

INF889J Semaine 7

Le langage impératif

TÂCHE 1. On aimerait spécifier, implémenter et vérifier une commande c qui affecte le minimum des valeurs des affectables a et b à l'affectable M .

- (1) Donnez une précondition P et une postcondition Q qui assurent le comportement désiré. Veuillez utiliser le langage de formules donné dans l'section A.4. La précondition P devrait être aussi faible que possible, tandis que la postcondition Q devrait être aussi forte que possible. Vous pouvez utiliser l'abréviation $e \leq e'$ pour $(e = e' \vee e < e')$ dans vos formules logiques. Vous ne pouvez pas supposer l'existence d'un opérateur $\min(m, n)$ qui donne directement le minimum de m et n .
- (2) Donnez une commande c avec le comportement désiré. En particulier, pour tous états σ tels que $\sigma(a) = m$ et $\sigma(b) = n$, il faut que $\llbracket c \rrbracket \sigma$ soit défini et que $\sigma'(M) = \min(m, n)$ où $\sigma' = \llbracket c \rrbracket \sigma$.
- (3) Vérifiez que votre commande c satisfait votre spécification en donnant une dérivation $\Sigma \Vdash \{P\} c \{Q\}$.

1 Les produits positifs (le fragment $\otimes$)

En classe, nous avons vu les *produits négatifs* $\tau_1 \times \tau_2$. Nous explorerons maintenant les *produits positifs* $\tau_1 \otimes \tau_2$. Leur constructeur $\langle e_1, e_2 \rangle$ est analogue au constructeur $\langle e_1, e_2 \rangle$ des produits négatifs. Cependant, les produits positifs ont une sémantique « stricte », tandis que les produits négatifs ont une sémantique « paresseuse » : $\langle e_1, e_2 \rangle$ est une valeur seulement si e_1 et e_2 sont des valeurs, tandis que $\langle e_1, e_2 \rangle$ est toujours une valeur.

Quant aux destructeurs, les produits négatifs sont munis de projections $\text{proj}_1(e)$ et $\text{proj}_2(e)$ qui extraient seulement la composante désirée et la composante qui n'est pas projetée n'est jamais évaluée (un comportement paresseux). En contraste, le destructeur des produits positifs, $\text{case } e \{ \langle x_1, x_2 \rangle \Rightarrow e' \}$, permet d'accéder simultanément aux deux composantes du produit. Il commence par réduire e à une valeur $\langle e_1, e_2 \rangle$ et ensuite, le destructeur se réduit à la substitution simultanée $[e_1, e_2/x_1, x_2]e'$. Cette sémantique implique que nous évaluons toujours *les deux composantes* d'une expression de type $\tau_1 \otimes \tau_2$ lorsque nous l'utilisons (un comportement strict).

Les expressions e du fragment $\otimes$ sont données par la grammaire suivante, où la colonne de gauche donne les arbres de syntaxe abstraite, et la colonne de droite donne la syntaxe concrète correspondante :

$e ::= x$	x
$\mid \text{fuse}(e_1; e_2)$	$\langle e_1, e_2 \rangle$
$\mid \text{split}(e; x_1, x_2.e')$	$\text{case } e \{ \langle x_1, x_2 \rangle \Rightarrow e' \}$
$\mid \dots$	$\dots$

Les types τ sont données par la grammaire :

$$\tau ::= \tau_1 \otimes \tau_2 \mid \dots$$

1.1 Sémantique à petites étapes

Les paires sont des valeurs lorsque chaque composante est une valeur :

$e \text{ val}$ L'expression e est une valeur

$$\frac{e_1 \text{ val} \quad e_2 \text{ val}}{\text{fuse}(e_1; e_2) \text{ val}} \text{ (VAL/FUSE)}$$

Les réductions à petites étapes sont données par les règles suivantes :

$e \mapsto e'$ L'expression e se réduit à e' en une étape

$$\frac{e_1 \mapsto e'_1}{\text{fuse}(e_1; e_2) \mapsto \text{fuse}(e'_1; e_2)} \text{ (SOS/FUSE/1)} \quad \frac{e_1 \text{ val} \quad e_2 \mapsto e'_2}{\text{fuse}(e_1; e_2) \mapsto \text{fuse}(e_1; e'_2)} \text{ (SOS/FUSE/2)}$$

$$\frac{e_0 \mapsto e'_0}{\text{split}(e_0; x_1, x_2.e) \mapsto \text{split}(e'_0; x_1, x_2.e)} \text{ (SOS/SPLIT)}$$

$$\frac{e_1 \text{ val} \quad e_2 \text{ val}}{\text{split}(\text{fuse}(e_1; e_2); x_1, x_2.e) \mapsto [e_1, e_2/x_1, x_2]e} \text{ (SOS/SPLIT/FUSE)}$$

1.2 Système de typage

TÂCHE 2. Donnez des règles de typage pour les nouveaux opérateurs du fragment des produits positifs. Assurez-vous qu'elles assurent le progrès et la préservation.

1.3 L'interdéfinition des produits positifs et négatifs

On peut implémenter les produits positifs en termes de produits négatifs et vice-versa.

TÂCHE 3. Supposant que notre langage contient aussi le fragment λ et l'opérateur triv du fragment $\times$, implémentez les opérateurs du fragment $\times$ en terme des opérateurs du fragment $\otimes$ et des autres opérateurs du langage. En particulier, complétez le schéma suivant :

$$\begin{aligned} \tau_1 \times \tau_2 &= (\text{unit} \rightarrow \tau_1) \otimes (\text{unit} \rightarrow \tau_2) \\ \text{pair}(e_1; e_2) &= \dots \\ \text{proj}_1(e) &= \dots \\ \text{proj}_2(e) &= \dots \end{aligned}$$

Assurez-vous que votre implémentation de $\times$ respecte sa sémantique opérationnelle! Par exemple, la sémantique du fragment ne réduit jamais e_2 lors de la réduction de $\text{proj}_1(\text{pair}(e_1; e_2))$ (voir (SOS/PROJL/PAIR)). Votre implémentation ne devrait pas réduire e_2 non plus.

Indice 1. Pour comprendre la présence de $\text{unit} \rightarrow \tau_i$ dans $\tau_1 \times \tau_2 = (\text{unit} \rightarrow \tau_1) \otimes (\text{unit} \rightarrow \tau_2)$, étudiez de près la règle (VAL/FUN) et son effet sur la réduction de e dans $\text{fun}(x.e)$.

TÂCHE 4. Supposant que notre langage contient aussi le fragment λ , implémentez les opérateurs du fragment $\otimes$ en terme des opérateurs du fragment $\times$ et des autres opérateurs du langage. En particulier, complétez le schéma suivant :

$$\begin{aligned} \tau_1 \otimes \tau_2 &= \tau_1 \times \tau_2 \\ \text{fuse}(e_1; e_2) &= \dots \\ \text{split}(e; x_1, x_2.e') &= \dots \end{aligned}$$

Assurez-vous que votre implémentation de $\otimes$ respecte sa sémantique opérationnelle! Par exemple, la sémantique de $\text{split}(e; x_1, x_2.e')$ substitue seulement des valeurs pour x_1 et x_2 (voir (SOS/SPLIT/FUSE)), et $\text{fuse}(e_1; e_2)$ est seulement une valeur lorsque e_1 et e_2 le sont.

Indice 2. Étudiez de près les règles (SOS/APP/2) et (SOS/APP/FUN) et observez qu'elles nous permettent de forcer l'évaluation de certaines expressions avant de les utiliser dans un contexte paresseux.

1.4 La sémantique dénotationnelle

On peut interpréter les produits positifs en terme de produits d'ensembles :

$$\llbracket \tau_1 \otimes \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket.$$

TÂCHE 5. Donnez une sémantique dénotationnelle pour les expressions typées par (OFT/FUSE) et (OFT/SPLIT). Ce seront des fonctions

$$\llbracket \Gamma \vdash \text{fuse}(e_1; e_2) : \tau_1 \otimes \tau_2 \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket \tau_1 \otimes \tau_2 \rrbracket$$

$$\llbracket \Gamma \vdash \text{split}(e; x_1, x_2.e') : ? \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket ? \rrbracket$$

où le type de la deuxième fonction dépendra de votre réponse à la tâche 2.

A Le langage impératif

Soient les métavariabes $n \in \mathbb{Z} = \{ \dots, -2, -1, 0, 1, 2, \dots \}$ pour les nombres entiers, $a, b, c, \dots \in \mathcal{A}$ pour les noms d'affectables, et $x, y, z, \dots \in \mathcal{V}$ pour les noms de variables. Les arbres de syntaxe abstraite pour notre langage impératif sont donnés par les grammaires suivantes (colonne de gauche), où les arbres e sont des expressions et les arbres c sont des commandes.

$e ::= x$	x
$\text{get}[a]$	a
$\text{num}[n]$	n
true	true
false	false
$\text{plus}(e_1; e_2)$	$e_1 + e_2$
$\text{lt}(e_1; e_2)$	$e_1 < e_2$
$\text{if}(e_1; e_2; e_3)$	if e_1 then e_2 else e_3
$\text{let}(e_1; x.e_2)$	let $x = e_1$ in e_2
$c ::= \text{skip}$	skip
$\text{set}[a](e)$	$a := e$
$\text{seq}(c_1; c_2)$	$c_1; c_2$
$\text{cif}(e; c_1; c_2)$	if e then c_1 else c_2
$\text{while}(e; c)$	while e do c
diverge	diverge

Les contextes de commandes $C[]$ sont générés par la grammaire suivante :

$$C[] ::= [] \mid \text{seq}(C[]; c) \mid \text{seq}(c; C[]) \mid \text{cif}(e; C[]; c) \mid \text{cif}(e; c; C[]) \mid \text{while}(e; C[])$$

A.1 Système de typage

Les types τ sont générés par la grammaire suivante :

$$\tau ::= \text{int} \mid \text{bool}$$

Les contextes Γ qui donnent les types des variables sont générés par la grammaire suivante, où $\cdot$ est le contexte vide et $x : \tau$ signifie que la variable x a le type τ :

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$

On utilise un contexte Σ pour énumérer les noms d'assignables qui sont alloués :

$$\Sigma ::= \cdot \mid \Sigma, a$$

Le jugement $\Sigma \mid \Gamma \vdash e : \tau$ veut dire que l'expression e a le type τ lorsqu'il y a les affectables Σ et lorsque les variables sont typées selon Γ . Ce jugement est bien formé seulement si chaque nom d'affectable apparaît au plus une fois dans Σ et seulement si chaque variable apparaît au plus une fois dans Γ . Le jugement $\Sigma \vdash c \text{ comm}$ signifie que c est une commande bien formée lorsqu'il y a les affectables Σ .

$\boxed{\Sigma \mid \Gamma \vdash e : \tau}$ L'expression e a le type τ avec les affectables Σ et les variables typées par Γ

$$\begin{array}{c}
\frac{(x : \tau) \in \Gamma}{\Sigma \mid \Gamma \vdash x : \tau} \text{ (OFT/VAR)} \quad \frac{a \in \Sigma}{\Sigma \mid \Gamma \vdash \text{get}[a] : \text{int}} \text{ (OFT/ASN)} \quad \frac{}{\Sigma \mid \Gamma \vdash \text{num}[n] : \text{int}} \text{ (OFT/NUM)} \\
\\
\frac{}{\Sigma \mid \Gamma \vdash \text{true} : \text{bool}} \text{ (OFT/TRUE)} \quad \frac{}{\Sigma \mid \Gamma \vdash \text{false} : \text{bool}} \text{ (OFT/FALSE)} \\
\\
\frac{\Sigma \mid \Gamma \vdash e_1 : \text{int} \quad \Sigma \mid \Gamma \vdash e_2 : \text{int}}{\Sigma \mid \Gamma \vdash \text{plus}(e_1; e_2) : \text{int}} \text{ (OFT/PLUS)} \quad \frac{\Sigma \mid \Gamma \vdash e_1 : \text{int} \quad \Sigma \mid \Gamma \vdash e_2 : \text{int}}{\Sigma \mid \Gamma \vdash \text{lt}(e_1; e_2) : \text{bool}} \text{ (OFT/LT)} \\
\\
\frac{\Sigma \mid \Gamma \vdash e_1 : \text{bool} \quad \Sigma \mid \Gamma \vdash e_2 : \tau \quad \Sigma \mid \Gamma \vdash e_3 : \tau}{\Sigma \mid \Gamma \vdash \text{if}(e_1; e_2; e_3) : \tau} \text{ (OFT/IF)} \\
\\
\frac{\Sigma \mid \Gamma \vdash e_1 : \tau_1 \quad \Sigma \mid \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Sigma \mid \Gamma \vdash \text{let}(e_1; x.e_2) : \tau_2} \text{ (OFT/LET)}
\end{array}$$

$\boxed{\Sigma \vdash c \text{ comm}}$ La commande c est bien formée sous les affectables Σ

$$\begin{array}{c}
\frac{}{\Sigma \vdash \text{skip comm}} \text{ (C-SKIP)} \quad \frac{a \in \Sigma \quad \Sigma \mid \cdot \vdash e : \text{int}}{\Sigma \vdash \text{set}[a](e) \text{ comm}} \text{ (C-AFF)} \\
\\
\frac{\Sigma \vdash c_1 \text{ comm} \quad \Sigma \vdash c_2 \text{ comm}}{\Sigma \vdash \text{seq}(c_1; c_2) \text{ comm}} \text{ (C-SEQ)} \quad \frac{\Sigma \mid \cdot \vdash e : \text{bool} \quad \Sigma \vdash c_1 \text{ comm} \quad \Sigma \vdash c_2 \text{ comm}}{\Sigma \vdash \text{cif}(e; c_1; c_2) \text{ comm}} \text{ (C-IF)} \\
\\
\frac{}{\Sigma \vdash \text{diverge comm}} \text{ (C-DIVERGE)} \quad \frac{\Sigma \mid \cdot \vdash e : \text{bool} \quad \Sigma \vdash c \text{ comm}}{\Sigma \vdash \text{while}(e; c) \text{ comm}} \text{ (C-WHILE)}
\end{array}$$

A.2 Sémantique à petites étapes

Soit l'ensemble $\mathbb{S}$ de fonctions partielles finies $\mathcal{A} \rightarrow_{\text{fin}} \mathbb{Z}$. Nous appelons ses éléments des **états** et nous utilisons la métavariable $\sigma \in \mathbb{S}$ pour ses éléments. La sémantique à petites étapes pour le langage impératif est donnée par les quatre jugements suivants, tous définis par induction. En premier, on doit réduire les expressions dans la présence d'un état :

$\boxed{e \text{ val}}$ L'expression e est une valeur (peut importe l'état)

$$\frac{}{\text{num}[n] \text{ val}} \text{ (VAL/NUM)} \quad \frac{}{\text{true val}} \text{ (VAL/TRUE)} \quad \frac{}{\text{false val}} \text{ (VAL/FALSE)}$$

$\sigma \models e \mapsto e'$ L'expression e se réduit à e' en une étape sous l'état σ

$$\begin{array}{c}
\frac{\sigma(a) = n}{\sigma \models \text{get}[a] \mapsto \text{num}[n]} \text{ (SOS/ASN)} \\
\\
\frac{}{\sigma \models \text{plus}(\text{num}[m]; \text{num}[n]) \mapsto \text{num}[m+n]} \text{ (SOS/PLUS)} \\
\\
\frac{\sigma \models e_1 \mapsto e'_1}{\sigma \models \text{plus}(e_1; e_2) \mapsto \text{plus}(e'_1; e_2)} \text{ (SOS/PLUS-1)} \quad \frac{e_1 \text{ val } \sigma \models e_2 \mapsto e'_2}{\sigma \models \text{plus}(e_1; e_2) \mapsto \text{plus}(e_1; e'_2)} \text{ (SOS/PLUS-2)} \\
\\
\frac{m < n}{\sigma \models \text{lt}(\text{num}[m]; \text{num}[n]) \mapsto \text{true}} \text{ (SOS/LT-TRUE)} \\
\\
\frac{\neg(m < n)}{\sigma \models \text{lt}(\text{num}[m]; \text{num}[n]) \mapsto \text{false}} \text{ (SOS/LT-FALSE)} \\
\\
\frac{\sigma \models e_1 \mapsto e'_1}{\sigma \models \text{lt}(e_1; e_2) \mapsto \text{lt}(e'_1; e_2)} \text{ (SOS/LT-1)} \quad \frac{e_1 \text{ val } \sigma \models e_2 \mapsto e'_2}{\sigma \models \text{lt}(e_1; e_2) \mapsto \text{lt}(e_1; e'_2)} \text{ (SOS/LT-2)} \\
\\
\frac{}{\sigma \models \text{if}(\text{true}; e_1; e_2) \mapsto e_1} \text{ (SOS/IF-TRUE)} \quad \frac{}{\sigma \models \text{if}(\text{false}; e_1; e_2) \mapsto e_2} \text{ (SOS/IF-FALSE)} \\
\\
\frac{\sigma \models e_0 \mapsto e'_0}{\sigma \models \text{if}(e_0; e_1; e_2) \mapsto \text{if}(e'_0; e_1; e_2)} \text{ (SOS/IF)} \\
\\
\frac{e_1 \text{ val}}{\sigma \models \text{let}(e_1; x.e_2) \mapsto [e_1/x]e_2} \text{ (SOS/LET)} \quad \frac{\sigma \models e_1 \mapsto e'_1}{\sigma \models \text{let}(e_1; x.e_2) \mapsto \text{let}(e'_1; x.e_2)} \text{ (SOS/LET1)}
\end{array}$$

Ensuite, on doit expliquer comment réduire des *configurations*, c'est-à-dire, une commande à exécuter sur un état. On dit qu'une configuration est finale si sa commande a fini d'exécuter et si la configuration représente la fin d'une exécution qui réussit.

$\langle \sigma, c \rangle \text{ final}$ La configuration $\langle \sigma, c \rangle$ est finale

$$\frac{}{\langle \sigma, \text{skip} \rangle \text{ final}} \text{ (FIN-SKI)}$$

$$\boxed{\langle \sigma, c \rangle \mapsto \langle \sigma', c' \rangle} \quad \text{La commande } c \text{ se réduit à } c' \text{ en une étape qui transforme l'état } \sigma \text{ à } \sigma'$$

$$\frac{}{\langle \sigma, \text{set}[a](\text{num}[n]) \rangle \mapsto \langle [\sigma \mid a \mapsto n], \text{skip} \rangle} \text{ (SOS/AFF)}$$

$$\frac{\sigma \models e \mapsto e'}{\langle \sigma, \text{set}[a](e) \rangle \mapsto \langle \sigma, \text{set}[a](e') \rangle} \text{ (SOS/AFF-1)}$$

$$\frac{}{\langle \sigma, \text{seq}(\text{skip}; c) \rangle \mapsto \langle \sigma, c \rangle} \text{ (SOS/SEQ)} \quad \frac{\langle \sigma, c_1 \rangle \mapsto \langle \sigma', c'_1 \rangle}{\langle \sigma, \text{seq}(c_1; c_2) \rangle \mapsto \langle \sigma', \text{seq}(c'_1; c_2) \rangle} \text{ (SOS/SEQ-1)}$$

$$\frac{}{\langle \sigma, \text{cif}(\text{true}; c_1; c_2) \rangle \mapsto \langle \sigma, c_1 \rangle} \text{ (SOS/CIF-TRUE)} \quad \frac{}{\langle \sigma, \text{cif}(\text{false}; c_1; c_2) \rangle \mapsto \langle \sigma, c_2 \rangle} \text{ (SOS/CIF-FALSE)}$$

$$\frac{\sigma \models e \mapsto e'}{\langle \sigma, \text{if}(e; c_1; c_2) \rangle \mapsto \langle \sigma, \text{if}(e'; c_1; c_2) \rangle} \text{ (SOS/CIF)}$$

$$\frac{}{\langle \sigma, \text{diverge} \rangle \mapsto \langle \sigma, \text{diverge} \rangle} \text{ (SOS/DIVERGE)}$$

$$\frac{}{\langle \sigma, \text{while}(b; c) \rangle \mapsto \langle \sigma, \text{cif}(b; \text{seq}(c; \text{while}(b; c)); \text{skip}) \rangle} \text{ (SOS/WHILE)}$$

Définition 3. Deux commandes c_1 et c_2 sont **opérationnellement équivalentes**, écrit $c_1 \simeq c_2$, si pour tout contexte de commande $C[\]$ et tous états σ et σ' , $\langle \sigma, C[c_1] \rangle \mapsto^* \langle \sigma', \text{skip} \rangle$ si et seulement si $\langle \sigma, C[c_2] \rangle \mapsto^* \langle \sigma', \text{skip} \rangle$.

A.3 Sémantique dénotationnelle

Soit le sous-ensemble d'états $\llbracket \Sigma \rrbracket = \{ \sigma \in \mathbb{S} \mid \text{dom}(\sigma) = \Sigma \}$. Pour chaque contexte Γ , soit son ensemble d'environnements $\llbracket \Gamma \rrbracket = \prod_{x \in \text{dom}(\Gamma)} \llbracket \Gamma(x) \rrbracket$.

La dénotation d'une expression e est une fonction partielle

$$\llbracket \Sigma \mid \Gamma \vdash e : \tau \rrbracket : \llbracket \Sigma \rrbracket \rightarrow \llbracket \Gamma \rrbracket \rightarrow \llbracket \tau \rrbracket$$

définie par induction sur la structure de la dérivation de typage de e :

$$\begin{aligned}
\llbracket \Sigma \mid \Gamma \vdash x : \tau \rrbracket \sigma u &= u(x) \\
\llbracket \Sigma \mid \Gamma \vdash \text{get}[a] : \text{int} \rrbracket \sigma u &= \sigma(a) \\
\llbracket \Sigma \mid \Gamma \vdash \text{num}[n] : \text{int} \rrbracket \sigma u &= n \\
\llbracket \Sigma \mid \Gamma \vdash \text{true} : \text{bool} \rrbracket \sigma u &= \text{tt} \\
\llbracket \Sigma \mid \Gamma \vdash \text{false} : \text{bool} \rrbracket \sigma u &= \text{ff} \\
\llbracket \Sigma \mid \Gamma \vdash \text{plus}(e_1; e_2) : \text{int} \rrbracket \sigma u &= \llbracket \Sigma \mid \Gamma \vdash e_1 : \text{int} \rrbracket \sigma u + \llbracket \Sigma \mid \Gamma \vdash e_2 : \text{int} \rrbracket \sigma u \\
\llbracket \Sigma \mid \Gamma \vdash \text{lt}(e_1; e_2) : \text{bool} \rrbracket \sigma u &= (\llbracket \Sigma \mid \Gamma \vdash e_1 : \text{int} \rrbracket \sigma u < \llbracket \Sigma \mid \Gamma \vdash e_2 : \text{int} \rrbracket \sigma u) \\
\llbracket \Sigma \mid \Gamma \vdash \text{if}(e_1; e_2; e_3) : \tau \rrbracket \sigma u &= \begin{cases} \llbracket \Sigma \mid \Gamma \vdash e_2 : \tau \rrbracket \sigma u & \text{si } \llbracket \Sigma \mid \Gamma \vdash e_1 : \text{bool} \rrbracket \sigma u \\ \llbracket \Sigma \mid \Gamma \vdash e_3 : \tau \rrbracket \sigma u & \text{sinon} \end{cases} \\
\llbracket \Sigma \mid \Gamma \vdash \text{let}(e_1; x.e_2) : \tau \rrbracket \sigma u &= \llbracket \Sigma \mid \Gamma, x : \tau_1 \vdash e_2 : \tau \sigma [u \mid x \mapsto \llbracket \Sigma \mid \Gamma \vdash e_1 : \tau_1 \rrbracket \sigma u] \rrbracket
\end{aligned}$$

La dénotation d'une commande $\Sigma \vdash c \text{ comm}$ est une fonction partielle

$$\llbracket \Sigma \vdash c \text{ comm} \rrbracket : \llbracket \Sigma \rrbracket \rightarrow \llbracket \Sigma \rrbracket$$

définie par induction sur la dérivation de $\Sigma \vdash c \text{ comm}$:

$$\begin{aligned}
\llbracket \Sigma \vdash \text{skip comm} \rrbracket \sigma &= \sigma \\
\llbracket \Sigma \vdash \text{set}[a](e) \text{ comm} \rrbracket \sigma &= [\sigma \mid a \mapsto \llbracket \Sigma \mid \cdot \vdash e : \text{int} \rrbracket \sigma \emptyset] \\
\llbracket \Sigma \vdash \text{seq}(c_1; c_2) \text{ comm} \rrbracket \sigma &= \llbracket \Sigma \vdash c_2 \text{ comm} \rrbracket (\llbracket \Sigma \vdash c_1 \text{ comm} \rrbracket \sigma) \\
\llbracket \Sigma \vdash \text{cif}(e; c_1; c_2) \text{ comm} \rrbracket \sigma &= \begin{cases} \llbracket \Sigma \vdash c_1 \text{ comm} \rrbracket \sigma & \text{si } \llbracket [\vdash \Sigma \text{ comm}] \cdot e \text{bool} \rrbracket \sigma \emptyset = tt \\ \llbracket \Sigma \vdash c_2 \text{ comm} \rrbracket \sigma & \text{si } \llbracket [\vdash \Sigma \text{ comm}] \cdot e \text{bool} \rrbracket \sigma \emptyset = ff \end{cases} \\
\llbracket \Sigma \vdash \text{diverge comm} \rrbracket \sigma &= \text{non défini}
\end{aligned}$$

La fonction partielle $\llbracket \Sigma \vdash \text{while}(e; c) \text{ comm} \rrbracket$ est donnée par le graphe :

$$\text{graphe}(\llbracket \Sigma \vdash \text{while}(e; c) \text{ comm} \rrbracket) = \bigcup_{i \in \mathbb{N}} \text{graphe}(\beta_i)$$

où les $\beta_i : \llbracket \Sigma \rrbracket \rightarrow \llbracket \Sigma \rrbracket$ sont définis par induction sur $i \in \mathbb{N}$:

$$\begin{aligned}
\beta_0(\sigma) &= \text{non défini} \\
\beta_{i+1}(\sigma) &= \begin{cases} \llbracket \Sigma \vdash c \text{ comm} \rrbracket (\beta_i(\sigma)) & \text{si } \llbracket \Sigma \mid \cdot \vdash e : \text{bool} \rrbracket \sigma \emptyset = tt \\ \sigma & \text{si } \llbracket \Sigma \mid \cdot \vdash e : \text{bool} \rrbracket \sigma \emptyset = ff \end{cases}
\end{aligned}$$

Définition 4. Deux commandes $\Sigma \vdash c_1 \text{ comm}$ et $\Sigma \vdash c_2 \text{ comm}$ sont **dénotationnellement équivalentes**, écrit $c_1 \equiv c_2$, si $\llbracket \Sigma \vdash c_1 \text{ comm} \rrbracket = \llbracket \Sigma \vdash c_2 \text{ comm} \rrbracket$.

A.4 Sémantique axiomatique

On élargit notre syntaxe d'expressions pour permettre diverses opérations arithmétiques dans nos spécifications de programmes : ¹

$$e ::= e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1^{e_2} \mid \dots$$

On fixe un langage de formules logiques P, Q, R du premier ordre :

$$P, Q, R ::= (e = e') \mid (e < e') \mid \text{true} \mid \text{false} \mid \neg P \mid P \supset Q \mid P \vee Q \mid P \wedge Q \mid \dots$$

On suppose que les expressions e qui apparaissent dans les formules sont closes : elles n'ont pas de variables libres, mais peuvent contenir des noms d'affectables.

On suppose qu'on a un système de logique du premier ordre, où on écrit $\Sigma \vdash P$ pour dire que P est prouvable dans ce système, et on suppose que ce système peut démontrer diverses propriétés mathématiques. Par exemple, $\Sigma \vdash (a > 5) \supset (a > 0)$ et $\Sigma \vdash (a + b = 5) \supset (a = 5 - b)$.

Les formules logiques ont la sémantique dénotationnelle $\llbracket \Sigma \vdash P \rrbracket : \llbracket \Sigma \rrbracket \rightarrow \mathbb{B}$ évidente. On dit qu'un état $\sigma \in \llbracket \Sigma \rrbracket$ **satisfait** P si $\llbracket \Sigma \vdash P \rrbracket \sigma = tt$. Par exemple, l'état $[a \mapsto 5, b \mapsto 3]$ satisfait $a > 0$, $a \times b = 15$ et $(2 \times a) > (9 - a)$, mais ne satisfait pas $b > a$, $a - 5 > 2 \times a$, ou $a = 3 \vee b = 5$.

On écrit $[a := e]e'$ et $[a := e]P$ pour le remplacement de toutes occurrences de $\text{get}[a]$ par l'expression e dans e' et P , respectivement. Cette opération est définie par induction sur la syntaxe de e' ou de P de la manière évidente, par exemple :

$$\begin{aligned}
[a := e](\text{get}[b]) &= \begin{cases} e & \text{si } a = b \\ \text{get}[b] & \text{sinon} \end{cases} & [a := e]\text{true} &= \text{true} \\
[a := e](e_1 > e_2) &= ([a := e]e_1) > ([a := e]e_2) & [a := e](P \wedge Q) &= ([a := e]P) \wedge ([a := e]Q)
\end{aligned}$$

La sémantique axiomatique est donnée par un jugement $\{P\} c \{Q\}$ qu'on appelle un triple de Hoare. Ce jugement syntaxique veut dire que si l'exécution de c à partir d'un état qui satisfait P

1. En principe, la grammaire est donnée en syntaxe abstraite, mais on utilise la syntaxe concrète pour la lisibilité.

termine, l'état qui en résulte satisfera Q . Un triple de Hoare $\Sigma \Vdash \{P\} c \{Q\}$ est **vrai** si pour tous états $\sigma, \sigma' \in \llbracket \Sigma \rrbracket$, si $\llbracket \Sigma \vdash P \rrbracket \sigma = tt$ et $\llbracket \Sigma \vdash c \text{ comm} \rrbracket \sigma = \sigma'$, alors $\llbracket \Sigma \vdash Q \rrbracket \sigma' = tt$. Il est **faux** s'il existe deux états $\sigma, \sigma' \in \llbracket \Sigma \rrbracket$ tels que $\llbracket \Sigma \vdash P \rrbracket \sigma = tt$ et $\llbracket \Sigma \vdash c \text{ comm} \rrbracket \sigma = \sigma'$, mais $\llbracket \Sigma \vdash Q \rrbracket \sigma' \neq tt$.

Le jugement $\Sigma \Vdash \{P\} c \{Q\}$, défini par induction par les règles suivantes, dit qu'un triple de Hoare $\{P\} c \{Q\}$ est **démontrable**. Ces règles nous donnent un système formel pour démontrer de façon syntaxique diverses propriétés des programmes.

$\Sigma \Vdash \{P\} c \{Q\}$ Le triple de Hoare $\{P\} c \{Q\}$ est démontrable

$$\begin{array}{c}
\frac{}{\Sigma \Vdash \{P\} \text{ skip } \{P\}} \text{ (H-SKIP)} \quad \frac{}{\Sigma \Vdash \{[a := e]P\} \text{ set}[a](e) \{P\}} \text{ (H-AFF)} \\
\frac{\Sigma \Vdash \{P\} c_1 \{Q\} \quad \Sigma \Vdash \{Q\} c_2 \{R\}}{\Sigma \Vdash \{P\} \text{ seq}(c_1; c_2) \{R\}} \text{ (H-SEQ)} \quad \frac{\Sigma \Vdash \{P \wedge e\} c_1 \{Q\} \quad \Sigma \Vdash \{P \wedge \neg e\} c_2 \{Q\}}{\Sigma \Vdash \{P\} \text{ cif}(e; c_1; c_2) \{Q\}} \text{ (H-IF)} \\
\frac{}{\Sigma \Vdash \{\text{true}\} \text{ diverge } \{\text{false}\}} \text{ (H-DIVERGE)} \quad \frac{\Sigma \Vdash \{P \wedge e\} c \{P\}}{\Sigma \Vdash \{P\} \text{ while}(e; c) \{P \wedge \neg e\}} \text{ (H-WHILE)} \\
\frac{\vdash P \supset P' \quad \Sigma \Vdash \{P'\} c \{Q\}}{\Sigma \Vdash \{P\} c \{Q\}} \text{ (H-STR)} \quad \frac{\Sigma \Vdash \{P\} c \{Q'\} \quad \vdash Q' \supset Q}{\Sigma \Vdash \{P\} c \{Q\}} \text{ (H-WEA)} \\
\frac{\Sigma \Vdash \{P\} c \{Q_1\} \quad \Sigma \Vdash \{P\} c \{Q_2\}}{\Sigma \Vdash \{P\} c \{Q_1 \wedge Q_2\}} \text{ (H-AND)} \quad \frac{\Sigma \Vdash \{P_1\} c \{Q\} \quad \Sigma \Vdash \{P_2\} c \{Q\}}{\Sigma \Vdash \{P_1 \vee P_2\} c \{Q\}} \text{ (H-OR)}
\end{array}$$

Il se pourrait que nos règles soient erronées. La correction (*soundness*) est une propriété d'une logique par rapport à sa sémantique qui dit que tout ce qui est démontrable en utilisant les règles de la logique est vrai dans la sémantique. La logique de Hoare est correcte, c'est-à-dire, les règles nous permettent seulement de démontrer des choses qui sont vraies dans la sémantique :

THÉORÈME 5 (CORRECTION). Si $\Sigma \Vdash \{P\} c \{Q\}$, alors $\Sigma \Vdash \{P\} c \{Q\}$ est vrai.

COROLLAIRE 6. Si $\Sigma \Vdash \{P\} c \{Q\}$ est faux, alors il n'y a pas de dérivation de $\Sigma \Vdash \{P\} c \{Q\}$.

Définition 7. La sémantique axiomatique induit une notion d'équivalence. Deux commandes c_1 et c_2 sont **démontrablement équivalentes**, $c_1 \Leftrightarrow c_2$, si pour tout P et Q , $\Sigma \Vdash \{P\} c_1 \{Q\}$ si et seulement si $\Sigma \Vdash \{P\} c_2 \{Q\}$.

B Les langages fonctionnels

On a vu quelques extensions du langage d'expressions. Les sous-sections suivantes présentent ces diverses extensions. On peut combiner ces fragments de langages. Par exemple, le langage $\lambda \times$ est formé des opérateurs et règles des fragments λ et $\times$.

B.1 Les fonctions d'ordre supérieur (le fragment λ)

Les expressions e du fragment λ sont données par la grammaire suivante, où la colonne de gauche donne les arbres de syntaxe abstraite, et la colonne de droite donne la syntaxe concrète correspondante :

$$\begin{array}{ll}
e ::= x & x \\
\mid \text{fun}(x.e) & \lambda x.e \\
\mid \text{app}(e_1; e_2) & e_1 e_2 \\
\mid \dots & \dots
\end{array}$$

Les types τ sont données par la grammaire :

$$\tau ::= \tau_1 \rightarrow \tau_2 \mid \dots$$

B.1.1 Système de typage. Les expressions typées par les règles suivantes :

$\boxed{\Gamma \vdash e : \tau}$ L'expression e a le type τ lorsque les variables typées par Γ

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \text{ (OFT/VAR)} \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fun}(x.e) : \tau_1 \rightarrow \tau_2} \text{ (OFT/FUN)}$$

$$\frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \text{app}(e_1; e_2) : \tau} \text{ (OFT/APP)}$$

B.1.2 Sémantique dénotationnelle. La dénotation du type $\tau_1 \rightarrow \tau_2$ est l'ensemble de fonctions

$$\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \{ f \mid f : \llbracket \tau_1 \rrbracket \rightarrow \llbracket \tau_2 \rrbracket \}.$$

Lorsque Γ est un contexte, on écrit $\Gamma(x)$ pour τ lorsque $(x : \tau) \in \Gamma$. Quand $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$, cette correspondance détermine une fonction $\Gamma : \{x_1, \dots, x_n\} \rightarrow \{\tau_1, \dots, \tau_n\}$. On utilise cette correspondance pour définir la dénotation d'un contexte $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$. Elle est l'ensemble

$$\llbracket \Gamma \rrbracket = \prod_{x \in \text{dom}(\Gamma)} \llbracket \Gamma(x) \rrbracket$$

qui a comme éléments les fonctions dépendantes u qui ont comme domaine $\{x_1, \dots, x_n\}$ et où $u(x_i) \in \llbracket \Gamma(x_i) \rrbracket = \llbracket \tau_i \rrbracket$. La sémantique dénotationnelle intrinsèque est donnée par induction sur la dérivation du jugement $\Gamma \vdash e : \tau$, où $\llbracket \Gamma \vdash e : \tau \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket \tau \rrbracket$:

$$\begin{aligned} \llbracket \Gamma \vdash x : \tau \rrbracket u &= u(x) \\ \llbracket \Gamma \vdash \text{fun}(x.e) : \tau_1 \rightarrow \tau_2 \rrbracket u &= f : \llbracket \tau_1 \rrbracket \rightarrow \llbracket \tau_2 \rrbracket \text{ où } f(v) = \llbracket \Gamma, x : \tau_1 \vdash e : \tau_2 \rrbracket [u \mid x \mapsto v] \\ \llbracket \Gamma \vdash \text{app}(e_1; e_2) : \tau \rrbracket u &= \llbracket \Gamma \vdash e_1 : \tau' \rightarrow \tau \rrbracket u(\llbracket \Gamma \vdash e_2 : \tau' \rrbracket u) \end{aligned}$$

B.1.3 Sémantique à petites étapes. Les fonctions sont des valeurs :

$\boxed{e \text{ val}}$ L'expression e est une valeur

$$\overline{\text{fun}(x.e) \text{ val}} \text{ (VAL/FUN)}$$

Les réductions à petites étapes sont données par les règles suivantes :

$\boxed{e \mapsto e'}$ L'expression e se réduit à e' en une étape

$$\frac{e_1 \mapsto e'_1}{\text{app}(e_1; e_2) \mapsto \text{app}(e'_1; e_2)} \text{ (SOS/APP/1)} \quad \frac{e_1 \text{ val} \quad e_2 \mapsto e'_2}{\text{app}(e_1; e_2) \mapsto \text{app}(e_1; e'_2)} \text{ (SOS/APP/2)}$$

$$\frac{e_2 \text{ val}}{\text{app}(\text{fun}(x.e_1); e_2) \mapsto [e_2/x]e_1} \text{ (SOS/APP/FUN)}$$

B.2 Les produits négatifs (le fragment $\times$)

Les expressions e du fragment $\times$ sont données par la grammaire suivante, où la colonne de gauche donne les arbres de syntaxe abstraite, et la colonne de droite donne la syntaxe concrète

correspondante :

$e ::= \text{triv}$	$\langle \rangle$
$ \text{pair}(e_1; e_2)$	$\langle e_1, e_2 \rangle$
$ \text{proj}_1(e)$	$e \cdot 1$
$ \text{proj}_2(e)$	$e \cdot 2$
$ \dots$	$\dots$

Les types τ sont données par la grammaire :

$$\tau ::= \text{unit} \mid \tau_1 \times \tau_2 \mid \dots$$

B.2.1 Système de typage. Les expressions typées par les règles suivantes :

$\boxed{\Gamma \vdash e : \tau}$ L'expression e a le type τ lorsque les variables typées par Γ

$$\begin{array}{c} \overline{\Gamma \vdash \text{triv} : \text{unit}} \quad (\text{OFT/TRIV}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{pair}(e_1; e_2) : \tau_1 \times \tau_2} \quad (\text{OFT/PAIR}) \\[10pt] \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{proj}_1(e) : \tau_1} \quad (\text{OFT/PROJL}) \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{proj}_2(e) : \tau_2} \quad (\text{OFT/PROJR}) \end{array}$$

B.2.2 Sémantique dénotationnelle. La dénotation du type unit est un ensemble avec un élément :

$$\llbracket \text{unit} \rrbracket = \{ * \}.$$

La dénotation du type $\tau_1 \times \tau_2$ est le produit cartésien

$$\llbracket \tau_1 \times \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket = \{ (v_1, v_2) \mid v_1 \in \llbracket \tau_1 \rrbracket, v_2 \in \llbracket \tau_2 \rrbracket \}.$$

Pour tout produit cartésien $A \times B$ d'ensembles A et B , il existe des fonctions de projection $\pi_1 : A \times B \rightarrow A$ et $\pi_2 : A \times B \rightarrow B$ telles que $\pi_1(a, b) = a$, $\pi_2(a, b) = b$, et $x = (\pi_1(x), \pi_2(x))$ pour tout $x \in A \times B$.

La sémantique dénotationnelle intrinsèque des expressions est donnée par induction sur la dérivation du jugement $\Gamma \vdash e : \tau$, où $\llbracket \Gamma \vdash e : \tau \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket \tau \rrbracket$:

$$\begin{aligned} \llbracket \Gamma \vdash \text{triv} : \text{unit} \rrbracket u &= * \\ \llbracket \Gamma \vdash \text{pair}(e_1; e_2) : \tau_1 \times \tau_2 \rrbracket u &= (\llbracket \Gamma \vdash e_1 : \tau_1 \rrbracket u, \llbracket \Gamma \vdash e_2 : \tau_2 \rrbracket u) \\ \llbracket \Gamma \vdash \text{proj}_1(e) : \tau_1 \rrbracket u &= \pi_1(\llbracket \Gamma \vdash e : \tau_1 \times \tau_2 \rrbracket u) \\ \llbracket \Gamma \vdash \text{proj}_2(e) : \tau_2 \rrbracket u &= \pi_2(\llbracket \Gamma \vdash e : \tau_1 \times \tau_2 \rrbracket u) \end{aligned}$$

B.2.3 Sémantique à petites étapes. Les paires négatives sont des valeurs :

$\boxed{e \text{ val}}$ L'expression e est une valeur

$$\overline{\text{triv val}} \quad (\text{VAL/TRIV}) \qquad \overline{\text{pair}(e_1; e_2) \text{ val}} \quad (\text{VAL/PAIR})$$

Les réductions à petites étapes sont données par les règles suivantes :

$\boxed{e \mapsto e'}$ L'expression e se réduit à e' en une étape

$$\begin{array}{c} \overline{\text{proj}_1(\text{pair}(e_1; e_2)) \mapsto e_1} \quad (\text{SOS/PROJL/PAIR}) \qquad \overline{\text{proj}_2(\text{pair}(e_1; e_2)) \mapsto e_2} \quad (\text{SOS/PROJR/PAIR}) \\[10pt] \frac{e \mapsto e'}{\text{proj}_1(e) \mapsto \text{proj}_1(e')} \quad (\text{SOS/PROJL}) \qquad \frac{e \mapsto e'}{\text{proj}_2(e) \mapsto \text{proj}_2(e')} \quad (\text{SOS/PROJR}) \end{array}$$

B.3 Les sommes (le fragment +)

Les expressions e du fragment $+$ sont données par la grammaire suivante, où la colonne de gauche donne les arbres de syntaxe abstraite, et la colonne de droite donne la syntaxe concrète correspondante :

$e ::= \text{abort}(e)$	abort (e)
$\text{case}(e_0; x_1.e_1; x_2.e_2)$	case $e_0 \{ l \cdot x_1 \hookrightarrow e_1 \mid r \cdot x_2 \hookrightarrow e_2 \}$
$\text{inj}[l](e)$	$l \cdot e$
$\text{inj}[r](e)$	$r \cdot e$
$\dots$	$\dots$

Les types τ sont données par la grammaire :

$$\tau ::= \text{void} \mid \tau_1 + \tau_2 \mid \dots$$

B.3.1 Système de typage. Les expressions typées par les règles suivantes :

$\Gamma \vdash e : \tau$ L'expression e a le type τ lorsque les variables typées par Γ

$$\frac{}{\Gamma \vdash \text{abort}(e) : \tau} \text{ (OFT/ABORT)}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{inj}[l](e) : \tau_1 + \tau_2} \text{ (OFT/INJL)} \quad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{inj}[r](e) : \tau_1 + \tau_2} \text{ (OFT/INJR)}$$

$$\frac{\Gamma \vdash e_0 : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \text{case}(e_0; x_1.e_1; x_2.e_2) : \tau} \text{ (OFT/CASE)}$$

B.3.2 Sémantique dénotationnelle. La dénotation du type void est l'ensemble vide :

$$\llbracket \text{void} \rrbracket = \emptyset.$$

Pour tout ensemble A , il existe une fonction unique $!_A : \emptyset \rightarrow A$. Elle a comme graphe l'ensemble $\text{graphe}(!_A) = \emptyset$.

La dénotation du type $\tau_1 + \tau_2$ est l'union disjointe

$$\llbracket \tau_1 + \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket + \llbracket \tau_2 \rrbracket = \{ 1 \} \times \llbracket \tau_1 \rrbracket \cup \{ 2 \} \times \llbracket \tau_2 \rrbracket.$$

Pour tous ensembles A et B , il existe des fonctions d'injection $\iota_1 : A \rightarrow A + B$ et $\iota_2 : B \rightarrow A + B$ telles que $\iota_1(a) = (1, a)$ et $\iota_2(b) = (2, b)$. La sémantique dénotationnelle intrinsèque des expressions est donnée par induction sur la dérivation du jugement $\Gamma \vdash e : \tau$, où $\llbracket \Gamma \vdash e : \tau \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket \tau \rrbracket$:

$$\begin{aligned} \llbracket \Gamma \vdash \text{abort}(e) : \tau \rrbracket u &= !_\llbracket \tau \rrbracket (\llbracket \Gamma \vdash e : \text{void} \rrbracket u), \\ \llbracket \Gamma \vdash \text{case}(e_0; x_1.e_1; x_2.e_2) : \tau \rrbracket u &= \llbracket \Gamma, x_i : \tau_i \vdash e_i : \tau \rrbracket [u \mid x_i \mapsto v] \\ &\quad \text{où } (i, v) = \llbracket \Gamma \vdash e_0 : \tau_1 + \tau_2 \rrbracket u, \\ \llbracket \Gamma \vdash \text{inj}[l](e) : \tau_1 + \tau_2 \rrbracket u &= \iota_1(\llbracket \Gamma \vdash e : \tau_1 \rrbracket u), \\ \llbracket \Gamma \vdash \text{inj}[r](e) : \tau_1 + \tau_2 \rrbracket u &= \iota_2(\llbracket \Gamma \vdash e : \tau_2 \rrbracket u). \end{aligned}$$

B.3.3 Sémantique à petites étapes. Les valeurs étiquetées sont des valeurs :

$e \text{ val}$ L'expression e est une valeur

$$\frac{e \text{ val}}{\text{inj}[l](e) \text{ val}} \text{ (VAL/INJL)} \quad \frac{e \text{ val}}{\text{inj}[r](e) \text{ val}} \text{ (VAL/INJR)}$$

Les réductions à petites étapes sont données par les règles suivantes :

$e \mapsto e'$

L'expression e se réduit à e' en une étape

$$\frac{e \mapsto e'}{\text{abort}(e) \mapsto \text{abort}(e')} \text{ (SOS/ABORT)}$$

$$\frac{e \mapsto e'}{\text{inj}[l](e) \mapsto \text{inj}[l](e')} \text{ (SOS/INJL)} \quad \frac{e \mapsto e'}{\text{inj}[r](e) \mapsto \text{inj}[r](e')} \text{ (SOS/INJR)}$$

$$\frac{e \text{ val}}{\text{case}(\text{inj}[l](e); x_1.e_1; x_2.e_2) \mapsto [e/x_1]e_1} \text{ (SOS/CASE/INJL)}$$

$$\frac{e \text{ val}}{\text{case}(\text{inj}[r](e); x_1.e_1; x_2.e_2) \mapsto [e/x_2]e_2} \text{ (SOS/CASE/INJR)}$$