Théorie de la programmation

Basé sur le cours de Daniel Hirschkoff
Notes prises par Hugo Salou

18 décembre 2024

Table des matières

Introduction.

Dans ce cours, on étudie la *sémantique des langages de programmation*. On présente des approches pour

- ▷ définir rigoureusement ce qu'est/ce que fait un programme ;
- ▷ établir mathématiquement des propriétés sur des programmes.

Par exemple,

- ▷ démontrer l'absence de *bug* dans un programme ;
- ▷ démontrer des propriétés sur des programmes de transformation de programme ;
- ▷ l'étude des nouveaux langages de programmation.

Dans ce cours, les langages fonctionnels (OCaml, Haskell, Scheme, ...) auront un rôle central.

1 Induction.

Sommaire.

1.1 Définitions inductives d'ensembles.

Dans ce cours, les ensembles définis par induction représenteront les données utilisées par les programmes. De plus, les notions d'ensembles et de types seront identiques : on identifiera :

$$\underbrace{n \in \mathsf{nat}}_{\text{ensemble}} \quad \longleftrightarrow \quad \underbrace{n : \mathsf{nat}}_{\text{type}}.$$

Exemple 1.1 (Types définis inductivement). Dans le code ci-dessous, on définit trois types : le type `nat` représentant les entiers naturels (construction de Peano) ; le type `nlist` représentant les listes d'entiers naturels ; et le type `t` représentant les arbres binaires étiquetés par des entiers aux nœuds.

```
type nat = Z | S of nat
type nlist = Nil | Const of nat * nlist
type t1 = F | N of t1 * nat * t1
```

Code 1.1 | *Trois types définis inductivement*

Définition 1.1. La *définition inductive d'un ensemble* $\mathtt{t}$ est la donnée de k constructeurs $\mathtt{C}_1, \ldots, \mathtt{C}_k$, où chaque $\mathtt{C}_i$ a pour argument un n_i-uplet dont le type est $\mathtt{u}_1^i * \mathtt{u}_2^i * \cdots * \mathtt{u}_{n_i}^i$. L'opération « $*$ » représente le produit cartésien, avec une notation « à la OCaml ». De plus, chaque $\mathtt{u}_j^i$ est, soit $\mathtt{t}$, soit un type pré-existant.

Exemple 1.2.

```
type t2 =
| F
| N2 of (t * nlist * t)
| N3 of (t * nat * t * nat * t)
```

Code 1.2 | *Un exemple de type*

Définition 1.2. Les *types algébriques* sont définis en se limitant à deux opérations :

- ▷ le produit cartésien « $*$ » ;
- ▷ le « ou », noté « | » ou « $+$ », qui correspond à la somme disjointe ;

et un type :

- ▷ le type `unit`, noté 1.

Exemple 1.3 (Quelques types algébriques...).

- ▷ Le type bool est alors défini par $1 + 1$.
- ▷ Le type « jour de la semaine » est alors défini par l'expression $1 + 1 + 1 + 1 + 1 + 1 + 1$.
- ▷ Le type nat vérifie l'équation $X = 1 + X$.
- ▷ Le type nlist vérifie l'équation $X = 1 + \mathsf{nat} * X$.
- ▷ Le type t option est alors défini par $1 + \mathsf{t}$.

Ces ensembles définis inductivement nous intéresse pour deux raisons :

- ▷ pour pouvoir calculer, c'est à dire définir des fonctions de `t` vers `t'` en faisant du *filtrage* (*i.e.* avec `match ... with`)
- ▷ raisonner / prouver des propriétés sur les éléments de `t` : « des preuves par induction ».

1.2 Preuves par induction sur un ensemble inductif.

Exemple 1.4. Intéressons nous à `nat`. Pour prouver $\forall x \in \texttt{nat}, \mathcal{P}(x)$, il suffit de prouver *deux* choses (parce que l'on a deux constructeurs à l'ensemble `nat`) :

1. on doit montrer $\mathcal{P}(0)$;
2. et on doit montrer $\mathcal{P}(\texttt{S}\ n)$ en supposant l'*hypothèse d'induction* $\mathcal{P}(n)$.

Remarque 1.1. Dans le cas général, pour prouver $\forall x \in \texttt{t}, \mathcal{P}(x)$, il suffit de prouver les n propriétés (n est le nombre de constructeurs de l'ensemble `t`), où la i-ème propriété s'écrit :

> On montre $\mathcal{P}(\texttt{C}_i(x_1, \ldots, x_n))$ avec les hypothèses d'inductions $\mathcal{P}(x_j)$ lorsque $\texttt{u}_j^i = \texttt{t}$.

Exemple 1.5. Avec le type $\texttt{t}_2$ défini dans l'exemple 1.2, on a trois constructeurs, donc trois cas à traiter dans une preuve par induction. Le second cas s'écrit :

> On suppose $\mathcal{P}(x_1)$ et $\mathcal{P}(x_3)$ comme hypothèses d'induction, et on montre $\mathcal{P}(\texttt{N2}(x_1, k, x_3))$, où l'on se donne $k \in \texttt{nat}$.

Exemple 1.6. On pose la fonction `red` définie par le code ci-dessous.

```
let rec red k ℓ = match ℓ with
| Nil -> Nil
| Cons(x, ℓ) -> let ℓ'' = red k ℓ'' in
                 if x = k then ℓ''
                 else Cons(x, ℓ'')
```

Code 1.3 | *Fonction de filtrage d'une liste*

Cette fonction permet de supprimer toutes les occurrences de k dans une liste ℓ.

Démontrons ainsi la propriété

$$\forall \ell \in \mathsf{nlist}, \underbrace{\forall k \in \mathsf{nat}, \mathtt{size}(\mathtt{red}\ k\ \ell) \leq \mathtt{size}\ \ell}_{\mathscr{P}(\ell)}.$$

Pour cela, on procède par induction. On a *deux* cas.

1. Cas `Nil` : $\forall k \in \mathsf{nat}, \mathtt{size}(\mathtt{red}\ k\ \mathtt{Nil}) \leq \mathtt{size}\ \mathtt{Nil}$;
2. Cas $\mathtt{Cons}(x, \ell')$: on suppose

   $$\forall k \in \mathsf{nat}, \mathtt{size}(\mathtt{red}\ k\ \ell) \leq \mathtt{size}\ \ell,$$

   et on veut montrer que

   $$\forall k \in \mathsf{nat}, \mathtt{size}(\mathtt{red}\ k\ \mathtt{Cons}(x, \ell')) \leq \mathtt{size}\ \mathtt{Cons}(x, \ell'),$$

   ce qui demandera deux sous-cas : si $x = k$ et si $x \neq k$.

1.3 Définitions inductives de relations.

Dans ce cours, les relations définies par inductions représenterons des propriétés sur des programmes.

Un premier exemple : notations et terminologies.

Une relation est un sous-ensemble d'un produit cartésien. Par exemple, la relation $\mathsf{le} \subseteq \mathsf{nat} * \mathsf{nat}$ est une relation binaire. Cette relation re-

présente $\leq$, « *lesser than or equal to* » en anglais.

Notation. On note $\mathsf{le}(n,k)$ dès lors que l'on a $(n,k) \in \mathsf{le}$.

Pour définir cette relation, on peut écrire :

> Soit $\mathsf{le} \subseteq \mathsf{nat} * \mathsf{nat}$ la relation qui vérifie :
>
> 1. $\forall n \in \mathsf{nat}, \mathsf{le}(n,n)$;
> 2. $\forall (n,k) \in \mathsf{nat} * \mathsf{nat}$, si $\mathsf{le}(n,k)$ alors $\mathsf{le}(n, \mathtt{S}\ k)$.

mais, on écrira plutôt :

> Soit $\mathsf{le} \subseteq \mathsf{nat} * \mathsf{nat}$ la relation définie (inductivement) à partir des règles d'inférence suivantes :
>
> $$\frac{}{\mathsf{le}(n,n)}\ \mathscr{L}_1 \qquad \frac{\mathsf{le}(n,k)}{\mathsf{le}(n, \mathtt{S}\ k)}\ \mathscr{L}_2.$$

Remarque 1.2.
- ▷ Dans la définition par règle d'inférence, chaque règle a *une* conclusion de la forme $\mathsf{le}(\cdot,\cdot)$.
- ▷ Les *métavariables* n et k sont quantifiées universellement de façon implicite.

Définition 1.3. On appelle *dérivation* ou *preuve* un arbre construit en appliquant les règles d'inférence (ce qui fait intervenir l'*instantiation des métavariables*) avec des axiomes aux feuilles.

Exemple 1.7. Pour démontrer $\mathsf{le}(2,4)$, on réalise la dérivation ci-dessous.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{le}(2,2)}\ \mathscr{L}_1}{\mathsf{le}(2,3)}\ \mathscr{L}_2}{\mathsf{le}(2,4)}\ \mathscr{L}_2$$

Exemple 1.8. On souhaite définir une relation triée sur nlist. Pour

cela, on pose les trois règles ci-dessous :

$$\frac{}{\mathsf{triée}\ \mathtt{Nil}}\ \mathscr{T}_1 \qquad \frac{}{\mathsf{triée}\ \mathtt{Cons}(x, \mathtt{Nil})}\ \mathscr{T}_2,$$

$$\frac{\mathsf{le}(x, y) \qquad \mathsf{triée}\ \mathtt{Cons}(x, \mathtt{Nil})}{\mathsf{triée}\ \mathtt{Cons}(x, \mathtt{Cons}(y, \ell))}\ \mathscr{T}_3.$$

Ceci permet de dériver, modulo quelques abus de notations, que la liste `[1;3;4]` est triée :

$$\dfrac{\dfrac{\dfrac{}{1 \leq 1}\ \mathscr{L}_1}{1 \leq 2}\ \mathscr{L}_2}{1 \leq 3}\ \mathscr{L}_2 \qquad \dfrac{\dfrac{\dfrac{}{3 \leq 3}\ \mathscr{L}_1}{3 \leq 4}\ \mathscr{L}_2 \qquad \dfrac{}{\mathsf{triée}\ [4]}\ \mathscr{R}_2}{\mathsf{triée}\ [3;4]}\ \mathscr{R}_3}{\mathsf{triée}\ [1;3;4]}\ \mathscr{R}_3.$$

Les parties en bleu de l'arbre ne concernent pas la relation triée, mais la relation le.

Exemple 1.9. On définit la relation mem d'*appartenance* à une liste. Pour cela, on définit $\mathsf{mem} \subseteq \mathsf{nat} * \mathsf{nlist}$ par les règles d'inférences :

$$\frac{}{\mathsf{mem}(k, \mathtt{Cons}(k, \ell))}\ \mathscr{M}_1 \qquad \frac{\mathsf{mem}(k, \ell)}{\mathsf{mem}(k, \mathtt{Cons}(x, \ell))}\ \mathscr{M}_2.$$

On peut constater qu'il y a plusieurs manières de démontrer

$$\mathsf{mem}(0, \mathtt{[0;1;0]}).$$

Ceci est notamment dû au fait qu'il y a deux '0' dans la liste.

Remarque 1.3. Attention ! Dans les prémisses d'une règle, on ne peut pas avoir « $\neg\mathsf{r}(\ldots)$ ». Les règles ne peuvent qu'être « constructive », donc pas de négation.

Exemple 1.10. On définit la relation $\mathsf{ne} \subseteq \mathsf{nat} * \mathsf{nat}$ de non égalité entre deux entiers.

On pourrait imaginer créer une relation d'égalité et de définir ne comme sa négation. Mais non, c'est ce que nous dit la remarque 1.3.

On peut cependant définir la relation ne par :

$$\frac{}{\mathsf{ne}(\mathtt{Z}, \mathtt{S}\ k)}\ \mathscr{N}_1 \qquad \frac{}{\mathsf{ne}(\mathtt{S}\ n, \mathtt{Z})}\ \mathscr{N}_2 \qquad \frac{\mathsf{ne}(n,k)}{\mathsf{ne}(\mathtt{S}\ n, \mathtt{S}\ k)}\ \mathscr{N}_3.$$

Il est également possible de définir ne à partir de la relation le.

Exemple 1.11. En utilisant la relation ne (définie dans l'exemple 1.10), on peut revenir sur la relation d'appartenance et définir une relation alternative à celle de l'exemple 1.9. En effet, soit la relation mem' définie par les règles d'inférences ci-dessous :

$$\frac{}{\mathsf{mem}'(n, \mathtt{Cons}(n, \ell))}\ \mathscr{M}'_1 \qquad \frac{\mathsf{mem}'(n,\ell) \qquad \mathsf{ne}(k,n)}{\mathsf{mem}'(n, \mathtt{Cons}(k, \ell))}\ \mathscr{M}'_2.$$

Il est (*sans doute ?*) possible de montrer que :

$$\forall (n, \ell) \in \mathsf{nat} * \mathsf{nlist}, \mathsf{mem}(n,\ell) \iff \mathsf{mem}'(n,\ell).$$

Remarque 1.4. Dans le cas général, une définition inductive d'une relation Rel, c'est k règles d'inférences de la forme :

$$\frac{H_1 \quad \cdots \quad H_n}{\mathsf{Rel}(x_1, \ldots, x_m)}\ \mathscr{R}_i,$$

où chaque H_j est :

- ▷ soit $\mathsf{Rel}(\ldots)$;

- ▷ soit une autre relation pré-existante (*c.f.* la définition de triée dans l'exemple 1.8).

On appelle les H_j les *prémisses*, et $\mathsf{Rel}(x_1, \ldots, x_m)$ la *conclusion*. Elles peuvent faire intervenir des *métavariables*.

1.4 Preuves par induction sur une relation inductive.

On souhaite établir une propriété de la forme

$$\forall (x_1, \ldots, x_m),\ \mathsf{Rel}(x_1, \ldots, x_m) \implies \mathscr{P}(x_1, \ldots, x_m).$$

Pour cela, on établit autant de propriétés qu'il y a de règles d'inférences sur la relation Rel. Pour chacune de ces propriétés, on a une hypothèse d'induction pour chaque prémisse de la forme $\mathsf{Rel}(\ldots)$.

Exemple 1.12 (Induction sur la relation le.)**.** Pour prouver une propriété

$$\forall (n, k) \in \mathsf{nat} * \mathsf{nat}, \quad \mathsf{le}(n, k) \implies \mathscr{P}(n, k),$$

il suffit d'établir *deux* propriétés :

1. $\forall n, \mathscr{P}(n, n)$;
2. pour tout (n, k), montrer $\mathscr{P}(n, \mathtt{S}\ k)$ en supposant $\mathscr{P}(n, k)$.

Exemple 1.13. Supposons que l'on ait une fonction ayant pour signature `sort` : nlist → nlist qui trie une nlist. On souhaite démontrer la propriété :

$$\forall \ell \in \mathsf{nlist}, \mathsf{triée}(\ell) \implies \mathtt{sort}(\ell) = \ell.$$

On considère deux approches pour la démonstration : par induction sur ℓ et par induction sur la relation triée.

1. par induction sur la liste ℓ, il y a *deux* cas à traiter :

   - ▷ montrer que $\mathsf{triée}(\texttt{Nil}) \implies \texttt{sort}(\texttt{Nil}) = \texttt{Nil}$,
   - ▷ montrer que :

$$\mathsf{triée}(\texttt{Cons}(n, \ell)) \implies \texttt{sort}(\texttt{Cons}(n, \ell)) = \texttt{Cons}(n, \ell);$$

2. par induction sur la relation $\mathsf{triée}(\ell)$, il y a *trois* cas à traiter :
   - ▷ montrer $\texttt{sort}(\texttt{Nil}) = \texttt{Nil}$,
   - ▷ montrer $\texttt{sort}(\texttt{Cons}(n, \texttt{Nil})) = \texttt{Cons}(n, \texttt{Nil})$,
   - ▷ montrer $\texttt{sort}(\texttt{Cons}(x, \texttt{Cons}(y, \ell)) = \texttt{Cons}(x, \texttt{Cons}(y, \ell)))$, en supposant :
     - $\mathsf{triée}(\texttt{Cons}(y, \ell))$ et $\mathscr{P}(\texttt{Cons}(y, \ell))$, pour la première prémisse ;
     - $\mathsf{le}(x, y)$, pour la seconde prémisse.

2 Théorèmes de point fixe.

Sommaire.

Dans cette section, on va formaliser les raisonnements que l'on a réalisé en section 1 à l'aide du théorème de Knaster-Tarksi.

Définition 2.1. Soit E un ensemble, une relation $\mathscr{R} \subseteq E^2$ est un *ordre partiel* si $\mathscr{R}$ est :

- ▷ réflexive : $\forall x \in E,\ x\ \mathscr{R}\ x$;
- ▷ transitive : $\forall x, y, z \in E,\ (x\ \mathscr{R}\ y \text{ et } y\ \mathscr{R}\ z) \implies x\ \mathscr{R}\ z$;
- ▷ antisymétrique : $\forall x, y \in E,\ (x\ \mathscr{R}\ y \text{ et } y\ \mathscr{R}\ x) \implies x = y$.

Exemple 2.1. Dans l'ensemble $E = \mathbb{N}$, les relations $\leq$ et $\mid$ (division) sont des ordres partiels.

Définition 2.2. Soit $(E, \sqsubseteq)$ un ordre partiel.

- ▷ Un *minorant* d'une partie $A \subseteq E$ est un $m \in E$ tel que
$$\forall x \in A,\ m \sqsubseteq x.$$
- ▷ Un *majorant* d'une partie $A \subseteq E$ 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 $A \subseteq E$ admet un *plus petit majorant*, noté $\bigsqcup A$, et un *plus grand minorant*, noté $\bigsqcap A$.

Remarque 2.1.
- ▷ Pour tout minorant m de A, on a $m \sqsubseteq \bigsqcap A$.
- ▷ Pour tout majorant m' de A, on a $\bigsqcup A \sqsubseteq m'$.
- ▷ Un minorant/majorant de A n'est pas nécessairement dans l'ensemble A. Ceci est notamment vrai pour $\bigsqcap A$ et $\bigsqcup A$.

Notation. On note généralement $\bot = \bigsqcap E$, et $\top = \bigsqcup E$.

Exemple 2.2.
- ▷ L'ensemble $(\mathbb{N}, \leq)$ n'est pas un treillis complet : si A est infini, il n'admet pas de plus petit majorant.
- ▷ L'ensemble $(\mathbb{N} \cup \{\infty\}, \leq)$ est un treillis complet avec la convention $\forall n \in \mathbb{N}, n \leq \infty$.
- ▷ L'ensemble $(\mathbb{N}, |)$ est un treillis complet :
  - pour $A \subseteq \mathbb{N}$ fini, on a
$$\bigsqcup A = \operatorname{ppcm} A \quad \text{et} \quad \bigsqcap A = \operatorname{pgcd} A;$$
  - pour $A \subseteq \mathbb{N}$ infini, les relations ci-dessus restent valables avec la convention :
$$\forall n \in \mathbb{N},\quad n \mid 0.$$

Exemple 2.3 (Exemple *très* important de treillis complet). Soit E_0 un ensemble. Alors l'ensemble $(\wp(E_0), \subseteq)$ des parties de E_0 est un treillis complet. En particulier, on a :

$$\bigsqcap = \bigcap, \quad \bigsqcup = \bigcup, \quad \bot = \emptyset \quad \text{et} \quad \top = E_0.$$

Théorème 2.1 (Knaster-Tarski). Soit $(E, \sqsubseteq)$ un treillis complet. Soit f une fonction croissante de E dans E.[1] On considère l'ensemble

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

l'ensemble des *prépoints fixes* de f. Posons $m = \bigsqcap F_f$. Alors, m est un point fixe de f, *i.e.* $f(m) = m$.

Preuve. Soit $y \in F_f$, alors $m \sqsubseteq y$, et par croissance de f, on a ainsi $f(m) \sqsubseteq f(y)$, ce qui implique $f(m) \sqsubseteq y$ par transitivité (et car $y \in F_f$). D'où, $f(m)$ est un minorant de F_f.

Or, par définition, $f(m) \sqsubseteq m$, et par croissance $f(f(m)) \sqsubseteq f(m)$, ce qui signifie que $f(m) \in F_f$. On en déduit $m \sqsubseteq f(m)$.

Par antisymétrie, on en conclut que $f(m) = m$. □

À la suite de ce théorème, on peut formaliser les raisonnements que l'on a réalisé en section 1. Pour cela, il nous suffit d'appliquer le théorème 2.1 de Knaster-Tarksi (abrégé en « théorème K-T »).

2.1 Définitions inductives de relations.

Remarque 2.2. Pour justifier la définition des relations, on applique le théorème K-T. En effet, on part de $E = E_1 * \cdots * E_n$. Les relations sont des sous-ensembles de E, on travaille donc dans le treillis complet $(\wp(E), \subseteq)$. On se donné une définition inductive d'une relation $\mathsf{Rel} \subseteq E$. Pour cela, on s'appuie sur les règles

1. Ceci signifie que $\forall a, b \in E, \quad a \sqsubseteq b \implies f(a) \sqsubseteq f(b)$.

d'inférences et on associe à chaque $\mathscr{R}_i$ une fonction

$$f_i : \wp(E) \to \wp(E).$$

On montre (constate) que les f_i définies sont croissantes. Enfin, on pose pour $A \subseteq E$,

$$f(A) = f_1(A) \cup \cdots \cup f_k(A).$$

La fonction $f \mapsto f(A)$ est croissante.

Par définition, Rel est défini comme le plus petit (pré)-point fixe de la fonction f, qui existe par le théorème K-T (théorème 2.1).

Exemple 2.4. Définissons $\mathsf{le} \subseteq \mathsf{nat} * \mathsf{nat}$. On rappelle les règles d'inférences pour cette relation :

$$\frac{}{\mathsf{le}(n,n)}\ \mathscr{L}_1 \qquad \frac{\mathsf{le}(n,k)}{\mathsf{le}(n,\mathtt{S}\ k)}\ \mathscr{L}_2.$$

Avec un ensemble $A \subseteq \mathsf{nat} * \mathsf{nat}$, on définit

$$f_1(A) = \{(n,n) \mid n \in \mathsf{nat}\},$$

$$f_2(A) = \{(n, \mathtt{S}\ k) \mid (n,k) \in A\};$$

et on pose enfin

$$f(A) = f_1(A) \cup f_2(A).$$

La définition formelle de la relation le est le plus petit point fixe de f.

Exemple 2.5 (Suite de l'exemple 1.8)**.** Définissons $\mathsf{triée} \subseteq \mathsf{nlist}$. On rappelle les règles d'inférences pour cette relation :

$$\frac{}{\mathsf{triée}\ \mathtt{Nil}}\ \mathscr{T}_1 \qquad \frac{}{\mathsf{triée}\ \mathtt{Cons}(x, \mathtt{Nil})}\ \mathscr{T}_2,$$

$$\frac{\mathsf{le}(x,y) \qquad \text{triée } \mathtt{Cons}(x,\mathtt{Nil})}{\text{triée } \mathtt{Cons}(x,\mathtt{Cons}(y,\ell))}\ \mathcal{T}_3.$$

Avec un ensemble $A \subseteq \mathsf{nlist}$, on définit

$$\begin{aligned} f_1(A) &= \{\mathtt{Nil}\},\\ f_2(A) &= \{\mathtt{Cons}(k,\mathtt{Nil}) \mid k \in \mathsf{nat}\},\\ f_3(A) &= \left\{ \mathtt{Cons}(x,\mathtt{Cons}(y,\ell)) \;\middle|\; \begin{array}{l} \mathtt{Cons}(y,\ell) \in A \\ \mathsf{le}(x,y) \end{array} \right\}, \end{aligned}$$

et on pose enfin

$$f(A) = f_1(A) \cup f_2(A).$$

La définition formelle de la relation le est le plus petit point fixe de f.

Remarque 2.3. Dans les exemples ci-avant, même si l'on ne l'a pas précisé, les fonctions f_i sont bien croissantes pour l'inclusion $\subseteq$. C'est ceci qui assure l'application du théorème K-T (théorème 2.1).

Comme dit dans la remarque 1.3, on ne définit pas de règles d'induction de la forme

$$\frac{\neg\mathsf{Rel}(x'_1,\ldots,x'_{n'})}{\mathsf{Rel}(x_1,\ldots,x_n)} \quad \longrightarrow \quad \text{C'est interdit !}$$

En effet, la fonction f définie n'est donc plus croissante.

Remarque 2.4. Une relation R définie comme le plus petit point fixe d'une fonction f vérifie, mais on ne demande en rien que l'on ait $A \subseteq f(A)$ quel que soit $A \subseteq E$. En effet, pour

$$f(\{(3,2)\}) = \{(n,n) \mid n \in \mathsf{nat}\} \cup \{(3,1)\}$$

ne vérifie pas cette propriété.

2.2 Définitions inductives d'ensembles.

Exemple 2.6. On reprend le type $\mathtt{t}_2$ défini à l'exemple 1.2 :

```
type t₂ =
| F
| N2 of (t * nlist * t)
| N3 of (t * nat * t * nat * t)
```

Code 2.1 | *Un exemple de type*

On le définit en utilisant le théorème K-T (théorème 2.1) en posant :

$$f_1(A) = \{\mathtt{F}\}$$
$$f_2(A) = \{(x, \ell, y) \mid \ell \in \mathsf{nlist} \text{ et } (x, y) \in A^2\}$$
$$f_3(A) = \left\{ (x, k_1, y, k_2, z) \;\middle|\; \begin{array}{l} (x, y, z) \in A^3 \\ (k_1, k_2) \in \mathsf{nat}^2 \end{array} \right\},$$

puis, quel que soit A,

$$f(A) = f_1(A) \cup f_2(A) \cup f_3(A).$$

On pose ensuite $\mathtt{t}_2$ comme le plus petit point fixe de f.

Exemple 2.7. Avec $\mathsf{nat} = \{\mathtt{Z}, \mathtt{S}\ \mathtt{Z}, \mathtt{S}\ \mathtt{S}\ \mathtt{Z}, \ldots\}$, on utilise

$$f(A) = \{\mathtt{Z}\} \cup \{\mathtt{S}\ n \mid n \in A\},$$

et on pose nat le plus petit point fixe de f.

Et si on retire le cas de base ? Que se passe-t-il ? On pose la fonction

$$f'(A) = \{\mathtt{S}\ n \mid n \in A\}.$$

Le plus petit point fixe de f est l'ensemble vide $\emptyset$. On ne définit donc pas les entiers naturels.

Remarque 2.5. Après quelques exemples, il est important de se demander comment f est définie. C'est une fonction de la forme

$$f : \wp(\boxed{?}) \to \wp(\boxed{?}).$$

Quel est l'ensemble noté « $\boxed{?}$ » ? Quel est l'*ensemble ambiant* ?

La réponse est : c'est l'ensemble des arbres étiquetés par des chaînes de caractères.

Remarque 2.6. Pour définir inductivement un relation, on peut considérer qu'on construit un ensemble de dérivation.

Par exemple, pour le, on aurait

$$f_2(A) = \left\{ \frac{\delta}{\mathsf{le}(n, \mathtt{S}\ k)} \;\middle|\; \begin{array}{l} \delta \text{ est une dérivation de } \mathsf{le}(n,k) \text{ } i.e., \\ \delta \text{ est une dérivation dont } \mathsf{le}(n,k) \text{ est} \\ \text{à la racine} \end{array} \right\}.$$

2.3 Preuves par induction sur un ensemble inductif.

Remarque 2.7. Soit $\mathtt{t}$ un ensemble défini par induction par les constructeurs $\mathtt{C}_1, \ldots, \mathtt{C}_n$. On pose f tel que $\mathtt{t}$ est le plus petit pré-point fixe de f.

On veut montrer $\forall x \in \mathtt{t}, \mathscr{P}(x)$. Pour cela, on pose

$$A = \{x \in \mathtt{t} \mid \mathscr{P}(x)\},$$

et on montre que $f(A) \subseteq A$, *i.e.* A est un pré-point fixe de f. Ceci implique, par définition de $\mathtt{t}$, que $\mathtt{t} \subseteq A$, d'où

$$\forall x, x \in \mathtt{t} \implies \mathscr{P}(x).$$

Exemple 2.8. Expliquons ce que veut dire « montrer $f(A) \subseteq A$ » sur un exemple.

Pour nlist, on pose deux fonctions

$$f_1(A) = \{\texttt{Nil}\}$$
$$f_2(A) = \{\texttt{Cons}(k, \ell) \mid \ell \in A\}$$
$$.$$

Pour montrer $f(A) \subseteq A$, il y a *deux* cas :

- ▷ (pour f_1) montrer $\mathscr{P}(\texttt{Nil})$;
- ▷ (pour f_2) avec l'hypothèse d'induction $\mathscr{P}(\ell)$, et $k \in \mathsf{nat}$, montrer $\mathscr{P}(\texttt{Cons}(n, \ell))$.

2.4 Preuves par induction sur une relation inductive.

2.4.1 Une première approche...

Remarque 2.8. Soit Rel une relation définie comme le plus petit (pré)point fixe d'une fonction f, associée aux k règles d'inférences $\mathscr{R}_1, \ldots, \mathscr{R}_k$. On veut montrer que

$$\forall (x_1, \ldots, x_m) \in E, \quad \mathsf{Rel}(x_1, \ldots, x_m) \implies \mathscr{P}(x_1, \ldots, x_m).$$

Pour cela, on pose $A = \{(x_1, \ldots, x_m) \in E \mid \mathscr{P}(x_1, \ldots, x_m)\}$, et on montre que $f(A) \subseteq A$, *i.e.* que A est un prépoint fixe de f. Ainsi, on aura $\mathsf{Rel} \subseteq A$ et on aura donc montré

$$\forall (x_1, \ldots, x_m) \in E, \quad \mathsf{Rel}(x_1, \ldots, x_m) \implies \mathscr{P}(x_1, \ldots, x_m).$$

Exemple 2.9. Pour le, prouver $f(A) \subseteq A$ signifie prouver deux propriétés :

1. $\forall n \in \mathsf{nat}, \mathscr{P}(n)$;
2. $\forall (n,k) \in \mathsf{nat}^2, \underbrace{\mathscr{P}(n,k)}_{\text{hyp. ind.}} \implies \mathscr{P}(n, \mathtt{S}\ k)$

Exemple 2.10. Pour triée, on a *trois* propriétés à prouver :

1. $\mathscr{P}(\mathtt{Nil})$;
2. $\forall k \in \mathsf{nat}, \mathscr{P}(\mathtt{Cons}(k, \mathtt{Nil}))$;
3. $\forall (x,y) \in \mathsf{nat}^2, \forall \ell \in \mathsf{nlist}$,

$$\underbrace{\mathscr{P}(\mathtt{Cons}(y,\ell))}_{\text{hyp.ind}} \wedge \mathsf{le}(x,y) \implies \mathscr{P}(\mathtt{Cons}(x, \mathtt{Cons}(y,\ell))).$$

Remarque 2.9. Remarquons que dans l'exemple 2.10 ci-dessus, dans le 3ème cas, on n'a pas d'hypothèse $\mathsf{triée}(\mathtt{Cons}(y,\ell))$. Ceci vient du fait que, dans la remarque 2.7, l'ensemble A ne contient pas que des listes triées. La contrainte de la relation n'a pas été appliquée, on n'a donc pas accès à cette hypothèse.

2.4.2 Une approche plus astucieuse...

Remarque 2.10. On modifie légèrement le raisonnement présenté en remarque 2.7. On pose

$$A' = \{(x_1, \dots, x_m) \in E \mid \mathsf{Rel}(x_1, \dots, x_m) \wedge \mathscr{P}(x_1, \dots, x_m)\}.$$

On montre $f(A') \subseteq A'$ et donc, par définition de Rel, on aura l'inclusion $\mathsf{Rel} \subseteq A'$. Avec ce raisonnement, on peut utiliser des hypothèses, comme montré dans les exemples 2.11 et 2.12. Le but de la preuve n'est donc plus $\mathscr{P}(\dots)$ mais $\mathsf{Rel}(\dots) \wedge \mathscr{P}(\dots)$.

En rouge sont écrits les différences avec le raisonnement précédent.

Exemple 2.11 (Version améliorée de l'exemple 2.9)**.** Pour le, prouver $f(A) \subseteq A$ signifie prouver deux propriétés :

1. $\forall n \in \mathsf{nat}, \mathsf{le}(n,n) \wedge \mathscr{P}(n)$;
2. $\forall (n,k) \in \mathsf{nat}^2, \underbrace{\mathsf{le}(n,k) \wedge \mathscr{P}(n,k)}_{\text{hyp. ind.}} \implies \mathsf{le}(n, \mathtt{S}\ k) \wedge \mathscr{P}(n, \mathtt{S}\ k)$

Exemple 2.12 (Version améliorée de l'exemple 2.10)**.** Pour $\mathsf{triée}$, on a *trois* propriétés à prouver :

1. $\mathsf{triée}(\mathtt{Nil}) \wedge \mathscr{P}(\mathtt{Nil})$;
2. $\forall k \in \mathsf{nat}, \mathsf{triée}(\mathtt{Cons}(k, \mathtt{Nil})) \wedge \mathscr{P}(\mathtt{Cons}(k, \mathtt{Nil}))$;
3. $\forall (x,y) \in \mathsf{nat}^2, \forall \ell \in \mathsf{nlist}$,

$$\overbrace{\mathsf{triée}(\mathtt{Cons}(y,\ell)) \wedge \mathscr{P}(\mathtt{Cons}(y,\ell))}^{\text{hyp.ind}} \wedge \mathsf{le}(x,y)$$
$$\Downarrow$$
$$\mathsf{triée}(\mathtt{Cons}(x, \mathtt{Cons}(y,\ell))) \wedge \mathscr{P}(\mathtt{Cons}(x, \mathtt{Cons}(y,\ell)))$$

2.5 Domaines et points fixes.

Définition 2.3. Soit $(E, \sqsubseteq)$ un ordre partiel. Une *chaîne infinie* dans l'ensemble ordonné $(E, \sqsubseteq)$ est une suite $(e_n)_{n\geq 0}$ telle que

$$e_0 \sqsubseteq e_1 \sqsubseteq e_2 \sqsubseteq \cdots .$$

On dit que $(E, \sqsubseteq)$ est *complet* si pour toute chaîne infinie, il existe $\bigsqcup_{n\geq 0} e_n \in E$, un plus petit majorant dans E.

Si, de plus, E a un plus petit élément $\bot$, alors $(E, \sqsubseteq)$ est un *domaine*.

Remarque 2.11. Un treillis complet est un domaine.

Théorème 2.2. Soit $(E, \sqsubseteq)$ un domaine. Soit $f : E \to E$ *continue* :

- ▷ f est croissante ;
- ▷ pour toute chaîne infinie $(e_n)_{n\geq 0}$,

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

Les $(f(e_n))_{n\geq 0}$ forment une chaîne infinie par croissance de la fonction f.

On pose, quel que soit $x \in E$, $f^0(x) = x$, et pour tout entier $i \geq 0$, on définit $f^{i+1}(x) = f(f^i(x))$.

On pose enfin

$$\begin{aligned}\mathrm{fix}(f) &= \bigsqcup_{n\geq 0} f^n(\bot) \\ &= \bot \sqcup f(\bot) \sqcup f^2(\bot) \sqcup \cdots .\end{aligned}$$

Alors, $\mathrm{fix}(f)$ est le plus petit point fixe de f.

Preuve. La preuve viendra plus tard. □

Les définitions inductives par constructeurs ou règles d'inférences peuvent être définis par des fonctions continues. Et, on peut se placer dans le domaine $(\wp(E), \subseteq)$ pour définir les ensembles définis par inductions.

Exemple 2.13. Avec les listes d'entiers, on définit

$$\mathtt{nat} = \underbrace{\emptyset}_{\bot} \cup \underbrace{\{\mathtt{Nil}\}}_{f(\bot)} \cup \underbrace{\{\mathtt{Cons}(k, \mathtt{Nil}) \mid k \in \mathtt{nat}\}}_{f^2(\bot)} \cup \cdots .$$

3 Les bases de Rocq.

3.1 Les définitions par induction : `Inductive`.

En Rocq (anciennement Coq), on peut définir des ensembles par induction. Pour cela, on utilise le mot `Inductive`.

Par exemple, pour définir un type de liste d'entiers, on utilise le code ci-dessous.

```
Inductive nlist : Set:=
| Nil : nlist
| Cons : nat → nlist → nlist.
```

Code 3.1 | *Définition du type* nlist *en Rocq*

En Rocq, au lieu de définir la fonction `Cons` comme une « fonction » de la forme `Cons` : nat∗nlist → nlist, on la *curryfie* en une « fonction » de la forme `Cons` : nat → nlist → nlist. Les types définis par les deux versions sont isomorphes.

Pour définir une relation, on utilise aussi le mot clé `Inductive` :

```
Inductive le : nat → nat → Prop :=
| le_refl : forall n, le n n
| le_S : forall n k, le n k → le (S n) (S k).
```

Code 3.2 | *Définition de la relation* le

Aux types définis par induction, on associe un principe d'induction (qu'on voir avec `Print le_ind.` ou `Print nlist_ind.`). Ce principe d'induction permet de démontrer une propriété $\mathcal{P}$ sur un ensemble/une relation définie par induction.

3.2 Quelques preuves avec Rocq.

On décide de prouver le lemme suivant avec Rocq.

> **"Lemme" 3.1.** Soit ℓ une liste triée, et soient a et b deux entiers tels que $a \leq b$. Alors la liste $a :: b :: \ell$ est triée.

Pour cela, on écrit en Rocq :

```
Lemma exemple_triee :
  forall l, triée l →
    forall a b, le a b →
      triée (Cons a (Cons b l)).
```

Il ne reste plus qu'à prouver ce lemme. On commence la démonstration par introduire les variables et hypothèses : les variables `l`, `a`, `b`, et les hypothèses (`H1`) : triée `l`, et (`H2`) : le `a` `b`. On commence par introduire la liste `l` et l'hypothèse `H1` et on s'occupera des autres un peu après.

```
Proof.
  intros l H1.
```

On décide de réaliser une preuve par induction sur la relation triée, qui est en hypothèse (`H1`).

```
  induction H1.
```

Dans le cas d'une preuve par induction sur triée, on a *trois* cas.

- ▷ *Cas 1.* On se trouve dans le cas `l` = `Nil`. Pas trop de problèmes pour prouver que `[a;b]` est triée avec l'hypothèse `a` ≤ `b`. On introduit les variables et hypothèses `a`, `b` et `H2`.

  ```
  - intros a b H2.
  ```

  À ce moment de la preuve, l'objectif est de montrer :

  $$\text{triée } \texttt{Cons}(\texttt{a}, \texttt{Cons}(\texttt{b}, \texttt{Nil})).$$

  Pour cela, on utilise deux fois les propriétés de la relation triée :

```
apply t_cons.
apply t_singl.
```

Notre objectif a changé, on doit maintenant démontrer `le a b`. C'est une de nos hypothèses, on peut donc utiliser :

```
assumption.
```

Ceci termine le cas 1.

- ▷ *Cas 2.* On se trouve dans le cas `l = [k]`. On doit de démontrer que la liste `[a;b;k]` est triée. On a l'hypothèse `a` $\leq$ `b`, mais aucune hypothèse de la forme `b` $\leq$ `k`. On est un peu coincé pour ce cas...

(Un jour je finirai d'écrire cette partie... Malheureusement, ce n'est pas aujourd'hui...)

4 Sémantique opérationnelle pour les expressions arithmétiques simples (EA).

Sommaire.

Depuis le début du cours, on s'est intéressé à la *méthode inductive*. On essaie d'appliquer cette méthode à « l'exécution » des « programmes ».

On définira un programme comme un ensemble inductif : un programme est donc une structure de donnée. L'exécution d'un programme sera décrit comme des relations inductives (essentiellement binaires) sur les programmes. Définir ces relations, cela s'appelle la *sémantique opérationnelle*.

On considèrera deux sémantiques opérationnelles

- ▷ la sémantique à grands pas, où l'on associe un résultat à un programme ;
- ▷ la sémantique à petits pas, où l'on associe un programme « un peu plus tard » à un programme.

Notre objectif, dans un premier temps, est de définir OCaml, ou plutôt un plus petit langage fonctionnel inclus dans OCaml.

On se donne l'ensemble $\mathbb{Z}$ (on le prend comme un postulat). On définit l'ensemble EA en Rocq par :

```
Inductive EA : Set :=
|Cst : ℤ → EA
|Add : EA → EA → EA.
```

Code 4.1 | *Définition des expressions arithmétiques simples*

Note 4.1. On se donne $\mathbb{Z}$ et on note $k \in \mathbb{Z}$ (vu comme une métavariable). On définit (inductivement) l'ensemble EA des expressions arithmétiques, notées $a, a', a_1, \ldots$ par la grammaire

$$a ::= \underline{k} \mid a_1 \oplus a_2.$$

Exemple 4.1. L'expression $\underline{1} \oplus (\underline{3} \oplus \underline{7})$ représente l'expression Rocq

$$\texttt{Add}(\texttt{Cst } 1, \texttt{Add } (\texttt{Cst } 3)\ (\texttt{Cst } 7)),$$

que l'on peut représenter comme l'arbre de syntaxe...

Remarque 4.1. Dans le but de définir un langage minimal, il n'y a donc pas d'intérêt à ajoute $\ominus$ et $\otimes$, représentant la soustraction et la multiplication.

4.1 Sémantique à grands pas sur EA.

On définit la sémantique opérationnelle à grands pas pour EA. L'intuition est d'associer l'exécution d'un programme avec le résultat. On définit la relation d'évaluation $\Downarrow \subseteq \mathsf{EA} * \mathbb{Z}$, avec une notation infixe, définie par les règles d'inférences suivantes :

$$\frac{}{\underline{k} \Downarrow k} \quad \text{et} \quad \frac{a_1 \Downarrow k_1 \qquad a_2 \Downarrow k_2}{a_1 \oplus a_2 \Downarrow k},$$

où, dans la seconde règle d'inférence, $k = k_1 + k_2$. Attention, le $+$ est la somme dans $\mathbb{Z}$, c'est une opération *externalisée*. Vu qu'on ne sait pas comment la somme a été définie dans $\mathbb{Z}$ (on ne sait pas si elle est définie par induction/point fixe, ou pas du tout), on ne l'écrit pas dans la règle d'inférence.

La forme générale des règles d'inférences est la suivante :

$$\text{Cond. App.}\ \frac{P_1 \quad \dots \quad P_m}{C}\ \mathscr{R}_i$$

où l'on donne les conditions d'application (ou *side condition* en anglais). Les $P_1, \dots, P_m, C$ sont des relations inductives, mais les conditions d'applications **ne sont pas** forcément inductives.

Exemple 4.2.

$$\dfrac{\underline{3} \Downarrow 3 \qquad \overset{2+5=7}{\dfrac{\underline{2} \Downarrow 2 \qquad \underline{5} \Downarrow 5}{(\underline{2} \oplus \underline{5}) \Downarrow 7}}}{\underline{3} \oplus (\underline{2} \oplus \underline{5}) \Downarrow 10}\ {\scriptstyle 3+7=10}\ .$$

4.2 Sémantique à petits pas sur EA.

On définit ensuite la sémantique opérationnelle à *petits pas* pour EA. L'intuition est de faire un pas exactement (la relation n'est donc pas réflexive) dans l'exécution d'un programme et, si possible, qu'elle soit déterministe.

Une relation *déterministe* (ou *fonctionnelle*) est une relation $\mathscr{R}$ telle que, si $a\ \mathscr{R}\ b$ et $a\ \mathscr{R}\ c$ alors $b = c$.

La relation de réduction $\to \subseteq \mathsf{EA} * \mathsf{EA}$, notée infixe, par les règles d'inférences suivantes

$$k = k_1 + k_2\ \frac{}{\underline{k_1} \oplus \underline{k_2} \to \underline{k}}\ \mathscr{A},$$

$$\frac{a_2 \to a_2'}{a_1 \oplus a_2 \to a_1 \oplus a_2'}\ \mathscr{C}_{\mathrm{d}} \quad \text{et} \quad \frac{a_1 \to a_1'}{a_1 \oplus \underline{k} \to a_1' \oplus \underline{k}}\ \mathscr{C}_{\mathrm{g}}.$$

Il faut le comprendre par « quand c'est fini à droite, on passe à gauche ».

Les règles $\mathscr{C}_{\mathrm{g}}$ et $\mathscr{C}_{\mathrm{d}}$ sont nommées respectivement *règle contextuelle droite* et *règle contextuelle gauche*. Quand $a \to a'$, on dit que a se *réduit* à a'.

Remarque 4.2. La notation $\underline{k} \not\to$ indique que, quelle que soit l'expression $a \in \mathsf{EA}$,on n'a pas $\underline{k} \to a$. Les constantes ne peuvent pas être exécutées.

Exercice 4.1. Et si on ajoute la règle

$$\frac{a_1 \to a_1' \qquad a_2 \to a_2'}{a_1 \oplus a_2 \to a_1' \oplus a_2'}\ ,$$

appelée *réduction parallèle*, que se passe-t-il ?

Remarque 4.3. Il n'est pas possible de démontrer $\underline{2} \oplus (\underline{3} \oplus \underline{4}) \to \underline{9}$. En effet, on réalise *deux* pas.

4.3 Coïncidence entre grands pas et petits pas.

On définit la clôture réflexive et transitive d'une relation binaire $\mathscr{R}$ sur un ensemble E, notée $\mathscr{R}^\star$. On la définit par les règles d'inférences suivantes :

$$\frac{}{x\ \mathscr{R}^\star\ x} \quad \text{et} \quad \frac{x\ \mathscr{R}\ y \qquad y\ \mathscr{R}^\star\ z}{x\ \mathscr{R}^\star\ z}\ .$$

Lemme 4.1. La relation $\mathscr{R}^\star$ est transitive.

Preuve. On démontre

$$\forall x, y \in E, \quad \text{si } x \mathrel{\mathscr{R}^\star} y \text{ alors } \underbrace{\forall z,\ y \mathrel{\mathscr{R}^\star} z \implies x \mathrel{\mathscr{R}^\star} z}_{\mathscr{P}(x,y)}$$

par induction sur $x \mathrel{\mathscr{R}^\star} y$. Il y a *deux* cas.

- ▷ *Réflexivité.* On a donc $x = y$ et, par hypothèse, $y \mathrel{\mathscr{R}^\star} z$.
- ▷ *Transitivité.* On sait que $x \mathrel{\mathscr{R}} a$ et $a \mathrel{\mathscr{R}^\star} y$. De plus, on a l'hypothèse d'induction

$$\mathscr{P}(a, y) : \forall z, y \mathrel{\mathscr{R}^\star} z \implies a \mathrel{\mathscr{R}^\star} z.$$

Montrons $\mathscr{P}(x, y)$. Soit z tel que $y \mathrel{\mathscr{R}^\star} z$. Il faut donc montrer $x \mathrel{\mathscr{R}^\star} z$. On sait que $x \mathrel{\mathscr{R}} a$ et, par hypothèse d'induction, $a \mathrel{\mathscr{R}^\star} z$. Ceci nous donne $x \mathrel{\mathscr{R}^\star} z$ en appliquant la seconde règle d'inférence.

□

Lemme 4.2. Quelles que soient a_2 et a_2', si $a_2 \to^\star a_2'$, alors pour tout a_1, on a $a_1 \oplus a_2 \to^\star a_1 \oplus a_2'$.

Preuve. On procède par induction sur $a_2 \to^\star a_2'$. Il y a *deux* cas.

1. On a $a_2' = a_2$. Il suffit donc de montrer que l'on a

$$a_1 \oplus a_2 \to^\star a_1 \oplus a_2,$$

ce qui est vrai par réflexivité.

2. On sait que $a_2 \to a$ et $a \to^\star a_2'$. On sait de plus que

$$\forall a_1, \quad a_1 \oplus a \to^\star a_1 \oplus a_2'$$

par hypothèse d'induction. On veut montrer que

$$\forall a_1, \quad a_1 \oplus a_2 \to^\star a_1 \oplus a_2'.$$

On se donne a_1. On déduit de $a_2 \to a$ que $a_1 \oplus a_2 \to a_1 \oplus a$ par $\mathscr{C}_\mathrm{d}$. Par hypothèse d'induction, on a $a_1 \oplus a \to^\star a_1 \oplus a_2'$. Par la seconde règle d'inférence, on conclut.

□

Lemme 4.3. Quelles que soient les expressions a_1 et a_1', si $a_1 \to^\star a_1'$ alors, pour tout k, $a_1 \oplus \underline{k} \to^\star a_1' \oplus \underline{k}$. □

Attention, le lemme précédent est faux si l'on remplace $\underline{k}$ par une expression a_2. En effet, a_2 ne peut pas être « spectateur » du calcul de a_1.

Proposition 4.1. Soient a une expression et k un entier. On a l'implication

$$a \Downarrow k \implies a \to^\star \underline{k}.$$

Preuve. On le démontre par induction sur la relation $a \Downarrow k$. Il y a *deux* cas.

1. Dans le cas $a = \underline{k}$, alors on a bien $\underline{k} \to^\star \underline{k}$.
2. On sait que $a_1 \Downarrow k_1$ et $a_2 \Downarrow k_2$, avec $k = k_1 + k_2$. On a également deux hypothèses d'induction :
   - ▷ $(H_1) : a_1 \to^\star \underline{k}_1$;
   - ▷ $(H_2) : a_2 \to^\star \underline{k}_2$.

   On veut montrer $a_1 \oplus a_2 \to^\star \underline{k}$, ce que l'on peut faire par :

$$a_1 \oplus a_2 \xrightarrow{(H_2)+\text{lemme 4.2}}{}^\star\ a_1 \oplus \underline{k}_2 \xrightarrow{(H_1)+\text{lemme 4.3}}{}^\star\ \underline{k}_1 \oplus \underline{k}_2 \xrightarrow{\mathscr{A}} \underline{k}.$$

□

Proposition 4.2. Soient a une expression et k un entier. On a l'implication

$$a \to^\star \underline{k} \implies a \Downarrow k.$$

□

4.4 L'ensemble EA avec des erreurs à l'exécution.

On exécute des programmes de EA. On considère que $\underline{k}_1 \oplus \underline{k}_2$ s'évalue comme

$$\frac{(k_1 + k_2) \times k_2}{k_2}.$$

Le cas $k_2 = 0$ est une situation d'erreur, une « ***situation catastrophique*** ». (C'est une convention : quand un ordinateur divise par zéro, il explose !)

4.4.1 Relation à grands pas.

On note encore $\Downarrow$ la relation d'évaluation sur $\mathsf{EA} * \mathbb{Z}_\perp$, où l'on définit l'ensemble $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$. Le symbole $\perp$ est utilisé pour représenter un cas d'erreur.

Les règles d'inférences définissant $\Downarrow$ sont :

$$\frac{}{\underline{k} \Downarrow k} \qquad \underset{k \neq 0}{^{k = k_1 + k_2}} \ \frac{a_1 \Downarrow k_1 \qquad a_2 \Downarrow k_2}{a_1 \oplus a_2 \Downarrow k} \qquad \frac{a_1 \Downarrow k_1 \qquad a_2 \Downarrow 0}{a_1 \oplus a_2 \Downarrow \perp},$$

et les règles de propagation du $\perp$:

$$\frac{a_1 \Downarrow \perp \qquad (a_2 \Downarrow r)}{a_1 \oplus a_2 \Downarrow \perp} \qquad \frac{(a_1 \Downarrow r) \qquad a_2 \Downarrow \perp}{a_1 \oplus a_2 \Downarrow \perp}.$$

4.4.2 Relation à petits pas.

On (re)-définit la relation $\to \subseteq \mathsf{EA} * \mathsf{EA}_\perp$, où $\mathsf{EA}_\perp = \mathsf{EA} \cup \{\perp\}$, par les règles d'inférences

$$\underset{k_2 \neq 0}{^{k = k_1 + k_2}} \ \frac{}{\underline{k}_1 \oplus \underline{k}_2 \to \underline{k}} \qquad {}^{a_2 \neq \perp} \ \frac{a_2 \to a_2'}{a_1 \oplus a_2 \to a_1 \oplus a_2'}$$

$$^{a_1 \neq \perp} \ \frac{a_1 \to a_1'}{a_1 \oplus \underline{k} \to a_1' \oplus \underline{k}} \qquad \frac{}{\underline{k}_1 \oplus \underline{0} \to \perp},$$

et les règles de propagation du $\bot$:

$$\frac{a_1 \to \bot}{a_1 \oplus \underline{k} \to \bot} \quad \text{et} \quad \frac{a_2 \to \bot}{a_1 \oplus a_2 \to \bot}.$$

Pour démontrer l'équivalence des relations grand pas et petits pas, ça semble un peu plus compliqué...

4.5 Sémantique contextuelle pour EA.

On définit la relation $\mapsto : \mathsf{EA} \times \mathsf{EA}$ par la règle :

$$\frac{}{E[\underline{k_1} \oplus \underline{k_2}] \mapsto E[\underline{k}],} \; k = k_1 + k_2$$

où E est un *contexte d'évaluation* que l'on peut définir par la grammaire

$$E ::= [\,] \mid \boxed{?}.$$

Le *trou* est une constante, notée $[\,]$ qui n'apparaît qu'une fois par contexte d'évaluation. Pour E un contexte d'évaluation et $a \in \mathsf{EA}$, alors $E[a]$ désigne l'expression arithmétique obtenue en remplaçant le trou par a dans E.

Exemple 4.3. On note $E_0 = \underline{3} \oplus ([\,] \oplus \underline{5})$ et $a_0 = \underline{1} \oplus \underline{2}$. Alors

$$\underline{3} \oplus ((\underline{1} \oplus \underline{2}) \oplus \underline{5}).$$

Que faut-il mettre à la place de ? ?

Exemple 4.4 (Première tentative)**.** On pose

$$E ::= [\,] \mid \underline{k} \mid E_1 \oplus E_2.$$

Mais, ceci peut introduire *plusieurs* trous (voire aucun) dans un même contexte. C'est raté.

Exemple 4.5 (Seconde tentative)**.** On pose

$$E ::= [\,] \mid a \oplus E \mid E \oplus a.$$

Mais, on pourra réduire une expression à droite avant de réduire à gauche. C'est encore raté.

Exemple 4.6 (Troisième (et dernière) tentative)**.** On pose

$$E ::= [\,] \mid a \oplus E \mid E \oplus \underline{k}.$$

Là, c'est réussi !

Lemme 4.4. Pour toute expression arithmétique $a \in \mathsf{EA}$ qui n'est pas une constante, il existe un unique triplet (E, k_1, k_2) tel que

$$a = E[\underline{k}_1 \oplus \underline{k}_2].$$

Ceci permet de justifier la proposition suivante, notamment au niveau des notations.

Proposition 4.3. Pour tout a, a', on a

$$a \to a' \quad \text{si, et seulement si,} \quad a \mapsto a'.$$

Preuve. Pour démontrer cela, on procède par double implication :

- ▷ « $\Longrightarrow$ » par induction sur $a \to a'$;
- ▷ « $\Longleftarrow$ » par induction sur E.

□

5 Sémantique opérationnelle des expressions arithmétiques avec déclarations locales (LEA).

Sommaire.

On suppose donnés $\mathbb{Z}$ les entiers relatifs et $\mathscr{V}$ un ensemble infini de variables (d'identifiants/d'identificateurs/de noms).

On définit LEA par la grammaire suivante :

$$a ::= \underline{k} \mid a_1 \oplus a_2 \mid \texttt{let}\, x \,\texttt{=}\, a_1 \;\texttt{in}\; a_2 \mid x,$$

où $x \in \mathscr{V}$ et $k \in \mathbb{Z}$.

En Rocq, on peut définir :

```
Inductive LEA : Set :=
| Cst : ℤ → LEA
| Add : LEA → LEA → LEA
| Let : 𝒱 → LEA → LEA → LEA
```

```
| Var : 𝒱 → LEA.
```

Code 5.1 | *Définition inductive de* LEA

Exemple 5.1. Voici quelques exemples d'expressions avec déclarations locales :

1. $\texttt{let}\, x = 3\ \texttt{in}\ x \oplus x$;
2. $\texttt{let}\, x = \underline{2}\ \texttt{in let}\, y = x \oplus \underline{2}\ \texttt{in}\ x \oplus y$;
3. $\texttt{let}\, x = (\texttt{let}\, y = \underline{5}\ \texttt{in}\ y \oplus y)\ \texttt{in}\ (\texttt{let}\, z = \underline{6}\ \texttt{in}\ z \oplus \underline{2}) \oplus x$;
4. $\texttt{let}\, x = \underline{7} \oplus \underline{2}\ \texttt{in}\ (\texttt{let}\, x = 5\ \texttt{in}\ x \oplus x) \oplus x$.

5.1 Sémantique à grands pas sur LEA.

On définit une relation d'*évaluation* $\Downarrow : \mathsf{LEA} * \mathbb{Z}$ [1] définie par :

$$\frac{}{\underline{k} \Downarrow k} \qquad {\scriptstyle k = k_1 + k_2}\ \frac{a_1 \Downarrow k_1 \qquad a_2 \Downarrow k_2}{a_1 \oplus a_2 \Downarrow k}\ ,$$

et on ajoute une règle pour le $\texttt{let}\, x = \ldots\ \texttt{in}\ \ldots$:

$$\frac{a_1 \Downarrow k_1 \qquad a_2\,[\underline{k_1}/x] \Downarrow k_1}{(\texttt{let}\, x = a_1\ \texttt{in}\ a_2) \Downarrow k_2.}$$

On note ici $a[\underline{k}/x]$ la substitution de $\underline{k}$ à la place de x dans l'expression a. Ceci sera défini après.

Attention : on n'a pas de règles de la forme

$$\frac{}{x \Downarrow \boxed{?}}\ ,$$

les variables sont censées disparaitre avant qu'on arrivent à elles.

Définition 5.1. Soit $a \in \mathsf{LEA}$. L'ensemble des *variables libres* d'une expression a noté $\mathcal{VL}(a)$, et est défini par induction sur a de la manière suivante :

- $\mathcal{V}\ell(\underline{k}) = \emptyset$;
- $\mathcal{V}\ell(x) = \{x\}$;
- $\mathcal{V}\ell(a_1 \oplus a_2) = \mathcal{V}\ell(a_1) \cup \mathcal{V}\ell(a_2)$;
- $\mathcal{V}\ell(\texttt{let}\, x = a_1 \;\texttt{in}\; a_2) = \mathcal{V}\ell(a_1) \cup (\mathcal{V}\ell(a_2) \setminus \{x\})$.

Exemple 5.2.

$$\mathcal{V}\ell(\texttt{let}\, x = \underline{3} \;\texttt{in}\; \texttt{let}\, y = x \oplus \underline{2} \;\texttt{in}\; y \oplus (z \oplus \underline{15})) = \{z\}.$$

Définition 5.2. Une expression $a \in \mathsf{LEA}$ est *close* si $\mathcal{V}\ell(a) = \emptyset$. On note $\mathsf{LEA}_0 \subseteq \mathsf{LEA}$ l'ensemble des expressions arithmétiques de closes.

Définition 5.3. Soient $a \in \mathsf{LEA}$, $x \in \mathcal{V}$ et $k \in \mathbb{Z}$. On définit par induction sur a (*quatre cas*) le résultat de la *substitution* de x par $\underline{k}$ dans a, noté $a[\underline{k}/x]$ de la manière suivante :

- $\underline{k}'[\underline{k}/x] = \underline{k}'$;
- $(a_1 \oplus a_2)[\underline{k}/x] = (a_1[\underline{k}/x]) \oplus (a_2[\underline{k}/x])$;
- $y[\underline{k}/x] = \begin{cases} \underline{k} & \text{si } x = y \\ y & \text{si } x \neq y \end{cases}$;
- $(\texttt{let}\, y = a_1 \;\texttt{in}\; a_2)[\underline{k}/x] = \begin{cases} \texttt{let}\, y = a_1[\underline{k}/x] \;\texttt{in}\; a_2 & \text{si } x = y \\ \texttt{let}\, y = a_1[\underline{k}/x] \;\texttt{in}\; a_2[\underline{k}/x] & \text{si } x \neq y. \end{cases}$

5.2 Sémantique à petits pas sur LEA.

On définit la relation $\to \subseteq \mathsf{LEA} * \mathsf{LEA}$ inductivement par :

$$\text{[}k = k_1 + k_2\text{]}\ \frac{}{\underline{k}_1 \oplus \underline{k}_2 \to \underline{k}}\ \mathscr{A},$$

1. On surcharge encore les notations.

$$\frac{a_2 \to a_2'}{a_1 \oplus a_2 \to a_1 \oplus a_2'}\ \mathscr{C}_{\mathrm{d}} \quad \text{et} \quad \frac{a_1 \to a_1'}{a_1 \oplus \underline{k} \to a_1' \oplus \underline{k}}\ \mathscr{C}_{\mathrm{g}},$$

puis les nouvelles règles pour le $\texttt{let}\, x = \ldots\ \texttt{in}\ \ldots$:

$$\frac{a_1 \to a_1'}{\texttt{let}\, x = a_1\ \texttt{in}\ a_2 \to \texttt{let}\, x = a_1'\ \texttt{in}\ a_2}\ \mathscr{C}_{\mathrm{l}}$$

$$\frac{}{\texttt{let}\, x = \underline{k}\ \texttt{in}\ a \to a[\underline{k}/x].}$$

On peut démontrer l'équivalence des sémantiques à grands pas et à petits pas.

5.3 Sémantique contextuelle pour LEA.

On définit les contextes d'évaluations par la grammaire suivante :

$$\begin{aligned} E ::= {} & [\,] \\ & \mid a \oplus E \\ & \mid E \oplus \underline{k} \\ & \mid \texttt{let}\, x = E\ \texttt{in}\ a. \end{aligned}$$

On définit *deux* relations $\mapsto_{\mathrm{a}}$ et $\mapsto$ par les règles :

$$^{k = k_1 + k_2}\ \frac{}{\underline{k}_1 \oplus \underline{k}_2 \mapsto_{\mathrm{a}} k_2} \qquad \frac{}{\texttt{let}\, x = \underline{k}\ \texttt{in}\ a \mapsto_{\mathrm{a}} a[\underline{k}/x]},$$

et

$$\frac{a \mapsto_{\mathrm{a}} a'}{E[a] \mapsto E[a']}.$$

5.4 Sémantique sur LEA avec environnement.

Définition 5.4. Soient A et B deux ensembles. Un *dictionnaire* sur (A, B) est une fonction partielle à domaine fini de A dans B.

Si D est un dictionnaire sur (A, B), on note $D(x) = y$ lorsque D associe $y \in B$ à $x \in A$.

Le domaine d'un dictionnaire D est

$$\mathrm{dom}(D) = \{x \in A \mid \exists y \in B, D(x) = y\}.$$

On note $\emptyset$ le dictionnaire vide.

Pour un dictionnaire D sur (A, B), deux éléments $x \in A$ et $y \in B$, on note $D[x \mapsto y]$ est le dictionnaire D' défini par

- $D'(x) = y$;
- $D'(z) = D(z)$ pour $z \in \mathrm{dom}(D)$ tel que $z \neq x$.

On ne s'intéresse pas à la construction d'un tel type de donné, mais juste son utilisation.

On se donne un ensemble Env d'*environnements* notés $\mathscr{E}, \mathscr{E}', \ldots$ qui sont des dictionnaires sur $(\mathscr{V}, \mathbb{Z})$.

5.4.1 Sémantique à grands pas sur LEA avec environnements.

On définit la relation $\Downarrow \subseteq \mathsf{LEA} * \mathrm{Env} * \mathbb{Z}$, noté $a, \mathscr{E} \Downarrow k$ (« a s'évalue en k dans $\mathscr{E}$ ») défini par

$$\frac{}{\underline{k}, \mathscr{E} \Downarrow k} \qquad \frac{a_1, \mathscr{E} \Downarrow k_1 \qquad a_2, \mathscr{E} \Downarrow k_2}{a_1 \oplus a_2, \mathscr{E} \Downarrow k}\ {\scriptstyle k = k_1 + k_2} \qquad \frac{}{x, \mathscr{E} \Downarrow k,}\ {\scriptstyle \mathscr{E}(x) = k}$$

$$\frac{a_1, \mathscr{E} \Downarrow k_1 \qquad a_2, \mathscr{E}[x \mapsto k_1] \Downarrow k_2}{\texttt{let}\, x = a_1 \ \texttt{in}\ a_2 \Downarrow k_2}\ .$$

Remarque 5.1.
- Dans cette définition, on n'a pas de substitutions (c'est donc plus facile à calculer).
- Si $\mathscr{V}\ell(a) \subseteq \mathrm{dom}(\mathscr{E})$, alors il existe $k \in \mathbb{Z}$ tel que $a, \mathscr{E} \Downarrow k$.
- On a $a \Downarrow k$ (sans environnement) si, et seulement si $a, \emptyset \Downarrow k$ (avec environnement).

Pour les petits pas avec environnements, c'est un peu plus compliqué...On verra ça en TD. (Écraser les valeurs dans un dictionnaire, ça peut être problématique avec les petits pas.)

6 Un petit langage fonctionnel : FUN.

Sommaire.

On se rapproche de notre but final en considérant un petit langage fonctionnel, nommé FUN.

On se donne l'ensemble des entiers relatifs $\mathbb{Z}$ et un ensemble infini de variables $\mathscr{V}$. L'ensemble des expressions de FUN, notées e, e' ou e_i, est défini par la grammaire suivante :

$$e ::= k \mid e_1 + e_2 \mid \underbrace{\texttt{fun}\, x \to e}_{\text{Fonction / Abstraction}} \mid \overbrace{e_1\ e_2}^{\text{Application}} \mid x.$$

Note 6.1. On simplifie la notation par rapport à EA ou LEA : on ne souligne plus les entiers, on n'entoure plus les plus.

On notera de plus $e_1\ e_2\ e_3$ pour $(e_1\ e_2)\ e_3$. Aussi, l'expression $\texttt{fun}\, x\ y \to e$ représentera l'expression $\texttt{fun}\, x \to (\texttt{fun}\, y \to e)$. On

n'a pas le droit à plusieurs arguments pour une fonction, mais on applique la curryfication.

6.1 Sémantique opérationnelle « informellement ».

Exemple 6.1. Comment s'évalue $(\texttt{fun}\, x \to x + x)(7 + 7)$?

- ▷ D'une part, $7 + 7$ s'évalue en 14.
- ▷ D'autre part, $(\texttt{fun}\, x \to x + x)$ s'évalue en elle même.
- ▷ On procède à une substitution de $(x + x)[^{14}/_x]$ qui s'évalue en 28.

Exemple 6.2. Comment s'évalue l'expression

$$(\overbrace{(\texttt{fun}\, f \to \underbrace{(\texttt{fun}\, x \to x + (f\ x))}_{B})}^{A}\ \underbrace{(\texttt{fun}\, y \to y + y)}_{C})\ 7\ ?$$

On commence par évaluer A et C qui s'évaluent en A et C respectivement. On continue en calculant la substitution

$$(\texttt{fun}\, x \to x + (f\ x))[^{\texttt{fun}\, y \to y + y}/_f],$$

ce qui donne

$$(\texttt{fun}\, x \to x + ((\texttt{fun}\, y \to y + y)\ x)).$$

Là, on **ne simplifie pas**, car c'est du code *dans* une fonction. On calcule ensuite la substitution

$$(x + ((\texttt{fun}\, y \to y + y)\ x))[^7/_x],$$

ce qui donne

$$7 + ((\texttt{fun}\, y \to y + y)\ 7).$$

On termine par la substitution

$$(y + y)[^7/_y] = 7 + 7.$$

On conclut que l'expression originelle s'évalue en 21.

Remarque 6.1. Dans FUN, le résultat d'un calcul (qu'on appellera *valeur*) n'est plus forcément un entier, ça peut aussi être une fonction.

L'ensemble des valeurs, notées v, est défini par la grammaire

$$v ::= k \mid \texttt{fun}\, x \to e.$$

Les fonctions sont des valeurs ! Et, le « contenu » la fonction n'est pas forcément une valeur.

On peut remarquer que l'ensemble des valeurs est un sous-ensemble des expressions de FUN.

6.2 Sémantique opérationnelle de FUN (version 1).

Définition 6.1. On définit l'ensemble des *variables libres* $\mathcal{VL}(e)$ d'une expression e par (on a 5 cas) :

- $\mathcal{VL}(x) = \{x\}$;
- $\mathcal{VL}(k) = \emptyset$;
- $\mathcal{VL}(e_1 + e_2) = \mathcal{VL}(e_1) \cup \mathcal{VL}(e_2)$;
- $\mathcal{VL}(e_1\ e_2) = \mathcal{VL}(e_1) \cup \mathcal{VL}(e_2)$;
- $\mathcal{VL}(\texttt{fun}\, x \to e) = \mathcal{VL}(e) \setminus \{x\}$. [1]

On dit que e est *close* si $\mathcal{VL}(e) = \emptyset$.

1. L'expression $\texttt{fun}\, x \to e$ est un *lieur* : x est liée dans e.

Définition 6.2. Pour $e \in \mathsf{FUN}$, $x \in \mathscr{V}$ et v une valeur **close**, on définit la *substitution* $e[v/x]$ de x par v dans e par :

- $k[v/x] = k$;
- $y[v/x] = \begin{cases} v & \text{si } x = y \\ y & \text{si } x \neq y \end{cases}$;
- $(\texttt{fun}\, y \to e)[v/x] = \begin{cases} \texttt{fun}\, y \to e & \text{si } x = y \\ \texttt{fun}\, y \to e[e/x] & \text{si } x \neq y; \end{cases}$
- $(e_1 + e_2)[v/x] = (e_1[v/x]) + (e_2[v/x])$;
- $(e_1\ e_2)[v/x] = (e_1[v/x])\ (e_2[v/x])$.

6.2.1 Grands pas pour FUN.

On définit la relation $\Downarrow$ sur couples (expression, valeur) par :

$$_{k = k_1 + k_2}\ \frac{e_1 \Downarrow k_1 \qquad e_2 \Downarrow k_2}{e_1 + e_2 \Downarrow k} \qquad \frac{}{v \Downarrow v}$$

$$\frac{e_1 \Downarrow \texttt{fun}\, x \to e \qquad e_2 \Downarrow v_2 \qquad e[v_2/x] \Downarrow v}{e_1\ e_2 \Downarrow v.}$$

Remarque 6.2. Certaines expressions ne s'évaluent pas :

$$x \not\Downarrow \qquad \text{et} \qquad z + (\texttt{fun}\, x \to x) \not\Downarrow$$

par exemple.

6.2.2 Petits pas pour FUN.

On définit la relation $\to \subseteq \mathsf{FUN} * \mathsf{FUN}$ par :

$$_{k = k_1 + k_2}\ \frac{}{k_1 + k_2 \to k}\ \mathscr{R}_{\text{pk}} \qquad \frac{}{(\texttt{fun}\, x \to e)\ v \to e[v/x]}\ \mathscr{R}_{\beta}$$

$$\frac{e_2 \to e_2'}{e_1 + e_2 \to e_1 + e_2'}\ \mathscr{R}_{\text{pd}} \qquad \frac{e_1 \to e_1'}{e_1 + k \to e_1' + k}\ \mathscr{R}_{\text{pg}}$$

$$\frac{e_2 \to e_2'}{e_1\ e_2 \to e_1\ e_2'}\ \mathscr{R}_{\text{ad}} \qquad \frac{e_1 \to e_1'}{e_1\ v \to e_1'\ v}\ \mathscr{R}_{\text{ag}}.$$

Remarque 6.3. Il existe des expressions que l'on ne peut pas réduire :

1. $k \not\to$;
2. $(\texttt{fun}\, x \texttt{ -> } x) \not\to$;
3. $e_1 + (\texttt{fun}\, x \texttt{ -> } x) \not\to$;
4. $3\ (5+7) \to 3\ 12 \not\to$.

Dans les cas 1. et 2., c'est cohérent : on ne peut pas réduire des valeurs.

Lemme 6.1. On a

$$e \Downarrow v \qquad \text{si, et seulement si,} \qquad e \to^\star v.$$

Remarque 6.4. Soit $e_0 = (\texttt{fun}\, x \texttt{ -> } x\ x)\ (\texttt{fun}\, x \texttt{ -> } x\ x)$. On remarque que $e_0 \to e_0$.

En FUN, il y a des divergences : il existe $(e_n)_{n\in\mathbb{N}}$ telle que l'on ait $e_n \to e_{n+1}$.

La fonction[2] définie par $\Downarrow$ est donc partielle.

Remarque 6.5 (Problème avec la substitution)**.** On a la chaîne de réductions :

$$\begin{aligned} & ((\texttt{fun}\, y \texttt{ -> } (\texttt{fun}\, x \texttt{ -> } x + y))\ (x+7))\ 5 \\ (\star) \qquad & \to (\texttt{fun}\, x \texttt{ -> } x + (x+7))\ 5 \\ & \to 5 + (5+7) \\ & \to^\star 17. \end{aligned}$$

Attention ! Ici, on a triché : on a substitué avec l'expression $x+7$

mais ce n'est pas une valeur (dans la réduction $(\star)$) !

Mais, on a la chaîne de réductions

$$\begin{aligned}&(\texttt{fun}\, f \to (\texttt{fun}\, x \to (f\ 3) + x))\ (\texttt{fun}\, t \to x + 7)\ 5\\ \to\ &(\texttt{fun}\, x \to ((\texttt{fun}\, t \to x + 7)\ 3) + x)\ 5\\ \to\ &(\texttt{fun}\, x \to ((\texttt{fun}\, t \to x + 7)\ 3) + x)\ 5.\end{aligned}$$

Et là, c'est le drame, on a **capturé la variable libre**. D'où l'hypothèse de v close dans la substitution.

Remarque 6.6. Les relations $\Downarrow$ et $\to$ sont définies sur des expressions **closes**. Et on a même $\to \subseteq \mathsf{FUN}_0 * \mathsf{FUN}_0$.[3]

Lemme 6.2. ▷ Si v est close et si $x \notin \mathscr{V}\ell(e)$ alors $e[^v/_x] = e$.
▷ Si v est close, $\mathscr{V}\ell(e[^v/_x]) = \mathscr{V}\ell(e) \setminus \{x\}$. □

Lemme 6.3. Si $e \in \mathsf{FUN}_0$ et $e \to e'$ alors $e' \in \mathsf{FUN}_0$.

Preuve. Montrons que, quelles que soient e et e', on a : si $e \to e'$ alors $(e \in \mathsf{FUN}_0) \implies (e' \in \mathsf{FUN}_0)$ On procède par induction sur la relation $e \to e'$. Il y a 6 cas :

1. Pour $\mathscr{R}_\beta$, on suppose $(\texttt{fun}\, x \to e)\ v$ est close, alors
   ▷ $(\texttt{fun}\, x \to e)$ est close ;
   ▷ v est close.

   On sait donc que $\mathscr{V}\ell(e) \subseteq \{x\}$, d'où par le lemme précédent, $\mathscr{V}\ell(e[^v/_x]) = \emptyset$ et donc $e[^v/_x]$ est close.

2–6. Pour les autres cas, on procède de la même manière.

□

2. Pour indiquer cela, il faudrait démontrer que la relation $\Downarrow$ est déterministe.

3. Il faudrait ici justifier que la réduction d'une formule close est close. C'est ce que nous allons justifier.

Remarque 6.7. De même, si $e \Downarrow v$ où e est close, alors v est close.

Les relations $\Downarrow$ et $\to$ sont définies sur les expressions et les valeurs closes.

Définition 6.3 (Définition informelle de l'α-conversion)**.** On définit l'α-conversion, notée $e =_\alpha e'$: on a $\texttt{fun}\, x \to e =_\alpha \texttt{fun}\, y \to e'$ si, et seulement si, e' s'obtient en replaçant x par y dans e à condition que $y \notin \mathcal{Vl}(e)$. [4]

On étend $e =_\alpha e'$ à toutes les expressions : « on peut faire ça partout ».

Exemple 6.3 (*Les variables liées sont muettes.*)**.** On a :

$$\begin{aligned} \texttt{fun}\, x \to x + z &=_\alpha \texttt{fun}\, y \to y + z \\ &=_\alpha \texttt{fun}\, t \to t + z \\ &\neq_\alpha \texttt{fun}\, z \to z + z. \end{aligned}$$

L'intuition est, quand on a $\mathsf{fun}\ x \to e$ et qu'on a besoin de renommer la variable x, pour cela on prend $x' \notin \mathcal{Vl}(e)$.

"Lemme" 6.1. Si $E_0 \subseteq \mathcal{V}$ est un ensemble fini de variables, alors il existe $z \notin E_0$ et $e' \in \mathsf{FUN}$ tel que $\texttt{fun}\, x \to e =_\alpha \texttt{fun}\, z \to e'$. □

Remarque 6.8 (**Fondamental**)**.** En fait FUN désigne l'ensemble des expressions décrites par la grammaire initiale *quotientée* par α-conversion.

Remarque 6.9. On remarque que

$$(e =_\alpha e') \implies \mathcal{Vl}(e) = \mathcal{Vl}(e').$$

4. C'est une « variable fraîche ».

D'après le "lemme", on peut améliorer notre définition de la substitution.

Définition 6.4. Pour $e \in \mathsf{FUN}$, $x \in \mathscr{V}$ et v une valeur **close**, on définit la *substitution* $e[^v/_x]$ de x par v dans e par :

- $k[^v/_x] = k$;
- $y[^v/_x] = \begin{cases} v & \text{si } x = y \\ y & \text{si } x \neq y \end{cases}$;
- $(\texttt{fun}\, x \texttt{ → } e)[^v/_x] = (\texttt{fun}\, y \texttt{ → } e)[^v/_x]$ lorsque $x \neq y$;
- $(e_1 + e_2)[^v/_x] = (e_1[^v/_x]) + (e_2[^v/_x])$;
- $(e_1\ e_2)[^v/_x] = (e_1[^v/_x])\ (e_2[^v/_x])$.

6.3 Ajout des déclarations locales (FUN+`let`).

On ajoute les déclarations locales (comme pour $\mathsf{EA} \to \mathsf{LEA}$) à notre petit langage fonctionnel. Dans la grammaire des expressions de FUN, on ajoute :

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

Ceci implique d'ajouter quelques éléments aux différentes opérations sur les expressions définies ci-avant :

- on définit $\mathscr{V}\ell(\texttt{let}\, x \texttt{ = } e_1 \texttt{ in } e_2) = \mathscr{V}\ell(e_1) \cup (\mathscr{V}\ell(e_2) \setminus \{x\})$;
- on ne change pas les valeurs : une déclaration locale n'est pas une valeur ;
- on ajoute $\texttt{let}\, x \texttt{ = } e_1 \texttt{ in } e_2 =_\alpha \texttt{let}\, y \texttt{ = } e_1 \texttt{ in } e_2'$, où l'on remplace x par y dans e_2 pour obtenir e_2' ;
- pour la substitution, on pose lorsque $x \neq y$ (que l'on peut toujours supposer modulo α-conversion)

$$(\texttt{let}\, y \texttt{ = } e_1 \texttt{ in } e_2)[^v/_x] = (\texttt{let}\, y \texttt{ = } e_1[^v/_x] \texttt{ in } e_2[^v/_x]).$$

- pour la sémantique à grands pas, c'est comme pour LEA ;

- pour la sémantique à petits pas, on ajoute les deux règles :

$$\frac{}{\texttt{let}\, x = v \;\texttt{in}\; e_2 \to e_2[v/x]}\ \mathscr{R}_{\text{lv}}$$

et

$$\frac{e_1 \to e_1'}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \to \texttt{let}\ x = e_1'\ \texttt{in}\ e_2}\ \mathscr{R}_{\text{lg}}.$$

Attention ! On n'a pas de règle

$$\frac{e_2 \to e_2'}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \to \texttt{let}\ x = e_1\ \texttt{in}\ e_2'}\ \mathscr{R}_{\text{ld}},$$

on réduit d'abord l'expression e_1 jusqu'à une valeur, avant de passer à e_2.

Le langage que l'on construit s'appelle FUN + `let`.

6.3.1 Traduction de FUN + `let` vers FUN.

On définit une fonction qui, à toute expression de e dans FUN + `let` associe une expression notée $[\![e]\!]$ dans FUN (on supprime les expressions locales). L'expression $[\![e]\!]$ est définie par induction sur e. Il y a 6 cas :

- $[\![k]\!] = k$;
- $[\![x]\!] = x$;
- $[\![e_1 + e_2]\!] = [\![e]\!]_1 + [\![e]\!]_2$;
- $[\![e_1\ e_2]\!] = [\![e]\!]_1\ [\![e]\!]_2$;
- $[\![\texttt{fun}\, x \to e]\!] = \texttt{fun}\, x \to [\![e]\!]$;
- $[\![\texttt{let}\, x = e_1\ \texttt{in}\ e_2]\!] = (\texttt{fun}\, x \to [\![e_2]\!])\ [\![e_1]\!]$.

Lemme 6.4. Pour tout $e \in$ (FUN + `let`),

- $[\![e]\!]$ est une expression de FUN [5] ;
- on a $\mathscr{V}\ell([\![e]\!]) = \mathscr{V}\ell(e)$;
- $[\![e]\!]$ est une valeur *ssi* e est une valeur ;
- $[\![e[v/x]]\!] = [\![e]\!]\,[[\![v]\!]/x]$ [6]. □

Pour démontrer le lemme 6.4, on procède par induction sur e. C'est long et rébarbatif, mais la proposition ci-dessous est bien plus intéressante.

Proposition 6.1. Pour toutes expressions e, e' de $\mathsf{FUN} + \texttt{let}$, si on a la réduction $e \to_{\mathsf{FUN}+\texttt{let}} e'$ alors $[\![e]\!] \to_{\mathsf{FUN}} [\![e']\!]$.

Preuve. On procède par induction sur $e \to e'$ dans $\mathsf{FUN} + \texttt{let}$. Il y a 8 cas car il y a 8 règles d'inférences pour $\to$ dans $\mathsf{FUN} + \texttt{let}$.

- ▷ *Cas* $\mathscr{R}_{\text{lv}}$. Il faut montrer que $[\![\texttt{let}\, x = v\ \texttt{in}\ e_2]\!] \to_{\mathsf{FUN}} [\![e[v/x]]\!]$. Par définition, l'expression de droite vaut

$$(\texttt{fun}\, x \to [\![e]\!]_2)\ [\![v]\!] \xrightarrow{\mathscr{R}_\beta}_{\mathsf{FUN}} [\![e]\!]_2\, [[\![v]\!]/x],$$

  car $[\![v]\!]$ est une valeur par le lemme 6.4, ce qui justifie $\mathscr{R}_\beta$. De plus, encore par le lemme 6.4, on a l'égalité entre $[\![e]\!]_2\, [[\![v]\!]/x] = [\![e[v/x]]\!]$.
- ▷ *Cas* $\mathscr{R}_{\text{lg}}$. On sait que $e_1 \to e_1'$ et, par hypothèse d'induction, on a $[\![e_1]\!] \to [\![e_1']\!]$. Il faut montrer que

$$[\![\texttt{let}\, x = e_1\ \texttt{in}\ e_2]\!] \to [\![\texttt{let}\, x = e_1'\ \texttt{in}\ e_2]\!]\,.$$

  L'expression de droite vaut

$$(\texttt{fun}\, x \to [\![e_2]\!])\ [\![e_1]\!] \xrightarrow{\mathscr{R}_{\text{ad}}\ \&\ \text{hyp. ind.}} (\texttt{fun}\, x \to [\![e_2]\!])\ [\![e_1']\!]\,.$$

  Et, par définition de $[\![\cdot]\!]$, on a l'égalité :

$$[\![\texttt{let}\, x = e_1'\ \texttt{in}\ e_2]\!] = (\texttt{fun}\, x \to [\![e_2]\!])\ [\![e_1']\!]\,.$$

- ▷ Les autres cas sont laissées en exercice.

□

5. *i.e.* $[\![e]\!]$ n'a pas de déclarations locales
6. On le prouve par induction sur e, c'est une induction à 6 cas

Proposition 6.2. Si $[\![e]\!] \to [\![e']\!]$ alors $e \to e'$.

Preuve. La proposition ci-dessus est mal formulée pour être prouvée par induction, on la ré-écrit. On démontre, par induction sur la relation $f \to f'$ la propriété suivante :

> « quel que soit e, si $f = [\![e]\!]$ alors il existe e' une expression telle que $f' = [\![e']\!]$ et $e \to e'$ (dans FUN + `let`) »,

qu'on notera $\mathscr{P}(f, f')$.

Pour l'induction sur $f \to f'$, il y a 6 cas.

- ▷ *Cas de la règle* $\mathscr{R}_{\text{ad}}$. On suppose $f_2 \to f_2'$ et par hypothèse d'induction $\mathscr{P}(f_2, f_2')$. On doit montrer $\mathscr{P}(f_1\ f_2, f_1\ f_2')$. On suppose donc $[\![e]\!] = f_1\ f_2$. On a deux sous-cas.
  - *1er sous-cas.* On suppose $e = e_1\ e_2$ et $[\![e_1]\!] = f_1 = f_2$. Par hypothèse d'induction et puisque $[\![e_2]\!] = f_2$, il existe e_2' tel que $e_2 \to e_2'$ et $[\![e_2']\!] = f_2'$. De $e_2 \to e_2'$, on en déduit par $\mathscr{R}_{\text{ad}}$ que $e_1\ e_2 \to e_1\ e_2'$. On pose $e' = e_1\ e_2'$ et on a bien $[\![e']\!] = [\![e_1]\!]\ [\![e_2']\!]$.
  - *2ème sous-cas.* On suppose $e = \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$. Alors,
    $$[\![e]\!] = \underbrace{(\texttt{fun}\ x \texttt{ -> } [\![e_2]\!])}_{f_1}\ \underbrace{[\![e_1]\!]}_{f_2}.$$
    Par hypothèse d'induction, il existe e_1' tel que $e_1 \to e_1'$ et $[\![e_1']\!] = f_2'$. Posons $e' = (\texttt{let}\ x\ \texttt{=}\ e_1'\ \texttt{in}\ e_2)$. On doit vérifier $[\![e]\!] \to [\![e']\!]$ ce qui est vrai par $\mathscr{R}_{\text{ad}}$ et que $[\![e']\!] = f_1\ f_2'$, ce qui est vrai par définition.
- ▷ *Cas de la règle* $\mathscr{R}_{\text{ag}}$. On suppose $f_1 \to f_1'$ et l'hypothèse d'induction $\mathscr{P}(f_1, f_1')$. On doit vérifier que $\mathscr{P}(f_1\ v, f_1'\ v)$. On suppose $[\![e]\!] = f_1\ v$ et on a deux sous-cas.
  - *1er sous-cas.* On suppose $e = e_1\ e_2$ et alors $[\![e]\!] = [\![e_1]\!]\ [\![e_2]\!]$ par le lemme 6.4 et parce que e_2 est une

valeur (car $[\![e_2]\!] = v$). On raisonne comme pour la règle $\mathscr{R}_{\text{ad}}$ dans le premier sous-cas, en appliquant $\mathscr{R}_{\text{ag}}$.

– *2nd sous-cas.* On suppose $e = (\texttt{let}\, x = e_1 \;\texttt{in}\; e_2)$ alors

$$[\![e]\!] = \underbrace{\texttt{fun}\, x \to [\![e_2]\!]}_{f_1}\ \underbrace{[\![e_1]\!]}_{f_2}.$$

On vérifie aisément ce que l'on doit montrer.

▷ Les autres cas se font de la même manière (attention à $\mathscr{R}_\beta$).

□

7 Typage en FUN.

Sommaire.

7.1 Définition du système de types.

L'ensemble Typ des types, notés $\tau, \tau_1, \tau', \ldots$, est définit par la grammaire suivante :

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

Note 7.1. Attention ! Le type $\tau_1 \to \tau_2 \to \tau_3$ n'est pas égal au type $(\tau_1 \to \tau_2) \to \tau_3$. En effet, dans le premier cas, c'est une fonction qui renvoie une fonction ; et, dans le second cas, c'est une fonction qui prend une fonction.

Définition 7.1. Un *environnent de typage*, noté $\Gamma, \Gamma_1, \Gamma', \ldots$, est un dictionnaire sur $(\mathcal{V}, \mathsf{Typ})$, où Typ est l'ensemble des types.

Une *hypothèse de typage*, notée $x : \tau$, est un couple (x, τ).

On note $\Gamma, x : \tau$ l'extension de Γ avec l'hypothèse de typage $x : \tau$ qui n'est définie que lorsque $x \notin \operatorname{dom} \Gamma$. [1]

Remarque 7.1. On peut voir/implémenter Γ comme des listes finies de couples (x, τ).

Définition 7.2. La *relation de typage*, notée $\Gamma \vdash e : \tau$ (« sous les hypothèses Γ, l'expression e a le type τ ») est définie par les règles d'inférences suivantes.

$$\frac{}{\Gamma \vdash k : \texttt{int}}\ \mathcal{T}_{\text{k}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \mathcal{T}_{\text{v}} \qquad \frac{\Gamma, x : \tau_1 \text{ [2]} \vdash e_2}{\Gamma \vdash \texttt{fun}\, x \to e : \tau_1 \to \tau_2}\ \mathcal{T}_{\text{f}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}\ \mathcal{T}_{\text{p}} \qquad \frac{\Gamma \vdash e : \tau_1 \to \tau_2 \qquad \Gamma \vdash e' : \tau_1}{\Gamma \vdash e\ e' : \tau_2}\ \mathcal{T}_{\text{a}}.$$

Remarque 7.2. Pour l'instant, on parle uniquement d'expressions et pas du tout de valeurs ou de sémantique opérationnelle.

Remarque 7.3.
1. On dit que e est *typable* s'il existe Γ et τ tel que $\Gamma \vdash e : \tau$.
2. Il y a une règle de typage par construction du langage des expressions.

Exemple 7.1. 1. L'expression $\texttt{fun}\, x \to x$ est particulière : on

1. La définition de $\Gamma, x : \tau$ est « comme on le pense ».
2. On peut toujours étendre Γ ainsi, modulo α-conversion.

peut la typer avec $\tau \to \tau$ quel que soit τ. Par exemple,

$$\dfrac{\dfrac{}{x : \texttt{int} \vdash x : \texttt{int}}\ \mathcal{T}_{\text{v}}}{\emptyset \vdash \texttt{fun}\, x \to x : \texttt{int} \to \texttt{int}}\ \mathcal{T}_{\text{f}}.$$

On aurait pu faire de même avec le type $(\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$.

2. Quel est le type de $\texttt{fun}\, g \to g\ (g\ 7)$?

$$\dfrac{\dfrac{\dfrac{}{g : \texttt{int} \to \texttt{int} \vdash g : \texttt{int} \to \texttt{int}}\ \mathcal{T}_{\text{v}} \quad \dfrac{\dfrac{}{\Gamma \vdash g : \texttt{int} \to \texttt{int}}\ \mathcal{T}_{\text{v}} \quad \dfrac{}{\Gamma \vdash 7 : \texttt{int}}\ \mathcal{T}_{\text{k}}}{g : \texttt{int} \to \texttt{int} \vdash g\ 7 : \texttt{int}}\ \mathcal{T}_{\text{p}}}{g : \texttt{int} \to \texttt{int} \vdash g\ (g\ 7)}\ \mathcal{T}_{\text{a}}}{\emptyset \vdash \texttt{fun}\, g \to g\ (g\ 7) : (\texttt{int} \to \texttt{int}) \to \texttt{int}}\ \mathcal{T}_{\text{f}}.$$

7.2 Propriétés du système de types.

Lemme 7.1. ▷ Si $\Gamma \vdash e : \tau$ alors $\mathcal{V}\ell(e) \subseteq \text{dom}(\Gamma)$.

▷ *Affaiblissement.* Si $\Gamma \vdash e : \tau$ alors

$$\forall x \notin \text{dom}(\Gamma),\ \forall \tau_0, \quad \Gamma, x : \tau_0 \vdash e : \tau.$$

▷ *Renforcement.* Si $\Gamma, x : \tau_0 \vdash e : \tau$, et si $x \notin \mathcal{V}\ell(e)$ alors on a le typage $\Gamma \vdash e : \tau$.

Preuve. Par induction sur la relation de typage (5 cas). □

7.2.1 Propriété de progrès.

Lemme 7.2. 1. Si $\emptyset \vdash e : \texttt{int}$ et $e \not\to$ alors, il existe $k \in \mathbb{Z}$ tel que $e = k$.

2. Si $\emptyset \vdash e : \tau_1 \to \tau_2$ et $e \not\to$ alors il existe x et e_0 tels que l'on ait $e = \texttt{fun}\, x \to e_0$.

Preuve. Vu en TD. □

Proposition 7.1 (Propriété de progrès). Si $\emptyset \vdash e : \tau$ alors on a la disjonction :

1. soit e est une valeur ;
2. soit il existe e' telle que $e \to e'$.

Remarque 7.4.

- ▷ Si $\emptyset \vdash e_1\ e_2 : \tau$ alors il existe e' tel que $e_1\ e_2 \to e'$.
- ▷ Si $\emptyset \vdash e_1 + e_2 : \tau$ alors il existe e' tel que $e_1 + e_2 \to e'$.

Remarque 7.5. Par le typage, on a exclu les expressions bloquées car « mal formées » (*e.g.* 3 2 ou $3 + (\texttt{fun}\, x \to x)$).

7.2.2 Propriété de préservation.

Cette propriété a plusieurs noms : préservation du typage, réduction assujettie, *subject reduction*.

Lemme 7.3 (typage et substitution). Si l'on a le typage $\emptyset \vdash v : \tau_0$ et $\Gamma, x : \tau_0 \vdash e : \tau$ alors on a $\Gamma \vdash e[v/x] : \tau$

Preuve. On prouve cette propriété par induction sur e. Il y a 5 cas.

- ▷ *Cas* $e = y$. On a deux sous-cas.
  - *1er sous-cas* $x \neq y$. Dans ce cas, $e[v/x] = y$. Il faut montrer $\Gamma \vdash y : \tau$ sachant que $\Gamma, x : \tau_0 \vdash y : \tau$. On applique le lemme de renforcement.
  - *2nd sous-cas* $x = y$. Dans ce cas, $e[v/x] = v$. Il faut montrer que $\Gamma \vdash v : \tau$. Or, on sait que $\Gamma, x : \tau_0 \vdash x : \tau$ (d'où $\tau = \tau_0$) et $\emptyset \vdash v : \tau_0$. On conclut par affaiblissement.
- ▷ Les autres cas sont en exercice.

□

Proposition 7.2 (Préservation du typage)**.** Si $\emptyset \vdash e : \tau$, et $e \to e'$ alors $\emptyset \vdash e' : \tau$.

Preuve. On montre la propriété par induction sur $\emptyset \vdash e : \tau$. Il y a 5 cas.

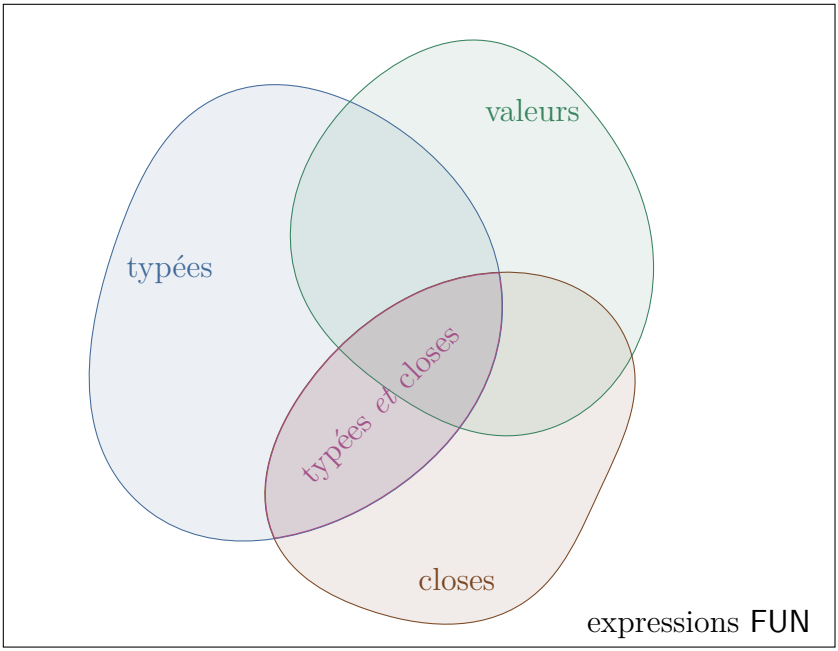
- ▷ *Cas* $\mathcal{T}_\text{v}$. C'est absurde ! (On n'a pas $\emptyset \vdash x : \tau$.)
- ▷ *Cas* $\mathcal{T}_\text{f}$. Si $(\texttt{fun}\, x \texttt{ → } e) \to e'$ alors ... On peut conclure immédiatement car les fonctions sont des valeurs, elles ne se réduisent donc pas.
- ▷ *Cas* $\mathcal{T}_\text{k}$. C'est le même raisonnement.
- ▷ *Cas* $\mathcal{T}_\text{a}$. On a $e = e_1\ e_2$. On sait qu'il existe τ_0 un type tel que $\emptyset \vdash e_1 : \tau_0 \to \tau$ (H_1) et $\emptyset \vdash e_2 : \tau_0$ (H_2). On a également les hypothèses d'induction :
  - (H_1') : si $e_1 \to e_1'$ alors $\emptyset \vdash e_1' : \tau_0 \to \tau$;
  - (H_2') : si $e_2 \to e_2'$ alors $\emptyset \vdash e_2' : \tau_0$.

  On doit montrer que si $e_1\ e_2 \to e'$ alors $\emptyset \vdash e' : \tau$. Supposons que $e_1\ e_2 \to e'$, il y a 3 sous-cas.
  - *Sous-cas* $\mathcal{R}_\text{ad}$. Cela veut dire que $e_2 \to e_2'$ et $e' = e_1\ e_2'$. On conclut $\emptyset \vdash e_1\ e_2' : \tau$ par (H_2') et (H_1).
  - *Sous-cas* $\mathcal{R}_\text{ag}$. Cela veut dire que $e_1 \to e_1'$ et $e' = e_1'\ e_2$. On conclut $\emptyset \vdash e_1'\ e_2 : \tau$ par (H_1') et (H_2).
  - *Sous-cas* $\mathcal{R}_\beta$. On a $e_1 = \texttt{fun}\, x \texttt{ → } e_0$, $e_2 = v$ et finalement $e' = e_0[v/x]$. On doit montrer $\emptyset \vdash e_0[v/x] : \tau$. De plus, (H_1) s'énonce par $\emptyset \vdash \texttt{fun}\, x \texttt{ → } e_0 : \tau_0 \to \tau$. Nécessairement (c'est un « `inversion` » en Rocq), cela provient de $x : \tau_0 \vdash e_0 : \tau$. On en conclut par le lemme de substitution.
- ▷ *Cas* $\mathcal{T}_\text{p}$. Laissé en exercice.

□

Remarque 7.6. Avec les propriétés de progrès et préservation implique qu'il n'y a pas de « mauvaises surprises » à l'exécution. On

a, en un sens, nettoyé le langage FUN.



C'est la considération d'un langage *statiquement typé*. On aime savoir qu'OCaml ou Rust ont, pour la sémantique et le système de types, une propriété de progrès et de préservation.

Exercice 7.1. Trouver e et e' deux expressions telles que $\emptyset : e' : \tau$ et $e \to e'$ mais que l'on ait pas $\emptyset \vdash e : \tau$.

Solution. Il suffit de trouver une valeur non typable e_1, par exemple $\texttt{fun}\, x \to (x\ x)$ ou $\texttt{fun}\, x \to (19\ 27)$, puis de considérer

$$e = (\texttt{fun}\, x \to 3)\ e_1 \to 3.$$

Or, 3 est typable mais e non.

7.3 Questions en lien avec la relation de typage.

- ▷ *Typabilité.* Pour e donné, existe-t-il Γ, τ tels que $\Gamma \vdash e : \tau$?
- ▷ *Vérification/Inférence de types.* Pour Γ et e donnés, existe-t-il τ tel que l'on ait $\Gamma \vdash e : \tau$? (▷ OCaml)
- ▷ *Habitation.* Pour τ donné, existe-t-il e tel que $\emptyset \vdash e : \tau$? (▷ Rocq[3])

7.4 Inférence de types.

7.4.1 Typage et contraintes.

Exemple 7.2. Dans une version étendue de FUN (on se rapproche plus au OCaml), si l'on considère le programme :

```
let rec f x g=
  ...g x ...
  ...if g f then ... else ...
  ...let h = x 7 in ...
```

On remarque que

- ▷ x et f ont le même type ;
- ▷ g a un type $? \to \texttt{bool}$;
- ▷ x a un type $\texttt{int} \to ?$.

On doit donc lire le programme, et « prendre des notes ». Ces « notes » sont des contraintes que doivent vérifier le programme.

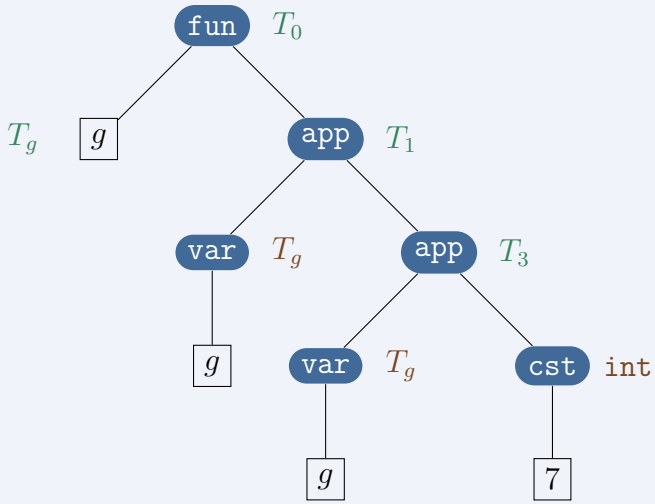
Exemple 7.3. On souhaite déterminer le type τ tel que

$$\emptyset \vdash \texttt{fun}\, g \to g\ (g\ 7) : \tau.$$

3. On peut voir une preuve d'un théorème en Rocq comme fournir une preuve qu'il existe une expression e avec type τ.

(On sait que $\tau = (\texttt{int} \to \texttt{int}) \to \texttt{int}$.)

On construit l'arbre de l'expression (l'AST) :



On procède en plusieurs étapes :

1. On ajoute des inconnues de types T_1, T_2 T_3, *etc* (en vert).
2. On écrit des contraintes faisant intervenir les T_i (en orange/marron).

$$T_0 = T_g \to T_1$$
$$T_g = T_2 \to T_1$$
$$T_g = \texttt{int} \to T_1.$$

3. On résout les contraintes pour obtenir

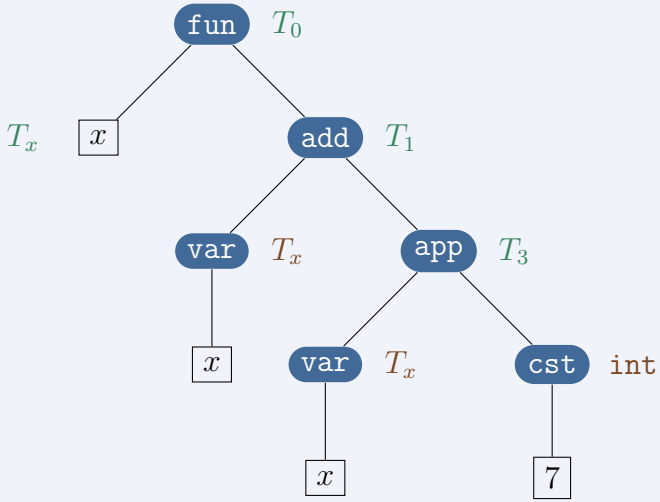
$$T_0 = (\texttt{int} \to \texttt{int}) \to \texttt{int}.$$

Exemple 7.4 (Cas limites). ▷ L'expression $\texttt{fun}\, x \to 7$ admet une infinité de types $(T_x \to \texttt{int})$.

▷ L'expression $(\texttt{fun}\, x \to 7)\ (\texttt{fun}\, z \to z)$ a toujours le type `int` mais admet une infinité de dérivations.

Exemple 7.5 (Et quand ça ne marche pas ?)**.** On essaie d'inférer le type de l'expression

$$\texttt{fun}\, x \to x + (x\ 2).$$



Les contraintes sont :

$$T_0 = T_x \to T_1$$
$$T_1 = T_x = T_2 = \texttt{int}$$
$$T_x = \texttt{int} \to T_2.$$

Catastrophe ! On ne peut pas résoudre ce système de contraintes (on ne peut pas avoir $T_x = \texttt{int}$ et $T_x = \texttt{int} \to T_2$ en même temps). L'expression n'est donc pas typable.

Définition 7.3. ▷ On se donne un ensemble infini IType d'*inconnues de type*, notées T, T_1, T', *etc.*

- On définit les *types étendus*, notés $\hat{\tau}$, par la grammaire :

$$\hat{\tau} ::= \texttt{int} \mid \hat{\tau}_1 \to \hat{\tau}_2 \mid T.$$

- L'ensemble des types (*resp.* étendus) est noté Typ (*resp.* $\widehat{\mathsf{Typ}}$).
- Les environnement de types étendus sont notés $\widehat{\Gamma}$.
- Ainsi défini, tout τ est un $\hat{\tau}$, tout Γ est un $\widehat{\Gamma}$.
- Un $\hat{\tau}$ est dit *constant* s'il ne contient pas d'inconnue de type (*i.e.* si c'est un τ).

Définition 7.4. Une *contrainte de typage* est une paire de types étendus[4], notée $\hat{\tau}_1 \stackrel{?}{=} \hat{\tau}_2$, ou parfois $\hat{\tau}_1 = \hat{\tau}_2$.

On se donne $e \in \mathsf{FUN}$. On suppose que toutes les variables liées de e sont :

- distinctes deux à deux ;
- différentes de toutes les variables libres de e.

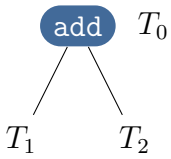
On se donne $\widehat{\Gamma}$ tel que $\mathscr{V}\ell(e) \subseteq \mathrm{dom}(\widehat{\Gamma})$. On choisit $T \in \mathrm{IType}$.

On définit un ensemble de contraintes, notée $\mathsf{CT}(e, \widehat{\Gamma}, T)$ par induction sur e, il y a 5 cas :

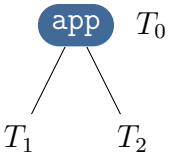
- $\mathsf{CT}(e_1 + e_2, \widehat{\Gamma}, T) = \mathsf{CT}(e_1, \widehat{\Gamma}, T_1) \cup \mathsf{CT}(e_2, \widehat{\Gamma}, T_2) \cup \{T_1 \stackrel{?}{=} \texttt{int}, T_2 \stackrel{?}{=} \texttt{int}, T \stackrel{?}{=} \texttt{int}\}$
- $\mathsf{CT}(e_1\ e_2, \widehat{\Gamma}, T) = \mathsf{CT}(e_1, \widehat{\Gamma}, T_1) \cup \mathsf{CT}(e_2, \widehat{\Gamma}, T_2) \cup \{T_1 \stackrel{?}{=} T_2 \to T\}$
- $\mathsf{CT}(x, \widehat{\Gamma}, T) = \{T \stackrel{?}{=} \widehat{\Gamma}(x)\}$
- $\mathsf{CT}(k, \widehat{\Gamma}, T) = \{T \stackrel{?}{=} \texttt{int}\}$
- $\mathsf{CT}(\texttt{fun}\, x \to e, \widehat{\Gamma}, T) = \mathsf{CT}(e, (\widehat{\Gamma}, x : T_x), T_2) \cup \{T \stackrel{?}{=} T_1 \to T_2\}$

où les variables T_1, T_2, T_x sont *fraîches* (on notera par la suite $\mathit{И}$ T_1, T_2, T_x).

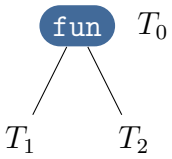
Remarque 7.7. On peut résumer les cas « plus », « application » et « abstraction ».



$T_0 = T_1 = T_2 = \texttt{int}$



$T_1 = T_2 \to T_0$



$T_0 = T_1 \to T_2$

Définition 7.5. Soit C un ensemble de contraintes de typage. On note $\mathrm{Supp}(C)$, le *support* de C, l'ensemble des inconnues de type mentionnées dans C.

Une *solution* σ de C est un dictionnaire sur $(\mathrm{ITyp}, \widehat{\mathsf{Typ}})$ tel que $\mathrm{dom}(\sigma) \supseteq \mathrm{Supp}(C)$ et que σ égalise toutes les contraintes de C.

Pour $(\hat{\tau}_1 \stackrel{?}{=} \hat{\tau}_2) \in C$, on dit que σ égalise $\hat{\tau}_1 \stackrel{?}{=} \hat{\tau}_2$ signifie que $\sigma(\hat{\tau}_1)$ et $\sigma(\hat{\tau}_2)$ sont le même type étendu.

Il reste à définir $\sigma(\hat{\tau})$, le résultat de l'application de σ à $\hat{\tau}$, par induction sur $\hat{\tau}$, il y a trois cas :

- $\sigma(\hat{\tau}_1 \to \hat{\tau}_2) = \sigma(\hat{\tau}_1) \to \sigma(\hat{\tau}_2)$;
- $\sigma(\texttt{int}) = \texttt{int}$;
- $\sigma(T)$ est le type étendu associé à T dans σ.

Exemple 7.6. Avec $\sigma = [T_1 \mapsto \texttt{int}, T_2 \mapsto (\texttt{int} \to T_3)]$, on a donc

$$\sigma(T_1 \to T_2) = \texttt{int} \to (\texttt{int} \to T_3).$$

4. ***Attention*** c'est une paire, pas un couple.

Exemple 7.7. La contrainte $T_1 \stackrel{?}{=} T_2 \to T_3$ est égalisée par la solution $\sigma = [T_1 \mapsto T_2 \to \texttt{int}, T_3 \mapsto \texttt{int}]$.

Définition 7.6. Une *solution constante* de C est un dictionnaire sur $(\mathrm{ITyp}, \mathsf{Typ})$ (et pas $(\mathrm{ITyp}, \widehat{\mathsf{Typ}})$) qui est une solution de C.

Proposition 7.3. Soit $e \in \mathsf{FUN}$ et soit Γ tel que $\mathscr{Vl}(e) \subseteq \mathrm{dom}(\Gamma)$. Soit $T \in \mathrm{ITyp}$. Si σ est une solution constante de $\mathsf{CT}(e, \Gamma, T)$, alors $\Gamma \vdash e : \tau$ où $\tau = \sigma(T)$.

Preuve. On procède par induction sur e ; il y a 5 cas.

- ▷ Dans le cas $e = e_1\ e_2$, on écrit

  $$\mathsf{CT}(e, \Gamma, T) = \mathsf{CT}(e_1, \Gamma, T_1) \cup \mathsf{CT}(e_2, \Gamma, T_2) \cup \{T_1 \stackrel{?}{=} T_2 \to T\},$$

  où $И\ T_1, T_2$. Soit σ une solution constante de $\mathsf{CT}(e, \Gamma, T)$. Alors,
  - σ est une solution constante de $\mathsf{CT}(e_1, \Gamma, T_1)$;
  - σ est une solution constante de $\mathsf{CT}(e_2, \Gamma, T_1)$.

  Et, par induction, on sait que
  - $\Gamma \vdash e_1 : \sigma(T_1)$;
  - $\Gamma \vdash e_2 : \sigma(T_2)$.

  Par ailleurs, $\sigma(T_1) = \sigma(T_2) \to \sigma(T)$. On en conclut en appliquant $\mathscr{T}_{\mathrm{a}}$.
- ▷ Les autres cas se traitent similairement.

□

Proposition 7.4. Supposons $\Gamma \vdash e : \tau$. Alors, pour tout $T \in \mathrm{ITyp}$, il existe σ une solution constante de $\mathsf{CT}(e, \Gamma, T)$ telle que l'on ait l'égalité $\sigma(T) = \tau$.

Preuve. On procède par induction sur e. Il y a 5 cas.

- ▷ Dans le cas $e = e_1\ e_2$, supposons $\Gamma \vdash e_1\ e_2 : \tau$. Nécessairement, cette dérivation provient de $\Gamma \vdash e_1 : \tau_2 \to \tau$ et aussi $\Gamma \vdash e_2 : \tau_2$.

  Soit $T_0 \in \mathrm{ITyp}$, on a

  $$\mathsf{CT}(e, \Gamma, T_0) = \mathsf{CT}(e_1, \Gamma, T_1) \cup \mathsf{CT}(e_2, \Gamma, T_2) \cup \{T_1 \stackrel{?}{=} T_2 \to T_0\}.$$

  Et, par induction, on a σ_1 et σ_2 des solutions constantes de $\mathsf{CT}(e_1, \Gamma, T_1)$ et $\mathsf{CT}(e_2, \Gamma, T_2)$ avec $\sigma_1(T_1) = \tau_2 \to \tau$ et $\sigma_2(T_2) = \tau_2$.

  On définit σ en posant :

  - $\sigma(T) = \sigma_1(T)$ si $T \in \mathrm{Supp}(\mathsf{CT}(e_1, \Gamma, T_1))$;
  - $\sigma(T) = \sigma_2(T)$ si $T \in \mathrm{Supp}(\mathsf{CT}(e_2, \Gamma, T_2))$;
  - $\sigma(T_0) = \tau$.

  On vérifie bien que σ est solution constante de $\mathsf{CT}(e, \Gamma, T_0)$.
- ▷ Les autres cas se traitent similairement.

□

Théorème 7.1. On a $\Gamma \vdash e : \tau$ si, et seulement si $\forall T \in \mathrm{ITyp}$, l'ensemble de contraintes $\mathsf{CT}(e, \Gamma, T)$ admet une solution constante σ tel que $\sigma(T) = \tau$. □

Remarque 7.8. On a caractérisé l'ensemble des dérivations de $\Gamma \vdash e : \tau$ avec l'ensemble des solutions constantes de $\mathsf{CT}(e, \Gamma, T)$.

7.4.2 Termes et unification.

On va momentanément oublier FUN, pour généraliser à tout ensemble d'expressions. Ceci permet d'appliquer cet algorithme à une grande variété de « langages ».

Définition 7.7. On se donne

- un ensemble fini Σ de *constantes*, notées f, g, a, b où chaque constante $f \in \Sigma$ a un entier naturel nommé *arité* ;
- un ensemble infini V d'*inconnues*/de *variables*/de *variables d'unification* ; notées X, Y, Z (mais parfois x, y, z).

L'ensemble $\mathsf{T}(\Sigma, \mathrm{V})$ des *termes* sur (Σ, V), notés t, u, *etc*, est défini de manière inductive, ce qui est décrit par la grammaire :

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

où f est une constante d'arité k.

Remarque 7.9. L'intuition est que l'on étend, comme lors du passage de Typ à $\widehat{\mathsf{Typ}}$, un langage de départ pour ajouter des inconnues. La définition inductive a $|\Sigma| + 1$ constructeurs.

Intuitivement, les $X \in \mathrm{V}$ ne fait pas partie du langage de départ. Il n'y a pas de liens pour X.

Exemple 7.8. Avec $\Sigma = \{f^2, g^1, a^0, b^0\}$,

$$t_0 := f(g(a), f(X, f(Y, g(X)))) \in \mathsf{T}(\Sigma, V)$$

est un terme.

Définition 7.8. On définit $\mathsf{Vars}(t)$ l'ensemble des inconnues/variables de t par induction sur t. Il y a deux familles de cas :

- $\mathsf{Vars}(f(t_1, \ldots, t_k)) = \mathsf{Vars}(t_1) \cup \cdots \cup \mathsf{Vars}(t_k)$;
- $\mathsf{Vars}(X) = \{X\}$.

Exemple 7.9. Avec l'expression t_0 précédente, on a

$$\mathsf{Vars}(t_0) = \{X, Y\}.$$

Définition 7.9. Une *substitution*, notée $\sigma, \sigma_1, \sigma'$, *etc*, est un dictionnaire sur $(\mathrm{V}, \mathsf{T}(\Sigma, \mathrm{V}))$.

Si $X \in \mathrm{dom}(\sigma)$, on dit que σ est *définie* en X.

Soit σ une substitution et $t \in \mathsf{T}(\Sigma, \mathrm{V})$. Le résultat de l'application de σ à t, noté $\sigma(t)$, est défini par induction sur t, il y a deux familles de cas :

- $\sigma(f(t_1, \ldots, t_k)) = f(\sigma(t_1), \ldots, \sigma(t_k))$;
- $\sigma(X) = X$ si $X \notin \mathrm{dom}(\sigma)$;
- $\sigma(X)$ est le terme associé à X dans σ si $X \in \mathrm{dom}(\sigma)$.

Exemple 7.10. Avec $\sigma = [X \mapsto g(Y), Y \mapsto b]$, on a

$$\sigma(t_0) = f(g(a), f(\underbrace{g(Y)}, f(b, g(\underbrace{g(Y)}))))).$$

Attention ! On n'a pas de terme en $g(b)$: c'est une substitution *simultanée*.

Note 7.2. On rappelle qu'un dictionnaire peut être vu comme un ensemble fini de couples (X, t) avec $X \in \mathrm{V}$ et $t \in \mathsf{T}(\Sigma, \mathrm{V})$ tel que, pour toute variable $X \in \mathrm{V}$, il y a au plus un couple de la forme (X, t) dans la liste.

On utilise la notation $[t/X]$ pour représenter la notation $[X \mapsto t]$. Ceci est utiliser que lorsqu'on ne change qu'*une* variable.

Définition 7.10. Un *problème d'unification* est la donnée d'un ensemble fini de paires de termes (les contraintes) dans $\mathsf{T}(\Sigma, \mathrm{V})$. On note un tel problème $\mathscr{P} = \{t_1 \stackrel{?}{=} u_1, \ldots, t_k \stackrel{?}{=} u_k\}$.

Une *solution*, un *unificateur*, d'un tel $\mathscr{P}$ est une substitution σ telle que, pour toute contrainte $t \stackrel{?}{=} u$ dans $\mathscr{P}$, $\sigma(t)$ et $\sigma(u)$ sont le même terme, ce que l'on note $\sigma(t) = \sigma(u)$.

On note $\mathrm{U}(\mathscr{P})$ l'ensemble des unificateurs de P.

Exemple 7.11. Avec le problème d'unification

$$\mathscr{P}_1 = \{f(a, g(X)) \stackrel{?}{=} f(Z, Y), g(T) \stackrel{?}{=} g(Z)\},$$

les substitutions

- $\sigma_1 = [Z \mapsto a, Y \mapsto g(X), T \mapsto a]$;
- $\sigma_2 = [Z \mapsto a, Y \mapsto g(b), T \mapsto a, X \mapsto b]$;

sont des solutions de $\mathscr{P}_1$. Mais,

$$\sigma_3 = [Z \mapsto f(b, b), T \mapsto f(b, b), Y \mapsto g(b), X \mapsto b]$$

n'est pas une solution.

Laquelle des solutions σ_1 et σ_2 est meilleure ? On remarque que $\sigma_2 = [{}^{b}/_{X}] \circ \sigma_1$ (où la composition est définie « comme on le pense »[5]). Ainsi, σ_1 est « plus général » que σ_2 ; σ_2 est un « cas particulier » de σ_1.

Exemple 7.12 (Aucune solution)**.** Les problèmes

- $\mathscr{P}_2 = \{f(X, Y) \stackrel{?}{=} g(Z)\}$;
- $\mathscr{P}_3 = \{f(X, Y) \stackrel{?}{=} X\}$

n'ont aucune solution : $\mathrm{U}(\mathscr{P}_2) = \mathrm{U}(\mathscr{P}_3) = \emptyset$.

7.4.3 Algorithme d'unification (du premier ordre).

Définition 7.11. Un *état* est soit un couple $(\mathscr{P}, \sigma)$, soit $\bot$ (*l'état d'échec*).

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

Un état qui n'est, ni échec, ni succès, peut s'écrire sous la

5. Elle sera définie formellement ci-après.

forme $(\{t \stackrel{?}{=} t'\} \sqcup \mathscr{P}, \sigma)$, la contrainte $t \stackrel{?}{=} t'$ étant choisie de manière non-déterministe.

On définit une relation binaire $\to$ entre états par :

- $\bot \not\to$;
- $(\emptyset, \sigma) \not\to$;
- Il ne reste que les cas ni succès, ni échec, que l'on traite par la disjonction de cas :

1. $(\{f(t_1, \ldots, t_k) \stackrel{?}{=} f(u_1, \ldots, u_n) \sqcup \mathscr{P}, \sigma\}) \to (\{t_1 \stackrel{?}{=} u_1, \ldots, t_k \stackrel{?}{=} u_k\} \cup \mathscr{P}, \sigma)$;
2. $(\{f(t_1, \ldots, t_k) \stackrel{?}{=} g(u_1, \ldots, u_n) \sqcup \mathscr{P}, \sigma\}) \to \bot$ si $f \neq g$;
3. $(\{X \stackrel{?}{=} t\} \sqcup \mathscr{P}, \sigma) \to (\mathscr{P}[t/X], [t/X] \circ \sigma)$ où
   - $X \notin \mathsf{Vars}(t)$,
   - $\mathscr{P}[t/X] = \{u[t/X] \stackrel{?}{=} u'[t/X] \mid (u \stackrel{?}{=} u') \in \mathscr{P}\}$,
   - et $[t/X] \circ \sigma$ est la substitution telle que, quel que soit $Y \in \mathrm{V}$, $([t/X] \circ \sigma)(Y) = (\sigma(Y))[t/X]$;
4. $(\{X \stackrel{?}{=} t\} \sqcup \mathscr{P}, \sigma) \to \bot$ si $X \in \mathsf{Vars}(t)$ et $t \neq X$;
5. $(\{X \stackrel{?}{=} X\} \sqcup \mathscr{P}, \sigma) \to (\mathscr{P}, \sigma)$.

L'*état initial* de l'algorithme correspond à $(\mathscr{P}, \emptyset)$: le problème $\mathscr{P}$ muni de la substitution vide $\emptyset$.

Exemple 7.13. On applique l'algorithme d'unification comme

montré ci-dessous :

$$
\begin{aligned}
& \{\underbrace{f(a,X) \stackrel{?}{=} f(Y,a)}_{\text{choix}}, g(X) \stackrel{?}{=} g(Y)\}, \emptyset \\
\to\ & \{\underbrace{a \stackrel{?}{=} Y}_{\text{choix}}, X \stackrel{?}{=} a, g(X) \stackrel{?}{=} g(Y)\}, \emptyset \\
\to\ & \{\underbrace{X \stackrel{?}{=} a}_{\text{choix}}, g(X) \stackrel{?}{=} g(a)\}, [Y \mapsto a] \\
\to\ & \{\underbrace{g(a) \stackrel{?}{=} g(a)}_{\text{choix}}\}, [Y \mapsto a, X \mapsto a] \\
\to\ & \{\underbrace{a \stackrel{?}{=} a}_{\text{choix}}\}, [Y \mapsto a, X \mapsto a] \\
\to\ & \emptyset, [Y \mapsto a, X \mapsto a] \\
& .
\end{aligned}
$$

On peut remarquer que l'ensemble des clés de σ n'apparaît pas dans le problème ni dans les autres termes de la substitution : lorsqu'on ajoute une clé, elle disparaît du problème.

Définition 7.12. Un état $(\mathscr{P}, \sigma)$ est en *forme résolue* si, pour toute clé $X \in \mathrm{dom}(\sigma)$, alors X n'apparaît pas dans $\mathscr{P}$ et, quel que soit la clé $Y \in \mathrm{dom}(\sigma)$ alors $X \notin \mathsf{Vars}(\sigma(Y))$.

Remarque 7.10 (Notation)**.** Une substitution σ peut être vue comme un problème d'unification, que l'on note $\overset{?}{\sigma}$. (On passe d'un ensemble de couples à un ensemble de paires.)

Proposition 7.5. Si $(\mathscr{P}_0, \sigma_0)$ est en forme résolue et $(\mathscr{P}_0, \sigma_0) \to (\mathscr{P}_1, \sigma_1)$ alors $(\mathscr{P}_1, \sigma_1)$ est en forme résolue et

$$\mathrm{U}(\mathscr{P}_0 \cup \overset{?}{\sigma_0}) = \mathrm{U}(\mathscr{P}_1 \cup \overset{?}{\sigma_1}).$$

Preuve. La vraie difficulté se trouve dans le 3ème cas (les cas 1 et 5 sont immédiats). Pour cela, on utilise le lemme « technique » ci-dessous.

> **Lemme 7.4.** Si $X \notin \mathrm{dom}(\sigma)$ alors
>
> $$[t/X] \circ \sigma = \left[X \mapsto t, Y_1 \mapsto (\sigma(Y_1))[t/X], \ldots, Y_l \mapsto (\sigma(Y_k))[t/X]\right],$$
>
> où $\mathrm{dom}(\sigma) = \{Y_1, \ldots, Y_k\}$. □

□

Proposition 7.6. On note $\to^\star$ la clôture réflexive et transitive de la relation $\to$.

1. Un *unificateur le plus général* (*mgu*[6] dans la littérature anglaise) est une solution $\sigma \in \mathrm{U}(\mathscr{P})$ telle que, quelle que soit $\sigma' \in \mathrm{U}(\mathscr{P})$, il existe σ'' telle que $\sigma' = \sigma'' \circ \sigma$.

   Si $(\mathscr{P}, \emptyset) \to^\star (\emptyset, \sigma)$ alors σ est un unificateur le plus général de $\mathscr{P}$.
2. Si $(\mathscr{P}, \emptyset) \to^\star \bot$ alors $\mathrm{U}(\mathscr{P}) = \emptyset$.

Preuve. 1. On montre par induction sur $(\mathscr{P}, \emptyset) \to^\star (\emptyset, \sigma)$ l'égalité $\mathrm{U}(\mathscr{P}) = \mathrm{U}(\overset{?}{\sigma})$ à l'aide de la proposition précédente. Puis, on conclut avec le lemme suivant.

> **Lemme 7.5.** Pour toute substitution σ, alors σ est un unificateur le plus général de $\overset{?}{\sigma}$.

6. Pour *Most Général Unifier*

Preuve. Soit $\sigma' \in \mathrm{U}(\overset{?}{\sigma})$ et soit $X \in \mathrm{V}$. On montre que $\sigma' \circ \sigma = \sigma'$.

- ▷ Si $X \in \mathrm{dom}(\sigma)$, alors $\sigma'(\sigma(X)) = \sigma'(X)$ car σ' satisfait la contrainte $X \overset{?}{=} \sigma(X)$.
- ▷ Si $X \notin \mathrm{dom}(\sigma)$ alors $\sigma'(\sigma(X)) = \sigma'(X)$.

Ainsi $\sigma' \circ \sigma = \sigma'$. □

2. On montre que si $(\mathscr{P}, \emptyset) \to \bot$ alors $\mathrm{U}(\mathscr{P} \cup \overset{?}{\sigma})$. Pour le 2nd cas, c'est immédiat. Pour le 4ème cas, on procède par l'absurde. Soit σ_0 qui satisfait $X \overset{?}{=} t$ avec $X \in \mathsf{Vars}(t)$ et $X \neq t$. Alors $\sigma_0(X) = \sigma_0(t)$, qui contient $\sigma_0(X)$ et c'est un sous-ensemble strict. Absurde.

   On raisonne ensuite par induction sur $\to^\star$ pour conclure que $(\mathscr{P}, \emptyset) \to^\star (\mathscr{P}_0, \sigma_0) \to \bot$.

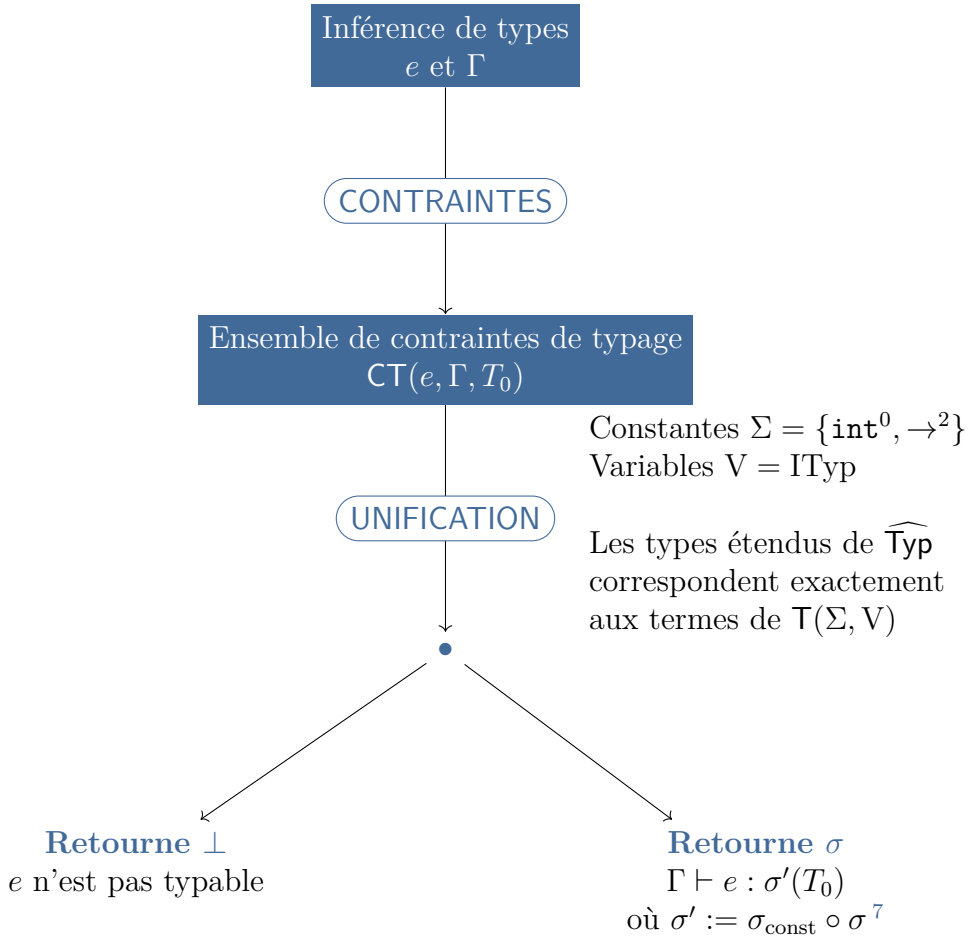
□

Lemme 7.6. La relation $\to$ est terminante (il n'y a pas de chaîne infinie avec cette relation).

Preuve. Vue plus tard. □

Théorème 7.2. L'algorithme d'unification calcule un unificateur le plus général si, et seulement si le problème initial a une solution. □

7.4.4 Retour sur l'inférence de types pour FUN.



Ceci conclut notre étude du petit langage fonctionnel FUN.

7. L'unificateur le plus général peut contenir des variables dans ses valeurs qui ne sont pas des clés (par exemple lors du typage de $\texttt{fun}\, x \to x$). Il faut donc composer σ avec une substitution « constante » pour effacer ces variables inutilisée.

8 Un petit langage impératif, IMP.

Sommaire.

8.1 Syntaxe et sémantique opérationnelle.

On se donne $\mathbb{Z}$ et V un ensemble infini de variables IMP, notées x, y, z. On définit plusieurs grammaires :

Arith. Les expressions arithmétiques $a ::= \underline{k} \mid a_1 \oplus a_2 \mid x$;[1]

Valeurs booléennes. $bv ::= \texttt{true} \mid \texttt{false}$;

Bool. Les expressions booléennes $b ::= bv \mid b_1 \wedge b_2 \mid a_1 \geq a_2$;

Com. Les commandes $c ::= x \texttt{ := } a \mid c_1 \texttt{ ; } c_2 \mid \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } b \texttt{ do } c \mid \texttt{skip}$.

Sans explicitement le dire, on s'autorise à étendre les expressions arithmétiques avec, par exemple, les produits, les soustractions. De même pour les expressions booléennes.

On définit, par induction sur c, $\mathsf{Vars}(c)$ l'ensemble des variables dans la commande c. Il y a 5 cas.

1. Et on arrêtera rapidement de mettre des barres sous les entiers et d'entourer les plus.

Exemple 8.1. La commande

$$z := 1 \ ; \ \texttt{while}\ (x > 0)\ \texttt{do}\ (z := z \times x \ ; \ x := x - 1)$$

représente un programme calculant la factorielle d'un nombre x. On le notera c_{fact}.

8.1.1 Sémantique opérationnelle à grands pas.

Définition 8.1 (États mémoire). On se donne $\mathscr{M}$ un ensemble de dictionnaires, notés σ, σ', *etc* sur $(\mathrm{V}, \mathbb{Z})$.

Si $x \in \mathrm{dom}(\sigma)$ et $k \in \mathbb{Z}$ on note $\sigma[x \mapsto k]$ l'état mémoire σ' défini par

- $\sigma'(x) := k$;
- $\sigma'(y) := \sigma(y)$ si $y \in \mathrm{dom}(\sigma) \setminus \{x\}$.

Ici, on *écrase* la valeur de x dans l'état mémoire σ.

On définit $c, \sigma \Downarrow \sigma'$ (l'évaluation de c sur σ produit σ', c fait passer de σ à σ') par les règles d'inférences ci-dessous

$$\frac{}{\texttt{skip}, \sigma \Downarrow \sigma}\ \mathscr{E}_{\text{skip}} \qquad \frac{c_1, \sigma \Downarrow \sigma' \qquad c_2, \sigma' \Downarrow \sigma''}{c_1 \ ; \ c_2, \sigma \Downarrow \sigma''}\ \mathscr{E}_{\text{seq}}$$

$$\frac{b, \sigma \Downarrow \texttt{true} \qquad c_1, \sigma \Downarrow \sigma'}{\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma \Downarrow \sigma'}\ \mathscr{E}_{\text{it}} \qquad \frac{b, \sigma \Downarrow \texttt{false} \qquad c_2, \sigma \Downarrow \sigma'}{\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma \Downarrow \sigma'}\ \mathscr{E}_{\text{if}}$$

$$\sigma' = \sigma[x \mapsto k]\ \frac{a, \sigma \Downarrow k}{x := a, \sigma \Downarrow \sigma'}\ \mathscr{E}_{\text{aff}} \qquad \frac{b, \sigma \Downarrow \texttt{false}}{\texttt{while}\ b\ \texttt{do}\ c, \sigma \Downarrow \sigma}\ \mathscr{E}_{\text{wf}}$$

$$\frac{b, \sigma \Downarrow \texttt{true} \qquad c, \sigma \Downarrow \sigma' \qquad \texttt{while}\ b\ \texttt{do}\ c, \sigma' \Downarrow \sigma''}{\texttt{while}\ b\ \texttt{do}\ c, \sigma \Downarrow \sigma''}\ \mathscr{E}_{\text{wf}}$$

où l'on a deux autres relations (la couleur a de l'importance ici) :

- *l'évaluation des expressions arithmétiques* $a, \sigma \Downarrow k$ (a s'*évalue* en k dans σ)

$$\frac{}{\underline{k}, \sigma \Downarrow k} \qquad \sigma(x) = k \ \frac{}{x, \sigma \Downarrow k} \qquad k = k_1 + k_2 \ \frac{a_1, \sigma \Downarrow k_1 \qquad a_2, \sigma \Downarrow k_2}{a_1 \oplus a_2, \sigma \Downarrow k}$$

- *l'évaluation des expressions booléennes* $b, \sigma \Downarrow bv$ (b s'*évalue* en bv dans σ)

$$\frac{}{bv, \sigma \Downarrow bv} \qquad bv = \texttt{true ssi } bv_1 \text{ et } bv_2 \ \frac{b_1, \sigma \Downarrow bv_1 \qquad b_2, \sigma \Downarrow bv_2}{b_1 \wedge b_2, \sigma \Downarrow bv}$$

$$bv = \texttt{true ssi } k_1 \geq k_2 \ \frac{a_1, \sigma \Downarrow k_1 \qquad a_2, \sigma \Downarrow k_2}{a_1 \geq a_2, \sigma \Downarrow bv.}$$

Remarque 8.1 (des « variables » partout !).

- Les variables dans FUN sont les paramètres des fonctions, elles peuvent être liées, libres, et on peut procéder à de l'α-conversion. [2]
- Les variables d'unification sont des inconnues. Il y a une notion de substitution, mais pas de liaison.
- Les variables dans IMP sont des cases mémoire, des registres, et il n'y a pas de liaison.

Remarque 8.2. Soit c une commande, et $\sigma \in \mathscr{M}$. Il peut arriver que, quel que soit $\sigma' \in \mathscr{M}$, on n'ait pas $c, \sigma \Downarrow \sigma'$, soit parce que $\mathrm{dom}(\sigma)$ est trop petit, et l'exécution se bloque ; soit parce que le programme diverge, par exemple

```
while true do skip
```

2. C'est similaire au cas de la variable x dans $\int_0^7 f(x)\,\mathrm{d}x$.

diverge car on n'a pas de dérivation finies :

$$\frac{\texttt{true}, \sigma \Downarrow \texttt{true} \qquad \overline{\texttt{skip}, \sigma \Downarrow \sigma} \qquad \texttt{while true do skip}, \sigma \Downarrow ?}{\texttt{while true do skip}, \sigma \Downarrow ?}\ \mathscr{E}_{\text{wt}}.$$

On peut définir des petits pas pour IMP (vu plus tard en cours, ou en TD), mais on s'intéresse plus à une autre sémantique, la *sémantique dénotationnelle*.

8.2 Sémantique dénotationnelle de IMP.

$$\begin{array}{ccc} c & \xrightarrow{\quad \mathscr{D} \quad} & \text{fonction partielle } \mathscr{M} \rightharpoonup \mathscr{M} \\ & & \updownarrow \\ & & \text{relation binaire sur } \mathscr{M} \text{ déterministe/fonctionnelle} \end{array}$$

On définit les relations

- $\mathscr{D}(a) \subseteq \mathscr{M} \times \mathbb{Z}$ fonctionnelle ;
- $\mathscr{D}(b) \subseteq \mathscr{M} \times \{\texttt{true}, \texttt{false}\}$ fonctionnelle ;
- $\mathscr{D}(c) \subseteq \mathscr{M} \times \mathscr{M}$ fonctionnelle.

On ne traitera que la définition de $\mathscr{D}(c)$, les autres sont laissées en exercice.

On définit $\mathscr{D}(c)$ par induction sur c, il y a 5 cas.

- $\mathscr{D}(\texttt{skip}) = \{(\sigma, \sigma)\}$;
- $\mathscr{D}(x \texttt{ := } a) = \{(\sigma, \sigma') \mid x \in \mathrm{dom}(\sigma), \sigma' = \sigma[x \mapsto k] \text{ et } (\sigma, k) \in \mathscr{D}(a)\}$;
- $\mathscr{D}(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2) = \{(\sigma, \sigma' \mid (\sigma, \texttt{true}) \in \mathscr{D}(b), (\sigma, \sigma') \in \mathscr{D}(c_1))\} \cup \{(\sigma, \sigma' \mid (\sigma, \texttt{false}) \in \mathscr{D}(b), (\sigma, \sigma') \in \mathscr{D}(c_2))\}$;
- $\mathscr{D}(c_1 \texttt{ := } c_2) = \{(\sigma, \sigma'') \mid \exists \sigma', (\sigma, \sigma') \in \mathscr{D}(c_1) \text{ et } (\sigma', \sigma'') \in \mathscr{D}(c_2)\}$;[3]
- $\mathscr{D}(\texttt{while } b \texttt{ do } c) = ???$.

3. C'est la composée de $\mathscr{D}(c_2)$ avec $\mathscr{D}(c_1)$.

Pour la sémantique dénotationnelle de la boucle `while`, on s'appuie sur l'« équivalence » des commandes

$$\texttt{while}\ b\ \texttt{do}\ c \qquad \text{et} \qquad \texttt{if}\ b\ \texttt{then}\ (c\ \texttt{:=}\ \texttt{while}\ b\ \texttt{do}\ c)\ \texttt{else skip}.$$

On introduit, pour $\mathsf{R} \subseteq \mathscr{M} \times \mathscr{M}$, la relation

$$\begin{aligned} F(\mathsf{R}) = \quad & \{(\sigma,\sigma) \mid (\sigma,\texttt{false}) \in \mathscr{D}(b)\} \\ & \cup \{(\sigma,\sigma') \mid (\sigma,\texttt{true}) \in \mathscr{D}(b), \exists \sigma', (\sigma,\sigma') \in \mathscr{D}(c) \text{ et } (\sigma',\sigma'') \in \mathsf{R}\}. \end{aligned}$$

On a envie de définir $\mathscr{D}(\texttt{while}\ b\ \texttt{do}\ c)$ comme un point fixe de F.

L'ensemble des relations binaires fonctionnelles sur $\mathscr{M}$ **n'est pas** un treillis complet (à cause se $\mathsf{R}_1 \cup \mathsf{R}_2$ qui n'est pas nécessairement fonctionnelle). On ne peut donc pas appliquer le théorème de Knaster-Tarski.

En revanche, c'est un domaine : si $e_0 \subseteq e_1 \subseteq \cdots \subseteq e_n \subseteq \cdots$ alors l'union $\bigcup_{i\geq 0} e_i$ existe. L'inclusion $e \subseteq e'$ signifie que e' est « plus définie » que e. L'ensemble des relations fonctionnelles sur $\mathscr{M}$ est donc un domaine avec $\bot = \emptyset$. On sait donc que, pour toute fonction F continue, alors F admet un point fixe, qui est égal à

$$\emptyset \cup F(\emptyset) \cup F^2(\emptyset) \cup \cdots = \bigcup_{i\geq 0} F^i(\emptyset).$$

La fonction F définie plus haut est continue, ce qui nous permet de définir

$$\mathscr{D}(\texttt{while}\ b\ \texttt{do}\ c) = \bigcup_{i\geq 0} F^i(\emptyset).$$

Exemple 8.2. On considère $c_0 = \texttt{while}\ x \neq 3\ \texttt{do}\ x\ \texttt{:=}\ x - 1$. Ainsi, la fonction F définie avant $c = c_0$ est

$$\begin{aligned} F_0(\mathsf{R}) = \quad & \{(\sigma,\sigma) \mid \sigma(x) = 3\} \\ & \cup \{(\sigma,\sigma'') \mid \sigma(x) \neq 3, \exists \sigma', \sigma = [x \mapsto \sigma(x) - 1], (\sigma,\sigma') \in \mathsf{R}\}. \end{aligned}$$

On a

- ▷ $F_0^0(\emptyset) = \{(\sigma,\sigma) \mid \sigma(x) = 3\}$;

- ▷ $F_0^1(\emptyset) = \{(\sigma,\sigma) \mid \sigma(x) = 3\} \cup \{(\sigma,\sigma') \mid \sigma' = [x \mapsto 3], \sigma(x) = 4\}$;
- ▷ $F_0^2(\emptyset) = \{(\sigma,\sigma) \mid \sigma(x) = 3\} \cup \{(\sigma,\sigma') \mid \sigma' = [x \mapsto 3], \sigma(x) \in \{4,5\}\}$;
- ▷ $F_0^2(\emptyset) = \{(\sigma,\sigma) \mid \sigma(x) = 3\} \cup \{(\sigma,\sigma') \mid \sigma' = [x \mapsto 3], \sigma(x) \in \{4,5,6\}\}$;
- ▷ *etc.*

On a bien

$$\emptyset \subseteq F_0(\emptyset) \subseteq F_0^2(\emptyset) \subseteq \cdots .$$

Si $\sigma(x) = 0$, alors quel que soit σ', on a $(\sigma,\sigma') \notin \mathscr{D}(c_0)$.

Exemple 8.3. Ainsi défini,

$$\mathscr{D}(\texttt{while true do skip}) = \emptyset.$$

Théorème 8.1. On a $c, \sigma \Downarrow \sigma'$ si et seulement si $(\sigma,\sigma') \in \mathscr{D}(c)$.

Preuve. ▷ « $\Longrightarrow$ » Par induction sur la relation $c, \sigma \Downarrow \sigma'$.

▷ « $\Longleftarrow$ » Par induction sur c, où l'on utilise le résultat suivant :

$$\forall n, \quad (\sigma,\sigma') \in F^n(\emptyset) \implies c, \sigma \Downarrow \sigma'.$$

□

Lemme 8.1. Quels que soient c, σ, σ_1, si c, σ, σ_1 alors,

$$\forall \sigma_2, \quad c, \sigma \Downarrow \sigma_2 \implies \sigma_1 = \sigma_2.$$

Preuve. Une mauvaise idée est de procéder par induction sur c. Il y a 5 cas, et dans le cas `while`, ça bloque parce que la relation grands pas n'est pas définie par induction sur c dans le cas `while`.

On procède par induction sur $c, \sigma \Downarrow \sigma_1$. □

De manière générale, avec IMP, on ne montre pas des résultats de la forme $c, \sigma \Downarrow \sigma' \implies \mathscr{P}$ par induction sur c, car cela ne fonctionne

pas, on n'a pas les bonnes hypothèses. On procède par induction sur la relation $c, \sigma \Downarrow \sigma'$.

8.3 Coinduction.

On retourne sur le théorème de Knaster-Tarski pour la définition d'ensembles et de relations. En notant E l'ensemble ambiant, on travaille dans le treillis complet $(\wp(E), \subseteq)$, avec des fonctions f croissantes dans $\wp(E)$. Le théorème de Knaster-Tarski nous donne ainsi le plus petit pré-point fixe de f, que l'on notera μf. Le principe de la preuve par induction est ainsi :

$$\text{si } A \subseteq E \text{ vérifie } f(A) \subseteq A \text{ alors on a } \mu f \subseteq A.$$

De plus, si f est continue (car $(\wp(E), \subseteq)$ est un domaine), alors on peut calculer explicitement ce plus petit (pré)-point fixe avec la formule $\bigcup_{n \in \mathbb{N}} f^n(\emptyset)$. On part du « bas » et on ajoute des éléments un par un.

Exemple 8.4. Pour l'exemple de `nat`, on a

$$\forall A \subseteq E, \qquad f(A) = \{\mathtt{O}\} \cup \{\mathtt{S}\ x \mid x \in A\},$$

c'est une fonction continue, et on a

$$\mu f = \{\mathtt{S}^n\mathtt{O} \mid n \in \mathbb{N}\},$$

avec $\mathtt{S}^n\ x = \mathtt{S}\ \mathtt{S}\ \cdots\ \mathtt{S}\ x$ et la convention $\mathtt{S}^0\ x = x$. En effet, on a l'appartenance de $\mathtt{S}^n\mathtt{O} \in \bigcup_{m \in \mathbb{N}} f^m(\emptyset)$ et $f(\{\mathtt{S}^n\ \mathtt{O}\}) = \{\mathtt{S}^{n+1}\ \mathtt{O}\}$.

Remarque 8.3 (Remarque fondamentale !)**.** Considérons un treillis complet $(E, \sqsubseteq)$. Alors, le treillis $(E, \sqsupseteq)$ est complet, où l'on note $y \sqsupseteq x$ dès lors que $x \sqsubseteq y$ (on renverse l'ordre).

Un majorant pour $\sqsubseteq$ est un minorant pour $\sqsupseteq$ et inversement. Ainsi, le plus plus petit des majorants $\bigsqcup_{\sqsubseteq} A$ pour $\sqsubseteq$ est le plus petit des minorants $\bigsqcap_{\sqsupseteq} A$ pour $\sqsupseteq$. Réciproquement, le plus petit

des majorants pour $\sqsubseteq$, $\bigsqcap_{\sqsubseteq} A$ est égal au plus grand majorant pour la relation $\sqsupseteq$, $\bigsqcup_{\sqsupseteq} A$.

On se place ainsi sur le treillis complet $(\wp(E), \supseteq)$. Une fonction est croissante pour $\subseteq$ si et seulement si elle est croissante pour $\supseteq$ (***attention***, elle n'est pas décroissance pour cette deuxième relation). Appliquons le théorème de Knaster-Tarski sur ce nouveau treillis complet à une fonction croissante. Le théorème nous fournis un pré-point fixe pour l'ordre $\supseteq$ (*i.e.* qui vérifie $f(A) \supseteq A$), c'est-à-dire un post-point fixe pour l'ordre $\subseteq$ (*i.e.* qui vérifie $A \subseteq f(A)$). Et, c'est le plus petit point fixe pour $\supseteq$, donc le plus grand point fixe pour $\subseteq$, que l'on notera νf.

Avec le théorème de point fixe sur les domaines, et en supposant f continue, on calcule explicitement que le plus grand point fixe νf vaut l'intersection $\bigcap_{n\in\mathbb{N}} f^n(E)$. On part du haut, et on nettoie progressivement, on raffine notre partie de E.

Ce que l'on a fait là, cela s'appelle de la ***coinduction***.

Exemple 8.5. Par exemple, on définit `conat` par coinduction. En Rocq, cela donne le code ci-dessous.

```
CoInductive conat : Set := cO | cS (n : conat).
```

Code 8.1 | *Définition de* `conat`

Pour illustrer le « nettoyage » effectué dans la définition coinductive, on considère une feuille étiquetée par le mot « `banane` ». A-t-on `cS banane` $\in$ `conat` ? Premièrement, on a `cS banane` $\in E$ car E est l'ensemble (très grand) des arbres étiquetés par des chaînes de caractères. Deuxièmeement, on a `cS banane` $\in f(E)$ car c'est le successeur de `banane` $\in E$. Troisièmement, et c'est là où ça casse, on a `cS banane` $\notin f^2(E)$ parce que `banane` $\notin f(E)$.

Avec la fonction f définie précédemment, on a

$$f^n(E) = \{\texttt{cO}, \texttt{cS cO}, \dots, \texttt{cS}^{n-1}\ \texttt{cO}\} \cup \{\texttt{cS}^n\ x \mid x \in E\}.$$

Ainsi, on récupère tous les entiers de nat, mais d'autres entiers (oui, il y en a plusieurs) infinis, ayant ainsi une dérivation infinie. Par exemple, il existe $\omega \in \mathsf{conat}$ tel que $\omega = \mathtt{cS}\ \omega$. En Rocq, pour le définir, on ferai :

```
CoFixpoint ω := cS ω
```

Code 8.2 | *Définition de ω, un entier infini*

Pour montrer que $\omega \in \mathsf{conat}$, il faut et il suffit de montrer l'inclusion $\{\omega\} \subseteq f(\{\omega\}) = \{\mathtt{c0}, \mathtt{cS}\ \omega\} = \{\mathtt{c0}, \omega\}$, qui est vraie, et on a ainsi $\{\omega\} \subseteq \mathsf{conat}$.

Le principe de la preuve par coinduction permet d'établir qu'un ensemble est contenu dans le plus grand point fixe. Avec le treillis des parties muni de $\subseteq$, cela permet de montrer que $A \subseteq E$ est inclus dans le plus grand post-point fixe de f et, pour cela, il suffit de montrer que $A \subseteq f(A)$, c'est-à-dire que A est un post-point fixe de f. C'est ce que l'on a fait dans l'exemple avec ω.

Par coinduction, on peut par exemple montrer que l'on a, pour tous états mémoire σ, σ',

$$\mathtt{while\ true\ do\ skip}, \sigma \Downarrow \sigma'.$$

8.4 Divergences en IMP.

On donne une définition coinductive de la divergence en IMP, que l'on notera $c, \sigma \Uparrow$ avec les règles

$$\frac{c_1, \sigma \Uparrow}{c_1\ ;\ c_2, \sigma \Uparrow} \qquad \frac{c_1, \sigma \Downarrow \sigma' \qquad c_2, \sigma' \Uparrow}{c_1\ ;\ c_2, \sigma \Uparrow} \qquad \frac{b, \sigma \Downarrow \mathtt{true} \qquad c_1, \sigma \Uparrow}{\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \sigma \Downarrow}$$

$$\frac{b, \sigma \Downarrow \mathtt{false} \qquad c_2, \sigma \Uparrow}{\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \sigma \Downarrow} \qquad \frac{b, \sigma \Downarrow \mathtt{true} \qquad c, \sigma \Uparrow}{\mathtt{while}\ b\ \mathtt{do}\ c, \sigma \Uparrow}$$

$$\frac{b, \sigma \Downarrow \mathtt{true} \qquad c, \sigma \Downarrow \sigma' \qquad \mathtt{while}\ b\ \mathtt{do}\ c, \sigma' \Uparrow}{\mathtt{while}\ b\ \mathtt{do}\ c, \sigma \Uparrow}$$

On n'a pas de règle pour la divergence si $b, \sigma \Downarrow \texttt{false}$, car dans ce cas là, on ne peut pas diverger (c'est équivalent à un `skip`).

Le plus grand point fixe ne contient que des dérivations infinies, qui correspondent à des exécutions divergentes d'un programme IMP à partir d'un état mémoire donné. En effet, ceci vient du fait que, si on interprète ces règles comme des règles inductives, la relation obtenue est l'ensemble vide...

9 Logiques de programmes

Sommaire.

9.1 Logique de Floyd–Hoare.

On considère des formules logiques, des *assertions* (définies formellement ci-après), que l'on notera A, A', B, *etc.* Un triplet de Hoare est de la forme $\{A\}c\{A'\}$ (la notation est inhabituelle pour les triplets, mais c'est une notation commune dans le cas des triplets de Hoare), où l'on nomme A la *précondition* et A' la *postcondition*.

Exemple 9.1. Les triplets suivants sont des triplets de Hoare :

1. $\{x \geq 1\}y := x + 2\{x \geq 1 \wedge y \geq 3\}$ qui est une conclusion naturelle ;
2. $\{n \geq 1\}c_{\text{fact}}\{r = n!\}$ où l'on note c_{fact} la commande

   $$x := n \ ; \ z := 1 \ ; \ \texttt{while}\,(x > 0)\,\texttt{do}\,(z := z \times x \ ; \ x := x - 1)\ ,$$

   qui calcule naturellement la factorielle de n ;
3. $\{x < 0\}c\{\texttt{true}\}$ même s'il ne nous dit rien d'intéressant (tout état mémoire vérifie `true`) ;
4. $\{x < 0\}c\{\texttt{false}\}$ qui diverge dès lors que $x < 0$.

On considère un ensemble $I \ni i$ infini d'*index*, des « inconnues ». On commence par définir les expressions arithmétiques étendues

$$a ::= \underline{k} \mid a_1 \oplus a_2 \mid x \mid i,$$

puis définit les *assertions* par la grammaire ci-dessous :

$$A ::= bv \mid A_1 \vee A_2 \mid A_1 \wedge A_2 \mid a_1 \geq a_2 \mid \exists i, A.$$

On s'autorisera à étendre, implicitement, les opérations réalisées dans les expressions arithmétiques, et les comparaisons effectuées dans les assertions.

On ajoute la liaison d'α-conversion : les assertions $\exists i, x = 3 * i$ et $\exists j, x = 3 * j$ sont α-équivalentes. On note $i\ell(A)$ l'ensemble des index libres de l'assertion A, et on dira que A est *close* dès lors que $i\ell(A) = \emptyset$. On note aussi $A[{}^{k}/_{i}]$ l'assertion A où $k \in \mathbb{Z}$ remplace $i \in I$.

Définition 9.1. Considérons A close et $\sigma \in \mathscr{M}$. On définit par induction sur A (4 cas) une relation constituée de couples (σ, A), notés $\sigma \models A$ (« σ satisfait A »), et en notant $\sigma \not\models A$ lorsque (σ, A) n'est pas dans la relation :

- $\sigma \models \texttt{true}\ \forall \sigma \in \mathscr{M}$;
- $\sigma \models A_1 \vee A_2$ si et seulement si $\sigma \models A_1$ ou $\sigma \models A_2$;
- $\sigma \models a_1 \geq a_2$ si et seulement si on a $a_1, \sigma \Downarrow k_1$ et $a_2, \sigma \Downarrow k_2$ et $k_1 \geq k_2$;
- $\sigma \models \exists i, A$ si et seulement s'il existe $k \in \mathbb{Z}$ tel que $\sigma \models A[{}^{k}/_{i}]$.

On écrit $\models A$ (« A est valide ») lorsque pour tout σ tel que $\mathrm{dom}(\sigma) \supseteq \mathsf{vars}(A)$, on a $\sigma \models A$.

9.1.1 Règles de la logique de Hoare : dérivabilité des triplets de Hoare.

Les triplets de Hoare, notés $\{A\}c\{A'\}$ avec A et A' closes, où A est *précondition*, c est commande IMP, et A' est *postcondition*. On définit

une relation $\vdash \{A\}c\{A'\}$ sur les triplets de Hoare :

$$\frac{\vdash \{A \wedge b\}c_1\{A'\} \quad \vdash \{A \wedge \neg b\}c_2\{A'\}}{\vdash \{A\}\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2\{A'\}} \qquad \frac{}{\vdash \{A\}\texttt{skip}\{A\}}$$

$$\frac{\vdash \{A\}c_1\{A'\} \quad \vdash \{A'\}c_2\{A''\}}{\vdash \{A\}c_1 \texttt{ ; } c_2\{A''\}} \qquad \frac{\vdash \{A \wedge b\}c\{A\}}{\{A\}\texttt{while } b \texttt{ do } c\{A \wedge \neg B\}}$$

$$\frac{\models B \implies A \quad \models A' \implies B'}{\{A\}c\{A'\}} \qquad \frac{}{\{A[a/x]\}x \texttt{ := } a\{A\}}$$

La dernière règle semble à l'envers, mais c'est parce que la logique de Hoare fonctionne fondamentalement à l'envers.

Dans la règle de dérivation pour la boucle `while`, l'assertion manipulée, A, est un *invariant*.

L'avant dernière règle s'appelle la *règle de conséquence* : on ne manipule pas le programme, la commande, mais plutôt les pré- et post-conditions.

La relation $\vdash \{A\}c\{A'\}$ s'appelle la *sémantique opérationnelle* de IMP.

Définition 9.2. On définit la relation de *satisfaction*, sur les triplets de la forme $\{A\}c\{A'\}$ avec A, A' closes, avec $\sigma \models \{A\}c\{A'\}$ si et seulement si dès lors que $\sigma \models A$ et $c, \sigma \Downarrow \sigma'$ alors on a $\sigma' \models A'$.

On définit ensuite la relation de *validité* par $\models \{A\}c\{A'\}$ si et seulement si pour tout $\sigma \in \mathscr{M}$, $\sigma \models \{A\}c\{A'\}$.

Théorème 9.1 (Correction de la logique de Hoare.)**.** Si $\vdash \{A\}c\{A'\}$ alors $\models \{A\}c\{A'\}$.

Preuve. On procède par induction sur $\vdash \{A\}c\{A'\}$. Il y a 6 cas.

▷ *Règle de conséquence.* On sait

$$\models B \implies A \text{ et } \models A' \implies B',$$

et l'hypothèse d'induction. On doit montrer $\models \{B\}c\{B'\}$. Soit σ tel que $\models B$, et supposons $c, \sigma \Downarrow \sigma'$. On a $\models A$ par hypothèse. Puis, par hypothèse d'induction, $\sigma' \models A'$ et donc $\sigma' \models B'$.

▷ *Règle* `while`. Considérons $c = \texttt{while}\ b\ \texttt{do}\ c_0$. On sait par induction que $\models \{A \wedge b\}c_0\{A\}$ et l'hypothèse d'induction. Il faut montrer $\models \{A\}\texttt{while}\ b\ \texttt{do}\ c_0\{A \wedge \neg b\}$, c'est à dire, si $\sigma \models A$ et $(\star)$: $\texttt{while}\ b\ \texttt{do}\ c_0, \sigma \Downarrow \sigma'$ alors $\sigma' \models A \wedge \neg b$. Pour montrer cela, il est nécessaire de faire une induction sur la dérivation de $(\star)$, « sur le nombre d'itérations dans la boucle ».

▷ Autres cas en exercice.

□

Le sens inverse, la réciproque, s'appelle la *complétude*. On l'étudiera rapidement après.

Remarque 9.1. Concrètement, on écrit des programmes annotés.

$\{x \geq 1\}$

$\{x \geq 1 \wedge x+2+x+2 \geq 6\}$

$y := x + 2$;

$\{x \geq 1 \wedge y + y \geq 6\}$

$z := y + y$

$\{x \geq 1 \wedge z \geq 6\}$

$\{x \geq 1 \wedge z \geq 6\}$

Pour démontrer la complétude de la logique de Hoare, on s'appuie sur la notion de *plus faible précondition* : étant données une commande c et une assertion B, alors la *plus faible précondition* associée à c, B est l'ensemble des états mémoire

$$\mathrm{wp}(c, B) := \{\sigma \mid c, \sigma \Downarrow \sigma' \implies \sigma' \models B\}.$$

Ainsi, $\mathrm{wp}(c, B)$ est l'ensemble des états mémoire à partir duquels on aboutit à un état satisfaisant B, après une exécution terminante de c.

Proposition 9.1. Pour toute commande c et toute formule B, il existe une assertion $\mathrm{W}(c, B)$ telle que $\sigma \models \mathrm{W}(c, B)$ si et seulement si $\sigma \in \mathrm{wp}(c, B)$.

Preuve. On procède par induction sur c. Tout fonctionne, sauf pour *while*... Pour le cas de la boucle *while*, on utilise la caractérisation suivante :

$$\sigma \in \mathrm{wp}(\texttt{while}\ b\ \texttt{do}\ c_0, B)$$

$$\Updownarrow$$

$$\forall k, \forall \sigma_0, \dots, \sigma_k \text{ si } \sigma_0 = \sigma \text{ et } \forall i < k, (\sigma_i, b \Downarrow \texttt{true} \text{ et } c_0, \sigma_i \Downarrow \sigma_{i+1})$$
$$\text{alors } \sigma_k \models b \vee B.$$

On peut définir cette assertion en définissant des assertions pour :

- ▷ décrire un état mémoire σ_i $(X_1^i = v_1 \wedge \cdots \wedge X_n^i = v_n)$;
- ▷ exprimer les conditions $\sigma_i, c \Downarrow \sigma_{i+1}$ par induction ;
- ▷ exprimer les quantifications $\forall k, \sigma_0, \dots, \sigma_k \dots$ on demande à Kurt Gödel.

Ainsi, on a bien une assertion $\mathrm{W}(c, B)$ telle que

$$\forall \sigma, \qquad \sigma \in \mathrm{wp}(c, B) \iff \sigma \models \mathrm{W}(c, B).$$

□

10 Mémoire structurée, logique de séparation

Sommaire.

La syntaxe des commandes de IMP avec tas est définie par la grammaire

$$c ::= x \texttt{ := } a \mid \texttt{skip} \mid c_1 \texttt{ ; } c_2 \mid \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } b \texttt{ do } c$$
$$x \texttt{ := } [a] \mid [x_1] \texttt{ := } a \mid x \texttt{ := alloc}(k) \mid \texttt{free}(a).$$

Les expressions arithmétiques et booléennes restent inchangées.

Ces quatre nouvelles constructions correspondent respectivement à

- ▷ l'accès à une case mémoire ;
- ▷ la modification d'une valeur d'une case mémoire ;
- ▷ l'allocation de k cases mémoires ;
- ▷ la libération de cases mémoires.

Remarque 10.1. C'est un langage impératif de bas niveau : on manipule de la mémoire directement. On ne s'autorise pas tout, cependant. On ne s'autorise pas, par exemple,

$$[x + i + 1] \texttt{ := } [t + i] + [t + i - 1],$$

mais on demande d'écrire

$$x := [t+i] \ ; \ y := [t+i-1] \ ; \ [x+i+1] := x+y.$$

On raffine IMP : avant, les cases vivaient quelque part, mais on ne sait pas où, ce sont des registres ; maintenant, peut aussi allouer des « blocs » de mémoire, et on peut donc parler de cases mémoires adjacentes.

L'allocation `alloc` est *dynamique*, similaire à `malloc` en C, où l'on alloue de la mémoire dans l'espace mémoire appelé *tas*.

10.1 Sémantique opérationnelle de IMP avec tas.

Définition 10.1 (États mémoire)**.** Un état mémoire est la donnée de

- ▷ σ un *registre*, c-à-d un dictionnaire sur $(\mathrm{V}, \mathbb{Z})$;
- ▷ h un *tas*, c-à-d un dictionnaire sur $(\mathbb{N}, \mathbb{Z})$, c'est un gros tableau. [1]

On définit $h[k_1 \mapsto k_2]$ que si $k_1 \in \mathrm{dom}(h)$ et alors il vaut le dictionnaire où l'on assigne k_2 à k_1.

On définit $h_1 \uplus h_2$ que si $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$ et vaut l'union de dictionnaires h_1 et h_2.

On définit ainsi la sémantique dénotationnelle comme la relation $\Downarrow$ est une relation quinaire (*i.e.* avec 5 éléments) notée $c, \sigma, h \Downarrow \sigma', h'$ où c est une commande, h, h' sont deux tas, et σ, σ' sont deux registres.

1. On appelle parfois IMP avec tas, IMP avec tableau.

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

$$\begin{matrix} k' = h(k) \\ \sigma' = \sigma[x \mapsto k'] \end{matrix} \quad \frac{a, \sigma \Downarrow k}{x := [a], \sigma, h \Downarrow \sigma' h} \qquad h' = h[k_1 \mapsto k_2]^{\,2} \quad \frac{a_1, \sigma \Downarrow k_1 \qquad a_2, \sigma \Downarrow k_2}{[a_1] := a_2, \sigma, h \Downarrow \sigma, h'}$$

$$\begin{matrix} \{k', \dots, k'+k-1\} \cap \mathrm{dom}(h) = \emptyset \\ k > 0 \\ h' = h \uplus \{k' \mapsto 0, \dots, k'+k-1 \mapsto 0\} \\ \sigma' = \sigma[x \mapsto k'] \end{matrix} \quad \frac{}{x := \texttt{alloc}(k), \sigma, h \Downarrow \sigma', h'}$$

$$h = h' \uplus \{k \mapsto k'\} \quad \frac{a, \sigma \Downarrow k}{\texttt{free}(a), \sigma, h \Downarrow \sigma, h'.}$$

10.2 Logique de séparation.

Définition 10.2. On définit les assertions dans IMP avec tas comme l'enrichissement des assertions IMP avec les constructions `emp`, $\mapsto$ et $*$:

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

On enrichit ainsi la relation de satisfaction :

- $\sigma, h \models \texttt{emp}$ si et seulement si $h = \emptyset$;
- $\sigma, h \models a_1 \to a_2$ si et seulement si $a_1, \sigma \Downarrow k_1$, et $a_2, \sigma \Downarrow k_2$ et $h = [k_1 = k_2]$ (le tas est un singleton) ;
- $\sigma, h \models A_1 * A_2$ si et seulement si $h = h_1 \uplus h_2$ 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.

Note 10.1 (Notations).
- On note $a \mapsto \_$ pour $\exists i, a \mapsto i$.
- On note $a_1 \hookrightarrow a_2$ pour $(a_1 \mapsto a_2) * \texttt{true}$.

- ▷ On note $a \mapsto (a_1, \ldots, a_n)$ pour

$$(a \mapsto a_1) * (a+1 \mapsto a_2) * \cdots * (a+n-1 \mapsto a_n).$$

- ▷ On note $[A]$ pour $A \wedge \texttt{emp}$.

Ces notations signifient respectivement :

- ▷ La première formule dit qu'une case mémoire est allouée en a (ou plutôt que si a s'évalue en k, alors il y a une case mémoire allouée en k).
- ▷ La seconde formule dit que la case mémoire a_1 est allouée, et contient a_2 quelque part dans le tas.
- ▷ La troisième formule dit que les n cases mémoires suivant a contiennent respectivement a_1, puis a_2, *etc.*
- ▷ La quatrième formule permet de ne pas parler du tas. Intuitivement A « observe » uniquement la composante σ et alors A est une formule de la logique de Hoare.

10.3 Triplets de Hoare pour la logique de séparation.

On rappelle que $\{A\}\{A'\}$ est définir avec A, A' closes. La validité d'un triplet de Hoare en logique de séparation est défini par $\models \{A\}c\{A'\}$ si et seulement si dès que $\sigma, h \models A$ et $c, \sigma, h \Downarrow \sigma', h'$ alors $\sigma', h' \models A'$.

Parmi les règles d'inférences pour la logique de séparation, il y a la « *frame rule* » ou « *règle d'encadrement* » :

$$\underset{\substack{\text{aucune variable } x \in \mathsf{vars}(B) \\ \text{n'est modifiée par } c}}{} \frac{\vdash \{A\}c\{A'\}}{\vdash \{A * B\}c\{A' * B\}.}$$

Elle permet de *zoomer*, et de ce concentrer sur un comportent local.

Les règles suivantes définissent une logique sur les commandes de IMP

avec tas.

$$\frac{}{\vdash \{\texttt{emp}\} x := \texttt{alloc}(k) \{x \mapsto (0, \ldots, 0)\}}$$

$$\frac{}{\vdash \{a \mapsto \_\}\texttt{free}(a)\{\texttt{emp}\}} \qquad \frac{}{\vdash \{a_1 \mapsto \_\}[a_1] := a_2 \{a_1 \mapsto a_2\}}$$

$$\frac{}{\vdash \{[x = x_0] * (a \mapsto x_1)\} x := [a] \{[x = x_1] * (a[^{x_0}/_x] \mapsto x_1)\}}$$

11 Réécriture.

Sommaire.

Définition 11.1. Soit $\to$ une relation binaire sur un ensemble E. Le 2-uplet $(E, \to)$ est un *SRA*, pour *système de réécriture abstraite*.

Soit $x_0 \in E$. Une *divergence* issue de x_0 est une suite $(x_i)_{i\in\mathbb{N}}$ telle que, pour tout i, on a $x_i \to x_{i+1}$.

La relation $\to$ est *terminante* ou *termine* si et seulement si, quel que soit $x \in E$, il n'y a pas de divergence issue de x.

La relation $\to$ diverge s'il existe une divergence.

Exemple 11.1. En général, une relation réflexive est divergente.

Théorème 11.1. Une relation $(E, \to)$ est terminante si et seulement si elle satisfait le *principe d'induction bien fondée (PIBF)* suivant :

Pour tout prédicat $\mathscr{P}$ sur E, si pour tout $x \in E$

$$\Big[\forall y \in E, x \to y \text{ implique } \mathscr{P}(y)\Big] \text{ implique } \mathscr{P}(x)$$

alors, pour tout $x \in E$, $\mathcal{P}(x)$.

En particulier, dans le principe d'induction bien fondée, on demande que les feuilles (les éléments sans successeurs) vérifient le prédicat.

Preuve. ▷ « PIBF $\implies$ terminaison ». Montrons que, quel que soit $x \in E$,

$$\mathcal{P}(x) \ : \ \text{« il n'y a pas de divergence issue de } x \text{ »}.$$

Soit $\text{Next}(x) = \{y \in E \mid x \to y\}$. On suppose que, pour tout $y \in \text{Next}(x)$, on a $\mathcal{P}(y)$. On en déduit $\mathcal{P}(x)$ car, sinon, une divergence ne passerait pas par $y \in \text{Next}(x)$. Par le principe d'induction bien fondée, on en déduit

$$\forall x \in E, \mathcal{P}(x),$$

autrement dit, la relation $\to$ termine.

▷ « ¬PIBF $\implies$ diverge », par contraposée. On suppose qu'il existe un prédicat $\mathcal{P}$ tel que,

$$\forall x, (\forall y, x \to y \text{ implique } \mathcal{P}(y)) \text{ implique } \mathcal{P}(x),$$

et que l'on n'ait pas, $\forall x \in E, \mathcal{P}(x)$ autrement dit qu'il existe $x_0 \in E$ tel que $\neg\mathcal{P}(x)$.

Intéressons-nous à $\text{Next}(x_0) = \{y \in E \mid x_0 \to y\}$. Si, pour tout $y \in \text{Next}(x_0)$ on a $\mathcal{P}(y)$ alors par hypothèse $\mathcal{P}(x_0)$, ce qui est impossible. Ainsi, il existe $x_1 \in \text{Next}(x_0)$ tel que $\neg\mathcal{P}(x_1)$. On itère ce raisonnement, ceci crée notre divergence.

□

Remarque 11.1. L'induction bien fondée s'appelle aussi l'induction *noethérienne*, en référence à Emmy Noether, mathématicienne allemande du IX–Xème siècle.

Une application de ce principe d'induction est le *lemme de König*.

Définition 11.2.
- ▷ Un arbre est *fini* s'il a un nombre fini de nœuds (*infini* sinon).
- ▷ Un arbre est à *branchement fini* si tout nœud a un nombre fini d'enfants immédiats.
- ▷ Une branche est *infinie* si elle contient un nombre infini de nœuds.

Lemme 11.1 (Lemme de König). Si un arbre est à branchement fini est infini alors il contient une branche infinie.

Preuve. On considère E l'ensemble des nœuds de l'arbre, et on définit la relation $\to$ par : on a $x \to y$ si y est enfant immédiat de x. On montre qu'un arbre à branchement fini sans branche infinie (*i.e.* la relation $\to$ termine) est fini. On choisit la propriété $\mathcal{P}(x)$: « le sous-arbre enraciné en x est fini. »

Montrons que, quel que soit x, $\mathcal{P}(x)$ et pour ce faire, utilisons le principe d'induction bien fondée puisque la relation $\to$ termine. On doit montrer que, si $\forall y \in \mathrm{Next}(x), \mathcal{P}(y)$ implique $\mathcal{P}(x)$. Ceci est vrai car l'embranchement est fini. □

11.1 Liens avec les définitions inductives.

On considère E l'ensemble inductif défini par la grammaire suivante :

$$t ::= \mathtt{F} \mid \mathtt{N}(t_1, k, t_2).$$

C'est aussi le plus petit point fixe de l'opérateur f associé (par le théorème de Knaster–Tarski).

On définit la relation $\to$ binaire sur E par : on a $x \to y$ si et seulement si on a $x = \mathtt{N}(y, k, z)$ ou $x = \mathtt{N}(z, k, y)$.

On sait que la relation $\to$ termine. En effet, l'ensemble des arbres finis est un point fixe de la fonction f, donc E ne contient que des arbres

finis.

Le principe d'induction bien fondée nous dit que, pour $\mathscr{P}$ un prédicat sur E, pour montrer $\forall x, \mathscr{P}(x)$, il suffit de montrer que, quel que soit x, si $(\forall y, x \to y$ implique $\mathscr{P}(y))$ alors $\mathscr{P}(x)$. Autrement dit, il suffit de montrer que $\mathscr{P}(\mathtt{E})$ puis de montrer que, si $\mathscr{P}(t_1)$ et $\mathscr{P}(t_2)$ alors on a que $\mathscr{P}(\mathtt{N}(t_1, k, t_2))$.

On retrouve le principe d'induction usuel.

Ce même raisonnement, on peut le réaliser quel que soit l'ensemble inductif, car la relation de « sous-élément » termine toujours puisque il n'y a que des éléments finis dans l'ensemble inductif.

11.2 Établir la terminaison.

Théorème 11.2. Soient $(B, >)$ un SRA terminant, et $(A, \to)$ un SRA. Soit $\varphi : A \to B$ un *plongement*, c'est à dire une application vérifiant

$$\forall a, a' \in A, \qquad a \to a' \text{ implique } \varphi(a) > \varphi(a').$$

Alors, la relation $\to$ termine.

Théorème 11.3. Soient $(A, \to_A)$ et $(B, \to_B)$ deux SRA.

Le *produit lexicographique* de $(A, \to_A)$ et $(B, \to_B)$ est le SRA, que l'on notera $(A \times B, \to_{A\times B})$, défini par

$$(a, b) \to_{A\times B} (a', b') \text{ ssi } \begin{cases} (1)\ a \to_A a' \text{ (et } b' \text{ quelconque)} \\ \qquad \text{ou} \\ (2)\ a = a' \text{ et } b \to_B b' \end{cases}.$$

Alors, les relations $(A, \to_A)$ et $(B, \to_B)$ terminent si et seulement si la relation $(A \times B, \to_{A\times B})$ termine.

Preuve. ▷ « $\Longrightarrow$ ». Supposons qu'il existe une divergence pour $(A \times B, \to_{A\times B})$:

$$(a_0, b_0) \to_{A\times B} (a_1, b_1) \to_{A\times B} (a_2, b_2) \to_{A\times B} \cdots .$$

Dans cette divergence,

- soit on a utilisé (1) une infinité de fois, et alors en projetant sur la première composante et en ne conservant que les fois où l'on utilise (1), on obtient une divergence $\to_A$;
- soit on a utilisé (1) un nombre fini de fois, et alors à partir d'un certain rang N, pour tout $i \geq N$, on a l'égalité $a_i = a_N$, et donc on obtient une divergence pour $\to_B$:

$$b_N \to_B b_{N+1} \to_B b_{N+2} \to \cdots .$$

▷ « $\Longleftarrow$ ». On montre que, si on a une divergence pour $\to_A$ alors on a une divergence pour $\to_{A\times B}$ (on utilise (1) une infinité de fois) ; puis que si on a une divergence pour $\to_B$ alors on a une divergence pour $\to_{A\times B}$ (on utilise (2) une infinité de fois).

□

11.3 Application à l'algorithme d'unification.

On note $(\mathscr{P}, \sigma) \to (\mathscr{P}', \sigma')$ la relation définie par l'algorithme d'unification (on néglige le cas où $(\mathscr{P}, \sigma) \to \bot$).

On note $|\mathscr{P}|$ la somme des tailles (vues comme des arbres) des contraintes de $\mathscr{P}$ et $|\mathsf{Vars}\ \mathscr{P}|$ le nombre de variables.

On définit $\varphi : (\mathscr{P}, \sigma) \mapsto (|\mathsf{Vars}\ \mathscr{P}|, |\mathscr{P}|)$.

Rappelons la définition de la relation $\to$ dans l'algorithme d'unification :

1. $(\{f(t_1,\dots,t_k) \stackrel{?}{=} f(u_1,\dots,u_n) \sqcup \mathscr{P}, \sigma\}) \to (\{t_1 \stackrel{?}{=} u_1,\dots,t_k \stackrel{?}{=} u_k\} \cup \mathscr{P}, \sigma)$;
2. $(\{f(t_1,\dots,t_k) \stackrel{?}{=} g(u_1,\dots,u_n) \sqcup \mathscr{P}, \sigma\}) \to \bot$ si $f \neq g$;
3. $(\{X \stackrel{?}{=} t\} \sqcup \mathscr{P}, \sigma) \to (\mathscr{P}[t/X], [t/X] \circ \sigma)$ où $X \notin \mathsf{Vars}(t)$;
4. $(\{X \stackrel{?}{=} t\} \sqcup \mathscr{P}, \sigma) \to \bot$ si $X \in \mathsf{Vars}(t)$ et $t \neq X$;
5. $(\{X \stackrel{?}{=} X\} \sqcup \mathscr{P}, \sigma) \to (\mathscr{P}, \sigma)$.

Appliquons le plongement pour montrer que $\to$ termine. On s'appuie sur le fait que le produit $(\mathbb{N}, >) \times (\mathbb{N}, >)$ est terminant (produit lexicographique).

Dans 1, $|\mathsf{Vars}\ \mathscr{P}|$ ne change pas et $|\mathscr{P}|$ diminue. Puis dans 3, $|\mathsf{Vars}\ \mathscr{P}|$ diminue. Et dans 5, on a $|\mathsf{Vars}\ \mathscr{P}|$ qui décroit ou ne change pas, mais $|\mathscr{P}|$ diminue. Dans les autres cas, on arrive, soit sur $\bot$.

On en conclut que l'algorithme d'unification termine.

11.4 Multiensembles.

Définition 11.3. Soit A un ensemble. Un *multiensemble* sur A est la donnée d'une fonction $M : A \to \mathbb{N}$. Un multiensemble M est *fini* si $\{a \in A \mid M(a) > 0\}$ est fini.

Le multiensemble vide, noté $\emptyset$, vaut $a \mapsto 0$.

Pour deux multiensembles M_1 et M_2 sur A, on définit

- $(M_1 \cup M_2)(a) = M_1(a) + M_2(a)$;
- $(M_1 \ominus M_2)(a) = M_1(a) \ominus M_2(a)$ où l'on a $(n+k) \ominus n = k$ mais $n \ominus (n+k) = 0$.

On note $M_1 \subseteq M_2$ si, pour tout $a \in A$, on a $M_1(a) \leq M_2(a)$.

La *taille* de M est $|M| = \sum_{a \in A} M(a)$.

On note $x \in M$ dès lors que $x \in A$ et que $M(x) > 0$.

Exemple 11.2. Si on lit $\{1,1,1,2,3,4,3,5\}$ comme un multiensemble M, on obtient que $M(1)=3$, et $M(2)=1$, et $M(3)=2$, et $M(4)=1$, et $M(5)=1$, et finalement pour tout autre entier n, $M(n)=0$.

Définition 11.4 (Extension multiensemble.)**.** Soit $(A,>)$ un SRA. On lui associe une relation notée $>_{\text{mul}}$ définie sur les multiensembles *finis* sur A en définissant $M >_{\text{mul}} N$ si et seulement s'il existe X, Y deux multiensembles sur A tels que

- $\emptyset \neq X \subseteq M$;
- $N = (M \ominus X) \cup Y$ [1]
- $\forall y \in Y, \exists x \in X, x > y$.

Les multiensembles X et Y sont les « témoins » de $M >_{\text{mul}} N$.

Exemple 11.3. Dans $(\mathbb{N},>)$, on a

$$\{1,2,\underbrace{5}_{X}\} >_{\text{mul}} \{1,2,\underbrace{4,4,4,4,3,3,3,3,3}_{Y}\}.$$

Théorème 11.4. La relation $>$ termine si et seulement si $>_{\text{mul}}$ termine.

Preuve.
- « $\Longleftarrow$ ». Une divergence de $>$ induit une divergence de $>_{\text{mul}}$.
- « $\Longrightarrow$ ». On se donne une divergence pour $>_{\text{mul}}$:

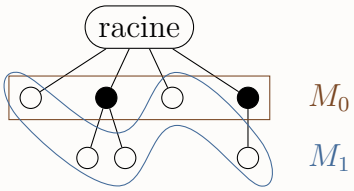
$$M_0 >_{\text{mul}} M_1 >_{\text{mul}} M_2 >_{\text{mul}} \cdots,$$

et on montre que $>$ diverge À chaque $M_i >_{\text{mul}} M_{i+1}$ correspondent X_i et Y_i suivant la définition de $>_{\text{mul}}$.

1. C'est ici la soustraction usuelle : il n'y a pas de soustraction qui « pose problème ».

On sait qu'il y a une infinité de i tel que $Y_i \neq \emptyset$. En effet, si au bout d'un moment Y_i est toujours vide alors $|M_i|$ décroit strictement, impossible.

Représentons cela sur un arbre.



On itère le parcours en obtenant un arbre à branchement fini, qui est infini (observation du dessin) donc par le lemme de König il a une branche infinie. Par construction d'enfant de a correspond à $a > a'$, d'où divergence pour $>$.

□