$$\frac{}{\rho \vdash x := e \Rightarrow \rho[x \mapsto [\![e]\!]\rho]}\ (:=) \qquad \frac{}{\rho \vdash \texttt{skip} \Rightarrow \rho}\ (\texttt{Skip})$$

$$\frac{\rho \vdash c_1 \Rightarrow \rho' \quad \rho' \vdash c_2 \Rightarrow \rho''}{\rho \vdash c_1; c_2 \Rightarrow \rho''}\ (\texttt{Seq})$$

$$\frac{\rho \vdash c_1 \Rightarrow \rho'}{\rho \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Rightarrow \rho'}\ (\texttt{if}_1) \quad \text{si } [\![e]\!]\rho \neq 0 \qquad \frac{\rho \vdash c_2 \Rightarrow \rho''}{\rho \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Rightarrow \rho''}\ (\texttt{if}_2) \quad \text{si } [\![e]\!]\rho = 0$$

$$\frac{\rho \vdash c \Rightarrow \rho' \quad \rho' \vdash \texttt{while}\ e\ \texttt{do}\ c \Rightarrow \rho''}{\rho \vdash \texttt{while}\ e\ \texttt{do}\ c \Rightarrow \rho''}\ (\texttt{while}) \quad \text{si } [\![e]\!]\rho \neq 0 \qquad \frac{}{\rho \vdash \texttt{while}\ e\ \texttt{do}\ c \Rightarrow \rho}\ (\texttt{while}_{\text{fin}}) \quad \text{si } [\![e]\!]\rho = 0$$

FIGURE 1 – La sémantique opérationnelle à grands pas de IMP.

Exercice 1 :

Calculer la sémantique opérationnelle à grand pas du programme :

$$c \stackrel{\text{def}}{=} x := 0\ ;\ y := 3\ ;\ \texttt{while}\ y\ \texttt{do}\ (x := x + y\ ;\ y := y + (-1)),$$

c'est-à-dire donner un jugement $\rho \vdash c \Rightarrow \rho'$ dérivable et sa dérivation.

Solution :

TODO

$$\frac{\dfrac{}{\rho \vdash x := 0 \Rightarrow \rho[x \mapsto 0]}\ (:=) \qquad \dfrac{}{\rho[x \mapsto 0] \vdash y := 3 \Rightarrow \rho[x \mapsto 0, y \mapsto 3]}\ (:=)}{\rho \vdash x := 0; y := 3 \Rightarrow \rho[x \mapsto 0, y \mapsto 3]}\ (\texttt{Seq})$$

Exercice 2 :

Prouver que pour tout environnement ρ, il n'existe pas d'environnement ρ' tel que $\rho \vdash \texttt{while}\ \dot{1}\ \texttt{do skip} \Rightarrow \rho'$ soit dérivable.

Solution :

Par l'absurde, supposons qu'il existe une telle dérivation. Tentons de la construire. Avec les règles à notre disposition, nous ne pouvons qu'appliquer (`while`). On arrive donc à la dérivation (inachevée) :

$$\frac{\dfrac{\vdots}{\rho \vdash \texttt{skip} \Rightarrow \rho'} \qquad \dfrac{\vdots}{\rho' \vdash \texttt{while}\ \dot{1}\ \texttt{do skip} \Rightarrow \rho''}}{\rho \vdash \texttt{while}\ \dot{1}\ \texttt{do skip} \Rightarrow \rho''}\ (\texttt{while})$$

Nous ne pouvons appliquer que la règle (`Skip`) pour les sous-arbre de gauche. On arrive donc à cette dérivation :

$$\frac{\dfrac{}{\rho \vdash \texttt{skip} \Rightarrow \rho}\ (\texttt{Skip}) \qquad \dfrac{\vdots}{\rho \vdash \texttt{while}\ \dot{1}\ \texttt{do skip} \Rightarrow \rho'}}{\rho \vdash \texttt{while}\ \dot{1}\ \texttt{do skip} \Rightarrow \rho'}\ (\texttt{while})$$

Et c'est ici qu'on retrouve notre contradiction ! On sait que cette dérivation est l'unique possible pour `while` $\dot{1}$ `do skip`, donc elle doit se répéter, dans le même état, en haut à droite. Notre dérivation se contiendrait donc elle même, et serait infinie.

On a donc contradiction, et il n'existe donc aucune paire d'environnements (ρ, ρ') tels que $\rho \vdash$ `while` $\dot{1}$ `do skip` $\Rightarrow \rho'$ soit dérivable.

Exercice 3 :

Étant donnés deux programmes c_1, c_2, on dit que c_1 et c_2 sont équivalents, noté $c_1 \sim c_2$ ssi pour tous environnements ρ, ρ', $(\rho \vdash c_1 \Rightarrow \rho')$ est dérivable ssi $(\rho \vdash c_2 \Rightarrow \rho')$ est dérivable.

1. Montrer que `while` e `do` c $\sim$ `if` e `then` $(c;$ `while` e `do` $c)$ `else skip`.
2. Quels sont les programmes équivalents à `while` $\dot{1}$ `do skip` ?
3. On appelle contexte un programme qui contient une variable $\square$, correspondant à une commande. La variable peut être utilisée à de multiples reprises.
   (a) Définir formellement la notion de contexte avec une grammaire.
   (b) Soit C un contexte. Montrer que $c_1 \sim c_2$ implique $C[c_1] \sim C[c_2]$.

Solution :

1. Bien faire attention à faire les deux sens de l'équivalence. Chaque sens se base sur le fait que si un jugement est dérivable et ne peut être obtenu que par une seule règle, les jugements nécessaires pour cette règle sont également dérivables.
2. Tous les programmes non dérivables.
3. (a) Il suffit d'ajouter $\square$ aux commandes pour avoir une grammaire des contextes. Ceci nous donne la grammaire suivante pour les contextes C.

$$
\begin{array}{rcll}
C & ::= & \square & \text{contexte trivial} \\
 & | & c & \text{commande (contexte sans trou)} \\
 & | & C_1; C_2 & \text{séquence} \\
 & | & \texttt{if } e \texttt{ then } C_1 \texttt{ else } C_2 & \text{conditionnelle} \\
 & | & \texttt{while } e \texttt{ do } C & \text{boucle while}
\end{array}
$$

   Le cas c semble surprenant, mais il est nécessaire pour pouvoir faire des séquences où le programme de droite ne contient pas de trou, par exemple. Si on prend un programme sans trou c comme un contexte, et c' un autre programme, on a $c[c'] = c$.

   (b) On commence par définir une *profondeur* p sur les contextes :

$$
\begin{cases}
p(c) = 0 \\
p(\square) = 0 \\
p(C_1; C_2) = 1 + \max(p(C_1), p(C_2)) \\
p(\texttt{if } e \texttt{ then } C_1 \texttt{ else } C_2) = 1 + \max(p(C_1), p(C_2)) \\
p(\texttt{while } e \texttt{ do } C) = 1 + p(C)
\end{cases}
$$

   Procédons par récurrence (forte) sur la profondeur des contextes C. On pose $\mathcal{H}(n)$ la phrase "Pour tout contexte C de profondeur n ou moins, pour toute paire de programmes équivalents c_1 et c_2, $C[c_1] \sim C[c_2]$"

Initialisation :
Soit $c_1 \sim c_2$.
Soit un contexte de profondeur 0 (soit une commande tierce c', soit $\square$).
$c'[c_1] = c'[c_2] = c'$, et trivialement $c' \sim c'$.
$\square[c_1] = c_1$, $\square[c_2] = c_2$, et donc par hypothèse $\square[c_1] \sim \square[c_2]$.
On a bien prouvé $\mathcal{H}(0)$

Hérédité :
Soit $n \geq 0$. On suppose qu'on a $\mathcal{H}(n)$. Nous allons prouver $\mathcal{H}(n+1)$.
Soit $c_1 \sim c_2$, C un contexte de profondeur $n+1$. Procédons par disjonction des cas.
Cas 1, $C = C_1; C_2$ *:*
Todo
Cas 2, $C =$ `if` e `then` C_1 `else` C_2 *:*
Todo
Cas 3, `while` e `do` C *:* Todo

On a bien prouvé que $\mathcal{H}(n)$ implique $\mathcal{H}(n+1)$, et ainsi la propriété demandée.

Exercice 4 :
Considérons les expressions booléennes suivantes :

$$b ::= \texttt{True} \mid \texttt{False} \mid e \doteq e \mid e \dot{<} e \mid \dot{\neg} b \mid b \dot{\vee} b \mid b \dot{\wedge} b$$

1. Donner une sémantique opérationnelle à petits pas **très naïve** pour les expressions booléennes.
2. Modifier vos règles pour qu'elles implémentent le "ou" paresseux : lors de l'évaluation de $b_1 \dot{\vee} b_2$, b_2 n'est pas évalué si b_1 s'évalue à `True`. Même chose pour le "et".
3. Modifier vos règles pour qu'elles implémentent l'évaluation parallèle du "ou". Même chose pour le "et".
4. Dans les trois systèmes de déduction obtenu, que peut-on dire du nombre de dérivations d'un triplet $(b, \rho) \to \_$?

Solution :

1. On commence par des règles pour décrire les tests sur les expressions :

$$\frac{[\![e_1]\!]\rho = [\![e_2]\!]\rho}{(e_1 \doteq e_2, \rho) \to (\texttt{True}, \rho)} \qquad \frac{[\![e_1]\!]\rho \neq [\![e_2]\!]\rho}{(e_1 \doteq e_2, \rho) \to (\texttt{False}, \rho)}$$

$$\frac{[\![e_1]\!]\rho < [\![e_2]\!]\rho}{(e_1 \dot{<} e_2, \rho) \to (\texttt{True}, \rho)} \qquad \frac{[\![e_1]\!]\rho \geq [\![e_2]\!]\rho}{(e_1 \dot{<} e_2, \rho) \to (\texttt{False}, \rho)}$$

Et puis des règles pour décrire $\dot{\neg}$

$$\frac{}{(\dot{\neg}\texttt{True}, \rho) \to (\texttt{False}, \rho)} \qquad \frac{(e, \rho) \to (e', \rho)}{(\dot{\neg} e, \rho) \to (\dot{\neg} e', \rho)} \qquad \frac{}{(\dot{\neg}\texttt{False}, \rho) \to (\texttt{True}, \rho)}$$

Ensuite, des règles de la forme $\dfrac{}{(x\ \alpha\ y, \rho) \to (z, \rho)}$ avec $a, b \in \{\texttt{True}, \texttt{False}\}$, $\alpha \in \{\dot{\wedge}, \dot{\vee}\}$ (par exemple, $(\texttt{True}\ \dot{\wedge}\ \texttt{False}, \rho) \to (\texttt{False}, \rho)$) pour ajouter les tables de

vérité des connecteurs binaires.

Il faut finalement des règles pour réduire les expressions en dessous de $\dot\wedge$ et $\dot\vee$. Dans ce qui suit, $\alpha \in \{\dot\wedge, \dot\vee\}$.

$$\frac{(e_1, \rho) \to (e_1', \rho)}{(e_1\ \alpha\ e_2, \rho) \to (e_1'\ \alpha\ e_2, \rho)} \qquad \frac{(e_2, \rho) \to (e_2', \rho)}{(e_1\ \alpha\ e_2, \rho) \to (e_1\ \alpha\ e_2', \rho)}$$

NB : les réductions de cette sémantique ne sont pas déterministes, comme on peut choisir à réduire une expression à la droite de son opérateur, puis à gauche, ou vis versa. Cependant, le système est confluent, on arrive finalement toujours à la même configuration irréductible.

2. Pour implémenter l'évaluation paresseuse, on ajouter à nos tables de vérité les règles suivantes, avec une version symétrique pour le côté droit :

$$\frac{}{(\texttt{False}\ \dot\wedge\ e, \rho) \to (\texttt{False}, \rho)} \qquad \frac{}{(\texttt{True}\ \dot\vee\ e, \rho) \to (\texttt{True})}$$

3. L'évaluation parallèle est une règle un peu plus compliquée, comme elle requiert deux hypothèse. On la construit ainsi avec $\alpha \in \{\dot\wedge, \dot\vee\}$:

$$\frac{(e_1, \rho) \to (e_1', \rho) \qquad (e_2, \rho) \to (e_2', \rho)}{(e_1\ \alpha\ e_2, \rho) \to (e_1'\ \alpha\ e_2', \rho)}$$

4. On peut facilement s'assurer que les systèmes paresseux et parallèles réduisent chacun le nombre d'étapes nécessaires dans une dérivation.

Exercice 5 :

L'un des objectifs des cours suivants sera d'introduire les outils mathématiques nécessaires pour pouvoir définir une sémantique dénotationnelle de IMP : $[\![c]\!]\rho = \rho'$, ce qui signifie que l'exécution du programme c dans l'environnement ρ mène à l'environnement ρ'. Cette écriture « fonctionnelle » suggère que l'environnement ρ' est entièrement déterminé par c et ρ (bien sûr, puisque nos programmes sont « déterministes »).

1. Montrer cette propriété pour la sémantique opérationnelle à grand pas, sans passer par un résultat d'équivalence avec une autre sémantique : pour tout c, ρ, ρ_1, ρ_2, si $\rho \vdash c \Rightarrow \rho_1$ et $\rho \vdash c \Rightarrow \rho_2$ alors $\rho_1 = \rho_2$.
2. On considère le langage non déterministe donné par la syntaxe suivante :

$$c ::= \texttt{skip} \mid x := e \mid c; c \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ e\ \texttt{do}\ c \mid c \vee c$$

où l'instruction $c_1 \vee c_2$ signifie « exécute c_1 ou exécute c_2, de manière non déterministe ». Proposer une sémantique opérationnelle à grand pas de ce langage.

Solution :

1. On procède à nouveau par récurrence sur la profondeur du programme, comme dans l'exercice 3, où l'on adapte la fonction p.

   TODO

2. Une solution possible est d'abandonner la propriété de déterminisme dès le début, et d'introduire deux règles, indexées par $i \in \{1, 2\}$:

$$\frac{\rho \vdash c_i \Rightarrow \rho'}{\rho \vdash c_1 \vee c_2 \Rightarrow \rho'} \vee_i$$

Il s'agit là de la réprésentation d'un choix non déterministe. On peut continuer la dérivation d'un côté ou de l'autre.

Il existe d'autres solutions. Par exemple, l'une d'entre elle consiste à changer les ρ utilisés, pour qu'ils associent des variables à un ensemble de valeurs possibles pour que $\vee$ aie une seul règle. Cependant, il est facile de voir que cela complexifiera les règles de `while` énormément.

Exercice 6 :

En cours, vous aver aperçu la distinction entre sémantique dénotationnelle et sémantique opérationnelle des programmes IMP. Cependant, vous n'aver utilisé qu'une sémantique dénotationnelle des expressions arithmétiques. Dans cet exercice, on s'étudie le cas de deux extensions des opérations arithmétiques :

1. On étend nos expressions arithmétiques par la syntaxe suivante :

$$e ::= x \mid \dot{n} \mid e \dot{+} e \mid \dot{-}e \mid \dot{f}(e)$$

Pour interpréter le symbole $\dot{f}$, on suppose disposer d'une fonction **partielle** f des entiers dans les entiers.

   (a) Donner une sémantique opérationnelle à grands pas (inspirer vous de celle pour IMP, vue en cours et rappelée en figure 1) pour ces expressions.
   (b) Étender la sémantique dénotationnelle vue en cours pour ces expressions.
   (c) Montrer l'équivalence des deux sémantiques.
   (d) Quelle différence essentielle rend la sémantique dénotationnelle pour les expressions arithmétiques beaucoup plus simple que celle pour les programmes ?

2. Expressions arithmétiques avec effets de bords : on étend nos expressions arithmétiques par la syntaxe suivante :

$$e ::= x \mid \dot{n} \mid e \dot{+} e \mid \dot{-}e \mid c \texttt{ resultis } e$$

Intuitivement, pour évaluer l'expression c `resultis` e, on évalue d'abord la commande c, puis on évalue e dans l'environnement obtenu.

   (a) Donner des sémantiques opérationnelle et dénotationnelle pour ces expressions (on supposera disposer d'une sémantique pour les programmes c).
   (b) Comment s'évalue le terme $((x := x \dot{+} \dot{1})$ `resultis` $x) \dot{+} ((y := x \dot{+} x)$ `resultis` $x)$ dans l'environnement $\rho = [x \mapsto 3]$.
   (c) Comprendre désormais les horreurs que permet d'écrire le C : `T[i++] = i++`.

Solution :

1. (a) Avant de se lancer dans la sémantique à grand pas, il faut décider ce que "rend" la sémantique. Dansl e cas de IMP, on transforme un environnement vers un autre par le biais d'un programme, d'où la forme $\rho \vdash c \Rightarrow \rho'$.

Dans le cas des expressions cependant, on veut rendre un entier naturel. Il nous faudra donc un jugement de la forme $\rho \vdash e \Rightarrow n$.

Ainsi, on utilisera ces règles :

$$\frac{}{\rho \vdash x \Rightarrow \rho(x)} \qquad \frac{}{\rho \vdash \dot{n} \Rightarrow n} \qquad \frac{\rho \vdash e \Rightarrow a}{\rho \vdash \dot{-}e \Rightarrow -a}$$

$$\frac{\rho \vdash e_1 \Rightarrow a_1 \quad \rho \vdash e_2 \Rightarrow a_2}{\rho \vdash e_1 \dot{+} e_2 \Rightarrow a_1 + a_2} \qquad \frac{\rho \vdash e \Rightarrow a \quad f(a) = b}{\rho \vdash \dot{f}(e) \Rightarrow b}$$

(b) La sémantique dénotationnelle est triviale :

$$\begin{aligned}
&[\![x]\!]\rho = \rho(x) \\
&[\![\dot{n}]\!]\rho = n \\
&[\![e_1 \dot{+} e_2]\!]\rho = [\![e_1]\!]\rho + [\![e_2]\!]\rho \\
&[\![\dot{-}e]\!]\rho = -[\![e]\!]\rho \\
&[\![\dot{f}(e)]\!]\rho = f([\![e]\!]\rho)
\end{aligned}$$

(c) TODO : il suffit de procéder par induction sur la structure des expressions pour prouver que $(\rho \vdash e \Rightarrow n) \Leftrightarrow [\![e]\!]\rho = n$

(d) La différence principale est que les expressions sont construites avec des symboles ayant un analogue direct dans notre langage cible, celui des expressions mathématiques. Ceci rend la sémantique dénotationelle (qui consiste principalement à "effacer" les points) transparente.

2. (a) On supposera qu'on a une sémantique $[\![-]\!]_C$ pour les programmes. $[\![c]\!]_C\rho$ sera un autre environnement, qui est modifié par c (quand c termine bien sûr).

Opérationelle : TODO

Pour la sémantique dénotationnelle, on réutilise la précédente plus une règle :

$$[\![c \texttt{ resultis } e]\!]\rho = [\![e]\!]([\![c]\!]_C\rho)$$

Rappellez vous : le changement à l'environnement apporté par `resultis` ne s'applique qu'à l'expression à sa gauche, pas le reste du programme.

(b) Appliquons notre sémantique :
$$\begin{aligned}
&[\![((x := x \dot{+} \dot{1}) \texttt{ resultis } x) \dot{+} ((y := x \dot{+} x) \texttt{ resultis } x)]\!]\rho \\
&= [\![(x := x \dot{+} \dot{1}) \texttt{ resultis } x]\!]\rho + [\![(y := x \dot{+} x) \texttt{ resultis } x]\!]\rho \\
&= [\![x]\!]([\![x := x \dot{+} \dot{1}]\!]_C\rho) + [\![x]\!]([\![y := x \dot{+} x]\!]_C\rho) \\
&= [\![x]\!](\rho[x \mapsto 4]) + [\![x]\!][x \mapsto 3] \\
&= 7
\end{aligned}$$

(c) L'instruction `i++` permet ce genre de sémantique, où les effets de bord ne sont pas forcément bien encapsulés. Ceci peut mener à des comportements inattendus du langage si on ne fait pas attention.