1 Leçon 1

1.1 Une brève introduction aux langages de programmation

Il existe de nombreux langages de programmation. En fait, en comptant les langages d'usage général, les langages de script, les langages dédiés à une application, ..., le nombre de langages existants se compte en dizaines de milliers.

Aujourd'hui, les langages les plus couramment répandus sont C, C++ et Java. Il s'agit de langages *impératifs*, c'est-à-dire où l'exécution procède par changements d'états successifs. Par exemple, la factorielle (un exemple que je reprendrai jusqu'à la nausée dans la suite de ce cours) s'écrit en C :

```
int fact (int n)
{
  int resultat;
  int i;

  resultat = 1;
  for (i=1; i<=n; i++)
    resultat = resultat * i;
  return resultat;
}
```

ce qui se lit informellement : "pour calculer la factorielle de n, mettre le résultat à 1, puis le multiplier successivement par 1, par 2, ..., par n".

À l'opposé, les langages *déclaratifs* tentent de décrire ce qu'on souhaite calculer sans décrire comment — ou, sans être aussi extrémiste, insistent sur le quoi au détriment du comment. Par exemple, la factorielle en un langage *fonctionnel* comme CaML s'écrira :

```
let rec fact n =
        if n=0
           then 1
        else n * fact (n-1);;
```

ce qui correspond bien à la notation mathématique définissant la factorielle :

$$n! = \begin{cases} 0 & \text{si } n = 0 \\ n.(n-1)! & \text{si } n \neq 0 \end{cases}$$

Un autre exemple est donné par les langages *logiques*, dont le représentant typique est Prolog, où l'on écrirait typiquement :

```
fact(0,1).
fact(N+1,Y) :- fact(N,Z), Y=(N+1)*Z.
```

(J'ai en réalité un peu triché dans l'écriture de ce programme, pour des raisons de lisibilité.) Ceci définit une relation `fact` entre deux entiers, telle que $\texttt{fact}(m, n)$ est censé être vrai si $n = m!$, et définie par le fait que $1 = 0!$, et $Y = (N+1)!$ s'il existe $Z = N!$ tel que $Y = (N+1)Z$. Je n'insisterai pas sur les langages logiques, que vous comprendrez mieux de toute façon après avoir suivi le cours de logique d'Hubert Comon.

Le but de ce cours n'est pas spécifiquement de vous apprendre à programmer dans tel ou tel langage de programmation, mais de vous apprendre quelques concepts de base communs à pratiquement tous les langages de programmation.

Un concept central est celui de *sémantique*. Plutôt que de dire de quoi il s'agit tout de suite, examinons les questions suivantes :

- Je vous donne un programme, par exemple le programme `fact` écrit en C ou en CaML comme ci-dessus. Que fait ce programme ?
  Vous pouvez vous laisser guider par une intuition plus ou moins vague de ce que fait chaque instruction du programme. Mais ceci ne mène pas toujours loin. Par exemple, sachant que, en C, x++ ajoute un au contenu de la variable x et retourne la valeur qu'avait x avant, que fait l'instruction suivante ?
  `x = x++;`
  Vous pouvez tester ce programme, et par exemple sur ma machine, ceci ajoute juste 1 à `x`. Il est probable que, si vous avez un compilateur C qui fonctionne comme le mien, vous soyez convaincus que c'est effectivement ce que doit faire cette instruction. (J'expliquerai la notion de compilateur un peu plus loin.) Mais un jour vous le ferez peut-être tourner sur une autre machine, ou avec un autre compilateur, et l'instruction ci-dessus ne changera pas la valeur de `x`. La raison en est que `x = x++;` demande essentiellement à effectuer trois opérations :

  1. lire la valeur de `x` avant, appelons-la x_0 ;
  2. réinscrire $x_0 + 1$ dans la variable `x` (l'effet de `x++`) ;
  3. mettre x_0 (la valeur de `x++`) dans `x`.

  Or les deux dernières entrent en conflit : on n'obtient pas le même résultat selon qu'on exécute 1 avant 2 ou 2 avant 1, et la norme du langage C ne précise aucun ordre entre ces actions. Plus surprenant, il n'est pas exclu qu'un compilateur décide de traduire cette instruction sous forme d'instructions assembleur qui mettent essentiellement *n'importe quoi* dans `x`.
  Savoir exactement les effets possibles d'une instruction donnée, c'est en connaître la sémantique.
- J'ai écrit deux versions de la factorielle, en C et en CaML, ci-dessus. Comment puis-je être sûr qu'ils calculent la même chose ?
  Évidemment, j'ai tout fait pour, donc ils devraient calculer la même chose... mais j'ai pu me tromper (il peut y avoir des *bogues*). Comment sais-je que je ne me suis pas trompé, ou comment puis-je trouver les bogues ? Il y a plusieurs approches : utiliser un *débogueur*, ou effectuer des campagnes de tests, permettent de détecter ou de comprendre des erreurs, du moins si ces erreurs se manifestent suffisamment fréquemment et de façon suffisamment reproductible.

On peut aussi tenter de démontrer un théorème d'équivalence des deux programmes. Ceci revient à démontrer que les deux programmes ont la *même sémantique*, ou dans les cas plus complexes, ont des sémantiques reliées par des relations ayant de bonnes propriétés (relations d'abstraction, de raffinement, bisimulations ; ce ne sera pas le programme de ce cours).

Pour ceci, on aura besoin de définir *mathématiquement* les diverses sémantiques de nos langages de programmation.

– J'ai parlé un peu plus haut de compilateurs (vers le langage assembleur). Un compilateur est un programme, qui prend le texte source d'un programme en entrée (par exemple en C ou en CaML), et le traduit en un autre langage (typiquement le langage machine, ou assembleur). Une sémantique, c'est fondamentalement pareil : c'est une fonction prenant en entrée un texte source et retournant un objet mathématique.

  Si l'on arrive à considérer les programmes (C, CaML, assembleur) comme des objets mathématiques, les compilateurs ne seront rien d'autre que des fonctions de sémantique particulières. L'avantage de cette vision sera en particulier que l'on pourra démontrer qu'un compilateur est correct par rapport à une sémantique : ceci reviendra à montrer que deux sémantiques sont équivalentes (typiquement, que ce sont deux fonctions égales).

Ce cours abordera donc des extrêmes assez éloignés :

– D'un côté nous aurons un peu de mathématiques : sémantiques dénotationnelles, opérationnelles ; théorie de l'ordre et théorèmes de points fixes, théorie de la démonstration et récurrences structurelles.
– De l'autre nous aurons à voir quelques considérations très pratiques : pour comprendre la sémantique de l'assembleur, il faudra comprendre comment fonctionne un ordinateur : processeur, mémoire, bus, portes logiques et flip-flops. C'est l'*architecture* des ordinateurs, à laquelle Béatrice Bérard vous donnera une introduction en deux séances de quatre heures.

Avant de passer à la suite, considérez le programme suivant en C, si vous comprenez déjà le C (on décortiquera ce programme en cours) :

```c
void merge (int *l1, int n1,
            int *l2, int n2,
            int *res) {
  while (n1!=0 && n2!=0) {
    if (*l1 < *l2)
       { *res++ = *l1++; n1--; }
    else { *res++ = *l2++; n2--; }
  }
  while (n1--!=0) *res++ = *l1++;
  while (n2--!=0) *res++ = *l2++;
}
```

```c
void sort1 (int *l, int n,
            int *res) {
  int k;

  if (n==0) return;
  if (n==1) { *res = *l; return; }
  k = n / 2;
  sort1 (l, k, res);
  sort1 (l+k, n-k, res+k);
  merge (res, k, res+k, n-k, l);
}
```

```c
void sort (int *l, int n) {
  int *aux = (int *) malloc (n * sizeof (int));
  sort1 (l, n, aux);
  free (aux);
}
```

Il s'agit d'un programme de tri d'un tableau `l` d'entiers, de longueur `n`, par la méthode dite de *tri par fusion*. Ce programme contient un gros bogue : lequel ? Vous pouvez le trouver par test et déboguage, ou en raisonnant formellement. Dans tous les cas, je ne pense pas que vous voyiez lequel il est en moins de cinq minutes. Ceci nous mène à l'idée qu'on aimerait disposer de méthodes automatiques de preuve de programme. Ce sera en particulier le sujet du cours d'analyse statique, une famille de techniques permettant de détecter certaines propriétés simples d'un programme, et celui des logiques de Hoare et variantes, qui seront vues plus tard dans ce cours de programmation.

Un autre exemple que je prendrai dans ce cours est le petit programme de la figure 1.

Tapez-le dans un fichier que vous nommerez `cat.c`. Il s'agit d'un autre exemple que je réutiliserai jusqu'à la nausée. Il s'agit du texte source de l'utilitaire `cat` d'Unix. (Du moins, d'une version naïve, mais qui fonctionne.) Ce programme est censé s'utiliser en tapant sous le *shell* (interprète de commandes Unix), par exemple :

```
cat a b
```

ce qui va afficher le contenu du fichier de nom `a`, suivi du contenu du fichier de nom `b`. (Vous aurez pris soin de créer deux fichiers nommés `a` et `b` d'abord, bien sûr.) En général, `cat` suivi de noms de fichiers $f_1, \ldots, f_n$, va afficher les contenus des fichiers $f_1, \ldots, f_n$ dans l'ordre ; ceci les con`cat`ène.

Si, au lieu d'utiliser l'utilitaire Unix `cat`, vous essayez d'utiliser le programme `cat.c` ci-dessus, par exemple en tapant :

```
cat.c a b
```

vous verrez que cela ne fonctionne pas. C'est parce que le *processeur* de la machine, c'est-à-dire le circuit intégré qui fait tous les calculs dans la machine en face de vous (Pentium, PowerPC, ou

```
#include <stdio.h>

main (int argc, char *argv[])
{
  int i, c;

  for (i=1; i<argc; i++)
    {
      FILE *f;

      f = fopen (argv[i], "r");
      while ((c = fgetc (f))!=EOF)
        fputc (c, stdout);
      fclose (f);
    }
  fflush (stdout);
  exit (0);
}
```

FIG. 1 – Un programme de concaténation en C

autre), ne comprend pas le texte, compréhensible aux humains (du moins on l'espère) du fichier `cat.c`.

Pour savoir ce que le processeur comprend, on peut aller voir à quoi ressemble l'utilitaire `cat`, le vrai. D'abord, on cherche où il se trouve (c'est un fichier comme un autre !), en demandant `which cat`. Sur ma machine, la réponse est `/bin/cat`. Si je charge `/bin/cat` sous XEmacs, un éditeur de texte, j'obtiens quelque chose qui ressemble à :

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@
`\211^D^H4^@^@^@x!^@^@^@^@^@^@4^@ ^@^F^@(^@^X
^@^W^@^F^@^@^@4^@^@^@4\200^D^H4\200^D^HÀ^@^@^@
À^@^@^@^E^@^@^@^D^@^@^@^C^@^@^@ô^@^@^@ô\200^D
^Hô\200^D^H^S^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A
^@^@^@^@^@^@^@^@\200^D^H^@\200^D^H3^_^@^@3^_^@
^@^E^@^@^@^@^P^@^@^A^@^@^@4^_^@^@4¯^D^H4¯^D^Ht
^A^@^@,^B^@^@^F^@^@^@^@^P^@^@^B^@^@^@^H ^@^@^H
°^D^H^H°^D^H ^@^@^@ ^@^@^@^F^@^@^@^D^@^@^@^D^@
^@^@^H^A^@^@^H\201^D^H^H\201^D^H ^@^@^@ ^@^@^@
^D^@^@^@^D^@^@^@/lib/ld-linux.so.2^@^@^D^@^@^@
^P^@^@^@^A^@^@^@GNU^@^@^@^@^@^B^@^@^@^@^@^@^@
```

etc. Peu importe de ce que cela signifie exactement pour le moment, le point important est que, si ceci est compréhensible pour le processeur, c'est loin d'être lisible pour un être humain ! A contrario, le programme `cat.c` est bien plus lisible, mais le processeur ne le comprend pas.

La traduction du fichier `cat.c` (le *source*) en `cat` (l'*exécutable*) se fait au moyen d'un *compilateur*. Par exemple, la commande `gcc` est le compilateur C, et si l'on tape :

```
gcc -o mycat cat.c
```

le compilateur `gcc` va traduire (compiler) le source `cat.c` en un exécutable que j'ai décidé de nommer `mycat`, pour ne pas prêter à confusion avec l'utilitaire standard `cat`. Vous pouvez alors lancer `mycat`, et vérifier qu'il donne bien les résultats attendus en tapant la ligne de commande suivante, et en comparant avec ce qui se passait avec le programme `cat` :

```
./mycat a b
```

(Le "`./`" avant `mycat` nous sert à dire au shell que l'outil de nom `mycat` à utiliser est celui qui est dans notre répertoire courant.)

▸ **EXERCICE 1.1**

Que se passe-t-il si vous tapez la ligne suivante ?

```
./mycat cat.c
```

Vous pouvez aussi voir comment fonctionne `mycat` en utilisant le débogueur `ddd` :

▸ **EXERCICE 1.2**

Sous Unix, avec `ddd` installé (sinon, utilisez `gdb`, mais c'est moins lisible...) :

- Recompilez `mycat` en tapant `gcc -ggdb -o mycat cat.c`. L'option `-ggdb` permettra au débogueur de vous afficher des informations pertinentes (pour plus d'information, tapez `man gcc`).
- Tapez `ddd mycat &`. La fenêtre de `ddd` s'ouvre, montrant le texte source `cat.c`. Fermez la fenêtre "Tip of the Day" si besoin est.
- Cliquez sur le début de la ligne `int i, c;` avec le bouton droit de la souris, déroulez le menu qui s'affiche, et cliquez sur "set breakpoint". Une petite icône figurant un panneau stop rouge s'affiche par-dessus la ligne.
- En plaçant la souris sur la case du bas, où s'est affiché le message :
  ```
  (gdb) break cat.c:5
  Breakpoint 1 at 0x8048536: file cat.c, line 5.
  (gdb)
  ```
  tapez `run a b`. Ceci lance le programme `mycat`, avec arguments les chaînes de caractères `a` et `b`. Le programme s'arrête sitôt lancé sur le "breakpoint" que l'on a mis ci-dessus.
- Tirez le menu "View", et cliquez sur "Data Window" ; cliquez ensuite sur les menus "Data/Display Arguments" et "Data/Display Local Variables". Pour voir les arguments ("a", "b") qui ont été passés au programme `mycat`, cliquez sur le menu "Data/Memory...", et écrivez 3 en face de "Examine", puis "string" au lieu de "octal", enfin tapez `argv[0]` dans la case après "from", puis cliquez sur les boutons "Display" puis "Close". Vous aurez probablement besoin de réorganiser le cadre du haut de la fenêtre de `ddd` pour bien voir les affichages "Args", "Locals" et "X".

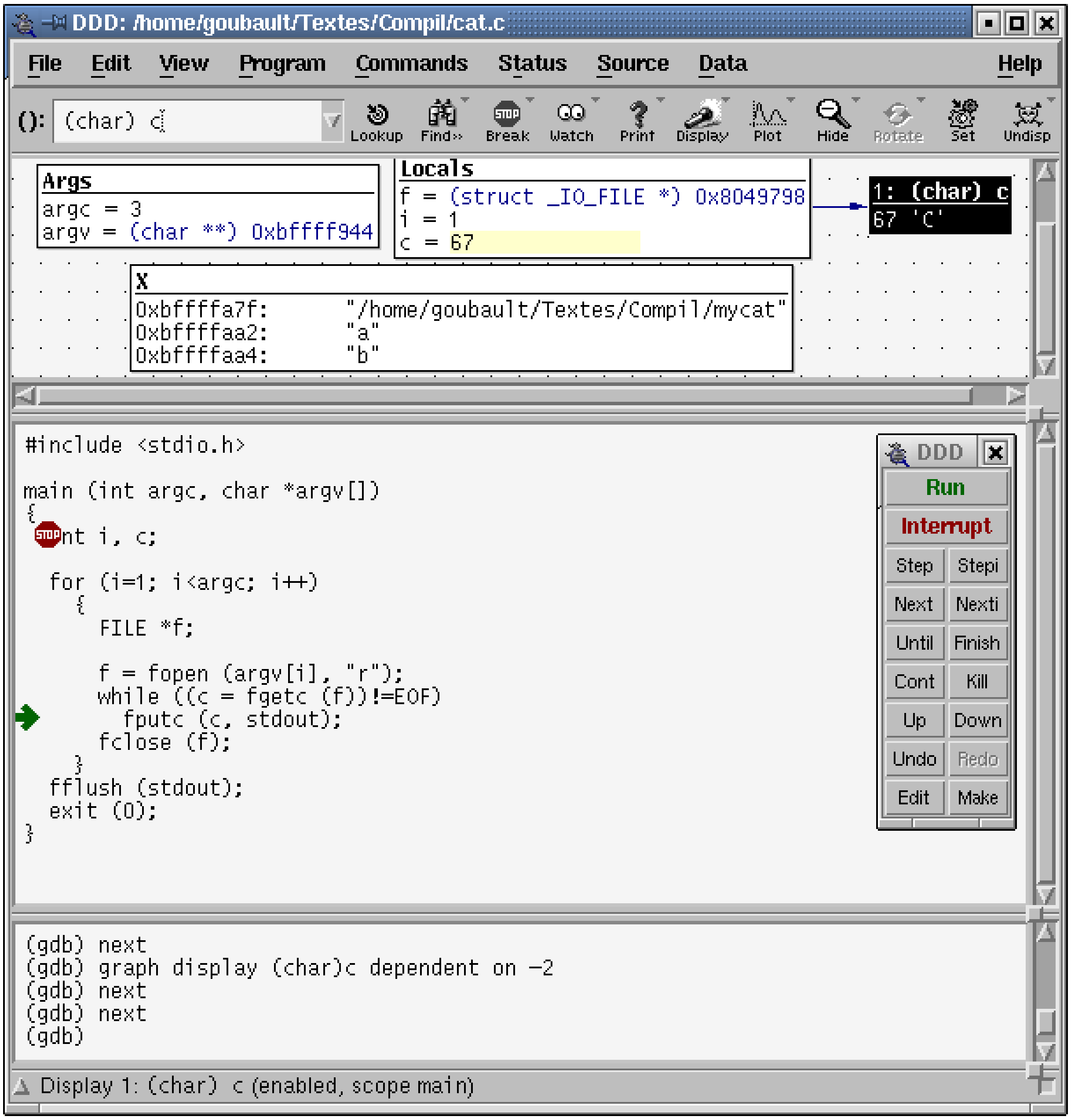


FIG. 2 – Une session sous ddd

– Cliquez ensuite sur le bouton “Next” de la fenêtre flottante “DDD” répétitivement pour voir le programme s’exécuter petit à petit. Si vous voulez voir le contenu de la variable `c` sous forme d’un caractère lisible plutôt que sous forme d’un entier, cliquez avec le bouton droit sur la ligne de `c` dans l’affichage “Locals”, tirez le menu local “Display/Others...”, écrivez `(char)c` dans le cadre sous “Display Expression”, puis cliquez sur “Display”.

Vous devriez obtenir quelque chose qui ressemble à la figure 2 ; Vous pouvez bien sûr effectuer la même manipulation avec d’autres programmes, y compris ceux de factorielle ou de tri décrits plus hauts.

1.2 Quelques bases de théorie de l’ordre

Pour vous reposer des aspects pratiques de la section précédente, on va faire ici un peu de mathématiques... mathématiques qui nous serviront dans la suite.

1.2.1 Points fixes et boucles

Sans dire tout de suite ce qu’est une sémantique, imaginons qu’il s’agit d’une fonction $[\![\_]\!]$ qui à chaque programme P associe une valeur $[\![P]\!]$ dans un certain domaine de *valeurs* D. Un des intérêts de cette approche, qui s’appelle la *sémantique dénotationnelle* et qui date de la fin des années 1960, c’est qu’en principe on pourra écrire et prouver des théorèmes de la forme :

$$[\![P_1]\!] = [\![P_2]\!] \qquad \text{(“les programmes } P_1 \text{ et } P_2 \text{ font la même chose”)}$$

ou bien

$$[\![P_1]\!] \neq [\![P_2]\!] \qquad \text{(“les programmes } P_1 \text{ et } P_2 \text{ ne font pas la même chose”)}$$

par exemple.

L’un des problèmes centraux à résoudre dans une telle approche sera de donner une sémantique aux *boucles*. Considérons en effet la boucle `while` du langage C. Cette construction prend la forme

```
while (b) e
```

où `b` est une expression booléenne (retournant vrai ou faux), et `e` est un bloc d’instructions écrites en C. Une telle boucle fait les opérations suivantes :

1. Calculer `b`.
2. Si le résultat vaut vrai, alors exécuter les instructions de `e`, et revenir à l’étape 1.
3. Sinon, sortir de la boucle.

Par exemple, la boucle

```
while (n1!=0 && n2!=0) {
  if (*l1 < *l2)
    { *res++ = *l1++; n1--; }
  else { *res++ = *l2++; n2--; }
  }
```

de la fonction `merge` teste si `n1` et `n2` sont tous les deux non nuls ; si c'est le cas, les trois lignes suivantes sont exécutées, puis l'on revient au début de la boucle ; sinon, la boucle est terminée. Un autre exemple est donné par la boucle

```
while ((c = fgetc (f))!=EOF)
  fputc (c, stdout);
```

du programme `cat.c`. La condition booléenne `b` est `(c = fgetc (f))!=EOF`, qui lit le caractère suivant dans le fichier `f` par `fgetc (f)`, le stocke dans la variable `c` (`c = fgetc (f)`), puis retourne vrai si et seulement si ce caractère est différent du marqueur de fin de fichier `EOF`. Si ce test réussit, autrement dit si l'on n'est pas encore à la fin du fichier `f`, l'instruction `e`, à savoir `fputc (c, stdout);`, est exécutée : elle affiche le caractère `c` sur le fichier `stdout` (la *sortie standard*), c'est-à-dire (en général) sur votre écran.

Maintenant, il semble intuitif que `while (b) e` devrait faire la même chose que le programme

```
if (b)
  { e; while (b) e }
```

qui teste d'abord si `b` est vrai, et si c'est le cas exécute `e` puis la boucle proprement dite. (On dit qu'on a *déroulé la boucle* un coup.) Ceci se manifeste par le fait que l'on souhaiterait que l'équation suivante soit vraie :

$$[\![\texttt{while}\,(\texttt{b})\ \texttt{e}]\!] = [\![\texttt{if}\,(\texttt{b})\ \{\texttt{e};\ \texttt{while}\,(\texttt{b})\ \texttt{e}\ \}]\!]$$

Notons x l'inconnue $[\![\texttt{while}\,(\texttt{b})\ \texttt{e}]\!]$. Il est intuitivement naturel de penser que ceci signifie que x et `if (b) { e;` x`}` devraient avoir la même sémantique. Si l'on note F la fonction qui à tout programme x associe `if (b) { e;` x`}`, on voit que nous cherchons x sous la forme d'un *point fixe* de F : x et $F(x)$ doivent avoir la même sémantique.

Cet argument n'est pas très rigoureux. Plus formellement, étendons la syntaxe des programmes de sorte qu'ils puissent contenir des *paramètres* X_1, …, X_k dénotant des programmes non encore explicités. Appelons *contexte* tout k-uplet $(v_1, \ldots, v_k)$ de valeurs (éléments de D). On peut imaginer étendre la fonction de sémantique $[\![\_]\!]$ à une fonction prenant un programme P à k paramètres, et un contexte $(v_1, \ldots, v_k)$, et retournant une valeur $[\![P]\!]\,(v_1, \ldots, v_k)$, dénotant intuitivement la valeur du programme P lorsque X_1 vaut v_1, …, X_k vaut v_k. (Cette intuition se formalise par l'équation $[\![P]\!]\,([\![Q_1]\!], \ldots, [\![Q_k]\!]) = [\![P[X_1 \leftarrow Q_1, \ldots, X_k \leftarrow Q_k]]\!]$, où $\leftarrow$ dénote la substitution.) Supposons $k = 1$ pour simplifier. Soit F la fonction qui à $v \in D$ associe $[\![\texttt{if}\,(\texttt{b})\,\{\texttt{e};\ X_1\ \}]\!]\,(v)$. Le déroulement de boucle revient à demander que $x = F(x)$, ce qui est l'énoncé précis du fait que x est un point fixe.

Un autre exemple où l'on aura besoin de la notion de point fixe est dans la définition de fonctions récursives. Considérons le calcul de factorielle 7 en Caml :

```
let rec fact n =
        if n=0
           then 1
        else n * fact (n-1)
in fact 7;;
```

On voit que `fact` est une fonction qui est définie en termes d'elle-même : la définition de `fact`, après le premier signe =, fait appel à `fact` (que l'on applique à `n-1`). (Le fait que `fact` soit définie en termes d'elle-même est la raison d'être du mot-clé `rec`, qui instruit le compilateur Caml de ce fait.) C'est peut-être un peu plus clair si l'on réécrit l'expression Caml ci-dessus en :

```
let rec fact =
        fun n -> if n=0
                    then 1
                 else n * fact (n-1)
in fact 7;;
```

où la construction `fun n -> [...]` dénote la fonction qui à `n` associe [...]. Remplaçons `fact` par l'inconnue x dans sa définition, et soit F la fonction qui au programme $\underline{x}$ associe le programme

```
fun n -> if n=0
            then 1
         else n * x (n-1)
```

On a donc défini `fact` comme étant un point fixe de F.

Les deux exemples des boucles `while` et des fonctions récursives ne sont pas si éloignés qu'il y paraît. La boucle `while (b) e` peut en effet se coder en CaML sous la forme :

```
let rec boucle () =
        if (b)
           then (e; boucle ())
        else ()
in
   boucle ();;
```

Rappelez-vous comment on calcule le point fixe d'une fonction contractante F de $\mathbb{R}$ dans $\mathbb{R}$: partant de n'importe quel point x_0, on calcule les itérés $F^n(x_0)$ (où $F^0(x) = x$, $F^{n+1}(x) = F(F^n(x))$), et la limite est l'unique point fixe de F. Nous ne pourrons pas supposer que nous sommes dans un espace métrique complet dans la suite, ni que la fonction F qui à un programme x associe `if (b) { e;` x`}` est contractante, mais nous tenterons de calculer la sémantique de la boucle comme une sorte de limite des itérés $F^n(x_0)$ pour une certaine valeur x_0. Observons que cela revient à dire que la sémantique de `while (b) e` est la "limite" de

$$\begin{gathered} x_0 \\ \texttt{if (b) \{ e; } x_0\} \\ \texttt{if (b) \{ e; if (b) \{ e; } x_0\}\} \\ \texttt{if (b) \{ e; if (b) \{ e; if (b) \{ e; } x_0\}\}\} \\ \ldots \end{gathered}$$

Le problème de base va être de trouver des catégories de domaines de valeurs où toute fonction (raisonnable) aura au moins un point fixe.

1.2.2 Treillis complets et théorème de Tarski

L'une des catégories les plus simples où l'on aura un théorème du point fixe est celle des treillis complets et des fonctions monotones.

Rappelons qu'une relation d'*ordre* $\leq$ sur un ensemble X est une relation binaire réflexive ($x \leq x$), antisymétrique ($x \leq y$ et $y \leq x$ impliquent $x = y$), et transitive ($x \leq y$ et $y \leq z$ impliquent $x \leq z$). Un *ensemble ordonné* est un couple $(X, \leq)$ formé d'un ensemble X et d'une relation d'ordre $\leq$ sur X. On notera souvent X au lieu de $(X, \leq)$, par abus de langage. Une fonction $f : X \to Y$ sera dite *monotone* si et seulement si elle préserve l'ordre : si $x \leq x'$, alors $f(x) \leq f(x')$.

Rappelons aussi qu'un *majorant* y d'une partie F de X est un élément supérieur ou égal à tout élément de X : $x \leq y$ pour tout $x \in X$. Un *minorant* y est inférieur ou égal à tout élément de X : $y \leq x$ pour tout $x \in X$. La borne supérieure de F est le plus petit des majorants de F dans X, si elle existe ; elle est alors nécessairement unique. De même, la borne inférieure de F est le plus grand des minorants de F dans X, si elle existe ; elle est alors nécessairement unique.

Définition 1 (Treillis complet) *Un* treillis complet *est un ensemble ordonné $(X, \leq)$ non vide tel que toute famille $F \subseteq X$ a une borne supérieure $\bigvee F$ et une borne inférieure $\bigwedge F$.*

Si $F = \emptyset$, $\bigwedge \emptyset$ est par définition le plus grand élément de X, et sera noté $\top$; $\bigvee \emptyset$ est le plus petit élément de X, et sera noté $\perp$. On notera aussi $\bigvee_{i \in I} x_i$ la borne supérieure de la famille $(x_i)_{i \in I}$, et de même $\bigwedge_{i \in I} x_i$ sa borne inférieure. On notera aussi $x \vee y = \bigvee\{x, y\}$ et $x \wedge y = \bigwedge\{x, y\}$. Pour faire court, on dira aussi "le sup" au lieu de "la borne supérieure", et "l'inf" pour "la borne inférieure".

▸ **EXERCICE 1.3**

Un *inf-demi-treillis complet* est un ensemble ordonné $(X, \leq)$ non vide tel que toute famille $F \subseteq X$ a une borne inférieure $\bigwedge F$. Montrer que toute famille $F \subseteq X$ a une borne supérieure $\bigvee F$, et que donc tout inf-demi-treillis complet est un treillis complet. (Indication : $\bigvee F$ s'il existe est le plus petit des majorants... comment peut-on donc le construire ?)

▸ **EXERCICE 1.4**

Montrer que tout sup-demi-treillis complet (notion que vous définirez par analogie avec la précédente) est un treillis complet.

▸ **EXERCICE 1.5**

Soit A un ensemble quelconque, $\mathbb{P}(A)$ l'ensemble de ses parties. Montrer que $(\mathbb{P}(A), \subseteq)$ est un treillis complet.

▸ **EXERCICE 1.6**

Par l'exercice 1.4, l'ensemble des ouverts $\mathcal{O}$ d'un espace topologique $(X, \mathcal{O})$ est un treillis complet : le sup d'une famille F d'ouverts est juste leur union. Quel est son inf ?

Le point important est que toute fonction monotone d'un treillis complet dans lui-même a un point fixe :

Théorème 1 (Tarski-Knaster) *Soit $(X, \leq)$ un treillis complet. Soit $f : X \to X$ une fonction monotone. Alors f a un plus petit point fixe $lfp(f)$ et un plus grand point fixe $gfp(f)$. De plus l'ensemble $Fix(f)$ de tous les points fixes de f, ordonnés par $\leq$, est treillis complet.*

Avant d'en faire la démonstration, considérons l'intuition donnée à la fin de la section 1.2.1. Partant de $\perp$, on va calculer $f(\perp)$, $f^2(\perp)$, ..., $f^n(\perp)$, ... Le sup de cette famille, $\bigvee_{n\in\mathbb{N}} f^n(\perp)$, vérifie :

$$\begin{aligned} f(\bigvee_{n\in\mathbb{N}} f^n(\perp)) &\geq f(f^n(\perp)) \qquad \text{(pour tout } n\text{)} \\ &= f^{n+1}(\perp) \end{aligned}$$

donc $f(\bigvee_{n\in\mathbb{N}} f^n(\perp)) \geq \bigvee_{n\in\mathbb{N}} f^n(\perp)$, mais l'inégalité inverse ne tient pas en général (voir exercice 1.10). On pourrait utiliser cette forme d'argument en itérant f de façon transfinie, mais ceci nécessiterait que nous parlions d'ordinaux... et ce n'est pas un cours de théorie des ensembles.

Démonstration. Considérons l'ensemble $Post(f) = \{x \in X | f(x) \leq x\}$ des *post-points fixes* de f. D'abord remarquons que $Post(f)$ est non vide, car $\top$ est dans $Post(f)$; ceci n'a pas en réalité beaucoup d'importance.

Puisque X est un treillis complet, $Post(f)$ a une borne inférieure ; appelons-la x_0. Comme x_0 est un minorant de $Post(f)$, $x_0 \leq x$ pour tout $x \in Post(f)$. Comme f est monotone, $f(x_0) \leq f(x)$, et comme $x \in Post(f)$, $f(x) \leq x$: donc $f(x_0) \leq x$. Ceci étant vrai pour tout $x \in Post(f)$, $f(x_0)$ est un minorant de $Post(f)$. Comme x_0 est le plus grand de tous ces minorants, $f(x_0) \leq x_0$. Mais ceci, par définition, signifie que x_0 est dans $Post(f)$, et est donc le *plus petit post-point fixe* de f.

Revenons sur le fait que $f(x_0) \leq x_0$. Par monotonie de f encore, $f(f(x_0)) \leq f(x_0)$, donc $f(x_0)$ est aussi un post-point fixe de f. Comme x_0 est le plus petit, $x_0 \leq f(x_0)$. Donc $x_0 = f(x_0)$ par antisymétrie. Ceci montre que x_0 est un point fixe de f.

C'est nécessairement le plus petit de tous les points fixes : tout autre point fixe x de f est clairement un post-point fixe de f, et est donc supérieur ou égal au plus petit post-point fixe x_0.

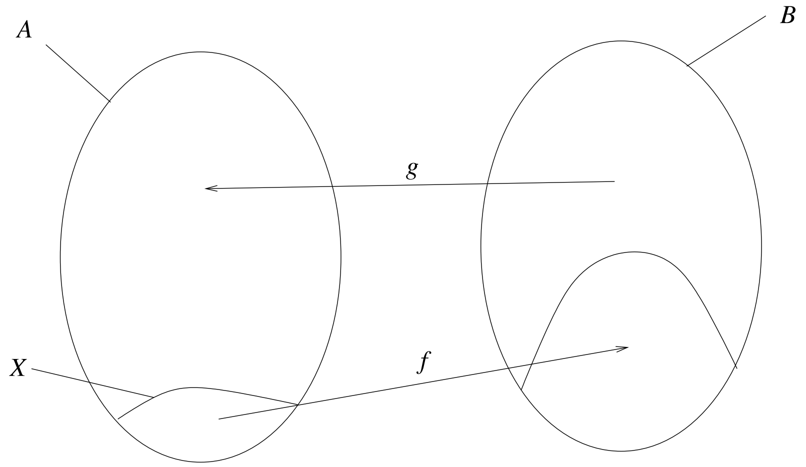
La fin de la démonstration est laissée en exercice. □

▸ **EXERCICE 1.7**

Terminez la démonstration du théorème de Knaster-Tarski. (Attention : le sup dans X d'une famille de points fixes de f n'est pas nécessairement un point fixe de f, et il faudra donc construire le sup *dans* $Fix(f)$ d'une famille de points fixes de f légèrement différemment ; indication : pensez post-points fixes...)

▸ **EXERCICE 1.8**

Démontrez le théorème de Cantor-Schröder-Bernstein en utilisant le théorème de Knaster-Tarski : si f est une injection de A dans B et g une injection de B dans A, alors il existe une bijection entre A et B. Indication : trouvez une partie X de A vérifiant certaines propriétés suggérées par la figure ci-dessous, en vous souvenant que $(\mathbb{P}(A), \subseteq)$ est un treillis complet (exercice 1.5).



▸ **EXERCICE 1.9**
Soit $f : Y \times X \to X$ une fonction monotone, au sens où $y \leq y'$ et $x \leq x'$ impliquent $f(y, x) \leq f(y', x')$. Par le théorème de Knaster-Tarski, $lfp(f(y, \_))$ existe pour tout y, où $f(y, \_)$ dénote la fonction monotone qui à x associe $f(y, x)$. Montrer que la fonction qui à y associe $lfp(f(y, \_))$ est monotone de Y dans X. (Indication : utiliser la caractérisation du plus petit point fixe comme plus petit post-point fixe.)

▸ **EXERCICE 1.10**
Soit X l'intervalle $[0, 1]$ de la droite réelle, muni de son ordre naturel $\leq$. Montrer que X est un treillis complet. Si f est une fonction monotone de X dans X, montrer que $f(\bigvee_{n\in\mathbb{N}} x_n) = \bigvee_{n\in\mathbb{N}} f(x_n)$ pour toute suite croissante $(x_n)_{n\in\mathbb{N}}$ si et seulement si f est continue à gauche sur $[0, 1]$. En choisissant judicieusement f, en déduire que $f(\bigvee_{n\in\mathbb{N}} f^n(\bot))$ est en général différent de $\bigvee_{n\in\mathbb{N}} f^n(\bot)$.

1.2.3 Cpos, fonctions Scott-continues

Historiquement, Dana Scott avait proposé d'utiliser les treillis complets comme domaines de valeurs D servant à donner des sémantiques aux langages de programmation. Une notion importante découverte par D. Scott est que toutes les fonctions calculables de D dans D sont non seulement monotones mais encore *continues* (au sens de Scott, voir plus loin). Plus tard, d'autres, et notamment Gordon Plotkin, se sont aperçus que l'on pouvait se passer de treillis complets, et utiliser des ensembles ordonnés plus généraux : les *cpos* ("complete partial order").

Définition 2 (Famille dirigée, fonctions Scott-continues) *Soit $(X, \leq)$ un ensemble ordonné. Une famille $D \subseteq X$ est dite* dirigée *si et seulement si :*
- *D est non vide ;*
- *et pour tous x, y dans D, x et y ont un majorant dans D.*

Une fonction $f : X \to Y$ est Scott-continue *si et seulement si, pour toute famille dirigée D qui a un sup $\bigvee D$, la famille $f(D)$ a un sup et $f(\bigvee D) = \bigvee f(D)$.*

Cette définition est une généralisation, qui s'est avérée naturelle, de ce que nous voulons vraiment représenter. Reprenons la construction du plus petit point fixe sous forme de $\bigvee_{n\in\mathbb{N}} f^n(\bot)$, construction qui avait échoué en section 1.2.2. La famille $(f^n(\bot))_{n\in\mathbb{N}}$ est dirigée, d'abord : c'est facile, et c'est le contenu de l'exercice 1.11 ci-dessous. Si cette famille a un sup $\bigvee_{n\in\mathbb{N}} f^n(\bot)$, la Scott-continuité de f est exactement l'hypothèse supplémentaire dont nous avions besoin pour montrer que $\bigvee_{n\in\mathbb{N}} f^n(\bot)$ est le plus petit point fixe de f : c'est le lemme 2.

▸ **EXERCICE 1.11**
Soit $(x_n)_{n\in\mathbb{N}}$ une suite croissante quelconque dans $(X, \leq)$. Autrement dit, $x_n \leq x_{n+1}$ pour tout $n \in \mathbb{N}$. Montrer que $(x_n)_{n\in\mathbb{N}}$ forme une famille dirigée. Si $f : X \to X$ est une fonction monotone, montrer que $(f^n(\bot))_{n\in\mathbb{N}}$ est une suite croissante, et forme donc une famille dirigée.

Lemme 2 *Soit f une fonction Scott-continue de X dans X, et supposons que X a un plus petit élément $\bot$. Si $\bigvee_{n\in\mathbb{N}} f^n(\bot)$ existe, c'est le plus petit point fixe de f.*

Démonstration. C'est un point fixe :

$$\begin{aligned} f(\bigvee_{n\in\mathbb{N}} f^n(\bot)) &= \bigvee_{n\in\mathbb{N}} f(f^n(\bot)) \qquad \text{(par Scott-continuité)} \\ &= \bigvee_{n\in\mathbb{N}} f^{n+1}(\bot) = \bigvee_{n\in\mathbb{N}} f^{n+1}(\bot) \vee \bot \qquad \text{(trivialement)} \\ &= \bigvee_{n\in\mathbb{N}} f^n(\bot) \end{aligned}$$

C'est le plus petit : supposons en effet que x est un point fixe, et commençons par montrer que $f^n(\bot) \leq f^n(x)$ pour tout n, par récurrence sur n ; c'est évident si $n = 0$, et par la monotonie de f dans le cas récurrent. On en déduit que $\bigvee_{n\in\mathbb{N}} f^n(\bot) \leq \bigvee_{n\in\mathbb{N}} f^n(x)$, or $\bigvee_{n\in\mathbb{N}} f^n(x) = \bigvee_{n\in\mathbb{N}} x = x$ puisque $f(x) = x$. Donc $\bigvee_{n\in\mathbb{N}} f^n(\bot) \leq x$. □

L'intuition derrière la notion de Scott-continuité est qu'un programme *calcule* une valeur, et qu'en particulier une boucle devrait *calculer* sa valeur en tant que point fixe. La construction du théorème de Knaster-Tarski est non constructive. L'intérêt de la construction du lemme 2 est que, en identifiant sup et limite, le plus petit point fixe est construit comme une limite d'itérés finis $f^n(\bot)$ de la fonction f, ce qui correspond à l'intuition que nous avons donnée à la fin de la section 1.2.1 sur la façon dont on pouvait imaginer calculer des points fixes.

Maintenant, $\bigvee_{n\in\mathbb{N}} f^n(\bot)$ existe toujours dans un treillis complet. Mais, alors que dans un treillis complet, les sups de n'importe quelle famille existent toujours, ici nous n'avons besoin que de sups de familles dirigées.

Définition 3 (Cpo) *Un* ordre partiel complet*, ou* cpo *("complete partial order") est un ensemble ordonné $(X, \leq)$ dans lequel toute famille dirigée a un sup. Un* cpo pointé *est un cpo ayant un élément minimal $\bot$.*

Corollaire 3 *Toute fonction continue f d'un cpo pointé X dans lui-même a un plus petit point fixe, qui est le sup des $f^n(\bot)$, $n \in \mathbb{N}$.*

On pourra s'interroger sur l'opportunité d'appeler une fonction f continue lorsqu'elle préserve les sups (vus comme limites). Il se trouve qu'il s'agit réellement de la notion de continuité usuelle vue en topologie générale.

Définition et lemme 4 (Topologie de Scott) *Soit* $(X, \leq)$ *un ensemble ordonné. Soit* $\mathcal{O}$ *l'ensemble des parties* O *de* X *telles que :*

- O *est* clos par le haut *: pour tout* x *dans* O*, si* $x \leq y$ *alors* y *est dans* O *;*
- *et* O *est* inaccessible par le bas *: pour toute famille dirigée* D *de* X *dont le sup existe et est dans* O*,* D *rencontre* O*, c'est-à-dire* $D \cap O \neq \emptyset$*.*

Alors $\mathcal{O}$ *est une topologie sur* X*, appelée la* topologie de Scott. *Les éléments de* $\mathcal{O}$ *sont appelés les* ouverts de Scott *de* X*.*

La propriété d'inaccessibilité par le bas dit, dans le cas où D est une suite croissante $(x_n)_{n\in\mathbb{N}}$, que si le sup (la "limite") des x_n est dans D, alors l'un des x_n est dans D. Par contraposée, toute suite croissante hors de O a un sup (limite) hors de O, ce qui est une façon de dire que le complémentaire de O est fermé, dans un sens intuitif.

Démonstration. D'abord, la partie vide et X lui-même sont dans $\mathcal{O}$, clairement.

Ensuite, si $(O_i)_{i\in I}$ est une famille (possiblement infinie) d'ouverts de Scott, on prétend que $\bigcup_{i\in I} O_i$ est un ouvert de Scott : $\bigcup_{i\in I} O_i$ est clairement clos par le haut, et si D est une famille dirigée dont le sup existe et est dans $\bigcup_{i\in I} O_i$, alors $\bigvee D$ est dans un O_i, donc D rencontre O_i, ce qui entraîne que D rencontre $\bigcup_{i\in I} O_i$

Finalement, si O_1 et O_2 sont deux ouverts de Scott, montrons que leur intersection en est encore un. Clairement $O_1 \cap O_2$ est clos par le haut. Pour toute famille dirigée D dont le sup existe et telle que $\bigvee D$ est dans $O_1 \cap O_2$, alors $\bigvee D$ est dans O_1 et dans O_2, donc il existe deux éléments $x_1 \in D \cap O_1$ et $x_2 \in D \cap O_2$. Comme D est dirigée, x_1 et x_2 ont un majorant x dans D, qui est dans $O_1 \cap O_2$ parce que O_1 et O_2 sont clos par le haut. Donc D rencontre $O_1 \cap O_2$. □

▸ **EXERCICE 1.12**

Pour tout x dans X, notons $\downarrow x$ l'ensemble de tous les $x' \leq x$ dans X. Montrer que $\downarrow x$ est un fermé de Scott. (On rappelle qu'un *fermé* est par définition le complémentaire d'un ouvert, et non une partie non ouverte.)

▸ **EXERCICE 1.13**

Soit f une fonction monotone de l'ensemble ordonné $(X, \leq)$ dans l'ensemble ordonné $(Y, \leq)$. Démontrer que f est Scott-continue si et seulement si f est continue pour la topologie de Scott, autrement dit si et seulement si l'image réciproque de tout ouvert de Scott est encore un ouvert de Scott. (Pour la direction seulement si, on montrera d'abord que l'image d'une famille dirigée par une fonction monotone est dirigée. Pour la direction si, plus difficile, on considérera l'ensemble F, intersection des $\downarrow y$ lorsque y parcourt l'ensemble des majorants de $f(D)$, et on démontrera que F est un fermé tel que $f^{-1}(F)$ contient D.)

Dans la suite, on ne dira plus "Scott-continu", mais simplement "continu", l'exercice 1.13 montrant qu'il n'y a en fait aucune ambiguïté.

▸ **EXERCICE 1.14**
Montrez que la topologie de Scott sur $(X, \leq)$ n'est en général pas séparée (ou Hausdorff, ou T_2 : pour tous $x \neq x'$, il existe un ouvert O contenant x et un ouvert O' contenant x' d'intersection vide), en ce sens que si elle est séparée, alors tous les éléments de X sont incomparables pour $\leq$. Montrez en revanche que toute topologie de Scott est T_0, c'est-à-dire que pour tous $x \neq x'$, il existe un ouvert de Scott contenant x mais pas x', ou bien contenant x' mais pas x. (Utilisez l'exercice 1.12.)

▸ **EXERCICE 1.15**
Étant donné un espace topologique $(X, \mathcal{O})$, son *préordre de spécialisation* $\preceq$ est défini par : pour tous x, x' dans X, $x \preceq x'$ si et seulement si, pour tout ouvert O contenant x, O contient x'. Montrer qu'il s'agit bien d'un préordre, c'est-à-dire d'une relation réflexive et transitive. Si de plus la topologie $\mathcal{O}$ est T_0, alors il s'agit d'une relation d'ordre.

▸ **EXERCICE 1.16**
Montrer que l'ordre de spécialisation (cf. exercice 1.15) de la topologie de Scott d'un ensemble ordonné $(X, \leq)$ quelconque est l'ordre $\leq$ lui-même. (Pour l'un des deux sens de l'implication à démontrer, pensez à utiliser l'exercice 1.12.)

L'exercice suivant montre en quoi les sups sont une bonne façon de parler de limites dans les cpos.

▸ **EXERCICE 1.17**
On dit que x est une *limite* de la famille dirigée D dans l'espace topologique X si et seulement si, pour tout ouvert O contenant x, il existe y dans D tel que pour tout $z \geq y$ dans D, z est dans O. Montrer que le sup de D s'il existe est une limite de D, pour la topologie de Scott. En vous aidant de l'exercice 1.16, montrer que le sup de D s'il existe est en fait la *plus grande limite* de D. (On rappelle à ceux que cette formulation étonnerait que le théorème d'unicité des limites n'est valable que dans les espaces séparés...)