ldots & & \end{array}$$

De façon plus générale, on considère k programmes « à trous »

$$P_1(f_1, \ldots, f_k), \ldots, P_k(f_1, \ldots, f_k)$$

écrits dans notre langage et on cherche $[\![f_1]\!], \ldots, [\![f_k]\!] : \text{Env}_\bot \to \text{Env}_\bot$ tels que (comme la sémantique est *compositionnelle*) :

$$\begin{array}{lcllcl} [\![f_1]\!] & = & [\![P_1(f_1, \ldots, f_k)]\!] & = & [\![P_1]\!]([\![f_1]\!], \ldots, [\![f_k]\!]) \\ \ldots & & & & & \\ [\![f_k]\!] & = & [\![P_k(f_1, \ldots, f_k)]\!] & = & [\![P_k]\!]([\![f_1]\!], \ldots, [\![f_k]\!]) \end{array}$$

Ce qui produit une fonctionnelle (continue !)

$$F : \mathcal{D}^k \to \mathcal{D}^k$$

pour laquelle on applique le théorème de Kleene !

On retrouve bien sûr comme cas particulier la définition récursive d'une boucle `while` :

$$\text{while } b \{ \text{ c } \} \sim \{f = \text{if } b \text{ then c ; f else skip}\}$$

6.5 Continuité et calculabilité

Le fait de pouvoir interpréter des schémas récursifs partiels généraux permet d'imaginer le paradigme suivant : notre sémantique ne donne que des fonctions continues, ce que l'on va prouver un peu plus bas. De façon plus générale, le credo est, *les fonctions calculables sont les fonctions continues.*

Rappelons que pour une fonction $f : \text{Env}_\perp \to \text{Env}_\perp$, être continue veut dire être croissante et pour toute ω-chaîne $\rho_0 \leq \rho_1 \leq \ldots \leq \rho_n \leq \ldots$ on a, pour tout $x \in \text{Var}$:

$$\begin{array}{rcl} \left(\bigcup_{i\in\mathbb{N}} f(\rho_i)\right)(x) & = & \left\{ \begin{array}{ll} f(\rho_j)(x) & \exists j \in \mathbb{N},\, f(\rho_j)(x) \neq \perp \\ \perp & \text{sinon} \end{array} \right. \\ & = & \\ f\left(\bigcup_{i\in\mathbb{N}} \rho_i\right)(x) & = & f\left(y \to \left\{ \begin{array}{ll} \rho_j(y) & \exists j \in \mathbb{N},\, \rho_j(y) \neq \perp \\ \perp & \text{sinon} \end{array} \right. \right)(x) \end{array}$$

La continuité est donc dans ce cas exactement le fait que l'on peut calculer par limite d'approximations finies ; ceci est directement lié aux schémas de définition récursifs.

Prouvons dans un premier temps la croissance de l'interprétation des expressions. Les expressions arithmétiques $a \in \text{AExpr}$ ou booléennes $b \in \text{BExpr}$:

$$\begin{array}{l} [\![a]\!] : \text{Env}_\perp \to \text{Val}_\perp \\ [\![b]\!] : \text{Env}_\perp \to \{true, false\}_\perp \end{array}$$

sont croissantes, par récurrence structurelle sur les expressions : si σ est une extension de ρ, $[\![a]\!]\rho$ est définie ($\neq \perp$) implique $[\![a]\!]\sigma$ définie ; de même pour $[\![b]\!]$. On prouve aisément un peu plus, que $[\![a]\!]$ et $[\![b]\!]$ sont continues.

Venons en maintenant à la croissance de l'interprétation de l'affectation. Soit $\phi \leq \psi \in \text{Env}_\perp$, alors $[\![v = e]\!]\phi = \phi[[\![e]\!]\sigma/v]$, où, pour $u, v \in \text{Var}$ et $n \in \text{Val}$:

$$\phi[n/v](u) = \left\{ \begin{array}{ll} \phi(u) & \text{si } u \neq v \\ n & \text{si } u = v \end{array} \right.$$

Et $[\![v = e]\!]\psi = \psi[[\![e]\!]\sigma/v]$ où, pour $u, v \in \text{Var}$ et $n \in \text{Val}$:

$$\psi[n/v](u) = \left\{ \begin{array}{ll} \psi(u) & \text{si } u \neq v \\ n & \text{si } u = v \end{array} \right.$$

Nécessairement : $\phi[n/v]$ restriction de $\psi[n/v]$ si ϕ restriction de ψ. On prouve également aisément que $[\![v = e]\!]$ est continue. C'est en effet à vérifier pour

chaque variable u : en $u = v$ conséquence de la continuité de $[\![e]\!]$ et en $u \neq v$ de par la continuité de l'identité.

Examinons maintenant la croissance de l'interprétation des tests. Ceci se prouve par récurrence sur les termes. Supposons que $[\![p1]\!]$, $[\![p2]\!]$ sont croissantes ; on considère le terme $i \sim$ if b $p1$ else $p2$:

$$[\![\text{if } b \ p1 \text{ else } p2]\!]\rho = \begin{cases} [\![p1]\!]\rho & \text{si } [\![b]\!]\rho = true \\ [\![p2]\!]\rho & \text{si } [\![b]\!]\rho = false \\ \bot & \text{si } [\![b]\!]\rho = \bot \end{cases}$$

Alors si $\rho \leq \rho'$:
- $[\![b]\!]\rho$ définie égale à $true$ implique $[\![b]\!]\rho'$ définie égale à $true$;
- dans ce cas $[\![i]\!]\rho' = [\![p1]\!]\rho'$ et comme par récurrence $[\![p1]\!]$ est croissante, $[\![p1]\!]\rho'$ est une extension de $[\![p1]\!]\rho = [\![i]\!]\rho$;
- de même quand $[\![b]\!]\rho$ est définie égale à $false$.

De même, ceci est continu. C'est en fait évident, il suffit de séparer le cas $[\![b]\!]\rho = true$ du cas $[\![b]\!]\rho = false$.

Enfin, pour ce qui est de l'interprétation des boucles, on procède par récurrence sur les termes c du langage. On suppose $[\![c]\!] \in \mathcal{D}^0$ (continue), alors pour $\phi \in \mathcal{D}^0$ (continue) :

$$F(\phi)(\rho) = \begin{cases} \phi\,([\![c]\!]\rho) & \text{si } [\![b]\!]\rho = true \\ \rho & \text{si } [\![b]\!]\rho = false \\ \bot & \text{si } [\![b]\!]\rho = \bot \end{cases}$$

est continue (car la composée de fonctions continues est continue).

Exercices

1. Soient D et E deux CPOs et $e \in E$. Soit f la fonction de D vers E qui associe à tout élément de D la constante e. Prouver que f est continue.
2. Soit N le domaine des entiers naturels plus les deux éléments $\top$ et $\bot$ avec l'ordre partiel défini par :
   - Pour tout x, $\bot \leq x$;
   - Pour tout x, $x \leq \top$;
   - Pour tout x, y entiers naturels, on n'a ni $x \leq y$ ni $y \leq x$.

   Prouver que N est un CPO. Est-ce un treillis complet ?
3. Soit f une fonction partielle de $\mathbb{N}$ vers $\mathbb{N}$, son extension de N vers N est la fonction : $f_\bot(\bot) = \bot$, $f_\bot(x) = \bot$ si $f(x)$ est non-définie et $f_\bot(x) = f(x)$ si $f(x)$ est définie. Prouver que $f_\bot$ est continue.
4. Définir un produit de CPOs vérifiant le diagramme définissant les types produits du chapitre 4. Est-ce un CPO ? Quand on part de treillis, et de treillis complets, obtient-on des treillis, et des treillis complets, respectivement ?
5. Montrez que les fonctions de curryfication et l'évaluation définis au chapitre 9, où X, Y et Z sont des CPOs, et $X \to Y$ dénote le CPO des

fonctions continues de X à Y, et $X \times Y$ est le CPO produit défini juste avant :

$$\begin{array}{rc} \mathit{eval} : & (X \to Z) \times X \to Z \\ \mathit{curry} : & ((X \times Y) \to Z) \to (X \to (Y \to Z)) \end{array}$$

sont continues.

6. Calculer le plus petit point fixe, par le théorème de Kleene, de la fonctionnelle associée au programme (par la sémantique dénotationnelle du cours) :

```
Y=1;
while (X>0) {
  Y=Y*X;
  X=X-1;
}
```

7. (++) Essayez de donner une sémantique dénotationnelle de PCF typé, en vous inspirant de la sémantique dénotationnelle du langage impératif de ce chapitre.

Chapitre 7

Logique, modèles et preuve

On en arrive à bientôt pouvoir raisonner (et prouver) sur les programmes, comme sur des objets mathématiques dont vous avez plus l'habitude, grâce à la sémantique du chapitre précédent. Avant cela, il nous faut parler de logique, format naturel dans lequel écrire les preuves. On commence par définir dans ce chapitre quelques concepts élémentaires en logique des prédicats du premier ordre. Cette logique nous sera également utile au chapitre 9 où elle nous permettra de présenter l'isomorphisme de Curry-Howard.

La logique a été créée dans un effort de formalisation des mathématiques, si l'on ignore comme ici sa partie philosophique. En particulier, les axiomes de l'arithmétique de Péano, la théorie des ensembles de Zermelo-Fraenkel, sont des théories exprimables dans la logique des prédicats du premier ordre.

7.1 Syntaxe de la logique des prédicats du premier ordre

Celle-ci est définie syntaxiquement comme suit :

- Elle comprend des opérateurs binaires (infixe) $\wedge$ (et), $\vee$ (ou), $\Rightarrow$ (implication), $\Leftrightarrow$ (équivalence), unaire $\neg$ (négation), 0-aire (constantes) 1 (vrai), 0 (faux) et un ensemble de variables infini
- Les quantificateurs : $\forall$ (pour tout), $\exists$ (il existe), des prédicats *de base*, d'arité variable, $P(x)$, $Q(x, y)$, $x \leq y$, $x = y$ etc. et des fonctions d'arité variable également, $f(x)$, $g(x, y)$, x^2, $x - y$ etc.

On appelle *termes* de la logique des prédicats les éléments de syntaxe formés à partir des variables, et inductivement, par application répétée de fonctions. Autrement dit, une variable x est un terme, et si $t_1, \ldots, t_n$ sont des termes, et f une fonction n-aire, alors $f(t_1, \ldots, t_n)$ est un terme. On appelle *formule* en logique des prédicats les éléments de syntaxe définis comme suit :

- $P(t_1, \ldots, t_n)$ est une formule quand P est un prédicat n-aire, et $t_1, \ldots, t_n$ sont des termes
- $\neg\Phi$ est une formule quand Φ est une formule

- $\Phi \wedge \Psi$, $\Phi \vee \Psi$, $\Phi \Rightarrow \Psi$, $\Phi \Leftrightarrow \Psi$ sont des formules quand Φ et Ψ sont des formules
- $\forall x.\Phi$ et $\exists x.\Phi$ sont des formules quand Φ est une formule

Dans une formule, on dit qu'une variable est *libre* quand elle n'est pas quantifiée. A contrario, une variable est *liée* quand elle est quantifiée. Par exemple, dans la formule $\forall x.P(x, y, z)$ x est liée, y et z sont libres.

7.2 Sémantique de la logique des prédicats du premier ordre

On peut définir une interprétation des termes, inductivement, exactement comme en sémantique dénotationnelle de langages de programmation, chapitre 6.

On se donne pour ce faire un *modèle*, ou *structure du premier ordre* D (un ensemble de « valeurs » ou de « dénotations »). Par exemple, on pourra prendre $\mathbb{R}$ comme modèle, pour interpréter des prédicats « parlant » d'ordres totaux $\leq$. A chaque symbole f d'arité n, on associe $[\![f]\!] : D^n \to D$ (par convention, pour les constantes, d'arité 0, $[\![f]\!] \in D$). De même, à chaque prédicat P d'arité n on associe une *fonction caractéristique* $\chi_P : D^n \to \{0, 1\}$. L'idée est que l'ensemble des valeurs de D^n, dans cette interprétation, telles que P est vraie, est $\chi_p^{-1}(1)$. Dans le cas évoqué plus haut, $D = \mathbb{R}$, et on pourra interpréter le prédicat d'arité $2 \leq$ que l'on se serait donné dans notre structure du premier ordre, par l'ordre total standard sur $\mathbb{R}$.

Etant donnés un modèle D et une interprétation (on pourrait l'appeler également « sémantique », mais il est plus classique dans ce domaine de l'appeler « interprétation ») $[\![.]\!]$, on doit aussi interpréter les variables x qui prennent des valeurs dans D, il nous faut donc une notion d'environnement, comme pour la sémantique des langages de programmation. Un environnement est ici une fonction $\rho : \mathrm{Var} \to D$.

L'évaluation des termes de la logique des prédicats se fait sans surprise :

$$\begin{array}{lcl} [\![x]\!]\rho & = & \rho(x) \\ [\![f(t_1, \ldots, t_n)]\!]\rho & = & [\![f]\!]([\![t_1]\!]\rho, \ldots, [\![t_n]\!]\rho) \end{array}$$

Pour les formules F de la logique des prédicats, $[\![F]\!]\rho$ va avoir une valeur dans $\{0, 1\}$:

$$\begin{array}{lcl} [\![\Phi \wedge \Psi]\!]\rho & = & ([\![\Phi]\!]\rho) * ([\![\Psi]\!]\rho) \\ [\![\neg\Phi]\!]\rho & = & 1 - [\![\Phi]\!]\rho \end{array}$$

On n'a pas besoin d'en dire plus, grâce aux lois de Morgan ($A \vee B = \neg((\neg A) \wedge (\neg B))$, $A \Rightarrow B = (\neg A) \vee B$ etc.), valides en logique propositionnelle (fragment de la logique des prédicats du premier ordre).

L'évaluation des formules se fait comme suit.

- $[\![P(t_1, \ldots, t_n)]\!]\rho = \chi_P([\![t_1]\!]\rho, \ldots, [\![t_n]\!]\rho)$

– Quantificateurs :

$$\begin{aligned}
[\![\forall x.\Phi]\!]\rho &= \begin{cases} 1 & \text{si } \forall \rho' \in \text{Env tq } \rho'(y) = \rho(y)\ \forall y \neq x \in \text{Var}, [\![\Phi]\!]\rho' = 1 \\ 0 & \text{sinon} \end{cases} \\
[\![\exists x.\Phi]\!]\rho &= \begin{cases} 1 & \text{si } \exists \rho' \in \text{Env tq } \rho'(y) = \rho(y)\ \forall y \neq x \in \text{Var}, [\![\Phi]\!]\rho' = 1 \\ 0 & \text{sinon} \end{cases}
\end{aligned}$$

Remarquez que l'égalité joue toujours un rôle particulier parmi les prédicats, son interprétation n'est pas « libre ». Si elle fait partie des prédicats d'une théorie, alors elle est toujours interprétée par l'égalité dans D :

$$[\![t_1 = t_2]\!]\rho = \begin{cases} 1 & \text{si } [\![t_1]\!]\rho = [\![t_2]\!]\rho \\ 0 & \text{sinon} \end{cases}$$

Une notion très importante est celle de la *satisfiabilité* d'une formule de logique des prédicats. Soit M une interprétation (domaine D, fonction sémantique $[\![.]\!]$, environnement ρ) et Φ une formule, alors on dit que M satisfait Φ, ou $M \models \Phi$ si $[\![\Phi]\!]\rho = 1$. Cela n'a à vrai dire réellement de sens que pour les formules Φ closes (c'est-à-dire toutes ses variables sont liées), même si on peut définir une relation de satisfiabilité générale, qui ne nous servira pas ici. On appelle *tautologie*, une formule qui est vraie dans *toutes* les interprétations.

Une théorie (du premier ordre) est un ensemble d'*axiomes*, c'est-à-dire de formules du premier ordre avec une certaine signature, formules que l'on suppose être vraies. D'une certaine façon, les axiomes définissent en termes logiques une structure, qui vérifie un ensemble de « contraintes ».

Par exemple, la théorie des groupes peut être axiomatisée en logique des prédicats en supposant la signature suivante. Celle-ci inclut des fonctions : $*$ (l'opération de groupe), $^{-1}$ (l'inversion) et 1 (l'unité du groupe). Elle inclut aussi un seul prédicat : $=$ (égalité). Les *axiomes* de la théorie des groupes, c'est-à-dire les formules définissants les groupes, « au premier ordre » sont :

$$\begin{aligned}
&\forall x.x * 1 = x \\
&\forall x.1 * x = x \\
&\forall x.x * x^{-1} = 1 \\
&\forall x.x^{-1} * x = 1 \\
&\forall x, y, z.x * (y * z) = (x * y) * z
\end{aligned}$$

La logique est dite du premier ordre car les quantificateurs ne s'appliquent qu'à des variables (simples), on ne peut par exemple, en logique du premier ordre, quantifier sur des ensembles dans lesquelles les variables pourraient évoluer. Plutôt qu'à spécifier des structures mathématiques, la logique des prédicats va surtout nous servir par la suite pour formaliser les propriétés de programmes (validation, chapitre 8).

En général on suppose que l'ensemble d'axiomes est fini ou récursivement énumérable (en fait, cela a une conséquence profonde en théorie des modèles, sinon a priori, cela semble être un prérequis raisonnable).

Les modèles, ou les sémantiques – car il peut y en avoir de nombreuses, dépendant de l'ensemble sous-jacent D – et les théories entretiennent des rapports

complexes. A tout modèle (ex. $\mathbb{R}$), on peut associer, étant donné une signature (ex. pour $\mathbb{R}$, $\{=, \times, +, -, /, 0, 1...\}$), sa théorie du premier ordre, c-.à-.d. l'ensemble de toutes les formules avec cette signature, que satisfait le modèle. Inversement, à toute théorie, on peut associer l'ensemble des modèles qui satisfont à cette théorie. Il n'y a pas néanmoins généralement de relation bijective entre modèles et théories, en logique du premier ordre. Une théorie un tant soit peu intéressante ne suffit généralement pas à décrire de façon unique le modèle que l'on voulait axiomatiser. Par exemple, la théorie des nombres réels au premier ordre, quelle que soit la signature raisonnable choisie pour les prédicats définit des modèles parfois bien différents des nombres réels construits par coupure de Dedekind. Ils sont appelés les réels non standards, et ont parfois un intérêt pour faire du calcul infinitésimal (analyse non-standard). Il existe même des modèles dénombrables de la théorie des ensembles de Zermelo-Fraenkel !

Pour ceux qui voudraient en savoir plus, tout ceci est développé en INF423 [3] (en particulier les théorèmes de compacité et de Lowenheim-Skolem en théorie des modèles).

7.3 Décidabilité des formules logiques et problème de l'arrêt

Revenons brièvement aux propriétés de calculabilité définies au chapitre 5. On va voir que certains prédicats, sur les entiers – on se restreint donc ici à $D = \mathbb{N}$ – sont calculables en un certain sens, c'est-à-dire que l'on peut déterminer si ils sont satisfiables ou non, algorithmiquement, alors que d'autres, pas. Dans cette dernière catégorie, on va trouver des propriétés particulièrement utiles à la validation de programmes, hélas, voir le chapitre 8.

Soit F une formule de la logique des prédicats du premier ordre. On choisit comme domaine d'interprétation $D = \mathbb{N}$. On dit que F est *décidable* si χ_F est dans R (récursive partielle).

Construisons maintenant un prédicat particulier, sur les programmes Java, que l'on voit comme des entiers naturels, grâce à un codage, que l'on ne donne pas précisément, mais dont l'existence est évidente (l'ensemble des programmes Java est dénombrable, car les programmes sont finis, écrits sur un alphabet fini).

On peut donc coder tout programme Java `J` en un entier naturel que l'on note $[J]$, que l'on peut même calculer de façon très algorithmique. On considère maintenant le prédicat sur $\mathbb{N}$, $P(n) =$« le programme de numéro n termine ».

Ce prédicat est indécidable. C'est-à-dire qu'il n'existe pas d'algorithme qui étant donné un programme, réponde en temps fini si ce programme termine ou pas.

Donnons-en ici quelques éléments de preuve. Elle procède par l'absurde : supposons qu'il existe un algorithme `A` qui prenne en argument un programme `J` de numéro x prenant en argument un entier, et un entier n et renvoyant `true` si `J(n)` termine, `false` sinon. Considérons maintenant le programme `K` suivant :

```
K(x) {
```

```
  if A(x,x)
    while (true) {}
}
```

Quelle est alors la valeur de `K([K])` ? De deux choses l'une : si `K` termine sur `[K]` alors `A([K],[K])` est vrai, donc `K([K])` fait `tant que (true) { }` et ne termine pas, contradiction. Ou alors, si `K` ne termine pas sur `[K]` alors `A([K])` est faux donc `K([K])` termine, contradiction encore une fois !

Remarque : il s'agit d'un argument dit de la « diagonale de Cantor », ou plus simplement, argument diagonal. Le problème que l'on vient de considérer s'appelle le « problème de l'arrêt ».

7.4 Pour aller plus loin...

Il existe un corpus très imposant de résultats fondateurs en décidabilité et indécidabilité de théories. Par exemple, l'arithmétique de *Péano* (l'arithmétique que vous connaissez) est indécidable. Par contre, la théorie de *Presburger*, décrivant une arithmétique de nombres naturels, plus faible, est décidable. On reporte le lecteur à [3] pour en apprendre plus.

7.5 Un peu de théorie de la démonstration

La théorie de la démonstration est une branche des mathématiques et de la logique qui se préoccupe de savoir non pas quand une formule est « vraie » (dans un modèle par exemple, c'est le problème de satisfiabilité traité auparavant) mais plutôt si une formule, dans un système formel, est prouvable, et de construire une preuve.

Il y a plusieurs manières de formaliser les preuves (comme les preuves mathématiques que vous faites au quotidien, sauf que celles-ci sont dans un format relativement informel, en langage naturel, et ne sont donc pas automatisables directement).

Le premier formalisme est celui de la déduction naturelle (Gentzen 1934), pour présenter des preuves en logique des prédicats du 1^{er} ordre.

On va ici se contenter de parler de théorie de la démonstration dans le cadre de la logique classique du premier ordre, c'est-à-dire de la logique propositionnelle quantifiée, qui est la logique des prédicats du premier ordre, mais où l'on n'a aucune fonction, et, à la place des prédicats généraux, des simples variables logiques (booléennes).

Dans un premier temps, définissons la notion de *règle d'inférence* R. Etant donné les preuves des propositions $p_1, \ldots, p_n$, « on prouve q » (en une étape) se note :

$$(R) : \frac{p_1 \; p_2 \; \cdots \; p_n}{q}$$

ou encore : « si on a une preuve de p_1, de p_2,..., de p_n, alors on a une preuve de q en utilisant l'inférence R ». Un système formel en déduction naturelle est la donnée de règles écrites dans ce format.

Ainsi, une preuve en déduction naturelle est un arbre construit à partir de telles règles, comme on va le montrer à partir d'un exemple un peu plus loin.

Voici donc le système de preuve présenté sous format « déduction naturelle » de la logique propositionnelle quantifiée du 1$^{\text{er}}$ ordre.

Il y a tout d'abord les *règles d'introduction*, nommées ainsi car elle permettent, à partir de la preuve de sous-formules d'une formule p, d'en déduire la preuve de p, construite à partir d'un connecteur logique (et, ou, implique, etc.) et de ces sous-formules. On « introduit » donc en quelque sorte ces connecteurs logiques.

L'introduction pour le « et » :

$$(\wedge I)\ \frac{p\ q}{p \wedge q}$$

Pour le « ou », à gauche :

$$(\vee I_g)\ \frac{p}{p \vee q}$$

Pour le « ou », à droite :

$$(\vee I_d)\ \frac{q}{p \vee q}$$

Pour l'implication :

$$(\Rightarrow I)\ \begin{array}{c}[p]\\ \vdots\\ \frac{q}{p \Rightarrow q}\end{array}$$

La notion $[p]$ demande une explication supplémentaire : on dit que l'on *décharge* l'hypothèse p. Ceci se lit naïvement « si on a prouvé p, et que à partir de cette preuve on peut prouver q, alors on peut prouver $p \Rightarrow q$ ».

Pour la quantification universelle :

$$(\forall I)\ \frac{p}{\forall x.p}$$

($(\forall I)$ valide seulement si x n'apparaît dans aucune des hypothèses [non déchargées]).

Pour la quantification existentielle :

$$(\exists I)\ \frac{p[a/x]}{\exists x.p}$$

On trouve ensuite les règles d'élimination : ce sont les règles inverses en quelque sorte. Si on a une preuve d'une formule composée de sous-formules, on veut en déduire la preuve d'une de ces sous-formules :

Pour le « et », à gauche, et à droite :

$$(\wedge E_g)\ \frac{p \wedge q}{p} \quad (\wedge E_d)\ \frac{p \wedge q}{q}$$

Pour l'implication :

$$(\Rightarrow E)\ \frac{p\ p \Rightarrow q}{q}$$

Pour le « ou », c'est un peu subtil :

$$(\vee E)\ \frac{p \vee q \qquad \begin{array}{c}[p]\\ \vdots\\ r\end{array} \qquad \begin{array}{c}[q]\\ \vdots\\ r\end{array}}{r}$$

Ce qui veut dire que si, étant donné une preuve (que l'on n'a pas) de p, on peut prouver r, et une preuve (que l'on n'a pas) de q, on peut prouver r, alors si on a une preuve de $p \vee q$, on a une preuve de r.

Pour la quantification universelle :

$$(\forall E)\ \frac{\forall x.p}{p[a/x]}$$

(a est n'importe quelle formule, et $[a/x]$ dénote comme toujours la substitution de la variable x par a).

Pour la quantification existentielle :

$$(\exists E)\ \frac{\exists x.p \qquad \begin{array}{c}[p]\\ \vdots\\ q\end{array}}{q}$$

Enfin, il y a des règles spécifiques liées à F (faux) :

$$(F)\ \frac{F}{p} \qquad\qquad (RPA)\ \frac{\begin{array}{c}[\neg p]\\ \vdots\\ F\end{array}}{p}$$

La première règle dit que de faux, on peut déduire ce que l'on veut. La dernière règle est la réduction par l'absurde.

Voici maintenant un exemple de preuve en déduction naturelle. On prouve ici que $p \wedge q \Rightarrow q \wedge p$ (qui est bien une formule vraie en logique propositionnelle). L'arbre de preuve est ainsi construit :

$$(\Rightarrow I)\ \frac{(\wedge I)\ \dfrac{(\wedge E_d)\ \dfrac{[p \wedge q]}{q} \qquad (\wedge E_g)\ \dfrac{[p \wedge q]}{p}}{q \wedge p}}{p \wedge q \Rightarrow q \wedge p}$$

Remarquez que les hypothèses sont déchargées par la règle d'introduction de l'implication ($\Rightarrow I$).

Un autre format, dont il est plus difficile à comprendre l'intérêt et la relative complexité par rapport à la déduction naturelle, est le calcul des séquents. Ce calcul a été introduit par Gentzen en 1936, après la déduction naturelle donc, pour obtenir une formulation plus symétrique des règles de preuve, et pour éviter la notion de preuve « déchargée ».

Le format des règles est comme en déduction naturelle :

$$\frac{\text{prémisses}}{\text{conclusion}}$$

Les prémisses et conclusion sont constitués de *jugements de preuves*.

$$\Gamma \vdash \Delta$$

où Γ et Δ sont des suites de formules logiques. En fait, séquent est une « mauvaise » traduction de l'allemand, et aurait sans doute du être traduit par « séquence » ou « suite ». Ce format de règle se lit informellement de la façon suivante : « en supposant toutes les formules de Γ prouvées, on peut prouver la disjonction de toutes les formules de Δ ».

On distingue traditionnellement plusieurs groupes de règles en calcul des séquents :

« Groupe identité » :

- On a la règle « axiome » ; d'une preuve de A je peux construire une preuve de A :

$$(ax)\ \frac{}{A \vdash A}$$

- La règle de coupure, fondamentale dans la relation de la théorie de la preuve avec l'informatique (isomorphisme de Curry-Howard, chapitre 9) :

$$(cut)\ \frac{\Gamma \vdash A, \Delta \quad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}$$

On peut d'une certaine façon éliminer le besoin d'une preuve de A pour prouver les séquents de Δ' si on a par ailleurs une preuve de A.

« Groupe structurel » :

- On a l'« affaiblissement à gauche » :

$$(weak_g)\ \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}$$

C'est-à-dire que l'on peut rajouter une hypothèse (A) et toujours à arriver à prouver Δ à partir du séquent Γ, A, si on a pu le prouver à partir de Γ.

- De même on a l'« affaiblissement à droite » :

$$(weak_d)\ \frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}$$

- On peut aussi « contracter » les séquents, si on a plusieurs copies de preuves :

$$(contr_g)\ \frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}$$

- Remarquez la symétrie (autour du symbole $\vdash$) des règles :

$$(contr_d)\ \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}$$

- On a également les règles d'échange :

$$(ex_g)\ \frac{\Gamma \vdash \Delta}{\sigma(\Gamma) \vdash \Delta}$$

$$(ex_d)\ \frac{\Gamma \vdash \Delta}{\Gamma \vdash \sigma(\Delta)}$$

où σ est une permutation agissant sur les séquents (l'ordre dans lequel on a listé les preuves).

« Groupe logique » :

- On a une règle d'introduction à droite de l'implication :

$$(\Rightarrow I_d)\ \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash (A \Rightarrow B), \Delta}$$

- De même, à gauche :

$$(\Rightarrow I_g)\ \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma, A \Rightarrow B \vdash \Delta}$$

- On a l'introduction à gauche du « et » :

$$(\wedge I_g)\ \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}$$

- Puis l'introduction à droite du « et » (toujours cette symétrie !) :

$$(\wedge I_d)\ \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}$$

Et d'autres règles encore...pour les autres connecteurs logiques. L'objectif de ce cours n'est pas d'être complet de ce côté, mais de donner le minimum de concepts logiques pour comprendre le problème de la validation de programme traité au chapitre 8, et le lien entre preuve et exécution (théorie du typage, isomorphisme de Curry-Howard), traité au chapitre 9. Normalement, ces sujets font l'objet d'un ou plusieurs cours de deuxième année de Master.

Voici maintenant un exemple de preuve en calcul des séquents, pour la même formule que l'on avait prouvée en déduction naturelle, c'est-à-dire $p \wedge q \Rightarrow q \wedge p$. On construit encore un arbre de preuve :

$$(\wedge I_d)\ \frac{(\wedge I_g)\ \dfrac{(ex_g)\ \dfrac{(w_g)\ \dfrac{(ax)\ \overline{q \vdash q}}{q, p \vdash q}}{p, q \vdash q}}{p \wedge q \vdash q} \qquad (\wedge I_g)\ \dfrac{(w_g)\ \dfrac{(ax)\ \overline{p \vdash p}}{p, q \vdash p}}{p \wedge q \vdash p}}{p \wedge q \vdash q \wedge p}$$

Notre explication de la théorie de la preuve a été assez informelle malgré tout ; il nous faudrait pour être complet, donner une « sémantique » des règles d'inférences introduites, que ce soit en déduction naturelle ou en calcul des séquents. Ceci n'est pas sans intérêt, même sans rentrer dans les détails, car cette sémantique ressemble tout à fait à celle développée pour les langages de programmation, au chapitre 6, tout comme la façon d'exprimer la satisfiabilité d'une formule de la logique des prédicats du premier ordre ressemblait à s'y méprendre à la sémantique (dénotationnelle) d'un langage de programmation.

Donc sans rentrer trop dans les détails, soit $\mathcal{P}$ l'ensemble des formules logiques que l'on peut écrire dans notre logique propositionnelle quantifiée du 1$^{\text{er}}$ ordre. Chaque règle d'inférence R définit une fonction $F_R : \mathcal{P} \to \mathcal{P}$ (de production de nouvelles formules vraies, et enlève les formules déchargées dans le cas de la déduction naturelle). L'ensemble des propositions prouvables par le système formel décrit en déduction naturelle/calcul des séquents est le plus petit ensemble invariant par l'application des F_R, R règle d'inférence. Il s'agit donc du calcul du plus petit point fixe d'une fonctionnelle sur les ensembles, comme pour la sémantique de la boucle `while` (voir chapitre 6). Dit de façon plus simple, il s'agit du calcul d'une clôture transitive de l'application des règles (« arbres de preuve »), ce qui donne le calcul effectif de ce plus petit point fixe par le théorème de Kleene.

Terminons ce chapitre en disant quelque mots du rapport entre la satisfiabilité et la preuve en logique propositionnelle quantifiée du premier ordre.

Notons $\vdash p$ si p est prouvable en logique propositionnelle quantifiée (par le système de déduction naturelle précédent par exemple), et $M \models p$ si p est satisfiable dans le modèle M de la théorie de la logique propositionnelle quantifiée du 1$^{\text{er}}$ ordre et $\models p$ si p est satisfiable dans *tous les modèles* M.

On a alors les faits suivants.

Le premier s'appelle la « correction » du système de preuve : si $\vdash p$ alors $\models p$. Ceci est vrai ici : si p est prouvable en déduction naturelle ou calcul des séquents, alors p est « vrai », ce qui est plutôt rassurant. En fait, c'est toujours le cas, sinon c'est un grave problème du système de preuve.

L'autre propriété potentiellement intéressante est la complétude : si $\models p$ alors $\vdash p$; c'est-à-dire que si p est vraie, elle est prouvable dans notre système formel. La encore, cela est vrai pour la logique propositionnelle du premier ordre, et notre calcul des séquents (prouvé dans la thèse de Gödel en 1929).

Mais d'une certaine façon c'est plutôt rare, cela n'est déjà plus vrai pour le calcul des *prédicats* du premier ordre, dès que l'on s'autorise des prédicats et une axiomatique un peu utiles. Ceci est lié au 2$^{\text{e}}$ problème de Hilbert (« mécanisation » de l'arithmétique) de 1900. On a par exemple l'incomplétude de l'arithmétique de Péano en calcul des prédicats du premier ordre (Gödel, encore).

Exercices

1. Prouver $p \vee \neg p$ en déduction naturelle ou en calcul des séquents.

2. Prouver la loi de Pierce : $((p \Rightarrow q) \Rightarrow p) \Rightarrow p$, toujours en déduction naturelle ou en calcul des séquents.

Chapitre 8

Validation et preuve de programmes

La validation des programmes, et des systèmes, est une activité essentielle du développement logiciel, et de systèmes de façon générale. Cette activité est généralement faite soit par des méthodes dites « formelles », soit par des techniques plus ad-hoc : relecture du code, batteries de tests (même s'il existe une théorie générale de la « couverture » des tests, et de leur génération automatique) etc. Dans ce chapitre, nous évoquons une manière de valider formellement des programmes séquentiels, dans un formalisme logique, appelée logique de Hoare, du nom de C.A.R. Hoare, prix Turing 1980. Il existe également d'autres moyens de prouver des programmes, de façon plus automatique, comme l'interprétation abstraite et le model-checking. Le lecteur intéressé pourra se reporter aux cours [10] et [8].

8.1 La validation, pour quoi faire ?

Prenons un petit exemple de programme, du bon fonctionnement duquel nous voudrions nous assurer. Il s'agit d'un code de transformée de Fourier rapide (dont on n'a pas indiqué complètement certains points, dont certaines valeurs de constantes, mais cela n'est pas important pour la preuve que l'on vise) :

```
fft(complex_array_ref a,int n)
 { complex_array_ref b[n/2], c[n/2];
   if (n > 2)
    { for (i=0 ; i < n ; i=i+2)
       { b[i/2] = a[i];
         c[i/2] = a[i+1]; }
      fft(b,n/2);
      fft(c,n/2);
      for (i=0 ; i < n ; i=i+1)
       a[i] = F1(n)*b[i] + F2(n)*c[i];
```

```
} else { ...
```

On souhaiterait pouvoir prouver que ce programme ne comporte pas de *bug* à l'exécution (division par zéro, accès à des tableaux en dehors de leurs bornes, dépassement de valeurs pour les types considérés etc.). On souhaiterait également prouver des propriétés plus fines, plus « fonctionnelles », par exemple vérifier l'égalité de Parseval (à la précision finie près) :

$$\sum_i \mid a'[i] \mid^2 = \sum_i \mid a[i] \mid^2$$

Pour ce faire, on souhaite entrelacer le code avec des commentaires qui décrivent, en utilisant la logique des prédicats du premier ordre, des propriétés que l'on arrive à prouver pour toutes les exécutions du programme, passant par cette ligne. On appelle cela des annotations de preuve.

Progressivement, on annote de la première ligne aux lignes suivantes :

```
1  fft(a,n)
2  // a.length=n ∧ ∃k > 0 n=2^k
3   { cplx b[n/2], c[n/2];
4     // a.length=n ∧ ∃k > 0 n=2^k ∧ b.length=n/2 ∧ c.length=n/2
5     if (n > 2)
6      { for (i=0;i<n;i=i+2)
7          { // a.length=n ∧ ∃k > 0 n=2^k ∧ b.length=n/2 ∧ c.length=n/2 ∧ i≥0 ∧ i<n
   ∧ ∃j ≥ 0 i=2j
8            b[i/2]=a[i];
9            // i+1<n ?
10           c[i/2]=a[i+1]; }
11     ...
```

On arrive facilement à prouver l'annotation en ligne 9, c'est-à-dire qu'il n'y a pas d'accès en dehors du tableau. En effet, $i + 1 = 2j + 1 \leq n$ et $n = 2^k$ impliquent $i + 1 < n$.

Un peu plus loin dans le code :

```
fft(a,n)
// a.length=n ∧ ∃k > 0 n=2^k
 { cplx b[n/2], c[n/2];
   // a.length=n ∧ ∃k > 0 n=2^k ∧ b.length=n/2 ∧ c.length=n/2
   if (n > 2)
    { for (i=0;i<n;i=i+2)
        { // a.length=n ∧ ∃k > 0 n=2^k ∧ b.length=n/2 ∧ c.length=n/2 ∧ i≥0 ∧ i<n
   ∧ ∃j ≥ 0 i=2j
          b[i/2]=a[i];
          // i+1<n
          c[i/2]=a[i+1]; }
      fft(b,n/2);
      fft(c,n/2);
      for (i=0;i<n;i=i+1)
       a[i]=F1(n)*b[i]+F2(n)*c[i];}
```

```
else
 { // a.length=2
   a[0]=g*a[0]+d*a[1];
   a[1]=a[0]-2*d*a[1];  } }
```

On arrive encore à prouver que l'on termine bien l'appel récursif avec `a.length=2`, car $n = 2^l$, $0 \leq l \leq k$ et $n \leq 2$ impliquent $a.length = n = 2$. Comment formaliser et automatiser cela ?

8.2 Preuve à la Hoare

Définissons dans un premier temps un langage d'assertion. Les expressions arithmétiques $a \in Aexprv$ sont de la forme :

$$a ::= n \mid X \mid i \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

Pour les expressions booléennes $Assn$:

$$\begin{array}{l} A ::= true \mid false \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \\ A_0 \vee A_1 \mid A_0 \wedge A_1 \mid \neg A \mid A_0 \implies A_1 \mid \forall i\ A \mid \exists i\ A \end{array}$$

Dans cette grammaire, on a deux types de variables : les variables des programmes que l'on va analyser (`X`$\in$ Var) et des variables supplémentaires, utilisées uniquement pour le raisonnement logique (`i`$\in IntVar$). Par exemple, pour la validation de la FFT à la section 8.1, on a utilisé le fait que la taille du tableau est une puissance de 2, et pour ce faire on a du introduire une variable k, pour écrire $n = 2^k$. Cette variable (k) n'est pas une variable du programme.

Une interprétation de ces formules est une fonction $I : IntVar \to \mathbb{Z}$, où $IntVar$ est l'ensemble des variables entières spécifiques aux assertions. La sémantique des assertions dépend de I et de l'environnement courant σ :

$$\begin{array}{lcl} [\![n]\!](I,\sigma) & = & n \\ [\![X]\!](I,\sigma) & = & \sigma(X) \\ [\![i]\!](I,\sigma) & = & I(i) \\ [\![a_0 + a_1]\!](I,\sigma) & = & [\![a_0]\!](I,\sigma) + [\![a_1]\!](I,\sigma) \end{array}$$

On définit maintenant par induction structurelle le jugement de preuve $\sigma \models^I A$, voulant dire : « pour $\sigma \in \Sigma$, σ satisfait A dans l'interprétation I » :

- $\sigma \models^I true$
- $\sigma \models^I (a_0 = a_1)$ si $[\![a_0]\!](I,\sigma) = [\![a_1]\!](I,\sigma)$
- $\sigma \models^I A \wedge B$ si $\sigma \models^I A$ et $\sigma \models^I B$
- $\sigma \models^I A \Rightarrow B$ si (non $\sigma \models^I A$) ou $\sigma \models^I B$
- $\sigma \models^I \forall i.A$ si $\sigma \models^{I[n/i]} A$ pour tout $n \in \mathbb{Z}$
- etc.

On souhaite maintenant donner un sens à des assertions (dites de correction partielle), appelées respectivement pré- et post-conditions, A et B, écrites dans le langage d'assertion $Assn$ défini plus haut, que l'on rajoute autour du code c, et que l'on écrit $\{A\}c\{B\}$. Cette dernière se lit, « Si A est vraie alors après

exécution de c, on a B qui est vrai ». On appelle parfois $\{A\}c\{B\}$ « triplet de Hoare ».

On définit une relation de satisfaction entre les environnements et les assertions de correction partielle (invariants), pour une interprétation I, comme suit :

$$\sigma \models^I \{A\}c\{B\} \; ssi \; \left(\sigma \models^I A \Rightarrow [\![c]\!]\sigma \models^I B\right)$$

On écrit $\models^I \{A\}c\{B\}$ si pour tout $\sigma \in \Sigma$, $\sigma \models^I \{A\}c\{B\}$.

Malgré tout, $\models^I$ n'est pas tout à fait ce que l'on voulait ; par exemple :

$$\{i < X\}X = X + 1\{i < X\}$$

dépend de i et de l'interprétation de i, à laquelle on ne s'intéresse pas du tout.

Une solution est de définir le jugement $\models \{A\}c\{B\}$ si pour toutes les interprétations I, $\models^I \{A\}c\{B\}$. On souhaite maintenant déterminer A et B de façon systématique, pour toutes les instructions c, telles que $\models \{A\}c\{B\}$.

On ne veut pas utiliser la définition de la validité des formules pour les triplets à la Hoare, reposant sur le test exhaustif de la validité des formules pour chaque environnement (satisfiabilité en *théorie des modèles*). On veut avoir un calcul d'*inférences logiques* permettant de prouver la validité d'un triplet de Hoare (*théorie de la démonstration*, voir la section 7.5).

On définit maintenant des règles de preuve de correction partielle. Les voici, une par une ; elles seront prouvées et expliquées un peu après (avec un exemple) :

– (skip)
$$\{A\}skip\{A\}$$

– (assign)
$$\{B[a/X]\}X = a\{B\}$$
Cela veut dire que pour que B soit vrai après l'affectation de a à la variable X, il faut que B où l'on a syntaxiquement substitué l'expression a à toutes les occurrences de X, soit vraie.

– (seq)
$$\frac{\{A\}c_0\{C\} \quad \{C\}c_1\{B\}}{\{A\}c_0; c_1\{B\}}$$

– (if)
$$\frac{\{A \wedge b\}c_0\{B\} \quad \{A \wedge \neg b\}c_1\{B\}}{\{A\}if \; b \; then \; c_0 \; else \; c_1\{B\}}$$

– (while)
$$\frac{\{A \wedge b\}c\{A\}}{\{A\}while \; b \; do \; c\{A \wedge \neg b\}}$$

– (conseq.)
$$\frac{\models (A \Rightarrow A') \quad \{A'\}c\{B'\} \quad \models (B' \Rightarrow B)}{\{A\}c\{B\}}$$
Le symbole $\models$ est ici celui de la section 7.2 : il dénote la satisfiabilité en logique des prédicats de la formule qui est à droite (étant donné des prémisses, à gauche de ce symbole).

Il y a deux propriétés importantes à examiner sur un système formel comme celui-ci, en rapport avec le programme analysé.

La première propriété est la correction de ce système formel : chaque assertion de correction partielle dérivée à partir du système formel précédent doit être vraie, c'est-à-dire cohérente avec la sémantique dénotationnelle du chapitre 6. Autrement dit :

$$\vdash \{A\}p\{B\} \Rightarrow \models \{A\}p\{B\}$$

Dans la formule plus haut $\vdash \{A\}p\{B\}$ veut dire que l'on peut prouver $\{A\}p\{B\}$ à partir du système formel donné plus haut, au sens de la théorie de la démonstration (section 7.5). Le symbole $\models$ à droite dans cette même formule dénote la satisfaction de la formule correspondante, définie, par sémantique dénotationnelle, au début de cette section.

La deuxième propriété qui pourrait être souhaitable est la complétude : toutes les assertions de correction partielle vraies dans notre sémantique dénotationnelle doivent être dérivables dans notre système formel de règles de correction partielle, c'est-à-dire, formellement :

$$\models \{A\}p\{B\} \Rightarrow \vdash \{A\}p\{B\}$$

On a alors le théorème suivant :

Théorème 3. *La logique de Hoare est correcte, mais pas complète.*

Preuve. Correction de (assign). On rappelle que la règle est : (assign) $\{B[a/X]\}X = a\{B\}$. On veut donc prouver que quelque soient $\sigma \in \text{Env}$, $I : IntVar \to \mathbb{Z}$,

$$\sigma \models^I B[a/X] \Rightarrow [\![X = a]\!]\sigma \models^I B$$

Cela se fait par induction structurelle sur B. Par exemple :

- $B = C \wedge D$, alors $\sigma \models^I B[a/X]$ est équivalent à $\sigma \models^I C[a/X]$ et $\sigma \models^I D[a/X]$. Par récurrence on sait que

$$\sigma \models^I C[a/X] \Rightarrow [\![X = a]\!]\sigma \models^I C$$
$$\sigma \models^I D[a/X] \Rightarrow [\![X = a]\!]\sigma \models^I D$$

- $B = (a_0 = a_1)$, par récurrence sur les expressions a_0, a_1. Juste un exemple : $a_0 = X$, $a_1 = n$ (constante), alors $B[a/X] = (a = n)$ et bien sûr $\sigma \models^I (a = n)$ implique $[\![a]\!]\sigma = n$ donc $[\![X = a]\!]\sigma(X) = [\![a]\!]\sigma = n$ d'où $[\![X = a]\!]\sigma \models^I (X = n)$.

Correction de la séquence. On rappelle la règle : (seq) $\frac{\{A\}c_0\{C\} \quad \{C\}c_1\{B\}}{\{A\}c_0;c_1\{B\}}$. La preuve est alors une conséquence directe de $[\![c_0; c_1]\!]\sigma = [\![c_1]\!]([\![c_0]\!])\,\sigma$. En effet, on suppose $[\![\sigma]\!] \models^I A$. Soit $\rho = [\![c_0]\!]\sigma$: par hypothèse (seq), $\rho \models^I C$. Par hypothèse (seq) encore, $[\![c_1]\!]\rho \models^I B$. Donc comme $[\![c_0; c_1]\!]\sigma = [\![c_1]\!]([\![c_0]\!])\,\sigma = [\![c_1]\!]\rho$ on a le résultat voulu.

Correction de la conditionnelle. On rappelle : (if) $\frac{\{A\wedge b\}c_0\{B\} \quad \{A\wedge\neg b\}c_1\{B\}}{\{A\}if\ b\ then\ c_0\ else\ c_1\{B\}}$. On suppose $\sigma \models^I A$. De deux choses l'une (j'ignore le cas $\perp$) :

- $[\![b]\!]\sigma = true$: alors $\sigma \models^I A \wedge b$. Donc par hypothèse (if) et par récurrence, $[\![c_0]\!]\sigma \models^I B$. Or dans ce cas $[\![\text{if } b \text{ then } c_0 \text{ else } c_1]\!]\sigma = [\![c_0]\!]\sigma$.
- Ou : $[\![b]\!]\sigma = false$: alors $\sigma \models^I A \wedge \neg b$. Donc par hypothèse (if) et par récurrence, $[\![c_1]\!]\sigma \models^I B$. Or dans ce cas $[\![\text{if } b \text{ then } c_0 \text{ else } c_1]\!]\sigma = [\![c_1]\!]\sigma$.

Correction de la boucle. On rappelle : (while) $\frac{\{A \wedge b\} c \{A\}}{\{A\} while\ b\ do\ c \{A \wedge \neg b\}}$. On suppose $\sigma \models^I A$, alors de deux choses l'une :

- $[\![b]\!]\sigma = true$: alors $\sigma \models^I A \wedge b$. Donc par hypothèse (while) et par récurrence, $\sigma_1 = [\![c]\!]\sigma \models^I A$. Or $[\![\text{w=while } b \text{ do } c]\!]\sigma = [\![\text{if } b \text{ then } c \text{ ;w}]\!]\sigma$ qui est égal dans ce cas à $[\![w]\!]([\![c]\!]\sigma)$. On arrive donc à une situation où on cherche à prouver $[\![w]\!]\sigma_1 \models^I A \wedge \neg B$ quand $\sigma_1 \models^I A$. Si le nombre d'itération est infini et $[\![w]\!]\sigma = \bot \models^I false$ et $false \Rightarrow A \wedge \neg b$! Sinon, par récurrence sur le nombre d'itération de la boucle si elle est finie, on arrive à $\sigma_n \models^I A$ et $[\![b]\!]\sigma_n = false$ (cas suivant).
- Ou : $[\![b]\!]\sigma = false$: alors $\sigma \models^I A \wedge \neg b$. Or dans ce cas $[\![\text{while } b \text{ do } c]\!]\sigma = \sigma$.

Maintenant, montrons informellement l'incomplétude des règles de preuve. Le langage d'assertion $Assn$ contient au moins l'arithmétique de Péano du premier ordre. Le théorème d'incomplétude de Gödel implique en particulier qu'il existe des assertions vraies pour un programme, qui ne peuvent être prouvées par notre système de déduction formel. □

De plus, notre schéma de preuve n'est pas entièrement automatisable. Les outils basés sur un système de preuve à la Hoare (Spec♯, Frama-C, ...) sont en général interactifs.

On considère le programme suivant, pour donner un exemple d'utilisation de notre système de preuve :

$$w \equiv (while\ X > 0\ do\ Y = X * Y;\ X = X - 1;)$$

On veut prouver :

$$\{X = n \wedge n \geq 0 \wedge Y = 1\} w \{Y = n!\}$$

Il faut en premier lieu trouver un invariant de boucle suffisamment fort pour pouvoir prouver notre assertion. C'est bien sûr là que réside la principale difficulté pour l'automatisation ! On pose :

$$I \equiv \{YX! = n! \wedge X \geq 0\}$$

On doit donc prouver :

$$\{I \wedge X > 0\} Y = X * Y; X = X - 1; \{I\}$$

Prouvons maintenant l'invariant de boucle. Par la règle d'affectation, on a :

$$\{I[(X-1)/X]\} X = X - 1; \{I\}$$

où $I[(X-1)/X] \equiv (Y(X-1)! = n!) \wedge X - 1 \geq 0$. Par la règle d'affectation encore une fois :

$$\{XY(X-1)! = n! \wedge X - 1 \geq 0\} Y = X * Y; \{I[(X-1)/X]\}$$

Par la règle de composition séquentielle :

$$\{XY(X-1)! = n! \wedge (X-1) \geq 0\} Y = X * Y; X = X - 1; \{I\}$$

On a bien sûr :

$$\begin{array}{rcl} I \wedge X > 0 & \Rightarrow & YX! = n! \wedge X \geq 0 \wedge X > 0 \\ & \Rightarrow & YX! = n! \wedge X \geq 1 \\ & \Rightarrow & XY(X-1)! = n! \wedge (X-1) \geq 0 \end{array}$$

Donc, par la règle d'affaiblissement :

$$\{I \wedge X > 0\} Y = X * Y; X = X - 1; \{I\}$$

On applique la règle pour les boucles `while` :

$$\{I\} w \{I \wedge X \not> 0\}$$

Et on a $(X = n) \wedge (n \geq 0) \wedge (Y = 1) \Rightarrow I$:

$$\begin{array}{rcl} I \wedge X \not> 0 & \Rightarrow & YX! = n! \wedge X \geq 0 \wedge X \not> 0 \\ & \Rightarrow & YX! = n! \wedge X = 0 \\ & \Rightarrow & Y0! = Y = n! \end{array}$$

Alors, par la règle d'affaiblissement :

$$\{(X = n) \wedge (Y = 1)\} w \{Y = n!\}$$

En général la preuve et le code sont imbriqués, pour mieux présenter la preuve, comme ce que l'on avait fait en début de cette section :

$\{X = n \wedge n \geq 0 \wedge Y = 1\}$
while (X>0) {
$\{I \wedge X > 0\}$
  Y=X*Y;
$\{I[(X-1)/X]\}$
  X=X−1;
$\{I\}$
}
$\{Y = n!\}$

Un dernier mot sur la décidabilité de ce système de preuve. Considérons pour un programme quelconque `P` le programme `Q` suivant (`x` est une variable n'apparaissant pas dans `P`) :

```
int x=0;
...
P
...
x=1;
```

On souhaite prouver le triplet de Hoare $\{true\}Q\{x = 1\}$. Ceci est équivalent au problème de l'arrêt (chapitre 7) qui est indécidable. En pratique, on arrive quand même à prouver la terminaison de nombreux programmes, en utilisant non plus des assertions « invariantes » comme en logique de Hoare, mais des fonctions dites « variants ». C'est typiquement une fonction dépendant de l'environnement d'exécution du programme, qui est positive et décroissante le long de toute exécution (par exemple, à chaque tour de boucle). Un exemple en avait été en fait donné au chapitre 5.

Ces méthodes, ou en tout cas des systèmes de preuve d'esprit similaire, on été implémentées en « vrai ». Une application classique est ce que l'on appelle la programmation *par contrats*, qui a une longue histoire depuis C.A.R.Hoare en 1974 : écriture des préconditions et postconditions pour chaque fonction (*contrats*) - en même temps que le code ; ajout d'invariants dans le code pour aider le prouveur pour une vérification. Par exemple, c'est inclus dans le langage Eiffel (Bertrand Meyer 1985). Ou plus récemment, dans les outils de développement Microsoft : Code Contracts/Spec♯ pour .net/C♯ (2009).

Exercices

1. Prouver :

```
{X=m ∧ Y =n ∧ n≥0}
R = 0;
while (X != 0) {
  R = R + Y;
  X = X - 1;
}
{R = m×n}
```

2. Valider le tri par insertion suivant :

```
public static void triInsertion(int tableau[]) {
  int longueur=tableau.length;

  for(int i=1;i<longueur;i++) {
    int memory=tableau[i];
    int compt=i-1;
    boolean marqueur;
    do {
         marqueur=false;
         if (tableau[compt]>memory) {
           tableau[compt+1]=tableau[compt];
           compt--;
           marqueur=true; }
         if (compt<0) marqueur=false;
       } while(marqueur);
    tableau[compt+1]=memory;
    }
}
```

c'est-à-dire prouver que le tri par insertion réalise bien le tri de n éléments quelconques donnés au début. On introduira les prédicats nécessaires et on cherchera un invariant suffisant pour la preuve, en tête des deux boucles.

Chapitre 9

Typage, et programmation fonctionnelle

Dans ce chapitre, nous abordons certains aspects de la programmation fonctionnelle. Au lieu de se concentrer sur la syntaxe d'un langage en particulier (OCaml, cf. [4] par exemple, que la plupart d'entre vous connaissent, mais pas tous), on va utiliser un langage un peu abstrait, qui comprend tous les ingrédients classiques des langages fonctionnels. On aurait pu utiliser le λ-calcul, voir [2], mais cela est abstrait de façon non nécessaire pour ce cours introductif, on a plutôt choisi PCF, qui est plus proche d'un vrai langage de programmation.

9.1 PCF (non typé)

Dans les langages fonctionnels, la particularité est que les fonctions sont des objets « de première classe ». Dans PCF en particulier, on aura la construction de fonction `fun` x `->` t correspondant au Caml (au nommage près) :

```
let f x = t
```

L'application d'une fonction à un argument est notée, comme en OCaml, $t\ t$. Remarquez que l'on peut appliquer une fonction à une fonction, et même à soi-même, dans PCF sans typage. Ceci nous posera d'ailleurs quelques soucis pour définir une sémantique compréhensible de PCF, on a choisi ici une sémantique dite « opérationnelle » qui a le double avantage d'être plus facile à développer (qu'une sémantique dénotationnelle classique), et qui vous permettra de voir une autre façon de donner une sémantique à un langage de programmation, en quelque sorte plus proche de l'implémentation en machine. Ceci sera développé à la section 9.2.

Finalement, la grammaire complète de PCF est la suivante :

$$
\begin{array}{rcl}
t & ::= & x \\
& | & \texttt{fun}\ x\ \texttt{->}\ t \\
& | & t\ t \quad | \quad t \times t \\
& | & n \\
& | & t + t \quad | \quad t - t \quad | \quad t * t \quad | \quad t/t \\
& | & \texttt{ifz}\ t\ \texttt{then}\ t\ \texttt{else}\ t \\
& | & \texttt{fix}\ x\ t \\
& | & \texttt{let}\ x = t\ \texttt{in}\ t
\end{array}
$$

Remarques : On pourrait rajouter une construction somme, comme au chapitre 4, mais nous ne compliquerons pas inutilement la sémantique ici. Remarquez également que le langage PCF est complet au sens de Turing. Il permet de calculer toutes les fonctions récursives partielles, cf. chapitre 5.

Nous avons déjà rencontré toutes les constructions syntaxiques plus haut, à des différences mineures. Par exemple `ifz` est le test à zéro, ce qui est un peu différent de ce que nous avons rencontré dans le langage impératif jouet du chapitre 6.

Nous n'avons pour l'instant par rencontré `fix`. Il s'agit d'un opérateur de point fixe qui permet de définir des fonctions récursives (interdites syntaxiquement dans PCF). Etant donné un terme PCF t et une variable libre x de t, `fix` $x\ t$ est « moralement » le (plus petit) point fixe de la « fonction » qui à tout x associe $t(x)$. Par exemple, la fonction factorielle sera définie en PCF par :

$$\texttt{fix}\ f\ \texttt{fun}\ n\ \texttt{->}\ \texttt{ifz}\ n\ \texttt{then}\ 1\ \texttt{else}\ n * (f(n-1))$$

La fonction factorielle est en effet la « plus petite » fonction f telle que

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * f(n-1) & \text{sinon} \end{cases}$$

La sémantique *dénotationnelle* en avait été donnée au chapitre 6 (plus petit point fixe d'une certaine fonctionnelle). En Caml, cela correspond au `let rec`.

9.2 Sémantique opérationnelle

On va décrire les actions, une à une, lors de l'exécution d'un programme PCF (sémantique *petits pas*). Cela va prendre la forme de *règles de réduction* ou de *réecriture* :

$$p \to q$$

ou « le terme p se réécrit (ou se réduit en une étape) en le terme q »

En fait, ces règles vont former un automate (cf. programme de taupe) à partir d'un programme PCF.

Les noeuds du graphe de transition, ou de l'automate, seront des termes PCF, c'est-à-dire des programmes. Les actions de l'automate, ou les arcs du graphe de transition sont appelés des règles de réduction, car en quelque sorte ces actions

consistent à modifier le programme, au fur et à mesure de son exécution, jusqu'à obtenir une forme résiduelle, où plus rien n'est exécutable.

La règle la plus importante est la β-réduction :

$$(\texttt{fun}\ x\ \texttt{->}\ t)u \to t[u/x]$$

où $t[u/x]$ est le terme t dans lequel on remplace syntaxiquement toutes les occurrences de la variable x par le terme u. C'est celle qui explique comment sont évaluées les fonctions, à partir des arguments.

On a également des règles décrivant le calcul arithmétique, qui sont assez tautologiques :

$$p + q \to n$$

si l'entier p plus l'entier q est égal à n. On ne les donne pas pour la multiplication ni pour les autres opérations, cela est bien sûr similaire.

Pour les conditionnelles on a les règles :

$$\begin{array}{l}\texttt{ifz}\ 0\ \texttt{then}\ t\ \texttt{else}\ u \to t \\ \texttt{ifz}\ n\ \texttt{then}\ t\ \texttt{else}\ u \to u \quad \text{si } n \neq 0\end{array}$$

Pour l'opérateur de point fixe :

$$\texttt{fix}\ x\ t \to t[\texttt{fix}\ x\ t/x]$$

On va un peu jouer avec cette règle par la suite, elle peut paraître un peu magique, mais cela doit vous rappeler les règles de calcul de point fixe que l'on a vues au chapitre 8 en preuve à la Hoare.

Enfin nous avons une règle pour la *définition* :

$$\texttt{let}\ x = t\ \texttt{in}\ u \to u[t/x]$$

Donnons un exemple simple : un petit calcul arithmétique.

$$\begin{array}{lcll}(\underline{\texttt{fun}\ x}\ \texttt{->}\ x+2)\ \underline{3} & \to & \underline{3+2} & \beta\text{-réduction} \\ & \to & 5 & \text{règles arithmétiques}\end{array}$$

Remarquez que l'on a souligné les parties des termes PCF qui *intéragissent*, et qui vont être réduites. Ces parties s'appellent des *rédex*.

En fait, ce langage est assez redondant. On pourrait se passer de l'arithmétique par exemple.

Définissons :

$$[n] = \texttt{fun}\ z\ \texttt{->}\ \texttt{fun}\ s\ \texttt{->}\ s(s(s(\ldots(s\ z)\ldots)))$$

(où on répète $n \in \mathbb{N}$ fois l'application de s) En quelque sorte, ce terme *représente* l'entier n. On peut ensuite coder facilement les opérations, addition, multiplication :

$$\begin{array}{lcl}+ & = & \texttt{fun}\ n\ \texttt{->}\ \texttt{fun}\ p\ \texttt{->}\ \texttt{fun}\ z\ \texttt{->}\ \texttt{fun}\ s\ \texttt{->}\ ns(psx) \\ \times & = & \texttt{fun}\ n\ \texttt{->}\ \texttt{fun}\ p\ \texttt{->}\ \texttt{fun}\ z\ \texttt{->}\ \texttt{fun}\ s\ \texttt{->}\ n(pf)z\end{array}$$

C'est un codage connu sous le nom de *entiers de Church* (du nom d'Alonzo Church, 1930). On pourrait faire de même pour les booléens et pour la conditionnelle.

Une question naturelle est la suivante : définissons-nous bien quelque chose avec ces règles de réduction ? Pour cela, il serait bon de pouvoir s'assurer de la terminaison du processus de réduction. Mais il n'est hélas pas vrai que le calcul de réduction termine toujours, en voici un exemple :

$$\texttt{fix}\ x\ x \to \texttt{fix}\ x\ x \to \dots$$

donc ne termine pas. En même temps, que veut-dire ce terme ? On va en reparler tout de suite, et aussi à la section 9.6.

Ce terme, et d'autres similaires, sont en effet pratiques, il permettent également de se passer du terme `fix` , en tout cas, tant que l'on est dans un cadre non typé.

Définissons le combinateur Y, qui permet en quelque sorte de remplacer le terme `fix` :

$$Y = \texttt{fun}\ f\ \texttt{->}\ (\texttt{fun}\ x\ \texttt{->}\ f\ (x\ x))(\texttt{fun}\ x\ \texttt{->}\ f\ (x\ x))$$

Soit alors g un terme PCF, on a :

$$\begin{array}{rcl} Y\ g & = & (\underline{\texttt{fun}\ f\ \texttt{->}\ (\texttt{fun}\ x\ \texttt{->}\ f\ (x\ x))} \\ & & \quad \underline{(\texttt{fun}\ x\ \texttt{->}\ (f\ (x\ x))))g} \\ & & \beta\text{-réduction externe} \\ & = & \underline{(\texttt{fun}\ x\ \texttt{->}\ g\ (x\ x))}(\underline{\texttt{fun}\ x\ \texttt{->}\ g\ (x\ x))} \\ & & \beta\text{-réduction interne} \\ & = & g(\texttt{fun}\ x\ \texttt{->}\ g\ (x\ x))(\texttt{fun}\ x\ \texttt{->}\ g\ (x\ x)) \\ & = & g\ (Y\ g) \end{array}$$

Oui mais, on aurait pu aussi évaluer de la façon suivante ce terme, en effectuant la deuxième β-reduction (interne) avant la première. On aurait alors eu :

$$\begin{array}{rcl} Y\ g & = & (\texttt{fun}\ f\ \texttt{->}\ (\underline{\texttt{fun}\ x}\ \texttt{->}\ (f\ (x\ x)) \\ & & \quad \underline{(\texttt{fun}\ x\ \texttt{->}\ (f\ (x\ x))))}\big)\ g \\ & & (\beta\text{-réduction interne}) \\ & = & (\texttt{fun}\ f\ \texttt{->}\ (f\ (\underline{\texttt{fun}\ x}\ \texttt{->}\ f\ (x\ x))) \\ & & \quad \underline{(\texttt{fun}\ x\ \texttt{->}\ f\ (x\ x)))}\ g \\ & & (\beta\text{-réduction interne}) \\ & = & (\texttt{fun}\ f\ \texttt{->}\ f\ f\ (\texttt{fun}\ x\ \texttt{->}\ f\ (x\ x)) \\ & & \quad (\texttt{fun}\ x\ \texttt{->}\ f\ (x\ x)))\ g \\ & = & \text{etc.}\,! \end{array}$$

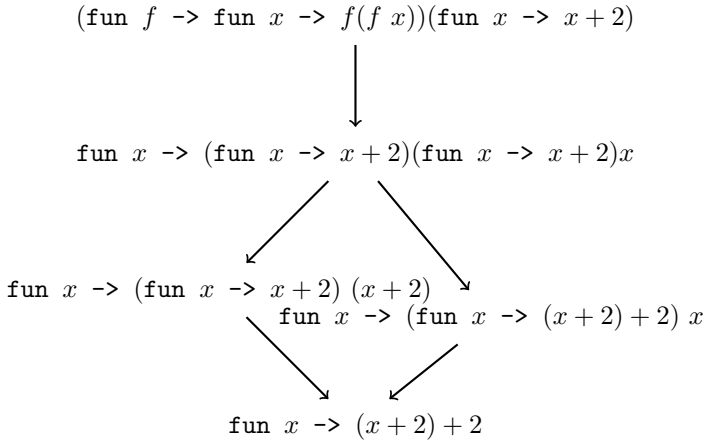
Et cela ne termine pas encore une fois (en fait cela devrait vraiment vous rappeler l'itération de Kleene). C'est bien sûr le même phénomène que l'on avait avec le terme `fix` $x\ x$!

9.3 Ordres d'évaluation

On voit qu'en fait, on n'a pas spécifié l'ordre d'utilisation des règles de réduction, et que l'on peut voir pour le même terme, des réductions qui terminent et d'autres qui ne terminent pas. Par contre, quand on choisit des réductions qui terminent, est-ce que le résultat dépend de la façon dont on réduit ?

Appelons *terme irréductible*, dans PCF, un terme sur lequel on ne peut appliquer aucune règle de réduction. On a alors une propriété de *confluence* : si on utilise les règles de réduction dans n'importe quel ordre, et de façon à *terminer sur un terme irréductible*, alors on termine toujours sur le même terme irréductible. Ceci est clairement faux si on oublie la condition d'irréductibilité [1].

Donnons un exemple :



On voit bien qu'en général, beaucoup d'ordres d'évaluation sont possibles, on va voir maintenant que parmi ceux-ci, un certain nombre ont une signification particulière. On va voir que certains ordres d'évaluation correspondent au passage d'argument par valeur (comme pour les langages impératifs dont nous avions donné la sémantique au chapitre 6), ou au passage d'argument par référence, ou encore par nécessité (sémantique du langage fonctionnel Haskell).

9.4 Appel par nom, appel par valeur et appel par nécessité

Commençons ici par imposer un ordre d'évaluation. Ici, on va réduire les sous-termes les plus profonds (sans être trop formel). Cela revient à évaluer d'abord les arguments des fonctions, avant les fonctions elles-mêmes. C'est le passage d'arguments par valeur.

1. Penser encore à `fix x x` !

Exemple :

$$\begin{array}{rcl}(\texttt{fun}\ x\ \texttt{->}\ (x+x))(\underline{2+3}) & \rightarrow & (\underline{\texttt{fun}\ x\ \texttt{->}\ (x+x)})\ \underline{5} \\ & & \text{évaluation de l'argument} \\ & \rightarrow & \underline{5+5} \\ & \rightarrow & 10 \\ & & \text{terme irréductible !}\end{array}$$

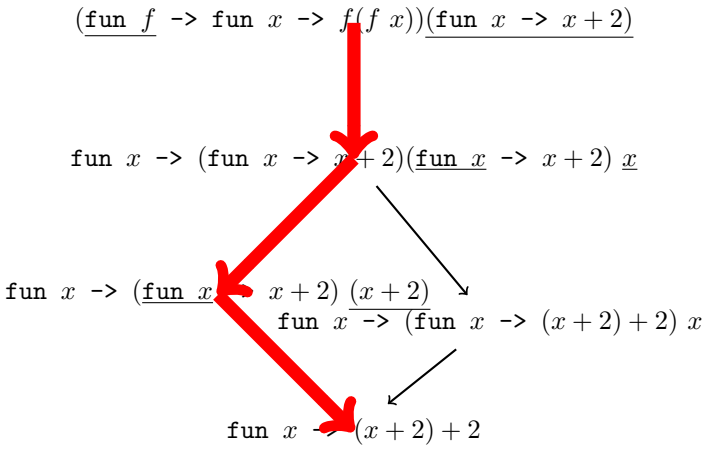
Evaluons maintenant les rédexs de l'extérieur vers l'intérieur. Cela revient à calculer ce que l'on peut d'une fonction, sans les arguments, et de n'évaluer les arguments qu'au besoin, petit à petit. Il s'agit d'un passage par référence des arguments : on ne regarde ce qui est pointé par une référence, qu'au besoin.

En voici un exemple : $Y\ g$, en tout cas, l'évaluation qu'on en avait faite au début de ce chapitre, et qui termine. On ne veut pas évaluer en effet à l'intérieur de Y, pour avoir la terminaison.

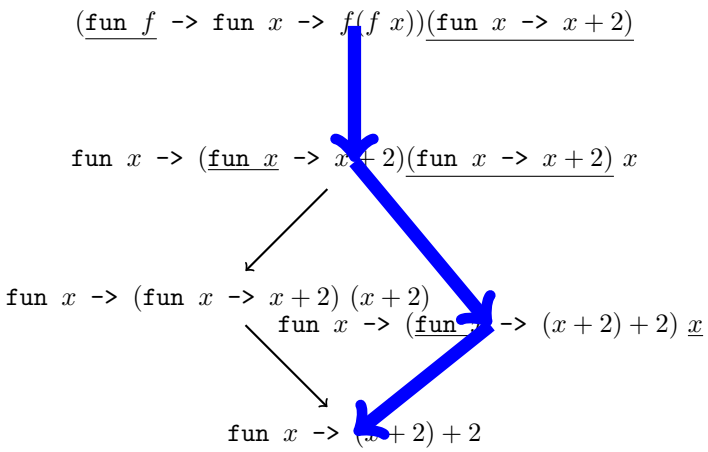
Une question naturelle est alors de savoir si l'on peut quand même définir un opérateur de point fixe pour l'appel par valeur. La réponse est positive, il s'agit du combinateur Z (dans un langage non typé, encore une fois) :

$$\begin{array}{rl}Z = \texttt{fun}\ f\ \texttt{->} & (\texttt{fun}\ x\ \texttt{->}\ f\ (\texttt{fun}\ v\ \texttt{->}\ ((x\ x)v))) \\ & (\texttt{fun}\ x\ \texttt{->}\ f\ (\texttt{fun}\ v\ \texttt{->}\ ((x\ x)v)))\end{array}$$

Donnons un autre exemple, ici d'appel par nom :



On rappelle que l'on avait choisi pour l'appel par valeur, l'évaluation suivante :



En fait, il existe une autre évaluation des termes PCF, qui s'appelle l'appel par nécessité, et qui est une variante de l'appel par nom avec partage des sous-termes et des réductions correspondantes. C'est donc assez proche de l'appel par nom, et permet aussi de définir simplement des combinateurs de points fixe type `Y`. Il est implémenté dans le langage fonctionnel Haskell (pas Caml), qui est un langage créé en 1987 et nommé en l'honneur du logicien Haskell Curry. Il est certes plus dur à compiler efficacement, mais parfois plus souple pour le programmeur. On en donne un exemple tout de suite avec l'implémentation de « structures de données infinies ».

Par exemple, en Haskell, on peut définir les choses suivantes :

```
numsFrom n = n : numsFrom (n+1)
squares = map (\^2) (numsfrom 0)
take 5 squares => [0,1,4,9,16]
```

`numsFrom n` construit une liste infinie d'entiers, commençant en `n`. `squares` applique la fonction « carrée » sur la liste infinie $(0, 1, 2, \ldots)$. `take` extrait un préfixe fini : c'est l'évaluation *par nécessité* de ce terme qui demande juste ce qu'il faut d'évaluation de la liste infinie `numsFrom 0`.

9.5 Combinateurs de point fixe, en Haskell, Caml et Java

En Haskell également, on peut programmer directement (bien que non nécessaire) un combinateur de point fixe (mais pas le code de Y), qui va terminer :

```
y(f) = f (y f)
fact f n = if (n == 0) then 1 else n * f (n-1)
y(fact) 10
...
```

Alors qu'en Caml, a priori, on ne peut pas coder le combinateur `Y` à cause de l'appel par valeur (et du typage, cf. plus loin dans ce chapitre), mais on peut tricher un peu : on peut utiliser une *clôture* :

```
let rec fix f x = f (fix f) x

let factabs fact = function
 0 -> 1
 | x -> x * fact (x-1)

let x = (fix factabs) 5
```

On peut s'en sortir aussi avec des références, bien sûr, et des types récursifs :

```
type 'a recc = In of ('a recc -> 'a)
let out (In x) = x

let y f = (fun x a -> f (out x x) a)
            (In (fun x a -> f (out x x) a))
```

Peut-on coder cela en Java ? Il faut être sérieusement fou, mais on va utiliser une forme faible des *clôtures*, en les codant par des objets JAVA (et des interfaces – cf. chapitre 4).

Commençons par définir :

```
class YFact {
  // int -> int
  interface IntFunc { int apply(int n); }

  // (int -> int) -> (int -> int)
  interface IntFuncToIntFunc {
              IntFunc apply(IntFunc f); };

  // Higher-order function returning an int function
  // F: F -> (int -> int)
  interface FuncToIntFunc {
              IntFunc apply(FuncToIntFunc x); }

  // Function from IntFuntToIntFunc to IntFunc
  // ((int -> int) -> (int -> int)) -> (int -> int)
  interface IntFuncToIntFuncToIntFunc {
              IntFunc apply(IntFuncToIntFunc r);};
```

Maintenant, le code JAVA de Z et de factorielle est le suivant :

```
(new IntFuncToIntFuncToIntFunc() {
  public IntFunc apply(final IntFuncToIntFunc r) {
    return (new FuncToIntFunc() {
      public IntFunc apply(final FuncToIntFunc f) {
             return f.apply(f); }})
      .apply(new FuncToIntFunc() {
       public IntFunc apply(final FuncToIntFunc f) {
```

```
        return r.apply(
          new IntFunc() { public int apply(int x) {
    return f.apply(f).apply(x); }});}});}}
```

Dans ce code, `new` correspond à une construction de fonction `fun`, et `apply` correspond à une application dans PCF (`apply(p).q`=$p\ q$).

On peut vérifier que ce code (du à Ken Schiriff) fonctionne :

```
public static void main(String args[]) {
  System.out.println(
  // Z combinator
  ...
  .apply(
  // Recursive function generator
  new IntFuncToIntFunc() {
    public IntFunc apply(final IntFunc f) {
      return new IntFunc() {
        public int apply(int n) {
          if (n == 0) return 1;
          else return n * f.apply(n-1); }};}}
  ).apply(
  // Argument
    Integer.parseInt(args[0]))); } }
```

En l'exécutant :

```
> javac YFact.java
> java YFact 10
3628800
```

9.6 Typage

Jusqu'à présent, nous avons considéré un langage fonctionnel non typé. Qu'est-ce que le typage ? L'intérêt du typage est d'éliminer des termes qui paraissent n'avoir aucun sens. En fait, comme on le verra plus tard, les types font partie d'une preuve de bon fonctionnement, au sens théorie de la preuve. Les types sont en un sens très précis, que l'on va illustrer dans cette section, des formules logiques assurant une partie de la preuve du programme typé.

Donc un bon système de typage doit par exemple éliminer des choses comme (`fun` x `->` x)$+1$. Et finalement, il sera difficile de ne pas éliminer non plus des termes tels `fun` x `->` $x\ x$ ni Y, car il est difficile de concevoir x comme à la fois une fonction, et un argument à lui-même. La première conséquence de cette remarque est donc qu'un langage typé aura a priori un combinateur de point fixe explicite.

On peut faire plusieurs choix concernant le typage, dans les langages de programmation : on peut avoir un langage où on déclare les types et où il y a une vérification minimale des types (Java, etc.), éventuellement avec règles de transtypage (Java, C, etc.). Il s'agit de ce que l'on appelle un *typage faible*.

Ou alors, on peut faire le choix d'un langage avec *inférence de types* (Caml, etc.), et où ceux-ci (hors références...) assurent un bon comportement des programmes, minimal, mais de l'ordre de la preuve (à la Hoare, ou presque) de certaines propriétés de programme. Il s'agit alors d'un typage fort. En quelque sorte, le typage fort est une preuve de cohérence du programme, très formelle. On verra brièvement cette idée à travers une illustration de la correspondance type et formule de logique / programme de ce type et preuve de cette formule (isomorphisme de Curry-Howard), à la fin de ce chapitre.

On peut démarrer par associer des types relativement simples à des termes PCF, appelés ici types monomorphes :

$$\begin{array}{rcl} \tau & ::= & int \\ & | & \tau \to \tau \\ & | & \tau \times \tau \end{array}$$

Les types sont donc ici, soit entier (int – soit à vrai dire des types de base pré-spécifiés), soit un type fonctionnel ($\to$), soit encore un type produit ($\times$).

Le défaut de tels types est que, par exemple, la fonction identité n'aura pas de « type plus général ». La fonction identité, appliquée à un entier, a le type $int \to int$. Appliquée à une fonction de type $int \to int$, elle aura le type $(int \to int) \to (int \to int)$ et ainsi de suite.

Une façon de remédier à ce défaut, est d'introduire du « polymorphisme » (on en a vu une forme dans les langages orientés objets au chapitre 4) :

$$\begin{array}{rcll} \tau & ::= & int \mid bool \mid \ldots & \text{types de base} \\ & | & \tau \times \tau & \text{type produit} \\ & | & \tau \to \tau & \text{type d'une fonction} \\ & | & \alpha & \text{variable de type} \\ & | & \forall \alpha.\tau & \text{type polymorphe} \end{array}$$

On a ainsi rajouté des « variables de type », qui permettent de gagner en aisance. Ainsi, la fonction identité aura comme type $\forall \alpha.\alpha \to \alpha$, où α pourra être instancié plus tard à n'importe quel autre type. C'est ce que fait le typage de OCaml par exemple.

Donnons maintenant une sémantique au typage des termes PCF. Pour typer une expression, on a besoin de la connaissance du typage de l'environnement Env. Au lieu d'avoir Env = Var $\to$ Val, un environnement Γ associe à chaque variable x, un type $\Gamma(x)$ dans notre grammaire de types. On écrira souvent $\Gamma, x : \tau$ pour l'environnement qui vaut Γ (défini sur toutes les variables sauf x), et dans lequel x a le type τ.

Dans l'environnement Γ, l'expression e (de PCF) a le type τ se note :

$$\Gamma \models e : \tau$$

C'est ce que l'on appelle un jugement de typage. On va définir un *système formel* comme au chapitre 8, permettant de dériver ces jugements de typage. La dérivation du terme d'un terme PCF donnera lieu à un arbre de typage, exactement comme pour les systèmes de preuve du chapitre 5.

Les règles de typage sont ainsi :
Pour les variables :

$$\frac{}{\Gamma \models x : \Gamma(x)}$$

Pour les constantes :

$$\frac{}{\Gamma \models n : int}$$

Pour les opérations arithmétiques :

$$\frac{\Gamma \models s : int \quad \Gamma \models t : int}{\Gamma \models s + t : int}$$

et ainsi de suite, pour les autres opérations arithmétiques, de façon évidente.
Pour la création de fonctions :

$$\frac{\Gamma, x : A \models t : B}{\Gamma \models \texttt{fun}\ x\ \texttt{->}\ t : A \to B}$$

Pour l'application :

$$\frac{\Gamma \models u : A \quad \Gamma \models v : A \to B}{\Gamma \models v\ u : B}$$

Pour l'affectation :

$$\frac{\Gamma \models t : A \quad \Gamma, x : A \models u : B}{\Gamma \models \texttt{let}\ x = t\ \texttt{in}\ u : B}$$

Pour la conditionnelle :

$$\frac{\Gamma \models t : int \quad \Gamma \models u : A \quad \Gamma \models v : A}{\Gamma \models \texttt{ifz}\ t\ \texttt{then}\ u\ \texttt{else}\ v : A}$$

Pour l'opérateur de point fixe :

$$\frac{\Gamma, x : A \models t : A}{\Gamma \models \texttt{fix}\ x\ t : A}$$

Et enfin pour la paire :

$$\frac{\Gamma \models u : A \quad \Gamma \models v : B}{\Gamma \models (u, v) : A \times B}$$

Donnons un exemple de typage. Considérons le terme : $\texttt{let}\ f = \texttt{fun}\ x\ \texttt{->}\ x + 1\ \texttt{in}\ f\ 2$. On a alors l'arbre de jugements de typage :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{...}{x : int \models 1 : int}}{x : int \models x + 1 : int}}{\emptyset \models \texttt{fun}\ x\ \texttt{->}\ x + 1 : int \to int} \qquad \dfrac{...}{f : int \to int \models 2 : int}}{\emptyset \models \texttt{let}\ f = \texttt{fun}\ x\ \texttt{->}\ x + 1\ \texttt{in}\ f\ 2 : int}$$

On se doute qu'il y a un rapport entre bon typage et le bon comportement d'un terme PCF. Autre remarque : le combinateur Y (ou autre combinateur de point fixe) ne type pas, c'est pourquoi on a introduit `fix` dans le langage.

On a le théorème suivant, que l'on ne démontrera pas :

Théorème 4. *Si $\emptyset \models t : \tau$ alors la réduction de t est infinie ou se termine sur une valeur (un terme irréductible).*

On vient de donner les règles pour vérifier les types. En fait, un langage comme OCaml infère le type des termes, et pour ce faire, utilise des algorithmes particuliers. Le problème de l'inférence de type est en général extrêmement complexe, car il est équivalent à trouver une preuve dans une certaine logique.

Les algorithmes communément utilisés sont l'algorithme de Hindley (monomorphe) et de Damas-Milner (polymorphe – à l'origine du typage Caml). Dans ce dernier algorithme, tout terme (de Caml, ou de PCF, par exemple) a un type principal (le plus général). L'algorithme est fondé sur *l'unification* de termes du premier ordre (sorte de résolution d'équation dans une algèbre libre de termes). La complexité de ce type d'algorithme est au pire exponentielle, mais en pratique elle est quasi linéaire, comme les programmeurs OCaml ont pu le constater (le typage est très rapide en pratique). L'inférence de types est une forme d'inférence de *preuve* (voir le calcul des séquents, section 7.5), comme on le montre, de façon un peu informelle, dans la section suivante.

9.7 Théorie de la démonstration et typage

Les règles de typage sont très proches de la déduction naturelle, dans un fragment de la logique du chapitre 8. En fait, c'est une présentation d'un fragment *intuitioniste* par un *calcul de séquents*. Pour s'en convaincre, oublions les termes PCF dans certaines règles de typages, un instant.

Reprenons la création de fonctions – $(\Rightarrow I_g)$:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

Rappel (chapitre 8) :

$$(\Rightarrow I_g)\ \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma, A \Rightarrow B \vdash \Delta}$$

Donc oui, c'est la même règle, avec $\Delta = \emptyset$.

Maitenant l'affectation – (cut) :

$$\frac{\Gamma \vdash A \quad \Gamma, x : A \vdash B}{\Gamma \vdash B}$$

Rappel (chapitre 8) :

$$(cut)\ \frac{\Gamma \vdash A, \Delta \quad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}$$

Donc oui, c'est la même règle, avec $\Delta = \emptyset$, $\Gamma' = \Gamma$ et $\Delta' = B$.

Revenons à la règle de typage de la paire – $(\wedge I_d)$:

$$\frac{\Gamma \vdash A \quad \Gamma \vdash v : B}{\Gamma \vdash (u, v) : A \wedge B}$$

Rappel (chapitre 8) :

$$(\wedge I_d)\ \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}$$

Prenons un exemple typique de cette correspondance type/formules, et preuves. Revenons aux produits un instant.

Une fonction de $f : X \times Y$ vers Z peut-être considérée comme :

(i) bien sûr une fonction qui à un couple de valeurs (x, y), avec $x \in X$ et $y \in Y$, renvoie $f(x, y) \in Z$;

(ii) une fonction de X vers $Y \to Z$, qui à un x dans X associe la fonction *partielle* $f_x : Y \to Z$ telle que $f_x(y) = f(x, y)$;

(iii) un élément de $X \times Y \to Z$ (soit une fonction de () (unit) vers $X \times Y \to Z$).

Passer de (i) à (ii) est « naturel » et on le fait constamment en OCaml. On a une fonction (*d'ordre supérieur*)

$$\mathit{curry} : ((X \times Y) \to Z) \to (X \to (Y \to Z))$$

qui s'appelle la « curryfication ».

En Caml, cela se programme de la façon suivante :

```
let curry f x y = f (x, y);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

On a également la dé-curryfication :

```
let uncurry f (x, y) = f x y;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

Que l'on peut utiliser par exemple comme suit :

```
let f (x, y) = x+y and g = curry f;;
val f : int * int -> int = <fun>
val g : int -> int -> int = <fun>

let f5 = g 5;;
val f5 : int -> int = <fun>

let h x = function y -> f (x, y) and
    i = function x -> function y -> f (x, y);;
val h : int -> int -> int = <fun>
val i : int -> int -> int = <fun>
```

Une autre fonction « naturelle » est l'évaluation :

$$eval : (X \to Z) \times X \to Z$$

qui a tout x de X, et toute fonction de $X \to Z$ associe $eval(f, x) = f(x)$ dans Z. En Caml, cela se programme de façon évidente :

```
let eval f x = f x;;
val eval : ('a -> 'b) -> 'a -> 'b = <fun>
```

Vous remarquerez la similarité avec la logique propositionnelle. Le paradigme est celui des « proofs as programs », dans une logique « constructive » au moins :

« Programme=preuve de son type »

Illustrons cela par un exemple simple. On rappelle les fonctions Caml suivantes :

```
let curry f x y = f (x, y);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

La fonction `curry` est en fait une « preuve » de :

$$((a \wedge b) \implies c) \implies (a \implies (b \implies c))$$

De même pour l'« application » qui s'écrit en Caml :

```
let apply = uncurry eval;;
val apply : ('_a -> '_b) * '_a -> '_b = <fun>
```

La fonction `apply` est une preuve du « modus ponens » :

$$((a \implies b) \wedge a) \implies b$$

Reprenons maintenant encore `apply`. Supposons qu'on ait les axiomes (=« combinateurs ») `eval` et `uncurry` : on peut en déduire une preuve constructive du *modus ponens* :

$$(uncurry) \quad ((u \implies v) \implies w) \implies ((u \wedge v) \implies w)$$

en faisant $u = (a \implies b)$, $v = a$, $w = b$ d'où :

$$(((a \implies b) \implies a) \implies b) \implies (((a \implies b) \wedge a) \implies b)$$

Mais on sait par $(eval)$:

$$(eval) \quad ((a \implies b) \implies a) \implies b$$

Donc :

$$(uncurry\ eval) \quad ((a \implies b) \wedge a) \implies b$$

Cette preuve correspond à l'*exécution* de la composition des fonctions `uncurry` et `eval` :

```
uncurry eval;;
- : ('_a -> '_b) * '_a -> '_b = <fun>
```

De façon générale, on a la correspondance de Curry-Howard. Un programme de type `t` est une preuve de t comme suit : (en logique `intuitioniste`)

Terme logique	Type informatique
Implication	type fonctionnel
conjonction	type produit
disjonction	type somme
vrai	type `unit`
faux	$\perp$ (exception/boucle infinie)

Les quantificateurs correspondent aux types *dépendants*.

9.8 Pour aller plus loin

La plupart des cours permettant d'approfondir ce thème se trouvent au MPRI (en M2). Parmi les thèmes très avancés, on retrouve :

- la généralisation de la correspondance de Curry-Howard à la logique classique (*call-with-current-continuation*, transformation *continuation passing style*) ;
- certains « grands » théorèmes ont été interprétés comme des programmes (ex. théorèmes de complétude et d'incomplétude de Gödel, forcing de Cohen etc. - par Jean-Louis Krivine) ;
- des liens entre la topologie algébrique et certains systèmes de types : cf. « Homotopical Foundations » de Vladimir Voevodsky (médaille Fields 2002), `http://homotopytypetheory.org` et les travaux de Steve Awodey (CMU, Pittsburgh).

Exercices

1. Calculer la sémantique opérationnelle du terme PCF :

   $$(\texttt{fix}\ f\ \texttt{fun}\ x\ \texttt{->}\ \texttt{ifz}\ x\ \texttt{then}\ x + f(x-1))\,3$$

   en appel par valeur.

2. Soit f et g deux termes PCF de types :

   $$\begin{array}{ll} f : & \mathbb{N}^{p+1} \to \mathbb{N} \\ g : & \mathbb{N}^{p} \to \mathbb{N} \end{array}$$

   Prouver que h défini à partir de f et g par récursion primitive (chapitre 5) peut se définir en PCF. En déduire que PCF permet au moins de coder toutes les fonctions récursives primitives.

3. Définir un codage des booléens comme on avait fait pour recoder les entiers au début de ce chapitre. Donner alors des termes PCF permettant d'interpréter les tests sur les booléens (sans faire appel à `ifz`).

4. Typer le programme PCF (schémas monomorphe et polymorphe) :

$$\texttt{fix}\ f\ \texttt{fun}\ x\ \texttt{->}\ \texttt{ifz}\ x\ \texttt{then}\ x + f(x-1)$$

Chapitre 10

Programmation réactive synchrone

Ce chapitre introduit à un paradigme de programmation original, la programmation réactive synchrone (en particulier Lustre), également très utile en pratique, par exemple pour le codage du contrôle commande d'avions, de centrales nucléaires, etc. Le code du calculateur primaire de vol de l'A380 par exemple, est écrit en Scade (Esterel Technologies), qui est une version industrielle de Lustre. C'est aussi un langage de programmation à la sémantique très propre, qui permet d'illustrer encore les outils de la sémantique du chapitre 6. Ce langage a une belle sémantique et a ainsi de nombreux outils associés, de preuve, test, etc.

Ce paradigme vient en quelque sorte d'un mariage du contrôle et de l'informatique. En général, on peut décrire les programmes dans ce paradigme, graphiquement, par schéma-bloc (comme Matlab/Simulink le fait par exemple) mais également à travers un langage textuel. Ce langage est déclaratif, un programme est un ensemble d'équations, comme nous allons l'expliciter plus bas.

10.1 Lustre

Lustre est un langage de programmation déclaratif, donné par un ensemble « d'équations » mutuellement récursives en général, calculant des suites de signaux de sortie, à partir de signaux d'entrée. On peut voir la machine d'exécution sous-jacente comme une machine parallèle, dans laquelle chaque équation (ou « noeud ») est un processus traitant des signaux cadencés à un certain rythme temporel, et en renvoyant d'autres aux autres noeuds. La machine sous-jacente est donc un graphe de processus, avec un modèle d'exécution très simple : tous les processus ont la même horloge globale, c'est-à-dire qu'ils lisent leur entrées, calculent, et produisent leurs sorties tous en même temps, à chaque « tic » d'horloge. Lustre implémente en fait les réseaux de Kahn (on les introduit à la section 10.4) synchrones. Par synchrone, on entend le fait qu'un message par arc

du réseau est envoyé et reçu à chaque « tic » de l'horloge globale. Cela permet d'éviter l'utilisation de *tampons de communications* potentiellement non bornés, avec une puissance de calcul similaire aux réseaux de Kahn généraux.

Un programme Lustre opère sur un flot, c'est-à-dire une *suite de valeurs* : une variable `x` en Lustre représente une suite infinie de valeurs $(x_0, x_1, \ldots, x_n, \ldots)$; x_i est la valeur de `x` au temps i. Un programme Lustre prend un flot et renvoie un flot et toutes les opérations sont globales sur un flot :

- L'équation de flot $x = e$ est un raccourci pour $\forall n,\ x_n = e_n$;
- L'expression arithmétique sur les flots $x + y$ renvoie le flot $(x_0 + y_0, x_1 + y_1, \ldots, x_n + y_n, \ldots)$.

Lustre introduit également des opérateurs temporels. Ceux-ci sont :

- `pre` (*précédent*) qui donne la valeur au temps précédent, d'un flot argument : `pre(x)` est le flot $(\bot, x_0, \ldots, x_{n-1}, \ldots)$;
- `->` (*suivi de*) est utilisé pour donner des valeurs initiales d'un flot : `x->y` est le flot $(x_0, y_1, \ldots, y_n, \ldots)$.

Remarquez que les flots sont typés. `bool` par exemple est le type des flots de booléens.

Les expressions arithmétiques et booléennes, syntaxiquement, sont les mêmes que d'habitude, mais étendues point à point aux flots. On a également une forme syntaxique pour l'affectation : `let ... = ... tel`, les conditionnelles : `if ... then ... else`, et la séquence.

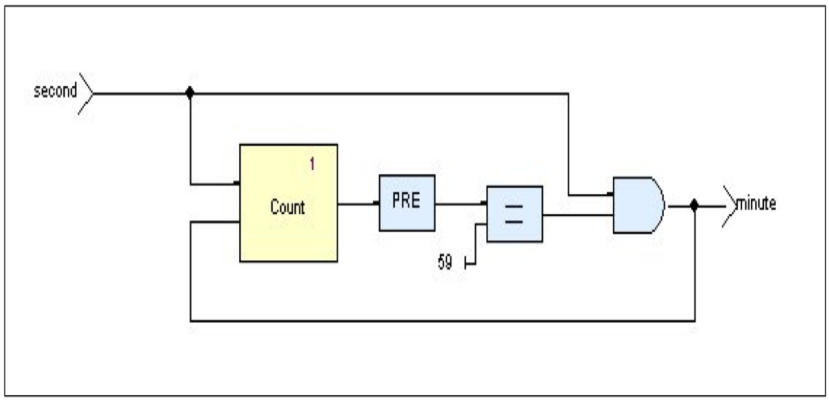
L'organisation d'un programme Lustre est faite ainsi. Un programme Lustre est un ensemble d'équations a priori potentiellement mutuellement récursives. Chaque équation est défini par un noeud identifié par le mot clé `node`. Une équation ou noeud est une fonction prenant des flots en argument, renvoyant un flot en résultat.

Donnons pour premier exemple un programme simple, compteur d'événements :

```
node Count(evt, reset: bool) returns (count: int);
let
  count = if (true->reset) then 0
          else if evt then pre(count)+1
          else pre(count);
tel
```

Dans ce programme, `true->reset` est un flot booléen, égal à vrai à l'instant initial et quand `reset` est vrai. Quand il est vrai, la valeur de `count` est renvoyée égale à zéro. Sinon, quand `evt` est vrai, on renvoie la valeur à l'instant précédent de `count` plus 1 ; sinon on conserve l'ancienne valeur.

La représentation graphique associé à cette version textuelle du programme est la suivante :



Voici un exemple d'utilisation de ce compteur d'événements.

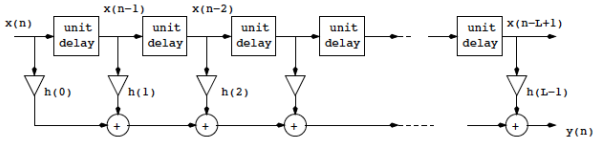
```
mod60 = Count(second, minute);
minute = second and pre(mod60)=59;
```

Dans ce programme, `mod60` est la sortie du noeud `Count`, qui compte les secondes, et se remet à zéro chaque minute. `minute` est vrai quand `seconde` est cadencé et que sa valeur précédente est de 59.

Prenons maintenant un exemple du monde du traitement du signal : les filtres linéaires à réponse finie. Ce sont des calculs récurrents qui prennent une entrée à l'instant n, x_n et renvoient en sortie, à l'instant n, y_n donnée par :

$$y_n = \sum_{m=0}^{L-1} h(m) x_{n-m}$$

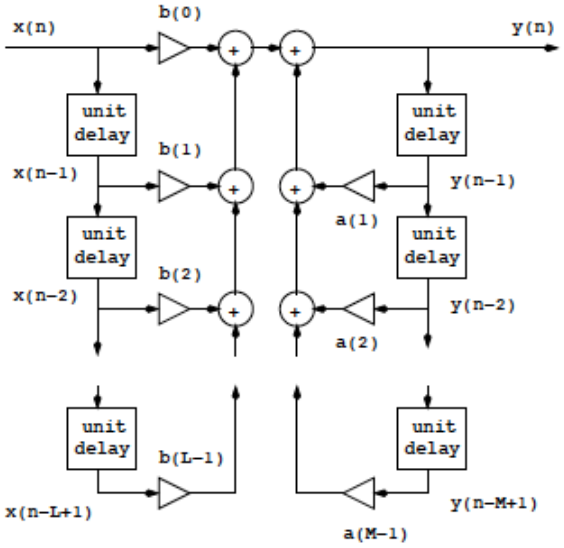
Graphiquement :



Prenons maintenant l'exemple des filtres linéaires à réponse infinie (filtres récursifs). Ils prennent en entrée à l'instant n, x_n, et renvoient en sortie à l'instant n, y_n donnée par :

$$y_n = \sum_{m=0}^{L-1} b(m) x_{n-m} + \sum_{m=1}^{M-1} a(m) y_{n-m}$$

Graphiquement :



Un code Lustre typique, correspondant, par exemple dans le cas où la sortie est donnée par : $y_n = x_n + 0.9y_{n-1}$:

```
node filter(x: real) returns (y: real);
  let
    y = x+0.0 -> 0.9*pre(y);
  tel;
```

Prenons maintenant un autre exemple : celui du chien de garde (*watchdog*). Il permet de gérer des échéances : il émet `alarm` quand `watchdog` est en attente et que `deadline` est vrai :

```
node WATCHDOG1(set, reset, deadline: bool) return (alarm: bool);
var watchdog_is_on: bool;
let
  alarm = deadline and watchdog_is_on;
  watchdog_is_on = false -> if set then true
                            else if reset then false
                            else pre(watchdog_is_on);
  assert not(set and reset);
tel;
```

(les flots booléens `set` et `reset` ne doivent pas être vrais en même temps).

Un des soucis principaux de la sémantique de tels langages est d'assurer la « causalité ». On veut ainsi que les équations définissant un programme aient une signification en termes de propagation d'information, et qu'en quelque sorte, celles-ci ne se mordent pas la queue. Pour assurer la causalité, comme tout réseau de Kahn se doit, on doit opérer des restrictions syntaxiques sur les programmes Lustre. Par exemple, `let x=x+1;` n'est pas un programme Lustre correct : le flot `x` dépend instantanément de lui-même, ce qui n'est pas possible (ou alors

il faudrait opérer une résolution d'équations, qui ne donnerait pas de solution ici). La condition syntaxique imposée ici est qu'une variable récursive doit être gardée par un délai. On ne peut pas écrire les choses suivantes :

```
x = x+1;
```

ni :

```
x = if b then y else z;
y = if b then t else x;
```

10.2 Cadencement et « calcul d'horloges »

Lustre fournit également des moyens de faire un « calcul d'horloges », c'est-à-dire, en particulier, de définir plusieurs horloges. Cela se fait entre autres par l'opérateur de sous-échantillonnage `when`. Celui-ci permet de cadencer différemment des processus (=noeuds), mais toujours selon un multiple du temps de base. Par exemple, l'opérateur de sous-échantillonage `X when B`, où `X` est un flot quelconque, `B` un flot booléen donne dans le cas plus bas :

B	*false*	*false*	*true*	*true*	*false*	*true*
X	X_0	X_1	X_2	X_3	X_4	X_5
Y=X when B			X_2	X_3		X_5

Ce calcul d'horloge repose également sur un opérateur de suréchantillonnage `current`. Celui-ci permet d'injecter un flot lent dans un nouveau flot rapide (cadencé au temps de base). Par exemple :

B	*false*	*false*	*true*	*true*	*false*	*true*
X	X_0	X_1	X_2	X_3	X_4	X_5
Y=X when B			X_2	X_3		X_5
Z=current Y	$\perp$	$\perp$	X_2	X_3	X_3	X_5

Remarque : au début `Z` n'a pas de valeur ; on utilise souvent `current ...->Y` plutôt que `current Y`

Considérons maintenant l'exemple suivant, du à Marc Pouzet. Il s'agit d'un additionneur :

```
node somme(i: int) returns (s: int);
let s = i -> pre s + i
tel;
```

On a par exemple :

1	1	1	1	1	1	1
cond	*true*	*false*	*true*	*true*	*false*	*true*
somme 1	1	2	3	4	5	6
somme(1 when cond)	1		2	3		4
(somme 1) when cond	1		3	4		6

Donc en général : `f(x when c)`$\neq$`(f x) when c` ; de même `current(x when c)`$\neq$`x`.

On pourrait vouloir écrire :

```
let half = true -> not (pre half);
    o = x & (x when half)
```

Que devrait-il se passer à la compilation ? Le code correspond au calcul $y_n = x_n \& x_{2n}$. Il faudrait donc un mécanisme de passage de valeurs par buffers qui ici ne serait pas borné $(n, \ldots, 2n)$. Ceci est interdit par un calcul d'horloge interne au compilateur.

Pour ce faire, les horloges utilisées par un noeud doivent être déclarées et visibles dans l'interface du noeud. Donnons un exemple de telle déclaration d'horloge :

```
node stables(i: int)
returns (s: int; ncond: bool; (ns: int) when ncond);
```

puis déclaration d'horloges locales :

```
var cond :bool;
    (l: int) when cond;
```

puis du code lui-même :

```
let
  cond = true -> i <> pre i;
  ncond = not cond;
  l = somme(i when cond);
  s = current(l);
  ns = somme(i when ncond):
tel;
```

Les horloges et sous-horloges sont ainsi vérifiées comme suit. Les constantes sont cadencées sur l'horloge de base du noeud courant . Par défaut, les variables sont sur l'horloge de base du noeud. On a aussi $clock(e_1\ op\ e_2) = clock(e_1) = clock(e_2)$, $clock(e\ when\ c) = c$, et $clock(current(e)) = clock(clock(e))$.

Les horloges sont déclarées et vérifiées, il n'y a pas d'inférence, tout est déclaré (ou règles implicites, voir plus haut). Deux horloges sont exactes si elles sont syntaxiquement égales.

10.3 Pour aller plus loin...

On peut vérifier des *propriétés temporelles*, parlant d'événements dans le futur (« toujours dans le futur » ou « un jour dans le futur »), qui sont plus générales que les invariants de la preuve à la Hoare : « Si à un instant n, x $(=x_n)$ est positif, alors il existe un instant $m > n$ tel que pour tous les instants $k \geq m$, y $(=y_k)$ est positif ». Une approche classique repose sur le fait que ces propriétés sont codables en Lustre ! (processus « observateur » – model-checking etc.)

Un autre point qui pourra intéresser les élèves motivés, il existe un mariage entre le paradigme fonctionnel, et réactif synchrone : Lucid synchrone, voir à ce propos [12].

10.4 Réseaux de Kahn et sémantique de Lustre

À l'origine de Lustre, on trouve une machine théorique formée d'un graphe dont les noeuds traitent des informations envoyées d'autres noeuds à travers des files non-bornées, et qui renvoient sur les arcs sortant des messages à d'autres noeuds.

Cette machine théorique abstrait en quelque sorte l'échantillonnage et le traitement discret des données (automatique, traitement du signal, etc.). Il va falloir néanmoins imposer une restriction sur le traitement fait par les noeuds pour que cela ait un sens.

Le domaine sémantique que nous allons utiliser est le suivant. Le domaine des données $\mathcal{S}$ est celui des *suites* de valeurs (dans Val) finies $(x_0, \ldots, x_n)$ ou pas $(x_0, \ldots, x_n, \ldots)$. On identifiera la suite finie $(x_0, \ldots, x_n)$ avec la suite infinie à valeur dans $\mathrm{Val} \cup \{\bot\}$: $(x_0, \ldots, x_n, \bot, \ldots, \bot, \ldots)$ donc $\mathcal{S} = \{x : \mathbb{N} \to \mathrm{Val}_\bot \mid x_i = \bot \Rightarrow (\forall j \geq i, x_j = \bot)\}$.

On définit alors l'ordre partiel *préfixe* sur $\mathcal{S}$ par, pour $x, y \in \mathcal{S}$, $x \leq y$ si :

$$x_i \neq \bot \Rightarrow y_i = x_i$$

Dit de façon plus simple, x est un préfixe de y ; et l'ordre préfixe est la restriction à $\mathcal{S}$ de l'ordre défini au chapitre 6 pour $\mathbb{N} \to \mathrm{Val}_\bot$ ($\mathrm{Val}_\bot$ étant un CPO). $\mathcal{S}$ est donc un CPO (vérification triviale).

Il faut maintenant définir les fonctions aux noeuds du graphe, d'un réseau de Kahn. On veut qu'elles soient calculables, on impose donc naturellement la continuité (cf. chapitre 6). En fait, on peut se contenter ici d'imposer pour $f : \mathcal{S}^n \to \mathcal{S}^m$ la commutation aux *sup* ; celle-ci peut s'imposer coordonnée par coordonnée (on suppose ici $m = n = 1$) : pour toute ω-chaîne $x^0 \leq x^1 \leq \ldots \leq x^j \leq \ldots$ de $\mathcal{S}$,

$$f\left(\bigcup_{j\in\mathbb{N}} x^j\right) = \bigcup_{j\in\mathbb{N}} f(x^j)$$

(comme $f(x^j)$ n'est pas nécessairement croissante, il faut supposer qu'il existe un *sup* de cette suite, égale au terme de gauche, dans cette définition).

Cette condition, historique, est en fait équivalente à la « continuité » dans notre cas. En effet, pour toute ω-chaîne $x^0 \leq x^1 \leq \ldots \leq x^n \leq \ldots$ on a, pour

tout $j \in \mathbb{N}$:

$$\begin{aligned}\left(\bigcup_{i\in\mathbb{N}} f(x^i)\right)_j &= \begin{cases} f(x^k)_j & \exists k \in \mathbb{N},\ f(x^k)_j \neq \bot \\ & \text{et } \forall l \geq k,\ f(x^l)_j = f(x^k)_j \\ \bot & \text{sinon} \end{cases} \\ &= \\ f\left(\bigcup_{i\in\mathbb{N}} x^i\right)_j &= f\left(y \to \begin{cases} x^k_j & \exists k' \in \mathbb{N},\ x^{k'}_j \neq \bot \\ \bot & \text{sinon} \end{cases}\right)(x)\end{aligned}$$

Intuitivement pour j fixé, la $j$ième valeur du flot de sortie de f est déterminée par l'image par f sur un préfixe fini du flot d'entrée. La continuité est ici une sorte d'axiome de « causes finies ».

Pour mieux comprendre l'importance de la condition de continuité, donnons ici un exemple de fonction non continue sur $\mathcal{S}$. Soit $g : \mathcal{S} \to \mathcal{S}$ telle que :

$$g(x) = \begin{cases} (0,\ldots,0,\ldots) & \text{si } x \text{ est fini} \\ (1,\ldots,1,\ldots) & \text{si } x \text{ est infini} \end{cases}$$

Soit y flot infini et y_i, $i = 0, 1, \ldots$, tous ses préfixes finis : $\bigcup_{i\in\mathbb{N}} y_i = y$ mais $g(\bigcup_{i\in\mathbb{N}} y_i) = (1,\ldots)$ et $\bigcup_{i\in\mathbb{N}} g(y_i) = \bigcup_{i\in\mathbb{N}}(0,\ldots) = (0,\ldots)$.

À l'origine, Kahn demandait juste la préservation des bornes supérieures. Quid de la croissance (qui est une forme de causalité) ? Supposons que l'on ait une fonction $f : \mathcal{S} \to \mathcal{S}$ telle que pour toute ω-chaîne $x^0 \leq x^1 \leq \ldots \leq x^j \leq \ldots$,

$$f\left(\bigcup_{j\in\mathbb{N}} x^j\right) = \bigcup_{j\in\mathbb{N}} f(x^j)$$

Soit la suite $x = (x^0 = x, x^1 = y, \ldots, x^n = y, \ldots) \in \mathcal{S}$, alors $f\left(\bigcup_{j\in\mathbb{N}} x^j\right) = f(y)$ mais $\bigcup_{j\in\mathbb{N}} f(x^j) = z$ est tel que $f(x) \leq z = f(y)$ par hypothèse, on a donc la croissance.

La sémantique de Lustre peut être entièrement donnée sur le CPO introduit plus haut. En fait Lustre n'est qu'une notation pour générer un réseau de Kahn, dont les noeuds sont les équations Lustre.

Exercices

1. (Marc Pouzet) L'objectif de cet exercice est de programmer le contrôleur d'une machine à café. Il dispose des entrées suivantes :
   - café, grand café, thé : permettent de sélectionner une boisson ;
   - annuler : ce bouton permet d'annuler la commande et de vider le monnayeur si des pièces de monnaie ont été données. Cette machine à café permet de commander plusieurs boissons et ne rend la monnaie que lorsque le bouton annuler est entré ;
   - pièce : permet d'introduire des pièces de monnaie. On supposera ici que les seules pièces possibles sont des pièces de 10 centimes et 20 centimes ;
   - prêt : indique au contrôleur que la boisson demandée est prête ;

– miliseconde : est un signal vrai toutes les milisecondes.

Les sorties de cette machine sont définies ci-dessous :

– preparer : indique que la boisson demandée doit être préparée (cette information contrôle le mécanisme de fabrication) ;
– sonnerie : lorsque la boisson est prête, un signal sonore est émis ;
– boisson : indique que la boisson est en train d'être préparée ;
– monnaie : elle affiche la monnaie introduite dans la machine ;
– vider monnaie : permet de vider le monnayeur.

Le prix des consommations est le suivant :

– un café coute 40 centimes ;
– un grand café coute 1 euro ;
– un thé coute 50 centimes.

Le fonctionnement de cette machine est le suivant : le consommateur introduit des pièces dans la machine puis sélectionne sa boisson. Le voyant boisson est alors allumé. Celui-ci s'éteint et le signal sonore sonnerie est émis pendant 5 secondes. Si le consommateur appuie sur le bouton annuler avant d'avoir sélectionné sa boisson, sa monnaie lui est rendue. Si les bouton café, grand café ou thé sont enfoncés avant que l'utilisateur ait introduit sa monnaie, le signal sonnerie est émis pendant 1 seconde. Plusieurs boissons peuvent être commandées à la suite lorsqu'il y a suffisamment de monnaie.

Programmer ce contrôleur en Lustre.

Bibliographie

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, aussi disponible à* `http://www.cs.bham.ac.uk/~axj/pub/papers/handy.ps.gz`, volume 3, pages 1–168. Clarendon Press, 1994.

[2] H.P. Barendregt. *The Lambda Calculus : Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1985.

[3] Olivier Bournez. Fondement de l'informatique : logique, modèles, calculs, 2012.

[4] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml. Avec CD-Rom.* O'Reilly Editions, aussi disponible à `http://www.pps.univ-paris-diderot.fr/Livres/ora/DA-OCAML/index.html`, 2000.

[5] François Morain. Introduction à la programmation et à l'algorithmique, 2012.

[6] Thomas H. Cormen. *Algorithmique :cours avec 957 exercices et 158 problèmes*. Dunod, 2010.

[7] Claude Delannoy. *Programmer en Java*. Eyrolles, 2012.

[8] Eric Goubault et Sylvie Putot. Vérification pour les systèmes embarqués, 2012.

[9] Léo Liberti. Les bases de la programmation et de l'algorithmique, 2012.

[10] Léo Liberti. Programmation en C++, 2012.

[11] Rémy Malgouyres, Rita Zrour, and Fabien Feschet Malgouyres. *Initiation à l'algorithmique et à la programmation en C : cours avec 129 exercices corrigés*. Dunod, 2012.

[12] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006. Distribution available at : `www.lri.fr/~pouzet/lucid-synchrone`.

[13] Benjamin Werner and François Pottier. Algorithmique et programmation, 2013.

[14] Glynn Winskel. *The formal semantics of programming languages : an introduction*. MIT Press, Cambridge, MA, USA, 1993.