CTD2 : Sémantique opérationnelle du λ-calcul et de Core ML

Nous nous interessons brièvement à la sémantique opérationnelle générale du λ-calcul avant de nous pencher plus spécifiquement sur celle de Core ML.

1 Sémantique opérationnelle du λ-calcul

La *sémantique opérationnelle* est la description formelle de la façon dont les λ-termes sont évalués. Cette description est la donnée d'une relation binaire sur les λ-termes dite *relation de réduction* ou encore β*-réduction* et notée $\rightarrow$. Ainsi on écrira des séquences de réduction

$$t \rightarrow t_1 \rightarrow \ldots \rightarrow t_n \rightarrow \ldots$$

pour représenter les étapes (i.e. applications de fonctions) successives de l'évaluation (ou exécution) du terme t. Notons le fait important qu'une telle évaluation peut être infinie ! Quand l'évaluation *termine* alors le terme v finalement obtenu est appelé *valeur* (i.e. un terme pour lequel $\nexists u.v \rightarrow u$). Comme exemples de valeurs nous pouvons déjà citer les constructeurs d'arité nulle (i.e. les constantes `0`, `1`, `true`, etc.).

On note $\rightarrow^*$ la cloture transitive et réflexive de $\rightarrow$.

1.1 β-réductions de base

Il existe trois règles de base permettant de réduire des λ-terme correspondants à
- l'application d'une fonction primitive à *tous* ses arguments ;
- l'application d'une fonction à un argument ;
- la liaison **`let`** d'une variable à un terme.

On fera l'hypothèse, pour le moment informelle, qu'on connaît l'évaluation des fonctions primitives lorsqu'elles sont (totalement) appliquées à des valeurs. La première règle de base est donc :

$$\texttt{f}\ v_1 \ldots v_n \rightarrow v \qquad \text{si } \text{“}\texttt{f}(v_1, \ldots, v_n) = v\text{”} \tag{δ}$$

Par exemple,

$$\texttt{+ 1 2} \rightarrow \texttt{3}$$

alors que les termes `+ 1` (application partielle) et `+ true false` (indéfini) ne se réduisent pas.

La règle fondamentale du λ-calcul est celle qui « applique » une fonction (plus exactement, une λ-abstraction) à un argument en substituant celui-ci aux occurrences de la variable abstraite dans le corps de la fonction. Formellement,

$$(\lambda x.t)u \rightarrow [u/_x]t \qquad \text{si } bv(t) \cap fv(u) = \emptyset \tag{β}$$

Ainsi, par exemple

$$(\lambda\ f.(\lambda g.g\ \texttt{2})f)(\lambda y.\texttt{+ 1}\ y) \rightarrow (\lambda g.g\ \texttt{2})(\lambda y.\texttt{+ 1}\ y) \rightarrow (\lambda y.\texttt{+ 1}\ y)\texttt{2} \rightarrow \texttt{+ 1 2} \rightarrow \texttt{3}$$

Notons que la condition $bv(t) \cap fv(u) = \emptyset$ est fondamentale pour éviter d'éventuelles **captures de variables**. En effet, sans cette condition la réduction suivante serait autorisée

$$(\lambda xy.x)y \rightarrow \lambda y.y$$

($bv(\lambda y.x) = \{y\}$ et $fv(y) = \{y\}$) et la variable y intialement libre devient liée après réduction pour obtenir la fonction identité ! Alors qu'on peut trouver un terme α-équivalent qui ne se réduit pas en l'identité :

$$(\lambda xz.x)y \rightarrow \lambda z.y$$

On dit que la condition $bv(t) \cap fv(u) = \emptyset$ impose une **liaison statique** (ou **lexicale**) des variables. Par opposition, l'abscence de cette condition impliquerait une **liaison dynamique** des variables (c'est le cas par exemple dans GNU Emacs Lisp et les premières versions de Lisp !).

La règle permettant de lier un terme u à une variable x dans un terme t substitue u aux occurences de x dans t (pensez à une macro-expansion). Formellement,

$$\texttt{let}\ \ x = u\ \ \texttt{in}\ \ t \to [u/x]t \qquad (\textit{Let})$$

Ainsi, par exemple

$$\texttt{let}\ \ id = \lambda x.x\ \ \texttt{in}\ \ (id\ \ id\ \ id\ \ \texttt{0}) \to (\lambda x.x)\ (\lambda x.x)\ (\lambda x.x)\ \texttt{0}$$

Notez que, du point de vue de la β-réduction, la construction $\texttt{let}\ \ x = u\ \ \texttt{in}\ \ t$ est équivalente à $(\lambda x.t)u$. Nous verrons leur différence lors de l'étude du typage.

1.2 Redex et contextes d'évaluation

Telle que définie jusqu'à présent, la β-réduction ne permet pas de réduire des termes tels que $\texttt{+}\ (\texttt{+}\ \texttt{1}\ \texttt{1})$ ou $\lambda x.\texttt{+}\ \texttt{1}\ \texttt{1}$. On peut souhaiter quand même pouvoir évaluer des λ-termes dont les sous-termes propres peuvent être réduits, c'est-à-dire

$$\texttt{+}\ (\underline{\texttt{+}\ \texttt{1}\ \texttt{1}}) \to \texttt{+}\ \texttt{2} \qquad \text{et} \qquad \lambda x.\underline{\texttt{+}\ \texttt{1}\ \texttt{1}} \to \lambda x.\texttt{2}$$

Les sous-termes soulignés, candidats de réduction, sont appelés **redex**. Plus précisémment, tout sous-terme de la forme $(\lambda x.t)u$, $\texttt{let}\ \ x = u\ \ \texttt{in}\ \ t$ ou $\texttt{f}\ v_1 \ldots v_n$ (tel que *arité*$(\texttt{f}) = n$) est un rédex. Par exemple, le λ-terme

$$*\ (\underline{\texttt{+}\ \texttt{1}\ \texttt{1}})\ (\underline{\texttt{+}\ \texttt{2}\ \texttt{2}}) \qquad (\dagger)$$

comprend deux redex correspondant aux sous-termes soulignés. Pour permettre de telles réductions on défini la notion de **contexte d'évaluation**. Un contexte d'évaluation E est un λ-terme à trou. Formellement, on les défini ainsi :

$$E\ ::=\ [\bullet]\ |\ E\ t\ |\ t\ E\ |\ \lambda x.E\ |\ \texttt{let}\ \ x = E\ \ \texttt{in}\ \ t\ |\ \texttt{let}\ \ x = u\ \ \texttt{in}\ \ E$$

On note $E[t]$ le λ-terme obtenu en substituant t au trou dans E. Par exemple, avec les contextes

$$E_1 = *\ [\bullet]\ (\texttt{+}\ \texttt{2}\ \texttt{2}) \quad \text{et} \quad E_2 = *\ (\texttt{+}\ \texttt{1}\ \texttt{1})\ [\bullet]$$

les termes $E_1[\texttt{+}\ \texttt{1}\ \texttt{1}]$ et $E_2[\texttt{+}\ \texttt{2}\ \texttt{2}]$ correspondent tous deux au terme $(\dagger)$.

Grâce aux contextes on peut étendre la β-réduction comme souhaitée en ajoutant à sa définition la règle suivante :

$$\text{si } t \to u \text{ alors } E[t] \to E[u] \qquad (\textit{Cont})$$

Avec le contexte $E_1' = *\ \texttt{2}\ [\bullet]$, on peut donc réduire le λ-terme $(\dagger)$ ainsi :

$$\begin{array}{rcll} *\ (\underline{\texttt{+}\ \texttt{1}\ \texttt{1}})\ (\texttt{+}\ \texttt{2}\ \texttt{2}) & \to & *\ \texttt{2}\ (\underline{\texttt{+}\ \texttt{2}\ \texttt{2}}) & \text{car } \texttt{+}\ \texttt{1}\ \texttt{1} \underset{(\delta)}{\longrightarrow} \texttt{2} \text{ et par } (\textit{Cont}),\ E_1[\texttt{+}\ \texttt{1}\ \texttt{1}] \to E_1[\texttt{2}] \\ & \to & \underline{*\ \texttt{2}\ \texttt{4}} & \text{car } \texttt{+}\ \texttt{2}\ \texttt{2} \underset{(\delta)}{\longrightarrow} \texttt{4} \text{ et par } (\textit{Cont}),\ E_1'[\texttt{+}\ \texttt{2}\ \texttt{2}] \to E_1'[\texttt{4}] \\ & \underset{(\delta)}{\longrightarrow} & \texttt{8} & \end{array}$$

Enfin, si une réduction ne peut se faire à cause d'une capture de variable, on autorise le renommage de la variable liée par α-équivalence :

$$\text{si } t =_\alpha t' \text{ et } t' \to u \text{ alors } t \to u \qquad (\alpha)$$

Ainsi :

$$(\lambda xy.xy)y \to \lambda z.yz \quad \text{car } \lambda xy.xy =_\alpha \lambda xz.xz \text{ et } (\lambda xz.xz)y \to \lambda z.yz$$

1.3 Confluence et divergence

Remarquons que la β-réduction ne nous impose pas d'ordre dans l'évaluation des redexes. Ainsi, la réduction du terme $(\dagger)$ peut aussi bien commencer par la réduction du deuxième redex. En effet, avec $E_2' = *\ [\bullet]\ \texttt{4}$, on a :

$$\begin{array}{rcll} *\ (\texttt{+}\ \texttt{1}\ \texttt{1})\ (\underline{\texttt{+}\ \texttt{2}\ \texttt{2}}) & \to & *\ (\underline{\texttt{+}\ \texttt{1}\ \texttt{1}})\ \texttt{4} & \text{car } \texttt{+}\ \texttt{2}\ \texttt{2} \underset{\delta}{\to} \texttt{4} \text{ et par } (\textit{Cont}),\ E_2[\texttt{+}\ \texttt{2}\ \texttt{2}] \to E_2[\texttt{4}] \\ & \to & \underline{*\ \texttt{2}\ \texttt{4}} & \text{car } \texttt{+}\ \texttt{1}\ \texttt{1} \underset{\delta}{\to} \texttt{2} \text{ et par } (\textit{Cont}),\ E_2'[\texttt{+}\ \texttt{1}\ \texttt{1}] \to E_2'[\texttt{2}] \\ & \underset{\delta}{\to} & \texttt{8} & \end{array}$$

On remarque que quelque soit l'ordre d'évaluation, toute réduction du terme $*\ (\texttt{+}\ \texttt{1}\ \texttt{1})\ (\texttt{+}\ \texttt{2}\ \texttt{2})$ est finie, c'est-dire que son évaluation *termine*. Un terme qui ne se réduit pas est dit en **forme normale** (aussi appelé *valeur* plus haut). Un terme t est dit **fortement normalisant** ssi toutes les réductions de t sont finies (et donc aboutissent à une valeur, la forme normale de t).

Proposition 1 *La β-réduction ne termine pas en général.* □

Preuve. Soit le λ-terme $\Omega = (\lambda x.xx)(\lambda x.xx)$, alors la seule réduction possible est la réduction infinie

$$\Omega \to \Omega \to \ldots \to \Omega \to \ldots$$

□

Un λ-terme t est dit **faiblement normalisant** ssi au moins une réduction de t est est finie.

Question 1 Montrer que le terme $(\lambda x.y)\Omega$ est faiblement mais pas fortement nomalisant. □

La question qui se pose ici est : si t est faiblement normalisant, y a-t-il unicité de sa forme normale ? En d'autres termes, est-ce que les évaluations de t qui terminent aboutissent toujours à la même valeur ? La réponse est oui !

Théorème 1 (Confluence) *Si $t \to^* t_1$ et $t \to^* t_2$ alors il existe u tel que $t_1 \to^* u$ et $t_2 \to^* u$.* □

Preuve. Difficile ! □

La propriété de confluence est aussi appelée (en fait, équivalente à la) propriété de *Church-Rosser*.

Corollaire 1 (Unicité des formes normales) *Si u_1 et u_2 sont deux formes normales de t alors $u_1 = u_2$.* □

Question 2 Réduire, autant que possible, les termes suivants :
- $(\lambda x.* \ \mathtt{2}\ x)\ \mathtt{3}$;
- $(\lambda x.((\lambda y.+\ x\ y)\ \mathtt{2}))\ \mathtt{1}$;
- $(\lambda x.x\ \mathtt{1})(\lambda x.x)$;
- $(\lambda x.x\ \mathtt{3})(\lambda x.*\ x\ \mathtt{2})$;
- $(\lambda f.((\lambda g.g\ \mathtt{2})f))\ (\lambda y.+\ \mathtt{1}\ y)$;
- $(\lambda xy.x\ y)(\lambda x.y\ x)$.

2 Sémantique opérationnelle de Core ML

2.1 Appel par nom et appel par valeur

Nous avons vu qu'en général (c'est-à-dire dans le λ-calcul) la réduction des termes est *non-déterministe* ce qui signifie qu'aucun ordre d'évaluation des redex n'est imposé par la sémantique opérationnelle. Lorsqu'on implante un langage de programmation fonctionnel (tel que Caml) il faut donc choisir un ordre d'évaluation (car les implantations sont déterministes !). On parle aussi de *stratégie d'évaluation*. On peut par exemple choisir de toujours réduire d'abord le rédex le *plus externe* et *le plus à gauche*. Par exemple, on aura l'unique réduction

$$\begin{array}{rcl} \underline{(\lambda x.+\ x\ x)\ (+\ \mathtt{1}\ \mathtt{1})} & \to & +\ (\underline{+\ \mathtt{1}\ \mathtt{1}})\ (+\ \mathtt{1}\ \mathtt{1}) \\ & \to & +\ \mathtt{2}\ (\underline{+\ \mathtt{1}\ \mathtt{1}}) \\ & \to & \underline{+\ \mathtt{2}\ \mathtt{2}} \\ & \to & \mathtt{4} \end{array}$$

Une telle stratégie est appelée **appel par nom**. On peut au contraire toujours choisir d'abord le redex le plus interne et le plus à gauche et ne pas réduire sous les λ-abstractions. Par exemple, on aura l'unique réduction

$$\begin{array}{rcl} (\lambda x.+\ x\ x)\ (\underline{+\ \mathtt{1}\ \mathtt{1}}) & \to & \underline{(\lambda x.+\ x\ x)\ \mathtt{2}} \\ & \to & \underline{+\ \mathtt{2}\ \mathtt{2}} \\ & \to & \mathtt{4} \end{array}$$

Cette stratégie, que nous allons maintenant détailler plus formellement, est appelée **appel par valeur** et c'est celle qui est implantée pour évaluer les expressions de Core ML.

2.2 Valeurs et sémantique des fonctions primitives

L'évaluation d'une expression arithmétique mène évidemment à une *valeur* qui est une constante entière ou flotante ou autre. On a vu d'autre part qu'il est possible d'évaluer partiellement des applications de fonction, le résultat étant alors une fonction. Enfin, tout constructeur de donnée appliqué à des valeurs est lui-même une valeur. Par exemple, la paire `pair 1 2` est une valeur. Plus formellement, les valeurs, notées v, sont définies ainsi :

v	$::=$	$\lambda x.t$	λ-abstraction
	$\mid$	$\mathtt{c}\ v_1 \ldots v_k$	donnée structurée, éventuellement partielle ($k \leq$ *arité*($\mathtt{c}$))
	$\mid$	$\mathtt{f}\ v_1 \ldots v_k$	fonction primitive partiellement appliquée ($k <$ *arité*($\mathtt{f}$))

Précisons toute suite que cette définition de valeur est différente de celle donnée précédemment pour le λ-calcul : tout terme sous forme normale n'est pas nécessairement une valeur. Par exemple, le terme xy est sous forme normale mais n'est pas une valeur.

On précise maintenant plus formellement l'évaluation des fonctions primitives qui requiert un traitement particulier. On suppose en effet leur sémantique connue et, pour chaque fonction primitive `f` d'arité n on représente cette sémantique par la relation

$$\delta_{\mathtt{f}} = \{(\mathtt{f}v_1 \dots v_n, v) \mid v_1, \dots, v_n \text{ sont des valeurs}\}$$

Ainsi, par exemple $\delta_{+} = \{(\mathtt{+\ 0\ 0}, \mathtt{0}), (\mathtt{+\ 0\ 1}, \mathtt{1}), \dots, (\mathtt{+\ 1\ 0}, \mathtt{1}), (\mathtt{+\ 1\ 1}, \mathtt{2}), (\mathtt{+\ 1\ 2}, \mathtt{3}), \dots\}$. On défini la relation δ par $\delta = \bigcup_{\mathtt{f}\in\mathcal{F}}(\delta_{\mathtt{f}})$.

Question 3 Déterminez pour chacun des termes suivants s'il s'agit d'une valeur :
- $\mathtt{+\ 1\ 1}$
- $\lambda x.(\mathtt{+}\ x\ \mathtt{1})$
- $\lambda x.(\mathtt{+\ 1\ 1})$
- x
- $\mathtt{let}\ x = y\ \mathtt{in}\ \lambda x.y$
- $\mathtt{+\ 1}$

2.3 β-réductions de base dans Core ML

Les règles de base de Core ML sont des variantes des trois règles de base du λ-calcul. Il s'agit de :
- l'application d'une fonction primitive à *tous* ses arguments ;
- l'application d'une fonction à une *valeur* ;
- la liaison **let** d'une variable à une *valeur*.

Il est ici important de comprendre qu'une fonction (éventuellement primitive) ne peut être appliquée à ses arguments que si ces derniers sont des valeurs.

L'évaluation d'une fonction primitive (totalement) appliquée à des valeurs est maintenant formellement définie grâce à la relation δ :

$$\mathtt{f}\ v_1 \dots v_n \to v \qquad \text{si } (\mathtt{f}\ v_1 \dots v_n, v) \in \delta \tag{δ}$$

L'application d'une λ-abstraction et la liaison `let` sont maintenant restreintes aux valeurs :

$$(\lambda x.t)v \to [v/x]t \tag{β_v}$$
$$\mathtt{let}\ x = v\ \mathtt{in}\ t \to [v/x]t \tag{Let_v}$$

L'utilisation de ces trois seules règles permet des réductions suivantes :

$$(\lambda\ f.(\lambda g.gx)f)(\lambda y.y) \to (\lambda g.gx)(\lambda y.y) \to (\lambda y.y)x \not\to x$$

$$\mathtt{let}\ id = \lambda x.x\ \mathtt{in}\ (id\ id\ id\ \mathtt{0}) \to (\lambda x.x)\ (\lambda x.x)\ (\lambda x.x)\ \mathtt{0}$$

$$\mathtt{let}\ inc = \lambda x.\mathtt{+}\ x\ \mathtt{1}\ \mathtt{in}\ inc\ \mathtt{0} \xrightarrow[(Let_v)]{} (\mathtt{+}\ x\ \mathtt{1})\ \mathtt{0} \xrightarrow[(\beta_v)]{} (\mathtt{+\ 0\ 1}) \xrightarrow[(\delta)]{} \mathtt{1}$$

Question 4 Exemples de réductions.

2.4 Contextes d'évaluation et appel par valeur

On a vu qu'en appel par valeur, seules les fonctions appliquées à des valeurs peuvent être réduites. Les sous-termes fonctions dans les applications (i.e. t dans $t\ u$) ainsi que les termes abstraits dans les liaisons **let** (i.e. t dans $\mathtt{let}\ x = t\ \mathtt{in}\ u$) sont réduits en priorité. Par ailleurs, on ne réduit pas sous les λ-abstractions (par définition des valeurs). Par conséquent, en appel par valeur, les contextes d'évaluation V sont restreints ainsi :

$$V\ ::=\ [\bullet] \mid V\ t \mid v\ V \mid \mathtt{let}\ x = V\ \mathtt{in}\ t$$

Dans Core ML, tous les redex d'un terme ne sont pas nécessairement candidats de réduction. Par exemple, le λ-terme

$$(\lambda xy.*\ x\ y)\ \underline{(\mathtt{+\ 1\ 1})}\ \underline{(\mathtt{+\ 2\ 2})} \tag{$\dagger$}$$

comprend deux redex correspondant aux sous-termes soulignés. Mais seul et premier redex est candidat de réduction.

On peut finalement définir la relation de réduction $\longrightarrow$ comme la réduction satisfaisant (β_v), (Let_v), (δ) et qui passe au contextes, c'est-à-dire telle que

$$\text{si } t \longrightarrow u \text{ alors } V[t] \longrightarrow V[u] \qquad (Cont_v)$$

et qui est définie modulo α-équivalence :

$$\text{si } t =_\alpha t' \text{ et } t' \longrightarrow u \text{ alors } t \longrightarrow u \qquad (\alpha)$$

On peut ainsi montrer que

$$\begin{array}{rcl}
(\lambda xy.* \ x \ y) \ \underline{(+ \ 1 \ 1)} \ (+ \ 2 \ 2) & \xrightarrow[(\delta)]{} & \underline{(\lambda xy.* \ x \ y) \ 1} \ (+ \ 2 \ 2) \\
& \xrightarrow[(\beta_v)]{} & (\lambda y.* \ 1 \ y) \ \underline{(+ \ 2 \ 2)} \\
& \xrightarrow[(\delta)]{} & \underline{(\lambda y.* \ 1 \ y) \ 4} \\
& \xrightarrow[(\beta_v)]{} & \underline{* \ 1 \ 4} \\
& \xrightarrow[(\delta)]{} & 4
\end{array}$$

Question 5 Donnez les contextes d'évaluation permettant la réduction précédente.

La réduction ainsi définie est bien une stratégie déterministe dans le sens suivant :

Proposition 2 *Pour tout λ-terme t, si $t \longrightarrow u_1$ et $t \longrightarrow u_2$ alors $u_1 =_\alpha u_2$.*

Question 6 Réduire, autant que possible, les termes suivant en appel par valeur. Précissez systèmatiquement les contextes d'évaluation.

- $(\lambda x.x) \ (\lambda x.x \ 1) \ 1$;
- **let** $id = \lambda x.x$ **in** $id \ id \ id \ 0$;
- $(\lambda xy.+ \ x \ y) \ ((\lambda x.+ \ x \ x) \ 1) \ 1$.

Question 7 Réduire les deux termes ci-dessous par la stratégie par valeur et par la stratégie par nom.

$$T_1 = (\lambda xyz.(x \ y \ z)) \ (\lambda xy.x)) \ (+ \ 1 \ 2) \ (* \ 2 \ 3) T_2 \qquad = (\lambda xyz.(x \ y \ z)) \ (\lambda xy.y)) \ (+ \ 1 \ 2) \ (* \ 2 \ 3)$$

Quelle stratégie offre la réduction la plus rapide ?

Comment peut–on interpréter les termes $\lambda xyz.(x \ y \ z)$, $\lambda xy.x$ et $\lambda xy.y$ dans un langage de programmation ?