1 Premiers pas : arbres, constructions inductives, relations

1.1 Listes, arbres, entiers — bref : arbres

1.1.1 Listes, en Caml, en Coq

Voici des programmes simples en Caml pour manipuler les listes :

```
let rec size l = match l with
  | [] -> 0
  | x::xs -> 1 + size xs

let rec append l1 l2 = match l1 with
  | [] -> l2
  | x::xs -> x::(append xs l2)

let rec reverse l = match l with
  | [] -> []
  | x::xs -> append (reverse xs) [x]

let rec sort l = ...
```

On remarque qu'il est courant dans un langage comme Caml de ne pas utiliser de parenthèses lorsqu'on appelle une fonction en lui passant son argument : ainsi, comprenez ce que veut dire `1 + size xs` dans la définition de `size`.

Dans la définition de `append`, l'écriture `x::(append xs l2)` désigne la liste qui est construite en ajoutant `x` en tête de la liste obtenue lorsque l'on fait l'appel `append xs l2` ; celui-ci se lit comme "`append` appliqué à `xs`, le tout appliqué à `l2`". Dans de nombreux langages de programmation, on définirait sans doute la fonction `append(l1,l2)`, et l'appel récursif s'écrirait `append(xs,l2)` — on revient sur ce point plus tard.

Les programmes ci-dessus s'écrivent de manière sensiblement équivalente en Coq :

```
Require Export List.

Fixpoint size (l:list nat) :=
match l with
| nil => 0
| x::xs => 1 + size xs
end.

Fixpoint append (l1 l2:list nat) :=
match l1 with
| nil => nil
| x::xs => x::(append xs l2)
end.

Fixpoint rev (l:list nat) :=
match l with
| nil => nil
| x::xs => append (rev xs) (x::nil)
end.

Fixpoint sort (l:list nat) := ...
```

On peut ensuite démontrer en Coq des propriétés sur ces fonctions, comme

```
Lemma size_reverse : forall l:list nat, size (rev l) = size l.

Lemma sort_does_sort : forall l:list nat, sorted (sort l).
```

1.1.2 Définitions inductives

Les listes d'entiers. En Caml, on peut redéfinir les listes d'entiers ainsi (*redéfinir* parce que Caml connaît déjà une notion de liste, qui est primitive) :

```
type ilist = Nil | Cons of (int*ilist)
```

Cette définition se lit *"il y a exactement deux manières de fabriquer une `ilist` ; `Nil` est une `ilist`, et, si on a un couple formé d'un entier et d'une `ilist`, en appliquant `Cons` à ce couple, ça donne une `ilist`."*

On définit ainsi un nouveau type, `ilist`, que l'on peut interpréter comme l'ensemble de toutes les listes finies d'entiers. `Nil` et `Cons` sont appelés les *constructeurs* du type `ilist`.

Les gens qui connaissent Caml savent qu'on écrit plutôt habituellement `3::l1` pour fabriquer la liste commençant par `3` et continuant comme `l1`. (Si `l1` a le type `int list`, alors `3::l1` a le type `int list`, qui est différent de `ilist`.)

En Coq, la définition des listes d'entiers est similaire :

```
Inductive ilist : Set := Nil : ilist | Cons : int -> ilist -> ilist.
```

La définition de `Nil` dit comme ci-dessus "`Nil` est une `ilist`". La définition de `Cons` dit "`(Cons k l)` est une `ilist` à partir du moment où `k` est un `int` et `l` une `ilist`".

Point important : le type de `Cons` donné ici se lit `int -> (ilist -> ilist)`, c'est le type d'une fonction qui, lorsqu'on lui passe un `int`, *renvoie une fonction* de type `ilist -> ilist`. Ainsi, `(Cons k l)`, qui se lit `((Cons k) l)`, a bien pour type `ilist` dans les hypothèses ci-dessus.

On pourrait changer la définition et écrire `Cons : (int*ilist) -> ilist`, pour coller davantage à l'approche Caml. Trois remarques à ce sujet :

- les types `int -> ilist -> ilist` et `(int*ilist) -> ilist` sont équivalents. On dit que le premier est la version *curryfiée*, le second la version *décurryfiée* ;
- on préfère l'approche curryfiée en Coq, car cela facilite les preuves ;
- en Caml, on n'a pas le choix (pourquoi ?), on opte pour l'approche décurryfiée.

Injectivité. Les constructeurs sont injectifs, au sens où `Nil` est différent de `Cons k l` quels que soient `k` et `l`, et, si `Cons k1 l1 = Cons k2 l2`, alors `k1 = k2` et `l1 = l2`.

Types de données inductifs. Renommons le type `ilist` (en Coq) et ses constructeurs, cela donne :

```
Inductive t : Set := N : t | C : int -> t -> t.
```

Dans sa forme générale, une *définition inductive d'ensemble* consiste à donner un nombre fini de constructeurs, chacun avec son type. Les constructeurs, par définition "fabriquent du `t`". Chaque constructeur prend zéro (comme `N`) ou plusieurs arguments (comme `C`). Ces arguments peuvent être pris dans des ensembles qui pré-existent à la définition de `t`, comme ici `int`, ou peuvent être pris dans `t` lui-même.

Considérons deux variantes :

```
Inductive t : Set := N : t  |  C : int -> t-> t -> t.

Inductive t : Set := N : t  |  C : t -> t.
```

Le premier type décrit des arbres binaires, avec des entiers aux nœuds, et rien aux feuilles.

Le second type est isomorphe au type des entiers naturels, les entiers de Peano, dont les constructeurs s'appellent plutôt zéro et successeur.

1.1.3 Raisonnements par induction

Voici comment est défini le type `nat` des *entiers de Peano* en Coq :

```
Coq < Print nat.
Inductive nat : Set :=  O : nat | S : nat -> nat
```

Les constructeurs sont zéro et successeur.

Le principe du raisonnement par récurrence s'appuie sur cette définition : pour prouver

$$\forall n, \mathsf{P}(n)\,,$$

où n est un entier naturel (ce que l'on note n: `nat`), et $\mathsf{P}(\cdot)$ une propriété qui parle d'un entier naturel, on montre :

$$\mathsf{P}(\texttt{O}) \qquad \text{et} \qquad \forall n, (\mathsf{P}(n) \Rightarrow \mathsf{P}(n+1)\,,$$

ou plutôt :

$$\mathsf{P}(\texttt{O}) \qquad \text{et} \qquad \forall n, (\mathsf{P}(n) \Rightarrow \mathsf{P}(\texttt{S}\ n)\,.$$

(on ne sait pas ce que c'est $\mathsf{P}(n+1)$ à ce stade : est-ce $\mathsf{P}(n + (\texttt{S O}))$? mais qui est $+$ dans ce cas ?)

Sur ce modèle, la définition du type `ilist` engendre le principe de *preuve par induction sur les listes d'entiers* suivant : pour prouver

$$\forall \ell\colon \texttt{ilist}, \mathsf{P}(\ell)\,,$$

on montre

$$\mathsf{P}(\texttt{Nil}) \qquad \text{et} \qquad \forall \ell \forall k, \mathsf{P}(\ell) \Rightarrow \mathsf{P}(\texttt{Cons}\ k\ \ell)\,.$$

Ici $\mathsf{P}(\cdot)$ désigne un prédicat sur les listes, alors que ci-dessus $\mathsf{P}(\cdot)$ désignait un prédicat sur les entiers naturels. On peut par exemple démontrer par induction sur ℓ que $\forall \ell\colon \texttt{ilist}, \texttt{size}(\texttt{reverse}(\ell)) = \texttt{size}(\ell)$.

Exercice : arbres binaires. Comprendre les définitions Caml suivantes pour des arbres binaires contenant des entiers aux feuilles ou aux nœuds :

```
type bintree1 = Leaf of int | Node of bintree1*bintree1

type bintree2 = Leaf | Node of int*bintree2*bintree2
```

Écrire les versions Coq de ces définitions, et énoncer les principes de raisonnement par induction associés.

Tout ramener à `nat`. On peut varier à l'infini les définitions de structures de données inductives, avec, à chaque fois, les principes de preuve associés. Sur le principe, on pourrait se contenter de faire des inductions sur des `nat`, en raisonnant suivant les cas sur la taille d'une liste, le nombre de nœuds d'un arbre, la profondeur d'un arbre, etc. Toutefois, utiliser un principe de preuve qui "colle" à la définition du type sur lequel on raisonne permet souvent de faire des preuves plus directes, et plus lisibles.

Exercice. On a fait comme plus haut si le type `int` existait en Coq. Proposez une définition des entiers relatifs en Coq.

Définition de fonctions par induction. La définition inductive d'un ensemble permet de définir des fonctions portant sur des éléments de cet ensemble : c'est le cas des fonctions `size` ou `reverse` sur les listes, par exemple. L'appel récursif sur une sous-liste correspond à l'usage d'une hypothèse d'induction dans une preuve.

1.1.4 Langages de programmation

Un langage de programmation est un ensemble de programmes, qui peuvent être décrits à l'aide d'une définition inductive comme celles qui précèdent.

Voir un programme comme une structure de données est naturel lorsque l'on conçoit un programme qui manipule des programmes, comme par exemple un interprète (qui exécute le programme) ou un compilateur (qui traduit le programme en un autre programme, dans autre langage). Un programme qui analyse le programme, pour prouver qu'il termine sur toutes ses entrées, ou qu'il n'accède pas à des ressources protégées, ou plus généralement qu'il n'a pas de bug, est aussi un exemple dans cette lignée.

Les expressions arithmétiques, que l'on peut définir comme ci-dessous, sont un exemple de "langage de programmation" très simple — il s'agit en réalité d'un sous-ensemble de bon nombre de langages de programmation :

```
Inductive aexpr : Set :=
  Cst : int -> aexpr | Add : aexpr -> aexpr -> aexpr | Mul : aexpr -> aexpr -> aexpr.
```

Un élément du type `aexpr` est un arbre binaire avec deux types de nœuds, et des entiers aux feuilles, comme par exemple `a = Add (Cst 3, Mul(Cst 4, Cst 5))`. C'est un programme qui s'exécute en `23` (ou `Cst 23`, suivant ce que l'on attend pour le type du résultat).

Dans le même esprit, l'ensemble des programmes du langage Imp est décrit à la partie 2.1 à l'aide de plusieurs définitions inductives.

De même, la syntaxe du petit langage fonctionnel, Fun, est définie à la partie 3.1.

Cette approche permet de définir de manière inductive la syntaxe d'un langage de programmation. À une telle définition est associé un principe de preuve sur la structure des programmes (ou sur la syntaxe des programmes), sur laquelle on revient à la partie 4.

1.2 Relations

La plupart du temps, on considère des relations binaires, qui sont des sous-ensembles d'un produit cartésien $E_1 \times E_2$, où E_1, E_2 sont des ensembles. Il arrivera fréquemment que $E_1 = E_2$. On se contentera parfois de faire référence simplement à des relations, sans préciser "binaires".

On manipulera aussi des relations $n-$aires, avec comme cas particulier $n = 1$: on parle alors aussi de *prédicat*, comme par exemple "être un nombre premier", ou "être un arbre rouge et noir".

- Relation fonctionnelle : $x\mathcal{R}y$ et $x\mathcal{R}z$ impliquent $y = z$.

  On peut aussi parler de *déterminisme* de la relation.

  On sera amené à définir certaines fonctions en extension, comme sous-ensembles d'un produit $E_1 \times E_2$. Une telle relation représente une fonction si elle est fonctionnelle. Elle est de plus totale si tout élément de E_1 est en relation avec un élément de E_2.

- Composition de relations : si $\mathcal{R}$ et $\mathcal{S}$ sont des relations, $\mathcal{RS}$ est la relation définie par $x\mathcal{RS}y$ si et seulement s'il existe z t.q. $x\mathcal{R}z$ et $z\mathcal{S}y$.

- Forme normale : x est une forme normale pour $\mathcal{R}$ s'il n'existe pas de y avec $x\mathcal{R}y$. On écrit $x \not\mathcal{R}$.

- Clôtures : transitive $\mathcal{R}^+$, réflexive et transitive $\mathcal{R}^*$.

2 Imp, le petit langage impératif

2.1 Syntaxe

On se donne

- Les entiers $\mathbb{Z}$, notés $k, k_1, k', \ldots$
- Un ensemble $\mathcal{V}$ de *variables* Imp, notées $X, Y, Z, \ldots$. Ces variables sont parfois appelées des *adresses*, lorsqu'on les interprète comme des adresses en mémoire ; `X := Y+2` signifie donc lire la valeur stockée à l'adresse `Y`, ajouter `2`, et stocker le résultat à l'adresse `X`.

On définit trois ensembles, Arith, Bool, Com, qui désignent respectivement les expressions arithmétiques, les expressions booléennes, et les commandes, par l'intermédiaire des grammaires suivantes.

$$a \quad ::= \quad k \mid X \mid a_1 + a_2 \qquad\qquad b \quad ::= \quad \texttt{true} \mid \texttt{false} \mid a_1 \geq a_2 \mid b_1 \wedge b_2$$

$$c \quad ::= \quad X := a \mid c_1; c_2 \mid \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ b\ \texttt{do}\ c \mid \texttt{skip}$$

Les grammaires ci-dessus peuvent être vues comme une notation commode pour une définition inductive, que l'on peut expliciter en donnant la définition Coq correspondante (ici pour les commandes) :

```
Inductive com : Set :=
  assign : var*arexpr -> com
| seq : com*com -> com
| ifte : boolexpr*com*com -> com
| while : boolexpr*com -> com
| skip : com.
```

(comme remarqué plus haut, on adopterait sans doute plutôt la version curryfiée en Coq, avec par exemple `seq:com->com->com`).

L'ensemble Com est défini comme un type inductif ayant 5 constructeurs ; à chaque constructeur est associé un nombre d'arguments (son *arité*), et chaque argument appartient à un ensemble donné. Ainsi, `skip` n'a pas d'argument, et `ifte` a trois arguments, le premier dans Bool, les deux autres dans Com.

Syntaxe concrète, syntaxe abstraite.

Un élément d'un ensemble défini inductivement est un arbre, dont les nœuds correspondent à des constructeurs. C'est un objet bidimensionnel, alors que les grammaires sont volontiers associées à une représentation unidimentionnelle. Quand on écrit des programmes Imp, on adopte aussi une représentation unidimensionnelle :

```
X := 1; Y := 2; Z := 3
if X>0 then Y:= 3 else Y:= 4; Z:=10
X := A; I := 1; while I<10 do ( I := I+1+Z; X := X*B )
```

La *syntaxe concrète* (au sens d'une suite de caractères) utilisée ci-dessus présente des ambiguïtés. Comment se parenthèsent `X := 1; Y := 2; Z := 3` et `I+1+Z` ? Est-ce que le `Z:=10` va avec le `else` ?

Ces questions ne nous préoccupent pas ici, dans la mesure où les objets avec lesquels on travaille sont des arbres, qui représentent ce qu'on appelle la *syntaxe abstraite*. On utilisera souvent la syntaxe unidimensionnelle, quitte à supposer de manière implicite des conventions d'écriture, comme par exemple que `I+1+Z` désigne par convention `(I+1)+Z`.

Écriture des expressions arithmétiques, injections de $\mathcal{V}$ et $\mathbb{Z}$ dans Arith.

- Si $X \in \mathcal{V}$, alors X est aussi une expression arithmétique. Cela se traduit par le fait qu'il y a un constructeur qui réalise une injection des variables Imp dans les expressions arithmétiques.
- De même, un constructeur, mettons qu'il s'appelle `Cst`, injecte $\mathbb{Z}$ dans Arith. Ainsi, $17 \in \mathbb{Z}$, et donc $17 \in$ Arith, mais en réalité c'est plutôt $\texttt{Cst}(17) \in$ Arith, sauf que l'on ne prend pas la peine d'écrire explicitement le constructeur `Cst` qui fait l'injection.

  Ainsi, lorsqu'on écrit $17, \sigma \Downarrow 17$ (sémantique opérationnelle à grands pas de Imp), on dit que l'expression arithmétique notée 17 s'évalue en l'entier 17 (et pour tout expliciter, on écrirait $\texttt{Cst}(17), \sigma \Downarrow 17$).
- Enfin, l'expression que l'on note $a_1 + a_2$ est en réalité un arbre de la forme $\texttt{Add}(a_1, a_2)$. Lorsque l'on écrit $3 + 2, \sigma \Downarrow 5$, on dit que l'expression (*l'arbre*) $\texttt{Add}(\texttt{Cst}(3), \texttt{Cst}(2)))$ s'évalue en 5.

  Dans la définition de la sémantique dénotationnelle, lorsque l'on écrit

$$\mathcal{D}(a_1 + a_2) \quad = \quad \{(\sigma, k_1 + k_2),\ (\sigma, k_1) \in \mathcal{D}(a_1), (\sigma, k_2) \in \mathcal{D}(a_2)\},$$

  $+$ désigne le constructeur des expressions arithmétiques dans $a_1 + a_2$, et $+$ désigne l'opération sur les entiers dans $k_1 + k_2$.

  Tout cela n'est certes pas très profond, il s'agit de clarifier certaines ambiguïtés de notations.

États mémoire. Les programmes Imp agissent sur la mémoire en lisant et écrivant des entiers dans des cases mémoire.

Définition 1 (État mémoire)**.** *On se donne un ensemble $\mathcal{M}$ d'*états mémoire*, notés $\sigma, \sigma', \ldots$, qui sont des fonctions de $\mathcal{V}$ dans $\mathbb{Z}$. On note $\sigma(X)$ l'entier associé à X dans σ. On note $\sigma[X \mapsto k]$ l'état mémoire σ' défini par $\sigma'(X) = k$ et $\sigma'(Y) = \sigma(Y)$ pour $Y \neq X$ (σ mis à jour en associant k à X).*

2.2 Sémantique dénotationnelle de Imp

On définit, par induction sur une commande c, sa sémantique dénotationnelle notée $\mathcal{D}(c)$, qui est une fonction, ou plutôt une relation sensée représenter une fonction.

2.2.1 Sans le `while`

Du fait de la définition "à étages" de la syntaxe de Imp, la définition de la sémantique dénotationnelle se fait en trois étapes. On simplifie la notation en écrivant $\mathcal{D}(a)$, $\mathcal{D}(b)$ et $\mathcal{D}(c)$ pour la sémantique dénotationnelle d'une expression arithmétique a, d'une expression booléenne b et d'une commande c : on définit trois fonctions différentes, et on les note de la même façon pour se simplifier la vie. Dans les trois cas, la sémantique dénotationnelle est définie comme une relation binaire (cf. partie 1.2).

Pour les expressions arithmétiques, on dit ce que vaut une expression arithmétique a dans un état mémoire σ: si a contient X, il nous faut σ pour savoir ce que vaut X.
$\mathcal{D}(a) \subseteq \mathcal{M} \times \mathbb{Z}$

$$\begin{array}{rcl} \mathcal{D}(X) & = & \{(\sigma, \sigma(X))\} \\ \mathcal{D}(12) & = & \{(\sigma, 12)\} \\ \mathcal{D}(a_1 + a_2) & = & \{(\sigma, k_1 + k_2),\ (\sigma, k_1) \in \mathcal{D}(a_1), (\sigma, k_2) \in \mathcal{D}(a_2)\} \end{array}$$

Pour les expressions booléennes, on laisse cela en exercice.

Pour les commandes, on a $\mathcal{D}(c) \subseteq \mathcal{M} \times \mathcal{M}$.

$$\begin{array}{rcl}
\mathcal{D}(\texttt{skip}) & = & \{(\sigma,\sigma)\} \\
\mathcal{D}(c_1;c_2) & = & \{(\sigma,\sigma'').\ \exists\sigma'.\ (\sigma,\sigma') \in \mathcal{D}(c_1) \text{ et } (\sigma',\sigma'') \in \mathcal{D}(c_2)\} \\
\mathcal{D}(X := a) & = & \{(\sigma,\sigma[X \mapsto \mathcal{D}(a)\sigma])\} \\
\mathcal{D}(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2) & = & \{(\sigma,\sigma'), \mathcal{D}(b)\sigma = \texttt{true} \text{ et } (\sigma,\sigma') \in \mathcal{D}(c_1)\} \\
& & \quad \cup \quad \{(\sigma,\sigma'), \mathcal{D}(b)\sigma = \texttt{false} \text{ et } (\sigma,\sigma') \in \mathcal{D}(c_2)\} \\
\mathcal{D}(\texttt{while } b \texttt{ do } c) & = & \text{cf. partie 2.2.2}
\end{array}$$

$\mathcal{D}(c)$ est une fonction ?

- Déterminisme.

  $\mathcal{D}(a)$ est défini par induction sur la structure d'une expression arithmétique a. On montre par induction sur la structure de a que cette relation est fonctionnelle. Cela nous permet d'écrire, par abus, $\mathcal{D}(X)\sigma = \sigma(X)$.

  Similairement, on aimerait bien écrire $\mathcal{D}(c)(\sigma) = \sigma'$ lorsque $(\sigma,\sigma') \in \mathcal{D}(c)$. Il faut pour cela savoir que $\mathcal{D}(\cdot)$ est fonctionnelle, i.e.,

  $$\big((\sigma,\sigma_1) \in \mathcal{D}(c) \ \wedge \ (\sigma,\sigma_2) \in \mathcal{D}(c)\big) \quad \Rightarrow \quad \sigma_1 = \sigma_2 \ .$$

  Lorsqu'on saura que c'est le cas, on pourra d'ailleurs remarquer que $\mathcal{D}(c_1;c_2) = \mathcal{D}(c_2) \circ \mathcal{D}(c_1)$.

- Fonction partielle.

  Tant qu'on n'a pas `while`, c'est simple.

  $$\mathcal{D}(\texttt{while } X > 0 \texttt{ do } X := X + 1) = \textbf{??}$$

  NB : pas besoin de savoir si $\mathcal{D}(c)$ est déterministe ou pas pour construire la sémantique de `while`.

2.2.2 Sémantique dénotationnelle du `while`

Rappels : $\mathcal{D}(a) \subseteq \mathcal{M} \times \mathbb{Z}$, $\mathcal{D}(b) \subseteq \mathcal{M} \times \mathbb{B}$, $\mathcal{D}(c) \subseteq \mathcal{M} \times \mathcal{M}$.

On considère l'instruction `while` b `do` c, et on suppose que les ensemble $\mathcal{D}(b)$ et $\mathcal{D}(c)$ sont fixés/connus : on se place dans la situation où l'on est en train de définir la sémantique dénotationnelle des commandes Imp par induction sur la structure de la commande.

On définit

$$\begin{array}{rcl}
F(S) & = & \{(\sigma,\sigma) \text{ avec } (\sigma,\texttt{false}) \in \mathcal{D}(b)\} \\
& & \quad \cup \quad \{(\sigma,\sigma'') \text{ avec } (\sigma,\texttt{true}) \in \mathcal{D}(b) \text{ et } \exists\sigma', (\sigma,\sigma') \in \mathcal{D}(c) \text{ et } (\sigma',\sigma'') \in S\}
\end{array}$$

On veut avoir $\mathcal{D}(\texttt{while } b \texttt{ do } c) = F(\mathcal{D}(\texttt{while } b \texttt{ do } c))$ (la sémantique de `while` b `do` c est un point fixe de F), équation qui traduit l'intuition `while` b `do` c = `if` b `then` $(c;$ `while` b `do` $c)$ `else skip`.

Pourquoi ne pas appliquer Knaster-Tarski ? $\mathcal{D}(c)$ représente une fonction *partielle* de $\mathcal{M}$ dans $\mathcal{M}$, c'est une partie de $\mathcal{M}^2$. On pense donc au treillis complet $(\mathcal{P}(\mathcal{M} \times \mathcal{M}), \subseteq)$, pour calculer le point fixe en appliquant Knaster-Tarski (cf. partie 4.1.1).

On fait alors la remarque suivante: on veut que $\mathcal{D}(c)$ soit une relation *fonctionnelle*. Or cette propriété n'est pas préservée par $\cup$. En se promenant dans ce treillis, on risque de calculer des relations qui ne sont pas fonctionnelles.

Cela nous amène à présenter un autre théorème du point fixe. (Si ça se trouve, en appliquant Knaster-Tarski sur les relations quelconques, on tombe sur un point fixe qui est une relation fonctionnelle. Plutôt que de démontrer cela a posteriori, on préfère raffiner la structure mathématique avec laquelle on travaille, ce qui correspond à une démarche typique en sémantique dénotationnelle.)

Domaines et points fixes.

Définition 2 (Domaine)**.** *Une* chaîne infinie *dans un ordre partiel $(E, \sqsubseteq)$ est une suite $(e_n)_{n\geq 0}$ telle que $e_0 \sqsubseteq e_1 \sqsubseteq e_2 \sqsubseteq \ldots$.*

Un ordre partiel $(E, \sqsubseteq)$ est complet *si pour toute chaîne infinie $(e_n)_{n\geq 0}$, il existe un plus petit majorant des $(e_n)_{n\geq 0}$, noté $\bigsqcup_{n\geq 0} e_n$, qui vérifie donc*

$$\forall i \geq 0, e_i \sqsubseteq \bigsqcup_{n\geq 0} e_n \quad \text{et} \quad \forall m \in E, \text{ si } \forall i \geq 0, e_i \sqsubseteq m, \text{ alors } \bigsqcup_{n\geq 0} e_n \sqsubseteq m.$$

Si de plus E a un plus petit élément (noté généralement $\bot$), on dit que E est un domaine*.*

Exemple important : les $\mathcal{R} \subseteq \mathcal{M} \times \mathcal{M}$ qui sont fonctionnelles, avec $\cup$ et $\emptyset$, sont un domaine. Elles ne sont pas un treillis complet car $\cup$ ne préserve pas en général le fait d'être une relation fonctionnelle. On revient là-dessus plus loin.

Définition 3 (Fonction continue)**.** *Soit $(E, \sqsubseteq)$ un domaine, une fonction $f : E \to E$ est* continue *ssi f est croissante et pour toute chaîne infinie $(e_n)_{n\geq 0}$ dans $(E, \sqsubseteq)$, on a*

$$f(\bigsqcup_{n\geq 0} e_n) = \bigsqcup_{n\geq 0} f(e_n).$$

Si de plus $f(\bot) = \bot$, on dit que f est stricte*.*

Remarque : on a le droit de calculer le $\bigsqcup$ à droite dans la définition ci-dessus car comme f est croissante, $(f(e_n))_{n\geq 0}$ est bien une chaîne infinie.

Théorème 4 (du point fixe sur les domaines)**.** *Soit $(E, \sqsubseteq)$ un domaine et $f : E \to E$ continue.*

Alors, en posant $f^0(x) = x, \forall x \in E$ et $f^{i+1}(x) = f(f^i(x)), \forall x \in E, \forall i \geq 0$, on définit

$$fix(f) = \bigsqcup_{n\geq 0} f^n(\bot).$$

$fix(f)$ est alors le plus petit point fixe de f.

Remarque : comme plus haut, on note que les $(f^n(\bot))_{n\geq 0}$ forment bien une chaîne infinie dans $(E, \sqsubseteq)$.

Application à Imp. L'ensemble des *relations fonctionnelles* sur $\mathcal{M}$, ordonné par $\subseteq$, a une structure de domaine. En particulier, $\bot = \emptyset$, et $\bigcup$ calcule les plus petits majorants.

On associe à une boucle `while` b `do` c un *plus petit point fixe*, donné par le théorème 4 ci-dessus. Pour ce faire, à $\mathcal{D}(b)$ et $\mathcal{D}(c)$ fixés, on doit montrer la continuité de la fonction F qui a été définie plus haut, ce que l'on laisse en exercice.

On peut donc poser

$$\mathcal{D}(\texttt{while } b \texttt{ do } c) = \bigcup_{n\geq 0} F^n(\emptyset),$$

qui, par application du théorème, est le plus petit point fixe de F.

Intuitivement, pour calculer $\mathcal{D}(\texttt{while } b \texttt{ do } c)$, on procède par approximations, dans l'espace des fonctions que parcourt $\mathcal{D}(\cdot)$. $\emptyset$ représente $\bot$, la fonction définie nulle part ("aucune information"). L'ordre $\subseteq$ traduit "davantage d'information".

Totalité. La fonction qui à une commande c associe sa dénotation $\mathcal{D}(c)$ n'est pas totale : pourquoi ?

Compositionalité. La sémantique dénotationnelle est *compositionnelle* : on calcule $\mathcal{D}(\texttt{while}\ b\ \texttt{do}\ c)$ à partir de $\mathcal{D}(c)$ (et de $\mathcal{D}(b)$). Alors que pour savoir si $\texttt{while}\ b\ \texttt{do}\ c, \sigma \Downarrow \sigma'$ (cf. partie 2.3), on doit potentiellement savoir si $\texttt{while}\ b\ \texttt{do}\ c, \sigma'' \Downarrow \sigma'$ pour un certain σ'', et ça parle encore de $\texttt{while}\ b\ \texttt{do}\ c$.

On a la coïncidence entre les sémantiques dénotationnelle et opérationnelle (définie plus bas), ce qui s'exprime ainsi :

$$\mathcal{D}(c) = \{(\sigma, \sigma'), c, \sigma \Downarrow \sigma'\}.$$

En quelque sorte, cela valide les choix de points fixes que l'on a faits.

2.3 Sémantique opérationnelle à grands pas de Imp

La définition de la sémantique opérationnelle de Imp s'appuie sur l'usage de *règles d'inférence* pour exprimer à quelle condition on peut *dériver* (ou *déduire*) $c, \sigma \Downarrow \sigma'$, ce qui exprime le fait que la commande c, exécutée à partir de l'état σ, termine son exécution dans l'état σ'. Pour définir cette relation, on définit des relations similaires sur les expressions arithmétiques et booléennes.

Pour les expressions arithmétiques : $a, \sigma \Downarrow k$, l'expression a s'évalue en k dans l'état σ.

$$\mathrm{E_{ak}}\ \frac{}{k, \sigma \Downarrow k} \qquad \mathrm{E_{av}}\ \frac{}{X, \sigma \Downarrow k}\ k = \sigma(X) \qquad \mathrm{E_{ap}}\ \frac{a_1, \sigma \Downarrow k_1 \qquad a_2, \sigma \Downarrow k_2}{a_1 + a_2, \sigma \Downarrow k}\ k = k_1 + k_2$$

La sémantique à grands pas s'appelle parfois évaluation, d'où le préfixe "E" dans le nom des règles d'inférence.
Pour les expressions booléennes, relation $b, \sigma \Downarrow \texttt{bv}$ (avec $\texttt{bv} \in \{\texttt{true}, \texttt{false}\}$) : laissé en exercice.
Pour les commandes :

$$\mathrm{E_{skip}}\ \frac{}{\texttt{skip}, \sigma \Downarrow \sigma} \qquad \mathrm{E_{assgn}}\ \frac{a, \sigma \Downarrow k}{X := a, \sigma \Downarrow \sigma'}\ \sigma' = \sigma[k/X] \qquad \mathrm{E_{seq}}\ \frac{c_1, \sigma \Downarrow \sigma' \qquad c_2, \sigma' \Downarrow \sigma''}{c_1; c_2, \sigma \Downarrow \sigma''} \qquad \mathrm{E_{it}}\ \frac{b, \sigma \Downarrow \texttt{true} \qquad c_1, \sigma \Downarrow \sigma'}{\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma \Downarrow \sigma'}$$

$$\mathrm{E_{if}}\ \frac{b, \sigma \Downarrow \texttt{false} \qquad c_2, \sigma \Downarrow \sigma'}{\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma \Downarrow \sigma'} \qquad \mathrm{E_{wf}}\ \frac{b, \sigma \Downarrow \texttt{false}}{\texttt{while}\ b\ \texttt{do}\ c, \sigma \Downarrow \sigma} \qquad \mathrm{E_{wt}}\ \frac{b, \sigma \Downarrow \texttt{true} \qquad c, \sigma \Downarrow \sigma' \qquad \texttt{while}\ b\ \texttt{do}\ c, \sigma' \Downarrow \sigma''}{\texttt{while}\ b\ \texttt{do}\ c, \sigma \Downarrow \sigma''}$$

Conditions d'application dans les règles d'inférence. Les trois règles pour X et $a_1 + a_2$ (expressions arithmétiques) et pour $X := a$ (commandes) comportent des *conditions d'application* (*side condition*). Il s'agit d'hypothèses qui ont la particularité qu'elles ne sont pas définies comme des relations inductives (en l'occurrence, on "fait un calcul").

On prend ici le parti de ne pas les mettre parmi les prémisses (au-dessus de la barre), mais on aurait aussi pu écrire, de manière plus compacte :

$$\frac{}{X, \sigma \Downarrow \sigma(X)} \qquad \frac{a_1, \sigma \Downarrow k_1 \qquad a_2, \sigma \Downarrow k_2}{a_1 + a_2, \sigma \Downarrow k_1 + k_2} \qquad \frac{a, \sigma \Downarrow k}{X := a, \sigma \Downarrow \sigma'[k/X]}$$

Les deux manières d'écrire sont admises.

Remarque 5. *Il y a deux règles d'inférence pour décrire l'exécution d'un* `if then else` *. On pourrait imaginer la règle maladroite suivante :*

$$\frac{b, \sigma \Downarrow \texttt{bv} \qquad c_1, \sigma \Downarrow \sigma_1 \qquad c_2, \sigma \Downarrow \sigma_2}{\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma \Downarrow \sigma'} \quad \begin{array}{l} \sigma' = \sigma_1\ \textit{si}\ \texttt{bv} = \texttt{true}, \\ \sigma' = \sigma_2\ \textit{si}\ \texttt{bv} = \texttt{false} \end{array}$$

Avec une telle règle, on évalue les deux commandes alors que c'est inutile.

La définition ci-dessus décrit la sémantique opérationnelle *à grands pas* de Imp. On verra également une version à petits pas. Ces deux présentations coïncident, en un certain sens. Elles sont également en accord avec la sémantique dénotationnelle de Imp : cf. discussion à la fin de la partie 2.2.

2.4 Imp avec tas

2.4.1 Extension du langage

Tas, adresses. On définit une extension de Imp dans laquelle la syntaxe pour les expressions arithmétiques et pour les commandes est enrichie de la manière suivante :

$$a \quad ::= \quad \cdots \mid [a] \qquad\qquad c \quad ::= \quad \cdots \mid [a_1] := a_2 \mid X := \texttt{alloc}(a)$$

Ces trois constructions supplémentaires correspondent à la lecture en mémoire, l'écriture en mémoire, et l'allocation d'un segment de mémoire. Elles agissent sur une zone de la mémoire qui s'appelle le *tas*. On raffine ce faisant la modélisation de l'état mémoire, qui comporte deux composantes :

- comme avant, σ indique la valeur associée à chaque variable Imp. L'accès à la mémoire via les variables Imp est complètement abstrait : les variables existent, on ne sait pas où elles se trouvent dans la mémoire. On peut imaginer que les variables Imp se comportent comme des registres.
- une nouvelle composante, h, représente le tas (*heap* en anglais). Dans le tas, on a accès à la structuration de la mémoire, dans la mesure où l'on peut parler de cases mémoires *adjacentes*.

On accède au tas à l'aide des constructions introduites ci-dessus. La commande `K := alloc(3)` a pour effet d'allouer trois cases consécutives dans le tas, qui sont initialisées à zéro. On parle d'*allocation dynamique*, car on étend le tas au cours de l'exécution du programme. Après exécution de cette commande, on sait que ces trois cases mémoire sont situées aux adresses `K`, `K+1` et `K+2`.

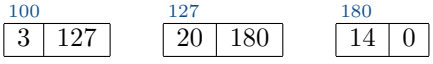
Pour l'écriture et la lecture, on traite $[a]$ comme le contenu de la case mémoire dont l'adresse est k, si a s'évalue en k (et si une telle case mémoire est effectivement allouée dans le tas). Ainsi, la commande `[K] := [K'+1]+2` a pour effet de lire ce que contient la case mémoire située à l'adresse `K'+1`, d'y ajouter 2, et stocker le résultat dans la case mémoire située à l'adresse `K` dans le tas.

Par exemple, le programme suivant

```
X := alloc(4);
[X] := T;
[X+1] := U;
[X+2] := [Y];
[X+3] := [Z]
```

alloue 4 cases mémoire *adjacentes/consécutives*, et y stocke (i) la valeur de deux variables `T` et `U` ; (ii) les valeurs stockées en mémoire aux adresses `Y` et `Z`.

Dans cette version étendue de Imp, on peut représenter des programmes manipulant des structures de données riches, que l'on stocke dans le tas. Voici par exemple à quoi pourrait ressembler une partie du tas où est stockée la liste `[3;20;14]`, à l'aide de *cellules* consistant en deux cases mémoire consécutives :



La première cellule est stockée à l'adresse 100, et contient les entiers 3 et 127. Par convention, l'adresse 0 correspond à la liste vide (et une liste non vide ne peut pas commencer à l'adresse 0). Avec une telle représentation pour les listes, on parcourt la structure de données avec une boucle de la forme

`while P<>0 do (...; P := [P+1])`,

où `P` "pointe" vers une case mémoire contenant une liste.

Voici un autre exemple simple de programme dans "Imp avec tas".

```
T := alloc(S);
I := 0;
while I<S do
  (if I=0 then PRED := 0 else PRED := I-1;
   [T+I] := (I+1) + [T+PRED])
```

Choix de modélisation. La version de Imp avec tas que nous présentons ici représente une possibilité parmi de multiples choix de conception pour une telle extension de Imp. Avec le langage que l'on a spécifié, il ne paraît pas très naturel, même si c'est possible, d'accéder à des cases mémoire en fournissant leur "vraie adresse", comme dans [17] : on procède plutôt en faisant `X := alloc(5)`, et en accédant à des adresses calculées à partir de `X`, sans dépendre de l'endroit effectif où la zone commençant en `X` a été allouée.

On pourrait imaginer travailler sans allocation dynamique, en se donnant simplement un grand tableau indicé par des entiers. On pourrait aussi ajouter une commande `free`(a), pour désallouer la mémoire — dans notre modèle, le tas peut grandir de manière non bornée au cours de l'exécution.

2.4.2 Sémantique opérationnelle.

L'état mémoire, formellement. Comme précédemment, on a un ensemble $\mathcal{M}$ de fonctions σ associant un entier à chaque variable Imp (en réalité un programme Imp donné manipule un ensemble fini de variables, que l'on peut lire sur le code du programme).

On se donne par ailleurs un ensemble $\mathcal{H}$ de tas, notés $h, h', h_1, \ldots$. Un tas est une fonction associant une valeur à un ensemble *fini* d'adresses (les adresses étant définies comme les entiers positifs — on s'appuie en particulier sur le fait que l'ensemble des adresses est infini).

On écrit par exemple $[2 \mapsto 10, 3 \mapsto 20]$ pour représenter le tas qui est défini pour les adresses 2 et 3, et où sont stockés 10 et 20.

Étant donné un tas h, $\mathrm{dom}(h)$ désigne l'ensemble des adresses définies dans h.

On note $h(k)$ la valeur associée à k dans le tas h, qui n'est définie que si $k \in \mathrm{dom}(h)$.

Si h est un tas, le tas $h[k \mapsto k']$ n'est défini que si $k \in \mathrm{dom}(h)$, et désigne le tas où l'on change la valeur associée à l'adresse k dans h, en la remplaçant par k'.

Étant donnés deux tas h_1 et h_2, le tas noté $h_1 \uplus h_2$ n'est défini que si $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$, et est par définition le tas défini sur les adresses $\mathrm{dom}(h_1) \cup \mathrm{dom}(h_2)$, qui associe la valeur "évidente" à chacune des adresses de cet ensemble.

Grands pas. L'évaluation des expressions arithmétiques s'appuie sur la mémoire, à la fois la composante σ (pour la valeur des variables Imp) et la composante h (pour les lectures en mémoire). Elle est décrite par un jugement de la forme $a, \sigma, h \Downarrow k$. On hérite des règles pour Imp sans tas en ajoutant simplement la composante h. Ainsi, par exemple, on a la règle

$$\frac{a_1, \sigma, h \Downarrow k_1 \qquad a_2, \sigma, h \Downarrow k_2}{a_1 + a_2, \sigma, h \Downarrow k} \; k = k_1 + k_2$$

La règle d'inférence pour la lecture en mémoire s'écrit ainsi :

$$\frac{a, \sigma, h \Downarrow k}{[a], \sigma, h \Downarrow k'} \; h(k) = k'$$

On peut remarquer que la lecture est bloquée si l'adresse mémoire n'est pas allouée dans le tas.

On écrit $b, \sigma, h \Downarrow bv$ pour l'évaluation des expressions booléennes. Le tas ne joue pas de rôle dans les règles d'inférence correspondantes, il est simplement nécessaire lorsque l'on fait appel à l'évaluation des expressions arithmétiques.

L'évaluation des commandes peut avoir pour effet de modifier à la fois σ et h : le jugement s'écrit par conséquent $a, \sigma, h \Downarrow \sigma', h'$.

Voici comment sont modifiées certaines règles de la sémantique opérationnelle à grands pas de IMP :

$$\frac{a, \sigma, h \Downarrow k}{X := a, \sigma, h \Downarrow \sigma[X \mapsto k], h} \qquad \frac{c_1, \sigma, h \Downarrow \sigma', h' \qquad c_2, \sigma', h' \Downarrow \sigma'', h''}{c_1; c_2, \sigma, h \Downarrow \sigma'', h''}$$

(les autres règles sont laissées en exercice).

Et voici les règles pour l'écriture en mémoire et pour l'allocation dynamique :

$$\frac{a_2, \sigma, h \Downarrow k_2 \qquad a_1, \sigma, h \Downarrow k_1}{[a_1] := a_2, \sigma, h \Downarrow h'} \quad \begin{array}{l} k_1 \in \mathrm{dom}(h) \\ h' = h[k_1 \mapsto k_2] \end{array}$$

$$\frac{a, \sigma, h \Downarrow k}{X := \texttt{alloc}(a), \sigma, h \Downarrow \sigma', h'} \quad \begin{array}{l} \{k', \ldots, k' + k - 1\} \cap \mathrm{dom}(h) = \emptyset \\ \sigma' = \sigma[X \mapsto k'] \\ h' = h \uplus [k' \mapsto 0, \ldots, k' + k - 1 \mapsto 0] \end{array}$$

On voit dans la dernière règle que le choix de l'adresse k' où commence la zone mémoire allouée est non-déterministe. Par ailleurs, on ne peut pas faire `alloc`(0).

3 Le langage Fun

3.1 Syntaxe de Fun

$k \in \mathbb{Z}, x, y, z \in \mathcal{V}$, où $\mathcal{V}$ est un ensemble infini de *variables* FUN.

Uniquement des *expressions*. On définit l'ensemble $\mathcal{E}_{\text{FUN}}$ des expressions FUN :

$$e \quad ::= \quad k \mid e_1 + e_2 \mid \texttt{fun}\ x \to e \mid e_1\ e_2 \mid x$$

$e_1\ e_2$ désigne l'*application* de e_1 à e_2.

x est la variable, qui est introduite par un `fun` (comme dans $\texttt{fun}\ x \to x + x$).

On pourra ainsi écrire, par exemple, $(\texttt{fun}\ x \to x + x)\ (7 + 3)$.

En quoi Fun est-il un langage de programmation ? Il n'y a pas de `if then else` , il n'y a pas de modification de la mémoire, pas de boucle. On peut les ajouter au langage. Ou alors répondre, sur un plan théorique, que tout calcul fait en IMP peut être programmé en FUN, et inversement (c'est plutôt une propriété à laquelle on s'intéresse dans le cours FDI).

Liaison de variable. On dit que la variable x est *liée* dans $\texttt{fun}\ x \to e$: c'est un nom arbitraire, qui fait référence au paramètre de la fonction. Il n'y a pas de raison de distinguer $\texttt{fun}\ x \to x + x$ de $\texttt{fun}\ u \to u + u$. Le même phénomène est à l'œuvre dans $\texttt{let}\ x = 5\ \texttt{in}\ x + 3$, que l'on aimerait identifier avec $\texttt{let}\ n = 5\ \texttt{in}\ n + 3$: dans $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, la variable x est *liée dans* e_2.

La relation dite d'*α-conversion* identifie les expressions qui ne diffèrent que par un renommage de leurs variables liées. On ne peut pas renommer n'importe comment. Si on considère $e_0 = \texttt{fun}\ x \to (\texttt{fun}\ y \to x + y + y)$, on peut renommer y en t, et identifier e_0 et $\texttt{fun}\ x \to (\texttt{fun}\ t \to x + t + t)$. On peut renommer x en a, et identifier e_0 et $\texttt{fun}\ a \to (\texttt{fun}\ y \to a + y + y)$. Mais on ne peut pas renommer y en x, car cela donnerait $\texttt{fun}\ x \to (\texttt{fun}\ x \to x + x + x)$, qui est une fonction différente ; de même, on ne peut pas renommer x en y.

Une expression FUN contient des variables liées et des variables libres, ce qui conduit à la définition suivante :

Définition 6 (Variables libres, expression close)**.** *L'ensemble des variables libres d'une expression e, noté $v\ell(e)$, est défini par induction sur e de la manière suivante (il y a 5 cas) :*

$$v\ell(k) = \emptyset \qquad v\ell(e_1 + e_2) = v\ell(e_1) \cup v\ell(e_2) \qquad v\ell(e_1\ e_2) = v\ell(e_1) \cup v\ell(e_2)$$

$$v\ell(x) = \{x\} \qquad v\ell(\texttt{fun}\ x\ \texttt{->}\ e) = v\ell(e) \setminus \{x\}$$

Une expression e est dite close *si $v\ell(e) = \emptyset$.*

3.2 Sémantique à petits pas pour Fun

Réduction, formes normales et valeurs. On souhaite introduire une relation binaire, notée $\longrightarrow$, entre expressions closes : $e \longrightarrow e'$ se lit "e se réduit en e'". L'intention est que $\longrightarrow$ corresponde à un pas élémentaire de calcul ; en particulier, la relation $\longrightarrow$ ne doit être réflexive nulle part. Les symboles $\longrightarrow$ (pour la réduction) et $\rightarrow$ (pour les fonctions Fun) se ressemblent beaucoup, surtout si on écrit à la main. Il n'y a en principe pas d'ambiguïté.
Formes normales : $e \not\longrightarrow$, pour de bonnes et de mauvaises raisons.
Valeurs : sous-ensemble des formes normales (donc des expressions) qui sont le résultat d'un calcul.

$$v \quad ::= \quad k \mid \texttt{fun}\ x\ \texttt{->}\ e$$

Par définition, une *valeur close* est une valeur n'ayant pas de variables libres ; dans le cas d'une fonction $\texttt{fun}\ x\ \texttt{->}\ e$, cela signifie $v\ell(e) \subseteq \{x\}$.

$$\mathrm{R_{pk}}\ \frac{}{k_1 + k_2 \longrightarrow k}\ k = k_1 + k_2 \qquad \mathrm{R_{pg}}\ \frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2} \qquad \mathrm{R_{pd}}\ \frac{e_2 \longrightarrow e_2'}{v + e_2 \longrightarrow v + e_2'}$$

$$\mathrm{R_{\beta}}\ \frac{}{(\texttt{fun}\ x\ \texttt{->}\ e)\ v \longrightarrow e[v/x]} \qquad \mathrm{R_{ag}}\ \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \mathrm{R_{ad}}\ \frac{e_2 \longrightarrow e_2'}{v\ e_2 \longrightarrow v\ e_2'}$$

Dans la règle R_β, on fait référence à $e[v/x]$, le résultat de la substitution de x par v dans e : voir la définition 7.

On a évoqué plus haut, au moment de définir la sémantique à grands pas de Imp, l'utilisation de conditions d'application dans les règles d'inférence pour indiquer des hypothèses ou des calculs qui se font "à part", qui ne correspondent pas à des relations définies inductivement. Ici, pour la règle où l'on réduit $(\texttt{fun}\ x\ \texttt{->}\ e)\ v$, on choisit de mettre directement $e[v/x]$, plutot que e' avec la condition d'application $e'[v/x]$. En revanche, pour réduire $k_1 + k_2$, on a gardé une condition d'application, car si l'on écrivait $k_1 + k_2 \longrightarrow k_1 + k_2$, cela pourrait prêter à confusion, et faire croire qu'il y a des expressions sur lesquelles $\longrightarrow$ est une relation réflexive. Ici, k est l'entier résultant de la somme de k_1 et k_2, et à droite de la flèche $\longrightarrow$, on trouve l'expression Fun constante égale à k.

Définition 7 (Substitution dans Fun)**.** *Soient $e \in \mathcal{E}_{\mathrm{Fun}}$ une expression quelconque et $e' \in \mathcal{E}_{\mathrm{Fun}}$ une expression* close*.*

On définit, par induction sur la structure de e, le résultat de la substitution de x par e' dans e*, noté $e[e'/x]$, de la manière suivante (il y a 5 cas) :*

$$k[e'/x] = k \qquad (e_1 + e_2)[e'/x] = (e_1[e'/x]) + (e_2[e'/x]) \qquad (e_1\ e_2)[e'/x] = (e_1[e'/x])\ (e_2[e'/x])$$

$$y[e'/x] = \begin{cases} y & si\ x \neq y \\ e' & si\ x = y \end{cases} \qquad (\texttt{fun}\ y\ \texttt{->}\ e_0)[e'/x] = \begin{cases} \texttt{fun}\ y\ \texttt{->}\ (e_0[e'/x]) & si\ x \neq y \\ \texttt{fun}\ x\ \texttt{->}\ e_0 & si\ x = y \end{cases}$$

On impose que e' soit close dans la définition 7. On pourrait faire l'économie de cette hypotèse dans la définition, mais la substitution ferait "n'importe quoi" dans certaines situations (exercice : pourquoi ?). De fait, on ne substituera qu'avec des expressions closes, raison pour laquelle l'hypothèse fait partie de la définition 7.

Lemme 8 (Substitution et variables libres)**.** *Si $e' \in \mathcal{E}_{\text{FUN}}$ est close, alors $v\ell(e[e'/x]) = v\ell(e) \setminus \{x\}$.*

Lemme 9 (Substitution et expressions closes)**.** *Si $e' \in \mathcal{E}_{\text{FUN}}$ est close et si $v\ell(e) \subseteq \{x\}$, alors $e[e'/x]$ est close.*

Une relation sur les expressions closes. La relation $\longrightarrow$ est a priori définie sur $\mathcal{E}_{\text{FUN}} \times \mathcal{E}_{\text{FUN}}$, où $\mathcal{E}_{\text{FUN}}$ est l'ensemble des expressions FUN. Nous allons nous restreindre à des situations où $\longrightarrow$ ne met en relation que des *expressions closes* (une expression est close si elle n'a pas de variable libre) : à chaque fois que l'on écrit $e \longrightarrow e'$, e et e' sont implicitement supposées closes. On peut procéder ainsi car on a la propriété que si $e \longrightarrow e'$ et e close, alors e' est close. La seule règle pour laquelle cela n'est pas immédiat est l'axiome pour l'application ; on utilise le Lemme 9 dans ce cas.

Puisque $\longrightarrow$ met en relation des expressions appartenant au même ensemble, on peut itérer cette relation, en considérant $\longrightarrow^+$ et $\longrightarrow^*$, ce qui correspond à faire plusieurs pas de calcul.

La relation de réduction $\longrightarrow$ définit la sémantique opérationnelle à petits pas pour FUN. On peut se demander, étant donńee $e \in \mathcal{E}_{\text{FUN}}$,

- s'il existe e' telle que $e \longrightarrow^* e'$ et $e' \not\longrightarrow$, et, dans ce cas, d'une part si e' est nécessairement une valeur, et d'autre part s'il existe e'' avec $e \longrightarrow^* e''$, $e'' \not\longrightarrow$ et $e' \neq e''$;
- s'il existe $(e_n)_{n\geq 0}$ avec $e_0 = e$ et $\forall i \geq 0, e_i \longrightarrow e_{i+1}$.

Exercice. Définissez une sémantique à grands pas pour FUN. Donnez les grandes étapes de la preuve que les sémantiques à petits et grands pas coïncident.

3.3 Ajout des déclarations à Fun

Le langage FUN se prête à de multiples extensions. La plus immédiate est l'ajout des *déclarations* :

$$e \quad ::= \quad \cdots \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$$

Dans $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, on définit x comme étant égal à e_1 dans $e_2 \ldots$ ou plutôt comme égal au résultat de l'évaluation de e_1 dans e_2. x peut donc être utilisé dans e_2. C'est le cas par exemple dans $\texttt{let}\ x = 7\ \texttt{in}\ x+x$. La variable x est *liée* dans e_2 : on a $v\ell(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) = v\ell(e_1) \cup (v\ell(e_2) \setminus \{x\})$.

Une déclaration n'est pas une valeur. Les règles de réduction pour les déclarations sont les suivantes :

$$\frac{}{\texttt{let}\ x = v\ \texttt{in}\ e \longrightarrow e[v/x]}\ \text{R}_{\text{lv}} \qquad \frac{e_1 \longrightarrow e_1'}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \longrightarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2}\ \text{R}_{\text{le}}$$

On ne fait la substitution après le `in` que lorsque l'on a atteint une valeur dans la déclaration de x.

NB : on *n'a pas* la règle suivante :

$$\frac{e_2 \longrightarrow e_2'}{\texttt{let}\ x = v\ \texttt{in}\ e_2 \longrightarrow \texttt{let}\ x = v\ \texttt{in}\ e_2'}$$

On peut montrer que $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ se comporte comme $(\texttt{fun}\ x \rightarrow e_2)\ e_1$: comment formuler cette propriété ? Comment la prouver ?

Puisqu'une déclaration peut être "programmée" en utilisant une application et une fonction, l'ajout des déclarations n'enrichit pas le pouvoir expressif du langage. Cependant, il s'agit d'une construction très commode en FUN, qui, par ailleurs, joue un rôle particulier dans le contexte de l'inférence de types.

3.4 Ajout des références, aspects impératifs

L'extension que l'on étudie permet de manipuler des *références*, qui sont des zones de la mémoire où sont stockées des valeurs, et dont le contenu peut changer pendant que le programme s'exécute.

Syntaxe. On se donne un ensemble infini d'adresses (parfois appelées localités), notées $\ell, \ell', \ell_1, \ldots$.

$$e \quad ::= \quad \cdots \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{ref}\ e \mid e_1 := e_2 \mid !e \mid \texttt{()} \mid \ell$$

Exemple : voici un programme utilisant l'allocation, la lecture et l'écriture de référence.

```
let r = ref 3 in  (* allocation *)
let x = !r in  (* lecture *)
( r := !r+1; let y = !r in x+y )  (* écriture *)
```

Remarques :

- La séquence `;` est utilisée ci-dessus, mais ne fait pas partie de la syntaxe. Elle peut être encodée : la notation $\texttt{let}\ \_ = e_1\ \texttt{in}\ e_2$ est utilisée pour désigner $\texttt{let}\ z = e_1\ \texttt{in}\ e_2$ où z est une variable choisie de manière à ce que $z \notin v\ell(e_2)$. De plus, $e_1; e_2$ est une notation pour $\texttt{let}\ \_ = e_1\ \texttt{in}\ e_2$.
- `()` désigne la constante "unité".
- Les ℓ sont des adresses en mémoire. Leur nature n'est pas spécifiée, en particulier il n'y a pas de notion d'adresses adjacentes (ce qui serait le cas si on disait que les adresses sont des entiers).

  On peut remarquer que l'utilisateur ne manipule pas directement les adresses, au sens où elles n'apparaissent pas dans la syntaxe des programmes que l'on écrit. Dans `let r = ref 3 in e`, la variable `r` désigne une référence, et sera remplacée par une adresse au moment où ce code sera exécuté, mais c'est bien `r` qui apparaît dans `e`.
- L'ajout des références n'introduit pas de lieur. On a :

$$v\ell(\texttt{ref}\ e) = v\ell(!e) = v\ell(e) \qquad v\ell(e_1 := e_2) = v\ell(e_1) \cup v\ell(e_2) \qquad v\ell(\texttt{()}) = v\ell(\ell) = \emptyset$$

Sémantique opérationnelle à petits pas. On peut commencer par remarquer que les adresses font partie des valeurs, dans la mesure où un calcul peut renvoyer une adresse (et une adresse est considérée comme close car elle ne contient pas de variable Fun).

Valeurs : $v \quad ::= \quad k \mid \texttt{fun}\ x \to e \mid \texttt{()} \mid \ell$

Un programme Fun qui s'exécute selon la sémantique à petits pas se transforme en une forme normale (ou diverge) ; il n'y a pas de notion d'état mémoire. Avec l'ajout des références, le programme peut faire ce qu'on appelle des *effets de bords*, correspondant aux modifications de la mémoire, à l'image de ce qui se passe quand on exécute un programme Imp.

On note $\mu, \mu', \ldots$ les états mémoire (ou plus simplement états), qui sont des fonctions à domaine fini associant à chaque adresse du domaine une *valeur close*. Les notations pour manipuler les μ sont héritées de Imp :

- $\mathrm{dom}(\mu)$ désigne l'ensemble des adresses pour lesquelles μ est défini ;
- $\mu(\ell)$ n'est défini que si $\ell \in \mathrm{dom}(\mu)$, et désigne la valeur associée à ℓ dans μ ;
- $\mu[\ell \mapsto v]$ n'est défini que si $\ell \in \mathrm{dom}(\mu)$, et désigne qui coïncide avec μ, sauf pour ℓ, où la valeur est v ;
- $\mu, [\ell \mapsto v]$ n'est défini que si $\ell \notin \mathrm{dom}(\mu)$, et désigne l'état obtenu en étendant μ avec l'association $\ell \mapsto v$.

La dernière opération ci-dessus est utile lors de l'évaluation d'une expression de la forme `ref` e, qui a pour effet d'allouer une nouvelle référence.

À noter que $\mu(\ell) = \ell'$ est possible : une adresse peut être stockée à une adresse.

On définit un jugement qui s'écrit $e, \mu \longrightarrow e', \mu'$.

$$\frac{}{!\ell, \mu \longrightarrow v, \mu}\ \text{si } v = \mu(l) \quad (\mathrm{R_{read}}) \qquad \frac{}{\texttt{ref}\ v, \mu \longrightarrow \ell, (\mu, [\ell \mapsto v])} \quad (\mathrm{R_{alloc}}) \qquad \frac{}{\ell := v, \mu \longrightarrow (), \mu[l \mapsto v]} \quad (\mathrm{R_{write}})$$

$$\frac{e, \mu \longrightarrow e', \mu'}{!e, \mu \longrightarrow !e', \mu'} \quad (\mathrm{R_{rr}}) \qquad \frac{e_2, \mu \longrightarrow e_2', \mu'}{e_1 := e_2, \mu \longrightarrow e_1 := e_2', \mu'} \quad (\mathrm{R_{rwd}}) \qquad \frac{e_1, \mu \longrightarrow e_1', \mu'}{e_1 := v_2, \mu \longrightarrow e_1' := v_2, \mu'} \quad (\mathrm{R_{rwg}}) \qquad \frac{e, \mu, \longrightarrow e', \mu'}{\texttt{ref}\ e, \mu \longrightarrow \texttt{ref}\ e', \mu'} \quad (\mathrm{R_{rr}})$$

$$\frac{}{k_1 + k_2, \mu \longrightarrow k, \mu}\ k = k_1 + k_2 \quad (\mathrm{R_{rpk}}) \qquad \frac{}{(\texttt{fun}\ x\ \rightarrow\ e)\ v, \mu \longrightarrow e[v/x], \mu} \quad (\mathrm{R_{r\beta}}) \qquad \frac{e_1, \mu \longrightarrow e_1', \mu'}{e_1 + e_2, \mu \longrightarrow e_1' + e_2, \mu'} \quad (\mathrm{R_{rpg}}) \qquad \ldots$$

Ci-dessus, on n'écrit pas la totalité des règles contextuelles pour la réduction de la partie fonctionnelle : celles-ci doivent être modifiées pour prendre en compte la partie où l'on "transporte l'état", et où la réduction au sein des expressions peut engendrer un effet de bord.

On peut remarquer que dans l'affectation, on réduit d'abord le membre de droite (pourquoi doit-on, dans le cas général, réduire également à gauche ?).

Exemple : on considère $e_0 = r :=!r + 1$, et on écrit :

$$\begin{array}{rcl}
\texttt{let}\ r = \texttt{ref}\ 3\ \texttt{in}\ e_0, \emptyset & \longrightarrow & \texttt{let}\ r = \ell\ \texttt{in}\ e_0, \overbrace{[\ell \mapsto 3]}^{\mu_0} \\
& \longrightarrow & e_0[\ell/r], \mu_0 \quad = \quad \ell := !\ell + 1, \mu_0 \\
& \longrightarrow & \ell := 3 + 1, \mu_0 \\
& \longrightarrow & \ell := 4, \mu_0 \\
& \longrightarrow & (), [\ell \mapsto 4]
\end{array}$$

Configurations. La sémantique opérationnelle décrit des réductions entre *configurations*, qui sont des couples e, μ.

On se restreint aux configurations *bien formées*, ce qui signifie, intuitivement, que toutes les variables ont un sens : on interdit les variables libres (on rappelle que les adresses et les variables Fun vivent dans deux mondes différents), et on demande à ce que μ associe une valeur à toute adresse que mentionne e, et à toute adresse qui est stockée dans μ. Formellement, on note $\mathrm{addr}(e)$ l'ensemble des adresses qui apparaissent dans e : cet ensemble se définit aisément par induction sur e, dans la mesure où il n'y a pas de lieur pour les ℓ, ℓ'. On note $\mathrm{addr}(\mu)$ l'ensemble des adresses qui apparaissent dans le *codomaine* de μ : $\mathrm{addr}(\mu) = \{\ell', \exists \ell \in \mathrm{dom}(\mu), \mu(\ell) = \ell'\}$. On dit qu'une configuration e, μ est bien formée si e est close et $\mathrm{addr}(e) \cup \mathrm{addr}(\mu) \subseteq \mathrm{dom}(\mu)$.

À l'image de ce qui est fait à la partie 3.2, on démontre que si e, μ est bien formée et $e, \mu \longrightarrow e', \mu'$, alors e', μ' est bien formée. Cela nous permet de nous restreindre à ne considérer que des réductions entre configurations bien formées.

Exercice : écrire la sémantique à grands pas pour cette extension de Fun. On pourra constater que contrairement au cas "pur" (= sans aspects impératifs), les règles pour les opérateurs binaires indiquent explicitement l'ordre d'évaluation.

4 Constructions et raisonnements par induction

4.1 Justification de l'induction

4.1.1 Théorème de Knaster-Tarski

Définition 10 (Ordre partiel)**.** *Soit E un ensemble, une relation $\mathcal{R} \subseteq E^2$ est un* ordre partiel *si $\mathcal{R}$ est*

- *réflexive* $\forall x \in E, x\mathcal{R}x$
- *transitive* $\forall x, y, z \in E^3$, *si* $x\mathcal{R}y$ *et* $y\mathcal{R}z$, *alors* $x\mathcal{R}z$
- *antisymétrique* $\forall x, y \in E^2$, *si* $x\mathcal{R}y$ *et* $y\mathcal{R}x$, *alors* $x = y$

Définition 11. *Soit $(E, \sqsubseteq)$ un ordre partiel.*

Un minorant *d'une partie A de E est un $m \in E$ tel que $\forall x \in A$, $m \sqsubseteq x$. Un* majorant *de A est un $M \in E$ tel que $\forall x \in A$, $x \sqsubseteq M$.*

Un treillis complet *est un ordre partiel $(E, \sqsubseteq)$ tel que toute partie A de E admet un* plus petit majorant*, noté $\bigsqcup A$, et un* plus grand minorant*, noté $\bigsqcap A$.*

On a donc $m \sqsubseteq \bigsqcap A$ pour tout minorant m de A, et $\bigsqcup A \sqsubseteq M$ pour tout majorant M de A.

On remarque qu'un minorant de A n'est pas nécessairement dans A; même chose pour les majorants, pour $\bigsqcap A$ et pour $\bigsqcup A$.

Théorème 12 ((du point fixe) de Knaster-Tarski)**.** *Soit $(E, \sqsubseteq)$ un treillis complet.*

Soit f une fonction croissante sur E, i.e., $\forall x, y \in E^2$, si $x \sqsubseteq y$ alors $f(x) \sqsubseteq f(y)$.

On considère

$$F_f \;=\; \{x \in E, f(x) \sqsubseteq x\},$$

l'ensemble des prépoints fixes *de f.*

On pose $m = \bigsqcap F_f$. Alors m est un point fixe de f (au sens où $f(m) = m$).

m est ainsi le plus petit (pré)point fixe de f.

4.1.2 Définitions inductives d'ensembles, preuves par induction structurelle

Pour le besoin de ce cours, nous appliquerons le théorème de Knaster-Tarski au treillis $\mathcal{P}(E)$ des parties d'un ensemble E, ordonné par la relation d'inclusion $\subseteq$.

On se donne une définition inductive, avec un nombre fini de constructeurs. Cette définition inductive peut être associée à une grammaire (il y a alors autant de constructeurs qu'il y a de cas dans la grammaire). À cette définition inductive correspond une fonction sur $\mathcal{P}(E)$, croissante, qui détermine un ensemble en appliquant Knaster-Tarski. On appellera cet ensemble *"l'ensemble determiné par la définition inductive"* ; dans le cas d'une grammaire, cet ensemble est *"l'ensemble défini inductivement par la grammaire"*.

Nous illustrons cela ci-dessous sur un exemple, la définition inductive d'Arith, l'ensemble des expressions arithmétiques de Imp.

Définition inductive d'ensemble. La grammaire pour les expressions arithmétiques de Imp, s'écrit

$$a \qquad ::= \qquad k \;\big|\; X \;\big|\; a_1 + a_2$$

Elle correspond à une définition inductive, que l'on formule en s'appuyant

- sur l'existence des entiers relatifs, $\mathbb{Z}$, et d'un ensemble infini $\mathcal{V}$ de variables Imp
- sur la donnée de trois *constructeurs*.

En Coq, on ferait une définition qui ressemblerait à

```
Inductive aexpr :  Set :=
  | Cst :  ℤ→aexpr
  | Var :  𝒱→aexpr
  | Plus :  aexpr→aexpr→aexpr.
```

Il est plus simple ici de supposer que le troisième constructeur "a le type `(aexpr*aexpr)→aexpr`", autrement dit, que si a_1 et a_2 sont des expressions arithmétiques, alors $\texttt{Plus}(a_1, a_2)$ est une expression arithmétique (c'est un point qui a déjà été abordé plus haut, cf. partie 1.1.2).

À cette définition est associée une fonction $f : \mathcal{P}(E) \to \mathcal{P}(E)$, définie de la manière suivante. Si $A \subseteq E$,

$$f(A) \;=\; \{\texttt{Cst}(k), k \in \mathbb{Z}\} \cup \{\texttt{Var}(X), X \in \mathcal{V}\} \cup \{\texttt{Plus}(a_1, a_2), a_1, a_2 \in A^2\}.$$

L'intuition est que f construit un ensemble à partir de A en appliquant les trois constructeurs des expressions arithmétiques. f ajoute toutes les constantes entières et toutes les variables (deux premiers constructeurs), et *construit* tous les arbres binaires en ajoutant le constructeur `Plus` à partir des éléments de A.

On peut vérifier que f *est croissante*.

On a par exemple

$$f(\{\heartsuit, \clubsuit\}) = \{\texttt{Cst}(k), k \in \mathbb{Z}\} \cup \{\texttt{Var}(X), X \in \mathcal{V}\} \cup \{\texttt{Plus}(\heartsuit, \heartsuit), \texttt{Plus}(\heartsuit, \clubsuit), \texttt{Plus}(\clubsuit, \heartsuit), \texttt{Plus}(\clubsuit, \clubsuit)\}$$

où $\{\heartsuit, \clubsuit\}$ est un ensemble plus ou moins arbitraire (que l'on donne ici à titre d'exemple).

Qui est E, l'*"ensemble ambiant"*, dans $\mathcal{P}(E)$? On peut par exemple dire que f détermine un sous-ensemble de tous les arbres dont les feuilles peuvent être décorées par des éléments des ensembles pré-existants.

Éléments infinis : dans l'ensemble des prépoints fixes de f, il y a des ensembles qui contiennent des éléments infinis (p.ex. le peigne infini de cœurs, $\texttt{Plus}(\heartsuit, \texttt{Plus}(\heartsuit, \texttt{Plus}(\ldots)))$, si l'on considère un prépoint fixe qui contient $\heartsuit$; ou alors, dans le cas des entiers de Peano, l'entier infini $S(S(S(\ldots)))$.

Pour résumer, à une définition inductive telle que la définition d'`aexpr` vue ci-dessus, on associe la fonction f que l'on a donnée. Le théorème de Knaster-Tarski nous dit que f a un plus petit point fixe. La convention est qu'en donnant la définition inductive, on définit cet ensemble, que l'on appelle en l'occurrence Arith.

Preuve par induction sur la structure d'une expression arithmétique. On veut prouver $\forall a, \mathsf{P}(a)$, où $\mathsf{P}(\cdot)$ est un prédicat portant sur les expressions arithmétiques.

Pour appliquer le théorème de Knaster-Tarski, on considère

$$A \;=\; \{a \in \text{Arith telle que } \mathsf{P}(a) \text{ est vrai}\}.$$

On a $A \subseteq$ Arith par définition. Afin de montrer $A =$ Arith, qui revient à $\forall a, \mathsf{P}(a)$, on établit Arith $\subseteq A$.

Cette inclusion est à son tour démontrée en prouvant que $f(A) \subseteq A$, puisqu'à partir du moment où A est un prépoint fixe de f, A contient le plus petit prépoint fixe de f, qui est Arith.

Reste à expliciter ce que signifie montrer $f(A) \subseteq A$: il faut prouver 3 choses (car il y a 3 constructeurs dans la définition d'Arith) :

1. $\forall k \in \mathbb{Z}, \mathsf{P}(\texttt{Cst}(k))$ (ce que l'on note usuellement $\forall k \in \mathbb{Z}, \mathsf{P}(k)$)
2. $\forall X \in \mathcal{V}, \mathsf{P}(X)$ (en adoptant une notation similaire)
3. $\forall a_1, a_2 \in \text{Arith}^2$, si $\mathsf{P}(a_1)$ et $\mathsf{P}(a_2)$ sont vrais, alors on a $\mathsf{P}(\texttt{Plus}(a_1, a_2))$

Au total, la définition inductive de Arith détermine à la fois un ensemble d'objets (des arbres étiquetés, si on veut), et un principe de raisonnement sur cet ensemble (pour prouver qu'une propriété est vraie pour tous les éléments d'Arith, il faut prouver les 3 propriétés ci-dessus).

4.1.3 Définitions inductives de relations : règles d'inférence, induction sur la relation

On peut appliquer le théorème de Knaster-Tarski pour définir des sous-ensembles d'un ensemble donné : c'est ce qu'on appelle définir inductivement une relation (ou un prédicat). Pour ce faire, on s'appuie sur des règles d'inférence. Les règles d'inférence sont le pendant pour les relations inductives des constructeurs pour les ensembles inductifs (les règles d'inférence généralisent les constructeurs).

Prenons l'exemple de l'évaluation des expressions arithmétiques IMP. Cette relation ternaire est un sous-ensemble du produit cartésien $\underline{\text{Arith}} \times \mathcal{M} \times \mathbb{Z}$. À ces trois règles d'inférence est associée une fonction f sur les parties de cet ensemble, définie, pour $A \subseteq \underline{\text{Arith}} \times \mathcal{M} \times \mathbb{Z}$, par

$$\begin{array}{rcl} f(A) & = & \{(k,\sigma,k), \forall k,\sigma\} \quad \cup \quad \{(X,\sigma,\sigma(k)), \forall X,\sigma\} \\ & \cup & \{(a_1+a_2,\sigma,k), \forall a_1,a_2,\sigma,k, \text{ avec } \begin{array}{l} \cdot\ \exists k_1,k_2 \in \mathbb{Z}^2, (a_1,\sigma,k_1) \in A \text{ et } (a_2,\sigma,k_2) \in A \\ \cdot\ k \text{ est la somme de } k_1 \text{ et } k_2 \end{array} \} \end{array}$$

(dans les deux premières composantes de $f(A)$, on utilise implicitement l'injection des entiers et des variables dans les expressions arithmétiques ; ainsi, dans (k,σ,k), le premier k est une expression, le second est un entier).

On peut montrer que f est croissante. Par Knaster-Tarski, les trois règles d'inférence ci-dessus déterminent donc un ensemble E_a de triplets, plus petit pré-point fixe de f. On note $a,\sigma \Downarrow k$ un triplet appartenant à E_a.

Preuve par induction sur l'évaluation des expressions arithmétiques. Le principe d'induction que donne le théorème de Knaster-Tarski permet de prouver des énoncés de la forme

$$\forall a,\sigma,k, (a,\sigma \Downarrow k) \ \Rightarrow\ \mathsf{P}(a,\sigma,k). \tag{1}$$

Il s'agit de montrer des prédicats portant sur des triplets (a,σ,k), en supposant que l'on a $a,\sigma \Downarrow k$, i.e., en se restreignant aux triplets qui appartiennent au plus petit prépoint fixe de f.

Pour montrer l'implication (1), on considère $A = \{(a,\sigma,k) \text{ t.q. } \mathsf{P}(a,\sigma,k)\}$.

On montre $f(A) \subseteq A$, ce que l'on appelle *faire une preuve par induction sur la relation d'évaluation des expressions arithmétiques*, ou, plus simplement, une preuve par induction sur l'évaluation des expressions arithmétiques.

Démontrer $f(A) \subseteq A$ se décompose en trois énoncés à prouver :

1. $\forall k,\sigma,\ \mathsf{P}(k,\sigma,k)$
2. $\forall X,\sigma,\ \mathsf{P}(X,\sigma,\sigma(X))$
3. $\forall a_1,a_2,\sigma,k_1,k_2,k$, si on suppose $\begin{array}{l} H_1 : \mathsf{P}(a_1,\sigma,k_1) \\ H_2 : \mathsf{P}(a_2,\sigma,k_2) \\ k \text{ est la somme de } k_1 \text{ et } k_2 \end{array}$ alors $\mathsf{P}(a_1+a_2,\sigma,k)$

Par application du théorème de Knaster-Tarski, on déduit de $f(A) \subseteq A$ que $E_a \subseteq A$, ce qui correspond bien à l'implication (1) ci-dessus.

Remarque : il peut aussi être utile de considérer

$$A' = A \cap E_a = \{(a,\sigma,k),\ a,\sigma \Downarrow k \wedge \mathsf{P}(a,\sigma,k)\}.$$

Montrer $f(A') \subseteq A'$ revient à faire la même preuve que plus haut, avec, dans le troisième cas, les hypothèses $H_i : a_i,\sigma \Downarrow k_i \wedge \mathsf{P}(a_i,\sigma,k_i)$, $i = 1,2$.

On retrouve ce principe dans l'induction sur une relation en Coq :

```
Inductive petit : nat -> nat -> Prop :=
| p0 : forall n, petit 0 n
| pS : forall n k, petit n k -> petit (S n) (S k).

Lemma petit3 : forall n k, petit n k -> petit n (k+3).
intros n k H.
   (* n, k : nat
      H : petit n k
      ____________________________________(1/1)
      petit n (k + 3)   *)
induction H.
constructor.
   (* n, k : nat
      H : petit n k
      IHpetit : petit n (k + 3)
      ____________________________________(1/1)
      petit (S n) (S k + 3) *)
```

Outre l'hypothèse d'induction `IHpetit`, on dispose de l'hypothèse `H`, qui traduit le fait que l'on se restreint aux couples `n,k` satisfaisant `petit n k`.

Croissance de f et prémisses "positives". On n'a pas détaillé la preuve de croissance de f. C'est une propriété immédiate et peu palpitante, à partir du moment où les règles d'inférence sont "raisonnables". Raisonnable signifie en particulier qu'il y a un nombre fini de prémisses, et *il n'y a pas de prémisse négative.*

L'ensemble des dérivations de $a, \sigma \Downarrow k$. Remarquons que l'on peut adopter, pour comprendre la définition de $a, \sigma \Downarrow k$, le point de vue des définitions inductives d'ensembles. Cela revient à voir la définition de l'évaluation des expressions arithmétiques comme la définition d'un *ensemble*, en l'occurrence, l'ensemble des dérivations d'un jugement de la forme $a, \sigma \Downarrow k$.

Aux trois règles d'inférence définissant l'évaluation des expressions arithmétiques est associée une fonction f_d sur les parties de l'ensemble des arbres étiquetés dont la racine est étiquetée par un triplet $\in \underline{\text{Arith}} \times \mathcal{M} \times \mathbb{Z}$, f_d étant défini par

$$f_d(A) = \{\forall k, \sigma, \frac{}{k, \sigma \Downarrow k}\} \cup \{\forall X, \sigma, \frac{}{X, \sigma \Downarrow \sigma(k)}\}$$
$$\cup \{\delta = \frac{\delta_1 \qquad \delta_2}{a_1 + a_2, \sigma \Downarrow k}, \text{ avec } \begin{cases} a_1, a_2 \in \underline{\text{Arith}}^2 \\ k_1, k_2 \in \mathbb{Z}^2 \\ k \text{ est la somme de } k_1 \text{ et } k_2 \\ \delta_i \in A \text{ est une dérivation de } a_i, \sigma \Downarrow k_i, i = 1, 2 \end{cases} \}$$

On peut montrer que f_d est croissante. En appliquant le théorème de Knaster-Tarski, on associe donc aux trois règles d'inférence ci-dessus un ensemble de dérivations, qui ont toutes, à la racine, un jugement de la forme $a, \sigma \Downarrow k$.

À partir de cet ensemble de dérivations, on retrouve l'ensemble E_a défini plus haut, en définissant E_a comme l'ensemble des triplets tels qu'il existe une dérivation décorée à la racine par ce triplet.

Le principe de preuve que donne Knaster-Tarski est alors la preuve *par induction sur la dérivation*, qui revient à raisonner sur la structure d'une dérivation de $a, \sigma \Downarrow k$ ("on décortique une dérivation").

Digression. On peut remarquer que dans le cas général, il y a "moins" d'éléments dans E_a que dans l'ensemble des dérivations, puisque suivant comment sont définies les règles d'inférence, il peut y avoir plusieurs dérivations ayant le même triplet en conclusion.
Par exemple, c'est le cas si on considère aussi la multiplication dans Arith*, et, en plus de la règle évidente pour la multiplication, on on ajoute les règles "optimisées"*

$$\frac{}{a_1 * 0, \sigma \Downarrow 0} \qquad \frac{}{0 * a_2, \sigma \Downarrow 0}$$

4.2 Définitions inductives — forme, conseils de rédaction

en rouge, ce qui est exigé dans une copie

Définitions inductives.

- on se donne
  des ensembles pré-existants, des relations pré-existantes
- on définit par induction
  - soit un ensemble, donné par les constructeurs suivants ... (ou par la grammaire suivante ...)
  - soit un jugement, noté ..., *[[précisions éventuelles sur les metavariables apparaissant dans le jugement]]*, (p.ex. *noté $\sigma, c \Downarrow \sigma'$, où c est une commande et σ, σ' sont des états mémoire*)
    donné par les règles d'inférence suivantes ...

Preuves par induction.

- on démontre l'énoncé suivant: *[[écrire l'énoncé]]* par induction
  - sur la structure de toto
    lorsque toto est un élément d'un ensemble E défini inductivement et l'énoncé est de la forme $\forall toto \in E, blabla$ ou bien
  - sur la relation titi
    lorsque titi est un jugement défini inductivement et l'énoncé est de la forme $\forall..\forall.., (titi) \Rightarrow blabla$

- il y a k cas:
  cas 1: *ici il faut être raisonnable: détailler les cas qui le méritent,*
  ... *regrouper les cas qui sont traités identiquement — on n'est*
  cas k: *pas des gallinacés*

  l'hypothèse d'induction dit *blabla*, ce qui permet de déduire... (énoncer l'hypothèse d'induction)

Définition de fonctions. lorsque E est (un ensemble/ un prédicat) défini par induction
on définit la fonction f par induction sur son argument $toto \in E$, il y a k cas
NB: *toto* peut aussi être une dérivation

5 Raisonnements inductifs, preuves de propriétés de Imp et Fun

5.1 Commandes Imp : preuve par induction sur la structure, sur la relation d'évaluation.

Par application du théorème de Knaster-Tarski, le principe de preuve par induction sur la structure d'une commande Imp permet d'établir $\forall c, \mathsf{P}(c)$, où $\mathsf{P}(\cdot)$ est un prédicat portant sur une commande Imp.

Il y a 5 propriétés à prouver, car il y a 5 constructeurs pour les commandes :

- $\mathsf{P}(X := a)$ pour toute variable X et expression arithmétique a
- $\mathsf{P}(\texttt{skip})$
- $\mathsf{P}(c_1; c_2)$, en supposant les deux hypothèses d'induction $HI_1 : \mathsf{P}(c_1)$ et $HI_2 : \mathsf{P}(c_2)$

- $\mathsf{P}(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)$, en supposant les deux hypothèses d'induction $HI_1 : \mathsf{P}(c_1)$ et $HI_2 : \mathsf{P}(c_2)$
- $\mathsf{P}(\texttt{while}\ b\ \texttt{do}\ c)$, en supposant l'hypothèse d'induction $HI : \mathsf{P}(c)$

Les hypothèses d'induction portent sur les arguments des constructeurs pour lesquels il fait sens d'écrire $\mathsf{P}(\cdot)$, à savoir les commandes : ainsi, par exemple, pas de $\mathsf{P}(b)$ dans le cas d'un branchement conditionnel ou d'une boucle.

S'agissant de la preuve par induction sur la relation $c, \sigma \Downarrow \sigma'$, on démontre

$$\forall c, \sigma, \sigma',\ (c, \sigma \Downarrow \sigma') \ \Rightarrow\ \mathsf{P}(c, \sigma, \sigma'),$$

où $\mathsf{P}(\cdots)$ porte sur des triplets dans $\underline{\mathrm{Com}} \times \mathcal{M} \times \mathcal{M}$, en établissant 7 propriétés (car il y a 7 règles d'inférence pour la relation $c, \sigma \Downarrow \sigma'$) :

- $\mathsf{P}(\texttt{skip}, \sigma, \sigma)$
- $\mathsf{P}(X := a, \sigma, \sigma[X \mapsto k])$, en supposant $a, \sigma \Downarrow k$.
- $\mathsf{P}(c_1; c_2, \sigma, \sigma'')$, en supposant les deux hypothèses d'induction $HI_1 : \mathsf{P}(c_1, \sigma, \sigma')$ et $HI_2 : \mathsf{P}(c_2, \sigma', \sigma'')$
- $\mathsf{P}(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma, \sigma')$, en supposant l'hypothèse d'induction $HI : \mathsf{P}(c_1, \sigma, \sigma')$, et en supposant que l'on a $b, \sigma \Downarrow \texttt{true}$
- $\mathsf{P}(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma, \sigma')$, en supposant l'hypothèse d'induction $HI : \mathsf{P}(c_2, \sigma, \sigma')$, et en supposant que l'on a $b, \sigma \Downarrow \texttt{false}$
- $\mathsf{P}(\texttt{while}\ b\ \texttt{do}\ c, \sigma, \sigma)$, en supposant $b, \sigma \Downarrow \texttt{false}$
- $\mathsf{P}(\texttt{while}\ b\ \texttt{do}\ c, \sigma, \sigma'')$, en supposant les deux hypothèses d'induction $HI_1 : \mathsf{P}(c, \sigma, \sigma')$ et $HI_2 : \mathsf{P}(\texttt{while}\ b\ \texttt{do}\ c, \sigma', \sigma'')$, et en supposant que l'on a $b, \sigma \Downarrow \texttt{true}$

5.2 Déterminisme de l'évaluation des commandes Imp

On ne développe pas la preuve des deux lemmes suivants (ces preuves sont plus simples que la preuve de la proposition 15, tout en étant plus simples) :

Lemme 13. *$\forall a, \sigma, k_1$, si $a, \sigma \Downarrow k_1$, alors $\forall k_2$, si $a, \sigma \Downarrow k_2$, on a $k_1 = k_2$.*

Lemme 14. *$\forall b, \sigma, t_1$, si $b, \sigma \Downarrow t_1$, alors $\forall t_2$, si $b, \sigma \Downarrow t_2$, on a $t_1 = t_2$.*

Proposition 15. *$\forall c, \sigma, \sigma_1$, si $c, \sigma \Downarrow \sigma_1$, alors $\forall \sigma_2$, si $c, \sigma \Downarrow \sigma_2$, on a $\sigma_1 = \sigma_2$.*

Preuve. On montre

$$\forall c, \sigma, \sigma_1,\ \text{si } c, \sigma \Downarrow \sigma_1, \text{ alors } \underbrace{\forall \sigma_2, \text{ si } c, \sigma \Downarrow \sigma_2 \text{ on a } \sigma_1 = \sigma_2}_{\mathsf{P}(c, \sigma, \sigma_1)}$$

par induction sur la relation $c, \sigma \Downarrow \sigma_1$.

Il y a 7 cas, un par règle d'inférence qui définit les grands pas pour Imp.

règle $\mathrm{E}_{\mathrm{skip}}$.

On a $c = \texttt{skip}$, et nécessairement $\sigma_1 = \sigma$. Supposons $c, \sigma \Downarrow \sigma_2$, alors on a nécessairement appliqué la règle $\mathrm{E}_{\mathrm{skip}}$ pour dériver cela, et $\sigma_2 = \sigma = \sigma_1$.

règle $\mathrm{E}_{\mathrm{assgn}}$.

On a $c = X := a$, et nécessairement, $\exists k$ t.q. $a, \sigma \Downarrow k$ et $\sigma_1 = \sigma[X \mapsto k]$. Supposons $c, \sigma \Downarrow \sigma_2$, alors on a nécessairement appliqué la règle $\mathrm{E}_{\mathrm{assgn}}$, et on a k' tel que $a, \sigma \Downarrow k'$ et $\sigma_2 = \sigma[X \mapsto k']$. Par le lemme 13, on déduit $k' = k$, ce qui donne $\sigma_1 = \sigma_2$.

règle $\mathrm{E}_{\mathrm{seq}}$.

On a $c = c_1; c_2$, et nécessairement il existe σ'_1 tel que $c_1, \sigma \Downarrow \sigma'_1$ et $c_2, \sigma'_1 \Downarrow \sigma_1$.

Les hypothèses d'induction sont $HI_1 : \mathsf{P}(c_1, \sigma, \sigma'_1)$ et $HI_2 : \mathsf{P}(c_2, \sigma'_1, \sigma_1)$.

Supposons $c, \sigma \Downarrow \sigma_2$, alors on a nécessairement appliqué la règle $\mathrm{E}_{\mathrm{seq}}$, et on a σ'_2 tel que $c_1, \sigma \Downarrow \sigma'_2$ et $c_2, \sigma'_2 \Downarrow \sigma_2$.

Avec HI_1 on déduit $\sigma'_1 = \sigma'_2$, ce qui permet d'appliquer HI_2 pour déduire $\sigma_1 = \sigma_2$.

règle E_{it}.

On a $c = \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2$, et nécessairement $b, \sigma \Downarrow \texttt{true}$ et $c_1, \sigma \Downarrow \sigma_1$.

L'hypothèse d'induction est $HI : \mathsf{P}(c_1, \sigma, \sigma_1)$.

Supposons $c, \sigma \Downarrow \sigma_2$. Étant donné que $c = \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2$, soit la règle E_{it} soit la règle E_{if} a été appliquée pour dériver ce triplet. Le cas E_{if} est exclu : en effet, cela donnerait $b, \sigma \Downarrow \texttt{false}$, ce qui, par le lemme 14, est contradictoire avec $b, \sigma \Downarrow \texttt{true}$.

Donc E_{it} a été utilisée pour dériver $c, \sigma \Downarrow \sigma_2$, ce qui donne $b, \sigma \Downarrow \texttt{true}$, et $c_1, \sigma \Downarrow \sigma_2$.

On peut alors appliquer HI pour déduire $\sigma_1 = \sigma_2$.

règle E_{if}. Ce cas est traité comme le précédent.

règle E_{wf}.

On a $c = \texttt{while } b \texttt{ do } c_0$, et nécessairement $b, \sigma \Downarrow \texttt{false}$ et $\sigma_1 = \sigma$.

Supposons $c, \sigma \Downarrow \sigma_2$, alors en raisonnant comme dans le cas de la règle E_{it} ci-dessus, on déduit que la règle E_{wf} a nécessairement utilisée pour dériver $c, \sigma \Downarrow \sigma_2$, ce qui donne $\sigma_2 = \sigma = \sigma_1$.

règle E_{wt}.

On a $c = \texttt{while } b \texttt{ do } c_0$, et nécessairement $b, \sigma \Downarrow \texttt{true}$ et il existe σ'_1 tel que $c_0, \sigma \Downarrow \sigma'_1$ et $c, \sigma'_1 \Downarrow \sigma_1$.

Les hypothèses d'induction sont

$$HI_1 : \mathsf{P}(c_0, \sigma, \sigma'_1) \qquad \text{et} \qquad HI_2 : \mathsf{P}(\texttt{while } b \texttt{ do } c_0, \sigma'_1, \sigma_1)$$

Supposons $c, \sigma \Downarrow \sigma_2$. En raisonnant comme plus haut, on sait que la règle E_{wt} a nécessairement été utilisée, ce qui donne l'existence de σ'_2 tel que $c_0, \sigma \Downarrow \sigma'_2$ et $c, \sigma'_2 \Downarrow \sigma_2$.

Le fait de savoir $c_0, \sigma \Downarrow \sigma'_2$ permet d'appliquer HI_1 pour déduire $\sigma'_1 = \sigma'_2$; cela permet d'appliquer HI_2 pour avoir finalement $\sigma_1 = \sigma_2$. □

Le cas central dans la preuve ci-dessus est le dernier, qui correspond à la règle E_{wt}. Dans ce cas, la seconde hypothèse d'induction porte sur la même commande que la propriété que l'on prouve, à savoir c. La preuve par induction nous permet de récupérer la propriété pour le triplet $(\texttt{while } b \texttt{ do } c_0, \sigma'_1, \sigma_1)$, qui correspond à une *sous-dérivation* de la dérivation de $\texttt{while } b \texttt{ do } c_0, \sigma \Downarrow \sigma_1$.

Remarque 16 (Échec de la preuve par induction sur la structure de c)**.** *Il est intéressant de comprendre où la preuve de la proposition 15 échoue si l'on procède par induction sur la commande c, pour (tenter d')établir*

$$\forall c,\ \underbrace{\forall \sigma, \sigma_1,\ \textit{si } c, \sigma \Downarrow \sigma_1,\ \textit{alors } \forall \sigma_2,\ \textit{si } c, \sigma \Downarrow \sigma_2,\ \textit{alors } \sigma_1 = \sigma_2}_{\mathsf{P}(c)}.$$

On s'intéresse au cas $c = \texttt{while } b \texttt{ do } c_0$. L'hypothèse d'induction est $HI : \mathsf{P}(c_0)$: elle porte sur c_0, qui est une sous-commande de c.

On suppose alors $c, \sigma \Downarrow \sigma_1$, et distingue deux cas suivant si c'est E_{wt} ou E_{wf} qui a été utilisée pour dériver ce triplet. On considère le cas où c'est E_{wt} : alors, on sait $b, \sigma \Downarrow \texttt{true}$, et il existe σ'_1 tel que $c_0, \sigma \Downarrow \sigma'_1$ et $c, \sigma'_1 \Downarrow \sigma_1$.

On suppose ensuite $c, \sigma \Downarrow \sigma_2$, on remarque que nécessairement E_{wt} a été employée, ce qui donne $c_0, \sigma \Downarrow \sigma'_2$ et $c, \sigma'_2 \Downarrow \sigma_2$ pour un certain σ'_2.

De $c_0, \sigma \Downarrow \sigma'_1$ et $c_0, \sigma \Downarrow \sigma'_2$, on déduit $\sigma'_1 = \sigma'_2$ grâce à HI. Mais on est ensuite bloqué, car on doit à nouveau raisonner sur c, et non sur c_0 : intuitivement, on se fait "happer par la boucle" si on essaye de raisonner à partir de $c, \sigma'_1 \Downarrow \sigma_1$ et $c, \sigma'_1 \Downarrow \sigma_2$.

6 Réécriture, terminaison, unification

6.1 Réécriture abstraite

Définition 17. *Un* système de réécriture abstraite *(SRA) est la donnée d'une relation binaire, souvent notée $\to$, sur un ensemble A. On notera $(A, \to)$, ou plus simplement $\to$. On adopte souvent la notation infixe $x \to y$ pour $(x, y) \in \to$.*

Un SRA est une relation sur A^2, on parle parfois simplement de relation plutôt que de SRA.

6.1.1 Terminaison

Définition 18. $(A, \to)$ termine *(ou* est terminante*) s'il n'existe pas de suite infinie d'éléments de A $(x_i)_{i\geq 0}$ telle que $\forall i \geq 0, x_i \to x_{i+1}$.*

Si $\to$ ne termine pas, autrement dit s'il existe $(x_i)_{i\geq 0}$ avec $\forall i \geq 0, x_i \to x_{i+1}$, on dit que $\to$ diverge *(ou* est divergente*).*

Théorème 19 (induction bien fondée / induction nœthérienne)**.** *Une relation $(A, \to)$ est terminante ssi elle satisfait le* principe d'induction bien fondée *(PIBF) suivant :*

> *Pour tout prédicat $\mathsf{P}(\cdot)$ sur A,*
> *si, pour tout $x \in A$, $(\forall y \in A, x \to y \Rightarrow \mathsf{P}(y))$ implique $\mathsf{P}(x)$,*
> *alors $\forall x \in A, \mathsf{P}(x)$.*

Preuve. L'implication $\Leftarrow$ se prouve en prenant $\mathsf{P}(x)$ = "il n'y a pas de chaîne infinie issue de x".

Pour l'implication $\Rightarrow$, on prouve la contraposée : on suppose qu'il existe $\mathsf{P}(\cdot)$ satisfaisant la propriété d' "hérédité" du PIBF et tel que $\neg\mathsf{P}(x_0)$ pour un certain $x_0 \in A$. On en déduit l'existence de $x_1 \in A$ tel que $x_0 \to x_1$ et $\neg\mathsf{P}(x_1)$. On itère cette construction pour obtenir une chaîne infinie à partir de x_0. □

Exemple d'application du principe d'induction bien fondée :

Lemme 20 (de König)**.** *On dit qu'un arbre est fini s'il a un nombre fini de nœuds, et il est dit infini dans le cas contraire.*

Un arbre est à branchement fini *si chaque nœud a un nombre fini de fils (immédiats).*

Si un arbre à branchement fini est infini, alors il admet une branche infinie, qui est une suite $(n_i)_{i\geq 0}$ de nœuds telle que n_0 est la racine et $\forall i \geq 0$, n_{i+1} est fils (immédiat) de n_i.

Preuve. On applique le PIBF, en posant $\mathsf{P}(n)$ = "le sous-arbre dont n est la racine est fini". □

Pour illustrer l'application du PIBF, on peut revenir sur le théorème de Knaster-Tarski, vu sous l'angle de la terminaison.

6.1.2 Confluence

Définition 21. *On dit qu'une relation binaire $\longrightarrow$ sur un ensemble A :*

- *est* confluente en $t \in A$ *ssi*

$$\forall u_1, u_2 \text{ t.q. } t \longrightarrow^* u_1 \text{ et } t \longrightarrow^* u_2,\ \exists t' \text{ t.q. } u_1 \longrightarrow^* t' \text{ et } u_2 \longrightarrow^* t'$$

  où $\longrightarrow^*$ désigne la clôture réflexive et transitive de $\longrightarrow$.

  Notation : $u_1\, {}^*\!\leftarrow t \longrightarrow^* u_2$ implique $\exists t'$ t.q. $u_1 \longrightarrow^* t'\, {}^*\!\leftarrow u_2$.

- *est* confluente *ssi $\forall t \in A$, elle est confluente en t.*
- *a la* propriété du diamant *ssi $\forall t, u_1, u_2$, si $u_1 \leftarrow t \longrightarrow u_2$, alors $\exists t'$ t.q. $u_1 \longrightarrow t' \leftarrow u_2$.*
- *est* localement confluente *ssi $\forall t, u_1, u_2$, si $u_1 \leftarrow t \longrightarrow u_2$, alors $\exists t'$ t.q. $u_1 \longrightarrow^* t'\, {}^*\!\leftarrow u_2$.*

- *a la propriété de Church-Rosser ssi* $\forall t_1, t_2$, *si* $t_1 \longleftrightarrow^* t_2$, *alors* $\exists t'$ *t.q.* $t_1 \longrightarrow^* t' \ ^* \longleftarrow t_2$,
  avec $\longleftrightarrow^*$ *défini de la manière suivante :* $\longleftarrow = (\longrightarrow)^{-1} = \{(y,x), x \longrightarrow y\}$, $\longleftrightarrow = \longrightarrow \cup \longleftarrow$, *et* $\longleftrightarrow^*$ *est la clôture réflexive et transitive de* $\longleftrightarrow$.

$\longleftrightarrow^*$ est l'équivalence engendrée par $\longrightarrow$. Inversement, on peut se poser la question consistant à orienter des égalités, pour faire émerger une notion de calcul à partir d'une notion d'équivalence.

Pour une relation déterministe, ou fonctionnelle, la question de la confluence est triviale. La confluence correspond à une notion de déterminisme lorsque la relation elle-même n'est pas déterministe.

Lemme 22 (de Newman)**.** *Soit* $(A, \longrightarrow)$ *un SRA. Si* $\longrightarrow$ *est localement confluente et terminante, alors* $\longrightarrow$ *est confluente.*

Preuve. On démontre $\forall t \in A, \mathsf{P}(t)$, avec $\mathsf{P}(t)$ défini comme "$\longrightarrow$ est confluente en t", en appliquant le principe d'induction bien fondée. □

La confluence locale est un outil souvent employé pour démontrer la confluence, en utilisant le lemme de Newman. On peut comprendre l'adjectif "locale" en observant que l'on considère juste $\longrightarrow$, et pas $\longrightarrow^*$: on regarde ce qui se passe "à proximité", et pas "loin".

6.2 Systèmes de réécriture de mots (SRM)

Définition 23. *Soit* Σ *un ensemble de caractères (/lettres/symboles). Un* mot sur Σ *est une suite finie d'éléments de* Σ. *On note* Σ^* *l'ensemble des mots sur* Σ. *On note généralement* ε *le mot vide, et* uv *désigne le mot obtenu en concaténant les mots* u *et* v, *pour* $u, v \in \Sigma^*$.

Définition 24 (Système de réécriture de mots, SRM)**.** *Soit* Σ *un ensemble de lettres. Un SRM sur* Σ *est donné par un ensemble* $R \subseteq \Sigma^* \times \Sigma^*$ *de couples de mots sur* Σ, *noté généralement* $R = \{(\ell, r), (\ell, r) \in {\Sigma^*}^2, \ell \neq \varepsilon\}$ *(la première composante de chaque couple doit être non vide).*

R détermine une relation binaire sur Σ^*, *notée* $\to_R$, *définie par*

$$u \to_R v \quad ssi \quad \exists (x,y) \in {\Sigma^*}^2, \exists (\ell, r) \in R \text{ t.q. } u = x\ell y \text{ et } v = xry.$$

On remarque que tout SRM peut être vu comme un SRA.

Exemple 25. *On prend* $\Sigma = \{a, b, c\}$. *On définit* $R_0 = \{(ab, ac), (ccc, bb), (aa, a)\}$, *ce que l'on note souvent*

$$R_0 = \begin{array}{l} ab \longrightarrow ac \\ ccc \longrightarrow bb \\ aa \longrightarrow a \end{array}$$

On a par exemple $aabc \to_{R_0} aacc$, *et* $aabc \to_{R_0} abc$.

Exemple 26 (Cailloux)**.** *On considère* $\Sigma = \{\circ, \bullet\}$, *et* $R = \circ\bullet \longrightarrow \bullet,\ \bullet\circ \longrightarrow \bullet,\ \circ\circ \longrightarrow \circ,\ \bullet\bullet \longrightarrow \circ$.

Étude de la confluence locale des SRM. Soit R un système de réecriture de mots, et soient t, u_1, u_2 tels que $t \to_R u_1$ et $t \to_R u_2$. On cherche à *joindre* u_1 et u_2, ce qui signifie exhiber v tel que $u_1 \to_R^* v$ et $u_2 \to_R^* v$; on dit alors que la paire $\{u_1, u_2\}$ est *joignable*, notion qui existe aussi dans les SRA.

On revient à la définition de $\to_R$, et on procède à une étude de cas, qui porte sur la manière dont t peut être réécrit de deux façons. Trois siuations sont possibles :

1. $t = x\ell_1 y\ell_2 z$, $u_1 = xr_1 y\ell_2 z$, $u_2 = x\ell_1 y r_2 z$, les deux règles de réécriture sont $\{(\ell_1, r_1), (\ell_2, r_2)\} \subseteq R$.

   Les deux redex ℓ_1 et ℓ_2 sont *indépendants*, et on a la propriété du diamant : $u_1 \to_R v$ et $u_2 \to_R v$, avec $v = xr_1 y r_2 z$.

2. $t = x\ell_1\ell_2\ell_3 y$, $u_1 = xr_1 y$, $u_2 = x\ell_1 r_2 \ell_3 y$, les deux règles de réécriture sont $\{(\ell_1\ell_2\ell_3, r_1), (\ell_2, r_2)\} \subseteq R$.

   Ici, le second redex est *inclus* dans le premier.

   On élimine les mots x et y, qui ne jouent pas de rôle ici, et on dit que $\{r_1, \ell_1 r_2 \ell_3\}$ est une *paire critique* de R.

3. $t = x\ell_1\ell_2\ell_3 y$, $u_1 = xr_1\ell_3 y$, $u_2 = x\ell_1 r_2 y$, les deux règles de réécriture sont $\{(\ell_1\ell_2, r_1), (\ell_2\ell_3, r_2)\} \subseteq R$, avec, par ailleurs $\ell_i \neq \varepsilon$ pour $i \in \{1, 2, 3\}$ (sinon on est ramené au cas précédent).

   Il y a ici un *conflit de ressources* entre les deux règles sur le sous-mot ℓ_2.

   On a alors que $\{r_1\ell_3, \ell_1 r_2\}$ est une *paire critique* de R.

Théorème 27. *Un SRM est localement confluent ssi toutes ses paires critiques sont joignables.*

Preuve. La confluence locale implique que les paires critiques sont joignables. Dans l'autre sens, on s'appuie sur l'analyse ci-dessus pour dire que l'on peut clore toute "question" pour le confluence locale dès lors que les paires critiques sont joignables. □

Il est possible de définir un algorithme qui calcule toutes les paires critiques d'un SRM fini — celles-ci correspondent aux cas 2 et 3 dans l'analyse qui précède le théorème. Il est à noter qu'il faut prendre en compte *tous* les appariements de règles de réécriture, y compris une règle avec elle-même.

Exemple : soit $\Sigma = \{a, b, c\}$ et soit R le SRM défini par les deux règles $\begin{matrix}(1) & aba \to abc \\ (2) & ab \to ba\end{matrix}$. On étudie les paires critiques de R en considérant les associations entre règles de réécriture :

- (1) avec (1) : le mot $ababa$ se réécrit en $abcba$ et $ababc$, $\{abcba, ababc\}$ est une paire critique.
- (1) avec (2) : on a une paire critique qui provient d'une situation d'"inclusion" : le mot aba se réécrit en abc et en baa, $\{abc, baa\}$ est une paire critique.

  On remarque aussi que l'on a une situation de "conflit de ressources" : le mot $abab$ se réécrit en $abcb$ et en $abba$, et $\{abcb, abba\}$ est une paire critique.

  On remarque également que $abab \to_R baab$. Cette réécriture ne donne pas lieu à la découverte d'une paire critique supplémentaire. En effet, avec la première réécriture ci-dessus, $\{abcb, baab\}$ est une situation qui est déjà couverte par la paire $\{abc, baa\}$. Avec la seconde réécriture, on a une situation d'indépendance : en réécrivant $abab$ en $baab$ et $abba$, on déclenche deux redex qui ne partagent rien.
- (2) avec (2) : pas de paire critique (car "pas moyen d'inclure ab dans ab ni de superposer ab avec ab", intuitivement).

Théorème 28 (Caractérisation de la terminaison dans les SRM)**.** *Soit $R \subseteq {\Sigma^*}^2$ un SRM sur un ensemble Σ. La relation $\to_R$ termine si et seulement s'il existe une relation binaire $>$ sur Σ^* telle que*

- $>$ *est une relation d'*ordre strict *: irréflexive et transitive*
- $>$ *termine*
- $\forall(\ell, r) \in R, \ell > r$
- $>$ *est compatible : $a > b$ implique $\forall x, y$, $xay > xby$.*

On peut remarquer qu'une relation d'ordre strict est asymétrique : $\forall x, y, (x > y) \Rightarrow \neg(y > x)$.

Procédure de complétion de Knuth-Bendix. On se donne R, un SRM sur Σ, et une relation $>$ sur Σ^* telle que :

- $>$ est un ordre strict
- $>$ termine
- $\forall(\ell, r) \in R$, $\ell > r$
- $\forall t, u$, si $t > u$, alors $\forall x, t$, $xty > xuy$
- $>$ est un ordre *total* : $\forall t, u$, soit $t = u$, soit $t > u$, soit $u > t$.

Par le théorème de caractérisation vu plus haut, on sait que $\to_R$ termine.

La procédure de Knuth-Bendix construit une suite $(R_i)_{i\geq 0}$ de SRM de la manière suivante :

Initialisation $R_0 = R$

Itération On calcule P_i, l'ensemble des paires critiques de R_i.

On initialise R_{i+1} comme étant egal à R_i.

Pour chaque paire $\{u_1, u_2\} \in P_i$,

- on calcule *une* forme normale $\widehat{u_1}$ (resp. $\widehat{u_2}$) de u_1 (resp. u_2) pour R_i. On est nécessairement dans l'un de ces trois cas :
  - si $\widehat{u_1} = \widehat{u_2}$, on ne fait rien
  - si $\widehat{u_1} > \widehat{u_2}$, on ajoute $(\widehat{u_1}, \widehat{u_2})$ à R_{i+1}
  - si $\widehat{u_2} > \widehat{u_1}$, on ajoute $(\widehat{u_2}, \widehat{u_1})$ à R_{i+1}

Proposition 29. *La procédure décrite ci-dessus a les propriétés suivantes :*

1. $\longleftrightarrow^*_{R_{i+1}} = \longleftrightarrow^*_{R_i}$
2. *si* $R_{i+1} = R_i$, *alors* $\forall j > i$, $R_{j+1} = R_j$ *et* R_i *est confluent*
3. *si* $\forall i$, $R_{i+1} \neq R_i$, *alors* $R_\infty = \cup_{i\geq 0} R_i$ *est un SRM infini, terminant et confluent, avec* $\longleftrightarrow^*_{R_\infty} = \longleftrightarrow^*_R$.

6.3 Établir la terminaison

Au-delà de la caractérisation de la terminaison donnée par le théorème 19, on décrit ici des outils que l'on peut utiliser et combiner pour prouver qu'une relation termine.

Les propriétés énoncées ci-dessous sont formulées pour le cas général des SRA, mais certains exemples utilisent des SRM.

Théorème 30 ("mesure")**.** *Soit* $(B, >)$ *un SRA terminant, et* $(A, \longrightarrow)$ *un SRA.*
S'il existe $\varphi : A \to B$ *telle que* $a \longrightarrow a'$ *implique* $\varphi(a) > \varphi(a')$, *alors* $\longrightarrow$ *termine.*

On peut par exemple prendre $B = (\mathbb{N}, >)$ pour montrer que la relation de réécriture de mots de l'exemple des cailloux (exemple 26) est terminante.

Définition 31. *Soient* $(A, \to_A)$ *et* $(B, \to_B)$ *deux SRA. Le* produit lexicographique *de* $(A, \to_A)$ *et* $(B, \to_B)$, *est le SRA noté* $(A \times B, \to_{A\times B})$ *et défini par*

$$(a, b) \to_{A\times B} (a', b') \quad ssi \quad \begin{array}{l} soit\ a \to_A a' \\ soit\ a = a'\ et\ b \to_B b'. \end{array}$$

Théorème 32. *On reprend les notations de la définition 31. Si* $\to_A$ *et* $\to_B$ *terminent, alors* $\to_{A\times B}$ *aussi.*

Lorsque l'on s'intéresse à la terminaison, les deux clauses qui définissent le produit lexicographique sont exclusives (si $a \to_A a$, $\to_A$ ne termine pas).

Exemple 33 (Retour à l'exemple 25)**.** *À cause de la première règle de réécriture, la taille des mots ne suffit pas à prouver que la relation* $\to_{R_0}$ *termine.*

Le SRA $(\mathbb{N}, >)$ *termine. Donc le produit lexicographique* $>_{\mathbb{N}\times\mathbb{N}}$ *(que l'on notera plutôt* $>^2$*) termine sur* $\mathbb{N} \times \mathbb{N}$. *On peut alors utiliser la "mesure" en posant* $\varphi : \Sigma^* \to \mathbb{N}^2$ *définie par* $\varphi(u) = (\text{taille}(u), \text{Nb}(u))$, *où* $\text{Nb}(u)$ *désigne le nombre de* b *dans le mot* u.

On doit vérifier que si $u \to_{R_0} v$, *alors* $\varphi(u) >^2 \varphi(v)$. *On remarque que* $\varphi(u_1u_2) = (\text{taille}(u_1) + \text{taille}(u_2), \text{Nb}(u_1) + \text{Nb}(u_2))$. *Ceci nous permet de nous ramener à vérifier* $\forall(\ell, r) \in R_0, \varphi(\ell) > \varphi(r)$, *ce qui se fait aisément.*

Définition 34 (Multiensemble)**.** *Soit A un ensemble. Un* multiensemble *sur A est la donnée d'une fonction $M : A \to \mathbb{N}$ (M désigne la multiplicité). On note ce multiensemble M.*

Un multiensemble M est fini *si $M(a) \neq 0$ pour un nombre fini de $a \in A$.*

Si M_1 et M_2 sont des multiensembles sur A, on définit

- $M_1 \cup M_2\,(a) = M_1(a) + M_2(a)$;
- $M_1 - M_2\,(a) = M_1(a) \ominus M_2(a)$, *où* $n \ominus (n+k) = 0$ *et* $(n+k) \ominus n = k$.

On note $M_1 \subseteq M_2$ si $\forall a \in A$, $M_1(a) \leq M_2(a)$. $|M|$ est la taille de M, définie par $|M| = \sum_{a \in A} M(a)$.

On écrit par exemple $M = \{3; 2; 1; 5; 7; 3; 3\}$, et note $\emptyset$ le multiensemble vide ($\forall a \in A$, $M(a) = 0$).

Définition 35. *Soit $(A, >)$ un SRA. On lui associe son* extension multiensemble*, notée $>_{\text{mul}}$, qui est une relation sur les multiensembles* finis *sur A, en posant $M >_{\text{mul}} N$ ssi*

$$\exists X, Y \text{ multiensembles sur } A \text{ t.q. } \begin{cases} \emptyset \neq X \subseteq M \\ N = (M - X) \cup Y \\ \forall y \in Y, \exists x \in X \text{ t.q. } x > y \end{cases}$$

Dans la définition ci-dessus, X et Y sont les "témoins" du fait que $M >_{\text{mul}} N$ (et sont finis).

La taille des multiensembles n'est pas décisive pour $>_{\text{mul}}$: on a $\{3; 2; 3\} >_{\text{mul}} \{2\}$, et $\{3\} >_{\text{mul}} \{2; 2\}$.

$Y = \emptyset$ signifie que l'on retire des éléments pour "justifier" une relation $>_{\text{mul}}$; si $Y \neq \emptyset$, on fait un "marché", avec l'idée qu'un lourd pèse davantage que plusieurs légers.

$>_{\text{mul}}$ contient $>$ au sens où $a > b$ implique $\{a\} >_{\text{mul}} \{b\}$. Entre ensembles finis ($\forall a \in A$, $M(a) \in \{0, 1\}$), $>_{\text{mul}}$ coïncide avec $\supset$.

Théorème 36 (Terminaison de l'extension multiensemble)**.** *$>$ termine si et seulement si $>_{\text{mul}}$ termine.*

Preuve. La terminaison de $>_{\text{mul}}$ implique immédiatement la terminaison de $>$ par la remarque ci-dessus.

Dans l'autre sens, on procède par contraposée : on se donne une chaîne infinie de multiensembles finis $(M_i)_{i \geq 0}$ tels que $\forall i$, $M_i >_{\text{mul}} M_{i+1}$. En revenant à la définition de $>_{\text{mul}}$, on construit à partir de cette chaîne un arbre, dans lequel la relation "est père de" correspond à une relation $a > b$. On obtient ainsi un arbre comportant une infinite de nœuds qui est à branchement fini : le Lemme de König permet de déduire l'existence d'une branche infinie, qui est une divergence pour $>$.

Un peu plus précisément, chaque relation $M_i >_{\text{mul}} M_{i+1}$ correspond à deux multiensembles X_i et Y_i. Les propriétés de l'arbre découlent de propriétés des Y_i : chaque Y_i est fini, sinon M_{i+1} ne serait pas fini. L'arbre est à branchement fini car on ajoute des nœuds y à un nœud x uniquement à l'étape où l'on a retiré x ($x \in X_i$). L'arbre est infini car il y a une infinité de $Y_i \neq \emptyset$, sinon on contredirait l'existence de la chaîne infinie pour $>_{\text{mul}}$. □

6.4 Termes

6.4.1 Termes, occurrences, substitutions

Définition 37. *On se donne*

- *un ensemble infini $\mathtt{V}$ de* variables *("variables de termes"), notées $X, Y, Z, x, y, z, \ldots$;*
- *un ensemble fini Σ de* constantes*, chacune avec son arité $\in \mathbb{N}$.*

On définit de manière inductive l'ensemble $\mathcal{T}(\Sigma, \mathtt{V})$ des termes *sur $(\Sigma, \mathtt{V})$, par la grammaire suivante :*

$$t \quad ::= \quad X \mid f^k(t_1, \ldots, t_k).$$

Dans les preuves par induction sur la structure d'un terme, on traite généralement deux cas : le cas de la variable, et les cas des constantes, qui sont regroupés.

Définition 38. *On définit par induction sur la structure d'un terme $t \in \mathcal{T}(\Sigma, \mathtt{V})$:*

- *l'ensemble* $\mathrm{vars}(t)$ *des variables apparaissant dans* t
- *l'ensemble* $pos(t)$ *des* positions *d'un terme* t*, qui sont des mots d'entiers naturels ("indications routières")*
- *l'ensemble des* sous-termes *de* t*. Si* $p \in pos(t)$*, on note* $t_{|p}$ *le sous-terme de* t *situé en* p*.*
  L'ensemble des sous-termes de t *est* $\{t_{|p}, p \in pos(t)\}$*.*
- *on note* $t[u]_p$ *le terme obtenu à partir de* t *en remplaçant* $t_{|p}$ *par* u*, pour* $p \in pos(t)$*.*
- *la taille de* t*, notée* $|t|$*, est définie de manière inductive ainsi :* $|X| = 1$*,* $|f(t_1, \ldots, t_k)| = 1 + \Sigma_{1 \geq i \geq k} |t_i|$*.*

Définition 39 (substitution)**.** *Étant donné* $\mathcal{T}(\Sigma, \mathtt{V})$*, une* substitution σ *est un ensemble* fini *de couples* $(X, t) \in \mathtt{V} \times \mathcal{T}(\Sigma, \mathtt{V})$ *tel que* $\forall X \in \mathtt{V}$*,* X *apparaît au plus une fois comme membre gauche d'un couple de* σ*.*

Le domaine *de* σ*, noté* $\mathrm{dom}(\sigma)$*, est* $\{X \in \mathtt{V}, \exists t, (X, t) \in \sigma\}$*. Lorsque* $X \in \mathrm{dom}(\sigma)$*, on dit que* σ *est définie en* X*, et* $\sigma(X)$ *désigne le terme* $t \in \mathcal{T}(\Sigma, \mathtt{V})$ *tel que* $(X, t) \in \sigma$*.*

Si σ *est une substitution et* $t \in \mathcal{T}(\Sigma, \mathtt{V})$*, on note* $\sigma(t)$ *le terme obtenu en remplaçant* simultanément *toutes les occurrences de* $X \in \mathrm{dom}(\sigma)$ *par* $\sigma(X)$ *dans* t*.*

Si σ_1, σ_2 *sont deux substitutions, on définit leur* composition*, notée* $\sigma_1 \circ \sigma_2$*, par* $(\sigma_1 \circ \sigma_2)(t) = \sigma_1(\sigma_2(t))$*.*

En réalité, $\sigma(t)$ est défini inductivement par

$$\begin{array}{ll} \sigma(X) = X & \text{si } X \notin \mathrm{dom}(\sigma) \\ \sigma(X) = \sigma(X) & \text{si } X \in \mathrm{dom}(\sigma) \end{array} \qquad \sigma(f(t_1, \ldots, t_k)) = f(\sigma(t_1), \ldots, \sigma(t_k))$$

Lemme 40 ($\sigma(X) = \sigma(t)$)**.** *Si* $\sigma(X) = \sigma(t)$*, alors* $\forall u, \sigma(u[t/X]) = \sigma(u)$ *(autrement dit,* $\sigma \circ [t/X] = \sigma$*).*

Preuve. Par induction sur la structure de u, on considère deux cas (variable et constante). □

6.4.2 Unification (du premier ordre)

Définition 41. *Un* problème d'unification *est la donnée d'un ensemble* fini *de paires de termes dans* $\mathcal{T}(\Sigma, \mathtt{V})$ *noté* $\mathcal{P} = \{\{t_1, u_1\}, \ldots, \{t_k, u_k\}\}$*.*

Un unificateur *(ou une* solution*) d'un tel* $\mathcal{P}$ *est une substitution* σ *telle que* $\forall \{t, u\} \in \mathcal{P}, \sigma(t) = \sigma(u)$*.*

L'ensemble des unificateurs de $\mathcal{P}$ *est noté* $U(\mathcal{P})$*. Un* unificateur le plus général de $\mathcal{P}$ *est un* $\sigma \in U(\mathcal{P})$ *tel que* $\forall \sigma' \in U(\mathcal{P}), \exists \sigma''$ *t.q.* $\sigma' = \sigma'' \circ \sigma$*.*

On écrira parfois $t = u$ ou $t \stackrel{?}{=} u$ pour faire référence à une composante $\{t, u\}$ d'un problème d'unification.

Lemme 42 ($U(\sigma)$)**.** $\forall \sigma' \in U(\sigma), \sigma' = \sigma' \circ \sigma$*, autrement dit* σ *est un mgu (unificateur le plus général) de* σ*.*

Définition 43 (Algorithme d'unification)**.** *L'algorithme est décrit comme un SRA sur un ensemble d'états, où un état est :*

- *soit un couple* $(\mathcal{P}, \sigma)$ $\mathcal{P}$ *problème d'unification,* σ *substitution ;*
- *soit* l'état d'échec*, noté* $\bot$*.*

Un état de la forme $(\emptyset, \sigma)$ *est appelé* état de succès*.*

On définit une relation entre états, notée $\longrightarrow$ *:*

- $\bot \not\longrightarrow$ *et* $(\emptyset, \sigma) \not\longrightarrow$
- *un état de la forme* $(\mathcal{P}, \sigma)$ *qui n'est pas un état de succès peut s'écrire* $(\{\{t, t'\} \uplus \mathcal{P}\}, \sigma)$*. On distingue deux cas, suivant que* $\{t, t'\} \cap \mathtt{V} = \emptyset$ *ou pas :*

– *si* $\{t, t'\} \cap \mathtt{V} = \emptyset$*, on distingue deux sous-cas :*

(1) $(\{f(t_1, \ldots, t_k), f(u_1, \ldots, u_k)\} \uplus \mathcal{P}, \sigma) \longrightarrow (\{t_1, u_1\} \cup \cdots \cup \{t_k, u_k\} \cup \mathcal{P}, \sigma)$

(2) $(\{f(t_1, \ldots, t_k), g(u_1, \ldots, u_n)\} \uplus \mathcal{P}, \sigma) \longrightarrow \bot$ *lorsque* $f \neq g$

– *si* $t' = X \in \mathtt{V}$ *(arbitrairement, on choisit que* t' *est une variable), on distingue trois sous-cas :*

(3) $(\{X, X\} \uplus \mathcal{P}, \sigma) \longrightarrow (\mathcal{P}, \sigma)$

(4) $(\{X, t\} \uplus \mathcal{P}, \sigma) \longrightarrow \bot$ *si* $X \in \mathrm{vars}(t)$ *et* $t \neq X$

(5) $(\{X, t\} \uplus \mathcal{P}, \sigma) \longrightarrow (\mathcal{P}[t/X], [t/X] \circ \sigma)$ *si* $X \notin \mathrm{vars}(t)$*,*
où $\mathcal{P}[t/X] = \{\{u_1[t/X], u_2[t/X]\}, \{u_1, u_2\} \in \mathcal{P}\}$*.*

Lemme 44. *La relation* $\longrightarrow$ *termine.*

Preuve. La cinquième clause concentre les difficultés : on remarque que dans une telle transition, le nombre de variables mentionnées dans la composante $\mathcal{P}$ d'un état décroît de 1.

On définit $\mathrm{vars}(\mathcal{P}) = \cup_{\{t,u\} \in \mathcal{P}} \mathrm{vars}(t) \cup \mathrm{vars}(u)$, et $|\mathcal{P}| = \Sigma_{\{t,u\} \in \mathcal{P}} |t| + |u|$. On raisonne avec le produit lexicographique défini à partir de $|\mathrm{vars}(\mathcal{P})|$ et $|\mathcal{P}|$ (dans cet ordre). En particulier, la première clause ne change pas $\mathrm{vars}(\mathcal{P})$et fait décroître strictement $|\mathcal{P}|$, et la cinquième peut faire croître $|\mathcal{P}|$, mais fait décroître strictement $|\mathrm{vars}(\mathcal{P})|$. □

Au vu de comment est définie $\longrightarrow$, la propriété ci-dessus implique que soit l'algorithme échoue, soit il toujours une substitution (les états d'échec et de succès sont en effet les seuls états qui n'ont pas de transition).

Lemme 45 (cas d'échec)**.** *Si* $(\mathcal{P}, \sigma) \longrightarrow \bot$*, alors* $U(\mathcal{P} \cup \sigma) = \emptyset$*.*

Preuve. Deux règles sont appliquables : la règle (2) est immédiate à traiter. S'agissant de la règle (4), on raisonne sur la taille de $\sigma(X)$ et $\sigma(t)$, en déduisant une contradiction à partir de $|\sigma(X)| = |\sigma(t)|$. □

L'algorithme préserve l'invariant suivant :

Définition 46 (forme résolue)**.** $(\mathcal{P}, \sigma)$ *est en* forme résolue *si* $\forall X \in \mathrm{dom}(\sigma)$*,* X *n'apparaît qu'une fois dans* σ *(dans un couple* (X, t)*), et pas du tout dans* $\mathcal{P}$*.*

Proposition 47. *Supposons* $(\mathcal{P}_0, \sigma_0)$ *en forme résolue et* $(\mathcal{P}_0, \sigma_0) \longrightarrow (\mathcal{P}_1, \sigma_1)$*.*
Alors $(\mathcal{P}_1, \sigma_1)$ *est en forme résolue, et* $U(\mathcal{P}_0 \cup \sigma_0) = U(\mathcal{P}_1 \cup \sigma_1)$*.*

Ci-dessus, comme plus haut, on voit une substitution comme un problème d'unification, lorsque l'on écrit $U(\mathcal{P}_i \cup \sigma_i)$.

Pour prouver le théorème, on utilise le lemme technique suivant :

Lemme 48. *Si* $X \notin \mathrm{dom}(\sigma)$*, alors* $[t/X] \circ \sigma = \{(X, t)\} \cup \{(Y, \sigma(Y)[t/X]), Y \in \mathrm{dom}(\sigma)\}$*.*

Preuve.[de la proposition 47] On établit les deux propriétés à la fois, en raisonnant par cas suivant la règle de l'algorithme qui est appliquée.

Les **cas 2 et 4** sont exclus.

Cas 3 : alors $(\mathcal{P}_1, \sigma_1)$ est en forme résolue, et on remarque que toute substitution unifie X et X.

Cas 1 : $(\mathcal{P}_1, \sigma_1)$ est en forme résolue car $\mathrm{vars}(\mathcal{P}_1) = \mathrm{vars}(\mathcal{P}_0)$. Il est facile de montrer que les solutions sont préservées.

Cas 5 (le cas intéressant) : on a

$$X \notin \mathrm{vars}(t) \qquad \mathcal{P}_0 = \{X, t\} \uplus \mathcal{P} \qquad \mathcal{P}_1 = \mathcal{P}[t/x] \qquad \sigma_1 = [t/x] \circ \sigma_0$$

Forme résolue : comme $(\mathcal{P}_0, \sigma_0)$ est en forme résolue et $X \in \mathrm{vars}(\mathcal{P}_0)$, on a nécessairement $X \notin dom(\sigma_0)$. Ceci nous permet d'appliquer le lemme 48 pour écrire

$$\sigma_1 \quad = \quad \{(X, t)\} \cup \{(Y, \sigma_0(Y)[t/X]), Y \in dom(\sigma_0)\} \tag{2}$$

Dès lors, comme $X \notin \text{vars}(t)$, $X \notin \text{vars}(\mathcal{P}_1)$, et X n'apparaît qu'une fois dans σ_1: $(\mathcal{P}_1, \sigma_1)$ est en forme résolue.

Unificateurs : on montre les deux inclusions.

- $U(\mathcal{P}_0 \cup \sigma_0) \subseteq U(\mathcal{P}_1 \cup \sigma_1)$. Soit $\sigma \in U(\mathcal{P}_0 \cup \sigma_0)$.
  Alors $\sigma(X) = \sigma(t)$, car $\{X, t\} \in \mathcal{P}_0$. Par le lemme 40, on a

  $$\forall u, \sigma(u) = \sigma(u[t/X]) \tag{3}$$

  On considère alors $\{u_1, u_2\} \in \mathcal{P}_1 \cup \sigma_1 = \mathcal{P}_1 \cup \{(X, t)\} \cup \{(Y, \sigma_0(Y)[t/X]), Y \in dom(\sigma_0)\}$ (par l'équation (2) ci-dessus). Cette écriture suggère de distinguer 3 cas :

  1. $\{u_1, u_2\} \in \mathcal{P}_1$, et donc $u_i = v_i[t/X], i = 1, 2$, pour $\{v_1, v_2\} \in \mathcal{P}$. Par hypothèse, $\sigma(v_1) = \sigma(v_2)$. On conclut $\sigma(u_1) = \sigma(u_2)$ en appliquant (3).
  2. $u_1 = X, u_2 = t$ (ou l'inverse). On a déjà remarqué ci-dessus $\sigma(X) = \sigma(t)$.
  3. $u_1 = Y \in dom(\sigma_0)$, $u_2 = \sigma_0(Y)[t/X]$.
     Comme $\sigma \in U(\mathcal{P}_0 \cup \sigma_0)$, $\sigma \in U(\sigma_0)$, et donc $\sigma(Y) = \sigma(\sigma_0(Y))$. Or $\sigma(\sigma_0(Y)[t/X] = \sigma(\sigma_0(Y))$ par (3). On peut donc conclure.

- $U(\mathcal{P}_0 \cup \sigma_0) \supseteq U(\mathcal{P}_1 \cup \sigma_1)$. Soit $\sigma \in U(\mathcal{P}_1 \cup \sigma_1)$. Comme ci-dessus, étant donné que $(X, t) \in \sigma_1$, on a

  $$\forall u, \sigma(u) = \sigma(u[t/X]) \tag{4}$$

  On considère alors $\{u_1, u_2\} \in \mathcal{P}_0 \cup \sigma_0$, en distinguant deux cas :

  1. $\{u_1, u_2\} \in \mathcal{P}_0$. Alors $\{u_1[t/X], u_2[t/X]\} \in \mathcal{P}_1$, et on conclut par (4).
  2. $\{u_1, u_2\} \in \sigma_0$, autrement dit $u_1 = Y$, $u_2 = \sigma_0(Y)$ (ou l'inverse).
     On remarque $Y \neq X$ car $X \in \text{vars}(\mathcal{P}_0)$ et $(\mathcal{P}_0, \sigma_0)$ est en forme résolue.
     On sait alors $(Y, \sigma_0(Y)[t/X]) \in \sigma_1$, donc σ unifie ces deux termes.
     Par (4), $\sigma(\sigma_0(Y)[t/X]) = \sigma(\sigma_0(Y))$, donc $\sigma(Y) = \sigma(\sigma_0(Y))$, et on conclut.

□

Théorème 49. *L'algorithme d'unification vérifie les propriétés suivantes :*

- *si* $(\mathcal{P}, \emptyset) \longrightarrow^* (\emptyset, \sigma)$, *alors* σ *est un unificateur le plus général de* $\mathcal{P}$.
- *si* $(\mathcal{P}, \emptyset) \longrightarrow^* \bot$, *alors* $U(\mathcal{P}) = \emptyset$.

La première propriété exprime la *correction* de l'algorithme : il calcule une solution la plus générale du problème initial. La seconde propriété exprime la *complétude* : il ne "rate" pas de solution.
Preuve. On raisonne dans les deux cas par induction sur le nombre d'étapes pour $\longrightarrow$. La proposition 47, ainsi que les lemmes 42 (pour la première propriété) et 45 (pour la seconde) permettent de conclure. □

6.4.3 Réécriture de termes

Définition 50 (Système de réécriture de termes)**.** *On se donne* $\mathcal{T}(\Sigma, \mathtt{V})$. *Une* règle de réécriture *est un couple* (ℓ, r) *de termes, parfois noté* $\ell \to r$, *tel que*

1. $\ell \notin \mathtt{V}$ *(*ℓ *n'est pas une variable), et*
2. $Vars(r) \subseteq Vars(\ell)$ *(pas d'introduction de "nouvelle variable").*

Un système de réécriture de termes *(SRT) est la donnée d'un ensemble de règles de réécriture* $R = \{(\ell_i, r_i)\}$. *Un tel* R *détermine une relation binaire sur* $\mathcal{T}(\Sigma, \mathtt{V})$, *notée* $\to_R$, *définie comme suit :*

$$t \to_R t' \text{ ssi } \exists(\ell, r) \in R, \exists p \in pos(t), \exists \sigma \text{ substitution t.q. } t_{|p} = \sigma(\ell) \text{ et } t' = t[\sigma(r)]_p.$$

On dit dans la situation ci-dessus que $\sigma(\ell)$ *est un* redex *(reducible expression).*

Étant donnée une substitution $\sigma = \{(X_i, t_i)\}$, *on considérera parfois le problème d'unification correspondant* $\{\{X_i, t_i\}\}$, *en notant également* σ *le problème d'unification.*

La définition ci-dessus a pour conséquence que l'on peut renommer les variables dans les règles, du moment que "l'on ne fait pas n'importe quoi" : la règle $f(X, Y) \to f(g(Y), X)$ est équivalente à la règle $f(Z, T) \to f(g(T), Z)$, car elle permet exactement les mêmes réécritures. Cette observation est utilisée dans la définition 52.

Exemple 51 (Réécriture de termes)**.** *On considère* $\Sigma = \{f^2, g^1, a^0, b^0\}$, *et le système de réécriture* $R = \{f(X, Y) \to f(g(X), f(Y, X))\}$. *On pose* $\ell = f(X, Y)$, *le membre gauche de l'unique règle de réécriture de* T. *On a les deux réécritures suivantes pour le terme* $f(Z, f(a, b))$ *:*

- $\underbrace{f(Z, f(a, b))}_{\sigma_1(\ell)} \to_R f(g(Z), f(f(a, b), Z))$, *avec* $\sigma_1 = [Z/X,\ f(a, b)/Y]$, *et*
- $f(Z, \underbrace{f(a, b)}_{\sigma_2(\ell)}) \to_R f(Z, f(g(a), f(b, a)))$, *avec* $\sigma_2 = [a/X,\ b/Y]$.

La notion de paire critique vue pour les SRM (partie 6.2) peut être adaptée aux termes. Deux redex peuvent uniquement être contenus l'un dans l'autre, la notion de paire critique correspondant à la situation où ceux-ci "partagent des constantes" (condition $\ell_{1|p} \notin \mathtt{V}$ ci-dessous).

Définition 52 (Paires critiques dans un SRT)**.** *Soit* R *un SRT sur* $(\Sigma, \mathtt{V})$. *On se donne deux règles de réécriture* (ℓ_1, r_1) *et* (ℓ_2, r_2) *de* R *en supposant qu'elles n'ont pas de variable en commun (quitte à renommer les variables).*

S'il existe $p \in pos(\ell_1)$ *telle que* $\ell_{1|p} \notin \mathtt{V}$, *et* σ *un mgu du problème d'unification* $\ell_{1|p} \stackrel{?}{=} \ell_2$, *alors* $\{\sigma(r_1), \sigma(\ell_1)[\sigma(r_2)]_{|p}\}$ *est une paire critique de* R.

Dans la définition ci-dessus, le terme $\sigma(\ell_1)$ peut être réécritu de deux façons : d'une part, en $\sigma(r_1)$, par définition de la réécriture, et d'autre part, en $\sigma(\ell_1)[\sigma(r_2)]_{|p}$, car $\sigma(\ell_1)$ contient $\sigma(\ell_2)$ comme sous-terme à la position p. C'est le rôle de σ que de "faire apparaître" une instance de ℓ_2 comme sous-terme d'une instance de ℓ_1.

7 Typage

7.1 Types simples pour Fun

7.1.1 Définition du système de types

On définit les *types*, notés $\tau, \tau', \tau_1, \ldots$, par la grammaire suivante :

$$\tau \quad ::= \quad \mathtt{int} \mid \tau_1 \to \tau_2.$$

On rappelle la règle pour l'associativité de la flèche : le type $\tau_1 \to \tau_2 \to \tau_3$ se lit comme $\tau_1 \to (\tau_2 \to \tau_3)$. Autrement dit, $\tau_1 \to \tau_2 \to \tau_3$ est un type de fonctions qui renvoient des fonctions, alors que $(\tau_1 \to \tau_2) \to \tau_3$ est un type de fonctions qui prennent en argument des fonctions.

Définition 53. *Une* hypothèse de typage *est un couple (x, τ), où x est une variable* Fun *et τ est un type. On la note parfois $x : \tau$.*

Un environnement de typage *(noté $\Gamma, \Gamma', \ldots$) est un ensemble fini Γ d'hypothèses de typage tel que pour toute variable x, x apparaît au plus une fois dans Γ.*

On note $\Gamma(x) = \tau$ si $(x, \tau) \in \Gamma$, et $\mathrm{dom}(\Gamma) = \{x, \exists \tau \text{ t.q. } (x, \tau) \in \Gamma\}$.

*L'*extension *de Γ avec (x, τ), notée $\Gamma, x : \tau$, n'est définie que si $x \notin \mathrm{dom}(\Gamma)$ et est alors définie comme $\Gamma \cup \{(x, \tau)\}$.*

On définit le jugement de typage noté $\Gamma \vdash e : \tau$ ("sous les hypothèses Γ, l'expression e a le type τ") par les règles d'inférence suivantes :

$$\mathrm{T_k}\ \frac{}{\Gamma \vdash k : \texttt{int}} \qquad \mathrm{T_p}\ \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

$$\mathrm{T_v}\ \frac{}{\Gamma \vdash x : \tau}\ \Gamma(x) = \tau \qquad \mathrm{T_a}\ \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'} \qquad \mathrm{T_f}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e : \tau_1 \to \tau_2}$$

On peut remarquer qu'il y a une règle de typage par construction du langage Fun.

Dans la règle $\mathrm{T_f}$ ci-dessus, on peut remarquer qu'il se peut que l'on ait à renommer la variable liée x dans $\texttt{fun}\ x\ \texttt{->}\ e$, de manière à pouvoir former $\Gamma, x : \tau_1$ dans la prémisse de la règle (autrement dit, si $x \in \mathrm{dom}(\Gamma)$, alors on type $\texttt{fun}\ x'\ \texttt{->}\ e[x'/x]$, où x' est une variable qui n'apparaît ni dans e ni dans $\mathrm{dom}(\Gamma)$, de manière à pouvoir dériver $\Gamma, x' : \tau_1 \vdash e[x'/x] : \tau_2$).

Exemple 54. *Voici une dérivation de typage :*

$$\dfrac{\dfrac{}{g : \texttt{int} \to \texttt{int} \vdash g : \texttt{int} \to \texttt{int}} \qquad \dfrac{g : \texttt{int} \to \texttt{int} \vdash g : \texttt{int} \to \texttt{int} \qquad g : \texttt{int} \to \texttt{int} \vdash 7 : \texttt{int}}{g : \texttt{int} \to \texttt{int} \vdash g\ 7 : \texttt{int}}}{g : \texttt{int} \to \texttt{int} \vdash g\ (g\ 7) : \texttt{int}}}{\emptyset \vdash \texttt{fun}\ g\ \texttt{->}\ g\ (g\ 7)) : (\texttt{int} \to \texttt{int}) \to \texttt{int}}$$

7.1.2 Propriétés du système de types

On met ici en relation les propriétés statiques d'une expression (relation $\vdash$) et la dynamique (relation $\longrightarrow$). Cela permet intuitivement de valider la définition du système de types.

Lemme 55 (Lemmes administratifs)**.** *1. si $\Gamma \vdash e : \tau$, alors $v\ell(e) \subseteq \mathrm{dom}(\Gamma)$.*

2. échange : si $\Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e : \tau$, alors $\Gamma, x_2 : \tau_2, x_1 : \tau_1 \vdash e : \tau$.

3. affaiblissement : si $\Gamma \vdash e : \tau$, alors $\forall x \notin \mathrm{dom}(\Gamma)$, $\forall \tau_0$, $\Gamma, x : \tau_0 \vdash e : \tau$.

4. renforcement : si $\Gamma, x : \tau_0 \vdash e : \tau$ et $x \notin v\ell(e)$, alors $\Gamma \vdash e : \tau$.

La propriété 2, d'échange, ne dit rien si l'on considère que Γ a une structure d'ensemble. Elle a prend son sens si l'on traite Γ comme une liste (ordonnée, et ne contenant pas deux couples (x, τ) et (x, τ'), pour toute variable x).

Les propriétés ci-dessus se prouvent en raisonnant par induction sur la relation de typage.

Progrès.

Lemme 56 (Typage et formes normales)**.**

- *si $\emptyset \vdash e : \texttt{int}$ et $e \not\longrightarrow$, alors $e = k \in \mathbb{Z}$;*

- *si* $\emptyset \vdash e : \tau_1 \to \tau_2$ *et* $e \not\longrightarrow$*, alors il existe* x *et* e' *t.q.* $e = \texttt{fun}\ x \to e'$.

Preuve. On démontre la propriété suivante

$$\forall e, \tau, \text{ si } \emptyset \vdash e : \tau, \text{ alors } e \not\longrightarrow \text{ implique } \begin{cases} \text{si } \tau = \tau_1 \to \tau_2, \exists e', x \text{ t.q. } e = \texttt{fun}\ x \to e' \\ \text{si } \tau = \texttt{int}, \exists k \in \mathbb{Z} \text{ t.q. } e = k \end{cases}$$

par induction sur la relation de typage. Il y a 5 cas.

- règle $\mathrm{T_k}$: ok, c'est ce qu'on veut démontrer.
- règle $\mathrm{T_p}$: on obtient par induction $e_1 = k_1$ et $e_2 = k_2$, donc $e = k_1 + k_2$, ce qui contredit l'hypothèse $e \not\longrightarrow$.
- règle $\mathrm{T_f}$: ok, c'est ce qu'on veut démontrer.
- règle $\mathrm{T_v}$: $\emptyset \vdash x : \tau$ est absurde par le Lemme 55:(1).
- règle $\mathrm{T_a}$: c'est le cas intéressant. On a $e = e_1\ e_2$.

  Par induction, on a HI1 : si $e_1 \not\longrightarrow$, alors $e_1 = \texttt{fun}\ x \to e_1'$ et HI2 : si $e_2 \not\longrightarrow$, alors $e_2 = v$ pour v une valeur (en effet, suivant la forme du type de e_2, soit $e_2 = k$, soit $e_2 = \texttt{fun}\ y \to e_2'$).

  Supposons maintenant $e \not\longrightarrow$. On a nécessairement $e_1 \not\longrightarrow$, sinon e pourrait se réduire. Donc $e_1 = \texttt{fun}\ x \to e_1'$ par HI1.

  Dès lors, on a aussi $e_2 \not\longrightarrow$, car sinon e pourrait se réduire, et par HI2, $e_2 = v$. On a finalement $e = (\texttt{fun}\ x \to e_1')\ v$, ce qui contredit l'hypothèse $e \not\longrightarrow$. Ainsi, e ne peut pas être une application.

□

Le lemme ci-dessus exprime que la relation de typage écarte les formes normales "absurdes", au sens où elles ne peuvent se réduire pour de mauvaises raisons (application d'un entier, somme avec une fonction).

Il entraîne directement la propriété suivante :

Proposition 57 (Progrès)**.** *Si* $\emptyset \vdash e : \tau$*, alors soit* e *est une valeur, soit* $\exists e'$ *t.q.* $e \longrightarrow e'$.

Reste maintenant à voir, dans le deuxième cas ci-dessus, ce que l'on peut dire de e' : c'est l'objet de la propriété de préservation.

Préservation.

Lemme 58 (Typage et substitution)**.** *Si* $\Gamma, x : \tau_0 \vdash e : \tau$ *et* $\emptyset \vdash v : \tau_0$*, alors* $\Gamma \vdash e[v/x] : \tau$.

Preuve. Par induction sur la structure de e (5 cas). □

Proposition 59 (Préservation / réduction assujettie, subject reduction)**.** *Si* $\emptyset \vdash e : \tau$ *et* $e \longrightarrow e'$*, alors* $\emptyset \vdash e' : \tau$.

Preuve. Par induction sur la dérivation de $\emptyset \vdash e : \tau$, il y a 5 cas. □

Les deux propriétés ci-dessus, de progrès et de préservation, ont pour conséquence ce qu'on appelle la *correction* du système de types : on a démontré en particulier qu'il n'y pas d'erreur à l'exécution (comme l'addition d'une fonction, ou l'application d'un entier) pour une expression bien typée.

Typage des déclarations. La règle de typage des déclarations est la suivante :

$$\frac{\Gamma \vdash e_1 : \tau' \qquad \Gamma, x : \tau' \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}\ \mathrm{T_{let}}$$

Les propriétés vues ci-dessus sont également valables pour l'extension de FUN avec les déclarations.

7.2 Inférence de types

Des types étendus. On considère les termes construits à partir de la signature $\Sigma = \{\texttt{int}^0, \to^2\}$ et d'un ensemble $\mathtt{V} = \{X, Y, \dots\}$ de variables de type. Cela revient à considérer la grammaire suivante pour les types étendus :

$$\tau \quad ::= \quad \texttt{int} \;\big|\; \tau_1 \to \tau_2 \;\big|\; X.$$

Les $X, X', X_1, Y, \dots$ sont les variables de types, on les appellera aussi *variables d'unification*.

On peut remarquer que l'on conserve la notation τ pour les types étendus. De même, on notera Γ les *environnements de types étendus*, qui sont définis comme les environnements de types, avec des types étendus à la place des types.

Un type étendu τ tel que $\mathrm{vars}(\tau) = \emptyset$ est dit *type constant* : les types que l'on a manipulés à la partie 7.1.1 sont les types constants.

Définition 60. *Soit e une expression* Fun, *Γ un environnement de typage étendu tel que $v\ell(e) \subseteq \mathrm{dom}(\Gamma)$, et $X \in \mathtt{V}$ une variable de type.*

On définit un problème d'unification sur $\mathcal{T}(\Sigma, \mathtt{V})$, noté $\mathcal{P}(\Gamma, e, X)$, par induction sur e, de la manière suivante (5 cas) :

$$\begin{array}{rcll}
\mathcal{P}(\Gamma, k, X) & = & \{X \stackrel{?}{=} \texttt{int}\} & \\
\mathcal{P}(\Gamma, e_1 + e_2, X) & = & \mathcal{P}(\Gamma, e_1, X_1) \cup \mathcal{P}(\Gamma, e_2, X_2) \cup \{X_1 \stackrel{?}{=} \texttt{int}, X_2 \stackrel{?}{=} \texttt{int}, X \stackrel{?}{=} \texttt{int}\} & X_1 \text{ et } X_2 \text{ fraîches} \\
\mathcal{P}(\Gamma, x, X) & = & \{X \stackrel{?}{=} \Gamma(x)\} & \\
\mathcal{P}(\Gamma, \texttt{fun } x \to e, X) & = & \mathcal{P}(\Gamma, x : X_1,\, e,\, X_2) \cup \{X \stackrel{?}{=} X_1 \to X_2\} & X_1 \text{ et } X_2 \text{ fraîches} \\
\mathcal{P}(\Gamma, e_1\ e_2, X) & = & \mathcal{P}(\Gamma, e_1, X_1) \cup \mathcal{P}(\Gamma, e_2, X_2) \cup \{X_2 \stackrel{?}{=} X_1 \to X\} & X_1 \text{ et } X_2 \text{ fraîches}
\end{array}$$

Ci-dessus, "X_1 et X_2 fraîches" signifie que l'on engendre de nouvelles variables d'unification, qui n'apparaissent pas dans Γ, sont distinctes de X, et sont "différentes de toutes les variables d'unification engendrées jusque là". Par exemple, dans les appels récursifs sur e_1 et e_2 dans la dernière équation, on peut imaginer que l'on commence par calculer le problème d'unification pour e_1, ce qui a pour effet d'engendrer des variables d'unification, puis on passe au problème d'unification pour e_2, et lorsqu'on calcule ce dernier, on "évite" toutes les variables d'unifications engendrées jusque là. Une intuition possible serait d'avoir un compteur global, que l'on incrémente à chaque fois qu'on a besoin d'une variable fraîche. Cette notion de fraîcheur n'est pas immédiate à formaliser de manière rigoureuse.

Définition 61 (Substitution constante)**.** *On dit que σ, substitution sur $\mathcal{T}(\Sigma, \mathtt{V})$, est* constante *si $\forall X \in \mathrm{dom}(\sigma)$, $\mathrm{vars}(\sigma(X)) = \emptyset$.*

Une substitution constante est une substitution qui n'a que des types constants dans son codomaine.

Théorème 62. *Soit e une expression* Fun, *et Γ un environnement de typage* non étendu *tel que $v\ell(e) \subseteq \mathrm{dom}(\Gamma)$.*

On choisit $X \in \mathtt{V}$. On a les propriétés suivantes :

(correction) $\forall \sigma \in U(\mathcal{P}(\Gamma, e, X))$, si σ est constante, alors $\Gamma \vdash e : \sigma(X)$.

(complétude) Pour tout τ tel que $\Gamma \vdash e : \tau$, il existe $\sigma \in U(\mathcal{P}(\Gamma, e, X))$ qui vérifie $\sigma(X) = \tau$.

Preuve. On prouve chacune des propriétés par induction sur e, il y a 5 cas à chaque fois. On ne traite ici que le cas de l'application.

1. $e = e_1\ e_2$, et on a par définition

$$\mathcal{P}(\Gamma, e_1\ e_2, X) \;=\; \mathcal{P}(\Gamma, e_1, X_1) \cup \mathcal{P}(\Gamma, e_2, X_2) \cup \{X_2 \stackrel{?}{=} X_1 \to X\} \tag{5}$$

   Par conséquent, étant donnée une solution constante σ de $\mathcal{P}(\Gamma, e, X)$, on a que σ est aussi solution constante des $\mathcal{P}(\Gamma, e_i, X_i)$, et aussi que $\sigma(X_1) = \sigma(X_2) \to \sigma(X)$.

   Par induction, on a $\Gamma \vdash e_i : \sigma(X_i)$, donc on peut appliquer la règle $\mathrm{T_a}$ pour dériver $\Gamma \vdash e : \sigma(X)$.

2. Supposons $\Gamma \vdash e_1\ e_2 : \tau$, alors il existe τ' t.q. $\Gamma \vdash e_1 : \tau' \to \tau$ et $\Gamma \vdash e_2 : \tau'$.

   On écrit comme ci-dessus le problème d'unification que l'on cherche à résoudre (équation 5).

   Par induction, on récupère deux substitutions σ_1 et σ_2, qui sont des solutions constantes des problèmes d'inférence calculés pour Γ, e_i, X_i.

   Étant donné que Γ est un environnement de typage non étendu, tous les types mentionnés dans Γ sont constants. Cela nous autorise à supposer que les variables mentionnées dans les deux problèmes d'unification sont toutes distinctes, et donc que σ_1 et σ_2 sont de domaines disjoints.

   On construit alors une solution de $\mathcal{P}(\Gamma, e_1\ e_2, X)$ comme $\sigma = \sigma_1 \cup \sigma_2 \cup [\tau/X]$.

   On vérifie bien que $\sigma(X_1) = \sigma(X_2 \to X) = \tau' \to \tau$.

□

8 Sémantique axiomatique de Imp : logique de Floyd-Hoare

8.1 Logique de Floyd-Hoare

8.1.1 Triplets de Hoare

Définition 63 (Pré-assertions, satisfaction)**.** *Les* préassertions*, notées,* $A, A', B, B', \ldots$ *sont définies comme les expressions booléennes (ensemble* Bool*).*

On note $\sigma \models A$ *ssi* $A, \sigma \Downarrow \texttt{true}$*, et* $\sigma \not\models A$ *ssi* $A, \sigma \Downarrow \texttt{false}$*.*

Par abus de langage, on parlera parfois simplement d'assertions. Les préassertions sont une "version simplifiée" des (vraies) assertions ; on les introduit à des fins pédagogiques. Les définitions qui suivent valent pour les pré-assertions comme pour les assertions (on a juste besoin, pour écrire les règles de la logique de Hoare, que tout $b \in$ Bool peut être vu comme une (pré-)assertion.

Définition 64 (Triplet de Hoare, satisfaction, validité)**.** *Lorsque* $\sigma \models A$*, on dit que* σ satisfait A*.*

On dit que A *est* valide*, noté* $\models A$*, lorsque* $\sigma \models A$ *pour tout* $\sigma \in \mathcal{M}$*.*

Un triplet de Hoare $\{A\}\, c\, \{A'\}$ *est la donnée de deux assertions* A, A' *et d'une commande* c*. A est appelée la* précondition*,* A' *la* postcondition*.*

On dit que $\sigma \in \mathcal{M}$ satisfait *le triplet* $\{A\}\, c\, \{A'\}$*, noté* $\sigma \models \{A\}\, c\, \{A'\}$ *ssi, dès que* $\sigma \models A$ *et* $c, \sigma \Downarrow \sigma'$*, on a* $\sigma' \models A'$*.*

Le triplet $\{A\}\, c\, \{A'\}$ *est* valide*, noté* $\models \{A\}\, c\, \{A'\}$*, lorsque* $\sigma \models \{A\}\, c\, \{A'\}$ *pour tout* $\sigma \in \mathcal{M}$*.*

On peut remarquer que par définition, lorsque l'on a $\models A$, A ne parle pas d'un état mémoire particulier : on n'a pas, par exemple, $\models X \geq 1$. A est au contraire une "propriété mathématique", comme $(X \geq 3) \vee (X < 5)$, ou $(X \geq 1) \Rightarrow (X + 2 + Y \geq Y)$, ou encore $\texttt{false} \Rightarrow \texttt{true}$, qui sont toutes des assertions valides. Lorsque le langage des assertions permet de l'exprimer, A peut être un théorème (très) compliqué en mathématiques.

8.1.2 Sémantique axiomatique de Imp : règles de la logique de Hoare

On définit le jugement noté $\vdash \{A\}\, c\, \{A'\}$, que l'on prononce "le triplet $\{A\}\, c\, \{A'\}$ est *dérivable*", à l'aide des règles d'inférence suivantes :

$$\frac{}{\vdash \{A\}\, \texttt{skip}\, \{A\}}\ \mathrm{H_{skip}} \qquad \frac{\vdash \{A\}\, c_1\, \{A'\} \qquad \vdash \{A'\}\, c_2\, \{A''\}}{\vdash \{A\}\, c_1; c_2\, \{A''\}}\ \mathrm{H_{seq}} \qquad \frac{\vdash \{A \wedge b\}\, c_1\, \{A'\} \qquad \vdash \{A \wedge \neg b\}\, c_2\, \{A'\}}{\vdash \{A\}\, \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\, \{A'\}}\ \mathrm{H_{ite}}$$

$$\frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash A \texttt{while}\ b\ \texttt{do}\ c A \wedge \neg b}\ \mathrm{H_{while}} \qquad \frac{}{\vdash \{A[a/X]\}\, X := a\, \{A\}}\ \mathrm{H_{assgn}} \qquad \frac{\vdash \{B\}\, c\, \{B'\} \quad \models A \Rightarrow B \quad \models B' \Rightarrow A'}{\vdash \{A\}\, c\, \{A'\}}\ \mathrm{H_{cons}}$$

Quelques remarques à propos des règles d'inférence ci-dessus :

- La règle de conséquence, $\mathrm{H}_{\mathrm{cons}}$, est souvent présentée ainsi :

$$\frac{\models A \Rightarrow B \qquad \vdash \{B\}\, c\, \{B'\} \qquad \models B' \Rightarrow A'}{\vdash \{A\}\, c\, \{A'\}}$$

  Cela ne change pas fondamentalement les choses. Il faut essentiellement voir que $\models A \Rightarrow B$ et $\models B' \Rightarrow A'$ ne sont pas des propriétés que l'on obtient avec des dérivations, mais "en maths usuelles". C'est pour cette raisons qu'elles apparaissent comme conditions d'application dans la présentation de $\mathrm{H}_{\mathrm{cons}}$ qui est adoptée ici.

- Pour pouvoir former l'assertion ou $A \wedge \neg b$ (règle $\mathrm{H}_{\mathrm{ifte}}$), il faut que toute expression booléenne puisse être vue comme une assertion (et, éventuellement, il faut avoir l'opérateur de négation dans $\underline{\mathrm{Bool}}$, ce qui est fort raisonnable).

- Il faut par ailleurs comprendre que lorsqu'on écrit $A[a/X]$ dans la règle $\mathrm{H}_{\mathrm{assgn}}$, on manipule formellement une assertion, on n'est pas en train d'accéder à la mémoire via X ou quoi que ce soit de cet ordre.

- Une manière de se convaincre de la règle $\mathrm{H}_{\mathrm{assgn}}$ est de considérer $c = X := X + 1$, et de tenter de dériver un triplet ressemblant à $\{X = 2\}\, X := X + 1\, \{X = 3\}$.

Lemme 65. *Pour tout $b \in \underline{\mathrm{Bool}}$, on a $\sigma \models b$ ssi $b, \sigma \Downarrow \texttt{true}$.*

Cette propriété est vraie par définition pour les pré-assertions, il faut la vérifier pour les assertions (définition 68 ci-dessous).

Lemme 66. *Si $\models A \Rightarrow B$ et $\models A$, alors $\models B$.*

La correction dit que "dérivable implique valide" :

Théorème 67 (Correction de la logique de Hoare)**.** *Pour tous c, A, A', si $\vdash \{A\}\, c\, \{A'\}$, alors $\models \{A\}\, c\, \{A'\}$.*

Preuve. Par induction sur la relation $\vdash \{A\}\, c\, \{A'\}$, il y a 6 cas.

On ne détaille que le cas de la règle $\mathrm{H}_{\mathrm{while}}$, qui est le plus compliqué (le cas de la règle pour l'affectation n'est pas simple non plus, mais pour des raisons essentiellement techniques en lien avec l'opération de substitution).

On a $c = \texttt{while}\ b\ \texttt{do}\ c_0$, on sait $\vdash \{A \wedge b\}\, c_0\, \{A\}$, et on a par induction

$$HI\ : \forall \sigma',\ \text{si } c_0, \sigma \Downarrow \sigma' \text{ et } \sigma \models A \wedge b,\ \text{alors } \sigma' \models A.$$

On doit montrer $\models \{A\}\, \texttt{while}\ b\ \texttt{do}\ c_0\, \{A'\}$. Supposons donc $\texttt{while}\ b\ \texttt{do}\ c_0, \sigma \Downarrow \sigma'$ et $\sigma \models A$, il nous faut déduire $\sigma' \models A \wedge \neg b$. L'intuition est que tant que l'on boucle, l'invariant A est préservé, et en sortie de boucle, on saura de plus que $\neg b$ est satisfait.

Cela nous suggère de raisonner par induction sur la relation de dérivation $\texttt{while}\ b\ \texttt{do}\ c_0, \sigma \Downarrow \sigma'$ (ou, si l'on veut, sur le nombre de tours dans la boucle). On énonce donc la propriété suivante :

$$\forall c_1, \sigma_1, \sigma_1',\ (c_1, \sigma_1 \Downarrow \sigma_1') \ \Rightarrow\ \big(\text{si } c_1 = \texttt{while}\ b\ \texttt{do}\ c_0 \text{ et } \sigma_1 \models A,\ \text{alors } \sigma_1' \models A \wedge \neg b\big)$$

par induction sur la relation $c_1, \sigma_1 \Downarrow \sigma_1'$. Il faut comprendre ici que A, b et c_0 sont *fixés*, et que la formulation ci-dessus avec le "si $c_1 = \texttt{while}\ b\ \texttt{do}\ c_0$" permet de "faire tourner la moulinette de l'induction" en se restreignant à la commande $\texttt{while}\ b\ \texttt{do}\ c_0$.

Il y a 7 cas dans cette induction interne, dont 5 sont immédiats car ils correspondent à des règles d'évaluation pour des commandes qui ne sont pas une boucle. Restent donc les deux règles pour le `while`.

- Cas de la règle E_{wf}. On sait $b, \sigma \Downarrow \texttt{false}$ et $\sigma' = \sigma$, ce qui permet de déduire $\sigma' \models A \wedge \neg b$, en utilisant le Lemme 65.
- Cas de la règle E_{wt}. On sait alors $b, \sigma \Downarrow \texttt{true}$, $\exists \sigma_0$ t.q. $c_0, \sigma \Downarrow \sigma_0$ et $\texttt{while}\ b\ \texttt{do}\ c_0, \sigma_0 \Downarrow \sigma'$, et on a les deux hypothèses d'induction

$$HI_1 : \text{si } \sigma \models A \text{ alors } \sigma_0 \models A \wedge \neg b \qquad \text{et} \qquad HI_2 : \text{si } \sigma_0 \models A \text{ alors } \sigma' \models A \wedge \neg b.$$

(on omet ci-dessus la condition triviale correspondant à "si $c_1 = \texttt{while}\ b\ \texttt{do}\ c_0 \ldots$").

On peut maintenant emboîter les pièces de la preuve : comme $b, \sigma \Downarrow \texttt{true}$, on sait $\sigma \models b$ par le Lemme 65. Comme par ailleurs $\sigma \models A$, on déduit que $\sigma \models A \wedge b$. On applique alors HI (avec σ_0) pour déduire $\sigma_0 \models A$. Par HI_2, a enfin $\sigma' \models A \wedge \neg b$.

Remarquons que l'on s'appuie ici sur deux inductions : la première qui déconstruit une dérivation en logique de Hoare, la seconde, interne, qui déconstruit une exécution pour une boucle.

□

8.1.3 Programmes annotés

On souhaite dériver le triplet de Hoare suivant :
$\{ X \geq 0 \wedge Y \geq 0 \}$ `A:=0; B:=X; while B>=Y do (B:=B-Y; A:=A+1)` $\{ X = A * Y + B \wedge B \geq 0 \wedge B < Y \}$.

Pour ce faire, plutôt que d'écrire une dérivation, on annote le programme de la manière suivante :

$X \geq 0 \wedge Y \geq 0$
$X = 0 * Y + X \wedge X \geq 0$ $*$
`A:=0`
$X = A * Y + X \wedge X \geq 0$
`B:=X`
$X = A * Y + B \wedge B \geq 0$
`while B>=Y do` $X = A * Y + B \wedge B \geq 0$
$X = A * Y + B \wedge B \geq 0 \wedge B \geq Y$
$X = (A+1) * Y + B - Y \wedge B = Y \geq 0$ $*$
`B:=B-Y`
$X = (A+1) * Y + B \wedge B \geq 0$
`A:=A+1`
$X = A * Y + B \wedge B \geq 0$
$X = A * Y + B \wedge B \geq 0 \wedge \neg(B \geq Y)$
$X = A * Y + B \wedge B \geq 0 \wedge B < Y$ $*$

Les $*$ indiquent les endroits où la règle de conséquence est utilisée pour passer d'une assertion à la suivante.

En violet, l'invariant de boucle qui est utilisé pour appliquer la règle pour le `while`.

Les autres assertions découlent "automatiquement" de l'application des règles de la logique de Hoare. Ainsi, par exemple, $X = A * Y + B \wedge B \geq 0 \wedge \neg(B \geq Y)$ est la postcondition que l'on obtient en appliquant la règle pour le `while`.

8.1.4 Des pré-assertions aux assertions

On introduit maintenant la "vraie version" des assertions, qui remplace les pré-assertions (définition 63). Il s'agit de pouvoir affirmer des choses comme $\exists i, X * i \geq Y$ dans les assertions. L'introduction de la quantification existentielle sur les *index* (le i dans cet exemple) fait intervenir un certain attirail technique, raison pour laquelle on a commencé par les pré-assertions.

Définition 68 (Assertions)**.** *On se donne un ensemble $\mathcal{I}$ d'index ("inconnues"), notés $i, i', j, \ldots$. On définit l'ensemble $\underline{\text{Arith}}_{\mathcal{I}}$, des* expressions arithmétiques étendues*, notées $a, a', \ldots$, par la grammaire*

$$a \quad ::= \quad k \mid X \mid a_1 + a_2 \mid i.$$

On définit les assertions*, notées $A, B, A', \ldots$, par la grammaire*

$$A \quad ::= \quad \texttt{true} \mid a_1 \geq a_2 \mid \neg A \mid A_1 \wedge A_2 \mid \exists i.A.$$

Dans la grammaire pour les assertions, le cas $a_1 \geq a_2$ fait intervenir des expressions arithmétiques étendues (on peut avoir par exemple $X \geq i + Y$).

Définition 69. *Pour $a \in \underline{\text{Arith}}_{\mathcal{I}}$, on définit par induction sur a l'ensemble des index libres de a, noté $i\ell(a)$. Il y a 4 cas :*

$$i\ell(k) = \emptyset \qquad i\ell(X) = \emptyset \qquad i\ell(a_1 + a_2) = i\ell(a_1) \cup i\ell(a_2) \qquad i\ell(i) = \{i\}$$

L'ensemble des index libres d'une assertion A, noté $i\ell(A)$, est défini par induction sur A. Il y a 5 cas :

$$i\ell(\texttt{true}) = \emptyset \qquad i\ell(a_1 \geq a_2) = i\ell(a_1) \cup i\ell(a_2) \qquad i\ell(\neg A) = i\ell(A) \qquad i\ell(A_1 \wedge A_2) = i\ell(A_1) \cup i\ell(A_2)$$

$$i\ell(\exists i.A) = i\ell(A) \setminus \{i\}$$

Lorsque $i\ell(A) = \emptyset$, on dit que l'assertion A est close*.*

Définition 70. *Pour $a \in \underline{\text{Arith}}_{\mathcal{I}}, i \in \mathcal{I}, k \in \mathbb{Z}$, on définit $a[k/i]$ par induction sur a, il y a 4 cas :*

$$k'[k/i] = k' \qquad X[k/i] = X \qquad (a_1 + a_2)[k/i] = (a_1[k/i) + (a_2[k/i]) \qquad j[k/i] = \begin{array}{ll} j & si\ i \neq j \\ i & sinon \end{array}$$

Même chose pour la substitution de i par k dans une assertion A,il y a 5 cas :

$$\texttt{true}[k/i] = \texttt{true} \qquad (a_1 \geq a_2)[k/i] = (a_1[k/i]) \geq (a_2[k/i]) \qquad (\neg A)[k/i] = \neg(A[k/i])$$

$$(A_1 \wedge A_2)[k/i] = (A_1[k/i]) \wedge (A_2[k/i]) \qquad \begin{cases} (\exists j.A)[k/i] = \exists j.(A[k/i])\ si\ i \neq j \\ (\exists i.A)[k/i] = \exists i.A \end{cases}$$

Définition 71. *Étant donnés une assertion A close et $\sigma \in \mathcal{M}$, on définit par induction sur A une fonction associant un booléen à A, σ. On notera $\sigma \models A$ lorsque ce booléen est* `true`*, et $\sigma \not\models A$ lorsque c'est* `false`*.*
Il y a 5 cas :

- $\sigma \models \texttt{true}$*, pour tout σ*
- $\sigma \models a_1 \geq a_2$ *ssi* $a_1 \geq a_2, \sigma \Downarrow \texttt{true}$
- $\sigma \models \neg A$ *ssi* $\sigma \not\models A$
- $\sigma \models A_1 \wedge A_2$ *ssi* $\sigma \models A_1$ *et* $\sigma \models A_2$
- $\sigma \models \exists i.A$ *ssi pour un certain* $k \in \mathbb{Z}$*, on a* $\sigma \models A[k/i]$

Dans la définition ci-dessus, on peut remarquer que a_1 et a_2 sont des expressions arithmétiques Imp, car $a_1 \geq a_2$ est close par hypothèse. Similairement, si $\exists i.A$ est close, A contient au plus i comme index libre, et pour tout $k \in \mathbb{Z}$, $A[k/i]$ est une assertion close.

Avec les assertions, les choses se passent comme plus haut : on reprend la définition 64 telle quelle, ce qui permet de définir $\models A$, $\sigma \models \{A\}\, c\, \{A'\}$ et $\models \{A\}\, c\, \{A'\}$. On établit la correction de la logique de Hoare (théorème 67) de la même manière.

À propos de la complétude de la logique de Hoare. On se contente ici de dire quelques mots à propos de la preuve de la propriété de complétude de la logique de Hoare, qui dit que valide implique dérivable. Cette preuve s'appuie sur la notion de *plus faible précondition* : étant donnée une commande c et une formule B, la plus faible précondition associée à c, B est définie comme l'ensemble d'états mémoire

$$\mathrm{wp}(c, B) = \{\sigma \text{ t.q. } c, \sigma \Downarrow \sigma' \;\Rightarrow\; \sigma' \models B\}.$$

$\mathrm{wp}(c, B)$ décrit les états qui permettront d'aboutir à un état satisfaisant B.

On montre que l'on peut caractériser $\mathrm{wp}(c, B)$ à l'aide d'une assertion. Autrement dit, il existe une assertion $\mathrm{W}(c, B)$ telle que $\sigma \models \mathrm{W}(c, B)$ si et seulement si $\sigma \in \mathrm{wp}(c, B)$.

Après quoi on montre $\vdash \{\mathrm{W}(c, B)\}\, c\, \{B\}$, autrement dit que la logique de Hoare permet de dériver ce triplet, qui exprime ce qu'est une plus faible précondition.

Dès lors, étant donné un triplet $\{A\}\, c\, \{B\}$, on s'appuie sur la règle de conséquence pour écrire la dérivation

$$\frac{\vdash \{\mathrm{W}(c, B)\}\, c\, \{B\}}{\vdash \{A\}\, c\, \{B\}} \;\models A \Rightarrow \mathrm{W}(c, B)$$

Ce raisonnement permet de ramener la question de la dérivabilité du triplet $\{A\}\, c\, \{B\}$ à la validité d'une assertion (qui ne fait pas référence à une commande). Ainsi, on établit la complétude *relative* de la logique de Hoare : si un triplet est valide, il est dérivable modulo la possibilité de prouver la validité des assertions.

Pour davantage de détails à propos de la complétude de la logique de Hoare (qui est hors programme pour ce cours), on peut se référer au chapitre 7 du livre de G. Winskel, *"The formal semantics of programming languages: an introduction"*, MIT Press.

8.2 Logique séparante

8.2.1 Modifications du langage

On s'appuie ici sur l'extension de Imp vue à la partie 2.4. Afin de pouvoir écrire plus facilement des règles pour la logique séparante, on modifie le langage vu à la partie 2.4 de la manière suivante :

- On retire $[a]$ de la grammaire des expressions arithmétiques. Par conséquent, les expressions arithmétiques sont comme en Imp.
- La lecture dans le tas se fait par le biais d'une affectation de variable, tout comme l'allocation de mémoire. Ainsi, pour écrire `[X] := [Y+1] + 5`, on doit faire `V := [Y+1]; [X] := V+5`.
- La quantité de mémoire allouée à l'aide de l'instruction $\texttt{alloc}(\cdot)$ est déterminée *statiquement*.

Les grammaires pour les expressions arithmétiques et les commandes sont par conséquent définies ainsi :

$$a \quad ::= \quad k \mid X \mid a_1 + a_2$$

$$c \quad ::= \quad X := a \mid c_1; c_2 \mid \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } b \texttt{ do } c \mid \texttt{skip} \mid [a_1] := a_2 \mid X := [a] \mid X := \texttt{alloc}(k)$$

(on remarque que l'argument de $\texttt{alloc}(\cdot)$ est une constante).

On modifie les règles de sémantique opérationnelle à grands pas de la manière suivante :

$$\frac{a_2, \sigma \Downarrow k_2 \qquad a_1, \sigma \Downarrow k_1 \quad k_1 \in \mathrm{dom}(h)}{[a_1] := a_2, \sigma, h \Downarrow \sigma, h' \quad h' = h[k_1 \mapsto k_2]} \qquad \frac{a, \sigma \Downarrow k \quad h(k) = k'}{X := [a], \sigma, h \Downarrow \sigma', h \quad \sigma' = \sigma[X \mapsto k']}$$

$$\frac{\begin{array}{l}\{k', \ldots, k' + k - 1\} \cap \mathrm{dom}(h) = \emptyset \\ \sigma' = \sigma[X \mapsto k'] \\ h' = h \uplus [k' \mapsto 0, \ldots, k' + k - 1 \mapsto 0]\end{array}}{X := \texttt{alloc}(k), \sigma, h \Downarrow \sigma', h'}$$

On remarque que l'évaluation des expressions arithmétiques ne dépend pas du tas. De manière naturelle, la règle pour la lecture ne modifie pas h, et la règle pour la modification de la mémoire ne modifie pas σ.

Comme indiqué plus haut, le champ des possibles est vaste s'agissant du choix des commandes et de leur fonctionnement : on limite un peu la puissance du langage par rapport à la version proposée à la partie 2.4, afin de faciliter la présentation de la logique séparante.

8.2.2 Assertions en logique séparante

Comme les programmes manipulent le tas, les observations que l'on peut faire d'un état sont plus riches. Par ailleurs, le fait de manipuler explicitement une mémoire structurée met en évidence des possibilités de bugs qui sont "sous le tapis" dans Imp : accès invalides à la mémoire, phénomènes de partage difficiles à gérer, etc.

La logique séparante (*separation logic* en anglais) est une extension de la logique de Floyd-Hoare avec des constructions permettant d'énoncer des propriétés ayant trait à la structuration de la mémoire. Les assertions sont étendues de la manière suivante :

$$A \quad ::= \cdots \mid \mathsf{emp} \mid a_1 \mapsto a_2 \mid A_1 * A_2$$

Les assertions héritées de la logique de Hoare parlent de la composante σ de l'état mémoire : la définition de la satisfaction et de la validité est inchangée pour celles-ci.

Les assertions nouvelles parlent de la composante h. La satisfaction est définie comme suit :

- $\sigma, h \models \mathsf{emp}$ ssi $h = \emptyset$ (le tas est vide)
- $\sigma, h \models a_1 \mapsto a_2$ ssi $a_1, \sigma \Downarrow k_1$, $a_2, \sigma \Downarrow k_2$ et $h = [k_1 \mapsto k_2]$ (le tas est singleton)
- $\sigma, h \models A_1 * A_2$ ssi $h = h_1 \uplus h_2$ avec $\sigma, h_1 \models A_1$ et $\sigma, h_2 \models A_2$.

L'opérateur $*$ est appelé *conjonction séparante* : on découpe le tas en deux, chaque partie étant "observée" par une sous-assertion.

On introduit les abréviations suivantes :

$$\begin{array}{lcl} a \mapsto \_ & = & \exists i.\, a \mapsto i \ x \notin i\ell(a) \\ a_1 \hookrightarrow a_2 & = & a_1 \mapsto a_2 * \mathtt{true} \\ a \mapsto a_1, \ldots, a_n & = & a \mapsto a_1 * \ldots * a + n - 1 \mapsto a_n \end{array}$$

Exercice : dessinez des "fragments de tas" correspondant aux assertions suivantes.
- $x \mapsto 3, y$
- $y \mapsto 3, x$
- $x \mapsto 3, y * y \mapsto 3, x$
- $x \mapsto 3, y \wedge y \mapsto 3, x$
- $x \hookrightarrow 3, y \wedge y \hookrightarrow 3, x$

8.2.3 Règles d'inférence pour la logique

La validité des triplets de Hoare est définie comme en logique de Floyd-Hoare. À noter qu'avec la présentation que l'on a adoptée, les programmes sont bloqués si le programme essaie d'accéder à une case mémoire qui n'est pas allouée. On pourrait aussi ajouter un état d'échec à la sémantique opérationnelle, afin de pouvoir parler des états d'échec du fait d'erreurs d'accès à la mémoire. La validité des triplets spécifierait alors que l'on ne doit pas engendrer de telle erreur.

Comme pour la logique de Floyd-Hoare, on donne une règle pour la dérivation de triplets de Hoare pour chaque nouvelle forme de commande.

Modification en mémoire.

$$\overline{\vdash \{(a \mapsto \_)\}\, [a] := a' \,\{a \mapsto a'\}}$$

La précondition impose que la case mémoire soit allouée. Intuitivement, le triplet de Hoare ne "voit" qu'une case mémoire, celle sur laquelle on agit avec l'instruction d'affectation.

Allocation.

$$\overline{\vdash \{\mathsf{emp}\}\, X := \mathtt{alloc}(k)\, \{X \mapsto \underbrace{(0, \ldots, 0)}_{k \text{ fois}}\}}$$

Lecture en mémoire.

$$\overline{\vdash \{\exists i.\, (a \hookrightarrow i \wedge A[i/X])\}\, X := [a]\, \{A\}}$$

En raisonnant comme pour la règle pour l'affectation en logique de Floyd-Hoare, on aimerait écrire le triplet $\{A[[a]/X]\}\, X := [a]\, \{A\}$ en conclusion de la règle, mais on ne peut remplacer X par $[a]$ dans A, car $[a]$ ne fait pas partie du langage des expressions arithmétiques.

Noter par ailleurs l'usage de $\hookrightarrow$ ci-dessus, et pas de $\mapsto$: contrairement aux deux règles précédentes, cette règle n'est pas "seyante" (*tight*), car elle ne se restreint pas à un tas de dimension minimale pour pouvoir l'appliquer (zéro pour l'allocation, un pour la modification). Ici, A est quelconque, elle peut parler de beaucoup de cases mémoire : on demande juste que l'adresse correspondant à a soit allouée.

La *"frame rule"*.

$$\frac{\vdash \{A\}\, c\, \{A'\}}{\vdash \{A * B\}\, c\, \{A' * B\}} \text{ aucune variable de vars}(B) \text{ n'est modifiée par } c$$

Cette règle est essentielle car elle permet de raisonner *localement*, en "zoomant" sur la zone du tas où il se passe quelque chose.

8.2.4 Programme annoté

On considère le programme suivant :

```
x :=alloc(2); y :=alloc(2); [x] := 5; v := [x]; [y] := v+2; [x+1] := y-x; [y+1] := x-y
```

On souhaite dériver le triplet de Hoare correspondant à la précondition emp et à la post-condition $\exists i.\, (x \mapsto 5, i) * (x + i \mapsto 7, -i)$, qui exprime que l'on a créé une petite structure cyclique.

Pour écrire un programme annoté, le principe est sensiblement le même qu'en logique de Hoare (partie 8.1.3). Un point à souligner est l'usage fréquent de la *frame rule*, dès que le programme interagit avec le tas, afin de "transporter" des sous-assertions parlant de la partie du tas qui n'est pas affectée.

emp

`x :=alloc(2)`

$x \mapsto (0,0)$

`y :=alloc(2)`

$x \mapsto (0,0) * y \mapsto (0,0)$ $*$

$x \mapsto (\_,\_) * y \mapsto (\_,\_)$ $*$

`[x] := 5`

$x \mapsto (5,\_) * y \mapsto (\_,\_)$ $*$

$\exists i.\, (x \hookrightarrow i \wedge (x \mapsto (5,\_) * y \mapsto (\_,\_) \wedge i = 5))$ $*$

`v := [x]`

$x \mapsto (5,\_) * y \mapsto (\_,\_) \wedge v = 5$ $*$

`[y] := v+2`

$x \mapsto (5,\_) * y \mapsto (v+2,\_) \wedge v = 5$ $*$

$x \mapsto (5,\_) * y \mapsto (7,\_)$ $*$

`[x+1] := y-x`

$x \mapsto (5, y-x) * y \mapsto (7,\_)$ $*$

`[y+1] := x-y`

$x \mapsto (5, y-x) * y \mapsto (7, x-y)$ $*$

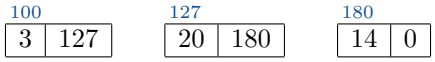
$\exists i.\, (x \mapsto (5,i)) * (x+i \mapsto (7,-i))$ $*$

$*$: usage de la règle de conséquence

$*$: usage de la *frame rule* (en lien avec un axiome pour l'une des commandes)

8.2.5 Structures de données et prédicats inductifs

On s'appuie sur la représentation des listes comme listes chaînées de "cellules" comportant deux cases mémoire consécutives dans le tas :



et on considère le programme suivant :

```
h := p;
s := 0;
while h <> 0 do
  s := s+1;
  h := [h+1]
done
```

Pour raisonner sur ce programme, on est amené à étendre le langage des assertions. L'idée est de travailler dans un cadre plus riche, où l'on peut faire des définitions inductives.

On introduit ainsi la définition suivante, qui exprime le fait que la représentation d'une liste Caml (premier argument du prédicat $\mathsf{listrep}(\cdot,\cdot)$) est stockée à l'adresse p :

$$\mathsf{listrep}([\,], p) = \mathsf{emp} \wedge p = 0 \qquad \mathsf{listrep}(\mathtt{x} :: \mathtt{xs}, p) = \exists q.\, \big(p \mapsto (\mathtt{x}, q) * \mathsf{listrep}(\mathtt{xs}, q)\big)$$

À l'aide de ce genre de formule, on peut exprimer une précondition et une postcondition qui font sens pour le programme ci-dessus, à savoir, respectivement, $\mathsf{listrep}(\mathtt{L}, p)$ et $(s = \mathtt{length}(L)) \wedge \mathsf{listrep}(\mathtt{L}, p) \wedge (h = 0)$.

On s'attache ensuite à exprimer un invariant pour la boucle. En cours d'exécution, la "vision" que le programme a de la mémoire fait intervenir une liste inachevée, ce que l'on peut appeler un *segment*, et que l'on peut exprimer ainsi :

$$\mathrm{listseg}([\,], p, q) = \mathsf{emp} \wedge p = q \qquad \mathrm{listseg}(\mathtt{x} :: \mathtt{xs}, p, q) = \exists p'.\, \big(p \mapsto (\mathtt{x}, p') * \mathrm{listseg}(\mathtt{xs}, p', q)\big)$$

listseg($\mathtt{L}, p, q$) exprime le fait que la liste `L` est stockée entre l'adresse p et l'adresse q, mais q n'est pas nécessairement la fin de la liste.

On peut alors écrire l'invariant $\exists L_1, L_2.\, (L = \mathtt{append}(L_1, L_2)) \wedge (s = \mathtt{length}(L_1)) \wedge \mathrm{listseg}(\mathtt{L}_1, p, h) * \mathsf{listrep}(\mathtt{L}_2, h)$

Exercice : le programme suivant manipule des listes implémentées comme ci-dessus. Comprendre ce qu'il fait, et en proposer une version annotée qui fasse sens.

```
q := 0;
while p <> 0 do
  r := [p+1];
  [p+1] := q;
  q := p;
  p := r
done
```