Chapter 1

Mini-ML: évaluation et typage

1.1 Syntaxe de mini-ML

Le langage mini-ML se compose d'expressions (notées a, a_1, a', ...) dont la syntaxe abstraite est la suivante:

Expressions:	$a ::= x$	identificateur (nom de variable)
	$\mid c$	constante
	$\mid op$	opération primitive
	$\mid \texttt{fun}\ x \to a$	abstraction de fonction
	$\mid a_1\ a_2$	application de fonction
	$\mid (a_1, a_2)$	construction d'une paire
	$\mid \texttt{let}\ x = a_1\ \texttt{in}\ a_2$	liaison locale

Nous omettons les détails de la syntaxe concrète (forme des identificateurs, parenthèses, priorités des opérations, ...).

La classe c contient des constantes comme par exemple des constantes entières 0, 1, 2, -1, ..., les booléens `true`, `false`, ou des chaînes littérales `"foo"`, ... La classe *op* contient des symboles d'opérations primitives, comme l'addition entière `+`, les projections `fst` et `snd` pour accéder aux composantes d'une paire, etc.

Exemples d'expressions:

```
+ (3, 2)
```
le calcul de 3 plus 2

```
3 + 2
```
le même, avec notation infixe pour le `+`

```
fun x → + (x, 1)
```
la fonction "successeur"

```
fun f → fun g → fun x → f(g x)
```
la composition de fonctions

```
let double = fun f → fun x → f(f x) in
let fois2 = fun x → + (x, x) in
```

```
let fois4 = double fois2 in
  double fois4
```

la fonction "fois 16"

D'autres constructions essentielles des langages de programmation peuvent également être présentées sous formes d'opérateurs prédéfinis. Par exemple, l'expression conditionnelle `if` a_1 `then` a_2 `else` a_3 peut être vue comme l'application d'un opérateur `ifthenelse` à $(a_1, (a_2, a_3))$. Plus surprenant: la définition de fonctions récursives peut également être ajoutée à mini-ML sous la forme de l'opérateur de point fixe `fix` (aussi noté Y dans la littérature du λ-calcul): intuitivement, `fix(fun` $f \rightarrow a$) est la fonction f qui vérifie $f = a$. Par exemple, la fonction factorielle s'écrit

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

et la fonction `power`$(f)(n)$, qui calcule la composée $f \circ \cdots \circ f$ (n compositions), s'écrit:

```
fix(fun power → fun f → fun n →
      if n = 0 then (fun x → x)
               else (fun x → power(f)(n-1)(f(x))))
```

Exercice 1.1 *(*) Pour définir des fonctions récursives en ML, on dispose de la construction spéciale* `let rec` $f\ x = a_1$ `in` a_2*. Traduire cette expression en mini-ML en terme de* `let` *et* `fix`*.*

Exercice 1.2 *(**) Même question pour la récursion mutuelle de ML:* `let rec` $f\ x = a_1$ `and` $g\ y = a_2$ `in` a_3*.*

1.2 Évaluation de mini-ML: sémantique opérationnelle "à grands pas"

Donner la sémantique d'un programme, c'est définir mathématiquement ce qu'il signifie, et en particulier comment se déroule son exécution. Nous allons maintenant définir la sémantique des expressions de mini-ML en formalisant leur évaluation, c'est-à-dire l'obtention du résultat (la "valeur") qu'elles dénotent. Ce style de sémantique s'appelle la sémantique opérationnelle, car elle s'attache à décrire le déroulement des opérations lors de l'exécution du programme. D'autres styles de sémantique prennent davantage de recul par-rapport au déroulement de l'exécution: sémantique dénotationnelle, sémantique axiomatique, ...

1.2.1 Les valeurs

La sémantique opérationnelle que nous donnons ici se présente comme une relation entre expressions a et valeurs v, notée $a \xrightarrow{v} v$ (lire: "l'expression a s'évalue en la valeur v"). Les valeurs v sont décrites par la grammaire suivante:

$$
\begin{array}{lll}
\text{Valeurs:} & v ::= \texttt{fun}\ x \to a & \text{valeurs fonctionnelles} \\
& \mid c & \text{valeurs constantes} \\
& \mid op & \text{primitives non appliquées} \\
& \mid (v_1, v_2) & \text{paire de deux valeurs}
\end{array}
$$

(Remarquons qu'ici les valeurs sont un sous-ensemble des expressions; ce n'est pas toujours le cas dans ce style de sémantique.)

1.2.2 Rappels sur les règles d'inférences

Nous allons utiliser des *règles d'inférence* pour définir la relation d'évaluation, ainsi que la plupart des relations utilisées dans ce cours. Une définition par règles d'inférence se compose d'axiomes A et de règles d'inférences de la forme

$$\frac{P_1 \quad P_2 \quad \ldots \quad P_n}{P}$$

qui se lisent "si $P_1, \ldots, P_n$ sont vraies, alors P est vraie". Une autre lecture de la règle d'inférence ci-dessus est comme l'implication $P_1 \wedge \ldots \wedge P_n \Rightarrow P$.

Les règles d'inférence et les axiomes peuvent contenir des variables libres, qui sont implicitement quantifiées universellement en tête de la règle. Par exemple, l'axiome $A(x)$ signifie $\forall x. A(x)$; la règle

$$\frac{P_1(x) \quad P_2(y)}{P(x, y)}$$

signifie $\forall x, y.\ P_1(x) \wedge P_2(y) \Rightarrow P(x, y)$.

Étant donné un ensemble d'axiomes et de règles d'inférences portant sur une ou plusieurs relations, on définit ces relations comme les plus petites relations qui satisfont les axiomes et les règles d'inférences. Par exemple, voici des règles sur des prédicats $\texttt{Pair}(n)$ et $\texttt{Impair}(n)$:

$$\texttt{Pair}(0) \qquad \frac{\texttt{Impair}(n)}{\texttt{Pair}(n+1)} \qquad \frac{\texttt{Pair}(n)}{\texttt{Impair}(n+1)}$$

Il faut les lire comme les conditions suivantes portant sur les prédicats `Pair` et `Impair`:

$$
\begin{array}{l}
\texttt{Pair}(0) \\
\forall n.\ \texttt{Impair}(n) \Rightarrow \texttt{Pair}(n+1) \\
\forall n.\ \texttt{Pair}(n) \Rightarrow \texttt{Impair}(n+1)
\end{array}
$$

Il y a de nombreux prédicats sur les entiers qui satisfont ces conditions, par exemple $\texttt{Pair}(n)$ et $\texttt{Impair}(n)$ vrais pour tout n. Cependant, les plus petits prédicats (les prédicats vrais les moins souvent) vérifiant ces conditions sont $\texttt{Pair}(n) = (n \bmod 2 = 0)$ et $\texttt{Impair}(n) = (n \bmod 2 = 1)$. Les règles d'inférence définissent donc `Pair` et `Impair` comme étant ces deux prédicats.

Exercice 1.3 *(**) Montrer que pour tout ensemble de règles d'inférence sur un prédicat, il existe toujours un plus petit prédicat satisfaisant ces règles. Pour être plus précis, on considèrera un ensemble d'axiomes et de règles sur un seul prédicat P à un paramètre:*

$$P(a_i(x)) \qquad\qquad \frac{P(b_j^1(x)) \quad \dots \quad P(b_j^n(x))}{P(c_j(x))}$$

(Indication: montrer que si on a une famille de prédicats $(P_k)_{k \in K}$ qui satisfont les règles, alors leur conjonction $\bigwedge_{k \in K} P_k$ les satisfait aussi.)

Une *dérivation* (encore appelée *arbre de preuve*) dans un système de règles d'inférence est un arbre portant aux feuilles des instances des axiomes et aux noeuds des conclusions de règles d'inférence dont les hypothèses sont justifiées par les fils du noeud dans l'arbre. La conclusion de la dérivation est le noeud racine de l'arbre. On représente généralement les dérivations par des "empilements" d'instances de règles d'inférence, avec la conclusion de la dérivation en bas. Par exemple, voici une dérivation qui conclut `Impair(3)` dans le système de règles ci-dessus:

$$\cfrac{\cfrac{\cfrac{\mathtt{Pair}(0)}{\mathtt{Impair}(1)}}{\mathtt{Pair}(2)}}{\mathtt{Impair}(3)}$$

Les dérivations caractérisent exactement les plus petits prédicats vérifiant un ensemble de règles d'inférences. Par exemple, $\mathtt{Pair}_{min}(n)$ est vrai (où $\mathtt{Pair}_{min}$ est le plus petit prédicat satisfaisant les règles d'inférence) si et seulement si il existe une dérivation qui conclut l'énoncé $\mathtt{Pair}(n)$.

Exercice 1.4 *(**) En se plaçant dans le même cadre que l'exercice 1.3, montrer que le prédicat défini par "il existe une dérivation de l'énoncé $P(x)$ dans le système de règles" est le plus petit prédicat satisfaisant le système de règles.*

1.2.3 Les règles d'évaluation

Nous définissons la relation $a \stackrel{v}{\to} v$ comme la plus petite relation satisfaisant les axiomes et les règles d'inférence suivants:

$$c \stackrel{v}{\to} c \ \ (1) \qquad\qquad op \stackrel{v}{\to} op \ \ (2) \qquad\qquad (\mathtt{fun}\ x \to a) \stackrel{v}{\to} (\mathtt{fun}\ x \to a) \ \ (3)$$

$$\frac{a_1 \stackrel{v}{\to} (\mathtt{fun}\ x \to a) \qquad a_2 \stackrel{v}{\to} v_2 \qquad a[x \leftarrow v_2] \stackrel{v}{\to} v}{a_1\ a_2 \stackrel{v}{\to} v}\ (4) \qquad\qquad \frac{a_1 \stackrel{v}{\to} v_1 \qquad a_2 \stackrel{v}{\to} v_2}{(a_1, a_2) \stackrel{v}{\to} (v_1, v_2)}\ (5)$$

$$\frac{a_1 \stackrel{v}{\to} v_1 \qquad a_2[x \leftarrow v_1] \stackrel{v}{\to} v}{(\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2) \stackrel{v}{\to} v}\ (6)$$

On a noté $a[x \leftarrow v]$ l'expression obtenue en substituant chaque occurrence de x libre dans a par v. Ainsi, $(+\ (x, 1))[x \leftarrow 2]$ est $+(2, 1)$, mais $(\mathtt{fun}\ x \to x)[x \leftarrow 2]$ est $(\mathtt{fun}\ x \to x)$.

Les règles 1, 2, 3 expriment que les constantes, les opérateurs et les fonctions sont déjà entièrement évalués: il n'y a rien à faire lors de l'évaluation. La règle 4 exprime que pour évaluer une application $a_1\ a_2$, il faut évaluer a_1 et a_2. Si la valeur de a_1 est une fonction $\mathtt{fun}\ x \to a$

(règle 4), le résultat de l'application est la valeur de a après substitution du paramètre formel x par l'argument effectif v_2 (la valeur de a_2). Enfin, pour une construction $\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2$, la règle 6 exprime qu'il faut d'abord évaluer a_1, puis substituer x par sa valeur dans a_2 et poursuivre avec l'évaluation de a_2.

Pour être complet, il faut ajouter des règles pour chaque opérateur *op* qui nous intéresse, décrivant l'évaluation des applications de cet opérateur. Par exemple, pour `+`, `fst` et `snd`, nous avons les règles

$$\frac{a_1 \stackrel{v}{\to} + \qquad a_2 \stackrel{v}{\to} (n_1, n_2) \qquad n_1, n_2 \text{ constantes entières et } n = n_1 + n_2}{a_1\ a_2 \stackrel{v}{\to} n} \quad (7)$$

$$\frac{a_1 \stackrel{v}{\to} \mathtt{fst} \qquad a_2 \stackrel{v}{\to} (v_1, v_2)}{a_1\ a_2 \stackrel{v}{\to} v_1} \quad (8) \qquad\qquad \frac{a_1 \stackrel{v}{\to} \mathtt{snd} \qquad a_2 \stackrel{v}{\to} (v_1, v_2)}{a_1\ a_2 \stackrel{v}{\to} v_2} \quad (9)$$

Exemple: nous avons $(\mathtt{fun}\ x \to +(x,1))\ 2 \stackrel{v}{\to} 3$, car la dérivation suivante est valide:

$$\frac{(\mathtt{fun}\ x \to +(x,1)) \stackrel{v}{\to} (\mathtt{fun}\ x \to +(x,1)) \qquad 2 \stackrel{v}{\to} 2 \qquad \dfrac{+ \stackrel{v}{\to} + \qquad \dfrac{2 \stackrel{v}{\to} 2 \qquad 1 \stackrel{v}{\to} 1}{(2,1) \stackrel{v}{\to} (2,1)}}{+(2,1) \stackrel{v}{\to} 3}}{(\mathtt{fun}\ x \to +(x,1))\ 2 \stackrel{v}{\to} 3}$$

Exercice 1.5 *(*) On considère l'expression $a = 1\ 2$. Existe-t'il une valeur v telle que $a \stackrel{v}{\to} v$? Même question avec l'expression $a' = (\mathtt{fun}\ f \to (f\ f))\ (\mathtt{fun}\ f \to (f\ f))$. Quelle différence voyez-vous entre ces deux exemples?*

Exercice de programmation 1.1 *(*) Implémenter un évaluateur pour le langage mini-ML qui suive les règles ci-dessus. Essayez-le sur les exemples d'expressions donnés dans ce chapitre. Comment votre évaluateur se comporte-t'il sur les expressions de l'exercice 1.5?*

1.2.4 Déterminisme de l'évaluation

Nous allons montrer que la sémantique de mini-ML est *déterministe*: une expression donnée ne peut pas s'évaluer en deux valeurs différentes. La preuve de cette propriété introduit une technique extrêmement puissante que nous utiliserons souvent par la suite: la récurrence sur une dérivation.

Proposition 1.1 (Déterminisme de l'évaluation) *Si $a \stackrel{v}{\to} v$ et $a \stackrel{v}{\to} v'$, alors $v = v'$.*

Démonstration: Nous savons par hypothèse qu'il existe des dérivations D de $a \stackrel{v}{\to} v$ et D' de $a \stackrel{v}{\to} v'$. Ces dérivations sont des arbres; nous pouvons donc raisonner par récurrence structurelle sur ces dérivations, et par cas sur la forme de l'expression a.

Cas a est une constante c. La seule règle d'évaluation qui s'applique à une constante est 1. Donc, les dérivations D et D' sont de la forme $c \stackrel{v}{\to} c$, et $v_1 = c$ et $v_2 = c$. D'où le résultat $v = v'$.

Cas a est un opérateur op ou une abstraction $\mathtt{fun}\ x \to a$. Comme le cas précédent.

Cas a est une paire (a_1, a_2). Une seule règle d'évaluation peut s'appliquer à a: la règle 5. Donc, la dérivation D est nécessairement de la forme

$$\frac{\begin{array}{cc} (D_1) & (D_2) \\ \vdots & \vdots \\ a_1 \stackrel{v}{\to} v_1 & a_2 \stackrel{v}{\to} v_2 \end{array}}{(a_1, a_2) \stackrel{v}{\to} (v_1, v_2)}$$

et de même D' est de la forme

$$\frac{\begin{array}{cc} (D'_1) & (D'_2) \\ \vdots & \vdots \\ a_1 \stackrel{v}{\to} v'_1 & a_2 \stackrel{v}{\to} v'_2 \end{array}}{(a_1, a_2) \stackrel{v}{\to} (v'_1, v'_2)}$$

Mais D_1 est une sous-dérivation de D, et D'_1 une sous-dérivation de D'. Nous pouvons donc appliquer l'hypothèse de récurrence à l'expression a_1, aux valeurs v_1 et v'_1 et aux dérivations D_1 et D'_1; il s'ensuit que $v_1 = v'_1$. Appliquant de même l'hypothèse de récurrence à l'expression a_2, aux valeurs v_2 et v'_2 et aux dérivations D_2 et D'_2, nous obtenons $v_2 = v'_2$. Donc, $v = (v_1, v_2) = (v'_1, v'_2) = v'$, ce qui est le résultat attendu.

Cas a est une application $a_1\ a_2$. Quatre règles d'évaluation s'appliquent à a: les règles 4, 7, 8 et 9. Donc, D et D' se terminent nécessairement par l'une de ces règles. Remarquons que ces quatre règles ont des prémisses de la forme $a_1 \stackrel{v}{\to} v_1$ et $a_2 \stackrel{v}{\to} v_2$ pour certaines valeurs de v_1 et v_2. Donc, D est nécessairement de la forme

$$\frac{\begin{array}{ccc} (D_1) & (D_2) & \\ \vdots & \vdots & \ldots \\ a_1 \stackrel{v}{\to} v_1 & a_2 \stackrel{v}{\to} v_2 & \end{array}}{a_1\ a_2 \stackrel{v}{\to} v}$$

et D' est aussi de la forme

$$\frac{\begin{array}{ccc} (D'_1) & (D'_2) & \\ \vdots & \vdots & \ldots \\ a_1 \stackrel{v}{\to} v'_1 & a_2 \stackrel{v}{\to} v'_2 & \end{array}}{a_1\ a_2 \stackrel{v}{\to} v'}$$

Comme D_1 est une sous-dérivation de D et D'_1 une sous-dérivation de D', nous pouvons appliquer l'hypothèse de récurrence à D_1 et D'_1. Il vient $v_1 = v'_1$. Procédant de même avec D_2 et D'_2, il vient $v_2 = v'_2$.

Nous pouvons maintenant raisonner par cas sur la forme de v_1, qui peut être une fonction ou un opérateur. Supposons par exemple $v_1 = \mathtt{fun}\ x \to a$. Compte tenu de ce que $v_1 = v'_1$ et $v_2 = v'_2$, les dérivations D et D' sont donc de la forme

$$\dfrac{\begin{array}{c}(D_1)\\ \vdots \\ a_1 \stackrel{v}{\to} \texttt{fun}\ x \to a\end{array} \quad \begin{array}{c}(D_2)\\ \vdots \\ a_2 \stackrel{v}{\to} v_2\end{array} \quad \begin{array}{c}(D_3)\\ \vdots \\ a[x \leftarrow v_2] \stackrel{v}{\to} v\end{array}}{a_1\ a_2 \stackrel{v}{\to} v}$$

$$\dfrac{\begin{array}{c}(D'_1)\\ \vdots \\ a_1 \stackrel{v}{\to} \texttt{fun}\ x \to a\end{array} \quad \begin{array}{c}(D'_2)\\ \vdots \\ a_2 \stackrel{v}{\to} v_2\end{array} \quad \begin{array}{c}(D'_3)\\ \vdots \\ a[x \leftarrow v_2] \stackrel{v}{\to} v'\end{array}}{a_1\ a_2 \stackrel{v}{\to} v'}$$

Appliquant une dernière fois l'hypothèse de récurrence aux dérivations D_3 et D'_3, il vient $v = v'$ comme attendu.

Si v_1 est un opérateur $+$, `fst`, ou `snd`, nous concluons directement $v = v'$ par examen des règles, sans avoir besoin d'invoquer l'hypothèse de récurrence une troisième fois.

Cas a est `let` $x = a_1$ `in` a_2. Le résultat découle de l'hypothèse de récurrence par un raisonnement analogue au cas de l'application. □

Exercice 1.6 *(*) Rédiger complètement le dernier cas de la preuve ci-dessus.*

Un corollaire de la proposition 1.1 est que l'évaluation de mini-ML est réellement une fonction partielle *eval* des expressions dans les valeurs: *eval*(a), si défini, est l'unique valeur v telle que $a \stackrel{v}{\to} v$.

Nous avons cependant gardé une présentation relationnelle, car elle s'étend plus facilement avec des primitives non-déterministes. Par exemple, donnons-nous une primitive `random`(n) qui renvoie un nombre réellement aléatoire entre 0 et n. Sa règle d'évaluation est:

$$\frac{a_1 \stackrel{v}{\to} \texttt{random} \qquad a_2 \stackrel{v}{\to} n \qquad n \text{ entier et } 0 \leq m \leq n}{a_1(a_2) \stackrel{v}{\to} m}$$

Avec cette règle, on a `random`$(1) \stackrel{v}{\to} 0$ et `random`$(1) \stackrel{v}{\to} 1$, exprimant que 0 et 1 sont deux valeurs correctes pour cette expression.

1.3 Typage de mini-ML

Le but du typage statique est de détecter et de rejeter à la compilation un certain nombre de programmes absurdes, comme 1 2 ou $(\texttt{fun}\ x \to x) + 1$. Cela passe par l'attribution d'un *type* à chaque sous-expression du programme (p.ex. `int` pour une expression entière, `int` $\to$ `int` pour une fonction des entiers dans les entiers, etc.) et la vérification de la cohérence de ces types. Pour être effctif, un système de types statique doit être décidable: il ne s'agit pas d'exécuter entièrement le programme lors de la compilation.

1.3.1 L'algèbre de types

Les expressions de types de mini-ML, ou plus simplement les types (notés τ), sont décrits par la syntaxe abstraite suivante:

Types:	$\tau ::= T$	type de base (`int`, `bool`, etc)
	$\mid \alpha$	variable de type
	$\mid \tau_1 \rightarrow \tau_2$	type des fonctions de τ_1 dans τ_2
	$\mid \tau_1 \times \tau_2$	type des paires de τ_1 et τ_2

1.3.2 Les règles de typage – typage monomorphe

La relation de typage porte sur des triplets (E, a, τ), où a est une expression, τ un type, et E un *environnement de typage* qui associe des types $E(x)$ aux identificateurs x libres dans a. La relation de typage est notée $E \vdash a : \tau$ et se lit "dans l'environnement E, l'expression a a le type τ", ou encore "sous les hypothèses de typage sur les variables exprimées dans E, l'expression a a le type τ".

Nous commençons par définir la relation de typage pour un système de types légèrement simplifié par-rapport à celui de ML: le typage monomorphe, aussi appelé "lambda-calcul simplement typé".

$$E \vdash x : E(x) \;\; \text{(var)} \qquad E \vdash c : TC(c) \;\; \text{(const)} \qquad E \vdash op : TC(op) \;\; \text{(op)}$$

$$\frac{E + \{x : \tau_1\} \vdash a : \tau_2}{E \vdash (\texttt{fun}\ x \rightarrow a) : \tau_1 \rightarrow \tau_2} \;\text{(fun)} \qquad \frac{E \vdash a_1 : \tau' \rightarrow \tau \qquad E \vdash a_2 : \tau'}{E \vdash a_1\ a_2 : \tau} \;\text{(app)}$$

$$\frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2} \;\text{(paire)} \qquad \frac{E \vdash a_1 : \tau_1 \qquad E + \{x : \tau_1\} \vdash a_2 : \tau_2}{E \vdash (\texttt{let}\ x = a_1\ \texttt{in}\ a_2) : \tau_2} \;\text{(let)}$$

Pour les règles (const) et (op), on se donne une fonction TC qui associe un type à chaque constructeur et opérateur; par exemple, $TC(0) = TC(1) = \texttt{int}$ et $TC(+) = \texttt{int} \times \texttt{int} \rightarrow \texttt{int}$.

Dans la règle (var), $E(x)$ est le type qui est associé à x dans l'environnement E. Dans les règles (fun) et (let), $E + \{x : \tau\}$ est l'environnement qui associe τ à x et qui est identique à E sur toute variable autre que x.

Exemples: voici une dérivation de typage dans le système ci-dessus:

$$\dfrac{\dfrac{\dfrac{\dfrac{\{x : \texttt{int}\} \vdash x : \texttt{int} \qquad \{x : \texttt{int}\} \vdash 1 : \texttt{int}}{\begin{array}{c}\{x : \texttt{int}\} \vdash (x, 1) : \texttt{int} \times \texttt{int} \\ \{x : \texttt{int}\} \vdash + : \texttt{int} \times \texttt{int} \rightarrow \texttt{int}\end{array}}}{\{x : \texttt{int}\} \vdash +(x, 1) : \texttt{int}}}{\emptyset \vdash \texttt{fun}\ x \rightarrow +(x, 1) : \texttt{int} \rightarrow \texttt{int}} \qquad \dfrac{\begin{array}{c}\{f : \texttt{int} \rightarrow \texttt{int}\} \vdash f : \texttt{int} \rightarrow \texttt{int} \\ \{f : \texttt{int} \rightarrow \texttt{int}\} \vdash 2 : \texttt{int}\end{array}}{\{f : \texttt{int} \rightarrow \texttt{int}\} \vdash f\ 2 : \texttt{int}}}{\emptyset \vdash \texttt{let}\ f = \texttt{fun}\ x \rightarrow +(x, 1)\ \texttt{in}\ f\ 2 : \texttt{int}}$$

Autres exemples de typages que l'on peut dériver:

$$\emptyset \vdash \texttt{fun}\ x \rightarrow x : \alpha \rightarrow \alpha$$
$$\emptyset \vdash \texttt{fun}\ x \rightarrow x : \texttt{bool} \rightarrow \texttt{bool}$$

Exemples de typages que l'on ne peut pas dériver:

$$\emptyset \vdash \texttt{fun}\ x \rightarrow +(x, 1) : \texttt{int}$$
$$\emptyset \vdash \texttt{fun}\ x \rightarrow +(x, 1) : \alpha \rightarrow \texttt{int}$$

Exemples d'expressions que l'on ne peut pas typer (il n'existe pas de E et de τ tels que $E \vdash a : \tau$):

$$1\ 2$$
$$\texttt{fun}\ f \rightarrow f\ f$$
$$\texttt{let}\ f = \texttt{fun}\ x \rightarrow x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$$

Exercice 1.7 *(*) Expliquer pourquoi ces trois dernières expressions ne sont pas typables.*

1.3.3 Schémas de types

La faiblesse du typage monomorphe est qu'un identificateur ne peut avoir qu'un seul type, même s'il est lié à une fonction naturellement polymorphe (comme la fonction identité f dans le dernier exemple). Pour dépasser cette limitation, nous introduisons la notion de schéma de types, une représentation compacte de tous les types qui peuvent être donnés à une expression polymorphe. Un schéma de types est une expression de types avec zéro, une ou plusieurs variables de types universellement quantifiées:

Schémas de types: $\sigma ::= \forall \alpha_1, \ldots, \alpha_n. \tau$

Lorsque l'ensemble des variables quantifiées est vide, on note simplement τ au lieu de $\forall. \tau$. Ainsi, les types peuvent être vus comme des schémas triviaux.

Les variables liées par $\forall$ peuvent être librement renommées (opération d'alpha-conversion), et les schémas de types sont considérés égaux modulo alpha-conversion:

$$\forall \alpha. \tau = \forall \beta. \tau[\alpha \leftarrow \beta] \qquad \text{si } \beta \text{ n'est pas libre dans } \tau \text{ ou si } \beta = \alpha$$

L'ensemble $\mathcal{L}(\tau), \mathcal{L}(\sigma), \mathcal{L}(E)$ des variables libres d'un type τ, d'un schéma de types σ ou d'un environnement E est formellement défini comme suit:

$$\begin{array}{rcl}
\mathcal{L}(T) & = & \emptyset \\
\mathcal{L}(\alpha) & = & \{\alpha\} \\
\mathcal{L}(\tau_1 \rightarrow \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\
\mathcal{L}(\tau_1 \times \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\
\mathcal{L}(\forall \alpha_1, \ldots, \alpha_n. \tau) & = & \mathcal{L}(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\} \\
\mathcal{L}(E) & = & \bigcup_{x \in \mathrm{Dom}(E)} \mathcal{L}(E(x))
\end{array}$$

Avec ces notations, l'égalité de deux schémas modulo alpha-conversion se définit formellement comme suit:

$$\forall \alpha_1 \ldots \alpha_n. \tau = \forall \beta_1 \ldots \beta_n. \tau[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n] \qquad \text{si } \beta_i \notin \mathcal{L}(\forall \alpha_1 \ldots \alpha_n. \tau) \text{ pour } i = 1, \ldots, n$$

Exercice 1.8 *(*)/(**) Montrer que* $\mathcal{L}(\tau[\alpha_1 \leftarrow \beta_1, \dots, \alpha_n \leftarrow \beta_n]) = \mathcal{L}(\tau)[\alpha_1 \leftarrow \beta_1, \dots, \alpha_n \leftarrow \beta_n]$. *En déduire que la définition de* $\mathcal{L}(\sigma)$ *passe bien au quotient par alpha-conversion.*

Un schéma de types peut être vu comme l'ensemble des types obtenus en instanciant (spécialisant) ses variables quantifiées par des types particuliers. Ainsi, $\forall\alpha.\,\alpha \to \alpha$ peut être vu comme l'ensemble des types $\{\tau \to \tau \mid \tau \text{ type}\}$. Pour formaliser cette intuition, on définit la relation $\tau \leq \sigma$ (lire: le type τ est une instance du schéma de types σ) de la manière suivante:

$$\tau \leq \forall\alpha_1 \dots \alpha_n.\,\tau' \text{ si et seulement si il existe } \tau_1, \dots, \tau_n \text{ tels que } \tau = \tau'[\alpha_1 \leftarrow \tau_1, \dots, \alpha_n \leftarrow \tau_n]$$

Exemples: $\texttt{int} \to \texttt{int}$ est une instance de $\forall\alpha.\,\alpha \to \alpha$, ainsi que $\texttt{bool} \to \texttt{bool}$, mais pas $\texttt{int} \to \texttt{bool}$.

Remarque: si σ est le schéma trivial $\forall.\tau'$, alors $\tau \leq \sigma$ est équivalent à $\tau = \tau'$.

1.3.4 Les règles de typage – typage polymorphe à la ML

Les règles de typage de mini-ML sont essentiellement les règles monomorphes de la section 1.3.2, avec les modifications suivantes:

- l'environnement de typage E associe des schémas de types (et non plus des types) aux identificateurs;
- de même, TC associe des schémas de types aux constantes et aux opérateurs, par exemple

$$TC(+) = \texttt{int} \times \texttt{int} \to \texttt{int} \qquad TC(\texttt{fst}) = \forall\alpha,\beta.\,\alpha \times \beta \to \alpha \qquad TC(\texttt{snd}) = \forall\alpha,\beta.\,\alpha \times \beta \to \beta$$

$$TC(\texttt{ifthenelse}) = \forall\alpha.\,\texttt{bool} \times \alpha \times \alpha \to \alpha \qquad TC(\texttt{fix}) = \forall\alpha.\,(\alpha \to \alpha) \to \alpha$$

- la règle de typage des identificateurs effectue une étape d'instanciation sur le schéma de type de l'identificateur;
- enfin, la règle du `let` généralise le type de l'expression liée avant de typer le corps du `let`.

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}\ \text{(var-inst)} \qquad \frac{\tau \leq TC(c)}{E \vdash c : \tau}\ \text{(const-inst)} \qquad \frac{\tau \leq TC(op)}{E \vdash op : \tau}\ \text{(op-inst)}$$

$$\frac{E + \{x : \tau_1\} \vdash a : \tau_2}{E \vdash (\texttt{fun}\ x \to a) : \tau_1 \to \tau_2}\ \text{(fun)} \qquad \frac{E \vdash a_1 : \tau' \to \tau \qquad E \vdash a_2 : \tau'}{E \vdash a_1\ a_2 : \tau}\ \text{(app)}$$

$$\frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2}\ \text{(paire)} \qquad \frac{E \vdash a_1 : \tau_1 \qquad E + \{x : Gen(\tau_1, E)\} \vdash a_2 : \tau_2}{E \vdash (\texttt{let}\ x = a_1\ \texttt{in}\ a_2) : \tau_2}\ \text{(let-gen)}$$

Dans la règle (let-gen), l'opérateur Gen est défini comme suit:

$$Gen(\tau_1, E) = \forall\alpha_1, \dots, \alpha_n.\,\tau_1 \quad \text{où} \quad \{\alpha_1, \dots, \alpha_n\} = \mathcal{L}(\tau_1) \setminus \mathcal{L}(E)$$

Autrement dit, $Gen(\tau_1, E)$ est τ_1 dans lequel on a généralisé toutes les variables qui ne sont pas libres dans l'environnement E.

Exemple:

$$\dfrac{\dfrac{\dfrac{\alpha \leq \alpha}{\{x:\alpha\}\vdash x:\alpha}}{\emptyset \vdash \texttt{fun}\ x \to x : \alpha\to\alpha} \qquad \dfrac{\dfrac{\texttt{int}\to\texttt{int} \leq \forall\alpha.\,\alpha\to\alpha}{\{f:\forall\alpha.\,\alpha\to\alpha\}\vdash f:\texttt{int}\to\texttt{int}} \qquad \{f:\forall\alpha.\,\alpha\to\alpha\}\vdash 1:\texttt{int}}{\{f:\forall\alpha.\,\alpha\to\alpha\}\vdash f\ 1:\texttt{int}}}{\emptyset\vdash \texttt{let}\ f = \texttt{fun}\ x\to x\ \texttt{in}\ f\ 1 : \texttt{int}}$$

Exercice 1.9 *(*) Peut-on typer les expressions ci-dessous en mini-ML? Avec quels types?*

$$\begin{array}{l}\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ f\ f \\ \texttt{fun}\ f \to f\ f\end{array}$$

Exercice 1.10 *(**) Une définition plus simple de Gen serait $Gen(\tau_1, E) = \forall\alpha_1, \ldots, \alpha_n.\,\tau_1$ où $\{\alpha_1, \ldots, \alpha_n\} = \mathcal{L}(\tau_1)$. (Autrement dit, on généralise toutes les variables de τ_1, même celles qui sont libres dans E.) Montrer sur un exemple que cela conduit à des typages incorrects (c.à.d. qui attribuent des types sémantiquement trop généraux à certaines expressions).*

Exercice de programmation 1.2 *Implémenter la fonction $\mathcal{L}$ (*), la fonction Gen (*) et le prédicat $\leq$ (**).*

1.3.5 Quelques propriétés des règles de typage

Nous donnons maintenant trois propriétés du prédicat de typage qui nous servirons par la suite. La première est que le typage est stable par substitution de variables de types: si on peut dériver un typage contenant des variables de types libres dans le type ou l'environnement, comme par exemple

$$\{f : \alpha \to \alpha;\ x : \alpha\} \vdash f\ x : \alpha$$

alors on peut aussi dériver tous les typages obtenus en remplaçant ces variables de types par des expressions de types, comme par exemple

$$\{f : \texttt{int} \to \texttt{int};\ x : \texttt{int}\} \vdash f\ x : \texttt{int}$$

Pour que cette propriété soit vraie, il faut supposer que les schémas $TC(c)$ et $TC(op)$ sont clos (sans variables libres) pour tous c et op. C'est le cas pour les constantes et opérateurs que nous avons vus jusqu'ici, et tous ceux que nous introduirons dans la suite du cours.

Proposition 1.2 (Stabilité du typage par substitution) *Soit φ une substitution. Si $E \vdash a : \tau$, alors $\varphi(E) \vdash a : \varphi(\tau)$.*

La seconde propriété est que les typages ne changent pas si dans l'environnement de typage on ajoute ou supprime des hypothèses de typage portant sur des variables non libres dans l'expression. Par exemple, si la seule variable libre dans a est x, alors on peut dériver

$$\{x : \sigma_x; y : \texttt{int}\} \vdash a : \tau$$

si et seulement si on peut dériver

$$\{x : \sigma_x; z : \texttt{bool}\} \vdash a : \tau$$

Proposition 1.3 (Indifférence du typage vis-à-vis des hypothèses inutiles) *Supposons $E_1(x) = E_2(x)$ pour tout identificateur x libre dans l'expression a. Alors $E_1 \vdash a : \tau$ si et seulement si $E_2 \vdash a : \tau$.*

La troisième propriété est que tous les typages que l'on peut dériver sous certaines hypothèses peuvent être dérivés sous des hypothèses "plus fortes". Pour formaliser cette notion de "plus fort", on dit qu'un schéma de type σ' est *plus général* qu'un autre schéma σ, et on note $\sigma' \geq \sigma$, si toute instance de σ est aussi instance de σ'. On montre facilement que $\sigma' \geq \forall \alpha_1 \ldots \alpha_n . \tau$ (où les α_i sont choisies non libres dans σ') si et seulement si $\tau \leq \sigma'$.

Proposition 1.4 (Stabilité du typage par renforcement des hypothèses) *Supposons $\mathrm{Dom}(E) = \mathrm{Dom}(E')$ et $E'(x) \geq E(x)$ pour tout $x \in \mathrm{Dom}(E)$. Alors $E \vdash a : \tau$ implique $E' \vdash a : \tau$.*

La preuve des propositions 1.2, 1.3 et 1.4 est facile dans le cas du système de types monomorphe de la section 1.3.2 (par récurrence sur les dérivations de typage), mais beaucoup plus difficile dans le système de types de ML (les récurrences "passent" difficilement sur le cas de la règle (let-gen)).

Exercice 1.11 *(***) Prouver la proposition 1.2. (On commencera par définir précisément l'image $\varphi(\sigma)$ d'un schéma de types σ par une substitution φ.)*