Théorie de la programmation

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

ENS DE LYON

8 novembre 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 $\mathtt{C}_1, \dots, \mathtt{C}_k$, 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 $\mathtt{t}$ option est alors défini par $1 + \mathtt{t}$.

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 $\forall x \in \mathsf{nat}, \mathscr{P}(x)$, il suffit de prouver *deux* choses (parce que l'on a deux constructeurs à l'ensemble nat) :

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

Remarque 1.1. Dans le cas général, pour prouver $\forall x \in \mathsf{t}, \mathscr{P}(x)$, 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 $\mathscr{P}(\mathtt{C}_i(x_1, \ldots, x_n))$ avec les hypothèses d'inductions $\mathscr{P}(x_j)$ lorsque $\mathtt{u}_j^i = \mathsf{t}$.

Exemple 1.5. Avec le type 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 $\mathscr{P}(x_1)$ et $\mathscr{P}(x_3)$ comme hypothèses d'induction, et on montre $\mathscr{P}(\mathtt{N2}(x_1, k, x_3))$, où l'on se donne $k \in \mathsf{nat}$.

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

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

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

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

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

et on veut montrer que

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

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 représente $\leq$, « *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 $\mathsf{triée}$ sur nlist. Pour cela, on pose les trois règles ci-dessous :

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

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

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

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

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 $\textsf{mem} \subseteq \textsf{nat} * \textsf{nlist}$ par les règles d'inférences :

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

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

$$\textsf{mem}(0, \texttt{[0;1;0]}).$$

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 « $\neg\textsf{r}(\dots)$ ». 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 :

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

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 :

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

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 :

$$\frac{H_1 \qquad \cdots \qquad H_n}{\mathsf{Rel}(x_1, \dots, x_m)}\ \mathcal{R}_i,$$

où chaque H_j est :

- ▷ soit $\mathsf{Rel}(\dots)$;

- 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 $\mathsf{Rel}(x_1, \dots, x_m)$ 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

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

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 $\mathsf{Rel}(\dots)$.

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 $\mathscr{P}(n, \mathsf{S}\ k)$ en supposant $\mathscr{P}(n, k)$.

Exemple 1.13. Supposons que l'on ait une fonction ayant pour signature $\texttt{sort} : \mathsf{nlist} \to \mathsf{nlist}$ qui trie une nlist. On souhaite démontrer la propriété :

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

On considère deux approches pour la démonstration : par induction sur ℓ et par induction sur la relation $\mathsf{triée}$.

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

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 :

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

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

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

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

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

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

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 $\texttt{t}_2$ 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 :

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

puis, quel que soit A,

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

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

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

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

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

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

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

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

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 $k \in$ 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 $\mathscr{R}_1, \dots, \mathscr{R}_k$. On veut montrer que

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

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

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

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

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

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

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

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

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

Alors, $\operatorname{fix}(f)$ 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

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

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 $\mathscr{P}$ 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 $\texttt{l} = \texttt{Nil}$. Pas trop de problèmes pour prouver que `[a;b]` est triée avec l'hypothèse $\texttt{a} \leq \texttt{b}$. 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 :

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

  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` ≤ `b`, mais aucune hypothèse de la forme `b` ≤ `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.

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.

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

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.

4.1.1 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 $\Downarrow \subseteq \textsf{EA} * \mathbb{Z}$, 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 :

$$\text{Cond. App.} \ \frac{P_1 \qquad \ldots \qquad P_m}{C} \ \mathscr{R}_i$$

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

Exemple 4.2.

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

4.1.2 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 $a \mathrel{\mathscr{R}} b$ et $a \mathrel{\mathscr{R}} c$ alors $b = c$.

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

$$\dfrac{}{\underline{k_1} \oplus \underline{k_2} \to \underline{k}}\ \mathscr{A} \quad {\scriptstyle k = k_1 + k_2},$$

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

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.

4.1.3 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 :

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

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, $a \mathscr{R}^\star z$. Ceci nous donne $x \mathscr{R}^\star z$ 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 :
   $$a_1 \oplus a_2 \xrightarrow{(H_2)+\text{lemme } 4.2}{}^\star\, a_1 \oplus \underline{k}_2 \xrightarrow{(H_1)+\text{lemme } 4.3}{}^\star\, \underline{k}_1 \oplus \underline{k}_2 \xrightarrow{\mathcal{A}} \underline{k}.$$

□

Proposition 4.2. Soient a une expression et k un entier. On a l'implication
$$a \to^\star \underline{k} \implies a \Downarrow k.$$

□

4.1.4 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 !)

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 :

$$\frac{}{\underline{k} \Downarrow k} \qquad {\scriptstyle k = k_1 + k_2 \atop k \neq 0}\ \frac{a_1 \Downarrow k_1 \qquad a_2 \Downarrow k_2}{a_1 \oplus a_2 \Downarrow k} \qquad \frac{a_1 \Downarrow k_1 \qquad a_2 \Downarrow 0}{a_1 \oplus a_2 \Downarrow \bot},$$

et les règles de propagation du $\bot$:

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

Relation à petits pas.

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

$$ {\scriptstyle k = k_1 + k_2 \atop k_2 \neq 0}\ \frac{}{\underline{k}_1 \oplus \underline{k}_2 \to \underline{k}} \qquad {\scriptstyle a_2 \neq \bot}\ \frac{a_2 \to a_2'}{a_1 \oplus a_2 \to a_1 \oplus a_2'}$$

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

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...

4.1.5 Sémantique contextuelle pour EA.

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

$$ {\scriptstyle k = k_1 + k_2}\ \frac{}{E[\underline{k}_1 \oplus \underline{k}_2] \mapsto E[\underline{k}]},$$

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 $\boxed{?}$?

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 :

- « $\implies$ » par induction sur $a \to a'$;
- « $\impliedby$ » par induction sur E.

□

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

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 4.2 | *Définition inductive de* LEA

Exemple 4.7. Voici quelques exemples d'expressions avec déclarations locales :

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

4.2.1 Sémantique à grands pas sur LEA.

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

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

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

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

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

Définition 4.1. Soit $a \in \mathsf{LEA}$. L'ensemble des *variables libres* d'une expression a noté $\mathscr{V\ell}(a)$, et est défini par induction sur a de la manière suivante :

- ▷ $\mathscr{V\ell}(\underline{k}) = \emptyset$;
- ▷ $\mathscr{V\ell}(x) = \{x\}$;

- ▷ $\mathscr{V}\ell(a_1 \oplus a_2) = \mathscr{V}\ell(a_1) \cup \mathscr{V}\ell(a_2)$;
- ▷ $\mathscr{V}\ell(\texttt{let}\, x = a_1 \texttt{ in } a_2) = \mathscr{V}\ell(a_1) \cup (\mathscr{V}\ell(a_2) \setminus \{x\})$.

Exemple 4.8.

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

Définition 4.2. Une expression $a \in \mathsf{LEA}$ est *close* si $\mathscr{V}\ell(a) = \emptyset$. On note $\mathsf{LEA}_0 \subseteq \mathsf{LEA}$ l'ensemble des expressions arithmétiques de closes.

Définition 4.3. Soient $a \in \mathsf{LEA}$, $x \in \mathscr{V}$ 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 :

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

4.2.2 Sémantique à petits pas sur LEA.

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

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

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

1. On surcharge encore les notations.

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

$$\frac{a_1 \to a_1'}{\texttt{let}\, x \texttt{ = } a_1 \texttt{ in } a_2 \to \texttt{let}\, x \texttt{ = } a_1' \texttt{ in } a_2}\ \mathscr{C}_1$$

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

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

4.2.3 Sémantique contextuelle pour LEA.

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

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

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

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

et

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

4.2.4 Sémantique sur LEA avec environnement.

Définition 4.4. 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

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

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 $z \in \operatorname{dom}(D)$ 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 $(\mathscr{V}, \mathbb{Z})$.

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

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

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

Remarque 4.4.
- ▷ Dans cette définition, on n'a pas de substitutions (c'est donc plus facile à calculer).
- ▷ Si $\mathscr{V}\ell(a) \subseteq \operatorname{dom}(\mathscr{E})$, 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.)

4.3 Un petit langage fonctionnel : FUN.

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 4.2. 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.

4.3.1 Sémantique opérationnelle « informellement ».

Exemple 4.9. 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 4.10. 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 4.5. 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.

4.3.2 Sémantique opérationnelle de FUN (version 1).

Définition 4.5. On définit l'ensemble des *variables libres* $\mathscr{V}\ell(e)$ d'une expression e par (on a 5 cas) :

- $\mathscr{V}\ell(x) = \{x\}$;
- $\mathscr{V}\ell(k) = \emptyset$;
- $\mathscr{V}\ell(e_1 + e_2) = \mathscr{V}\ell(e_1) \cup \mathscr{V}\ell(e_2)$;
- $\mathscr{V}\ell(e_1\ e_2) = \mathscr{V}\ell(e_1) \cup \mathscr{V}\ell(e_2)$;
- $\mathscr{V}\ell(\texttt{fun}\, x \to e) = \mathscr{V}\ell(e) \setminus \{x\}$.[2]

On dit que e est *close* si $\mathscr{V}\ell(e) = \emptyset$.

Définition 4.6. 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 :

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

Grands pas pour FUN.

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

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

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

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

Remarque 4.6. Certaines expressions ne s'évaluent pas :

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

par exemple.

Petits pas pour FUN.

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

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

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

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

Remarque 4.7. Il existe des expressions que l'on ne peut pas réduire :

1. $k \not\to$;
2. $(\texttt{fun}\, x \to x) \not\to$;
3. $e_1 + (\texttt{fun}\, x \to x) \not\to$;
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 4.5. On a

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

Remarque 4.8. Soit $e_0 = (\texttt{fun}\, x \to x\ x)\ (\texttt{fun}\, x \to x\ x)$. 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 [3] définie par $\Downarrow$ est donc partielle.

Remarque 4.9 (Problème avec la substitution)**.** On a la chaîne de réductions :

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

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 4.10. Les relations $\Downarrow$ et $\to$ sont définies sur des expressions **closes**. Et on a même $\to \subseteq \mathsf{FUN}_0 * \mathsf{FUN}_0$. [4]

Lemme 4.6.
- ▷ Si v est close et si $x \notin \mathscr{V}\ell(e)$ alors $e[{}^v/_x] = e$.
- ▷ Si v est close, $\mathscr{V}\ell(e[{}^v/_x]) = \mathscr{V}\ell(e) \setminus \{x\}$. □

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

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

Lemme 4.7. 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 $(\mathtt{fun}\, x \to e)\ v$ est close, alors
   - ▷ $(\mathtt{fun}\, x \to e)$ 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.

□

Remarque 4.11. 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 4.7 (Définition informelle de l'α-conversion)**.** On définit l'α-conversion, notée $e =_\alpha e'$: on a $\mathtt{fun}\, x \to e =_\alpha \mathtt{fun}\, y \to e'$ si, et seulement si, e' s'obtient en replaçant x par y dans e à condition que $y \notin \mathscr{V}\ell(e)$.[5]

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

Exemple 4.11 (*Les variables liées sont muettes.*)**.** On a :

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

5. C'est une « variable fraîche ».

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

"Lemme" 4.1. Si $E_0 \subseteq \mathscr{V}$ est un ensemble fini de variables, alors il existe $z \notin E_0$ et $e' \in$ FUN tel que $\texttt{fun}\, x \texttt{ -> } e =_\alpha \texttt{fun}\, z \texttt{ -> } e'$. □

Remarque 4.12 (Fondamental). En fait FUN désigne l'ensemble des expressions décrites par la grammaire initiale *quotientée* par α-conversion.

Remarque 4.13. On remarque que

$$(e =_\alpha e') \implies \mathscr{V\ell}(e) = \mathscr{V\ell}(e').$$

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

Définition 4.8. Pour $e \in$ 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 $x \neq y$;
- ▷ $(e_1 + e_2)[{}^v/_x] = (e_1[{}^v/_x]) + (e_2[{}^v/_x])$;
- ▷ $(e_1\ e_2)[{}^v/_x] = (e_1[{}^v/_x])\ (e_2[{}^v/_x])$.

4.3.3 Ajout des déclarations locales (FUN + let).

On ajoute les déclarations locales (comme pour EA $\to$ 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 :

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

et

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

Attention ! On n'a pas de règle

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

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 $\mathsf{FUN} + \texttt{let}$.

Traduction de $\mathsf{FUN} + \texttt{let}$ vers FUN.

On définit une fonction qui, à toute expression de e dans $\mathsf{FUN} + \texttt{let}$ 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 :

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

Lemme 4.8. Pour tout $e \in (\mathsf{FUN} + \texttt{let})$,

- $[\![e]\!]$ est une expression de FUN [6] ;
- on a $\mathscr{V}\!\ell([\![e]\!]) = \mathscr{V}\!\ell(e)$;
- $[\![e]\!]$ est une valeur *ssi* e est une valeur ;
- $[\![e[{}^{v}/_{x}]]\!] = [\![e]\!]\,[{}^{[\![v]\!]}/_{x}]$ [7]. □

Pour démontrer le lemme 4.8, on procède par induction sur e. C'est long et rébarbatif, mais la proposition ci-dessous est bien plus intéressante.

Proposition 4.4. 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* $\mathscr{R}_{\mathrm{lv}}$. Il faut montrer que $[\![\texttt{let}\, x \texttt{ = } v \texttt{ in } e_2]\!] \to_{\mathsf{FUN}} [\![e[{}^{v}/_{x}]]\!]$. Par définition, l'expression de droite vaut
$$(\texttt{fun}\, x \texttt{ -> } [\![e]\!]_2)\ \ [\![v]\!] \xrightarrow{\mathscr{R}_\beta}_{\mathsf{FUN}} [\![e]\!]_2\,[{}^{[\![v]\!]}/_{x}],$$
car $[\![v]\!]$ est une valeur par le lemme 4.8, ce qui justifie $\mathscr{R}_\beta$. De plus, encore par le lemme 4.8, on a l'égalité entre $[\![e]\!]_2\,[{}^{[\![v]\!]}/_{x}] = [\![e[{}^{v}/_{x}]]\!]$.
- *Cas* $\mathscr{R}_{\mathrm{lg}}$. On sait que $e_1 \to e_1'$ et, par hypothèse d'induction,

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

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

$$(\texttt{fun}\, x \to [\![e_2]\!])\ [\![e_1]\!] \xrightarrow{\mathscr{R}_{\mathrm{ad}}\ \&\ \text{hyp. ind.}} (\texttt{fun}\, x \to [\![e_2]\!])\ [\![e'_1]\!]\,.$$

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.

□

Proposition 4.5. 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 $\mathsf{FUN} + \texttt{let}$) »,

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

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

▷ *Cas de la règle $\mathscr{R}_{\mathrm{ad}}$.* 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.

- *1^er^ 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 $\mathscr{R}_{\mathrm{ad}}$ 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,

$$\llbracket e \rrbracket = \underbrace{(\texttt{fun}\, x \texttt{ -> } \llbracket e_2 \rrbracket)}_{f_1} \; \underbrace{\llbracket e_1 \rrbracket}_{f_2}.$$

Par hypothèse d'induction, il existe e_1' tel que $e_1 \to e_1'$ et $\llbracket e_1' \rrbracket = f_2'$. Posons $e' = (\texttt{let}\, x \texttt{ = } e_1' \texttt{ in } e_2)$. On doit vérifier $\llbracket e \rrbracket \to \llbracket e' \rrbracket$ ce qui est vrai par $\mathscr{R}_{\text{ad}}$ et que $\llbracket e' \rrbracket = f_1\ f_2'$, 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 $\llbracket e \rrbracket = f_1\ v$ et on a deux sous-cas.

– *1er sous-cas.* On suppose $e = e_1\ e_2$ et alors $\llbracket e \rrbracket = \llbracket e_1 \rrbracket\ \llbracket e_2 \rrbracket$ par le lemme 4.8 et parce que e_2 est une valeur (car $\llbracket e_2 \rrbracket = v$). 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 $e = (\texttt{let}\, x \texttt{ = } e_1 \texttt{ in } e_2)$ alors

$$\llbracket e \rrbracket = \underbrace{\texttt{fun}\, x \texttt{ -> } \llbracket e_2 \rrbracket}_{f_1} \; \underbrace{\llbracket e_1 \rrbracket}_{f_2}.$$

On vérifie aisément ce que l'on doit montrer.

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

□

4.4 Typage en FUN.

4.4.1 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 4.3. 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 4.9. Un *environnement de typage*, noté $\Gamma, \Gamma_1, \Gamma', \dots$, est un dictionnaire sur $(\mathscr{V}, \mathsf{Typ})$, 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 $x \notin \operatorname{dom} \Gamma$. [8]

Remarque 4.14. On peut voir/implémenter Γ comme des listes finies de couples (x, τ).

Définition 4.10. 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.

$$\frac{}{\Gamma \vdash k : \mathtt{int}}\ \mathscr{T}_{\mathrm{k}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \mathscr{T}_{\mathrm{v}} \qquad \frac{\Gamma, x : \tau_1{}^{9} \vdash e_2}{\Gamma \vdash \mathtt{fun}\, x \to e : \tau_1 \to \tau_2}\ \mathscr{T}_{\mathrm{f}}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{int} \qquad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash e_1 + e_2 : \mathtt{int}}\ \mathscr{T}_{\mathrm{p}} \qquad \frac{\Gamma \vdash e : \tau_1 \to \tau_2 \qquad \Gamma \vdash e' : \tau_1}{\Gamma \vdash e\ e' : \tau_2}\ \mathscr{T}_{\mathrm{a}}.$$

Remarque 4.15. Pour l'instant, on parle uniquement d'expressions et pas du tout de valeurs ou de sémantique opérationnelle.

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

Remarque 4.16.
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 4.12.
1. L'expression $\texttt{fun}\, x \to x$ est particulière : on peut la typer avec $\tau \to \tau$ quel que soit τ. Par exemple,

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

   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 $\texttt{fun}\, g \to g\ (g\ 7)$?

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

4.4.2 Propriétés du système de types.

Lemme 4.9.
- Si $\Gamma \vdash e : \tau$ alors $\mathscr{V}\ell(e) \subseteq \mathrm{dom}(\Gamma)$.
- *Affaiblissement.* Si $\Gamma \vdash e : \tau$ alors

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

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

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

4.4.3 Propriété de progrès.

Lemme 4.10.
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 $e = \texttt{fun}\, x \texttt{ -> } e_0$.

Preuve. Vu en TD. □

Proposition 4.6 (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 4.17.

- ▷ 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 4.18. Par le typage, on a exclu les expressions bloquées car « mal formées » (*e.g.* $3\ 2$ ou $3 + (\texttt{fun}\, x \texttt{ -> } x)$).

4.4.4 Propriété de préservation.

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

Lemme 4.11 (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.
  - *1^er^ 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.

– *2^{nd} 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 4.7 (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 $\mathcal{T}_\mathrm{v}$.* C'est absurde ! (On n'a pas $\emptyset \vdash x : \tau$.)

▷ *Cas $\mathcal{T}_\mathrm{f}$.* Si $(\texttt{fun}\, x \to e) \to e'$ alors ... On peut conclure immédiatement car les fonctions sont des valeurs, elles ne se réduisent donc pas.

▷ *Cas $\mathcal{T}_\mathrm{k}$.* C'est le même raisonnement.

▷ *Cas $\mathcal{T}_\mathrm{a}$.* 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.

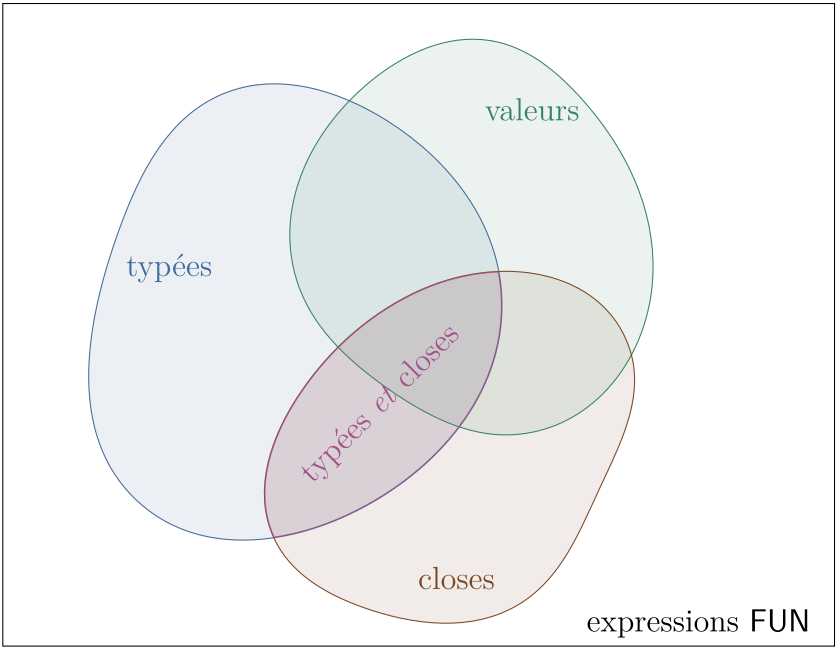
– *Sous-cas $\mathcal{R}_\mathrm{ad}$.* Cela veut dire que $e_2 \to e_2'$ et $e' = e_1\ e_2'$. On conclut $\emptyset \vdash e_1\ e_2' : \tau$ par (H_2') et (H_1).
– *Sous-cas $\mathcal{R}_\mathrm{ag}$.* Cela veut dire que $e_1 \to e_1'$ et $e' = e_1'\ e_2$. On conclut $\emptyset \vdash e_1'\ e_2 : \tau$ par (H_1') et (H_2).
– *Sous-cas $\mathcal{R}_\beta$.* On a $e_1 = \texttt{fun}\, x \to e_0$, $e_2 = v$ et finalement $e' = e_0[{}^v/_x]$. On doit montrer $\emptyset \vdash e_0[{}^v/_x] : \tau$. De plus, (H_1) s'énonce par $\emptyset \vdash \texttt{fun}\, x \to e_0 : \tau_0 \to \tau$. 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* $\mathcal{T}_\mathrm{p}$. Laissé en exercice.

$\square$

Remarque 4.19. 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 4.2. 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.

4.4.5 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[10])

4.4.6 Inférence de types.

Typage et contraintes.

Exemple 4.13. 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 $\texttt{int} \to ?$.

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

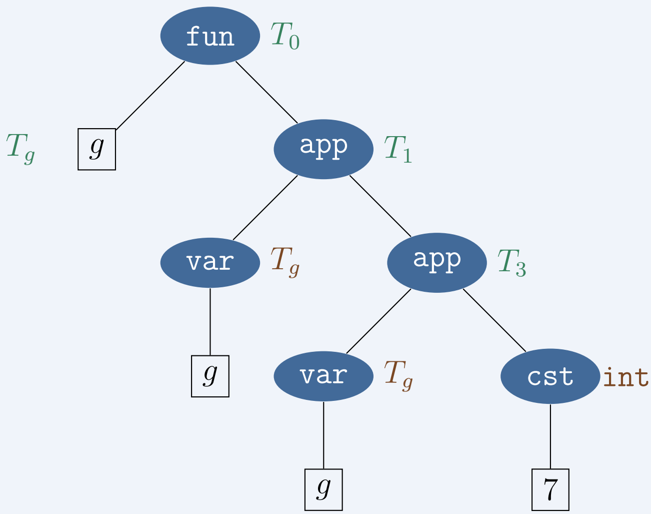
On doit donc lire le programme, et « prendre des notes ». Ces « notes » sont des contraintes que doivent vérifier le programme.

Exemple 4.14. On souhaite déterminer le type τ tel que

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

(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).

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

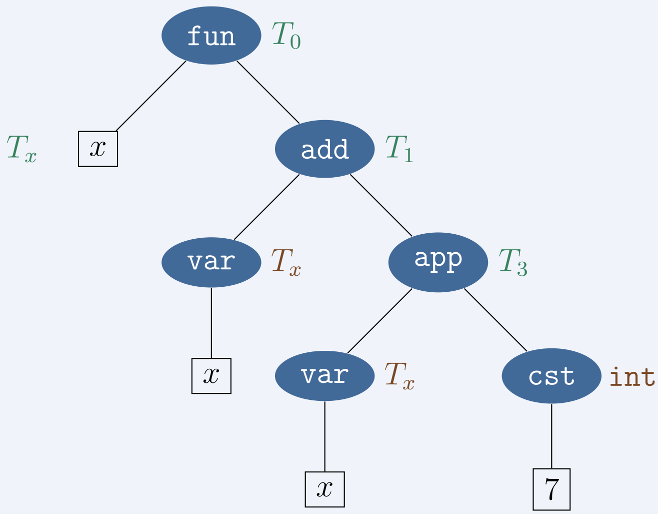
3. On résout les contraintes pour obtenir

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

Exemple 4.15 (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 $\texttt{int}$ mais admet une infinité de dérivations.

Exemple 4.16 (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 4.11. ▷ 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 4.12. Une *contrainte de typage* est une paire de types étendus, notée $\hat{\tau}_1 \overset{?}{=} \hat{\tau}_2$, 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 $\mathscr{V}\ell(e) \subseteq \operatorname{dom}(\widehat{\Gamma})$. On choisit $T \in \text{IType}$.

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

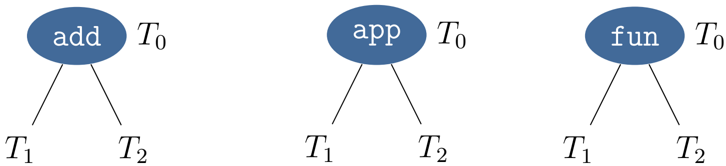
▷ $$\begin{aligned}\mathsf{CT}(e_1 + e_2, \widehat{\Gamma}, T) = \;& \mathsf{CT}(e_1, \widehat{\Gamma}, T_1) \cup \mathsf{CT}(e_2, \widehat{\Gamma}, T_2) \\ & \cup \{T_1 \overset{?}{=} \texttt{int}, T_2 \overset{?}{=} \texttt{int}, T \overset{?}{=} \texttt{int}\}\end{aligned}$$

▷ $$\begin{aligned}\mathsf{CT}(e_1\ e_2, \widehat{\Gamma}, T) = \;& \mathsf{CT}(e_1, \widehat{\Gamma}, T_1) \cup \mathsf{CT}(e_2, \widehat{\Gamma}, T_2) \\ & \cup \{T_1 \overset{?}{=} T_2 \to T\}\end{aligned}$$

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

où les variables T_1, T_2, T_x sont *fraîches* (on notera par la suite И T_1, T_2, T_x).

Remarque 4.20. On peut résumer les cas « plus », « application » et « abstraction ».



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

Définition 4.13. 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 4.17. Avec $\sigma = [T_1 \mapsto \texttt{int}, T_2 \mapsto (\texttt{int} \to T_3)]$, on a donc

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

Exemple 4.18. La contrainte $T_1 \overset{?}{=} T_2 \to T_3$ est égalisée par la solution $\sigma = [T_1 \mapsto T_2 \to \texttt{int}, T_3 \mapsto \texttt{int}]$.

Définition 4.14. 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 4.8. Soit $e \in \mathsf{FUN}$ et soit Γ tel que $\mathscr{V}\!\ell(e) \subseteq \mathrm{dom}(\Gamma)$. 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

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

  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 $\mathcal{T}_{\mathrm{a}}$.
- ▷ Les autres cas se traitent similairement.

□

Proposition 4.9. Supposons $\Gamma \vdash e : \tau$. Alors, pour tout $T \in$ 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 $T_0 \in$ 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.

$\square$

Théorème 4.1. On a $\Gamma \vdash e : \tau$ si, et seulement si $\forall T \in$ ITyp, l'ensemble de contraintes $\mathsf{CT}(e, \Gamma, T)$ admet une solution constante σ tel que $\sigma(T) = \tau$. $\square$

Remarque 4.21. On a caractérisé l'ensemble des dérivations de $\Gamma \vdash e : \tau$ avec l'ensemble des solutions constantes de $\mathsf{CT}(e, \Gamma, T)$.

Termes et unification.

Définition 4.15. 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 $\mathsf{T}(\Sigma, V)$ des *termes* sur (Σ, V), notés t, u, *etc*, est défini de manière inductive, ce qui est décrit par la grammaire :

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

où f est une constante d'arité k.

Remarque 4.22. 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 $X \in V$ ne fait pas partie du langage de départ. Il n'y a pas de liens pour X.

Exemple 4.19. Avec $\Sigma = \{f^2, g^1, a^0, b^{\text{ř}}\}$,

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

est un terme.

Définition 4.16. On définit $\mathsf{Vars}(t)$ l'ensemble des inconnues/variables de t par induction sur t. Il y a deux familles de cas :

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

Définition 4.17. Une *substitution*, notée $\sigma, \sigma_1, \sigma'$, *etc*, est un dictionnaire sur $(V, \mathsf{T}(\Sigma, V))$.

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

Soit σ une substitution et $t \in \mathsf{T}(\Sigma, V)$. 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 4.20. 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*.