Chapitre 1

Présentation du langage ML

Note

Les notes de cours des chapitres 1 et 2 seront distribuées indépendemment. Les notes données ci-dessous permettent de faire le lien entre les chapitres 1 et 2 et les autres chapitres de ce cours. Les notations sont aussi légèrement différentes.

Dans cette partie nous présentons le noyau ML. Après avoir introduit la syntaxe du langage, nous décrivons sa sémantique opérationnelle à grands pas, aussi connue sous le nom de sémantique naturelle. Enfin, nous introduirons son système de types.

1.1 Rappel du noyau ML

Les expressions du langage sont essentiellement les termes du λ-calcul avec constantes : la construction de liaison $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$ sera utilisée pour marquer certains radical $(\texttt{fun}\ (x)\ a_2)\ a_1$ afin de mieux les typer :

$a ::= z$		
	$\mid \texttt{fun}\ (x)\ a$	Abstractions
	$\mid a_1\ a_2$	Applications
	$\mid \texttt{let}\ x = a_1\ \texttt{in}\ a_2$	Déclarations
$z ::= x$		Variables
	$\mid c$	Constantes
$c ::= C$		Constructeurs
	$\mid f$	Primitives

Parmi, les constantes nous distinguons les *constructeurs* et les *primitives*. Chaque constante est donnée avec une arité. Nous donnerons plus loin une distinction précise entre constructeurs et primitives. Pour l'instant, nous pouvons nous contenter de l'idée simple suivante : les primitive complètement appliqué (i.e. autant de fois que leur arité) ne sont pas des valeurs ; les primitives partiellement appliquées ou les constructeurs, complètement ou partiellement appliqués, sont des valeurs. Par exemple, les entiers,

les chaînes de caractères, le symbole de tuple sont des constructeurs ; l'addition, le successeur, la concaténation sont des primitives. Nous désignons par la lettre z une variable ou une constante, indifféremment. Nous notons également c^k, f^k ou C^k pour préciser que la constante, le constructeur ou la primitive est d'arité k. Certaines constantes sont parfois notées de façon particulières. Par exemple l'addition est notée $(\_ + \_)$; cela est simplement une convenance de notation qui permet d'écrire $(a_1 + a_2)$ au lieu de $(\_ + \_)\ a_1\ a_2$, mais nous considérerons toujours les deux notations comme équivalentes.

Les expressions du langage égales par α-conversion (renommage des variables liées) sont identifiées. La substitution de x par a_1 dans a_2 est notée $a_2[a_1/x]$. Nous supposons que les variables liées dans a_2 ont été renommées en des variables autres que les variables libres dans a_1, afin d'éviter leur capture. Par exemple $(\texttt{fun}\ (x)\ \ y)[a/x]$ est égal à $(\texttt{fun}\ (z)\ z\ y)[a/x]$ par α-conversion, puis $\texttt{fun}\ (z)\ ((z\ y)[a/x])$, i.e. $\texttt{fun}\ (z)\ (z\ y)$ ou en utilisant à nouveau l'α-conversion. Une approche équivalente est de dire que $\theta(\texttt{fun}\ (x)\ a)$ est égal à $\texttt{fun}\ (x)\ (\theta \setminus \{x\})(a)$ si θ est une substitution et $\theta \setminus \{x\}$ sa restriction en dehors de x. Par contre, $\theta(a_1\ a_2)$ est égal à $\theta(a_1)\ \theta(a_2)$.

1.2 Sémantiques opérationnelles

Nous n'avons jusqu'à présent définit que la *syntaxe abstraite* du langage. Celle-ci décrit simplement les constructions du langage, et la façon de les désigner formellement. Pour être complet, nous devrions également définir la *syntaxe concrète*, c'est-à-dire les suites de caractères qui sont reconnus comme des programmes ainsi que la représentation abstraite du programme associé. Par exemple, il n'est pas décrit ci-dessus la façon de parenthéser les expressions, la priorité des opérateurs, etc.

Donner la syntaxe, concrète ou abstraite, d'un langage n'est qu'un préliminaire à sa définition. Il faut ensuite donner un sens aux expressions, c'est-à-dire décrire leur sémantique. La façon la plus formelle est de construire un modèle mathématique indépendamment de la syntaxe, puis d'associer à chaque expression un objet du modèle. Le modèle devra bien sûr rendre compte fidèlement du langage, c'est-à-dire identifier au maximum les programmes qui se comportent de la même façon et ceux-ci seulement. Ces modèles sont souvent des constructions mathématiques difficiles.

Dans cette partie nous allons nous contenter de décrire la *sémantique opérationnelle* du langage, c'est-à-dire la façon dont se déroule le calcul (on parle ainsi quelques fois de sémantique dynamique, mais on évitera ce terme ambigu).

Il existe deux approches assez différentes pour présenter la sémantique opérationnelle d'un langage. La *sémantique opérationnelle structurelle* s'intéresse seulement aux résultats de l'évaluation ; elle met en relation des programmes (expressions closes et bien formées du langage) avec un ou plusieurs résultats possibles. La *sémantique à réduction* s'intéresse au calcul proprement dit en décrivant chaque étape élémentaire ; elle consiste à définir une relation de réécriture des programmes sur eux-mêmes. Ainsi, un programme se réduit pas à pas, soit indéfiniment, soit jusqu'à l'obtention d'une forme normale. On parle de sémantique *à grands pas* pour la sémantique opérationnelle struc-

turelle et de sémantique *à petits pas* pour la sémantique à réduction.

Dans ce chapitre nous présenterons uniquement la sémantique à grands pas, plus concrète, et nous l'utiliserons pour définir un évaluateur du langage. La sémantique à petits pas, plus formelle et plus précise, sera décrite dans le chapitre 2, et utilisée pour montrer que le typage garantit le bon déroulement des calculs.

1.2.1 Sémantique à grands pas

Cette approche, autrefois appelée sémantique naturelle, est encore appelée *sémantique opérationnelle structurelle* (S.O.S.).

Elle consiste à définir une relation $a \Rightarrow r$ entre les programmes et les résultats. Intuitivement, les résultats sont les valeurs résultant de l'exécution d'un programme. Pour rendre compte d'une exécution anormale, il faut aussi ajouter un résultat `error` (qui n'est pas une valeur). Les résultats ne sont donc pas un sous-ensemble des programmes. Les valeurs quant à elles peuvent être un sous-ensemble des expressions du langage, mais cela n'est pas une obligation, et il est fréquent de les distinguer des programmes, ce qui laisse plus de flexibilité et permet facilement de modéliser un calcul efficace.

En général, la relation d'évaluation est définie par un ensemble de règles d'inférence qui décrivent l'évaluation d'un programme à partir de sa structure et en s'appuyant sur l'évaluation de sous-programmes. Pour cela on utilise une relation auxiliaire $e \vdash a \Rightarrow r$ qui signifie que dans l'environnement d'évaluation e, le programme a s'évalue en le résultat r. Un environnement d'évaluation lie des variables à des valeurs. L'évaluation du programme a au sens restreint est alors son évaluation dans l'environnement vide aussi notée $a \Rightarrow r$. Par exemple, l'ensemble des valeurs peut être défini par :

$$\begin{aligned} v &::= \texttt{closure}\ (e, x, a) \\ &\quad \mid C^k v_1\ \ldots v_k \\ r &::= v \mid \texttt{error} \end{aligned}$$

Nous illustrons quelques règles d'inférence définissant la sémantique opérationnelle structurelle de ML.

(App-val)
$$\frac{e \vdash a_1 \Rightarrow \texttt{closure}\ (e', x, a) \qquad e \vdash a_2 \Rightarrow v_2 \qquad e \oplus e_1 \oplus (x : v_2) \vdash a \Rightarrow v}{e \vdash a_1\ a_2 \Rightarrow v}$$

(App-Err-Gen)
$$\frac{a \vdash a_1 \Rightarrow v \qquad r \neq \texttt{closure}\ (e', x, a)}{a \vdash a_1\ a_2 \Rightarrow \texttt{error}}$$

(App-Err-Prop-1)
$$\frac{a \vdash a_1 \Rightarrow \texttt{error}}{a \vdash a_1\ a_2 \Rightarrow \texttt{error}}$$

(App-Err-Prop-2)
$$\frac{a \vdash a_1 \Rightarrow v \qquad a \vdash a_2 \Rightarrow \texttt{error}}{a \vdash a_1\ a_2 \Rightarrow \texttt{error}}$$

Nous voyons ici qu'il existe deux catégories de règles. Les plus intéressantes décrivent le calcul dans le cas d'une évaluation normale. Les autres génèrent ou propagent les

erreurs. Les règles de propagation sont les plus nombreuses et, à tord, elles sont souvent omises et laissées implicites. Bien que d'apparence systématique, elles sont au contraire essentielles car elle déterminent la stratégie d'évaluation du langage. Par exemple, la règle App-Err-Prop-2 ci-dessus impose que la composante fonctionnelle soit évaluée en premier, car si le programme a_1 ne retourne pas de résultat, alors même si l'évaluation de a_2 produirait une erreur, l'application $a_1\ a_2$ ne produira pas de résultat.

Nous verrons plus tard que la correction du typage doit garantir qu'un programme bien typé ne peut pas s'évaluer en un résultat erroné, c'est-à-dire que pour un programme bien typé, si $a \Rightarrow r$ alors $r \neq \texttt{error}$.

Évaluateur

Évaluations infinies

Un problème de la sémantique à grands pas est de ne pas pouvoir décrire les programmes qui ne terminent pas. Parmi ces programmes, il y a en particulier tous les programmes qui bouclent. Mais ils pourraient également y avoir des programmes bloqués. La non terminaison n'est pas définissable à partir de la sémantique à grands pas, puisqu'elle se limite dès le départ aux programmes qui rendent un résultat.

Pour caractériser la non terminaison, il faut se placer en dehors du formalisme. Si, comme ci-dessus, la relation d'évaluation est définie à partir de règles d'inférence, il est alors possible de distinguer deux types de programmes qui ne se réduisent pas. Ceux pour lesquels aucune règle ne possède une conclusion qui filtre le programme à évaluer ; la recherche d'une preuve d'évaluation est donc sans espoir, ce qui correspond à une situation où l'évaluation est bloquée. Ceux, au contraire, pour lesquels il existe effectivement une règle dont la conclusion filtre le programme à évaluer, mais dont toujours une des prémisses correspond à une évaluation qui n'est pas définie mais non bloquée. La recherche d'une preuve d'évaluation est une régression infinie : le programme boucle. Il est souhaitable que cette seconde situation soit la seule possible, afin de pouvoir justement définir la non terminaison ou *divergence* de l'évaluation comme l'ensemble des programmes pour lesquels l'évaluation n'est pas définie. Cela est en effet toujours possible en ajoutant des règles qui permettent d'évaluer en `error` les programmes bloqués. Malheureusement, cette condition indispensable à la définition de la non-terminaison ne peut être énoncée qu'en dehors du formalisme.

Lorsqu'il est ainsi possible de définir une notion de divergence, nous noterons $e \vdash a \uparrow$ pour dire que le programme a diverge dans l'environnement d'évaluation e. Mais cela ne résout pas le problème inhérent de la sémantique à grands pas de ne pouvoir rien dire sur les programmes qui bouclent. En particulier, que se passerait-il s'il l'on essayait d'évaluer un programme qui boucle ?

Il est possible d'étendre la sémantique naturelle avec des variables logiques pour rendre compte de la recherche d'une dérivation infinie, et donc parler de la divergence. On peut alors montrer qu'il est toujours possible d'étendre une évaluation partielle. Pour en savoir plus à ce sujet, on pourra lire [**?**].

La sémantique à réduction est une autre façon de définir la sémantique opérationnelle, à la fois plus simple, plus intuitive, plus précise et plus rigoureuse, mais aussi, malheureusement, moins concrète et donc plus éloignée de l'évaluation d'un programme par un interprète.

1.3 Typage

Nous venons de voir que les programmes syntaxiquement corrects se répartissent en trois catégories. La catégorie la plus intéressante est celle des programmes qui s'évaluent sur des valeurs. Mais les programmes qui bouclent sont quelques fois indispensables, par exemple pour l'écriture d'un système d'exploitation qui dans un monde parfait fonctionnerait indéfiniment. De toute façon, les programmes qui bouclent sont inévitables, parce que l'on ne sait pas statiquement les distinguer des programmes qui terminent sans se limiter à un sous-ensemble de programmes trop petit pour être intéressant. La dernière catégorie est celle des programmes dont l'évaluation peut se bloquer. Ces programmes sont clairement des programmes que l'on ne souhaite pas écrire.

Le typage est une façon de restreindre l'ensemble des programmes syntaxiquement corrects. Un de ses buts principaux est l'élimination de tous les programmes dont l'évaluation peut se bloquer. Un système de typage sera correct s'il réalise cet objectif. On parlera aussi d'un langage fortement typé. Cela ne peut cependant pas se faire sans éliminer aussi d'autres programmes. Un système de typage sera d'autant plus puissant qu'il rejettera moins de programmes corrects (i.e. qui s'évaluent en des valeurs). La correction est une propriété fondamentale d'un système de typage.

En fait un autre but du typage est d'éliminer les erreurs du programmeur. Ce qui n'est pas disjoint du but précédent, car il est clair qu'un programme qui se bloquerait n'est pas désiré par le programmeur et résulte sans doute d'une erreur de sa part. Mais on peut aussi souhaiter éliminer certains programmes qui s'évaluent correctement, par exemple un programme dont la correction reposerait sur l'ordre d'évaluation des arguments ! Dans ce but, un système de typage doit être relativement simple afin de permettre à l'utilisateur d'interpréter les erreurs de typage, ainsi que les informations de types éventuellement retournées par un vérificateur de types. Le typage est aussi un moyen de structurer les programmes. En particulier, il est souhaitable que les types puissent être calculés de façon incrémentale et qu'ils permettent la compilation séparée.

Ceci montre en particulier que la recherche de la puissance d'un système de typage n'est pas un but ultime, et qu'elle ne va pas sans compromis avec des considérations pratiques, de simplicité, de modularité.

1.4 Le système de type de ML

Les symboles de types sont les flèches $\rightarrow$, et les types de bases (types des entiers, type des chaînes de caractères, etc.) que nous désignons par g. D'une façon générale, nous utiliserons $\bar{e}$, pour désigner un tuple $(e_1, \ldots e_n)$. Les expressions de types, les schémas

de types, et les environnements de typage sont définis comme suit :

$\tau ::= \alpha \mid \tau \to \tau \mid g(\bar{\tau})$	Types
$\sigma ::= \forall\, \bar{\alpha}.\, \tau$	Schémas de type
$A ::= \emptyset \mid A \oplus (z : \sigma)$	Environnements de typage

Nous notons $FV(\tau)$ les variables de τ et $FV(\forall\, \bar{\alpha}.\, \tau)$ les variables libres dans le schéma de type $\forall\, \bar{\alpha}.\, \tau$, c'est-à-dire $FV(\tau) \setminus \{\bar{\alpha}\}$. Par extension, l'ensemble des variables libres $FV(A)$ dans un environnement de typage A est $\bigcup_{x \in dom(A)} FV(A(x))$.

Nous dirons que les types de la forme $\tau \to \tau'$ sont des types fonctionnel, et que les types de la forme $g(\bar{\tau})$ sont des types construits.

La relation de typage est définie comme la plus petite relation sur des triplets (A, a, τ), notée $A \vdash a : \tau$ satisfaisant les règles d'inférence données dans la figure 1.1, page 12. La généralisation du type τ dans l'environnement de typage A, notée $\mathrm{Gen}(\tau, A)$, est égale à $FV(\tau) \setminus FV(A)$.

(Fun)
$$\frac{A \oplus (x : \tau_2) \vdash a : \tau_1}{A \vdash \texttt{fun}\ (x)\ a : \tau_2 \to \tau_1}$$

(App)
$$\frac{A \vdash a_1 : \tau_2 \to \tau_1 \qquad A \vdash a_2 : \tau_2}{A \vdash a_1\ a_2 : \tau_1}$$

(Var-Inst)
$$\frac{z : \forall\, \bar{\alpha}.\, \tau_0 \in A}{A \vdash z : \tau_0[\bar{\tau}/\bar{\alpha}]}$$

(Gen-Let)
$$\frac{A \vdash a_1 : \tau_1 \qquad A \oplus (x : \mathrm{Gen}(\tau_1, A)) \vdash a_2 : \tau_2}{A \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

Figure 1.1 : Règles de typage pour le noyau ML.

1.5 Propriétés du typage

La relation de typage vérifie les propriétés suivantes :

Proposition 1 (Stabilité du typage par substitution) *Si $A \vdash a : \tau$ alors $\theta(A) \vdash a : \theta(\tau)$ pour toute substitution θ.*

Proposition 2 (Extension de l'environnement de typage) *Si les affectations de types A et B sont identiques partout sauf peut-être sur les variables qui ne sont pas libres dans a, alors $A \vdash a : \tau$ est dérivable si et seulement si $B \vdash a : \tau$ l'est.*

Une *instance* d'un schéma de type $\forall\, \bar{\alpha}.\, \tau$ est un type de la forme $\tau[\bar{\tau}'/\bar{\alpha}]$. Nous disons qu'un schéma de type $\forall\, \bar{\alpha}_1.\, \tau_1$ est *plus précis* qu'un schéma de type $\forall\, \bar{\alpha}_2.\, \tau_2$, si toute instance de $\forall\, \bar{\alpha}_1.\, \tau_1$ est aussi une instance de $\forall\, \bar{\alpha}_2.\, \tau_2$. C'est-à-dire, si τ_2 est de la forme $\tau_1[\bar{\tau}_1'/\bar{\alpha}_1]$ et les variables $\bar{\alpha}_2$ sont incluses dans $FV(\bar{\tau}_i')$ mais en dehors de $FV(\forall\, \bar{\alpha}_1.\, \tau_1)$.

Attention ! si σ_1 est plus précis que σ_2, on dit aussi que σ_2 est plus général que σ_1, mais ce vocabulaire est moins intuitif.

Proposition 3 (Généralisation du contexte) *Si $A \vdash a : \tau$ et si A et B ont même domaine, et que pour tout z de ce domaine, $B(z)$ est plus général que $A(z)$, alors $B \vdash a : \tau$.*

La stabilité par substitution est relativement difficile à montrer, mais indispensable à toute étude liée au typage. Les dernières propriétés sont très faciles. Toutes trois sont utilisées dans les preuves de ce chapitre. On pourra les montrer à titre d'exercice.