Sémantique opérationnelle.

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.

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 1 | *Définition des expressions arithmétiques simples*

Note 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 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 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.

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

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

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 2.

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

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 $a \mathscr{R} b$ 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

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

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

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 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 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 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.

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 :

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

Lemme 1. La relation $\mathscr{R}^\star$ est transitive.

Preuve. On démontre

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

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

- ▷ *Réflexivité.* On a donc $x = y$ et, par hypothèse, $y \mathscr{R}^\star z$.
- ▷ *Transitivité.* On sait que $x \mathscr{R} a$ et $a \mathscr{R}^\star y$. 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 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 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 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 :

$$a_1 \oplus a_2 \xrightarrow{(H_2)+\text{lemme } 2}{}^\star a_1 \oplus \underline{k}_2 \xrightarrow{(H_1)+\text{lemme } 3}{}^\star \underline{k}_1 \oplus \underline{k}_2 \xrightarrow{\text{Add}} \underline{k}.$$

□

Proposition 2. Soient a une expression et k un entier. On a l'implication

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

□

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 :

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

et les règles de propagation du $\perp$:

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

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

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

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

et les règles de propagation du $\perp$:

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

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

1.5 Sémantique contextuelle pour EA.

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

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

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 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 (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 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 6 (Troisième (et dernière) tentative)**.** On pose

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

Là, c'est réussi !

Lemme 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 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.

□

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 2 | *Définition inductive de* LEA

Exemple 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. $\texttt{let}\, x = \underline{7} \oplus \underline{2}\ \texttt{in}\ (\texttt{let}\, x = \underline{5}\ \texttt{in}\ x \oplus x) \oplus x$.

2.1 Sémantique à grands pas sur LEA.

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

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

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

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

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

Définition 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 :

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

Exemple 8.

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

Définition 2. Une expression $a \in \mathsf{LEA}$ est *close* si $\mathcal{VL}(a) = \emptyset$. On note $\mathsf{LEA}_0 \subseteq \mathsf{LEA}$ l'ensemble des expressions arithmétiques de closes.

Définition 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 :

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

1. On surcharge encore les notations.

2.2 Sémantique à petits pas sur LEA.

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

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

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

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.

2.3 Sémantique contextuelle pour LEA.

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

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

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

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

et

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

2.4 Sémantique sur LEA avec environnement.

Définition 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 $(\mathscr{V}, \mathbb{Z})$.

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

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

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

Remarque 4.
- ▷ 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.)

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 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.

3.1 Sémantique opérationnelle « informellement ».

Exemple 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 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 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.

3.2 Sémantique opérationnelle de FUN (version 1).

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

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

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

Définition 6. Pour $e \in \mathsf{FUN}$, $x \in \mathcal{V}$ et v une valeur **close**, on définit la *substitution* $e[v/x]$ de x par v dans e par :

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

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

Grands pas pour FUN.

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

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

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

Remarque 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 :

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

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

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

Remarque 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 5. On a

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

Remarque 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 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.

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

Remarque 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 6. ▷ Si v est close et si $x \notin \mathscr{V}\ell(e)$ alors $e[^v/_x] = e$.
▷ Si v est close, $\mathscr{V}\ell(e[^v/_x]) = \mathscr{V}\ell(e) \setminus \{x\}$. □

Lemme 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 $(\texttt{fun}\, x \texttt{ -> } e)\ v$ est close, alors
   ▷ $(\texttt{fun}\, x \texttt{ -> } e)$ 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 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 7 (Définition informelle de l'α-conversion)**.** On définit l'α-conversion, notée $e =_\alpha e'$: on a $\texttt{fun}\, x \texttt{ -> } e =_\alpha \texttt{fun}\, y \texttt{ -> } e'$ 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

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

partout ».

Exemple 11 (*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 $x' \notin \mathscr{V}\ell(e)$.

"Lemme" 1. Si $E_0 \subseteq \mathscr{V}$ 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 12 (Fondamental). En fait FUN désigne l'ensemble des expressions décrites par la grammaire initiale *quotientée* par α-conversion.

Remarque 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 8. 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 :

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

5. C'est une « variable fraîche ».

- $(e_1 + e_2)[v/x] = (e_1[v/x]) + (e_2[v/x])$;
- $(e_1\ e_2)[v/x] = (e_1[v/x])\ (e_2[v/x])$.

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

On ajoute les déclarations locales (comme pour EA → LEA) à notre petit langage fonctionnel. Dans la grammaire des expressions de FUN, on ajoute :

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

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

- on définit $\mathcal{V}\ell(\texttt{let}\, x = e_1\ \texttt{in}\ e_2) = \mathcal{V}\ell(e_1) \cup (\mathcal{V}\ell(e_2) \setminus \{x\})$;
- on ne change pas les valeurs : une déclaration locale n'est pas une valeur ;
- on ajoute $\texttt{let}\, x = e_1\ \texttt{in}\ e_2 =_\alpha \texttt{let}\, y = e_1\ \texttt{in}\ e_2'$, 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)

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

- 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`.

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 :

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

Lemme 8. Pour tout $e \in (\text{FUN} + \texttt{let})$,

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

Pour démontrer le lemme 8, on procède par induction sur e. C'est long et rébarbatif, mais la proposition ci-dessous est bien plus intéressante.

Proposition 4. Pour toutes expressions e, e' de FUN + `let`, si on a la réduction $e \to_{\text{FUN+let}} e'$ alors $[\![e]\!] \to_{\text{FUN}} [\![e']\!]$.

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

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

$$(\texttt{fun}\, x \texttt{ -> } [\![e]\!]_2)\ [\![v]\!] \xrightarrow{\mathcal{R}_\beta}_{\text{FUN}} [\![e]\!]_2\,[[\![v]\!]/x],$$

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

car $[\![v]\!]$ est une valeur par le lemme 8, ce qui justifie $\mathscr{R}_\beta$. De plus, encore par le lemme 8, 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.

□

Proposition 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 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 8 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 $e = (\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2)$ alors

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

On vérifie aisément ce que l'on doit montrer.

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

□

4 Typage en FUN.

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 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 9. 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 $x \notin \operatorname{dom} \Gamma$. [8]

Remarque 14. On peut voir/implémenter Γ comme des listes finies de couples (x, τ).

Définition 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.

$$\frac{}{\Gamma \vdash k : \texttt{int}}\ \mathcal{T}_{\text{k}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \mathcal{T}_{\text{v}} \qquad \frac{\Gamma, x : \tau_1 \text{ [9]} \vdash e_2}{\Gamma \vdash \texttt{fun}\, x \to e : \tau_1 \to \tau_2}\ \mathcal{T}_{\text{f}}$$

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

Remarque 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 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 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,

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

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

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

4.2 Propriétés du système de types.

Lemme 9. ▷ 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). □

4.3 Propriété de progrès.

Lemme 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 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 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 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 Propriété de préservation.

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

Lemme 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.
  - *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 (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 $(\texttt{fun}\, x \to e) \to e'$ 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 $\emptyset \vdash e_2' : \tau_0$.

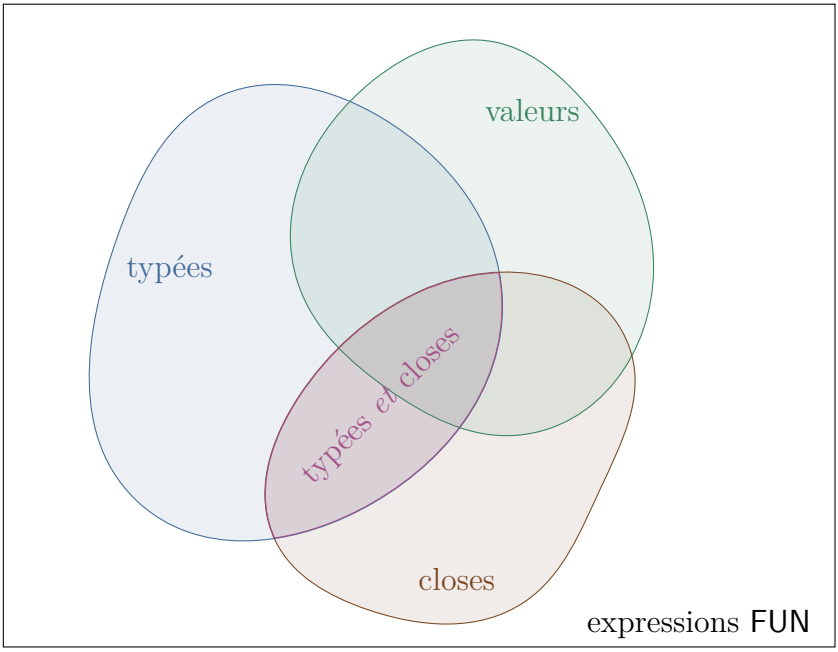
  On doit montrer que si $e_1\ e_2 \to e'$ alors $\emptyset \vdash e' : \tau$. Supposons que $e_1\ e_2 \to e'$, il y a 3 sous-cas.
  - *Sous-cas* $\mathcal{R}_\text{ad}$. Cela veut dire que $e_2 \to e_2'$ et $e' = e_1\ e_2'$. On conclut $\emptyset \vdash e_1\ e_2' : \tau$ par (H_2') et (H_1).
  - *Sous-cas* $\mathcal{R}_\text{ag}$. 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 $e_1 = \texttt{fun}\, x \to e_0$, $e_2 = v$ 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.

□

Remarque 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 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.

4.5 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])

4.6 Inférence de types.

Typage et contraintes.

Exemple 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.

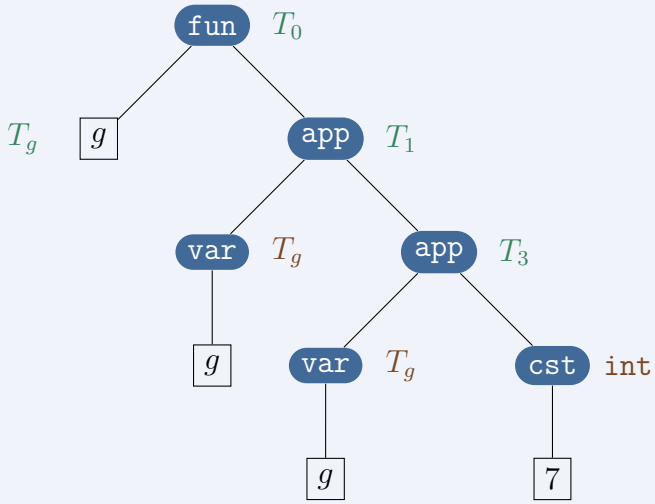
On doit donc lire le programme, et « prendre des notes ». Ces « notes » sont des contraintes que doivent vérifier le programme.

Exemple 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

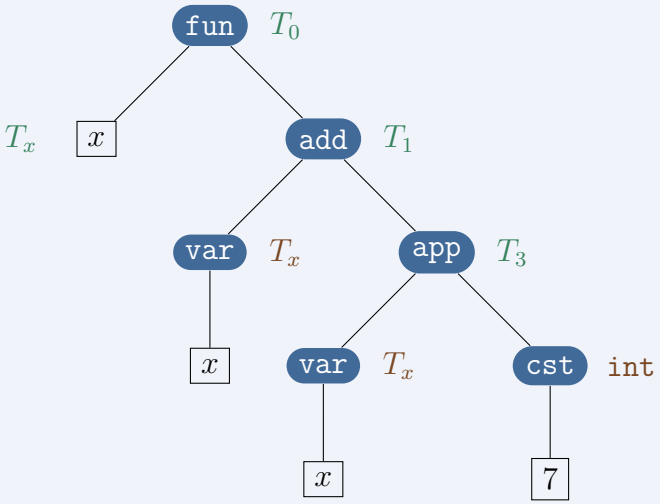
$$T_0 = (\texttt{int} \to \texttt{int}) \to \texttt{int}.$$

Exemple 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 `int` mais admet une infinité de dérivations.

Exemple 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 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 12. Une *contrainte de typage* est une paire de types étendus[11], 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 :

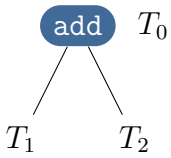
- ▷ $\mathsf{CT}(e_1 + e_2, \widehat{\Gamma}, T) = \mathsf{CT}(e_1, \widehat{\Gamma}, T_1) \cup \mathsf{CT}(e_2, \widehat{\Gamma}, T_2) \cup \{T_1 \stackrel{?}{=} \texttt{int}, T_2 \stackrel{?}{=} \texttt{int}, T \stackrel{?}{=} \texttt{int}\}$

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

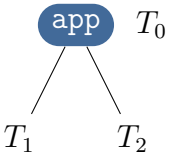
- $\mathsf{CT}(x, \widehat{\Gamma}, T) = \{T \overset{?}{=} \widehat{\Gamma}(x)\}$
- $\mathsf{CT}(k, \widehat{\Gamma}, T) = \{T \overset{?}{=} \texttt{int}\}$
- $\mathsf{CT}(\texttt{fun}\, x \to e, \widehat{\Gamma}, T) = \mathsf{CT}(e, (\widehat{\Gamma}, x : T_x), T_2) \cup \{T \overset{?}{=} T_1 \to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

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

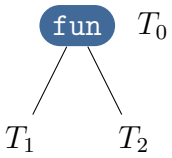
Remarque 20. On peut résumer les cas « plus », « application » et « abstraction ».



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



$T_1 = T_2 \to T_0$



$T_0 = T_1 \to T_2$

Définition 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 $(\hat{\tau}_1 \overset{?}{=} \hat{\tau}_2) \in C$, on dit que σ égalise $\hat{\tau}_1 \overset{?}{=} \hat{\tau}_2$ 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 :

- $\sigma(\hat{\tau}_1 \to \hat{\tau}_2) = \sigma(\hat{\tau}_1) \to \sigma(\hat{\tau}_2)$;

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

- ▷ $\sigma(\texttt{int}) = \texttt{int}$;
- ▷ $\sigma(T)$ est le type étendu associé à T dans σ.

Exemple 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 18. 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 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 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

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

  où $\mathit{И}\ T_1$, 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 9. 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 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 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.

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 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 $\mathsf{T}(\Sigma, \mathrm{V})$ 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 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 $X \in \mathrm{V}$ ne fait pas partie du langage de départ. Il n'y a pas de liens pour X.

Exemple 19. 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 16. On définit $\mathsf{Vars}(t)$ l'ensemble des inconnues/variables de t par induction sur t. Il y a deux familles de cas :

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

- ▷ $\mathsf{Vars}(X) = \{X\}$.

Exemple 20. Avec l'expression t_0 précédente, on a

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

Définition 17. 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 21. 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 4. 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 18. 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 $\mathscr{P} = \{t_1 \stackrel{?}{=} u_1, \dots, t_k \stackrel{?}{=} u_k\}$.

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 22. 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 » [12]). Ainsi, σ_1 est « plus général » que σ_2 ; σ_2 est un « cas particulier » de σ_1.

Exemple 23 (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 : $\mathrm{U}(\mathscr{P}_2) = \mathrm{U}(\mathscr{P}_3) = \emptyset$.

12. Elle sera définie formellement ci-après.

Algorithme d'unification (du premier ordre).

Définition 19. Un *état* est soit un couple $(\mathcal{P}, \sigma)$, 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 forme $(\{t \stackrel{?}{=} t'\} \sqcup \mathcal{P}, \sigma)$, 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. $(\{f(t_1, \dots, t_k) \stackrel{?}{=} f(u_1, \dots, u_n) \sqcup \mathcal{P}, \sigma\}) \to (\{t_1 \stackrel{?}{=} u_1, \dots, t_k \stackrel{?}{=} u_k\} \cup \mathcal{P}, \sigma)$;
2. $(\{f(t_1, \dots, t_k) \stackrel{?}{=} g(u_1, \dots, u_n) \sqcup \mathcal{P}, \sigma\}) \to \bot$ si $f \neq g$;
3. $(\{X \stackrel{?}{=} t\} \sqcup \mathcal{P}, \sigma) \to (\mathcal{P}[t/X], [t/X] \circ \sigma)$ où
   - $X \notin \mathsf{Vars}(t)$,
   - $\mathcal{P}[t/X] = \{u[t/X] \stackrel{?}{=} u'[t/X] \mid (u \stackrel{?}{=} u') \in \mathcal{P}\}$,
   - 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. $(\{X \stackrel{?}{=} t\} \sqcup \mathcal{P}, \sigma) \to \bot$ si $X \in \mathsf{Vars}(t)$ et $t \neq X$;
5. $(\{X \stackrel{?}{=} X\} \sqcup \mathcal{P}, \sigma) \to (\mathcal{P}, \sigma)$.

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

Exemple 24. On applique l'algorithme d'unification comme

montré ci-dessous :

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

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 20. 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 23 (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 10. 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 12.** Si $X \notin \mathrm{dom}(\sigma)$ alors
>
> $$[t/X] \circ \sigma = \Big[X \mapsto t, Y_1 \mapsto (\sigma(Y_1))[t/X], \ldots, Y_l \mapsto (\sigma(Y_k))[t/X]\Big],$$
>
> où $\mathrm{dom}(\sigma) = \{Y_1, \ldots, Y_k\}$. □

□

Proposition 11. On note $\to^\star$ la clôture réflexive et transitive de la relation $\to$.

1. Un *unificateur le plus général* (*mgu*[13] 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 13.** Pour toute substitution σ, alors σ est un unificateur le plus général de $\overset{?}{\sigma}$.

13. 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 $(\mathscr{P}, \emptyset) \to^\star (\mathscr{P}_0, \sigma_0) \to \bot$.

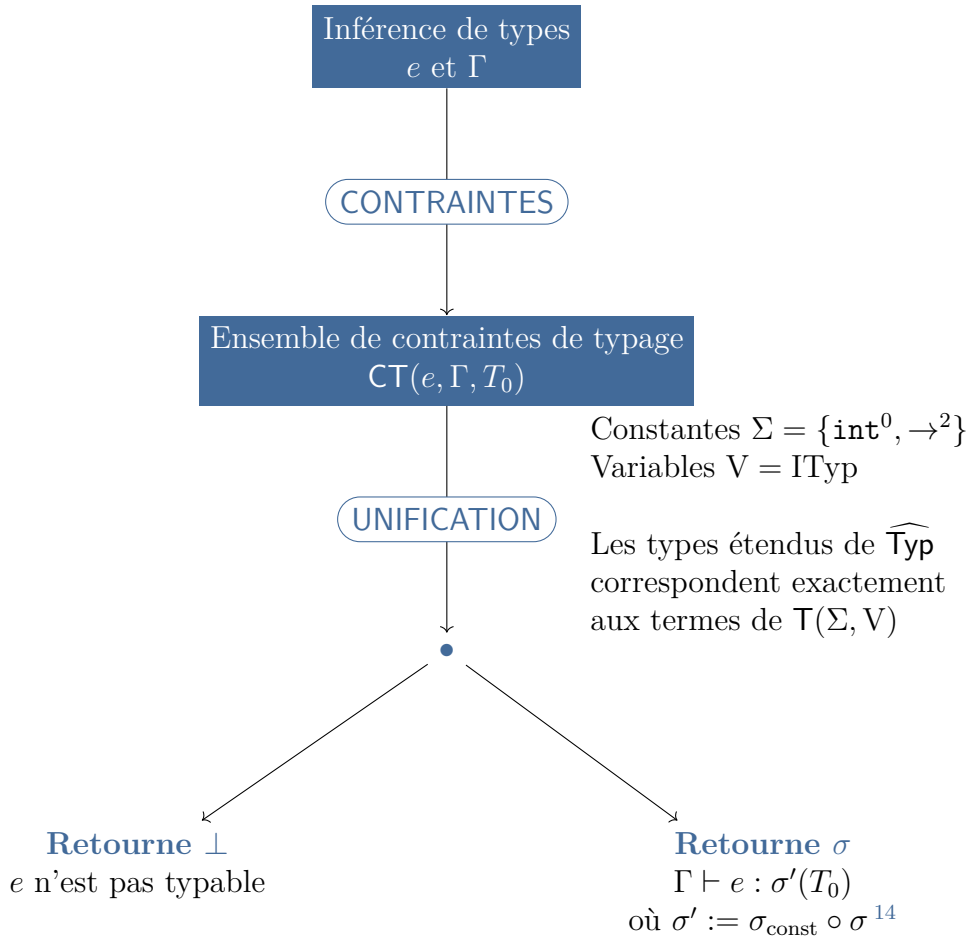
   □

Lemme 14. La relation $\to$ est terminante (il n'y a pas de chaîne infinie avec cette relation).

Preuve. Vue plus tard. □

Théorème 2. L'algorithme d'unification calcule un unificateur le plus général si, et seulement si le problème initial a une solution. □

Retour sur l'inférence de types pour FUN.



Ceci conclut notre étude du petit langage fonctionnel FUN.

14. 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.

5 Un petit langage impératif, IMP.

5.1 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 $a ::= \underline{k} \mid a_1 \oplus a_2 \mid x$;[15]

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.

Exemple 25. La commande

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

représente un programme calculant la factorielle d'un nombre x.

Sémantique opérationnelle à grands pas.

Définition 21 (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

- $\sigma'(x) := k$;

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

▷ $\sigma'(y) := \sigma(y)$ 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

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

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

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

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

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 24 (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. [16]
- ▷ 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 25. 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

$$\texttt{while true do skip}$$

diverge car on n'a pas de dérivation finies :

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

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*.

5.2 Sémantique dénotationnelle de IMP.

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

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

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.

- $\mathscr{D}(\texttt{skip}) = \{(\sigma, \sigma)\}$;
- $\mathscr{D}(x \texttt{ := } a) = \{(\sigma, \sigma') \mid x \in \mathrm{dom}(\sigma), \sigma' = \sigma[x \mapsto k] \text{ et } (\sigma, k) \in \mathscr{D}(a)\}$;
- $\mathscr{D}(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2) = \{(\sigma, \sigma' \mid (\sigma, \texttt{true}) \in \mathscr{D}(b), (\sigma, \sigma') \in \mathscr{D}(c_1))\} \cup \{(\sigma, \sigma' \mid (\sigma, \texttt{false}) \in \mathscr{D}(b), (\sigma, \sigma') \in \mathscr{D}(c_2))\}$;
- $\mathscr{D}(c_1 \texttt{ := } c_2) = \{(\sigma, \sigma'') \mid \exists \sigma', (\sigma, \sigma') \in \mathscr{D}(c_1) \text{ et } (\sigma', \sigma'') \in \mathscr{D}(c_2)\}$;[17]
- $\mathscr{D}(\texttt{while } b \texttt{ do } c) = ???$.

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

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

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

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

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 26. 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 27. Ainsi défini,

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

Théorème 3. On a $c, \sigma \Downarrow \sigma'$ si et seulement si $(\sigma, \sigma') \in \mathsf{D}(c)$.

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 15. 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 $c, \sigma \Downarrow \sigma' \implies \mathcal{P}$ par induction sur c, car cela ne fonctionne pas, on n'a pas les bonnes hypothèses. On procède par induction sur la relation $c, \sigma \Downarrow \sigma'$.