Chapter 2

Sûreté du typage et sémantique à réduction

Le but du typage statique est d'éliminer toute une classe de programmes absurdes, comme par exemple 1 2 ou $1 + (\texttt{fun}\ x \to x)$. Dans ce cours, nous allons prouver que c'est le cas pour les systèmes de types introduits au chapitre 1. Cette propriété que tout programme bien typé s'évalue "sans problèmes" est appelée *sûreté* du typage vis-à-vis de l'évaluation.

2.1 Distinguer erreurs d'exécution et non-terminaison

Avant tout, il faut définir les "problèmes" qui peuvent se produire pendant l'évaluation et qu'un système de types sûr doit empêcher. Il s'agit des erreurs à l'exécution correspondant à l'application d'une opération de base sur des arguments incorrects: application d'une expression qui ne s'évalue ni en une fonction ni en un opérateur (exemple: 1 2); application d'un opérateur à un argument du mauvais type (exemple: `+` appliqué à un argument qui n'est pas une paire d'entiers; `fst` appliqué à un argument qui n'est pas une paire). Une autre erreur que l'on veut éviter est l'évaluation d'une variable libre (si le terme de départ n'était pas clos).

Dans une implémentation grandeur nature d'un langage, on ne veut pas (pour des raisons de rapidité et de compacité du code généré) tester explicitement à l'exécution si ces erreurs se produisent. Donc, l'exécution d'une de ces expressions erronées peut soit provoquer un "plantage" du programme (*segmentation fault*), soit continuer l'exécution avec un résultat absurde (par exemple, $1 + (\texttt{fun}\ x \to x)$ renvoie un plus l'adresse mémoire représentant la fonction). On compte sur le typage statique pour assurer que cela ne se produira pas.

2.1.1 Tout programme bien typé s'évalue-t'il en une valeur?

La sémantique opérationnelle structurée du chapitre 1 n'a pas de règles d'évaluation qui s'applique à ces expressions erronées. Autrement dit, si a est une expression dont l'évaluation provoque une de ces erreurs, il n'existe pas de valeur v telle que $a \xrightarrow{v} v$. Naïvement, nous pourrions prendre ceci comme une caractérisation des programmes erronés, et essayer de prouver la propriété suivante:

> Si a est bien typée, alors il existe une valeur v telle que $a \xrightarrow{v} v$.

Le problème de cette approche est qu'il y a une autre classe d'expressions qui ne s'évalue pas en une valeur, mais pourtant ne déclenche pas d'erreurs à l'exécution: les expressions qui ne terminent pas. (Leur calcul "boucle" sans jamais effectuer d'opération incorrecte.)

Certains systèmes de types ne laissent passer que des programmes qui terminent toujours. On dit que ce sont des systèmes de types *fortement normalisants*. (C'est le cas des systèmes de types du chapitre 1 tant qu'on n'y ajoute pas un opérateur de point fixe.) Ces systèmes de types sont très importants dans le monde de la logique constructive, mais pas très intéressants pour les langages de programmation. En général, on souhaite qu'un langage de programmation soit Turing-complet, c'est-à-dire qu'il puisse exprimer toutes les fonctions calculables; étant donné l'indécidabilité du problème de l'arrêt, cela veut dire que leur système de types ne peut pas laisser passer tous les programmes qui terminent et rejeter tous ceux qui ne terminent pas.

Pour cette raison, nous allons considérer des systèmes de types qui ne garantissent pas que les programmes bien typés terminent. Un exemple simple est le système de types de mini-ML du chapitre 1 muni de l'opérateur de point fixe `fix`. Pour un langage tel que mini-ML + `fix`, la propriété

> Si a est bien typée, alors il existe une valeur v telle que $a \xrightarrow{v} v$.

est fausse. Par exemple,

```
let fact = fix(fun fact → fun n → if n = 0 then 1 else n * fact(n-1))
in fact (-1)
```

est bien typée (vérifiez-le!), mais ne termine pas, et donc ne s'évalue en aucune valeur v. Il faut donc trouver une autre caractérisation des programmes erronés.

2.1.2 Règles d'erreurs en sémantique opérationnelle structurelle

Dans le cadre de la sémantique opérationnelle structurelle, la technique standard est d'ajouter des règles d'évaluation qui détectent les erreurs à l'exécution et renvoient une constante spéciale `err` en résultat pour signaler qu'une opération erronée a été effectuée. La relation d'évaluation devient $a \xrightarrow{v} r$, où r est un *résultat*: ou bien une valeur v si l'évaluation de a s'est bien passée, ou bien `err` si l'évaluation de a provoque une erreur. On ajoute des règles d'évaluation pour détecter les erreurs et produire le résultat `err`:

$$x \xrightarrow{v} \texttt{err} \quad (10) \qquad \frac{a_1 \xrightarrow{v} v_1 \quad v_1 \text{ n'est ni } \texttt{fun}\ x \to a \text{ ni } op}{a_1\ a_2 \xrightarrow{v} \texttt{err}} \quad (11)$$

$$\frac{a_1 \xrightarrow{v} + \quad a_2 \xrightarrow{v} v_2 \quad v_2 \text{ n'est pas une paire d'entiers}}{a_1\ a_2 \xrightarrow{v} \texttt{err}} \quad (12)$$

$$\frac{a_1 \xrightarrow{v} \texttt{fst} \quad a_2 \xrightarrow{v} v_2 \quad v_2 \text{ n'est pas une paire}}{a_1\ a_2 \xrightarrow{v} \texttt{err}} \quad (13)$$

$$\frac{a_1 \xrightarrow{v} \texttt{snd} \quad a_2 \xrightarrow{v} v_2 \quad v_2 \text{ n'est pas une paire}}{a_1\ a_2 \xrightarrow{v} \texttt{err}} \quad (14)$$

Il faut aussi ajouter des règles pour propager `err` vers le haut: si l'évaluation d'une sous-expression produit une erreur, l'évaluation de l'expression tout entière la produit aussi.

$$\frac{a_1 \xrightarrow{v} \texttt{err}}{a_1\ a_2 \xrightarrow{v} \texttt{err}}\ (15) \qquad \frac{a_1 \xrightarrow{v} v_1 \quad a_2 \xrightarrow{v} \texttt{err}}{a_1\ a_2 \xrightarrow{v} \texttt{err}}\ (16) \qquad \frac{a_1 \xrightarrow{v} \texttt{err}}{(a_1, a_2) \xrightarrow{v} \texttt{err}}\ (17)$$

$$\frac{a_1 \xrightarrow{v} v_1 \quad a_2 \xrightarrow{v} \texttt{err}}{(a_1, a_2) \xrightarrow{v} \texttt{err}}\ (18) \qquad \frac{a_1 \xrightarrow{v} \texttt{err}}{\texttt{let}\ x = a_1\ \texttt{in}\ a_2 \xrightarrow{v} \texttt{err}}\ (19) \qquad \frac{a_1 \xrightarrow{v} v_1 \quad a_2 \xrightarrow{v} \texttt{err}}{\texttt{let}\ x = a_1\ \texttt{in}\ a_2 \xrightarrow{v} \texttt{err}}\ (20)$$

La propriété de sûreté du typage s'énonce alors ainsi:

> Si a est bien typée et $a \xrightarrow{v} r$, alors $r \neq \texttt{err}$.

Autrement dit, une expression a bien typée peut s'évaluer en une valeur ($a \xrightarrow{v} v$) ou bien ne pas terminer ($a \not\xrightarrow{v} r$ pour tout r), mais ne peut pas provoquer une erreur d'exécution ($a \xrightarrow{v} \texttt{err}$).

Cette propriété de sûreté est vraie, et se prouve par récurrence structurelle sur la dérivation de $a \xrightarrow{v} r$. Cependant, cette approche n'est pas entièrement satisfaisante, pour plusieurs raisons. Tout d'abord, il faut ajouter beaucoup de règles, et cela rend peu lisible la sémantique du langage. De plus, il y a un gros risque d'oublier d'ajouter certaines règles d'erreur. Par exemple, si nous oublions la règle 12, la propriété de sûreté ci-dessus reste vraie (puisqu'il y a moins de programmes a qui vont s'évaluer en `err`), mais elle ne nous garantit plus qu'il est inutile de vérifier à l'exécution que les deux arguments de `+` sont des entiers. Rien ne nous prouve formellement que nous avons mis toutes les bonnes règles d'erreur et que donc les vérifications à l'exécution sont inutiles.

Pour ces raisons, nous allons abandonner la sémantique opérationnelle structurelle et introduire un nouveau style de sémantique, la sémantique à réductions, qui nous permettra de prouver un résultat plus fort de sûreté du typage.

2.2 Sémantique à réductions

La sémantique opérationnelle à grands pas (*big-step semantics*) $a \xrightarrow{v} r$ ne s'intéresse pas aux étapes du calcul, mais seulement au résultat final r. Autrement dit, tout se passe comme si on faisait un grand saut du programme initial a vers son résultat r. Au contraire, la sémantique à réduction, aussi appelée sémantique à petits pas (*small-step semantics*) décrit toutes les étapes intermédiaires du calcul – la progression du programme initial vers son résultat final. Elle se présente sous la forme d'une relation $a \to a'$, où a' est l'expression obtenue à partir de a par une seule étape de calcul. Pour évaluer complètement l'expression, il faut bien sûr enchaîner plusieurs étapes

$$a \to a_1 \to a_2 \to \ldots \to v$$

jusqu'à ce qu'on atteigne une valeur (une expression complètement évaluée).

Pour définir la relation "se réduit en une étape" $\to$, on commence par se donner des axiomes pour la relation $\xrightarrow{\varepsilon}$ "se réduit en une étape en tête du terme". Les deux axiomes de base sont la β réduction pour les fonctions et pour le `let`

$$\begin{aligned} (\texttt{fun}\ x \to a)\ v &\xrightarrow{\varepsilon} a\{x \leftarrow v\} \qquad (\beta_{fun}) \\ (\texttt{let}\ x = v\ \texttt{in}\ a) &\xrightarrow{\varepsilon} a\{x \leftarrow v\} \qquad (\beta_{let}) \end{aligned}$$

On se donne aussi des axiomes pour les opérateurs complètement appliqués. Ces axiomes s'appellent aussi des δ-règles et dépendent bien sûr des opérateurs considérés. Voici les δ-règles pour `+`, `fst`, `snd`, `ifthenelse`, et `fix`:

$$
\begin{array}{rcll}
+\,(n_1, n_2) & \xrightarrow{\varepsilon} & n \text{ si } n_1, n_2 \text{ entiers et } n = n_1 + n_2 & (\delta_+) \\
\texttt{fst}\ (v_1, v_2) & \xrightarrow{\varepsilon} & v_1 & (\delta_{fst}) \\
\texttt{snd}\ (v_1, v_2) & \xrightarrow{\varepsilon} & v_2 & (\delta_{snd}) \\
\texttt{fix}\ (\texttt{fun}\ x \to a) & \xrightarrow{\varepsilon} & a\{x \leftarrow \texttt{fix}\ (\texttt{fun}\ x \to a)\} & (\delta_{fix}) \\
\texttt{ifthenelse}(\texttt{true}, (a_1, a_2)) & \xrightarrow{\varepsilon} & a_1 & (\delta_{if}) \\
\texttt{ifthenelse}(\texttt{false}, (a_1, a_2)) & \xrightarrow{\varepsilon} & a_2 & (\delta_{if'})
\end{array}
$$

Bien sûr, on ne réduit pas toujours en tête de l'expression. Considérons par exemple

$$a = (\texttt{let}\ x = +(1, 2)\ \texttt{in}\ +\,(x, 3))$$

Aucun des axiomes ci-dessus ne s'applique à a. Pour évaluer a, il est clair qu'il faut commencer par réduire "en profondeur" la sous-expression $+(1, 2)$. Cette notion de réduction en profondeur est exprimée par la règle d'inférence suivante:

$$\frac{a \xrightarrow{\varepsilon} a'}{\Gamma(a) \to \Gamma(a')}\ \text{(contexte)}$$

Dans cette règle, Γ est un *contexte d'évaluation*. Les contextes d'évaluation sont définis par la syntaxe abstraite suivante:

Contextes d'évaluation:

$\Gamma ::= [\,]$	évaluation en tête
$\mid \Gamma\ a$	évaluation à gauche d'une application
$\mid v\ \Gamma$	évaluation à droite d'une application
$\mid \texttt{let}\ x = \Gamma\ \texttt{in}\ a$	évaluation à gauche d'un `let`
$\mid (\Gamma, a)$	évaluation à gauche d'une paire
$\mid (v, \Gamma)$	évaluation à droite d'une paire

Rappels sur les contextes: Un contexte est une expression avec un "trou", noté $[\,]$. Par exemple, $+([\,], 2)$. L'opération de base sur un contexte C est l'application $C(a)$ à une expression a. $C(a)$ est l'expression dénotée par C dans laquelle le "trou" $[\,]$ est remplacé par a. Par exemple, $+([\,], 2)$ appliqué à 1 est l'expression $+(1, 2)$.

Les contextes d'évaluation Γ ne sont pas n'importe quelle expression avec un trou. Par exemple, $(+(1, 2), [\,])$ n'est pas un contexte d'évaluation, car le membre gauche de la paire n'est pas une valeur. L'idée est de forcer un certain ordre d'évaluation en restreignant les contextes. La syntaxe des contextes ci-dessus force une évaluation en appel par valeur et de gauche à droite (on évalue la sous-expression gauche d'une application, d'une paire ou d'un `let` avant la sous-expression droite).

Exemple Considérons l'expression $a = (+(1,2), +(3,4))$. Il n'y a qu'une manière de l'écrire sous la forme $a = \Gamma_1(a_1)$ afin d'appliquer la règle (contexte): il faut prendre $\Gamma_1 = ([\,], +(3,4))$ et $a_1 = +(1,2)$. Comme $a_1 \xrightarrow{\varepsilon} 3$, on obtient $a \to \Gamma_1(3) = (3, +(3,4))$. Cette dernière expression peut alors s'écrire sous la forme $\Gamma_2(a_2)$ avec $\Gamma_2 = (3, [\,])$ et $a_2 = +(3,4)$. Appliquant la règle (contexte), on obtient le résultat $\Gamma_2(7) = (3,7)$. On a donc obtenu la séquence de réductions suivante:

$$(+(1,2), +(3,4)) \to (3, +(3,4)) \to (3,7)$$

qui nous emmène de a vers la valeur $(3,7)$ en évaluant toujours la sous-expression gauche en premier. Si nous voulons commencer par évaluer la sous-expression droite $+(3,4)$, il faudrait écrire $a = \Gamma_3(a_3)$ avec $\Gamma_3(+(1,2), [\,])$ et $a_3 = +(3,4)$, mais cela n'est pas possible car Γ_3 n'est pas un contexte d'évaluation.

Réduction multiple et formes normales On définit la relation $\xrightarrow{*}$ (lire: "se réduit en zéro, une ou plusieurs étapes") comme la fermeture réflexive et transitive de $\to$. C'est-à-dire, $a \xrightarrow{*} a'$ ssi $a = a'$ ou il existe $a_1, \ldots, a_n$ tels que $a \to a_1 \to \ldots \to a_n \to a'$.

On dit que a est en *forme normale* si $a \not\to$, c'est-à-dire s'il n'existe pas d'expression a' telle que $a \to a'$. Remarquons que les valeurs v sont en forme normale. Il y a aussi d'autres expressions qui sont en forme normale et qui ne sont pas des valeurs, comme p.ex. 1 2. Par la suite, nous caractériserons les expressions erronées comme les formes normales qui ne sont pas des valeurs.

Exercice 2.1 *(*) Modifier la syntaxe abstraite des contextes Γ pour forcer une évaluation de droite à gauche. Même question si l'on ne veut pas spécifier l'ordre d'évaluation des sous-expressions.*

Exercice de programmation 2.1 *Implémenter la sémantique par réduction de mini-ML.*

Écrire les fonctions auxiliaires suivantes: `est_une_valeur` *(teste si une expression est une valeur);* `réduction_tête` *(effectue une étape de réduction en tête ou signale une erreur si c'est impossible);* `décompose` *(étant donné une expression a qui n'est pas en forme normale, renvoie Γ et a_1 tels que $a = \Gamma(a_1)$ et a_1 peut se réduire en tête).*

En Caml, on pourra représenter les contextes Γ comme les fonctions Caml $a \mapsto \Gamma(a)$, c.à.d. comme les fonctions Caml qui prennent l'expression a et renvoient l'expression $\Gamma(a)$ en résultat.

Écrire une fonction qui prend une expression a et renvoie a' telle que $a \to a'$, ou bien signale que a est déjà en forme normale.

Écrire une fonction qui prend une expression a et renvoie sa forme normale si elle existe.

Tester votre code sur les exemples vus jusqu'ici.

Exercice 2.2 *(**) Montrer que la sémantique par réduction de mini-ML est équivalente à la sémantique opérationnelle structurelle du chapitre 1: $a \xrightarrow{v} v$ si et seulement si $a \xrightarrow{*} v$. (Indication: l'implication $\Rightarrow$ est une récurrence facile sur la dérivation de $a \xrightarrow{v} v$. Pour l'implication $\Leftarrow$, on montrera et on utilisera les deux lemmes suivants: (1) $v \xrightarrow{v} v$ pour toute valeur v; (2) si $a \to a'$ et $a' \xrightarrow{v} v$, alors $a \xrightarrow{v} v$.)*

2.3 Sûreté du typage

Dans une sémantique à réductions, il y a trois scénarios possibles pour l'évaluation d'un terme a:

1. a se réduit en un nombre fini d'étapes vers une valeur v:

$$a \to a_1 \to \ldots \to a_n \to v$$

   Cela correspond à un calcul qui termine et ne rencontre pas d'erreurs pendant l'évaluation.

2. a se réduit à l'infini

$$a \to a_1 \to \ldots \to a_n \to \ldots$$

   C'est un calcul qui ne termine pas, mais ne rencontre pas d'erreurs pendant l'évaluation.

3. a se réduit en une forme normale qui n'est pas une valeur

$$a \to a_1 \to \ldots \to a_n \not\to$$

   Dans ce cas, a_n est une expression absurde (du genre de 1 2). Ici, le calcul "plante" car il provoque une erreur d'exécution.

Nous allons prouver que si l'expression a est bien typée, le cas (3) ne peut pas se produire: ou bien a se réduit en une valeur (cas (1)), ou bien a ne termine pas (cas (2)), mais dans tous les cas aucune erreur d'exécution ne se produit. Ceci est un corollaire des deux propriétés suivantes:

- La réduction préserve le typage: si $\emptyset \vdash a : \tau$ et $a \to a'$, alors $\emptyset \vdash a' : \tau$
- Les formes normales bien typées sont des valeurs: si a est en forme normale vis-à-vis de $\to$ et si $\emptyset \vdash a : \tau$, alors a est une valeur.

Théorème 2.1 (Sûreté du typage) *Si $\emptyset \vdash a : \tau$ et $a \xrightarrow{*} a'$ et a' est une forme normale, alors a' est une valeur.*

Démonstration: Par la propriété de préservation du typage par réduction, nous avons $\emptyset \vdash a' : \tau$. Par conséquent, a' est une forme normale bien typée; c'est donc une valeur. □

Nous allons maintenant prouver les deux propriétés utilisées dans la preuve ci-dessus.

2.3.1 La relation "être moins typable"

Nous disons que a_1 est moins typable que a_2, et nous notons $a_1 \sqsubseteq a_2$, si

$$\text{pour tout } E \text{ et } \tau, (E \vdash a_1 : \tau) \text{ implique } (E \vdash a_2 : \tau)$$

Autrement dit, tous les types possibles pour a_1 doivent être des types possibles pour a_2.

Proposition 2.1 (Croissance de $\sqsubseteq$) *Pour tout contexte d'évaluation Γ, $a_1 \sqsubseteq a_2$ implique $\Gamma(a_1) \sqsubseteq \Gamma(a_2)$.*

Démonstration: Soient E et τ tels que $E \vdash \Gamma(a_1) : \tau$ (1). Nous montrons que $E \vdash \Gamma(a_2) : \tau$ par récurrence structurelle sur le contexte Γ.

Cas de base $\Gamma = [\,]$. Immédiat.

Cas $\Gamma = \Gamma'\ a$. Une dérivation de (1) se termine par

$$\frac{E \vdash \Gamma'(a_1) : \tau' \to \tau \qquad E \vdash a : \tau}{E \vdash \Gamma'(a_1)\ a : \tau}$$

Nous appliquons l'hypothèse de récurrence à la prémisse de gauche. Il vient $E \vdash \Gamma'(a_2) : \tau' \to \tau$, d'où la dérivation du résultat attendu:

$$\frac{E \vdash \Gamma'(a_2) : \tau' \to \tau \qquad E \vdash a : \tau}{E \vdash \Gamma'(a_2)\ a : \tau}$$

Cas $\Gamma = v\ \Gamma'$, $\Gamma = \texttt{let}\ x = \Gamma'\ \texttt{in}\ a$, $\Gamma = (\Gamma', a)$ ou $\Gamma = (v, \Gamma')$: même preuve que dans le cas précédent. □

2.3.2 Hypothèses sur les opérateurs

Pour que le typage soit sûr, il faut naturellement faire des hypothèses sur les opérateurs, leurs types, et leurs δ-règles. (Par exemple, le typage n'est certainement pas sûr si nous prenons un opérateur `cast` de type $\forall\alpha, \beta.\ \alpha \to \beta$ et qui se réduit par $\texttt{cast}\ v \xrightarrow{\varepsilon} v$.) Nous faisons donc les hypothèses suivantes:

H0 Pour tout opérateur op, $TC(op)$ est un type flèche de la forme $\forall\vec{\alpha}.\tau \to \tau'$. Pour toute constante c, $TC(c)$ est un type de base T.

H1 Si $a \xrightarrow{\varepsilon} a'$ par une δ-règle, alors $a \sqsubseteq a'$.

H2 Si $\emptyset \vdash op\ v : \tau$, alors il existe a' telle que $op\ v \xrightarrow{\varepsilon} a'$ par une δ-règle.

L'hyopthèse H0 dit que les opérateurs et les constantes ont des types "raisonnables". L'hypothèse H1 exprime que les δ-règles doivent préserver les typages. L'hypothèse H2 dit qu'il y a suffisamment de δ-règles pour réduire toutes les applications bien typées d'opérateurs à une valeur.

2.3.3 Typage et substitution

Pour montrer la préservation du typage par réduction, nous avons besoin d'un lemme technique sur le typage et les substitutions.

Proposition 2.2 (Lemme de substitution) *Supposons*

$$\begin{aligned} E &\vdash a' : \tau' \\ E + \{x : \forall\alpha_1 \ldots \alpha_n.\tau'\} &\vdash a : \tau \end{aligned}$$

avec $\alpha_1, \ldots, \alpha_n$ non libres dans E, et aucune des variables x liées dans a n'est libre dans a'. Alors,

$$E \vdash a[x \leftarrow a'] : \tau$$

Démonstration: par récurrence sur la structure de l'expression a. On écrit E_x pour $E + \{x : \forall\alpha_1 \ldots \alpha_n.\tau_1\}$.

Cas a est x. Nous avons alors $a[x \leftarrow a'] = a'$. Par hypothèse, $E_x \vdash x : \tau$, et donc $\tau \leq \forall\alpha' \ldots \alpha_n.\tau'$. C'est-à-dire que $\tau = \varphi(\tau')$ pour une certaine substitution φ sur les α_i. Appliquant la propriété de stabilité du typage par substitution du chapitre 1 (proposition 1.2) à l'énoncé $E \vdash a' : \tau'$ et à la substitution φ, il vient $\varphi(E) \vdash a' : \varphi(\tau')$, c'est-à-dire $E \vdash a' : \varphi(\tau')$ puisque les α_i ne sont pas libres dans E. Nous avons donc montré $E \vdash a' : \tau$; c'est le résultat attendu.

Cas x n'est pas libre dans a. Ceci recouvre les cas suivants: a est une variable $y \neq x$; a est une constante c ou un opérateur op; et $a = \texttt{fun}\ x \rightarrow a_1$. Alors, $a[x \leftarrow a'] = a$. De plus, $E_x \vdash a : \tau$ implique $E \vdash a : \tau$ par la proposition 1.3 (indifférence du typage vis-à-vis des hypothèses inutiles). CQFD.

Cas $a = \texttt{fun}\ y \rightarrow a_1$ avec $y \neq x$. Si les α_i apparaissent dans τ, nous pouvons les renommer en des variables β_i non libres dans E et distinctes des α_i par la substitution $\theta = [\alpha_i \leftarrow \beta_i]$. Si les α_i n'apparaissent pas dans τ, nous prenons θ égale à l'identité. Par la proposition 1.2, nous avons $E_x \vdash a : \theta(\tau)$ dans les deux cas.

Comme $E_x \vdash a : \theta(\tau)$, nous avons $\theta(\tau) = \tau_2 \rightarrow \tau_1$ et

$$\frac{E_x + \{y : \tau_2\} \vdash a_1 : \tau_1}{E_x \vdash \texttt{fun}\ y \rightarrow a_1 : \tau_2 \rightarrow \tau_1}$$

avec les α_i non libres dans $E_x + \{y : \tau_2\}$. Appliquant l'hypothèse de récurrence à la prémisse, il vient $E + \{y : \tau_2\} \vdash a_1[x \leftarrow a'] : \tau_1$, d'où $E \vdash \texttt{fun}\ y \rightarrow a_1[x \leftarrow a'] : \tau_2 \rightarrow \tau_1$, c'est-à-dire $E \vdash a : \theta(\tau)$. Nous concluons $E \vdash a : \tau$ par la proposition 1.2 appliquée au renommage inverse θ^{-1}.

Cas $a = a_1\ a_2$. Par hypothèse $E_x \vdash a : \tau$, nous avons

$$\frac{E_x \vdash a_1 : \tau_2 \rightarrow \tau \qquad E_x \vdash a_2 : \tau_2}{E_x \vdash a_1\ a_2 : \tau}$$

Appliquant l'hypothèse de récurrence aux deux prémisses, il vient la dérivation suivante:

$$\frac{E \vdash a_1[x \leftarrow a'] : \tau_2 \rightarrow \tau \qquad E \vdash a_2[x \leftarrow a'] : \tau_2}{E \vdash a_1[x \leftarrow a']\ a_2[x \leftarrow a'] : \tau}$$

D'où le résultat annoncé $E \vdash a[x \leftarrow a'] : \tau$.

Cas $a = \texttt{let}\ y = a_1\ \texttt{in}\ a_2$. Par hypothèse $E_x \vdash a : \tau$, nous avons

$$\frac{(1) E_x \vdash a_1 : \tau_1 \qquad (2) E_x + \{y : Gen(\tau_1, E_x)\} \vdash a_2 : \tau}{E_x \vdash \texttt{let}\ y = a_1\ \texttt{in}\ a_2 : \tau}$$

Appliquant l'hypothèse de récurrence à la prémisse (1), il vient $E \vdash a_1[x \leftarrow a'] : \tau_1$.

Si $y = x$, nous avons $a[x \leftarrow a'] = \texttt{let}\ x = a_1[x \leftarrow a']\ \texttt{in}\ a_2$, et $E_x + \{x : Gen(\tau_1, E_x)\} = E + \{x : Gen(\tau_1, E_x)\}$. Remarquons que $Gen(\tau_1, E) \geq Gen(\tau_1, E_x)$ puisque $\mathcal{L}(E) \subseteq \mathcal{L}(E_x)$. Par stabilité du

typage par renforcement des hypothèses (proposition 1.4), $E + \{x : Gen(\tau_1, E_x)\} \vdash a_2 : \tau$ implique $E + \{x : Gen(\tau_1, E)\} \vdash a_2 : \tau$. D'où la dérivation du résultat attendu:

$$\frac{E \vdash a_1[x \leftarrow a'] : \tau_1 \qquad E + \{x : Gen(\tau_1, E)\} \vdash a_2 : \tau}{E_x \vdash \texttt{let } x = a_1[x \leftarrow a'] \texttt{ in } a_2 : \tau}$$

Si $y \neq x$, nous avons $a[x \leftarrow a'] = \texttt{let } x = a_1[x \leftarrow a'] \texttt{ in } a_2[x \leftarrow a']$. et $E_x + \{y : Gen(\tau_1, E_x)\} = (E + \{y : Gen(\tau_1, E_x)\}) + \{x : \forall \alpha_i.\tau'\}$. Appliquant l'hypothèse de récurrence à la prémisse (2), il vient $E + \{y : Gen(\tau_1, E_x)\} \vdash a_2[x \leftarrow a'] : \tau$. Comme dans le sous-cas $x = y$, nous avons $Gen(\tau_1, E) \geq Gen(\tau_1, E_x)$, et donc par la proposition 1.4, il s'ensuit $E + \{y : Gen(\tau_1, E)\} \vdash a_2[x \leftarrow a'] : \tau$. Nous pouvons donc dériver le résultat attendu:

$$\frac{E \vdash a_1[x \leftarrow a'] : \tau_1 \qquad E + \{y : Gen(\tau_1, E)\} \vdash a_2[x \leftarrow a'] : \tau}{E_x \vdash \texttt{let } y = a_1[x \leftarrow a'] \texttt{ in } a_2[x \leftarrow a'] : \tau}$$

Cas $a = (a_1, a_2)$. Même preuve que pour l'application. □

2.3.4 Préservation du typage par réduction

Nous pouvons maintenant prouver que la réduction de tête $\xrightarrow{\varepsilon}$ préserve le typage.

Proposition 2.3 (Préservation du typage par réduction de tête) *Si $a \xrightarrow{\varepsilon} a'$, alors $a \sqsubseteq a'$.*

Démonstration: par cas sur la règle de réduction utilisée.

Cas d'une δ-règle. Le résultat est vrai par l'hypothèse H1.

Cas de la règle β_{fun}. Nous avons donc $a = (\texttt{fun } x \rightarrow a_1)\ v$ et $a' = a_1[x \leftarrow v]$. Soient E et τ tels que $E \vdash a : \tau$. La dérivation de cet énoncé est nécessairement de la forme

$$\frac{\dfrac{E + \{x : \tau'\} \vdash a_1 : \tau}{E \vdash (\texttt{fun } x \rightarrow a_1) : \tau' \rightarrow \tau} \qquad E \vdash v : \tau'}{E \vdash (\texttt{fun } x \rightarrow a_1)\ v : \tau}$$

Nous avons donc $E + \{x : \tau'\} \vdash a_1 : \tau$ et $E \vdash v : \tau'$. Les hypothèses du lemme de substitution (proposition 2.2) sont vérifiées (aucune variable généralisée α, et aucune variable libre dans la valeur v). Il s'ensuit $E \vdash a_1[x \leftarrow v] : \tau$, c'est-à-dire $E \vdash a' : \tau$. Ceci valant pour tout E et tout τ tel que $E \vdash a : \tau$, nous avons bien montré $a \sqsubseteq a'$.

Cas de la règle β_{let}. Ici, $a = (\texttt{let } x = v \texttt{ in } a_1)$ et $a' = a_1[x \leftarrow v]$. Soient E et τ tels que $E \vdash a : \tau$. La dérivation de cet énoncé est nécessairement de la forme

$$\frac{E \vdash v : \tau' \qquad E + \{x : Gen(\tau', E)\} \vdash a_1 : \tau}{E \vdash (\texttt{let } x = v \texttt{ in } a_1) : \tau}$$

Nous appliquons le lemme de substitution (proposition 2.2) aux deux prémisses. Il vient $E \vdash a_1[x \leftarrow v] : \tau$, c'est-à-dire $E \vdash a' : \tau$. D'où le résultat attendu $a \sqsubseteq a'$. □

Proposition 2.4 (Préservation du typage par réduction) *Si $a \to a'$, alors $a \sqsubseteq a'$.*

Ce résultat s'appelle également *subject reduction* dans la littérature.

Démonstration: Si c'est une réduction de tête, le résultat s'ensuit par la proposition 2.3. Sinon, c'est une application de la règle (contexte):

$$\frac{a_1 \xrightarrow{\varepsilon} a_1'}{\Gamma(a_1) \to \Gamma(a_1')}$$

Par la proposition 2.3, $a_1 \sqsubseteq a_1'$. Par croissance de $\sqsubseteq$ (proposition 2.1), $\Gamma(a_1) \sqsubseteq \Gamma(a_1')$. C'est le résultat attendu. □

2.3.5 Les formes normales bien typées sont des valeurs

Encore un lemme technique qui montre que le type d'une valeur conditionne sa "forme" (fonction, constante, paire, ...).

Proposition 2.5 (Forme des valeurs selon leur type) *Supposons $\emptyset \vdash v : \tau$.*

1. *Si $\tau = \tau_1 \to \tau_2$, alors ou bien v est de la forme* `fun` $x \to a$, *ou bien v est un opérateur op.*
2. *Si $\tau = \tau_1 \times \tau_2$, alors v est une paire (v_1, v_2).*
3. *Si τ est un type de base T, alors v est une constante c.*

Démonstration: par examen des règles de typage qui peuvent s'appliquer suivant la forme de τ. Seules les règles (const-inst), (op-inst), (fun) et (paire) sont à considérer, car les autres règles s'appliquent à des expressions qui ne sont pas des valeurs. Notons que par hypothèse H0, (const-inst) ne s'applique que si τ est un type de base, et (op-inst) que si τ est un type flèche. □

En conséquence, un terme bien typé ou bien est une valeur, ou bien peut se réduire.

Proposition 2.6 (Lemme de progression) *Si $\emptyset \vdash a : \tau$, ou bien a est une valeur, ou bien il existe a' telle que $a \to a'$.*

Démonstration: par récurrence sur la structure de a, et par cas sur la forme de a.

Cas $a = x$. Impossible, car a ne serait pas bien typée dans l'environnement vide.

Cas $a = c$ ou $a = op$ ou $a =$ `fun` $x \to a$. a est une valeur.

Cas $a = a_1\ a_2$. On a alors

$$\frac{\emptyset \vdash a_1 : \tau' \to \tau \qquad \emptyset \vdash a_2 : \tau'}{\emptyset \vdash a_1\ a_2 : \tau}$$

Si a_1 n'est pas une valeur, en appliquant l'hypothèse de récurrence, il vient que a_1 peut se réduire. Par la règle (contexte), $a_1\ a_2$ peut aussi se réduire.

Si a_1 est une valeur mais pas a_2, on applique l'hypothèse de récurrence à a_2. Il vient que a_2 peut se réduire, et donc aussi $a_1\ a_2$ par la règle (contexte).

Si a_1 et a_2 sont des valeurs, par la proposition 2.5, a_1 est ou bien une fonction $\texttt{fun}\ x \to a_3$ ou bien un opérateur op. Dans le premier cas, $a_1\ a_2$ se réduit par la règle β_{fun}. Dans le deuxième cas, l'hypothèse H2 dit que $a_1\ a_2$ peut aussi se réduire.

Cas $a = \texttt{let}\ x = a_1\ \texttt{in}\ a_2$. On a

$$\frac{\emptyset \vdash a_1 : \tau_1 \qquad \emptyset + \{x : Gen(\tau_1, \emptyset)\} \vdash a_2 : \tau_2}{\emptyset \vdash (\texttt{let}\ x = a_1\ \texttt{in}\ a_2) : \tau_2}$$

Si a_1 n'est pas une valeur, l'hypothèse de récurrence montre qu'elle peut se réduire, et donc a peut aussi se réduire par la règle (contexte). Si a_1 est une valeur, la règle β_{let} s'applique à a.

Cas $a = (a_1, a_2)$. On a

$$\frac{\emptyset \vdash a_1 : \tau_1 \qquad \emptyset \vdash a_2 : \tau_2}{\emptyset \vdash (a_1, a_2) : \tau_1 \times \tau_2}$$

Si a_1 n'est pas une valeur, par hypothèse de récurrence, elle peut se réduire, et donc a peut aussi se réduire par la règle (contexte). Même raisonnement si a_1 est une valeur mais pas a_2. Si a_1 et a_2 sont toutes deux des valeurs, a est aussi une valeur. CQFD. □

Proposition 2.7 (Les formes normales bien typées sont des valeurs) *Si $\emptyset \vdash a : \tau$ et a est en forme normale vis-à-vis de $\to$, alors a est une valeur*

Démonstration: conséquence immédiate de la proposition 2.6. □

2.3.6 Pour finir

Nous avons donc montré la sûreté du typage (théorème 2.1) sous les hypothèses H0, H1 et H2. Bien sûr, il ne faut pas oublier de vérifier que ces hypothèses sont vraies pour les opérateurs et les constantes qui nous intéressent.

Exercice 2.3 *(*) Vérifier H0, H1 et H2 pour les constantes entières et les opérateurs* `+`, `fst`, `snd`, *et* `fix`.