Notes du cours de DEA
“Typage et programmation”

François Pottier, d'après les notes de Xavier Leroy
Francois.Pottier@inria.fr

Version du 5 décembre 2002

Table des matières

Chapitre 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 unaire
		$\mid$ `fun` $x \to a$	abstraction de fonction
		$\mid a\ a$	application de fonction
		$\mid (a, a)$	construction d'une paire
		$\mid$ `fst` a	projection gauche
		$\mid$ `snd` a	projection droite
		$\mid$ `let` $x = a$ `in` a	liaison locale

Nous omettons les détails de la syntaxe concrète : forme des identificateurs, parenthèses, priorités des opérations, etc. C'est pourquoi on parle de syntaxe abstraite : une expression est ici un objet mathématique – un arbre – et non une chaîne de caractères. La définition ci-dessus peut être vue comme la définition d'un ensemble d'arbres, ou comme la définition d'une grammaire sous forme BNF. Le principal symbole non-terminal y est a, la méta-variable que nous employons pour représenter une expression. Les autres symboles non-terminaux sont x, c et op ; ces méta-variables représentent respectivement un identificateur, une constante, et une opération primitive.

La construction `fun` $x \to a$ représente une fonction qui au paramètre formel x associe la valeur calculée par l'expression a. (On la note également $\lambda x.a$.) L'identificateur x peut bien sûr apparaître dans l'expression a. La construction $a_1\ a_2$ représente l'application de l'expression a_1 (dont la valeur doit être une fonction) à l'argument a_2. L'expression (a_1, a_2) produit une paire dont les composantes sont les (valeurs calculées par les) expressions a_1 et a_2. Enfin, si a produit une paire, alors les constructions `fst` a et `snd` a permettent d'en obtenir les composantes. En termes de représentation machine, une paire est un bloc de mémoire contenant deux pointeurs (un pointeur vers chacune de ses composantes). En termes de syntaxe abstraite, cependant, une paire (a_1, a_2) est simplement un terme doté de deux sous-termes a_1 et a_2. Enfin, la construction `let` $x = a_1$ `in` a_2 *lie* l'identificateur x à la valeur produite par a_1, puis évalue a_2. L'expression a_2 peut donc faire référence à la variable x. Pour résumer cette définition de façon succincte, on peut dire que mini-ML est un λ-calcul étendu par une construction `let`, des paires, des constantes et des opérations primitives.

L'ensemble des identificateurs n'est pas spécifié ; il est simplement supposé dénombrable. L'ensemble des constantes peut contenir, par exemple, les entiers 0, 1, 2, -1, …, les booléens `true` et `false`, les chaînes de caractères littérales `"foo"`, …, etc. Parmi les opérations primitives, on pourra trouver par exemple l'addition entière `+`, la concaténation de chaînes de caractères `^`, etc. Par souci de simplicité, on suppose que toutes les opérations primitives sont *unaires*, c'est-à-dire qu'elles s'appliquent à un seul argument. L'addition entière est bien sûr considérée habituellement comme une opération binaire ; ici, on l'appliquera à une *paire* d'entiers, donc à un seul argument. On aurait pu présenter les choses de façon plus générale en supposant chaque opération dotée d'une *arité* propre, c'est-à-dire d'un entier indiquant combien d'arguments elle attend. Les constantes seraient alors des opérations d'arité 0.

Exemples d'expressions :

`+ (3, 2)`	le calcul de 3 plus 2
`3 + 2`	le même, avec notation infixe pour le `+` (*sucre syntaxique*)
`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 utiles, voire essentielles, peuvent être ajoutées au langage à l'aide de codages et/ou d'opérations primitives supplémentaires. Par exemple, on peut se doter de n-uplets (également appelés *tuples*) en les codant en termes de paires imbriquées.

Exercice 1.1 *(*) Traduire la construction d'un n-uplet* $(a_1, a_2, \ldots, a_n)$ *et l'accès à la k-ième composante d'un n-uplet* $\pi_k\ a$ *à l'aide des opérations de construction et d'accès aux paires.*

Plus surprenant, la définition de fonctions récursives peut également être ajoutée à mini-ML sous la forme d'un opérateur de point fixe `fix` (souvent 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 1.2 *(*) 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 1.3 *(**) Même question pour la récursion mutuelle de ML :* `let rec` $f\ x = a_1$ `and` $g\ y = a_2$ `in` a_3.

L'intérêt de pouvoir réduire ces constructions évoluées, à travers un codage, à de simples opérations primitives, est de démontrer que le langage mini-ML, muni de ces primitives, est suffisamment expressif. Cela signifie que nous nous concentrerons, dans la suite, sur mini-ML, et n'aurons pas besoin, dans nos preuves, de considérer un langage plus lourd.

1.2 α-conversion et substitution sans capture

Pour compléter notre définition de la syntaxe de mini-ML, il faut préciser que la construction $\texttt{fun}\ x \to a$ *lie* la variable x dans l'expression a, et que la construction $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$ lie x dans a_2. Intuitivement, cela signifie que le nom x est *local* à ces constructions et n'a pas d'importance à l'extérieur : on pourrait le remplacer par un autre sans changer la signification de la construction. Formalisons cette idée. On commence par définir l'ensemble des variables *libres* dans une expression a, noté $\mathcal{L}(a)$, par induction sur la structure de a :

$$\begin{aligned}
\mathcal{L}(x) &= x \\
\mathcal{L}(c) = \mathcal{L}(op) &= \emptyset \\
\mathcal{L}(\texttt{fun}\ x \to a) &= \mathcal{L}(a) \setminus \{x\} \\
\mathcal{L}(a_1\ a_2) = \mathcal{L}((a_1, a_2)) &= \mathcal{L}(a_1) \cup \mathcal{L}(a_2) \\
\mathcal{L}(\texttt{fst}\ a) = \mathcal{L}(\texttt{snd}\ a) &= \mathcal{L}(a) \\
\mathcal{L}(\texttt{let}\ x = a_1\ \texttt{in}\ a_2) &= \mathcal{L}(a_1) \cup (\mathcal{L}(a_2) \setminus \{x\})
\end{aligned}$$

On dit qu'une variable x apparaît *libre* dans a si et seulement si $x \in \mathcal{L}(a)$. Intuitivement, cela signifie que le nom x a de l'importance dans a ; ce n'est pas un nom interne. En effet, en observant la définition ci-dessus, on note que x n'est pas libre dans $\texttt{fun}\ x \to a$, même s'il est libre dans a ; il est supprimé de l'ensemble des noms libres par le *lieur* `fun`. On dit alors que x est *liée*. Le mécanisme est le même pour le lieur `let`. Une expression a est dite *close* si elle n'a aucune variable libre.

Une fois cette définition acquise, on considère que deux expressions sont équivalentes (on dit : α-convertibles) si elles ne diffèrent que par un renommage de leurs variables liées. Plus précisément, si y est une variable quelconque *mais non libre dans* a, alors les expressions $\texttt{fun}\ x \to a$ et $\texttt{fun}\ y \to a[x \leftarrow y]$ sont α-convertibles. La notation $a[x \leftarrow y]$ représente la substitution *sans capture* de x par y dans l'expression a, que nous définissons plus bas.

Exercice 1.4 *(*) Pourquoi interdit-on que y soit libre dans a ? Examiner ce qu'on obtient si on lève cette restriction.*

De même, si y n'est pas libre dans a_2, alors $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$ et $\texttt{let}\ y = a_1\ \texttt{in}\ a_2[x \leftarrow y]$ sont α-convertibles. On formalise ainsi l'idée que le nom des variables internes n'a pas d'importance. Dans la suite, on travaillera toujours "modulo α-conversion", c'est-à-dire qu'on se permettra d'effectuer silencieusement, si besoin, de telles étapes d'α-conversion. En d'autres termes, on quotiente en fait l'ensemble des expressions par la relation d'α-convertibilité. D'un point de vue strictement formel, cela impose de distinguer éléments de l'ensemble quotient et éléments de l'ensemble originel. En pratique, cette distinction reste la plupart du temps implicite, par souci de concision. Néanmoins, à chaque fois que l'on définit une notion par induction sur la structure des expressions, il faut en principe vérifier qu'elle est indépendante du représentant choisi.

Exercice 1.5 *(*) Vérifier que la fonction $\mathcal{L}$ est compatible avec l'α-conversion, c'est-à-dire que l'ensemble des variables libres de a est indépendant du représentant choisi pour a.*

Reste, pour compléter notre définition de l'α-conversion, à définir la notion de substitution *sans capture* évoquée ci-dessus. La forme générale en est $a[x \leftarrow a']$, que l'on lit "la substitution sans capture de la variable x par l'expression a' dans l'expression a". Elle se définit par induction sur la structure des expressions :

$$\begin{array}{rll}
x[x \leftarrow a'] & = a' & \\
y[x \leftarrow a'] & = y & \text{si } y \neq x \\
c[x \leftarrow a'] & = c & \\
op[x \leftarrow a'] & = op & \\
(\texttt{fun } y \rightarrow a)[x \leftarrow a'] & = \texttt{fun } y \rightarrow (a[x \leftarrow a']) & \text{si } y \neq x \text{ et } y \notin \mathcal{L}(a') \\
(a_1\ a_2)[x \leftarrow a'] & = a_1[x \leftarrow a']\ a_2[x \leftarrow a'] & \\
(\texttt{let } y = a_1 \texttt{ in } a_2)[x \leftarrow a'] & = \texttt{let } y = a_1[x \leftarrow a'] \texttt{ in } a_2[x \leftarrow a'] & \text{si } y \neq x \text{ et } y \notin \mathcal{L}(a')
\end{array}$$

(Nous avons omis les cas concernant les paires.) Dans l'ensemble, la définition est simple ; les seuls points intéressants sont les conditions portant sur la variable y lorsque l'on s'intéresse à une sous-expression où y est liée, par exemple $\texttt{fun } y \rightarrow a$. On exige alors que y soit hors de $\{x\} \cup \mathcal{L}(a')$, condition que l'on peut toujours satisfaire en appliquant une étape d'α-conversion à la sous-expression $\texttt{fun } y \rightarrow a$, puisque la variable y lui est interne. Cette condition, qui n'est donc pas restrictive, évite la *capture* de y. En termes intuitifs, on est train de "passer sous" un lieur pour y, donc la signification du nom y change : elle n'est pas la même dans a et à l'extérieur de $\texttt{fun } y \rightarrow a$. Or la signification de la substitution $[x \leftarrow a']$ ne doit pas changer, donc y ne doit pas être x ni apparaître (libre) dans a'.

Exercice 1.6 *(*) Calculer* $(x + 1)[x \leftarrow 2]$. *Calculer* $(\texttt{fun } x \rightarrow x)[x \leftarrow 2]$. *Calculer* $(\texttt{fun } x \rightarrow x + y)[y \leftarrow x]$.

Exercice 1.7 *(*) Vérifier que si x n'apparaît pas libre dans a, alors $a[x \leftarrow a']$ est égal à a.*

Le lecteur attentif aura remarqué que nous avons défini la relation d'α-convertibilité à l'aide de l'opération de substitution sans capture, et vice-versa. Nos définitions seraient-elles donc mal fondées ? Un argument de taille montre que ce n'est pas le cas : pour définir l'α-convertibilité entre deux termes, nous avons appliqué l'opération de substitution à un terme plus petit ; pour définir l'application d'une substitution à un terme, nous avons appliqué l'α-conversion à ce même terme.

1.3 Rappels sur les règles d'inférence

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'inférence 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 $P_1 \wedge \ldots \wedge P_n \Rightarrow P$. Dans une telle règle, $P_1, P_2, \ldots P_n$ sont appelées *prémisses* et P *conclusion.*

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 $\forall x, y.\ P_1(x) \wedge P_2(y) \Rightarrow P(x, y)$. On notera qu'un axiome peut être vu comme une règle dotée de zéro prémisses ; dans la suite, on parlera donc de "règles" pour "axiomes ou règles".

Étant donné un ensemble règles d'inférence portant sur une ou plusieurs relations, on définit ces relations comme les plus petites relations qui satisfont les règles d'inférence. En d'autres termes, ces relations sont définies de façon *inductive* par les règles. Par exemple, voici des règles sur des prédicats $\texttt{Pair}(n)$ et $\texttt{Impair}(n)$:

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

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 1.8 *(**) 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 de règles sur un seul prédicat P à un paramètre :*

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

(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 à chaque nœud la conclusion d'une règle d'inférence dont les prémisses correspondent aux fils de ce nœud. Les feuilles de l'arbre portent donc des axiomes. La conclusion de la dérivation est le nœud 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 :

$$\dfrac{\dfrac{\dfrac{\texttt{Pair}(0)}{\texttt{Impair}(1)}}{\texttt{Pair}(2)}}{\texttt{Impair}(3)}$$

Les plus petits prédicats vérifiant un ensemble de règles d'inférence sont donnés, de façon explicite, par les dérivations, comme énoncé par l'exercice ci-dessous.

Exercice 1.9 *(**) En se plaçant dans le même cadre que l'exercice 1.8, 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.*

Par exemple, $\texttt{Pair}_{min}(n)$ est vrai (où $\texttt{Pair}_{min}$ est le plus petit prédicat satisfaisant les règles d'inférence) si et seulement si il existe une dérivation qui conclut l'énoncé $\texttt{Pair}(n)$. Un énoncé (également appelé *jugement*) qui admet une dérivation est dit *dérivable*.

1.4 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 consiste en une série de règles qui modélisent le déroulement des opérations lors de l'exécution du programme. Plus précisément, il s'agit d'une sémantique opérationnelle *à grands pas*, car elle donne directement le résultat de l'exécution ; par opposition à la sémantique opérationnelle *à petits pas*, également appelée sémantique à réductions, que nous verrons au chapitre 2. La sémantique opérationnelle à grands pas est également appelée *sémantique opérationnelle structurée* (SOS) ou *sémantique naturelle* – un vocabulaire redondant, mais qu'il peut être utile de connaître. Signalons qu'il existe des sémantiques non opérationnelles ; par exemple, la sémantique *dénotationnelle* consiste à interpréter chaque programme comme un objet dans un certain domaine mathématique ; la sémantique *axiomatique* décrit chaque construction du langage par ses pré-condition et post-condition, c'est-à-dire par la relation logique qui existe entre les états avant et après son exécution. Enfin, mentionnons un dernier point de vocabulaire : lorsque nous Français parlons de *sémantique*, les anglophones parlent souvent de *dynamic semantics*, et lorsque nous parlons de *typage*, ils parlent parfois de *static semantics*.

1.4.1 Les valeurs

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

Valeurs :	$v ::= \mathtt{fun}\ x \to a$	valeurs fonctionnelles
	$\mid c$	valeurs constantes
	$\mid op$	primitives non appliquées
	$\mid (v_1, v_2)$	paire de deux valeurs

Pourquoi ne s'intéresser qu'aux expressions closes ? Une expression close est une expression sans variables libres, c'est-à-dire un programme sans aucune référence externe. On peut proposer une analogie avec l'édition de liens : une expression non close, dans laquelle subsistent des noms libres, peut être vue comme un composant de programme ; une fois les différents composants assemblés par un *linker*, on obtient un programme complet, c'est-à-dire une expression close. En pratique, on n'exécute un programme qu'après avoir effectué l'édition de liens ; en termes formels, on n'exécute que des programmes clos. Naturellement, si l'expression de départ a est close, et si a produit la valeur v, alors v sera close également, car aucune règle d'exécution ne pourra faire apparaître une nouvelle référence externe ; on peut vérifier que les règles données plus bas vérifient cette propriété.

Remarquons qu'ici les valeurs sont un sous-ensemble des expressions ; ce n'est pas toujours le cas dans ce style de sémantique. En effet, la sémantique peut faire intervenir des notions qui n'apparaissent pas dans la syntaxe du langage, car elles correspondent à des représentations internes, qui n'existent qu'en machine. C'est le cas par exemple des sémantiques où l'évaluation d'une fonction produit une *clôture*, une forme particulière de valeur qui contient un pointeur de code et un pointeur d'environnement.

1.4.2 Les règles d'évaluation

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

$$c \twoheadrightarrow c \;\; (1) \qquad op \twoheadrightarrow op \;\; (2) \qquad (\texttt{fun}\ x \to a) \twoheadrightarrow (\texttt{fun}\ x \to a) \;\; (3)$$

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

$$\frac{a \twoheadrightarrow (v_1, v_2)}{\texttt{fst}\ a \twoheadrightarrow v_1 \qquad \texttt{snd}\ a \twoheadrightarrow v_2}\ (6) \qquad \frac{a_1 \twoheadrightarrow v_1 \qquad a_2[x \leftarrow v_1] \twoheadrightarrow v}{(\texttt{let}\ x = a_1\ \texttt{in}\ a_2) \twoheadrightarrow v}\ (7)$$

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 $\texttt{fun}\ x \to a$, 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). Notons que l'argument effectif a_2 est évalué avant l'appel de la fonction. Le résultat de cette évaluation, à savoir la valeur v_2, devient alors l'argument effectif de la fonction. Cette stratégie, connue sous le nom d'*appel par valeur*, est celle employée par les langages Standard ML et O'Caml. D'autres, comme Haskell, emploient une stratégie dite d'*appel par nom*, qui consiste à passer l'argument *non évalué* à la fonction ; celui-ci sera évalué seulement si et dans la mesure où la fonction en a effectivement besoin. Nous ne donnerons pas plus de détails ici. Les règles (5) et (6) décrivent la construction et l'accès aux paires. Notons que la règle (6) a deux conclusions, ce qui est légèrement inhabituel ; on peut voir cela comme une notation compacte pour deux règles dotées chacune d'une seule conclusion. Enfin, pour une construction $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$, la règle (7) 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 `+`, nous avons la règle

$$\frac{a_1 \twoheadrightarrow + \qquad a_2 \twoheadrightarrow (n_1, n_2)}{a_1\ a_2 \twoheadrightarrow n_1 + n_2}\ (8)$$

Notons que la notation $n_1 + n_2$ qui apparaît dans la conclusion représente l'expression constante $n_1 + n_2$, et non l'application de l'opérateur $+$ à la paire (n_1, n_2). En d'autres termes, il s'agit d'une addition dans le méta-langage (mathématique), et non dans le langage (de programmation).

La sémantique de l'opérateur `fix` est un peu plus complexe :

$$\frac{a \twoheadrightarrow (\texttt{fun}\ f \to a_1) \qquad a_1[f \leftarrow \texttt{fix}\ (\texttt{fun}\ f \to a_1)] \twoheadrightarrow v}{\texttt{fix}\ a \twoheadrightarrow v}\ (9)$$

Cette règle exprime le fait que $\texttt{fix}\ \texttt{fun}\ f \to a_1$ se comporte comme l'expression a_1, dans laquelle la variable f a été remplacée par l'expression $\texttt{fix}\ \texttt{fun}\ f \to a_1$ elle-même. Ainsi, toute référence à f dans a_1 se comporte comme une référence *récursive* à l'objet $\texttt{fix}\ \texttt{fun}\ f \to a_1$.

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

$$\dfrac{(\texttt{fun}\ x \to +(x,1)) \twoheadrightarrow (\texttt{fun}\ x \to +(x,1)) \qquad 2 \twoheadrightarrow 2 \qquad \dfrac{+ \twoheadrightarrow + \qquad \dfrac{2 \twoheadrightarrow 2 \qquad 1 \twoheadrightarrow 1}{(2,1) \twoheadrightarrow (2,1)}}{+(2,1) \twoheadrightarrow 3}}{(\texttt{fun}\ x \to +(x,1))\ 2 \twoheadrightarrow 3}$$

Exercice 1.10 *(*) On considère l'expression $a = 1\ 2$. Existe-t'il une valeur v telle que $a \twoheadrightarrow v$? Même question avec l'expression $a' = (\texttt{fun}\ f \to (f\ f))\ (\texttt{fun}\ f \to (f\ f))$. 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 1.10 ?*

1.4.3 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 (ou *induction*) sur une dérivation.

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

Démonstration: Nous savons par hypothèse qu'il existe des dérivations D de $a \twoheadrightarrow v$ et D' de $a \twoheadrightarrow v'$. 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 $c \twoheadrightarrow c$, et $v = c$ et $v' = c$. D'où le résultat $v = v'$.

Cas a est un opérateur op ou une abstraction $\texttt{fun}\ x \to a$. 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

$$\dfrac{\begin{array}{c}(D_1)\\ \vdots \\ a_1 \twoheadrightarrow v_1\end{array} \qquad \begin{array}{c}(D_2)\\ \vdots \\ a_2 \twoheadrightarrow v_2\end{array}}{(a_1, a_2) \twoheadrightarrow (v_1, v_2)}$$

et de même D' est de la forme

$$\dfrac{\begin{array}{c}(D'_1)\\ \vdots \\ a_1 \twoheadrightarrow v'_1\end{array} \qquad \begin{array}{c}(D'_2)\\ \vdots \\ a_2 \twoheadrightarrow v'_2\end{array}}{(a_1, a_2) \twoheadrightarrow (v'_1, v'_2)}$$

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 $a_1\ a_2$. Deux règles d'évaluation s'appliquent à a : les règles (4) et (8). Donc, D et D' se terminent nécessairement par l'une de ces règles. Remarquons que ces deux règles ont des prémisses de la forme $a_1 \twoheadrightarrow v_1$ et $a_2 \twoheadrightarrow v_2$ pour certaines valeurs de v_1 et v_2. Donc, D est nécessairement de la forme

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

et D' est aussi de la forme

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

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 $v_1 = \texttt{fun}\ x \to a$. Compte tenu de ce que $v_1 = v'_1$ et $v_2 = v'_2$, les dérivations D et D' sont donc de la forme

$$\frac{\begin{array}{ccc} (D_1) & (D_2) & (D_3) \\ \vdots & \vdots & \vdots \\ a_1 \twoheadrightarrow \texttt{fun}\ x \to a & a_2 \twoheadrightarrow v_2 & a[x \leftarrow v_2] \twoheadrightarrow v \end{array}}{a_1\ a_2 \twoheadrightarrow v}$$

$$\frac{\begin{array}{ccc} (D'_1) & (D'_2) & (D'_3) \\ \vdots & \vdots & \vdots \\ a_1 \twoheadrightarrow \texttt{fun}\ x \to a & a_2 \twoheadrightarrow v_2 & a[x \leftarrow v_2] \twoheadrightarrow v' \end{array}}{a_1\ a_2 \twoheadrightarrow v'}$$

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 $+$, nous concluons directement $v = v'$ par examen des règles, sans avoir besoin d'invoquer l'hypothèse de récurrence une troisième fois.

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

Exercice 1.11 *(*) 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 en réalité une fonction partielle *eval* des expressions dans les valeurs : $eval(a)$, si défini, est l'unique valeur v telle que $a \twoheadrightarrow v$.

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

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

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

1.5 Typage de mini-ML

Le but du typage statique est de détecter et de rejeter dès la compilation un certain nombre de programmes absurdes, comme 1 2 ou $(\texttt{fun}\ x \rightarrow x) + 1$. Cela passe par l'attribution d'un *type* à chaque sous-expression du programme (p.ex. `int` pour une expression entière, $\texttt{int} \rightarrow \texttt{int}$ pour une fonction des entiers dans les entiers, etc.) et la vérification de la cohérence de ces types.

Déterminer si un programme provoque une erreur, tout comme déterminer si un programme s'arrête, est un problème indécidable. Or, les systèmes de types que nous utiliserons seront décidables, car on souhaite que le compilateur termine, quel que soit le programme qu'on lui soumet. Il sera donc impossible de ne rejeter *que* les programmes erronés. Tout système de types rejette certains programmes innocents, et c'est pourquoi la quête de "meilleurs" systèmes de types est sans fin.

1.5.1 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

On notera que la grammaire des types ne contient aucun lieur ; on dit que les types sont des termes du *premier ordre*. Ainsi, une variable α *apparaît libre* dans un type τ si et seulement si elle apparaît dans τ. On continue néanmoins, par habitude, à employer la locution "apparaît libre". Par ailleurs, aucune notion d'α-conversion sur les types n'est introduite ; l'égalité sur les types est l'égalité syntaxique.

1.5.2 Les règles de typage – typage monomorphe

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 λ-calcul simplement typé.

La relation de typage porte sur des triplets (Γ, a, τ), où a est une expression, τ un type, et Γ un *environnement de typage* qui associe un type $\Gamma(x)$ à tout identificateur x libre dans a. Un environnement peut être vu soit comme une fonction partielle des identificateurs vers les types, soit comme une liste d'associations d'un type à un identificateur. L'environnement vide $\emptyset$ est alors compris soit comme la fonction de domaine vide, soit comme la liste vide ; et l'extension $\Gamma; x : \tau$ est comprise soit comme la fonction qui à x associe τ et à toute autre variable y associe $\Gamma(y)$, soit simplement comme la liste obtenue en étendant la liste Γ à droite par l'association $x : \tau$. Dans le cas où l'environnement est vu comme une liste, par convention, les entrées les plus à droite sont les plus récentes, et ont priorité si un même identificateur apparaît plusieurs fois. Ainsi, les deux visions coïncident.

La relation de typage est notée $\Gamma \vdash a : \tau$ et se lit "dans l'environnement Γ, l'expression a a le type τ", ou encore "sous les hypothèses de typage sur les variables exprimées dans Γ, l'expression a a le type τ".

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

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

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash (a_1, a_2) : \tau_1 \times \tau_2}\ \text{(paire)} \qquad \frac{\Gamma \vdash a : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}\ a : \tau_1 \qquad \Gamma \vdash \texttt{snd}\ a : \tau_2}\ \text{(proj)}$$

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad \Gamma; x : \tau_1 \vdash a_2 : \tau_2}{\Gamma \vdash (\texttt{let}\ x = a_1\ \texttt{in}\ a_2) : \tau_2}\ \text{(let)}$$

Pour les règles (const) et (op), on se donne une fonction TC qui associe un type clos à chaque constante et à chaque opérateur ; ainsi, pour toute constante entière k, on a $TC(k) = \texttt{int}$; de plus, $TC(+) = \texttt{int} \times \texttt{int} \to \texttt{int}$. Notons qu'on ne peut pas définir $TC(\texttt{fix})$ de façon satisfaisante. En effet, pour tout type clos τ, le type $(\tau \to \tau) \to \tau$ conviendrait. Mais comment choisir τ ? Aucun choix n'est supérieur aux autres. Ce problème sera résolu plus loin lorsque nous introduirons les schémas de types.

La règle (var) s'appuie sur l'environnement pour déterminer le type de l'identificateur x. La règle (fun) peut se lire "si, sous l'hypothèse que x a le type τ_1, l'expression a a le type τ_2, alors la fonction $\texttt{fun}\ x \to a$ a le type $\tau_1 \to \tau_2$". Il s'agit d'une lecture de haut en bas. On peut également lire la règle de bas en haut : "pour déterminer le type de la fonction $\texttt{fun}\ x \to a$, ajoutez à l'environnement l'hypothèse que x a un certain type τ_1 ; déduisez que a admet un certain type τ_2 ; le type recherché est alors $\tau_1 \to \tau_2$". Il est courant qu'une règle d'inférence puisse ainsi se lire de deux façons légèrement différentes, l'une purement déductive, l'autre plus calculatoire. La règle (app) exige que le type de l'argument effectif coïncide avec le type attendu par la fonction ; cela est exprimé par l'occurrence de τ' dans chacune des deux prémisses. Les règles (paire) et (proj) concernent la construction et la destruction de paires ; on constate qu'elles *introduisent* et *éliminent* respectivement le constructeur de types $\times$. Enfin, la règle (let) ne présente pas, *dans ce système monomorphe*, d'intérêt particulier, comme le montre l'exercice suivant.

Exercice 1.12 *(*) Vérifier que les constructions* $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$ *et* $(\texttt{fun}\ x \to a_2)\ a_1$ *sont équivalentes du point de vue de l'évaluation – c'est-à-dire que l'une produit une valeur v si et seulement si l'autre produit également v – et du point de vue du typage monomorphe – c'est-à-dire que si l'une admet le type τ_2 sous l'environnement Γ, alors l'autre de même.*

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

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

Voici d'autres jugements de typage dérivables :

$$\begin{array}{l} \emptyset \vdash \mathtt{fun}\ x \to x : \alpha \to \alpha \\ \emptyset \vdash \mathtt{fun}\ x \to x : \mathtt{bool} \to \mathtt{bool} \end{array}$$

Voici des jugements non dérivables :

$$\begin{array}{l} \emptyset \vdash \mathtt{fun}\ x \to +(x,1) : \mathtt{int} \\ \emptyset \vdash \mathtt{fun}\ x \to +(x,1) : \alpha \to \mathtt{int} \end{array}$$

Enfin, voici quelques expressions non typables. Une expression a est non typable s'il n'existe pas Γ et τ tels que $\Gamma \vdash a : \tau$ soit dérivable.

$$\begin{array}{l} 1\ 2 \\ \mathtt{fun}\ f \to f\ f \\ \mathtt{let}\ f = \mathtt{fun}\ x \to x\ \mathtt{in}\ (f\ 1, f\ \mathtt{true}) \end{array}$$

Exercice 1.13 *(*) Expliquer pourquoi ces trois dernières expressions ne sont pas typables.*

1.5.3 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, le langage ML introduit une notion de *schéma de types*, une représentation compacte (et finie !) de tous les types qui peuvent être donnés à une expression polymorphe. Un schéma de types σ 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.

Notons que les schémas de types ne sont pas des types ; les deux notions restent distinctes. Cela signifie qu'on ne peut trouver aucun quantificateur universel dans un type ; par exemple, l'entité $(\forall \alpha . \alpha) \to \mathtt{int}$ n'existe pas en ML. On caractérise cette restriction en disant que ML n'autorise que la quantification universelle *prénexe*. D'autres systèmes de types adoptent une approche différente. Le principal exemple en est le système F, également connu sous le nom de "λ-calcul typé" ou encore de "λ-calcul du second ordre". Celui-ci enrichit le langage de types par la production $\tau ::= \ldots \mid \forall \alpha . \tau$, et n'introduit pas de notion de schéma de types. La quantification universelle peut donc être employée de façon plus flexible ; par exemple, $(\forall \alpha . \alpha) \to \mathtt{int}$ est un type du système F. La raison pour laquelle ML ne suit pas cette voie est pragmatique : l'inférence de types pour le système F est indécidable, tandis que l'inférence de types pour ML est décidable et réalisable de façon efficace.

Dans un schéma de types, les variables de types introduites par le quantificateur $\forall$ sont *liées*. Cela induit une notion d'α-conversion, similaire à celle que nous avons introduite pour les expressions ; deux schémas de types α-convertibles sont identifiés. Cela induit également la définition suivante de l'ensemble des variables libres dans un schéma de types :

$$\begin{array}{rcl}
\mathcal{L}(T) & = & \emptyset \\
\mathcal{L}(\alpha) & = & \{\alpha\} \\
\mathcal{L}(\tau_1 \to \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\
\mathcal{L}(\tau_1 \times \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\
\mathcal{L}(\forall \alpha_1, \ldots, \alpha_n. \tau) & = & \mathcal{L}(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\}
\end{array}$$

Rappelons que, les types restant des termes du premier ordre, toute variable qui apparaît dans un type τ y apparaît nécessairement libre ; c'est ce que formalisent les premières lignes de cette définition. La dernière ligne définit l'ensemble des variables de types libres dans un schéma de types.

Rappelons également que l'égalité de deux schémas modulo α-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 1.14 *(*)/(**) Montrer que $\mathcal{L}(\tau[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n]) = \mathcal{L}(\tau)[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n]$. En déduire que la définition de $\mathcal{L}(\sigma)$ est compatible avec l'α-conversion.*

Dans le système de types de ML, un environnement Γ associe à chaque identificateur non plus un type, mais un schéma de types. L'ensemble des variables de types libres dans Γ, noté $\mathcal{L}(\Gamma)$, est défini comme suit :

$$\mathcal{L}(\Gamma) = \bigcup_{x \in \mathrm{Dom}(\Gamma)} \mathcal{L}(\Gamma(x))$$

Intuitivement, un schéma de types représente l'ensemble des types obtenus en instanciant (c'est-à-dire en remplaçant) chaque variable universellement quantifiée par un type particulier arbitraire. 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 $\sigma \leq \tau$ (lire : le type τ est une instance du schéma de types σ) de la manière suivante :

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

Par exemple, $\mathtt{int} \to \mathtt{int}$ est une instance de $\forall \alpha. \alpha \to \alpha$, ainsi que $\mathtt{bool} \to \mathtt{bool}$, mais $\mathtt{int} \to \mathtt{bool}$ n'en est pas une. Notons que si σ est un schéma trivial de la forme $\forall. \tau'$, alors $\sigma \leq \tau$ est équivalent à $\tau = \tau'$.

1.5.4 Les règles de typage – typage polymorphe à la ML

Les règles de typage de mini-ML sont essentiellement les règles monomorphes de la section 1.5.2, avec les modifications suivantes :

- Comme indiqué plus haut, l'environnement de typage Γ associe des schémas de types, et non plus simplement des types, aux identificateurs.

– De même, les constantes et opérateurs se voient maintenant associer des schémas de types clos, et non plus simplement des types clos, par la fonction TC. Par exemple, on peut poser

$$\begin{aligned} TC(+) &= \mathtt{int} \times \mathtt{int} \to \mathtt{int} \\ TC(\mathtt{fix}) &= \forall\alpha.\,(\alpha \to \alpha) \to \alpha \end{aligned}$$

– La règle de typage des identificateurs, à savoir (var-inst), autorise le schéma fourni par l'environnement à être instancié de façon arbitraire. Ainsi, si l'environnement associe à x un schéma de types non trivial, chaque référence à x pourra recevoir un type différent, si besoin.
– enfin, la règle associée à la construction `let`, à savoir (let-gen), *généralise* le type de a_1, de façon à en faire un schéma de types, avant de l'introduire dans l'environnement.

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

$$\frac{\Gamma; x : \tau_1 \vdash a : \tau_2}{\Gamma \vdash (\mathtt{fun}\ x \to a) : \tau_1 \to \tau_2}\ \text{(fun)} \qquad \frac{\Gamma \vdash a_1 : \tau' \to \tau \quad \Gamma \vdash a_2 : \tau'}{\Gamma \vdash a_1\ a_2 : \tau}\ \text{(app)}$$

$$\frac{\Gamma \vdash a_1 : \tau_1 \quad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash (a_1, a_2) : \tau_1 \times \tau_2}\ \text{(paire)} \qquad \frac{\Gamma \vdash a : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{fst}\ a : \tau_1 \quad \Gamma \vdash \mathtt{snd}\ a : \tau_2}\ \text{(proj)}$$

$$\frac{\Gamma \vdash a_1 : \tau_1 \quad \Gamma; x : Gen(\tau_1, \Gamma) \vdash a_2 : \tau_2}{\Gamma \vdash (\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2) : \tau_2}\ \text{(let-gen)}$$

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

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

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

Exemple :

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

Exemple :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\alpha \le \alpha}{x : \alpha; z : \beta \vdash x : \alpha}}{x : \alpha \vdash \mathtt{fun}\ z \to x : \beta \to \alpha} \qquad \dfrac{\forall\beta.\beta \to \alpha \le \gamma \to \alpha}{x : \alpha; y : \forall\beta.\beta \to \alpha \vdash y : \gamma \to \alpha}}{x : \alpha \vdash \mathtt{let}\ y = \mathtt{fun}\ z \to x\ \mathtt{in}\ y : \gamma \to \alpha}}{\emptyset \vdash \mathtt{fun}\ x \to \mathtt{let}\ y = \mathtt{fun}\ z \to x\ \mathtt{in}\ y : \alpha \to \gamma \to \alpha}}{}$$

Exercice 1.15 *(*) Peut-on typer les expressions ci-dessous en mini-ML ? Avec quels types ?*

$$\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ f\ f$$
$$\texttt{fun}\ f \to f\ f$$

Exercice 1.16 *(**) Une définition plus simple de Gen serait* $Gen(\tau_1, \Gamma) = \forall \alpha_1, \ldots, \alpha_n. \tau_1$ *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* Γ*.) 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* $\leq$ *(**).*

1.5.5 Quelques propriétés des règles de typage

Nous donnons maintenant trois propriétés du prédicat de typage qui nous serviront par la suite. La première est que le typage est stable par substitution de variables de types : si on peut dériver un jugement non clos, c'est-à-dire contenant des variables de types libres, par exemple

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

alors on peut aussi dériver tous les jugements obtenus en remplaçant ces variables par des types arbitraires, par exemple

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

Pour démontrer cette propriété, on utilise entre autres le fait que les schémas $TC(c)$ et $TC(op)$ sont clos pour tous c et op.

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

Notons que cette propriété signifie, intuitivement, que toutes les variables libres dans un jugement de typage peuvent être vues comme quantifiées universellement en tête du jugement. En d'autres termes, le jugement

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

peut être lu informellement

$$\forall \alpha \quad (f : \alpha \to \alpha; x : \alpha \vdash f\ x : \alpha)$$

La seconde propriété est qu'un jugement reste vrai si dans l'environnement 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) *Si, pour tout identificateur libre dans a, les hypothèses* $\Gamma_1(x)$ *et* $\Gamma_2(x)$ *coïncident, alors* $\Gamma_1 \vdash a : \tau$ *est équivalent à* $\Gamma_2 \vdash a : \tau$*.*

La troisième propriété est que tous les jugements 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 $\sigma' \leq \sigma$, si toute instance de σ est aussi instance de σ'. On montre facilement que $\sigma' \leq \forall \alpha_1 \ldots \alpha_n . \tau$ (où les α_i sont choisies non libres dans σ') si et seulement si $\sigma' \leq \tau$.

Proposition 1.4 (Stabilité du typage par renforcement des hypothèses) *Si Γ et Γ' ont même domaine et si, pour tout $x \in \mathrm{Dom}(\Gamma)$, on a $\Gamma'(x) \leq \Gamma(x)$, alors $\Gamma \vdash a : \tau$ implique $\Gamma' \vdash a : \tau$.*

La preuve des propositions 1.2, 1.3 et 1.4 est facile dans le cas du système de types monomorphe de la section 1.5.2 (par récurrence sur les dérivations de typage), mais plus délicate dans le système de types de ML. La difficulté n'est pas fondamentale ; il s'agit simplement de respecter la liaison des variables de types et de mettre à profit l'α-convertibilité entre schémas de types. Il est malheureusement aisé de s'égarer dans les méandres de la preuve si on manque de familiarité avec ces notions.

Exercice 1.17 *(***) Prouver la proposition 1.2. (On commencera par définir précisément l'image $\varphi(\sigma)$ d'un schéma de types σ par une substitution φ.)*

Chapitre 2

Sûreté du typage et sémantique à réductions

Le but du typage statique est d'éliminer toute une classe de programmes absurdes, comme par exemple 1 2 ou $1 + ($**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 produit ni une fonction ni un opérateur (exemple : 1 2) ; projection d'une expression qui ne produit pas une paire (exemple : `fst` 1) ; application d'un opérateur primitif à un argument de nature inappropriée (exemple : $+$ (**fun** $x \to x$)).

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, $1 + ($**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 règle 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 $a \twoheadrightarrow v$. 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 $a \twoheadrightarrow v$.

Le problème de cette approche est qu'il y a une autre classe d'expressions qui ne s'évaluent pas en une valeur, mais pourtant ne déclenchent 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 ou, en d'autres termes, qu'il accepte tous les programmes qui terminent. Si le système rejetait de plus tous les programmes qui ne terminent pas, on aurait résolu le problème de l'arrêt, qui est connu pour être indécidable. Tout langage Turing-complet et décidable contient donc des programmes 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 $a \twoheadrightarrow v$.

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 (la sémantique de notre langage suppose en effet que nous sommes dotés d'entiers non bornés), 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 $a \twoheadrightarrow r$, 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.

Certaines règles existantes doivent être légèrement modifiées pour autoriser un résultat quelconque là où une valeur était exigée auparavant :

$$\frac{a_1 \twoheadrightarrow (\texttt{fun}\ x \to a) \qquad a_2 \twoheadrightarrow v_2 \qquad a[x \leftarrow v_2] \twoheadrightarrow r}{a_1\ a_2 \twoheadrightarrow r}\ (4) \qquad \frac{a_1 \twoheadrightarrow v_1 \qquad a_2[x \leftarrow v_1] \twoheadrightarrow r}{(\texttt{let}\ x = a_1\ \texttt{in}\ a_2) \twoheadrightarrow r}\ (7)$$

De plus, on doit ajouter de nouvelles règles d'évaluation pour détecter les erreurs et produire le résultat `err` :

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

$$\frac{a_1 \twoheadrightarrow + \qquad a_2 \twoheadrightarrow v_2 \qquad v_2 \text{ n'est pas une paire d'entiers}}{a_1\ a_2 \twoheadrightarrow \texttt{err}}\ (11)$$

$$\frac{a \twoheadrightarrow v \qquad v \text{ n'est pas une paire}}{\texttt{fst}\ a \twoheadrightarrow \texttt{err} \qquad \texttt{snd}\ a \twoheadrightarrow \texttt{err}}\ (12)$$

Enfin, on doit ajouter de nouvelles 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.

$$\frac{a_1 \twoheadrightarrow \texttt{err}}{a_1\ a_2 \twoheadrightarrow \texttt{err}}\ (13) \qquad \frac{a_1 \twoheadrightarrow v_1 \quad a_2 \twoheadrightarrow \texttt{err}}{a_1\ a_2 \twoheadrightarrow \texttt{err}}\ (14) \qquad \frac{a_1 \twoheadrightarrow \texttt{err}}{(a_1, a_2) \twoheadrightarrow \texttt{err}}\ (15)$$

$$\frac{a_1 \twoheadrightarrow v_1 \quad a_2 \twoheadrightarrow \texttt{err}}{(a_1, a_2) \twoheadrightarrow \texttt{err}}\ (16) \qquad \frac{a_1 \twoheadrightarrow \texttt{err}}{\texttt{let}\ x = a_1\ \texttt{in}\ a_2 \twoheadrightarrow \texttt{err}}\ (17)$$

On peut noter au passage un phénomène intéressant : notre sémantique, augmentée de ces nouvelles règles de propagation des erreurs, spécifie que les applications et les paires sont évaluées de gauche à droite. En d'autres termes, pour évaluer l'expression (a_1, a_2), on évalue d'abord a_1, puis a_2. Cela se voit du fait que les règles (15) et (16) ne sont pas symétriques ; la première spécifie que, si a_1 provoque une erreur, alors l'expression (a_1, a_2) également ; la seconde spécifie que, *si* a_1 *termine* et si a_2 provoque une erreur, alors l'expression (a_1, a_2) provoque une erreur. La sémantique donnée au chapitre précédent, qui était dénuée de notion d'erreur, ne permettait pas de faire cette distinction.

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

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

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

Cette propriété de sûreté est vraie, et se prouve par récurrence structurelle sur la dérivation de $a \twoheadrightarrow r$. Cependant, cette approche n'est pas entièrement satisfaisante, pour plusieurs raisons. Tout d'abord, il faut ajouter beaucoup de règles, et cela rend la sémantique du langage peu lisible et redondante. De plus, il y a un gros risque d'oublier d'ajouter certaines règles d'erreur. Par exemple, si nous oublions la règle 11, 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*) $a \twoheadrightarrow r$ 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 réductions, également 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. (On suppose toujours a et a' closes.) Pour évaluer complètement l'expression, il faut bien sûr enchaîner plusieurs étapes

$$a \to a_1 \to a_2 \to \ldots \to a_n \not\to$$

jusqu'à atteindre une expression *irréductible* a_n. Une expression a est irréductible s'il n'existe aucune expression a' telle que $a \to a'$; on écrit parfois $a \not\to$. Bien sûr, l'existence d'un tel a_n n'est pas garantie ; il se peut que la séquence de réductions issue de a soit infinie.

Pour définir la relation "se réduit en une étape" $\rightarrow$, on commence par se donner des axiomes pour la relation $\stackrel{\varepsilon}{\rightarrow}$ "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`

$$\begin{array}{rcll} (\texttt{fun}\ x \rightarrow a)\ v & \stackrel{\varepsilon}{\rightarrow} & a[x \leftarrow v] & (\beta) \\ (\texttt{let}\ x = v\ \texttt{in}\ a) & \stackrel{\varepsilon}{\rightarrow} & a[x \leftarrow v] & (\mathit{let}) \\ \texttt{fst}\ (v_1, v_2) & \stackrel{\varepsilon}{\rightarrow} & v_1 & (\mathit{fst}) \\ \texttt{snd}\ (v_1, v_2) & \stackrel{\varepsilon}{\rightarrow} & v_2 & (\mathit{snd}) \end{array}$$

Notons que, encore une fois, nous décrivons ici la stratégie d'*appel par valeur* employée par Standard ML et O'Caml. En effet, dans les règles (β) et (let), le paramètre effectif, par lequel le paramètre formel x est remplacé, doit être une valeur v, et ne peut être une expression arbitraire. (Notons que la définition des valeurs reste celle donnée au chapitre précédent.) La stratégie d'*appel par nom* autoriserait un paramètre effectif arbitraire. Dans les règles (fst) et (snd), on exige que les deux composantes de la paire soient des valeurs v_1 et v_2 ; la stratégie d'appel par nom se contenterait ici aussi d'expressions arbitraires, car pour cette stratégie, toute paire (a_1, a_2) est une valeur, même si ses composantes ne sont pas encore évaluées. Ainsi, ces stratégies diffèrent par les règles de réduction et par la définition de la notion de valeur. (Elles diffèrent également en ce qui concerne les contextes d'évaluation, définis plus bas. Nous ne donnerons pas plus de détails.)

Ensuite, on se donne des axiomes concernant la sémantique des opérateurs. Ces axiomes, parfois appelés δ-règles, dépendent bien sûr des opérateurs considérés. Voici les δ-règles pour `+` et `fix` :

$$\begin{array}{rcll} +\ (n_1, n_2) & \stackrel{\varepsilon}{\rightarrow} & n_1 + n_2 & (\delta_+) \\ \texttt{fix}\ (\texttt{fun}\ x \rightarrow a) & \stackrel{\varepsilon}{\rightarrow} & a[x \leftarrow \texttt{fix}\ (\texttt{fun}\ x \rightarrow a)] & (\delta_{\text{fix}}) \end{array}$$

Enfin, parce qu'on ne réduit pas toujours *en tête* de l'expression considérée, une dernière règle est nécessaire. 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 :

$$\frac{a \stackrel{\varepsilon}{\rightarrow} a'}{E[a] \rightarrow E[a']}\ (\text{contexte})$$

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

Contextes d'évaluation :

$E ::= [\,]$	évaluation en tête
$\mid E\ a$	évaluation à gauche d'une application
$\mid v\ E$	évaluation à droite d'une application
$\mid \texttt{let}\ x = E\ \texttt{in}\ a$	évaluation à gauche d'un `let`
$\mid (E, a)$	évaluation à gauche d'une paire
$\mid (v, E)$	évaluation à droite d'une paire
$\mid \texttt{fst}\ E$	évaluation de l'argument d'une projection
$\mid \texttt{snd}\ E$	

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 E ne sont pas des contextes arbitraires. Ainsi, le contexte $\texttt{fun}\ x \to [\,]$ n'est pas un contexte d'évaluation, ce qui indique qu'il n'est pas permis de réduire le corps d'une fonction. Par ailleurs, le contexte $(+(1, 2), [\,])$ n'est pas non plus un contexte d'évaluation, car le membre gauche de la paire n'est pas une valeur. En fait, la production $E ::= (v, E)$, indique qu'il n'est permis d'évaluer la seconde composante de la paire *qu'après* que la première composante a été réduite en une valeur. La définition des contextes d'évaluation permet donc de spécifier la stratégie d'évaluation. La définition ci-dessus correspond à la stratégie d'appel par valeur et précise de plus qu'applications et paires doivent être évaluées de gauche à droite.

Exemple Considérons l'expression $a = (+(1, 2), +(3, 4))$. Il n'y a qu'une manière de l'écrire sous la forme $a = E_1[a_1]$ afin d'appliquer la règle (contexte) : il faut prendre $E_1 = ([\,], +(3, 4))$ et $a_1 = +(1, 2)$. Comme $a_1 \stackrel{\varepsilon}{\to} 3$, on obtient $a \to E_1[3] = (3, +(3, 4))$. Cette dernière expression peut alors s'écrire sous la forme $E_2[a_2]$ avec $E_2 = (3, [\,])$ et $a_2 = +(3, 4)$. Appliquant la règle (contexte), on obtient le résultat $E_2[7] = (3, 7)$. 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 voulions commencer par évaluer la sous-expression droite $+(3, 4)$, il faudrait écrire $a = E_3[a_3]$ avec $E_3(+(1, 2), [\,])$ et $a_3 = +(3, 4)$, mais cela n'est pas possible car le contexte E_3 n'est pas un contexte d'évaluation.

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

Pour finir, on définit la relation $\to^\star$ (lire : "se réduit en zéro, une ou plusieurs étapes") comme la fermeture réflexive et transitive de $\to$. C'est-à-dire, $a \to^\star a'$ ssi $a = a'$ ou il existe $a_1, \ldots, a_n$ tels que $a \to a_1 \to \ldots \to a_n \to a'$. Notre définition de la sémantique à petits pas est maintenant complète.

Notons que toute valeur est irréductible, ce qui correspond bien à l'idée qu'une valeur est un résultat : c'est une expression qui ne demande plus aucun calcul. Cependant, toute expression irréductible n'est pas une valeur : par exemple, l'application 1 2. Les expressions irréductibles qui ne sont pas des valeurs sont dites *bloquées* et représentent de fait des erreurs. Il n'est donc pas nécessaire, dans une sémantique à petits pas, d'introduire une constante `err`.

Exercice de programmation 2.1 *(**) Implémenter une fonction* `décompose` *qui, étant donnée une expression close a : (i) si a est une valeur, soulève une exception ; (ii) dans le cas contraire, renvoie E et a_1 tels que $a = E[a_1]$ et tels que E soit maximal pour cette propriété.*

En Caml, on pourra représenter les contextes E comme les fonctions Caml $a \mapsto E[a]$, c.à.d. comme les fonctions Caml qui prennent l'expression a et renvoient l'expression $E[a]$ en résultat. On évite ainsi de définir séparément la grammaire des contextes, et de devoir écrire l'opération d'application d'un contexte à une expression.

Vérifier, par un raisonnement inductif, que si `décompose` *renvoie E et a_1, alors l'expression a_1 est de la forme $v_1\ v_2$ ou* `let` *$x = v$* `in` *a ou* `fst` *v ou* `snd` *v. Écrire alors une fonction* `réduit_en_tête` *qui, étant donnée une telle expression a_1, effectue une étape de réduction en tête $a_1 \to a_1'$ et renvoie* *le réduit a'_1, ou bien soulève une exception si a_1 est irréductible. (Noter que dans ce cas, a_1 n'étant pas une valeur, elle est bloquée.)*

En déduire une fonction `évalue` *qui, étant donnée une expression a, en renvoie la valeur, ou bien soulève une exception si a se réduit en une expression bloquée.*

Tester votre code sur les exemples vus jusqu'ici.

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

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 :

$$a \rightarrow a_1 \rightarrow \ldots \rightarrow a_n \rightarrow v$$

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

2. a se réduit à l'infini

$$a \rightarrow a_1 \rightarrow \ldots \rightarrow a_n \rightarrow \ldots$$

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

3. a se réduit en une expression bloquée :

$$a \rightarrow a_1 \rightarrow \ldots \rightarrow a_n \not\rightarrow$$

   Ici, le calcul provoque une erreur d'exécution.

Nous allons prouver que si l'expression a est bien typée, le cas (3) ne peut pas se produire. C'est là l'objet de la suite de ce chapitre.

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

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

Autrement dit, tous les typages possibles pour a_1 doivent être des typages possibles pour a_2. Cette relation jouit de la propriété suivante, commune à la plupart des systèmes de types, et qui provient simplement du fait que pour typer une expression, on ne s'intéresse qu'aux *types* de ses sous-expressions, et non à leur *forme* :

Proposition 2.1 (Congruence de $\sqsubseteq$) *Pour tout contexte C, $a_1 \sqsubseteq a_2$ implique $C[a_1] \sqsubseteq C[a_2]$.*

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

Cas de base $C = [\,]$. Immédiat.

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

$$\frac{\Gamma \vdash C'[a_1] : \tau' \to \tau \qquad \Gamma \vdash a : \tau'}{\Gamma \vdash C'[a_1]\ a : \tau}$$

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

$$\frac{\Gamma \vdash C'[a_2] : \tau' \to \tau \qquad \Gamma \vdash a : \tau'}{\Gamma \vdash C'[a_2]\ a : \tau}$$

Les autres cas se traitent de façon analogue. □

2.3.2 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*

$$\begin{array}{rcl} \Gamma & \vdash & a' : \tau' \\ \Gamma; x : \forall \alpha_1 \ldots \alpha_n.\tau' & \vdash & a : \tau \end{array}$$

avec $\alpha_1, \ldots, \alpha_n$ *non libres dans* Γ. *Alors,*

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

Démonstration: par induction sur la structure de l'expression a. On écrit Γ_x pour $\Gamma; x : \forall \alpha_1 \ldots \alpha_n.\tau_1$.

Cas a est x. Nous avons alors $a[x \leftarrow a'] = a'$. Par hypothèse, $\Gamma_x \vdash x : \tau$, donc $\forall \alpha_1 \ldots \alpha_n.\tau' \leq \tau$. 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 $\Gamma \vdash a' : \tau'$ et à la substitution φ, il vient $\varphi(\Gamma) \vdash a' : \varphi(\tau')$, c'est-à-dire $\Gamma \vdash a' : \varphi(\tau')$ puisque les α_i ne sont pas libres dans Γ. Nous avons donc montré $\Gamma \vdash a' : \tau$; c'est le résultat attendu.

Cas a est une variable $y \neq x$ ou une constante c ou un opérateur op. Alors, x n'est pas libre dans a, donc $a[x \leftarrow a'] = a$. De plus, $\Gamma_x \vdash a : \tau$ implique $\Gamma \vdash a : \tau$ par la proposition 1.3 (indifférence du typage vis-à-vis des hypothèses inutiles).

Cas $a = \texttt{fun}\ y \to a_1$. On suppose, sans perte de généralité, que $y \neq x$. En effet, on peut toujours obtenir cela par α-conversion, c'est-à-dire en renommant y si besoin.

On peut de plus supposer, toujours sans perte de généralité, que les α_i n'apparaissent pas dans τ. En effet, soit θ un renommage associant une variable "fraîche", c'est-à-dire non libre dans Γ ou dans τ, à chaque α_i. Alors la proposition 1.2, appliquée à la première hypothèse, donne $\theta(\Gamma) \vdash a' : \theta(\tau')$, c'est-à-dire $\Gamma \vdash a' : \theta(\tau')$, puisque les α_i ne sont pas libres dans Γ. De plus, la seconde hypothèse, par α-conversion, peut être lue $\Gamma; x : \forall \theta(\alpha_1) \ldots \theta(\alpha_n).\theta(\tau') \vdash a : \tau$. Nous sommes ainsi ramenés à la situation initiale, avec l'hypothèse supplémentaire souhaitée.

Comme $\Gamma_x \vdash a : \tau$, nous avons $\tau = \tau_2 \to \tau_1$ et

$$\frac{\Gamma_x; y : \tau_2 \vdash a_1 : \tau_1}{\Gamma_x \vdash \texttt{fun}\ y \to a_1 : \tau_2 \to \tau_1}$$

Les α_i ne sont libres ni dans Γ ni dans τ, donc ne sont pas libres dans $\Gamma; y : \tau_2$. Nous pouvons donc appliquer l'hypothèse de récurrence à la prémisse, ce qui donne $\Gamma; y : \tau_2 \vdash a_1[x \leftarrow a'] : \tau_1$. Il en découle $\Gamma \vdash \texttt{fun}\ y \to a_1[x \leftarrow a'] : \tau_2 \to \tau_1$, c'est-à-dire $\Gamma \vdash a[x \leftarrow a'] : \tau$.

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

$$\frac{\Gamma_x \vdash a_1 : \tau_2 \to \tau \qquad \Gamma_x \vdash a_2 : \tau_2}{\Gamma_x \vdash a_1\ a_2 : \tau}$$

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

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

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

Cas $a = \texttt{let}\ y = a_1\ \texttt{in}\ a_2$. On suppose, sans perte de généralité, que $y \neq x$ et que y n'apparaît pas libre dans a'. En effet, on peut toujours obtenir cela par α-conversion, c'est-à-dire en renommant y si besoin. Par hypothèse $\Gamma_x \vdash a : \tau$, nous avons

$$\frac{\Gamma_x \vdash a_1 : \tau_1 \qquad \Gamma_x; y : \mathit{Gen}(\tau_1, \Gamma_x) \vdash a_2 : \tau}{\Gamma_x \vdash \texttt{let}\ y = a_1\ \texttt{in}\ a_2 : \tau}$$

Appliquant l'hypothèse de récurrence à la première prémisse, il vient $\Gamma \vdash a_1[x \leftarrow a'] : \tau_1$. Par ailleurs, les α_i n'apparaissent pas libres dans $\mathit{Gen}(\tau_1, \Gamma_x)$, sans quoi ils apparaîtraient libres dans Γ_x, donc dans Γ. De plus, comme y n'apparaît pas libre dans a', la proposition 1.3 donne $\Gamma; y : \mathit{Gen}(\tau_1, \Gamma_x) \vdash a' : \tau'$. Ces deux remarques permettent d'appliquer l'hypothèse de récurrence à la seconde prémisse, ce qui donne $\Gamma; y : \mathit{Gen}(\tau_1, \Gamma_x) \vdash a_2[x \leftarrow a'] : \tau$. Parce que toute variable libre dans Γ l'est également dans Γ_x, nous avons $\mathit{Gen}(\tau_1, \Gamma) \leq \mathit{Gen}(\tau_1, \Gamma_x)$, d'où, par la proposition 1.4, il s'ensuit $\Gamma; y : \mathit{Gen}(\tau_1, \Gamma)\} \vdash a_2[x \leftarrow a'] : \tau$. Nous pouvons maintenant dériver le résultat attendu :

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

Les cas restants sont analogues à celui de l'application. □

2.3.3 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 $\texttt{cast}\ v \stackrel{\varepsilon}{\to} v$.) Nous faisons donc les hypothèses suivantes :

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

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

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

L'hyopthèse H0 dit que les opérateurs et les constantes ont des types “raisonnables” ; par exemple, aucune constante ne prétend être une fonction ou une paire. L'hypothèse H1 exprime que les δ-règles doivent préserver le typage. L'hypothèse H2 dit qu'il y a suffisamment de δ-règles pour réduire toutes les applications bien typées d'opérateurs à une valeur, ou en d'autres terms, qu'aucune application bien typée d'un opérateur n'est bloquée.

2.3.4 Préservation du typage par réduction

Grâce au lemme de substitution établi ci-dessus, il est à présent facile de prouver que la réduction de tête 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). Nous avons donc $a = (\texttt{fun}\ x \to a_1)\ v$ et $a' = a_1[x \leftarrow v]$. Soient Γ et τ tels que $\Gamma \vdash a : \tau$. La dérivation de cet énoncé est nécessairement de la forme

$$\dfrac{\dfrac{\Gamma; x : \tau' \vdash a_1 : \tau}{\Gamma \vdash (\texttt{fun}\ x \to a_1) : \tau' \to \tau} \qquad \Gamma \vdash v : \tau'}{\Gamma \vdash (\texttt{fun}\ x \to a_1)\ v : \tau}$$

Nous avons donc $\Gamma; x : \tau' \vdash a_1 : \tau$ et $\Gamma \vdash v : \tau'$. Les hypothèses du lemme de substitution (proposition 2.2) sont vérifiées, dans le cas simple où le type de la variable substituée est monomorphe, i.e. $n = 0$. Par conséquent, on a $\Gamma \vdash a_1[x \leftarrow v] : \tau$, c'est-à-dire $\Gamma \vdash a' : \tau$. Ceci valant pour tout Γ et tout τ tel que $\Gamma \vdash a : \tau$, nous avons bien montré $a \sqsubseteq a'$.

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

$$\dfrac{\Gamma \vdash v : \tau' \qquad \Gamma; x : \mathit{Gen}(\tau', \Gamma) \vdash a_1 : \tau}{\Gamma \vdash (\texttt{let}\ x = v\ \texttt{in}\ a_1) : \tau}$$

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

Cas de la règle (*fst*). Ici, $a = \texttt{fst}\ (v_1, v_2)$ et $a' = v_1$. Soient Γ et τ_1 tels que $\Gamma \vdash a : \tau_1$. La dérivation de cet énoncé est de la forme

$$\dfrac{\dfrac{\Gamma \vdash v_1 : \tau_1 \qquad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash (v_1, v_2) : \tau_1 \times \tau_2}}{\Gamma \vdash \texttt{fst}\ (v_1, v_2) : \tau_1}$$

d'où on extrait $\Gamma \vdash v_1 : \tau_1$, qui est le résultat attendu. □

Nous pouvons immédiatement en déduire que la réduction préserve le typage. Ce résultat est couramment appelé *subject reduction* dans la littérature.

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

Démonstration: L'énoncé $a \to a'$ est nécessairement obtenu par une instance de la règle (contexte) :

$$\dfrac{a_1 \xrightarrow{\varepsilon} a'_1}{E[a_1] \to E[a'_1]}$$

D'après la proposition 2.3, $a_1 \sqsubseteq a'_1$. Donc, par congruence de $\sqsubseteq$ (proposition 2.1), on a $E[a_1] \sqsubseteq E[a'_1]$. C'est le résultat attendu. □

2.3.5 Un terme clos bien typé ne peut être bloqué

Le lemme suivant garantit qu'un terme clos bien typé ne peut être bloqué : soit c'est une valeur, soit il est réductible.

Proposition 2.5 (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 induction sur la structure de a.

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

Cas $a = c$ ou $a = op$ ou $a = \texttt{fun}\ x \to a'$. 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 peut se réduire, alors, d'après la règle (contexte), $a_1\ a_2$ également. Si au contraire a_1 est irréductible, alors l'hypothèse de récurrence, appliquée à la première prémisse, indique que a_1 est une valeur.

De même, si a_2 peut se réduire, alors, d'après la règle (contexte), $a_1\ a_2$ également, puisque a_1 est une valeur. Si au contraire a_2 est irréductible, alors l'hypothèse de récurrence, appliquée à la seconde prémisse, indique que a_2 est une valeur.

Par examen des règles de typage et grâce à l'hypothèse H0, une valeur close de type $\tau' \to \tau$ – telle a_1 – est nécessairement une fonction $\texttt{fun}\ x \to a_3$ ou un opérateur op.

Dans le premier cas, $a_1\ a_2$ se réduit par la règle (β). Dans le second cas, d'après l'hypothèse H2, $a_1\ a_2$ se réduit par une δ-règle.

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 peut se réduire, alors, d'après la règle (contexte), a de même. Si au contraire a_1 est irréductible, alors l'hypothèse de récurrence, appliquée à la première prémisse, indique que a_1 est une valeur. Alors, a se réduit par la règle (let).

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 peut se réduire, alors a de même ; sinon, par hypothèse de récurrence, a_1 est une valeur. Dès lors, si a_2 peut se réduire, alors a de même ; sinon, par hypothèse de récurrence, a_2 est une valeur. Enfin, a_1 et a_2 étant des valeurs, la paire (a_1, a_2) est une valeur également.

Cas $a = \texttt{fst}\ a'$. Si a_1 peut se réduire, alors, d'après la règle (contexte), a de même. Si au contraire a' est irréductible, alors l'hypothèse de récurrence, appliquée à la première prémisse, indique que a' est une valeur.

Par examen des règles de typage et grâce à l'hypothèse H0, une valeur close de type $\tau_1 \times \tau_2$ – telle a' – est nécessairement une paire (v_1, v_2). Il en résulte que a peut se réduire par (fst). □

2.3.6 Pour finir

Nous allons maintenant vérifier que la sûreté du typage découle des propriétés de préservation du typage et de progression établies précédemment.

Théorème 2.1 (Sûreté du typage) *Si* $\emptyset \vdash a : \tau$ *et* $a \to^\star a' \not\to$*, alors* a' *est une valeur.*

Démonstration: Le typage étant préservé par réduction (proposition 2.4), nous avons $\emptyset \vdash a' : \tau$. Puisque a' est irréductible, la propriété de progression (proposition 2.5) garantit alors que a' est une valeur. □

Nous avons donc établi la sûreté du typage 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* `+` *et* `fix`*.*

Chapitre 3

Inférence de types

3.1 Introduction à l'inférence de types

Pour un langage statiquement typé, un *typeur* est un algorithme qui, étant donné un programme, détermine si celui-ci est typable et, si oui, en produit un type. Si le programme admet plusieurs types, un typeur devra produire un *type principal*, c'est-à-dire un type plus général que tous les autres types possibles (pour une notion de "plus général" qui dépend du système de types considéré). Par exemple, en mini-ML, **fun** $x \rightarrow x$ a les types $\tau \rightarrow \tau$ pour n'importe quel type τ, mais le type $\alpha \rightarrow \alpha$ est principal, puisque tous les autres types possibles s'en déduisent par substitution.

Bien sûr, la tâche du typeur est facilitée si la définition du langage exige que certaines informations de typage soient fournies par le programmeur lui-même, à l'aide d'*annotations* de types. On découvre en fait un spectre de possibilités, selon la quantité d'annotations que l'on exige. Nous en décrivons quelques-unes ci-dessous.

Vérification pure : Dans ce cas de figure, toutes les sous-expressions du programme, ainsi que tous les identificateurs, doivent être annotés par leur type. Voici un exemple de programme acceptable :

```
fun (x : int) → (
  let y : int = (+ : int×int→int) ((x : int, 1 : int) : int×int) in
  y : int
) : int
```

Le typeur est alors très simple, puisque le programmeur n'écrit en fait plus uniquement une expression, mais une dérivation de typage complète. Bien sûr, un tel langage est inutilisable en pratique ; aucun langage réaliste n'adopte cette approche extrême.

Déclaration des types des identificateurs et propagation des types : Dans ce cas de figure, le programmeur doit déclarer les types des paramètres de fonctions et des variables locales. Le typeur infère alors aisément le type de chaque sous-expression à partir des types de ses sous-expressions. Autrement dit, l'information de typage est propagée à travers l'expression, des feuilles vers la racine. Par exemple, sachant que `x` est de type `int`, le typeur peut non seulement vérifier que l'expression `x+1` est bien typée, mais aussi inférer qu'elle a le type `int`. L'exemple ci-dessus devient :

```
fun (x : int) → let y : int = + (x, 1) in y
```

Le typeur infère le type `int → int` pour cette expression. Une approche similaire à celle-ci est adoptée par la plupart des langages impératifs : Pascal, C, Java, etc.

Déclaration des types des paramètres de fonction et propagation des types : Dans ce cas de figure, le programmeur doit déclarer les types des paramètres de fonctions. La différence par rapport au cas précédent est donc que les variables locales (liées par `let`) ne sont plus obligatoirement annotées lorsqu'elles sont introduites. Le typeur en détermine alors le type en se basant sur le type de l'expression à laquelle elles sont associées. Notre exemple devient :

```
fun (x : int) → let y = + (x, 1) in y
```

Ayant déterminé que `+ (x, 1)` est de type `int`, le typeur associe à `y` le type `int`.

Inférence complète de types : Dans ce cas de figure, plus aucune annotation n'est obligatoire. Le typeur détermine le type des paramètres de fonctions d'après l'utilisation qui en est faite par la fonction. Notre exemple devient :

```
fun x → let y = + (x, 1) in y
```

Puisque l'addition `+` n'opère que sur des paires d'entiers, `x` est nécessairement de type `int`. Cet intéressant processus de déduction, ou *inférence*, est celui utilisé par les langages de la famille ML.

Dans ce qui suit, nous allons nous concentrer sur la dernière approche, à savoir l'inférence complète, car les autres sont techniquement très simples. Toutefois, pour se familiariser d'abord avec l'avant-dernière approche, on pourra traiter l'exercice suivant.

Exercice de programmation 3.1 *Écrire un typeur pour mini-ML avec typage monomorphe (section 1.5.2) et où les paramètres de fonctions sont obligatoirement annotés par leur type :*

Expressions : $a \ ::= \ldots \mid \texttt{fun}\ (x : \tau) \to a$

Il s'agit donc de la troisième approche dans la liste ci-dessus. On écrira le typeur sous la forme d'une fonction d'une expression a et d'un environnement de typage Γ, laquelle renvoie le type τ de a dans l'environnement Γ s'il existe et échoue dans le cas contraire.

Si certaines annotations de types dans le programme a contiennent des variables de types, de quelle façon celles-ci se comportent-elles ? Quel problème se pose si l'on veut introduire le polymorphisme de ML, tout en conservant la restriction que les paramètres de fonctions doivent être annotés ?

3.2 Inférence de types pour mini-ML avec typage monomorphe

Dans cette section, nous considérons le problème de l'inférence de types pour mini-ML muni du système de typage monomorphe de la section 1.5.2, c'est-à-dire le λ-calcul simplement typé. Nous allons procéder en deux temps :

1. À partir du programme source, on construit un système d'équations entre types qui caractérise tous les typages possibles pour ce programme.
2. On résout ensuite ce système d'équations. S'il n'y a pas de solution, le programme est mal typé. Sinon, on détermine une solution *principale* du système d'équations ; cela nous donne un typage *principal* du programme.

En combinant ces deux phases, on obtient un algorithme qui détermine si un programme est typable et si oui, en fournit un typage principal.

Pourquoi traiter explicitement le cas du λ-calcul simplement typé, alors que nous intéresserons à mini-ML, qui l'englobe, dans la suite ? D'abord et surtout, ce cas particulier est beaucoup plus simple que le cas général, et mérite donc d'être bien compris avant que l'on puisse s'attaquer à mini-ML au complet. Par ailleurs et accessoirement, dans le cas du λ-calcul simplement typé, l'algorithme calcule, à partir d'une expression a, un environnement Γ et un type τ principaux tels que $\Gamma \vdash a : \tau$; tandis que, dans le cas de mini-ML, l'algorithme calcule, à partir d'un environnement Γ et d'une expression a, un type τ principal tel que $\Gamma \vdash a : \tau$. En termes techniques, on dit que le λ-calcul admet des *typages* principaux—le mot "typage" désigne alors une paire (Γ, τ)—tandis que mini-ML admet seulement des *types* principaux.

3.2.1 Construction du système d'équations

Considérons un programme (c'est-à-dire une expression) a_0 fixée. On souhaite associer à chaque identificateur qui apparaît dans a, qu'il soit libre ou interne, une variable de types. Pour cela, nous allons supposer que tous les identificateurs qui apparaissent dans a, libres ou internes, sont distincts, ce que l'on peut toujours obtenir par α-conversion. Nous pouvons alors, à chaque identificateur x, associer une variable de types distincte α_x. De plus, à chaque sous-expression a de a_0, nous associerons une variable de types distincte α_a. (Notons que, si une sous-expression a apparaît plusieurs fois dans a_0, il faut associer à ses différentes occurrences des variables de types distinctes ; la notation α_a est donc ambiguë, mais cela ne nous gênera pas dans cette présentation légèrement informelle.) De plus, pour chaque expression a de la forme `fst` b ou `snd` b, nous aurons besoin d'une variable de types distincte β_a. Le système d'équations $C(a)$ associé à chaque sous-expression a est alors défini par induction sur la structure :

- Si a est une variable x, alors $C(a) = \{\alpha_a \stackrel{?}{=} \alpha_x\}$.
- Si a est une constante c, alors $C(a) = \{\alpha_a \stackrel{?}{=} TC(c)\}$.
- Si a est un opérateur op, alors $C(a) = \{\alpha_a \stackrel{?}{=} TC(op)\}$.
- Si a est une fonction $\texttt{fun}\ x \to b$, alors $C(a) = \{\alpha_a \stackrel{?}{=} \alpha_x \to \alpha_b\} \cup C(b)$.
- Si a est une application $b\ c$, alors $C(a) = \{\alpha_b \stackrel{?}{=} \alpha_c \to \alpha_a\} \cup C(b) \cup C(c)$.
- Si a est une paire (b, c), alors $C(a) = \{\alpha_a \stackrel{?}{=} \alpha_b \times \alpha_c\} \cup C(b) \cup C(c)$.
- Si a est une projection $\texttt{fst}\ b$, alors $C(a) = \{\alpha_a \times \beta_a \stackrel{?}{=} \alpha_b\} \cup C(b)$.
- Si a est une projection $\texttt{snd}\ b$, alors $C(a) = \{\beta_a \times \alpha_a \stackrel{?}{=} \alpha_b\} \cup C(b)$.
- Si a est une définition $\texttt{let}\ x = b\ \texttt{in}\ c$, alors $C(a) = \{\alpha_x \stackrel{?}{=} \alpha_b;\ \alpha_a \stackrel{?}{=} \alpha_c\} \cup C(b) \cup C(c)$.

Exemple : on considère le programme

$$a = \underbrace{(\texttt{fun}\ x \to \underbrace{\texttt{fun}\ y \to \underbrace{1}_{d}}_{c})}_{b}\ \underbrace{\texttt{true}}_{e}$$

On a :

$$\begin{aligned} C(a) = \{\ & \alpha_b \stackrel{?}{=} \alpha_e \to \alpha_a; \\ & \alpha_b \stackrel{?}{=} \alpha_x \to \alpha_c; \\ & \alpha_c \stackrel{?}{=} \alpha_y \to \alpha_d; \\ & \alpha_d \stackrel{?}{=} \texttt{int}; \\ & \alpha_e \stackrel{?}{=} \texttt{bool}\} \end{aligned}$$

Exercice de programmation 3.2 *Écrire une fonction qui prend une expression a en entrée et construit son ensemble d'équations $C(a)$. Pour associer les variables α_b, α_x aux sous-expressions b et aux identificateurs x, on pourra ou bien utiliser une table d'association globale (table de hachage ou autre), ou bien écrire une première passe sur a qui annote les identificateurs et les sous-expressions par des variables de types.*

3.2.2 Lien entre typages et solutions des équations

Une *solution* de l'ensemble d'équations $C(a)$ est une substitution φ telle que pour toute équation $\tau_1 \stackrel{?}{=} \tau_2$ dans $C(a)$, on ait $\varphi(\tau_1) = \varphi(\tau_2)$. Autrement dit, une solution est un *unificateur* du système d'équations $C(a)$.

Les deux propositions suivantes montrent que les solutions de $C(a)$ caractérisent exactement les typages de a.

Proposition 3.1 (Correction des équations) *Si φ est une solution de $C(a)$, alors $\Gamma \vdash a : \varphi(\alpha_a)$ où Γ est l'environnement de typage $\{x : \varphi(\alpha_x) \mid x \in \mathcal{L}(a)\}$.*

Démonstration: par récurrence structurelle sur a. Les cas de base $a = x$, $a = c$ et $a = op$ sont immédiats.

Pour le cas $a = \texttt{fun}\ x \to b$, on a $C(a) = \{\alpha_a \stackrel{?}{=} \alpha_x \to \alpha_b\} \cup C(b)$. Comme φ est une solution de $C(a)$, c'est aussi une solution de $C(b)$, et de plus $\varphi(\alpha_a) = \varphi(\alpha_x) \to \varphi(\alpha_b)$. Appliquant l'hypothèse de récurrence à b, il vient $\Gamma; x : \varphi(\alpha_x) \vdash b : \varphi(\alpha_b)$. On peut donc construire la dérivation suivante :

$$\frac{\Gamma; x : \varphi(\alpha_x) \vdash b : \varphi(\alpha_b)}{\Gamma \vdash \texttt{fun}\ x \to b : \varphi(\alpha_x) \to \varphi(\alpha_b)}$$

c'est-à-dire $\Gamma \vdash a : \varphi(\alpha_a)$, comme attendu.

Pour le cas $a = b\ c$, on a $C(a) = \{\alpha_b \stackrel{?}{=} \alpha_c \to \alpha_a\} \cup C(b) \cup C(c)$. Donc, φ est une solution de $C(b)$ et de $C(c)$. Appliquant deux fois l'hypothèse de récurrence, il vient $\Gamma \vdash b : \varphi(\alpha_b)$ et $\Gamma \vdash c : \varphi(\alpha_c)$. Comme $\varphi(\alpha_b) = \varphi(\alpha_c) \to \varphi(\alpha_a)$, on a la dérivation suivante :

$$\frac{\Gamma \vdash b : \varphi(\alpha_c) \to \varphi(\alpha_a) \qquad \Gamma \vdash c : \varphi(\alpha_c)}{\Gamma \vdash b\ c : \varphi(\alpha_a)}$$

C'est le résultat attendu. Les cas restants sont tout aussi simples. □

Proposition 3.2 (Complétude des équations) *Soit a une expression. S'il existe un environnement Γ et un type τ tels que $\Gamma \vdash a : \tau$, alors le système d'équations $C(a)$ admet une solution φ telle que $\varphi(\alpha_a) = \tau$ et $\varphi(\alpha_x) = \Gamma(x)$ pour tout $x \in \mathcal{L}(a)$.*

Démonstration: On construit une solution φ par récurrence sur la dérivation de $\Gamma \vdash a : \tau$ et par cas sur la dernière règle de typage utilisée. On fera en sorte que le domaine de φ soit exactement l'ensemble des variables de types apparaissant dans le système $C(a)$. On donne deux cas représentatifs de la preuve ; les autres sont similaires et laissés au lecteur.

Cas la dérivation se termine par la règle (fun).

$$\frac{\Gamma; x : \tau_1 \vdash b : \tau_2}{\Gamma \vdash \texttt{fun}\ x \to b : \tau_1 \to \tau_2}$$

On a donc $a = \texttt{fun}\, x \to b$ et $\tau = \tau_1 \to \tau_2$. Par application de l'hypothèse de récurrence, il existe une solution φ' de $C(b)$ vérifiant $\varphi'(\alpha_b) = \tau_2$ et $\varphi'(\alpha_x) = \tau_1$ et $\varphi'(\alpha_y) = \Gamma(y)$ pour tout $y \in \mathcal{L}(b)$. Définissons alors $\varphi = \varphi'\{\alpha_a \mapsto \tau\}$. Il est facile de voir que φ est une solution de $C(a) = C(b) \cup \{\alpha_a \stackrel{?}{=} \alpha_x \to \alpha_b\}$. En effet,

$$\varphi(\alpha_a) = \tau = \tau_1 \to \tau_2 = \varphi(\alpha_x) \to \varphi(\alpha_b)$$

et d'autre part φ est solution de $C(b)$ puisque φ prolonge φ'.

Cas la dérivation se termine par la règle (app).

$$\frac{\Gamma \vdash b : \tau' \to \tau \qquad \Gamma \vdash c : \tau'}{\Gamma \vdash b\ c : \tau}$$

On a donc $a = b\ c$. En appliquant deux fois l'hypothèse de récurrence à b et c, il vient des solutions φ_1 et φ_2 de $C(b)$ et $C(c)$ qui satisfont (en particulier) $\varphi_1(\alpha_x) = \varphi_2(\alpha_x)$ pour tout $x \in \mathcal{L}(a_1) \cap \mathcal{L}(a_2)$, et ces variables α_x sont les seules communes aux domaines de φ_1 et φ_2. Définissons donc $\varphi = \varphi_1 \cup \varphi_2 \cup \{\alpha_a \mapsto \tau\}$. Cette substitution prolonge φ_1 et φ_2, donc est solution de $C(b)$ et $C(c)$. Enfin,

$$\varphi(\alpha_b) = \varphi_1(\alpha_b) = \tau' \to \tau = \varphi_2(\alpha_c) \to \varphi(\alpha_a) = \varphi(\alpha_c) \to \varphi(\alpha_a).$$

Donc, φ est solution de $C(a)$. □

3.2.3 Résolution des équations

L'ensemble $C(a)$ peut être vu comme un problème d'unification du premier ordre : il s'agit en effet d'un ensemble d'équations entre types, qui sont des termes du premier ordre, éléments de l'algèbre libre engendrée par les variables, les types de base, et les constructeurs $\to$ et $\times$. (On entend par termes *du premier ordre* des termes ne contenant pas de lieurs.) L'unification du premier ordre dans une théorie vide (c'est-à-dire dans une algèbre libre) est *décidable*, c'est-à-dire qu'il existe un algorithme capable de déterminer si un problème est soluble ou non, et *unitaire*, c'est-à-dire que tout problème soluble admet une solution *principale*. Une solution φ de C est dite *principale* si et seulement si toute solution ψ de C vérifie $\varphi \leq \psi$, où l'ordre $\leq$ est défini comme suit : $\varphi \leq \psi$ est vérifié si et seulement s'il existe une substitution θ telle que $\psi = \theta \circ \varphi$.

Il existe donc un algorithme `mgu` qui, étant donné un problème C, soit échoue, ce qui signifie que C n'a pas de solution, soit produit une solution principale de C. (On se réferera au cours de Delia Kesner et Jean-Pierre Jouannaud pour plus de détails.) En voici une définition possible, due à Robinson.

$$\begin{array}{rcl}
\texttt{mgu}(\emptyset) & = & id \\
\texttt{mgu}(\{\alpha \stackrel{?}{=} \alpha\} \cup C) & = & \texttt{mgu}(C) \\
\texttt{mgu}(\{\alpha \stackrel{?}{=} \tau\} \cup C) & = & \texttt{mgu}(C[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau] \text{ si } \alpha \text{ n'est pas libre dans } \tau \\
\texttt{mgu}(\{\tau \stackrel{?}{=} \alpha\} \cup C) & = & \texttt{mgu}(C[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau] \text{ si } \alpha \text{ n'est pas libre dans } \tau \\
\texttt{mgu}(\{\tau_1 \to \tau_2 \stackrel{?}{=} \tau_1' \to \tau_2'\} \cup C) & = & \texttt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_1'; \tau_2 \stackrel{?}{=} \tau_2'\} \cup C) \\
\texttt{mgu}(\{\tau_1 \times \tau_2 \stackrel{?}{=} \tau_1' \times \tau_2'\} \cup C) & = & \texttt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_1'; \tau_2 \stackrel{?}{=} \tau_2'\} \cup C)
\end{array}$$

Dans tous les autres cas, `mgu`(C) échoue. Les principales propriétés de l'algorithme, que nous ne démontrons pas ici, sont les suivantes :

Proposition 3.3 (Propriétés de mgu) *L'algorithme* `mgu` *jouit des propriétés suivantes :*

1. *Correction : si* $\texttt{mgu}(C) = \varphi$, *alors* φ *est solution de* C.
2. *Complétude : si* C *admet une solution* ψ, *alors* $\texttt{mgu}(C)$ *réussit et produit une solution* φ *telle que* $\varphi \leq \psi$.

Exemple : on considère le jeu d'équations obtenus dans l'exemple de la section 3.2.1. La solution principale est

$$\begin{array}{ll} \alpha_x \leftarrow \texttt{bool} & \alpha_e \leftarrow \texttt{bool} \\ \alpha_a \leftarrow \alpha_y \rightarrow \texttt{int} & \alpha_c \leftarrow \alpha_y \rightarrow \texttt{int} \\ \alpha_d \leftarrow \texttt{int} & \end{array}$$

Les autres solutions s'en déduisent en remplaçant α_y par un type quelconque.

On dit qu'une variable α est *hors de portée* d'une substitution φ si $\varphi(\alpha) = \alpha$ et pour tout $\beta \neq \alpha$, α n'est pas libre dans $\varphi(\beta)$. On notera que cette notion est préservée par composition. On dit qu'un ensemble de variables V est hors de portée de φ si chacun de ses éléments l'est.

Proposition 3.4 (Propriétés auxiliaires) *L'algorithme* `mgu` *jouit des propriétés suivantes :*

1. *toute variable hors de* $\mathcal{L}(C)$ *est hors de portée de* $\texttt{mgu}(C)$ *;*
2. $\texttt{mgu}(C)$ *est idempotente.*

Démonstration: La propriété (1) résulte du fait que $\texttt{mgu}(C)$ est la composée de substitutions élémentaires de la forme $[\alpha \leftarrow \tau]$, où α et τ proviennent de C. Démontrons à présent la propriété (2), par induction sur la définition de l'algorithme `mgu`. Tous les cas sont immédiats, exceptés les troisième et quatrième. Nous traitons le premier uniquement, le second étant symétrique. Supposons donc $\varphi = \texttt{mgu}(C[\alpha \leftarrow \tau])$. Il nous faut montrer que $\varphi \circ [\alpha \leftarrow \tau]$ est idempotente. α n'est pas libre dans τ, donc n'est pas libre dans $C[\alpha \leftarrow \tau]$. Dès lors, d'après la propriété (1), si α apparaît dans l'image d'un terme par φ, alors α apparaît dans ce terme. Cela implique que α n'apparaît pas dans l'image de $\varphi \circ [\alpha \leftarrow \tau]$, donc $\varphi \circ [\alpha \leftarrow \tau] \circ \varphi \circ [\alpha \leftarrow \tau] = \varphi \circ \varphi \circ [\alpha \leftarrow \tau] = \varphi \circ [\alpha \leftarrow \tau]$, l'idempotence de φ provenant de l'hypothèse d'induction. □

Exercice de programmation 3.3 *Implémenter la fonction* `mgu`.

3.2.4 L'algorithme d'inférence

En combinant construction et résolution des équations, on obtient l'algorithme d'inférence de types I suivant :
Entrée : Une expression a.
Sortie : Un typage (Γ, τ), ou bien un échec.
Calcul : Calculer $\varphi = \texttt{mgu}(C(a))$ (d'où échec possible). En cas de succès, renvoyer l'environnement $\{x : \varphi(\alpha_x) \mid x \in \mathcal{L}(a)\}$ et le type $\varphi(\alpha_a)$.

Proposition 3.5 (Propriétés de l'algorithme I) *L'algorithme* I *jouit des propriétés suivantes :*

1. *Correction : si* $I(a) = (\Gamma, \tau)$, *alors* $\Gamma \vdash a : \tau$.
2. *Principalité : si* $\Gamma' \vdash a : \tau'$, *alors* $I(a)$ *réussit et produit un typage* (Γ, τ) *plus général que* (Γ', τ'), *c'est-à-dire qu'il existe une substitution* θ *telle que* $\tau' = \theta(\tau)$ *et* $\Gamma'(x) = \theta(\Gamma(x))$ *pour tout* $x \in \mathcal{L}(a)$.

Démonstration: La correction de I est conséquence de la correction des équations engendrées (proposition 3.1) et de la correction de l'algorithme `mgu` (proposition 3.3).

Pour établir la principalité de I, supposons $\Gamma' \vdash a : \tau'$. D'après la proposition 3.2, le système $C(a)$ admet une solution φ' telle que $\tau' = \varphi'(\alpha_a)$ et $\Gamma'(x) = \varphi'(\alpha_x)$ pour tout $x \in \mathcal{L}(a)$. Alors, par complétude de l'algorithme `mgu` (proposition 3.3), l'exécution de $\mathtt{mgu}(C(a))$ réussit et produit une solution φ telle que $\varphi \leq \varphi'$, c'est-à-dire $\varphi' = \theta \circ \varphi$ pour une certaine substitution θ. Donc, $\tau' = \varphi'(\alpha_a) = \theta(\varphi(\alpha_a)) = \theta(\tau)$. De même, $\Gamma'(x) = \theta(\Gamma(x))$ pour tout $x \in \mathcal{L}(a)$. □

L'existence de l'algorithme I montre que la *typabilité* dans le λ-calcul simplement typé est décidable, et que tout programme admet un *typage principal*. Le système d'équations utilisé est de taille linéaire et est engendré en temps linéaire ; l'unification du premier ordre peut être implémentée en temps (quasi-)linéaire ; la *typabilité* dans le λ-calcul simplement typé est donc décidable en temps (quasi-)linéaire.

Exercice de programmation 3.4 *Implémenter l'algorithme I.*

3.3 Inférence de types pour mini-ML avec typage polymorphe

Peut-on appliquer au problème de l'inférence de types pour mini-ML une approche semblable à celle de la section 3.2, c'est-à-dire distinguer une phase de génération de contraintes et une phase de résolution ? La réponse est oui en principe, mais il faut alors enrichir le langage de contraintes. En effet, il ne suffit plus de manipuler des égalités entre types ; dans le cas de la règle (var-inst), il faut une contrainte exprimant qu'un certain type est instance d'un certain schéma ; et dans le cas de la règle (let-gen), il faut une contrainte exprimant qu'un certain schéma s'obtient par généralisation d'un certain type. L'algorithme de résolution de contraintes est donc nécessairement plus complexe.

Plutôt que de présenter un tel algorithme de résolution de contraintes, ce qui nous mènerait un peu loin, nous allons nous limiter pour l'instant à étudier l'algorithme monolithique proposé voilà vingt ans environ. Cet algorithme ne sépare pas production et résolution en deux phases distinctes ; les deux activités sont réalisées simultanément, en un seul parcours de l'arbre de syntaxe abstraite du programme.

3.3.1 L'algorithme W de Damas-Milner-Tofte

Commençons par définir l'algorithme. La version présentée ici est une variante de la version originale ; j'y ai apporté quelques modifications qui, j'espère, rendent l'algorithme légèrement plus lisible. J'ai néanmoins conservé le nom original : W.

Pour plus de lisibilité, l'algorithme est présenté dans un style d'apparence impérative, c'est-à-dire qu'il emploie deux variables globales φ et V. φ est la *substitution courante* ; elle représente l'unificateur principal des équations qui ont été résolues jusqu'ici ; par convention, il faudra donc appliquer φ aux types produits par l'algorithme ou trouvés dans l'environnement pour obtenir leur "véritable" signification. (La signification de cette phrase est clarifiée par le théorème 3.1.) V est un ensemble de *variables fraîches* ; à chaque fois que l'algorithme aura besoin d'une "nouvelle" variable intermédiaire, il la choisira dans V et l'en retirera, de façon à ne jamais utiliser deux fois la même, car cela conduirait à un résultat plus restrictif (moins général) que nécessaire.

Il est possible de déduire de cette présentation une autre description, rédigée de façon purement fonctionnelle, dans laquelle φ et V apparaissent comme deux entrées et deux sorties supplémentaires de l'algorithme. C'est à cette seconde description que nos preuves s'appliqueront en fait. Nous ne donnons pas cette seconde description, du fait de sa lourdeur, mais nous invitons le lecteur à la rédiger explicitement ; cela peut être instructif.

Entrée : un environnement de typage Γ et une expression a.

Sortie : le type inféré τ, ou bien échec.

État : une substitution courante φ et un ensemble infini de variables V.

Invariant : φ est de la forme $\mathtt{mgu}(C)$ pour un certain problème C ; aucune variable de V n'apparaît libre dans C ou dans Γ.

Notons que l'invariant entraîne d'une part que φ est idempotente, d'autre part que V est hors de portée de φ. Notons également qu'au lieu de mémoriser la substitution courante φ, on pourrait mémoriser le système d'équations courant C, et calculer $\varphi = \mathtt{mgu}(C)$ lorsqu'on en a besoin, c'est-à-dire en fait uniquement aux noeuds `let`. Le lecteur pourra considérer cette modification comme un exercice.

Définition :

$$
\begin{array}{lcl}
\mathtt{fresh} & = & \mathtt{do}\,\alpha \in V \\
 & & \mathtt{do}\,V \leftarrow V \setminus \{\alpha\} \\
 & & \mathtt{return}\,\alpha \\
W(\Gamma \vdash x) & = & \mathtt{let}\,\forall\alpha_1 \ldots \alpha_n.\tau = \Gamma(x) \\
 & & \mathtt{do}\,\beta_1, \ldots, \beta_n = \mathtt{fresh}, \ldots, \mathtt{fresh} \\
 & & \mathtt{return}\,\tau[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n] \\
W(\Gamma \vdash \mathtt{fun}\ x \rightarrow a_1) & = & \mathtt{do}\,\alpha = \mathtt{fresh} \\
 & & \mathtt{do}\,\tau_1 = W(\Gamma; x : \alpha \vdash a_1) \\
 & & \mathtt{return}\,\alpha \rightarrow \tau_1 \\
W(\Gamma \vdash a_1\,a_2) & = & \mathtt{do}\,\tau_1 = W(\Gamma \vdash a_1) \\
 & & \mathtt{do}\,\tau_2 = W(\Gamma \vdash a_2) \\
 & & \mathtt{do}\,\alpha = \mathtt{fresh} \\
 & & \mathtt{do}\,\varphi \leftarrow \mathtt{mgu}(\varphi(\tau_1) \stackrel{?}{=} \varphi(\tau_2 \rightarrow \alpha)) \circ \varphi \\
 & & \mathtt{return}\,\alpha \\
W(\Gamma \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2) & = & \mathtt{do}\,\tau_1 = W(\Gamma \vdash a_1) \\
 & & \mathtt{do}\,\sigma = \mathit{Gen}(\varphi(\tau_1), \varphi(\Gamma)) \\
 & & \mathtt{return}\,W(\Gamma; x : \sigma \vdash a_2) \\
W(\Gamma \vdash (a_1, a_2)) & = & \mathtt{do}\,\tau_1 = W(\Gamma \vdash a_1) \\
 & & \mathtt{do}\,\tau_2 = W(\Gamma \vdash a_2) \\
 & & \mathtt{return}\,\tau_1 \times \tau_2 \\
W(\Gamma \vdash \mathtt{fst}\ a) & = & \mathtt{do}\,\tau = W(\Gamma \vdash a) \\
 & & \mathtt{do}\,\alpha_1, \alpha_2 = \mathtt{fresh}, \mathtt{fresh} \\
 & & \mathtt{do}\,\varphi \leftarrow \mathtt{mgu}(\varphi(\tau) \stackrel{?}{=} \alpha_1 \times \alpha_2) \circ \varphi \\
 & & \mathtt{return}\,\alpha_1 \\
W(\Gamma \vdash \mathtt{snd}\ a) & = & \mathtt{do}\,\tau = W(\Gamma \vdash a) \\
 & & \mathtt{do}\,\alpha_1, \alpha_2 = \mathtt{fresh}, \mathtt{fresh} \\
 & & \mathtt{do}\,\varphi \leftarrow \mathtt{mgu}(\varphi(\tau) \stackrel{?}{=} \alpha_1 \times \alpha_2) \circ \varphi \\
 & & \mathtt{return}\,\alpha_2
\end{array}
$$

Notons que l'algorithme ne peut échouer que de deux façons : soit lorsque qu'une variable x non définie dans l'environnement Γ est rencontrée, soit lorsque l'algorithme d'unification `mgu` lui-même échoue.

La fonction `fresh` choisit une variable α quelconque dans V, la retire de V, et la renvoie. Notons qu'une telle variable existe toujours, car l'invariant garantit que V est infini. Elle n'est pas unique, donc l'algorithme est non déterministe ; mais, comme le prouvera le théorème 3.2, le type renvoyé par W est toujours principal, donc ce non-déterminisme n'est pas gênant.

Dans le cas où l'expression est une variable, les quantificateurs $\alpha_1, \ldots, \alpha_n$ et le type τ sont définis à renommage près seulement, mais cela n'introduit pas de non-déterminisme supplémentaire, car les α_i sont de toute façon remplacés par les β_i dans le type résultat ; le choix de leurs noms n'a donc pas d'incidence. L'algorithme, dans ce cas, produit une instance *générique* du schéma de types $\Gamma(x)$, c'est-à-dire une instance aussi générale que possible : chaque variable quantifiée est instanciée par une variable fraîche distincte.

Lorsque l'expression est une fonction, on infère le type de son corps, après ajout d'une hypothèse supplémentaire dans l'environnement. Celle-ci emploie une variable fraîche, car le domaine de la fonction n'est a priori pas connu. Une fois le type du corps inféré, on forme le type de la fonction.

Lorsque l'expression est une application, on infère séparément les types du membre gauche et du membre droit. On engendre ensuite une équation, laquelle exige qu'ils soient compatibles et nomme le type du résultat ; on emploie une variable fraîche pour représenter ce dernier. On résoud immédiatement l'équation à l'aide de l'algorithme `mgu`.

Notons que les deux cas précédents correspondent à l'algorithme I pour le λ-calcul simplement typé.

Dans le cas où l'expression est une construction `let`, on infère d'abord le type du membre gauche, à savoir τ_1. Ensuite, on le généralise. On ne calcule pas $Gen(\tau_1, \Gamma)$, car cela n'aurait aucun sens ; par exemple, dans le cas (courant) où τ_1 est une variable fraîchement choisie, elle ne serait pas libre dans Γ, et on obtiendrait le schéma de types $\forall\alpha.\alpha$. Il faut en fait tenir compte de la substitution φ, car pour satisfaire le système d'équations C, elle envoie souvent des variables distinctes vers le même terme ; ainsi, même si τ_1 et Γ ne partagent aucune variable, $\varphi(\tau_1)$ et $\varphi(\Gamma)$ peuvent en partager. Le schéma $Gen(\varphi(\tau_1), \varphi(\Gamma))$ est donc celui qui nous intéresse. (Cette explication sera précisée par les preuves des théorèmes 3.1 et 3.2.) Une fois le schéma obtenu, on l'ajoute à l'environnement, et on infère le type du membre droit.

Exemple on considère $a = \texttt{fun}\ x \to +(x, 1)$. Le déroulement de $W(\emptyset, a, V)$ est le suivant. (Nous utilisons ici la version purement fonctionnelle de W, où φ et V sont deux entrées et deux sorties supplémentaires de l'algorithme. Soit $\mathcal{V}$ l'ensemble de toutes les variables de types.)

$$\begin{array}{rcl}
W(x : \alpha \vdash +, id, \mathcal{V} \setminus \{\alpha\}) & = & (\texttt{int} \times \texttt{int} \to \texttt{int}, id, \mathcal{V} \setminus \{\alpha\}) \\
W(x : \alpha \vdash x, id, \mathcal{V} \setminus \{\alpha\}) & = & (\alpha, id, \mathcal{V} \setminus \{\alpha\}) \\
W(x : \alpha \vdash 1, id, \mathcal{V} \setminus \{\alpha\}) & = & (\texttt{int}, id, \mathcal{V} \setminus \{\alpha\}) \\
W(x : \alpha \vdash (x, 1), id, \mathcal{V} \setminus \{\alpha\}) & = & (\alpha \times \texttt{int}, id, \mathcal{V} \setminus \{\alpha\}) \\
W(x : \alpha \vdash +(x, 1), id, \mathcal{V} \setminus \{\alpha\}) & = & (\beta, [\alpha \leftarrow \texttt{int}, \beta \leftarrow \texttt{int}], \mathcal{V} \setminus \{\alpha, \beta\}) \\
W(\emptyset \vdash a, \texttt{id}, \mathcal{V}) & = & (\alpha \to \beta, [\alpha \leftarrow \texttt{int}, \beta \leftarrow \texttt{int}], \mathcal{V} \setminus \{\alpha, \beta\})
\end{array}$$

Cela signifie, d'après les théorèmes 3.1 et 3.2, que le type principal de a dans l'environnement vide est $\texttt{int} \to \texttt{int}$.

Exercice de programmation 3.5 *Implémentez l'algorithme W et testez votre implémentation à l'aide de quelques exemples. On pourra représenter l'ensemble de variables V à l'aide d'un simple compteur entier : pour engendrer une variable fraîche, on lui attribue le numéro courant, et on incrémente le compteur.*

Pour plus d'informations sur une implantation (simple mais pédagogique) de l'inférence de types pour mini-ML, on pourra consulter

Mark P. Jones. *Typing Haskell in Haskell.* Proceedings of the 1999 Haskell Workshop, Paris, France, October 1999. Published in Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht. URL : `http://www.cse.ogi.edu/~mpj/thih/`

Cet article traite du langage Haskell, qui contient bien plus que mini-ML ; en particulier, Haskell a un mécanisme de *type classes*, qui permet de surcharger les identificateurs, et dont l'impact sur le système de types est important. Néanmoins, c'est une lecture intéressante, et le code source du typeur (écrit en Haskell) est disponible.

3.3.2 Propriétés de l'algorithme W

Proposition 3.6 (Invariant) *L'invariant donné plus haut est correct et garantit, à l'issue de l'exécution de l'algorithme, qu'aucune variable de V n'est libre dans le type produit par l'algorithme.*

Démonstration: Notons d'abord que toute assertion de la forme "aucune variable de V n'apparaît libre dans..." est préservée lorsque V décroît. Notons également que la variable produite par `fresh` n'est pas élément de V à l'issue de l'exécution de `fresh`. La preuve de la proposition se fait par induction sur la définition de l'algorithme W.

∘ *Cas* x. Puisque qu'aucune variable de V n'apparaît libre dans Γ, V et $\mathcal{L}(\forall\alpha_1 \ldots \alpha_n.\tau)$ sont disjoints. De plus, les variables $\beta_1, \ldots, \beta_n$ produites par `fresh` ne sont pas éléments de V. Le type produit par l'algorithme, à savoir $\tau[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n]$, ne contient donc aucune variable de V.

∘ *Cas* `fun` $x \rightarrow a_1$. La variable α produite par `fresh` n'est pas élément de V, donc aucune variable de V n'apparaît libre dans l'environnement $\Gamma; x : \alpha$. Par hypothèse d'induction, l'appel récursif à W préserve l'invariant et produit un type τ_1 disjoint de V. Le type $\alpha \rightarrow \tau_1$ est donc également disjoint de V.

∘ *Cas* $a_1\, a_2$. Par hypothèse d'induction, les appels récursifs à W préservent l'invariant et produisent des types τ_1 et τ_2 disoints de V. La variable α produite par `fresh` est également hors de V. D'après l'invariant, φ est initialement de la forme $\mathtt{mgu}(C)$ pour un problème C disjoint de V. Le problème $C \cup \{\tau_1 \stackrel{?}{=} \tau_2 \rightarrow \alpha\}$ est donc également disjoint de V. De plus, par définition de l'algorithme `mgu`, il est facile de vérifier que $\mathtt{mgu}(C \cup \{\tau_1 \stackrel{?}{=} \tau_2 \rightarrow \alpha\})$ n'est autre que $\mathtt{mgu}(\varphi(\tau_1) \stackrel{?}{=} \varphi(\tau_2 \rightarrow \alpha)) \circ \varphi$. La nouvelle valeur de la variable d'état φ respecte donc l'invariant. Le type produit par l'algorithme, à savoir α, est disjoint de V.

∘ *Cas* `let` $x = a_1$ `in` a_2. Par hypothèse d'induction, le premier appel récursif à W préserve l'invariant et produit un type τ_1 disjoint de V. Parce que V est hors de portée de φ, le type $\varphi(\tau_1)$ est donc également disjoint de V, donc le schéma σ et l'environnement $\Gamma; x : \sigma$ de même. Par hypothèse d'induction, le second appel récursif à W préserve l'invariant et produit un type disjoint de V. □

Théorème 3.1 (Correction de l'algorithme W) *Si $W(\Gamma \vdash a)$ termine dans l'état (φ, V) et renvoie τ, alors $\varphi(\Gamma) \vdash a : \varphi(\tau)$ est dérivable.*

Démonstration: La preuve se fait par induction sur la définition de l'algorithme W. Notons que la substitution courante φ ne peut être que raffinée au cours de l'exécution de l'algorithme W, c'est-à-dire composée avec une nouvelle substitution. Or, les jugements de typage sont stables par substitution (proposition 1.2). Donc, si le jugement $\varphi(\Gamma) \vdash a : \varphi(\tau)$ est dérivable à un certain moment de l'exécution, il le reste ensuite à jamais, en dépit des modifications possibles de la substitution courante φ.

∘ *Cas* x. Par définition de la relation $\leq$, nous avons $\forall\alpha_1 \ldots \alpha_n.\tau \leq \tau[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n]$, donc le jugement $\Gamma \vdash x : \tau$ est valide. D'après la proposition 1.2, nous avons donc également $\varphi(\Gamma) \vdash x : \varphi(\tau)$.

∘ *Cas* $\texttt{fun}\ x \to a_1$. Par hypothèse d'induction, nous avons $\varphi(\Gamma); x : \varphi(\alpha) \vdash a_1 : \varphi(\tau_1)$. D'après la règle (fun), nous avons donc $\varphi(\Gamma) \vdash \texttt{fun}\ x \to a_1 : \varphi(\alpha \to \tau_1)$, à savoir le résultat attendu.

∘ *Cas* $a_1\ a_2$. Par hypothèse d'induction et d'après la remarque préliminaire ci-dessus, nous avons $\varphi(\Gamma) \vdash a_1 : \varphi(\tau_1)$ et $\varphi(\Gamma) \vdash a_2 : \varphi(\tau_2)$. De plus, φ est de la forme $\texttt{mgu}(C)$ où C contient l'équation $\tau_1 \stackrel{?}{=} \tau_2 \to \alpha$, donc φ est en particulier solution de cette équation : nous avons $\varphi(\tau_1) = \varphi(\tau_2) \to \varphi(\alpha)$. Par conséquent, la règle (app) donne $\varphi(\Gamma) \vdash a_1\ a_2 : \varphi(\alpha)$, à savoir le résultat attendu.

∘ *Cas* $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$. Notons φ_1 la valeur de la substitution courante φ à l'issue du premier appel récursif à W, et φ_2 le raffinement apporté par le second appel récursif, de sorte que dans l'état final $\varphi = \varphi_2 \circ \varphi_1$. Par hypothèse d'induction et d'après la remarque préliminaire ci-dessus, nous avons $\varphi(\Gamma) \vdash a_1 : \varphi(\tau_1)$ (1). Notons $\sigma = \mathit{Gen}(\varphi_1(\tau_1), \varphi_1(\Gamma))$ et τ_2 le type renvoyé par le second appel récursif. L'hypothèse d'induction, appliquée à celui-ci, donne $\varphi(\Gamma); x : \varphi(\sigma) \vdash a_2 : \varphi(\tau_2)$ (2).

Or, $\varphi_1(\sigma)$ n'est autre que σ. En effet, si ce n'était pas le cas, une variable libre dans σ serait affectée par φ_1 ; mais toute variable libre dans σ est, par construction, libre dans $\varphi_1(\Gamma)$, donc $\varphi_1(\Gamma)$ serait également affecté par φ_1 ; or φ_1 est idempotente, donc ceci est impossible. Nous avons donc $\varphi(\sigma) = \varphi_2(\sigma)$.

De plus, les variables généralisées, c'est-à-dire libres dans $\varphi_1(\tau_1)$ et non dans $\varphi_1(\Gamma)$, sont hors de portée de φ_2. En effet, φ_2 est le raffinement apporté par le second appel récursif à W, et ne peut donc affecter que des variables libres dans l'argument $\varphi_1(\Gamma)$ ou fraîches (c'est-à-dire éléments de V) lors de cet appel, ce qui n'est pas le cas de ces variables. Nous avons donc $\varphi_2(\sigma) = \varphi_2(\mathit{Gen}(\varphi_1(\tau_1), \varphi_1(\Gamma))) = \mathit{Gen}(\varphi_2 \circ \varphi_1(\tau_1), \varphi_2 \circ \varphi_1(\Gamma)) = \mathit{Gen}(\varphi(\tau_1), \varphi(\Gamma))$.

L'assertion (2) peut donc s'écrire $\varphi(\Gamma); x : \mathit{Gen}(\varphi(\tau_1), \varphi(\Gamma)) \vdash a_2 : \varphi(\tau_2)$. De (1) et (2), la règle (let-gen) permet alors de déduire $\varphi(\Gamma) \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \varphi(\tau_2)$, à savoir le résultat attendu. □

Théorème 3.2 (Complétude et principalité de l'algorithme W) *Soit Γ un environnement de typage ; soit (φ_0, V_0) l'état initial de l'algorithme, de sorte que l'invariant de l'algorithme soit initialement satisfait. Supposons donnés θ_0 et τ_0 tels que $\theta_0\varphi_0(\Gamma) \vdash a : \tau_0 \quad (J_0)$. Alors, l'exécution de $W(\Gamma \vdash a)$ réussit : soient (φ_1, V_1) son état final et τ_1 le type renvoyé. L'algorithme étant correct, on a $\varphi_1(\Gamma) \vdash a : \varphi_1(\tau_1) \quad (J_1)$. Alors il existe une substitution θ_1 telle que $\theta_0\varphi_0$ et $\theta_1\varphi_1$ coïncident en dehors de V_0 et $\tau_0 = \theta_1\varphi_1(\tau_1)$.*

Démonstration: Notons que l'existence d'une telle substitution θ_1 implique que J_0 est instance de J_1, c'est-à-dire que J_0 s'obtient à partir de J_1 par application de la substitution θ_1. Pour vérifier cela, il suffit en effet de remarquer que les variables libres dans Γ sont, par hypothèse, hors de V_0. La preuve se fait par induction sur la structure de l'expression a.

∘ *Cas* x. L'existence du jugement J_0 garantit que x est défini dans Γ. Par conséquent, l'exécution de $W(\Gamma \vdash x)$ réussit, et on a $\varphi_0 = \varphi_1$. Notons $\Gamma(x) = \forall\bar{\alpha}.\tau$, où les variables $\bar{\alpha}$ sont choisies hors de portée de θ_0 et de φ_0 et dans V_0. Le type τ_1 renvoyé par l'algorithme est de la forme $\tau[\bar{\alpha} \leftarrow \bar{\beta}]$. Le jugement J_0 doit être conséquence de la règle (var-inst), donc nous avons $\theta_0\varphi_0(\forall\bar{\alpha}.\tau) \leq \tau_0$, ce que l'on peut écrire $\forall\bar{\alpha}.\theta_0\varphi_0(\tau) \leq \tau_0$, ou encore $\rho\theta_0\varphi_0(\tau) = \tau_0$, pour une certaine substitution ρ dont le domaine est $\bar{\alpha}$. Définissons alors $\theta_1 = \rho\theta_0 \circ [\bar{\beta} \leftarrow \bar{\alpha}]$. Nous avons $\theta_1 \circ [\bar{\alpha} \leftarrow \bar{\beta}] = \rho\theta_0$. De plus, les variables $\bar{\alpha}$ et $\bar{\beta}$ sont hors de portée de φ_0, par construction pour les premières, d'après l'invariant de l'algorithme pour les secondes ; par conséquent, $\varphi_0 \circ [\bar{\alpha} \leftarrow \bar{\beta}] = [\bar{\alpha} \leftarrow \bar{\beta}] \circ \varphi_0$. Nous pouvons maintenant déduire

$$\tau_0 = \rho\theta_0\varphi_0(\tau) = \theta_1 \circ [\bar{\alpha} \leftarrow \bar{\beta}] \circ \varphi_0(\tau) = \theta_1\varphi_0 \circ [\bar{\alpha} \leftarrow \bar{\beta}](\tau) = \theta_1\varphi_0(\tau_1) = \theta_1\varphi_1(\tau_1)$$

Considérons maintenant une variable $\gamma \notin V_0$. D'après la définition de l'algorithme, les variables $\bar{\beta}$ sont dans V_0, donc $[\bar{\beta} \leftarrow \bar{\alpha}](\gamma) = \gamma$. Par ailleurs, par construction, les variables $\bar{\alpha}$ sont dans V_0 et sont hors de portée de θ_0 et de φ_0, donc, puisque ces variables forment le domaine de ρ, on a $\rho\theta_0\varphi_0(\gamma) = \theta_0\varphi_0(\gamma)$. Nous pouvons donc écrire

$$\theta_1\varphi_1(\gamma) = \rho\theta_0 \circ [\bar{\beta} \leftarrow \bar{\alpha}] \circ \varphi_0(\gamma) = \rho\theta_0\varphi_0 \circ [\bar{\beta} \leftarrow \bar{\alpha}](\gamma) = \rho\theta_0\varphi_0(\gamma) = \theta_0\varphi_0(\gamma)$$

donc $\theta_0\varphi_0$ et $\theta_1\varphi_1$ coïncident en dehors de V_0.

∘ *Cas* `fun` $x \to a_1$. Le jugement J_0 doit être conséquence de la règle (fun), donc τ_0 est de la forme $\tau_0^1 \to \tau_0^2$ et on a $\theta_0\varphi_0(\Gamma); x : \tau_0^1 \vdash a_1 : \tau_0^2 \quad (J_0')$. Définissons $\theta_0' = \theta_0\{\alpha \mapsto \tau_0^1\}$, où α est la variable fraîche choisie par l'algorithme. Notons que, parce que α est élément de V_0, l'invariant garantit que α est hors de portée de φ_0 et n'apparaît pas libre dans Γ. Le jugement J_0' peut donc s'écrire $\theta_0'\varphi_0(\Gamma; x : \alpha) \vdash a_1 : \tau_0^2$. L'hypothèse d'induction s'applique alors : l'appel $W(\Gamma; x : \alpha \vdash a_1)$ réussit, renvoie un type τ_1^2, et il existe une substitution θ_1 telle que $\theta_1\varphi_1$ et $\theta_0'\varphi_0$ coïncident en dehors de $V_0 \setminus \{\alpha\}$ (1) (c'est-à-dire hors de V_0 et sur α) et telle que $\theta_1\varphi_1(\tau_1^2) = \tau_0^2$. De (1), on déduit que $\theta_1\varphi_1$ et $\theta_0\varphi_0$ coïncident en dehors de V_0, à savoir la première partie de notre but. De (1) toujours, on tire également $\theta_1\varphi_1(\alpha) = \theta_0'\varphi_0(\alpha) = \tau_0^1$, ce qui permet ensuite de dériver $\theta_1\varphi_1(\alpha \to \tau_1^2) = \tau_0^1 \to \tau_0^2 = \tau_0$, à savoir la seconde partie de notre but.

∘ *Cas* $a_1\ a_2$. Le jugement J_0 doit être conséquence de la règle (app), donc nous avons $\theta_0\varphi_0(\Gamma) \vdash a_1 : \tau_0' \to \tau_0 \quad (J_0^1)$ et $\theta_0\varphi_0(\Gamma) \vdash a_2 : \tau_0' \quad (J_0^2)$. L'hypothèse d'induction garantit que le premier appel récursif à W réussit ; soient τ_1' le type renvoyé et (φ_1', V_1') l'état de l'algorithme à l'issue de cet appel. Il existe donc une substitution θ_1' telle que $\theta_1'\varphi_1'$ et $\theta_0\varphi_0$ coïncident hors de V_0 et $\theta_1'\varphi_1'(\tau_1') = \tau_0' \to \tau_0$. Parce que Γ est hors de V_0, le jugement J_0^2 peut s'écrire $\theta_1'\varphi_1'(\Gamma) \vdash a_2 : \tau_0'$, et l'hypothèse d'induction s'applique : le second appel récursif à W réussit. Soient τ_2' le type renvoyé et (φ_2', V_2') l'état de l'algorithme à l'issue de cet appel. Il existe donc une substitution θ_2' telle que $\theta_2'\varphi_2'$ et $\theta_1'\varphi_1'$ coïncident hors de V_1' et $\theta_2'\varphi_2'(\tau_2') = \tau_0'$. D'après la proposition 3.6, le type τ_1' est hors de V_1' ; nous avons donc $\theta_2'\varphi_2'(\tau_1') = \theta_1'\varphi_1'(\tau_1') = \tau_0' \to \tau_0$. Définissons à présent $\theta_2'' = \theta_2'\{\alpha \mapsto \tau_0\}$, où α est la variable fraîche choisie par l'algorithme. La variable α est dans V_2', donc hors de portée de φ_2' et hors de τ_1' et τ_2'. Par conséquent, nous avons $\theta_2''\varphi_2'(\tau_1') = \theta_2'\varphi_2'(\tau_1') = \tau_0' \to \tau_0$ et $\theta_2''\varphi_2'(\tau_2' \to \alpha) = \theta_2'\varphi_2'(\tau_2') \to \theta_2''(\alpha) = \tau_0' \to \tau_0$. La substitution θ_2'' est donc solution de l'équation $\varphi_2'(\tau_1') \stackrel{?}{=} \varphi_2'(\tau_2' \to \alpha)$. Cela garantit que l'appel à l'algorithme `mgu` réussit et produit une substitution ψ dont θ_2'' peut être dérivée ; en d'autres termes, il existe une substitution θ_1 telle que $\theta_2'' = \theta_1\psi$. Dans l'état final de l'algorithme, on a par définition $\varphi_1 = \psi\varphi_2'$. Considérons maintenant une variable $\gamma \notin V_0$. Nous avons $\theta_1\varphi_1(\gamma) = \theta_1\psi\varphi_2'(\gamma) = \theta_2''\varphi_2'(\gamma)$. Puisque α est distinct de γ et hors de portée de φ_2', nous pouvons continuer avec $\theta_2''\varphi_2'(\gamma) = \theta_2'\varphi_2'(\gamma) = \theta_1'\varphi_1'(\gamma) = \theta_0\varphi_0(\gamma)$. $\theta_1\varphi_1$ et $\theta_0\varphi_0$ coïncident donc en dehors de V_0, ce qui forme la première partie de notre but. Pour en établir la seconde partie, on vérifie

$$\theta_1\varphi_1(\alpha) = \theta_1\psi\varphi_2'(\alpha) = \theta_1\psi(\alpha) = \theta_2''(\alpha) = \tau_0$$

∘ *Cas* `let` $x = a_1$ `in` a_2. Le jugement J_0 doit être conséquence de la règle (let-gen), donc nous avons $\theta_0\varphi_0(\Gamma) \vdash a_1 : \tau_0' \quad (J_1')$ et $\theta_0\varphi_0(\Gamma); x : Gen(\tau_0', \theta_0\varphi_0(\Gamma)) \vdash a_2 : \tau_0 \quad (J_2')$. L'hypothèse d'induction s'applique au premier de ces jugements, et garantit donc que le premier appel récursif à W réussit. Soient τ_1' le type renvoyé et (φ_1', V_1') l'état de l'algorithme à l'issue de cet appel. Alors il existe une substitution θ_1' telle que $\theta_1'\varphi_1'$ et $\theta_0\varphi_0$ coïncident en dehors de V_0 et $\theta_1'\varphi_1'(\tau_1') = \tau_0'$. Soit σ le schéma construit par l'algorithme, c'est-à-dire $\sigma = Gen(\varphi_1'(\tau_1'), \varphi_1'(\Gamma))$. Notons que, parce que

φ_1' est idempotente, on a $\varphi_1'(\sigma) = \sigma$. Par conséquent, nous avons

$$\begin{array}{rcll} Gen(\tau_0', \theta_0\varphi_0(\Gamma)) & = & Gen(\theta_1'\varphi_1'(\tau_1'), \theta_1'\varphi_1'(\Gamma)) & \text{(d'après l'invariant, } \Gamma \text{ est hors de } V_0) \\ & \geq & \theta_1'(Gen(\varphi_1'(\tau_1'), \varphi_1'(\Gamma))) & \text{(lemme laissé au lecteur)} \\ & = & \theta_1'(\sigma) = \theta_1'\varphi_1'(\sigma) & \end{array}$$

Nous pouvons donc appliquer à (J_2') la proposition 1.4, et écrire le jugement obtenu sous la forme $\theta_1'\varphi_1'(\Gamma; x : \sigma) \vdash a_2 : \tau_0$. L'hypothèse d'induction s'applique alors à ce jugement et garantit que le second appel récursif à W réussit ; soient τ_1 le type renvoyé et φ_1 la substitution courante à l'issue de cet appel. Il existe alors une substitution θ_1 telle que $\theta_1\varphi_1$ et $\theta_1'\varphi_1'$ coïncident hors de V_1' et $\theta_1\varphi_1(\tau_1) = \tau_0$. Il en découle en particulier que $\theta_1\varphi_1$ et $\theta_0\varphi_0$ coïncident hors de V_0. Notre but est donc atteint. □

La preuve de complétude de l'algorithme W est restée pendant de longues années du domaine du “folklore”, c'est-à-dire que la seule preuve existante, celle de Damas, était non publiée, et personne n'en proposait d'autre. Lorsqu'on lit la preuve ci-dessus, cette situation s'explique assez aisément ! La preuve est relativement difficile et finalement tellement obscure que, en la lisant, on n'apprend rien de plus sur l'algorithme W...

Ce n'est que récemment que certains se sont attaqués à une mécanisation de la preuve de W, c'est-à-dire à la réalisation d'une preuve vérifiable par une machine. On pourra consulter les références suivantes :

> Catherine Dubois et Valérie Ménissier-Morain. *Typage de ML : Spécification et Preuve en Coq.* Actes du GDR Programmation, Rennes, novembre 1997. URL : `http://www.irisa.fr/lande/ridoux/GDR_annexe/dubois.ps.gz`
>
> Catherine Dubois et Valérie Ménissier-Morain. *Certification of a Type Inference Tool for ML : Damas-Milner within Coq.* Journal of Automated Reasoning 23(3–4) :319–346, novembre 1999. URL : `ftp://ftp.inria.fr/INRIA/Projects/cristal/Valerie.Menissier/submission_CD_VMM_JAR.ps.gz`

Notons pour finir qu'on peut imaginer diverses variantes de l'algorithme W. Dans une variante connue sous le nom de M, l'algorithme attend une entrée supplémentaire, à savoir le type attendu pour l'expression. (Ce type peut être une variable si en fait on n'attend aucun type particulier.) La façon dont les équations sont engendrées est alors légèrement différente, et les propriétés de l'algorithme sont les mêmes en cas de succès ; un intérêt de M est que, en cas d'échec, l'erreur est détectée plus tôt et le message d'erreur peut éventuellement être plus pertinent. Le lecteur intéressé pourra feuilleter par exemple

> Oukseh Lee et Kwangkeun Yi. *Proofs about a Folklore Let-Polymorphic Type Inference Algorithm.* ACM Transanctions on Programming Languages and Systems, 20(4) :707-723, July 1998. URL : `http://ropas.kaist.ac.kr/lib/dock/LeYi1998.pdf`

3.3.3 Typage polymorphe de ML par expansion des `let`

Il est important de savoir qu'il existe une autre façon, beaucoup plus simple, d'effectuer l'inférence de types pour mini-ML avec polymorphisme. Cette approche est motivée par la remarque suivante. Supposons l'expression $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$ bien typée et de type τ. Nous avons donc :

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad \Gamma; x : Gen(\tau_1, \Gamma) \vdash a_2 : \tau}{\Gamma \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau}\ \text{(let-gen)}$$

Le lemme 2.2, appliqué aux deux prémisses, garantit alors que $\Gamma \vdash a_2[x \leftarrow a_1] : \tau$. De plus, dans le langage mini-ML tel que nous l'avons présenté, les expressions $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$ et $a_2[x \leftarrow a_1]$ sont équivalentes, à terminaison près ; c'est-à-dire que, si toutes deux terminent, alors elles produisent la même valeur. (Il se peut toutefois que la première boucle tandis que la seconde termine, car la première évalue certainement a_1, tandis que la seconde pas nécessairement.)

Cela suggère que, au lieu de généraliser le type de a_1 et de typer a_2 sous l'hypothèse $x : Gen(\tau_1, \Gamma)$, nous pourrions simplement substituer x par a_1 partout dans a_2, et typer l'expression $a_2[x \leftarrow a_1]$. Plus formellement, cela revient à remplacer la règle (let-gen) ci-dessus par la règle (let-subst) suivante :

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad \Gamma \vdash a_2[x \leftarrow a_1] : \tau}{\Gamma \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau} \text{ (let-subst)}$$

La raison pour laquelle (let-subst) exige aussi que a_1 soit bien typée, et non pas uniquement que $a_2[x \leftarrow a_1]$ soit bien typée, est que sinon nous laisserions passer des expressions de la forme

$$\texttt{let}\ x = \text{(expression mal typée)}\ \texttt{in}\ \text{(expression n'utilisant pas } x\text{)}$$

comme par exemple `let x = 1 2 in 3`.

Exemple : avec la règle (let-subst), on a

$$\emptyset \vdash \texttt{let f} = \texttt{fun}\ x \rightarrow x\ \texttt{in}\ (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}$$

parce que $\emptyset \vdash \texttt{fun}\ x \rightarrow x : \alpha \rightarrow \alpha$, ce qui satisfait la première prémisse de (let-subst), et parce que $\emptyset \vdash ((\texttt{fun}\ x \rightarrow x)\ 1, (\texttt{fun}\ x \rightarrow x)\ \texttt{true}) : \texttt{int} \times \texttt{bool}$, ce qui satisfait la seconde.

Une fois (let-gen) remplacée par (let-subst), l'environnement de typage Γ ne contient plus que les identificateurs liés par `fun` (ceux liés par `let` ne sont jamais ajoutés à Γ). Donc, nous n'avons plus besoin de schémas de types : il suffit de dire que Γ fait correspondre des types simples τ aux identificateurs x, comme dans le système de types monomorphe. De même, la règle (var-inst) n'est plus nécessaire, et nous pouvons la remplacer par l'axiome (var) du système monomorphe :

$$\Gamma \vdash x : \Gamma(x) \text{ (var)}$$

Nous sommes donc ramenés au système de types monomorphe de la section 1.5.2, auquel on a ajouté un mécanisme d'expansion *textuel*. La construction `let`, dans cette approche, n'est rien d'autre qu'une *macro*.

Pour justifier complètement l'approche décrite ci-dessus, il faut vérifier que le système monomorphe étendu par la règle (let-subst) permet de dériver exactement les mêmes jugements de typage que le système polymorphe. C'est une conséquence du théorème suivant :

Théorème 3.3 (Expansion du `let`) *Le jugement $\Gamma \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau$ est dérivable dans mini-ML polymorphe si et seulement s'il existe τ_1 tel que $\Gamma \vdash a_1 : \tau_1$ et $\Gamma \vdash a_2[x \leftarrow a_1] : \tau$.*

Démonstration: (esquissée). Comme on l'a fait remarquer plus haut, la partie "seulement si" est conséquence du lemme 2.2. Pour la partie "si", on montre à l'aide du théorème de principalité de W (théorème 3.2) qu'il existe un type τ_0 qui est principal pour a_1 dans Γ : c'est-à-dire, $\Gamma \vdash a : \tau_0$, et de plus pour tout type τ' tel que $\Gamma \vdash a : \tau'$, il existe une substitution ψ telle que $\tau' = \psi(\tau_0)$ et $\text{Dom}(\psi) \subseteq \mathcal{L}(\tau) \setminus \mathcal{L}(\Gamma)$. On prend alors $\sigma = Gen(\tau_0, \Gamma)$, et on montre par récurrence structurelle sur a que si $\Gamma; \Gamma' \vdash a_2[x \leftarrow a_1] : \tau$ avec $\text{Dom}(\Gamma) \cap \text{Dom}(\Gamma') = \emptyset$, alors $\Gamma; x : \sigma; \Gamma' \vdash a_2 : \tau$. □

Il faut noter que l'algorithme d'inférence à base de (let-subst) décrit dans cette section, bien que produisant les mêmes résultats que l'algorithme W, est cependant beaucoup moins efficace en pratique : lorsqu'il rencontre `let` $x = a_1$ `in` a_2, l'algorithme W type a_1 une seule fois, lui associe un schéma de types σ, et utilise une instance générique de σ à chaque fois que x est référencé dans a_2. L'algorithme basé sur l'expansion textuelle, cependant, re-type a_1 à chaque fois que x est référencé dans a_2, ce qui le conduit à ré-inférer à chaque fois une instance fraîche de σ. Comme σ est typiquement petit, tandis que a_1 peut être relativement gros, la différence d'efficacité est importante en pratique.

Par ailleurs, l'algorithme W travaille sur le programme lui-même, tandis que l'algorithme simpliste travaille sur le programme après macro-expansion des `let` ; cela peut se sentir lorsqu'on affiche des messages d'erreurs.

Enfin, lorsqu'on ajoute des traits impératifs au langage (cf. chapitre 5), les constructions `let` $x = a_1$ `in` a_2 et $a_2[x \leftarrow a_1]$ ne sont plus sémantiquement équivalentes, car la première évalue a_1 exactement une fois, tandis que la seconde l'évalue un nombre indéterminé de fois ; les effets de bord (allocation d'une cellule mémoire, écriture dans une cellule, etc.) ne sont pas les mêmes. L'expansion textuelle n'est alors plus possible, sauf si on exige que a_1 soit une valeur. Cette restriction est en fait souvent imposée, mais on peut en principe s'en passer (cf. chapitre 5).

3.3.4 Complexité du typage polymorphe de ML

Pour une étude détaillée de l'efficacité de l'algorithme W et de la complexité du problème de typabilité de ML (décider si un programme est typable), on pourra se reporter à

> Paris Kanellakis, Harry Mairson, et John Mitchell. *Unification and ML type reconstruction.* In Computational Logic—Essays in Honor of Alan Robinson, MIT Press, 1991, pages 444–478. Adresse : `http://theory.stanford.edu/people/jcm/papers/complexity-type-inf.dvi.Z`.

Les principaux résultats en sont les suivants :
- Si n est la taille du programme d'entrée, l'algorithme W tourne en temps $O(2^{2^n})$ si le type résultat est représenté par un arbre, et en temps $O(2^n)$ si on représente le type par un graphe acyclique (pour partager des sous-termes identiques).
- Le problème de typabilité de ML est NP-dur et DEXPTIME-complet.

Voici un exemple simple de programme de taille $O(n)$ dont le type principal est de taille $O(2^n)$, qu'il soit représenté par un arbre ou par un graphe :

$$\texttt{let } f_0 = \texttt{fun } x \to x \texttt{ in let } f_1 = (f_0, f_0) \texttt{ in } \ldots \texttt{let } f_n = (f_{n-1}, f_{n-1}) \texttt{ in } f_n$$

Malgré cette complexité extrêmement élevée, l'algorithme W est très efficace en pratique (empiriquement, quasi-linéaire en la taille du programme source). Ceci est dû au fait que les programmes réalistes ne ressemblent pas à l'exemple ci-dessus.

Chapitre 4

Extensions simples de mini-ML

Nous décrivons dans ce chapitre quelques traits du véritable langage ML qui s'ajoutent facilement à notre langage noyau, mini-ML. D'autres traits plus difficiles à ajouter sont décrits dans les chapitres suivants.

4.1 Les n-uplets

Pour passer des paires de mini-ML aux n-uplets de ML, nous devons d'abord remplacer la construction "paire" et les deux constructions "projection" du langage par des constructions plus générales. Pour chaque entier $n \geq 0$, nous introduisons une construction "n-uplet" et n constructions "projection", indexées par $i \in \{1, \ldots, n\}$. Ce changement affecte non seulement la définition des expressions, mais aussi celles des valeurs et des contextes d'évaluation.

$$
\begin{array}{lrl}
\text{Expressions :} & a & ::= \ldots \mid (a_1, \ldots, a_n) \mid \mathtt{proj}_i^n\ a \\
\text{Valeurs :} & v & ::= \ldots \mid (v_1, \ldots, v_n) \\
\text{Contextes d'évaluation :} & E & ::= \ldots \mid (E, a_2, \ldots, a_n) \mid (v_1, E, \ldots, a_n) \mid \ldots \mid (v_1, v_2, \ldots, v_{n-1}, E) \\
 & & \quad \mid \mathtt{proj}_i^n\ E
\end{array}
$$

La règle de réduction associée à ces constructions devient

$$\mathtt{proj}_i^n\ (v_1, \ldots, v_n) \xrightarrow{\varepsilon} v_i$$

En ce qui concerne le système de types, le constructeur de types $\times$ (produit binaire) disparaît. À sa place, nous introduisons, pour chaque entier $n \geq 0$, un constructeur de types $\times_n$ (produit n-aire). Par souci de lisibilité, le type $\times_n(\tau_1, \ldots, \tau_n)$ sera noté $\tau_1 \times \ldots \times \tau_n$. Nous nous dotons alors des règles de typage suivantes :

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash a_n : \tau_n}{\Gamma \vdash (a_1, \ldots, a_n) : \tau_1 \times \ldots \times \tau_n} \qquad\qquad \frac{\Gamma \vdash a : \tau_1 \times \ldots \times \tau_n}{\Gamma \vdash \mathtt{proj}_i^n\ a : \tau_i}$$

Il est facile de vérifier que les résultats des chapitres précédents (correction du typage, existence d'un type principal) sont préservés par cette extension.

On notera que, dans le cas où $n = 0$, nous nous sommes dotés d'un type $\times_0$, lequel contient une unique valeur (), à laquelle aucune projection ne s'applique. Ce type correspond au type Caml `unit`.

Exercice 4.1 *(**) Vérifier que, en ce qui concerne la sémantique du langage, on pourrait se contenter d'introduire des constructions* $\mathtt{proj}_i\ a$, *capables d'accéder à la* i*-ème composante d'un tuple de longueur* $n \geq i$. *Démontrer cependant que, si l'on effectuait ce changement et si l'on conservait la famille de constructeurs de types* $(\times_n)_{n\geq 0}$, *alors la propriété d'existence de types principaux disparaîtrait.*

Exercice 4.2 *(*)/(**) Une autre manière de traiter les* n*-uplets est de les encoder par des paires imbriquées :* $(a_1, \ldots, a_n)$ *est vu comme une abréviation pour* $(a_1, (a_2, (a_3, \ldots, (a_{n-1}, a_n))))$.

(*) Définir la projection $\mathtt{proj}_{i,n}$ *en termes de* `fst` *et* `snd`.

(**) On note ML_t *le langage mini-ML avec tuples "primitifs" (non encodés) et* ML_p *le langage mini-ML avec paires "primitives" et tuples encodés comme expliqué précédemment. Formaliser la traduction* T *des programmes de* ML_t *dans* ML_p, *et montrer qu'elle commute avec la réduction : si* $a \to a'$ *dans* ML_t, *alors* $T(a) \to^\star T(a')$ *dans* ML_p. *Réciproquement, est-il vrai que si* $T(a) \to a''$, *alors il existe* a' *tel que* $a \to a'$ *et* $a'' = T(a')$ *?*

4.2 Les types sommes

Nous avons vu qu'une valeur de type $\tau_1 \times \tau_2$ est un objet composite formé d'une valeur de type τ_1 *et* d'une valeur de type τ_2. De façon duale, il est naturel d'introduire un type $\tau_1 + \tau_2$, dont les valeurs seront des objets formés d'une valeur de type τ_1 *ou* d'une valeur de type τ_2. De tels types sont appelés *types sommes.*

Par ailleurs, comme dans le cas des produits, on peut définir des types sommes à n branches, pour $n \geq 0$ arbitraire. C'est ce que nous allons faire directement.

Commençons donc par étendre la définition du langage. Pour chaque entier $n \geq 0$, on se donne n constructions "injection", permettant de construire une valeur de type somme, et une construction "match", permettant d'analyser une telle valeur. On notera la dualité vis-à-vis des produits : on a ici n constructeurs et un destructeur, tandis qu'un n-uplet a un constructeur et n destructeurs.

$$
\begin{array}{lrl}
\text{Expressions :} & a & ::= \ldots \mid \mathtt{inj}_i^n\ a \mid \mathtt{match}_n\, a\, a_1\, \ldots\, a_n \\
\text{Valeurs :} & v & ::= \ldots \mid \mathtt{inj}_i^n\ v \\
\text{Contextes d'évaluation :} & E & ::= \ldots \mid \mathtt{inj}_i^n\ E \\
 & & \quad \mid \mathtt{match}_n\, E\, a_1\, \ldots\, a_n \mid \mathtt{match}_n\, v\, E\, \ldots\, a_n \mid \ldots \mid \mathtt{match}_n\, v\, v_1\, \ldots\, E
\end{array}
$$

La règle de réduction associée à ces constructions est

$$\mathtt{match}_n\, (\mathtt{inj}_i^n\ v)\, v_1\, \ldots\, v_n \xrightarrow{\varepsilon} v_i\, v$$

En d'autres termes, l'opération `match` détermine à quelle branche de la somme la valeur considérée appartient : c'est l'entier i, que l'on appelle parfois *tag* (étiquette) de la valeur ; cela fait, elle appelle la i-ème fonction v_i, et lui passe la valeur v en paramètre.

En ce qui concerne le système de types, nous introduisons, pour chaque entier $n \geq 0$, un constructeur de types $+_n$ (somme n-aire). Par souci de lisibilité, le type $+_n(\tau_1, \ldots, \tau_n)$ sera noté $\tau_1 + \ldots + \tau_n$. Nous nous dotons alors des règles de typage suivantes :

$$\frac{\Gamma \vdash a : \tau_i}{\Gamma \vdash \mathtt{inj}_i^n\ a : \tau_1 + \ldots + \tau_n} \qquad \frac{\Gamma \vdash a : \tau_1 + \ldots + \tau_n \quad \Gamma \vdash a_1 : \tau_1 \to \tau \quad \ldots \quad \Gamma \vdash a_n : \tau_n \to \tau}{\Gamma \vdash \mathtt{match}_n\, a\, a_1\, \ldots\, a_n : \tau}$$

La règle de gauche est simple : si on dispose d'une valeur de type τ_i, alors en lui accolant le *tag* i, on obtient une valeur de type $\tau_1 + \ldots + \tau_n$. La règle de droite est plus complexe : si a est une valeur de type $\tau_1 + \ldots + \tau_n$, et si chaque fonction a_i est capable, à partir d'une valeur de type τ_i, de produire un résultat de type τ, alors la construction `match` est bien typée et produit elle-même un résultat de type τ. On pourra vérifier que, si on interprète le symbole $+$ par $\vee$ ("ou") et le symbole $\rightarrow$ par $\Rightarrow$ ("implique"), alors cette règle est une règle de déduction logique correcte ; c'est là un exemple de la correspondance de Curry-Howard.

Exercice 4.3 *(**) Vérifier que la correction du typage est préservée par cette extension.*

Exercice 4.4 *(*) Indiquer comment adapter l'algorithme W au langage ainsi étendu.*

Exercice 4.5 *(***) Prouver la correction et la complétude de l'algorithme W ainsi adapté.*

4.3 Les types récursifs

Jusqu'ici, nous avons raisonné en termes d'un univers de types *finis*. On peut néanmoins relaxer cette contrainte et se placer dans un univers contenant également des types *récursifs*, c'est-à-dire où les types sont interprétés comme des arbres réguliers. Nous allons décrire cette idée de façon succincte seulement.

Examinons d'abord l'impact de ce changement sur l'algorithme d'inférence. Il est mineur. On ne change pas la syntaxe des types manipulés par l'algorithme, qui restent donc des arbres finis contenant des variables de types. L'algorithme engendre et résoud toujours des équations entre types. La différence apparaît lorsque l'on s'intéresse à la notion de *solution* d'un ensemble d'équations. On considérera dorénavant des substitutions φ des variables vers les arbres réguliers (sans variables), et non plus des variables vers les types. Ainsi, une équation de la forme $\alpha \stackrel{?}{=} \alpha \rightarrow \alpha$ admettra une solution, à savoir la substitution qui à α associe l'arbre régulier $\mu t.t \rightarrow t$. (Cette notation représente l'arbre binaire complet où chaque nœud est étiqueté par $\rightarrow$.) L'algorithme d'unification sera modifié de deux façons : d'une part, l'*occur-check*, qui rejetterait l'équation ci-dessus, est supprimé ; d'autre part, du fait que ce test est supprimé, les structures manipulées par l'algorithme ne seront plus des DAGs comme précédemment, mais des graphes potentiellement cycliques.

En ce qui concerne les règles de typage, on peut procéder au moins de deux façons. L'une est d'étendre le langage de types par une construction $\tau ::= \ldots \mid \mu\alpha.\tau$ permettant de décrire tout arbre régulier, et de considérer les types modulo la loi d'expansion $\mu\alpha.\tau \approx \tau[\alpha \leftarrow \mu\alpha.\tau]$. L'autre consiste à ne pas étendre le langage de types, mais à ajouter un système d'équations C parmi les hypothèses de chaque jugement, ainsi qu'une règle permettant d'utiliser ces équations, c'est-à-dire de substituer un type par un autre si un ou plusieurs équations impliquent qu'ils sont égaux.

L'ajout de types récursifs au système permet d'accepter plus de programmes. Par exemple, le programme $\lambda x.(xx)$ est bien typé. Un jugement le décrivant est (dans la première approche)

$$\emptyset \vdash \lambda x.(xx) : \mu\alpha.\alpha \rightarrow \beta$$

ou encore (dans la seconde approche)

$$\{\alpha \stackrel{?}{=} \alpha \rightarrow \beta\}, \emptyset \vdash \lambda x.(xx) : \alpha$$

On peut essayer ce système à l'aide de `ocaml -rectypes`.

En pratique, les types récursifs se marient mal avec l'inférence de types de ML, car trop de programmes objectivement erronés (au sens où ils ne correspondent pas à ce que le programmeur avait en tête) sont alors acceptés. Par exemple, tout programme du λ-calcul pur est bien typé et admet le type $\mu\alpha.\alpha \rightarrow \alpha$! C'est pourquoi ils ne sont pas disponibles en Caml.

Pourtant, les types récursifs ont une utilité potentielle : combinés avec les types produits et sommes, ils permettent de décrire des structures de données arbitraires : listes, arbres, etc. Par exemple, on peut considérer le type des listes d'entiers comme la solution de l'équation récursive

$$\text{IntList} = \texttt{unit} + \texttt{int} \times \text{IntList}$$

Cette équation indique qu'une liste est soit la liste vide, représentée par $\texttt{inj}_1^2\ ()$, soit une liste à au moins un élément, de la forme $\texttt{inj}_2^2\ (x, l)$, où x est le premier élément (entier) et l la queue de la liste. Ce type permet de décrire des listes de longueur arbitraire, ce qui est impossible dans mini-ML doté uniquement de types produits et sommes.

Pour permettre la définition du type des listes, tout en excluant les types récursifs, le langage ML a recours à une construction nouvelle, à savoir la définition de *types algébriques*. Celle-ci permet de définir un type abstrait IntList ainsi que des fonctions permettant de convertir un objet de type IntList en un objet de type $\texttt{unit} + \texttt{int} \times \text{IntList}$, et vice-versa. Ainsi, on obtient essentiellement l'effet souhaité, sans pour autant introduire de types récursifs dans le langage. On peut dire que le type IntList est *isomorphe* au type $\texttt{unit} + \texttt{int} \times \text{IntList}$, au sens où il existe des fonctions de conversion, mais non *égal* : en ce qui concerne l'algorithme d'inférence, ces deux types n'ont aucun lien. On parle parfois de types *iso-récursifs* dans ce cas, et de types *équi-récursifs* lorsqu'on dispose de véritables types récursifs.

4.4 Les types algébriques

Nous avons expliqué ci-dessus que le mécanisme de définition de types algébriques permet d'éviter les problèmes liés à l'introduction de types récursifs. Ce mécanisme vise par ailleurs à régler un problème indépendant, à savoir celui du repérage par *position* des composantes des types produits et sommes. Dans les sections 4.1 et 4.2, ces composantes sont indexées par un entier i. Le programmeur doit donc explicitement mémoriser la signification de chaque index. On aimerait mieux repérer les composantes des types produits et sommes par *nom*, c'est-à-dire leur attribuer un nom qui serait utilisé à la fois lors de la construction et lors de l'accès à une valeur de type produit ou somme. La déclaration de types algébriques permet cela. On pourra ainsi définir des types produits à champs nommés (on parlera de types *enregistrements* ou *records*) et des types sommes à constructeurs nommés (on parlera parfois de types *concrets*).

En ML, une déclaration d'un type algébrique déclare soit un type enregistrement, soit un type concret ; elle nomme le type que l'on définit, et nomme également chacun des champs (pour un type enregistrement) ou des constructeurs (pour un type concret). De plus, elle fixe le type du contenu de chacun des champs (pour un type enregistrement) ou de l'argument de chaque constructeur (pour un type somme).

Exemples : Un type concret `num` regroupant nombres entiers et nombres en virgule flottante s'écrit

```
type num = Entier of int | Flottant of float
```

Le type des points dans l'espace s'écrit :

```
type point = { x : float ; y : float ; z : float }
```

Le type concret des expressions arithmétiques est :

```
type expr = Constante of int
          | Variable of string
```

```
                     | Add of expr * expr
                     | Diff of expr * expr
                     | Prod of expr * expr
                     | Quotient of expr * expr
```

On notera que le type `expr` introduit par cette déclaration est déjà mentionné dans la partie droite de la déclaration. C'est ainsi que le type `expr` est iso-récursif.

Un type algébrique peut être paramétré par un ou plusieurs types :

```
type 'a option = None | Some of 'a
type 'a list = Nil | Cons of 'a * 'a list
type ('a, 'b) paire = { fst : 'a ; snd : 'b }
```

Ainsi, `option` et `list` sont non pas des types, mais des constructeurs de types d'arité 1 ; `paire` est un constructeur de types d'arité 2. `int list` et `(int, float) paire` sont des types.

4.4.1 Types concrets

La forme générale d'une déclaration de type concret est :

$$\texttt{type}\ (\alpha_1, \dots, \alpha_p)\ t = C_1\ \texttt{of}\ \tau_1 \mid \dots \mid C_n\ \texttt{of}\ \tau_n$$

$\alpha_1, \dots, \alpha_p$ sont les paramètres du type t. (Dans le cas fréquent $p = 0$, on écrit juste $\texttt{type}\ t = \dots$.) $C_1, \dots, C_n$ sont les noms attribués aux constructeurs de données (i.e. aux injections) du type t. $\tau_1, \dots, \tau_n$ sont les types de leurs arguments. (Un constructeur de données sans arguments, comme `None` plus haut, peut être vu comme un constructeur à un argument de type `None of unit`.) On impose $\mathcal{L}(\tau_i) \subseteq \{\alpha_1, \dots, \alpha_p\}$ pour tout i.

Une telle déclaration étend le langage de la manière suivante :

Expressions :	a	$::= \dots \mid C_i\, a \mid \texttt{match}\, a\ C_1 : a_1\ \dots\ C_n : a_n$
Valeurs :	v	$::= \dots \mid C_i(v)$
Contextes d'évaluation :	E	$::= \dots \mid C_i(E)$
		$\mid \texttt{match}\, E\ C_1 : a_1\ \dots\ C_n : a_n \mid \texttt{match}\, v\ C_1 : E\ \dots\ C_n : a_n \mid \dots$
		$\mid \texttt{match}\, v\ C_1 : v_1\ \dots\ C_n : E$
Types :	τ	$::= \dots \mid (\tau_1, \dots, \tau_p)\ t$

L'opération de *filtrage* qui s'écrit en ML :

$$\texttt{match}\ a\ \texttt{with}\ C_1(x_1) \to a_1 \mid \dots \mid C_n(x_n) \to a_n$$

est codée ici par :

$$\texttt{match}\, a\ C_1 : (\texttt{fun}\ x_1 \to a_1)\ \dots\ C_n : (\texttt{fun}\ x_n \to a_n)$$

La règle de réduction est :

$$\texttt{match}\, (C_i\, v)\ C_1 : v_1\ \dots\ C_n : v_n \quad \xrightarrow{\varepsilon} \quad v_i\ v$$

On notera la similitude importante avec les définitions données dans la section 4.2 ; la seule différence, en ce qui concerne la sémantique, est le fait que les constructeurs de données (les injections) sont nommés.

Les règles de typage associées aux constructions C_i et `match` sont :

$$\frac{\Gamma \vdash a : \varphi(\tau_i) \qquad \mathrm{dom}(\varphi) = \{\alpha_1, \dots, \alpha_p\}}{\Gamma \vdash C_i\, a : \varphi((\alpha_1, \dots, \alpha_p)\ t)}$$

$$\frac{\begin{array}{c}\Gamma \vdash a : \varphi((\alpha_1, \dots, \alpha_p)\ t) \qquad \Gamma \vdash a_1 : \varphi(\tau_1 \to \tau) \qquad \dots \qquad \Gamma \vdash a_n : \varphi(\tau_n \to \tau) \\ \mathrm{dom}(\varphi) = \{\alpha_1, \dots, \alpha_p\}\end{array}}{\Gamma \vdash \mathtt{match}\, a\ C_1 : a_1\ \dots\ C_n : a_n : \varphi(\tau)}$$

On comparera avec la façon dont les sommes n-aires anonymes sont typées. Si on avait affaire à des sommes anonymes, on verrait apparaître le type $\tau_1 + \dots + \tau_n$ au lieu de $(\alpha_1, \dots, \alpha_p)\ t$. Les types donnés aux opérateurs d'injection et de filtrage exploitent donc l'isomorphisme entre ces deux types. L'opération d'injection permet de fait non seulement d'introduire un type somme $\tau_1 + \dots + \tau_n$, mais aussi de le convertir immédiatement en $(\alpha_1, \dots, \alpha_p)\ t$. À l'inverse, l'opération de filtrage permet de convertir le type $(\alpha_1, \dots, \alpha_p)\ t$ en $\tau_1 + \dots + \tau_n$, puis d'éliminer celui-ci. Les opérations sur les types algébriques combinent donc deux notions distinctes : d'une part introduction ou élimination d'un type somme (ou produit) anonyme, d'autre part introduction ou élimination du type nommé t. Par ailleurs, ici les types τ_i sont ceux fixés par la déclaration du type algébrique t, tandis que dans le cas des sommes anonymes, les τ_i sont arbitraires. L'emploi de la substitution φ indique que la règle de typage est valide pour toute instantiation des paramètres $\alpha_1, \dots, \alpha_p$.

On notera que plutôt que de voir les injections C_i et l'opération de filtrage $\mathtt{match}\, C_1, \dots C_n$ comme des constructions du langage, on peut les voir comme des constructeurs et des destructeurs. Dans ce cas, au lieu de leur associer une règle de typage, on leur associera simplement un schéma de types.

Exemples : pour le type `num` défini précédemment, on a :

$$\begin{array}{rcl} \mathtt{Entier} & : & \mathtt{int} \to \mathtt{num} \\ \mathtt{Flottant} & : & \mathtt{float} \to \mathtt{num} \\ \mathtt{match}_{\mathtt{num}} & : & \forall \beta.\ \mathtt{num} \to (\mathtt{int} \to \beta) \to (\mathtt{float} \to \beta) \to \beta \end{array}$$

Pour le type α `list`, on a de même :

$$\begin{array}{rcl} \mathtt{Nil} & : & \forall \alpha.\ \mathtt{unit} \to \alpha\ \mathtt{list} \\ \mathtt{Cons} & : & \forall \alpha.\ (\alpha \times \alpha\ \mathtt{list}) \to \alpha\ \mathtt{list} \\ \mathtt{match}_{\mathtt{list}} & : & \forall \alpha, \beta.\ \alpha\ \mathtt{list} \to (\mathtt{unit} \to \beta) \to (\alpha \times \alpha\ \mathtt{list} \to \beta) \to \beta \end{array}$$

Exercice 4.6 *(*) Déterminer les schémas de types associés aux constructeurs C_i et au destructeur $\mathtt{match}\, C_1, \dots C_n$, dans le cas général.*

4.4.2 Types enregistrements

La forme générale d'une déclaration de type enregistrement est :

$$\mathtt{type}\ (\alpha_1, \dots, \alpha_p)\ t = \{e_1 : \tau_1; \dots; e_n : \tau_n\}$$

Les e_i sont des *étiquettes* (*labels* en anglais) utilisées pour nommer les champs des enregistrements. La déclaration ci-dessus étend le langage de la manière suivante :

Expressions : $\quad a ::= \dots \mid \{e_1 = a_1; \dots; e_n = a_n\} \mid a.e_i$

Valeurs :	v	$::= \ldots \mid \{e_1 = v_1; \ldots ; e_n = v_n\}$
Contextes d'évaluation :	E	$::= \ldots \mid \{e_1 = E; \ldots ; e_n = a_n\} \mid \ldots$
		$\mid \{e_1 = v_1; \ldots ; e_n = E\} \mid E.e$
Types :	τ	$::= \ldots \mid (\tau_1, \ldots , \tau_p)\ t$

La règle de réduction est :

$$\{e_1 = v_1; \ldots ; e_n = v_n\}.e_i \stackrel{\varepsilon}{\rightarrow} v_i$$

Encore une fois, on notera la similitude importante avec les définitions données dans la section 4.1.

Les règles de typage associées sont :

$$\frac{\Gamma \vdash a_1 : \varphi(\tau_1) \qquad \ldots \qquad \Gamma \vdash a_n : \varphi(\tau_n) \qquad \mathrm{dom}(\varphi) = \{\alpha_1, \ldots , \alpha_p\}}{\Gamma \vdash \{e_1 = a_1; \ldots ; e_n = a_n\} : \varphi((\alpha_1, \ldots , \alpha_n)\ t)} \qquad \frac{\Gamma \vdash a : \varphi((\alpha_1, \ldots , \alpha_n)\ t) \qquad \mathrm{dom}(\varphi) = \{\alpha_1, \ldots , \alpha_p\}}{\Gamma \vdash a.e_i : \varphi(\tau_i)}$$

Encore une fois, on constatera que les règles sont similaires à celles des produits anonymes, mais exploitent de plus l'isomorphisme entre $\tau_1 \times \ldots \times \tau_n$ et $(\alpha_1, \ldots , \alpha_p)\ t$.

4.4.3 Remarques et exercices

Dans le langage ML, on peut définir plusieurs types algébriques en même temps ; dans ce cas, la définition peut être mutuellement récursive, c'est-à-dire que les membres droits de la définition peuvent faire référence à tous les types que l'on est en train de définir.

L'ordre des étiquettes dans l'expression $\{e_1 = a_1; \ldots ; e_p = a_p\}$, ou bien l'ordre des clauses dans l'expression `match` $a\ C_1 : a_1 \ldots C_n : a_n$, pourraient être autorisés à différer de l'ordre fixé par la définition du type algébrique t. En pratique, on autorise cela ; libre au compilateur de ré-ordonner silencieusement s'il le désire.

Les types abstraits algébriques ne permettent pas à deux types concrets d'avoir un constructeur de même nom, ou à deux types enregistrements d'avoir un champ de même nom. En effet, notre formalisation exige que chaque constructeur donne lieu à une construction du langage distincte, dotée de sa propre règle de typage ; de même pour les champs. On pourrait songer à identifier ces constructions dans le langage, mais on obtiendrait alors des constructions dotées chacune de plusieurs règles de typage, et l'algorithme d'inférence ne saurait pas laquelle choisir ; d'où perte de la propriété d'existence de types principaux. Nous verrons au chapitre 6 une solution possible à ce problème.

Exercice de programmation 4.1 *Ajouter les n-uplets et les types concrets à l'un des évaluateurs mini-ML écrits précédemment (programmes 1.1 ou 2.1).*

Exercice 4.7 *(*) Montrer que les booléens et l'expression conditionnelle* `if...then...else` *sont des cas particuliers de types concrets.*

Exercice 4.8 *(**) On considère le type concret suivant :*

```
type t = C of t → t
```

Donner quelques valeurs de ce type. Montrer qu'il existe deux fonctions totales `enrouler : (t → t) → t` *et* `dérouler : t → (t → t)`. *Utiliser ces fonctions pour donner un codage des termes du λ-calcul pur (non typé) sous forme d'expressions de type* `t`. *Est-ce que mini-ML sans opérateur de point fixe* `fix` *mais avec les types concrets est normalisant ?*

Exercice 4.9 *(***) Le langage ML offre des mécanismes de filtrage plus puissants que la construction de filtrage* `match` *utilisé ci-dessus. En particulier, on peut tester non seulement sur le constructeur de tête d'une valeur, mais aussi sur d'autres parties de la valeur. Exemple :*

```
match l with Nil -> 0 | Cons(x, Nil) -> 1 | Cons(x, y) -> 2
```

Cette expression renvoie 0 pour les listes vides, 1 pour les listes à un élément, et 2 pour les autres listes. Pour faire le même calcul en mini-ML, il faut emboîter deux filtrages comme suit :

```
match l Nil:(fun _ → 0) Cons:(fun y → match y Nil:(fun _ → 1) Cons:(fun _ → 2))
```

On considère mini-ML étendu comme suit :

Expressions :	a	$::= \ldots \mid (\texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid \_ \to a_3)$
Motifs :	p	$::= \_ \mid x \mid c \mid C(p) \mid (p_1, \ldots, p_n)$

Le motif _ filtre toutes les valeurs. Le motif x (où x est un identificateur) filtre également toutes les valeurs, et lie x à la valeur filtrée. Un motif restreint à une constante c filtre les valeurs égales à c uniquement. Le motif $C(p)$ filtre les valeurs de la forme $C(v)$ où de plus l'argument v est filtré par p. Enfin, le motif $(p_1, \ldots, p_n)$ filtre les n-uplets $(v_1, \ldots, v_n)$ tels que p_i filtre v_i pour $i = 1, \ldots, n$. La construction $\texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid \_ \to a_3$ *teste si p filtre la valeur de a_1 ; si oui, a_2 est évaluée après remplacement des identificateurs liés dans p par leurs valeurs ; si non, a_3 est évaluée.*

1) Donner la règle de réduction de la construction `match`.

2) Donner la règle de typage de la construction `match`.

3) Montrer que pour toute construction `match`*, il existe une expression mini-ML équivalente n'utilisant que les prédicats de filtrage* `match`.

4.5 Les contraintes de types

En ML, l'expression $(a : \exists\alpha_1, \ldots, \alpha_n.\tau)$, où $\mathcal{L}(\tau) \subseteq \{\alpha_1, \ldots, \alpha_n\}$, s'évalue comme a, mais force a à avoir le type τ, pour une certaine valeur de $\alpha_1, \ldots, \alpha_n$. En d'autres termes, elle force a à avoir un type de la forme $\varphi(\tau)$, où φ est une substitution arbitraire de domaine $\{\alpha_1, \ldots, \alpha_n\}$. Ainsi, $(1 : \exists\alpha.\alpha)$ est correct et a le type `int`. Dans le cas simple où τ est clos et $n = 0$, écrire $(a : \tau)$ oblige l'expression a à avoir exactement le type τ.

En mini-ML, on peut considérer $(a : \exists\alpha_1, \ldots, \alpha_n.\tau)$ comme l'application d'un opérateur $[\exists\alpha_1, \ldots, \alpha_n.\tau]$ à a. On donne à l'opérateur $[\exists\alpha_1, \ldots, \alpha_n.\tau]$ le schéma de types et la règle d'évaluation suivants :

$$\begin{array}{rcl} [\exists\alpha_1, \ldots, \alpha_n.\tau] & : & \forall\alpha_1, \ldots, \alpha_n.\ \tau \to \tau \\ [\exists\alpha_1, \ldots, \alpha_n.\tau]\, v & \stackrel{\varepsilon}{\to} & v \end{array}$$

Nous avons décrit ici une situation simple, où toutes les variables qui apparaissent dans le type τ doivent être localement liées. Ainsi, il est impossible d'utiliser deux annotations pour exiger, par exemple, que les deux composantes d'une paire aient le même type : si on écrit $(x : \exists\alpha.\alpha, y : \exists\alpha.\alpha)$, rien ne contraint x et y à avoir le même type, car les deux annotations ne font pas référence à la même variable α. En pratique, on peut souhaiter introduire un mécanisme plus complexe, dans lequel les variables de types qui apparaissent dans les annotations ne sont pas nécessairement liées localement. On pourra alors écrire $(x : \alpha, y : \alpha)$ pour obliger x et y à avoir le même type. Cependant, en contrepartie, on devra nécessairement indiquer *où* la variable α est liée (introduite) pour comprendre la signification précise d'une telle contrainte. (En Caml, les variables de types sont implicitement introduites à "toplevel". Cela est parfois surprenant pour l'utilisateur et trop restrictif.)

Chapitre 5

La programmation impérative

Dans ce chapitre, nous ajoutons à mini-ML plusieurs traits caractéristiques des langages impératifs, à savoir la modification en place du contenu de certaines cellules mémoire (appelées *références*), et les exceptions.

5.1 Les références

Le premier trait impératif que nous considérons est la possibilité de modifier en place le contenu d'un bloc de mémoire. Pour ce faire, nous ajoutons à mini-ML la notion de *référence*. Une référence est un bloc (ou cellule) de mémoire dont le contenu peut être modifié. Le vocabulaire employé est légèrement ambigu : le mot *référence* peut désigner soit la cellule mémoire elle-même, soit son adresse. Cela ne devrait pas nous gêner.

On crée une référence à l'aide de l'opération d'*allocation* **ref** a, qui crée une nouvelle cellule mémoire, dans laquelle est initialement stockée la valeur de l'expression a, et en renvoie l'adresse. Si a représente une référence, c'est-à-dire l'adresse d'une cellule, alors l'expression de *déréférencement* $!a$ en renvoie le contenu actuel. Enfin, on peut changer le contenu de la référence a_1 et y stocker la valeur d'une expression a_2 par l'opération d'*affectation* $a_1 := a_2$.

L'opération d'affectation ne renvoie aucun résultat ; ou plutôt, parce que toute expression doit avoir un résultat, elle se contente de renvoyer le résultat () de type `unit`. Ce résultat ne contenant aucune information, on le jette en général aussitôt, c'est-à-dire qu'on écrit typiquement

$$\texttt{let}\ x = (a_1 := a_2)\ \texttt{in}\ a$$

où x n'apparaît pas libre dans a. Cet idiome étant courant, on introduit le sucre syntaxique suivant :

$$a_1; a_2 \qquad \text{signifie} \qquad \texttt{let}\ x = a_1\ \texttt{in}\ a_2$$

où x n'apparaît pas libre dans a_2. Cette construction s'appelle la *séquence*.

On dit d'une opération qui modifie le contenu d'une référence, comme l'affectation, qu'elle a des *effets de bord*.

Exemple : L'expression `let r = ref 3 in r :=!r + 1 ; !r` produit la valeur 4. En effet, le contenu de la référence `r` est initialement 3. On calcule d'abord `!r + 1`, c'est-à-dire `3 + 1`, et on stocke cette valeur dans la référence `r`. Enfin, on renvoie le contenu courant de `r`, à savoir 4.

Une référence liée à un identificateur joue le même rôle qu'une variable dans un langage impératif. Par exemple, voici une fonction `gensym` qui renvoie un entier différent à chaque appel :

```
let compteur = ref 0
let gensym () = compteur := !compteur + 1 ; !compteur
```

La référence liée à l'identificateur `compteur` peut être vue comme une variable globale, que la fonction `gensym` lit puis modifie à chaque fois qu'elle est appelée. Il est intéressant de constater qu'on peut jouer sur l'endroit où `compteur` est définie pour la rendre invisible en dehors de la fonction :

```
let gensym =
  let compteur = ref 0 in
  function () ->
    compteur := !compteur + 1 ; !compteur
```

Ici, `compteur` se comporte toujours comme une variable globale, car l'expression `ref 0` n'est évaluée qu'une fois, et non une fois à chaque appel de `gensym`. Mais le seul identificateur lié par cette définition est `gensym`, donc l'identificateur `compteur` reste indéfini à l'extérieur. En d'autres termes, seule la fonction `gensym` peut accéder à la référence ainsi allouée. Ce comportement est proche de ce qu'on appelle "variable statique" en C : la *portée lexicale* de l'identificateur `compteur` est restreinte, mais la *durée de vie* de la référence à laquelle il est lié est non bornée. Pour finir, notons que le code

```
let un () =
  let compteur = ref 0 in
  compteur := !compteur + 1 ; !compteur
```

provoquerait l'allocation d'une nouvelle référence à chaque appel de `un`. Cette référence aurait alors un comportement similaire à celui des "variables locales" dans les langages impératifs. (La fonction `un` ci-dessus est peu intéressante : elle renvoie toujours 1.)

Une structure de données (liste, arbre, etc.) contenant des références est dite *mutable*, c'est-à-dire modifiable en place. Par exemple, le code suivant définit des listes dont chaque élément est une référence. L'épine dorsale (le chaînage) d'une telle liste est alors non mutable, mais le contenu de chaque élément l'est.

```
type α liste_à_éléments_mutables = α ref list
let rec nième l n = match l with x :: l' → if n = 0 then x else nième l' (n-1)
let lire_élément l i = !(nième l i)
let écrire_élément l i v = (nième l i) := v
```

Une liste simplement chaînée dont le chaînage même est mutable s'écrit :

```
type α liste_mutable = α liste_mut ref
and  α liste_mut = Nil | Cons of α * α liste_mutable
```

La concaténation en place de deux telles listes s'écrit :

```
let rec concat l1 l2 =
  match !l1 with
    Nil → l1 := !l2
  | Cons(head1, tail1) → concat tail1 l2
```

Digression : pour visualiser la représentation mémoire des structures de données, on pourra utiliser le module `Display` écrit par Jean-Christophe Filliâtre (`http://www.lri.fr/~filliatr/ftp/ocaml/display/`). Si `x` est une structure arbitraire, l'instruction `display "toto" x` stocke une représentation de cette structure dans le fichier `toto`, que l'on peut visualiser à l'aide de la commande `dot -Tps toto | gv -`.

Exercice 5.1 *(*) Que calcule la fonction* `f` *suivante ?*

```
let f =
  let r = ref (fun x → 0) in
  r := fun x → if x = 0 then 1 else x * (!r)(x-1);
 !r
```

La fonction `fun x → 0` *est-elle jamais appelée ? Quel est son rôle ?*

Exercice 5.2 *(***) Montrer que l'on peut définir le combinateur de point fixe* `fix` *à l'aide des références.*

5.2 Sémantique à réduction pour les références

Dans un langage purement fonctionnel, la valeur d'une expression a ne dépend que de la valeur de ses variables libres. En d'autres termes, lorsque ces dernières ont des valeurs connues, l'évaluation des sous-expressions de a peut s'effectuer indépendamment et suivant n'importe quelle stratégie ; la valeur produite sera la même. Cela n'est plus vrai dans un langage impératif : l'évaluation d'une sous-expression peut avoir des effets de bord, c'est-à-dire modifier des références, et donc affecter l'évaluation de toute autre sous-expression qui mentionne ces mêmes références.

Cette notion de partage de références doit être reflétée de façon formelle par la définition de la sémantique. Comme nous allons le voir, cela va nous obliger à distinguer l'*opération* d'allocation d'une référence, d'une part, et le *résultat* de cette allocation, c'est-à-dire l'adresse de la cellule ainsi allouée, d'autre part. Supposons en effet que nous identifiions ces deux notions. Nous poserions alors que toute expression de la forme `ref` v est une valeur, et représente (l'adresse d')une référence contenant la valeur v. Considérons à présent le terme

```
let r = ref 1 in r := 2; !r
```

Ici, on voudrait que les deux occurrences de `r` correspondent à la même référence, allouée une fois pour toutes par le `ref 1` en partie gauche du `let`. Mais puisque `ref` 1 est une valeur, la règle de β-réduction indique que cette expression se réduit en

```
(ref 1) := 2; !(ref 1)
```

Or, ce terme alloue deux références distinctes, modifie la première, et lit la seconde. Son comportement ne peut donc pas correspondre à celui que nous espérions initialement. Le problème vient du fait que nous avons *dupliqué* l'expression `ref` 1, et rien n'indique plus dans le résultat qu'il devrait s'agir de la *même* référence. La solution classique à ce problème consiste à considérer `ref` 1 comme une expression (non une valeur) capable de se réduire en une nouvelle sorte d'objet : une *adresse mémoire* ℓ. La β-réduction pourra alors dupliquer librement ℓ. Deux occurrences distinctes de ℓ représenteront bien la même référence, parce qu'il s'agira de la même adresse.

On se donne donc un ensemble infini d'adresses mémoires (*locations* en anglais), notées ℓ. Un état mémoire (*store*) s est une fonction partielle (à domaine fini) des adresses mémoires dans les valeurs. Les expressions et leurs valeurs possibles sont :

Expressions :	a	$::= \ldots$	comme précédemment
		$\mid \ell$	adresse mémoire
Valeurs :	v	$::= \ldots$	comme précédemment
		$\mid \ell$	adresse mémoire

On choisit ici de considérer `ref`, `!` et `:=` comme des opérateurs, ce qui explique pourquoi on ne les a pas ajoutés à la syntaxe des expressions. Une opération de création de référence s'évaluera en une adresse mémoire ℓ. Les programmes écrits par l'utilisateur ne peuvent pas contenir d'adresses mémoire ℓ ; ces dernières n'apparaîtront qu'au cours de l'évaluation.

On appellera *configuration* $a \,/\, s$ une paire d'une expression a et d'un état mémoire s. On pourra identifier les configurations modulo renommage des adresses mémoire. Ainsi, par exemple, les configurations $\ell \,/\, \{\ell \mapsto 3\}$ et $\ell' \,/\, \{\ell' \mapsto 3\}$ seront identifiées. En d'autres termes, les noms employés pour représenter les adresses n'ont aucune signification externe. Par souci de simplicité, nous considérerons uniquement des configurations closes, c'est-à-dire que toutes les adresses apparaissant dans a ou dans l'image de s doivent être dans le domaine de s.

La relation de réduction devient alors $a \,/\, s \to a' \,/\, s'$ (lire : "dans l'état mémoire initial s, l'expression a se réduit en l'expression a', et l'état mémoire à la fin de la réduction est s'"). Les règles définissant la relation de réduction sont les suivantes :

$$\begin{array}{rcllr}
(\texttt{fun}\ x \to a)\ v \,/\, s & \stackrel{\varepsilon}{\to} & a\{x \leftarrow v\} \,/\, s & & (\beta) \\
(\texttt{let}\ x = v\ \texttt{in}\ a) \,/\, s & \stackrel{\varepsilon}{\to} & a\{x \leftarrow v\} \,/\, s & & (\mathit{let}) \\
\texttt{fst}\ (v_1, v_2) \,/\, s & \stackrel{\varepsilon}{\to} & v_1 \,/\, s & & (\mathit{fst}) \\
\texttt{snd}\ (v_1, v_2) \,/\, s & \stackrel{\varepsilon}{\to} & v_2 \,/\, s & & (\mathit{snd}) \\
\texttt{ref}\ v \,/\, s & \stackrel{\varepsilon}{\to} & \ell \,/\, s\{\ell \mapsto v\} & \text{si } \ell \notin \mathrm{Dom}(s) & (\delta_{\mathrm{ref}}) \\
!\ell \,/\, s & \stackrel{\varepsilon}{\to} & s(\ell) \,/\, s & & (\delta_{\mathrm{deref}}) \\
:= (\ell, v) \,/\, s & \stackrel{\varepsilon}{\to} & (\,) \,/\, s\{\ell \mapsto v\} & & (\delta_{\mathrm{assign}})
\end{array}$$

$$\frac{a_1 \,/\, s_1 \stackrel{\varepsilon}{\to} a_2 \,/\, s_2}{E(a_1) \,/\, s_1 \to E(a_2) \,/\, s_2} \ \text{(contexte)}$$

On notera que, pour toutes les réductions provenant de mini-ML "pur", on a simplement transformé la règle de réduction originale $a_1 \stackrel{\varepsilon}{\to} a_2$ en $a_1 \,/\, s \stackrel{\varepsilon}{\to} a_2 \,/\, s$. On exprime ainsi le fait que ces réduction ne dépendent pas de l'état mémoire, et ne le modifie pas non plus.

La règle (δ_{ref}) alloue une nouvelle cellule mémoire. Cela est représenté par le choix d'une adresse mémoire ℓ arbitraire mais non déjà utilisée. Comme les configurations sont identifiées modulo renommage, la sémantique reste déterministe. La règle $(\delta_{\mathrm{deref}})$ consulte la valeur associée, dans l'état mémoire courant, à l'adresse ℓ. Notons que l'expression $!v$ est bloquée si v n'est pas une adresse mémoire. Notons également que l'on a nécessairement $\ell \in \mathrm{Dom}(s)$ du fait que l'on considère uniquement des configurations closes. (On vérifie aisément que cette propriété est préservée par toutes les règles de réduction.) La règle $(\delta_{\mathrm{assign}})$ modifie la valeur associée à ℓ dans l'état mémoire, et renvoie la valeur conventionnelle $(\,)$.

Exemple : La séquence de réductions suivante est valide :

$$\begin{array}{ll}
\multicolumn{2}{l}{\texttt{let}\ r = \texttt{ref}\ 3\ \texttt{in}\ r := !r + 1; !r \,/\, \emptyset} \\
\to & \texttt{let}\ r = \ell\ \texttt{in}\ r := !r + 1; !r \,/\, \{\ell \mapsto 3\} \\
\to & \ell := !\ell + 1; !\ell \,/\, \{\ell \mapsto 3\} \\
\to & \ell := 3 + 1; !\ell \,/\, \{\ell \mapsto 3\} \\
\to & \ell := 4; !\ell \,/\, \{\ell \mapsto 3\} \\
\to & (\,); !\ell \,/\, \{\ell \mapsto 4\} \\
\to & !\ell \,/\, \{\ell \mapsto 4\} \\
\to & 4
\end{array}$$

Exercice 5.3 *(*) Donner une sémantique opérationnelle "à grands pas" (dans le style de la section 1.4) pour mini-ML avec références. (Indication : la relation d'évaluation est de la forme* $a \,/\, s \twoheadrightarrow v \,/\, s'$*.)*

Exercice de programmation 5.1 *Ajouter les références à l'évaluateur de l'exercice 2.1.*

5.3 Typage des références

Pour typer les références, étendons notre algèbre de types :

Expressions de types : $\tau \ ::= \ldots \mid \tau\ \texttt{ref}$

$\tau\ \texttt{ref}$ sera le type des références dont le contenu est de type τ. Donnons ensuite aux opérateurs `ref`, ! et := les schémas de types naturels suivants :

$$\begin{aligned} \texttt{ref} &: \forall\alpha.\ \alpha \to \alpha\ \texttt{ref} \\ ! &: \forall\alpha.\ \alpha\ \texttt{ref} \to \alpha \\ := &: \forall\alpha.\ \alpha\ \texttt{ref} \times \alpha \to \texttt{unit} \end{aligned}$$

Malheureusement, ces définitions naturelles conduisent à un système de types incorrect, du moins en présence du polymorphisme de ML. (Les mêmes définitions, dans le cadre du λ-calcul simplement typé de la section 1.5.2, conduisent à un système sûr.) Considérons le programme suivant :

```
let r = ref (fun x → x) in
r := (fun x → x+1) ;
( !r) true
```

`r` reçoit le type polymorphe $\forall\alpha.\ (\alpha \to \alpha)\ \texttt{ref}$. L'affectation `r := (fun x → x+1)` est donc bien typée (on utilise `r` au type $(\texttt{int} \to \texttt{int})\ \texttt{ref}$), ainsi que l'application `( !r) true` (on utilise `r` au type $(\texttt{bool} \to \texttt{bool})\ \texttt{ref}$). L'expression ci-dessus est donc bien typée. Pourtant, sa réduction se bloque sur `true+1` qui n'est ni une valeur, ni réductible. Ce phénomène s'appelle "le problème des références polymorphes" dans la littérature.

Analyse du problème : esquissons une "preuve" de sûreté du typage pour voir précisément le problème. Pour étendre les preuves du chapitre 2, il faut savoir typer les étapes intermédiaires de la réduction, et donc les expressions contenant des adresses mémoire ℓ. Nous traitons les adresses mémoires comme des identificateurs : l'environnement de typage Γ associe des types aux adresses mémoire. Ces types peuvent être des schémas ou des types simples.

Si Γ associe des schémas σ aux adresses ℓ : la règle de typage pour les adresses ℓ est alors la même que pour les identificateurs "normaux", à savoir :

$$\frac{\Gamma(\ell) \le \tau}{\Gamma \vdash \ell : \tau}\ \text{(loc-inst)}$$

Cette approche n'est clairement pas sûre, car il suffit qu'une adresse ℓ reçoive un schéma de type $\forall\alpha.\tau$ non trivial (c'est-à-dire où α apparaît libre dans τ) pour que l'on puisse écrire dans ℓ une valeur d'un certain type $\tau[\alpha \leftarrow \texttt{int}]$ p.ex., puis relire cette même valeur en prétendant qu'elle est d'un autre type $\tau[\alpha \leftarrow \texttt{bool}]$ p.ex. Or, ces deux types sont incompatibles. (C'est ce qui se passe dans l'exemple ci-dessus.)

Si Γ associe des types simples τ aux adresses ℓ : la règle de typage des adresses est alors :

$$\Gamma \vdash \ell : \Gamma(\ell) \ \ \text{(loc)}$$

Il est maintenant impossible d'utiliser une référence avec plusieurs types différents : on est certain que les valeurs écrites dans ℓ puis relues depuis ℓ auront toutes le même type $\Gamma(\ell)$. Le typage des opérations ! et := redevient sûr. En revanche, la généralisation des types au moment du `let` pose alors problème : le typage n'est plus préservé par la règle (δ_{ref}). Considérons

$$\emptyset \vdash \texttt{let}\ r = \texttt{ref}\,(\texttt{fun}\ x \to x)\ \texttt{in}\ (!r)\,1; (!r)\,\texttt{true} : \texttt{bool}$$

Cette expression est bien typée puisque $\texttt{ref}(\texttt{fun}\ x \to x)$ a le type $(\alpha \to \alpha)\ \texttt{ref}$ et α est généralisable car non libre dans l'environnement de typage. Après réduction de $\texttt{ref}\,(\texttt{fun}\ x \to x)$, on obtient le terme $\texttt{let}\ r = \ell\ \texttt{in}\ (!r)\,1; (!r)\,\texttt{true}$ dans l'état mémoire $\{\ell \mapsto \texttt{fun}\ x \to x\}$. Cependant, ce terme n'est plus typable : il faudrait que

$$\ell : (\alpha \to \alpha)\ \texttt{ref} \vdash \texttt{let}\ r = \ell\ \texttt{in}\ (!r)\,1; (!r)\,\texttt{true} : \texttt{bool}$$

mais cela n'est pas possible car α est libre dans l'environnement de typage (dans le type de ℓ) et donc n'est plus généralisable.

Conclusion : il faut restreindre le typage statique de manière à ce qu'il garantisse la propriété suivante :

> Lors du typage de $\texttt{let}\ x = a\ \texttt{in}\ b$, on ne généralise pas dans le type de a les variables de types qui pourraient apparaître dans le type d'une référence allouée lors de l'évaluation de a.

5.4 Restreindre la généralisation aux expressions non expansives

La manière la plus simple d'assurer la propriété ci-dessus, et donc d'assurer la sûreté du typage des références, est de ne généraliser que les types des expressions *non expansives*, c'est-à-dire des expressions dont la forme (la syntaxe) est suffisante garantir que leur évaluation ne crée pas de références :

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad a_1 \text{ non expansive} \qquad \Gamma; x : \mathit{Gen}(\tau_1, \Gamma) \vdash a_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

Si l'expression a_1 est expansive, il n'est pas nécessaire de rejeter le programme, mais on fera en sorte que le type attribué à x ne soit pas généralisé. On peut exprimer cela en ajoutant à la règle ci-dessus une la règle suivante, qui est celle du λ-calcul simplement typé :

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad \Gamma; x : \tau_1 \vdash a_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

Il existe plusieurs façons de définir la classe des expressions non expansives. La plus simple, peut-être, est de considérer qu'une expression est non expansive si et seulement si c'est une valeur. Cette définition est clairement correcte : l'évaluation d'une valeur ne crée aucune référence, puisqu'elle ne donne lieu à aucun calcul. On parle, dans ce cas, de *restriction du polymorphisme aux valeurs* (*value restriction on polymorphism*). On peut cependant souhaiter adopter une définition légèrement plus permissive de cette classe. En voici un exemple. Les expressions non expansives a_{ne} sont décrites par la grammaire suivante :

Expressions non expansives :

$$
\begin{array}{lll}
a_{ne} & ::= x & \text{identificateurs} \\
& \mid c & \text{constantes} \\
& \mid op & \text{opérateurs} \\
& \mid \texttt{fun}\ x \to a & \text{fonctions} \\
& \mid (a'_{ne}, a''_{ne}) & \text{paires d'expressions non expansives} \\
& \mid \texttt{fst}\ a_{ne} & \text{projections d'expressions non expansives} \\
& \mid \texttt{snd}\ a_{ne} & \\
& \mid op(a_{ne}) & \text{si } op \neq \texttt{ref} \\
& \mid \texttt{let}\ x = a'_{ne}\ \texttt{in}\ a''_{ne} & \text{liaison } \texttt{let}
\end{array}
$$

On exclut des expressions non expansives les applications de l'opérateur `ref` (qui crée une nouvelle référence), ainsi que les applications de fonctions (car en général on ne sait pas si le corps de la fonction va créer ou non de nouvelles références).

Exemples : l'exemple problématique de référence polymorphe :

```
let r = ref (fun x → x) in
r := (fun x → x+1) ;
( !r) true
```

est maintenant rejeté. En effet, `ref (fun x → x)` est expansive, donc `r` doit recevoir un type simple de la forme $(\tau \to \tau)$ `ref`. La seconde ligne exige que τ soit égal à `int`, et la troisième que τ soit égal à `bool`. Le programme n'est donc pas typable.

Les exemples suivants restent bien typés car l'expression liée par `let` est non expansive :

```
let id = fun x → x in (id 1, id true)
let id = fst((fun x → x), 1) in (id 1, id true)
```

En revanche, l'exemple suivant devient non-typable avec la restriction de la généralisation :

```
let k = fun x → fun y → x in
let f = k 1 in
(f 2, f true)
```

En effet, l'expression `k 1` n'est pas non expansive, donc `f` doit recevoir un type simple de la forme $\tau \to$ `int`. `f` ne peut donc être utilisée de manière polymorphe par la suite. Pour rendre ce programme typable, le programmeur doit le modifier en effectuant une étape d'η-expansion manuelle :

```
let k = fun x → fun y → x in
let f = fun x → k 1 x in
(f 2, f true)
```

De manière générale, l'η-expansion permet de rendre non expansive (et donc généralisable) toute définition de fonction résultant d'un calcul (comme l'application `k 1` ci-dessus). Si l'expression η-expansée n'a aucun effet de bord, alors la sémantique du programme n'est pas affectée par cette modification.

La raison pour laquelle toute application de fonction est considérée comme potentiellement expansive est qu'elle peut cacher (de manière plus ou moins évidente) une création de références. Voici un exemple de création "évidente" :

```
let f x = ref(x) in
let r = f(fun x → x) in ...
```

Voici un exemple nettement moins évident, où la référence est encapsulée dans une paire de fonctions, l'une pour écrire dans la référence, l'autre pour lire son contenu courant :

```
let ref_fonctionnelle =
  fun x →
    let r = ref x in ((fun newx → r := newx), (fun () → !r)) in
let p = ref_fonctionnelle(fun x → x) in
let écrire = fst p in
let lire = snd p in
écrire(fun x → x+1) ;
(lire()) true
```

`ref_fonctionnelle` a le type $\forall\alpha.\ \alpha \rightarrow (\alpha \rightarrow \texttt{unit}) \times (\texttt{unit} \rightarrow \alpha)$. Bien que ce type ne mentionne aucun type `ref`, le résultat de `ref_fonctionnelle` est cependant fonctionnellement équivalent à α `ref`. Si le résultat de `ref_fonctionnelle(fun x → x)` était généralisé, le reste du programme serait bien typé et provoquerait une erreur à l'exécution. Il est donc crucial de considérer l'application `ref_fonctionnelle(fun x → x)` comme expansive.

La restriction de la généralisation est-elle gênante ? En pratique, parfois, mais très rarement. En effet, presque toutes les expressions polymorphes "utiles" sont des définitions de fonctions `fun` $x \rightarrow a$. Lorsqu'une fonction polymorphe est le résultat d'un calcul (d'une application de fonction p.ex.), une étape d'η-expansion permet souvent d'obtenir un programme équivalent et typable (voir l'exemple avec `k` ci-dessus).

Exercice 5.4 *(**) Trouvez un exemple où l'η-expansion rendue nécessaire par la restriction de la généralisation change le comportement du programme ou le rend moins efficace. (***) Essayez de trouver un tel exemple qui soit réaliste (qu'on pourrait rencontrer dans un programme réel).*

5.5 Preuve de sûreté du typage

Dans cette section, nous prouvons la sûreté du typage pour mini-ML + références + généralisation restreinte aux expressions non expansives. La preuve suit exactement la même approche que celle de la section 2.3.

Remarque : tous les résultats des chapitres 1 et 2 concernant la relation de typage $\Gamma \vdash a : \tau$ (en particulier le lemme de substitution 2.2) restent valides une fois que l'on a ajouté les adresses mémoire ℓ aux expressions a et leurs types aux environnements Γ. En effet, il suffit de considérer les adresses mémoires ℓ comme des identificateurs liés à des types monomorphes.

Définition : on dit qu'un état mémoire s est bien typé dans un environnement de typage étendu Γ, et on écrit $\Gamma \vdash s$, si et seulement si $\mathrm{Dom}(s) = \mathrm{Dom}(\Gamma)$ et pour toute adresse $\ell \in \mathrm{Dom}(s)$, il existe τ tel que $\Gamma(\ell) = \tau$ `ref` et $\Gamma \vdash s(\ell) : \tau$.

Définition : on dit qu'un environnement Γ' étend un environnement Γ si et seulement si Γ' étend Γ au sens des fonctions partielles.

Définition : on dit que a_1 / s_1 est moins typable que a_2 / s_2, et on note $a_1 / s_1 \sqsubseteq a_2 / s_2$, si pour tout environnement Γ et tout type τ,

- Si a_1 est non expansive : a_2 est non expansive, et de plus $\Gamma \vdash a_1 : \tau$ et $\Gamma \vdash s_1$ impliquent $\Gamma \vdash a_2 : \tau$ et $\Gamma \vdash s_2$.

– Si a_1 est expansive : $\Gamma \vdash a_1 : \tau$ et $\Gamma \vdash s_1$ impliquent qu'il existe Γ' étendant Γ tel que $\Gamma' \vdash a_2 : \tau$ et $\Gamma' \vdash s_2$.

Dans le cas où l'expression est expansive, son évaluation peut créer de nouvelles références, pour lesquelles des hypothèses de typage devront être ajoutées à l'environnement. C'est pourquoi on prévoit un environnement Γ' étendu. Dans le cas où l'expression est non expansive, l'environnement reste le même, et ce fait sera exploité dans la preuve.

Proposition 5.1 (Préservation du typage par réduction de tête) *Si $a_1 \,/\, s_1 \stackrel{\varepsilon}{\to} a_2 \,/\, s_2$, alors $a_1 \,/\, s_1 \sqsubseteq a_2 \,/\, s_2$.*

Démonstration: soient Γ et τ tels que $\Gamma \vdash a_1 : \tau$ et $\Gamma \vdash s_1$. On raisonne par cas sur la règle de réduction.

Cas des règles (β), (let), et de toutes les δ-règles du chapitre 2 : ces règles ne font pas intervenir l'état mémoire, c'est-à-dire que $s_2 = s_1$ et de plus $a_1 \stackrel{\varepsilon}{\to} a_2$ est une réduction de mini-ML sans les références. On démontre alors, comme au chapitre 2, que $\Gamma \vdash a_2 : \tau$.

De plus, on vérifie facilement par inspection des règles que si a_1 est non expansive, alors a_2 l'est aussi :

$$\begin{array}{rcll}
\mathtt{let}\ x = v\ \mathtt{in}\ a_{ne} & \stackrel{\varepsilon}{\to} & a_{ne}[x \leftarrow v] & (let) \\
+\ (n_1, n_2) & \stackrel{\varepsilon}{\to} & n & (\delta_+) \\
\mathtt{fst}\ (a^1_{ne}, a^2_{ne}) & \stackrel{\varepsilon}{\to} & a^1_{ne} & (fst) \\
\mathtt{snd}\ (a^1_{ne}, a^2_{ne}) & \stackrel{\varepsilon}{\to} & a^2_{ne} & (snd) \\
!\ell \,/\, s & \stackrel{\varepsilon}{\to} & s(\ell) \,/\, s & (\delta_{\mathrm{deref}}) \\
:= (\ell, v) \,/\, s & \stackrel{\varepsilon}{\to} & (\,) \,/\, s\{\ell \mapsto v\} & (\delta_{\mathrm{assign}})
\end{array}$$

Dans l'avant-dernier cas, nous utilisons le fait que $s(\ell)$ est une valeur, et que toute valeur est non expansive. Dans le premier cas, nous utilisons le fait que l'ensemble des expressions non expansives est clos par substitution de variables par des expressions non expansives, donc *a fortiori* par des valeurs.

Le résultat attendu découle facilement de ces deux remarques.

Cas de la règle (δ_{ref}) : on a $a_1 = \mathtt{ref}\ v$ et $a_2 = \ell$ et $s_2 = s_1\{\ell \mapsto v\}$ avec $\ell \notin \mathrm{Dom}(\Gamma)$. Remarquons que a_1 est expansive. La dérivation de typage concernant a_1 est nécessairement de la forme

$$\dfrac{\dfrac{TC(\mathtt{ref}) \leq \tau \to \tau\ \mathtt{ref}}{\Gamma \vdash \mathtt{ref} : \tau \to \tau\ \mathtt{ref}} \qquad \Gamma \vdash v : \tau}{\Gamma \vdash \mathtt{ref}\ v : \tau\ \mathtt{ref}}$$

Ainsi, on a $\Gamma \vdash v : \tau$. Soit $\Gamma' = (\Gamma; \ell : \tau\ \mathtt{ref})$. Puisque $\Gamma \vdash s_1$ et puisque $\ell \notin \mathrm{Dom}(s_1)$, on a bien $\Gamma' \vdash s_2$. De plus, $\Gamma' \vdash \ell : \tau\ \mathtt{ref}$ par la règle de typage des adresses mémoire.

Cas de la règle $(\delta_{\mathrm{deref}})$: on a $a_1 = !\ell$ et $a_2 = s_1(\ell)$ et $s_2 = s_1$. La dérivation de typage concernant a_1 est nécessairement de la forme

$$\dfrac{\dfrac{TC(!) \leq \tau\ \mathtt{ref} \to \tau}{\Gamma \vdash\ ! : \tau\ \mathtt{ref} \to \tau} \qquad \Gamma \vdash \ell : \tau\ \mathtt{ref}}{\Gamma \vdash !\ell : \tau}$$

De $\Gamma \vdash s_1$ et $\Gamma \vdash \ell : \tau\ \mathtt{ref}$, on déduit $\Gamma(\ell) = \tau\ \mathtt{ref}$ puis $\Gamma \vdash s_1(\ell) : \tau$. On a bien $\Gamma \vdash a_2 : \tau$ et $\Gamma \vdash s_2$ comme désiré (car a_1 est non expansive).

Cas de la règle (δ_{assign}) : a_1 est $:= (\ell, v)$ et a_2 est $()$ et $s_2 = s_1\{\ell \mapsto v\}$. La dérivation de typage concernant a_1 est nécessairement de la forme

$$\dfrac{\dfrac{TC(:=) \leq \tau\ \texttt{ref} \times \tau \to \texttt{unit}}{\Gamma \vdash (:=) : \tau\ \texttt{ref} \times \tau \to \texttt{unit}} \qquad \dfrac{\Gamma \vdash \ell : \tau\ \texttt{ref} \qquad \Gamma \vdash v : \tau}{\Gamma \vdash (\ell, v) : \tau\ \texttt{ref} \times \tau}}{\Gamma \vdash := (\ell, v) : \texttt{unit}}$$

Il s'ensuit $\Gamma(\ell) = \tau\ \texttt{ref}$ et donc $\Gamma \vdash s_2$. Par ailleurs, $\Gamma \vdash () : \texttt{unit}$ trivialement. D'où le résultat annoncé (a_2 est non expansive). □

Proposition 5.2 (Croissance de $\sqsubseteq$) *Pour tout contexte d'évaluation E, $a_1 / s_1 \sqsubseteq a_2 / s_2$ implique $E[a_1] / s_1 \sqsubseteq E[a_2] / s_2$.*

Démonstration: par induction sur la structure des contextes d'évaluation. Le seul cas intéressant est celui où le contexte considéré est de la forme $\texttt{let}\ x = E\ \texttt{in}\ a$. C'est le cas que l'on serait incapable de prouver si on n'avait pas restreint la généralisation aux expressions non expansives. Soient donc Γ et τ tels que $\Gamma \vdash \texttt{let}\ x = E[a_1]\ \texttt{in}\ a : \tau$ et $\Gamma \vdash s_1$. La dérivation de typage est de la forme suivante :

$$\dfrac{\Gamma \vdash E[a_1] : \tau_1 \qquad E[a_1] \text{ non expansive} \qquad \Gamma; x : \mathit{Gen}(\tau_1, \Gamma) \vdash a : \tau}{\Gamma \vdash \texttt{let}\ x = E[a_1]\ \texttt{in}\ a : \tau}$$

Appliquant l'hypothèse de récurrence à $E[a_1]$, il vient $E[a_1] / s_1 \sqsubseteq E[a_2] / s_2$. De cela, et du fait que $E[a_1]$ est non expansive, il découle $\Gamma \vdash E[a_2] : \tau_1$ et $\Gamma \vdash s_2$ et $E[a_2]$ est non expansive. Par conséquent, on peut construire la dérivation suivante :

$$\dfrac{\Gamma \vdash E[a_2] : \tau_1 \qquad E[a_2] \text{ non expansive} \qquad \Gamma; x : \mathit{Gen}(\tau_1, \Gamma) \vdash a : \tau}{\Gamma \vdash \texttt{let}\ x = E[a_2]\ \texttt{in}\ a : \tau}$$

d'où le résultat désiré. □

Proposition 5.3 (Préservation du typage par réduction) *Si $a_1 / s_1 \to a_2 / s_2$, alors $a_1 / s_1 \sqsubseteq a_2 / s_2$.*

Démonstration: conséquence des lemmes 5.1 et 5.2. □

Proposition 5.4 (Forme des valeurs selon leur type) *Soit Γ un environnement qui ne lie aucun identificateur x mais seulement des adresses ℓ. Supposons $\Gamma \vdash v : \tau$ et $\Gamma \vdash s$.*

1. *Si $\tau = \tau_1 \to \tau_2$, alors ou bien v est de la forme $\texttt{fun}\ x \to a$, 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.*
4. *Si $\tau = \tau_1\ \texttt{ref}$, alors v est une adresse mémoire $\ell \in \mathrm{Dom}(s)$.*

Démonstration: par examen des règles de typage applicables suivant la forme de τ. □

Proposition 5.5 (Lemme de progression) *Soit Γ un environnement qui ne lie aucun identificateur x mais seulement des adresses ℓ. Supposons $\Gamma \vdash a : \tau$ et $\Gamma \vdash s$. Alors, ou bien a est une valeur, ou bien il existe a' et s' tels que $a / s \to a' / s'$.*

Démonstration: semblable à celle du lemme 2.5. □

Théorème 5.1 (Sûreté du typage) *Si $\emptyset \vdash a : \tau$ et $a / \emptyset \to^\star a' / s'$ et a' / s' est en forme normale vis-à-vis de $\to$, alors a' est une valeur.*

5.6 L'approche de SML'90

De nombreux systèmes de types ont été proposés pour résoudre le problème des références poymorphes. Bien que parfois plus souples que l'approche de la section 5.4, ces autres systèmes de types ont tous été abandonnés en pratique car présentant un moins bon rapport expressivité/complexité que l'approche de la section 5.4.

L'approche suivie dans Standard ML 90 consiste à distinguer deux sortes de variables de types, les variables *applicatives* α_a et les variables *impératives* α_i. Une variable applicative peut être instanciée (substituée) par n'importe quelle expression de type τ, mais une variable impérative ne peut être instanciée que par un *type impératif* $\bar{\tau}$, qui est un type ne contenant pas de variables applicatives :

Types : $\quad \tau ::= \alpha_a \mid \alpha_i \mid T \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1\ \texttt{ref}$

Types impératifs : $\quad \bar{\tau} ::= \alpha_i \mid T \mid \bar{\tau}_1 \rightarrow \bar{\tau}_2 \mid \bar{\tau}_1 \times \bar{\tau}_2 \mid \bar{\tau}_1\ \texttt{ref}$

Les substitutions sont donc de la forme générale $[\alpha_a \leftarrow \tau, \alpha_i \leftarrow \bar{\tau}]$.

La plupart des constantes et opérateurs reçoivent des schémas de types "applicatifs" (où les variables quantifiées sont applicatives, leur permettant d'être instanciées par n'importe quel type ensuite). En particulier, on a :

$$\begin{aligned} ! &: \forall \alpha_a.\ \alpha_a\ \texttt{ref} \rightarrow \alpha_a \\ := &: \forall \alpha_a.\ \alpha_a\ \texttt{ref} \times \alpha_a \rightarrow \texttt{unit} \end{aligned}$$

En revanche, l'opérateur `ref` a un schéma de type "impératif", où la variable quantifiée est impérative et ne peut être instanciée plus tard que par des types impératifs :

$$\texttt{ref} \ : \ \forall \alpha_i.\ \alpha_i \rightarrow \alpha_i\ \texttt{ref}$$

La règle de généralisation du `let` est alors modifiée pour ne généraliser que les variables applicatives, mais pas les variables impératives (qui, intuitivement, sont les variables qui ont pu "participer" à une opération de création de référence polymorphe) :

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad \Gamma; x : \mathit{GenAppl}(\tau_1, \Gamma) \vdash a_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

On définit $\mathit{GenAppl}(\tau, \Gamma) = \forall \alpha_{a,1} \ldots \alpha_{a,n}.\ \tau$ où $\{\alpha_{a,1}, \ldots, \alpha_{a,n}\} = \mathcal{L}_a(\tau) \setminus \mathcal{L}_a(\Gamma)$ sont les variables applicatives libres dans τ mais non dans Γ.

En fait, SML'90 va plus loin et permet la généralisation des variables impératives lorsque l'expression liée par `let` est non expansive, ce que l'on peut exprimer en ajoutant à la règle ci-dessus la règle de la section 5.4 :

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad a_1 \text{ non expansive} \qquad \Gamma; x : \mathit{Gen}(\tau_1, \Gamma) \vdash a_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

On peut donc voir l'approche de la section 5.4 comme une simplification de celle de SML'90 où toutes les variables sont traitées comme des variables impératives.

Le principal intérêt d'une telle approche est que tout programme purement fonctionnel bien typé dans mini-ML sans références est également bien typé dans mini-ML avec références (ce qui n'est pas le cas lorsqu'on restreint le polymorphisme aux expressions non expansives). Mais on considère en général que cela ne justifie pas la complexité accrue du système.

Exemples :

`let id = fun x → x in`	`id :` $\forall\alpha_a.\ \alpha_a \to \alpha_a$
`let f = id id in`	`f :` $\forall\alpha_a.\ \alpha_a \to \alpha_a$
`(f 1, f true)`	`ok`
`let r = ref(fun x → x) in`	`r :` $(\alpha_i \to \alpha_i)\ ref$
`r := fun x → x+1 ;`	α_i `devient int`
`( !r) true`	`erreur`
`let f = fun x → ref(x) in`	`f :` $\forall\alpha_i.\ \alpha_i \to \alpha_i$
`let r = f(fun x → x) in`	`r :` $(\alpha_i \to \alpha_i)\ ref$
`r := fun x → x+1 ;`	α_i `devient int`
`( !r) true`	`erreur`

5.7 Systèmes d'effets et de régions

Ces systèmes ne se contentent pas d'un critère syntaxique pour déterminer si une expression risque ou non de créer des références, mais déduisent cette information à l'aide d'un système de types plus complexe, dans lequel chaque expression a non seulement un type, mais également un *effet*, lequel résume ses effets de bord potentiels. On pourra consulter

> *The type and effect discipline*, Jean-Pierre Talpin et Pierre Jouvelot, *Information and Computation* 111(2), 1994.
>
> *Typed Memory Management in a Calculus of Capabilities*, Karl Crary, David Walker, Greg Morrisett, *Conference Record of POPL'99, San Antonio, Texas.*

5.8 Les exceptions

Les exceptions sont un mécanisme très souple permettant au programmeur de signaler une erreur au cours d'un calcul et de laisser l'erreur se propager à travers les fonctions appelantes jusqu'à un point où un morceau de code a été prévu pour la traiter. On peut aussi s'en servir comme d'une structure générale de contrôle, p.ex. pour programmer des sorties de boucles.

Expressions : $a ::= \ldots \mid$ `try` a_1 `with` $x \to a_2$

Opérations : $op ::= \ldots \mid$ `raise`

L'opérateur `raise` interrompt l'évaluation courante et lève une exception. La construction `try` a_1 `with` $x \to a_2$ évalue a_1 et en renvoie la valeur si aucune exception n'est levée par ce calcul ; dans le cas contraire, elle évalue alors a_2 et en renvoie la valeur. Les exceptions portent un argument que l'on peut fournir à l'opérateur `raise` et qui sera lié à x dans a_2 par la construction `try` a_1 `with` $x \to a_2$. On peut ainsi communiquer une donnée du point où l'exception est levée vers le point où elle est rattrapée ; par exemple, un message indiquant la cause de l'erreur.

Exemple : `try 1 + (raise "Hello") with x → x` s'évalue en `"Hello"`.

Sémantique : La sémantique des exceptions est plus facile à définir que celle des objets mutables, car elle s'exprime directement par réduction des programmes, sans avoir besoin d'un état mémoire ou autre construction globale. On ajoute simplement les règles de réduction suivantes pour `try`/`with` :

$$\begin{array}{rcl} \texttt{try}\ v\ \texttt{with}\ x \to a & \stackrel{\varepsilon}{\to} & v \\ \texttt{try}\ \texttt{raise}\ v\ \texttt{with}\ x \to a & \stackrel{\varepsilon}{\to} & a[x \leftarrow v] \end{array}$$

Il faut aussi ajouter une règle exprimant la propagation des exceptions vers le haut des expressions :

$$\Delta[\texttt{raise}\ v] \quad \to \quad \texttt{raise}\ v \qquad \text{si } \Delta \text{ n'est pas } [\,]$$

Dans cette dernière règle, Δ est un contexte de réduction incapable de traiter une exception, c'est-à-dire où le trou n'est pas à l'intérieur d'une construction `try`... `with` :

Contextes d'évaluation :

$E \ ::= \ldots \mid \texttt{try}\ E\ \texttt{with}\ x \to a$

Contextes d'exceptions :

$\Delta \ ::= [\,] \mid \Delta\ a \mid v\ \Delta \mid \texttt{let}\ x = \Delta\ \texttt{in}\ a \mid (\Delta, a) \mid (v, \Delta) \mid \texttt{fst}\ \Delta \mid \texttt{snd}\ \Delta$

Les deuxième et troisième règles expriment donc que si l'évaluation d'une sous-expression lève une exception, l'exécution continue au niveau du `try`/`with` le plus proche englobant la sous-expression.

On notera qu'une expression de la forme `raise` v n'est pas une valeur mais est néanmoins irréductible. Un programme qui s'exécute correctement et termine produit donc à présent *soit* une valeur, *soit* une telle expression, c'est-à-dire une exception non rattrapée. Nous appellerons cette classe les *réponses* :

Réponses :

$r \ ::= v \mid \texttt{raise}\ v$

Nous dirons doranévant qu'une expression irréductible est *bloquée* si et seulement si ce n'est pas une réponse.

Typage des exceptions : ML associe un type spécial `exn` aux valeurs fournies à `raise` et rattrapées à l'aide de la construction `try`/`with`. (Ces valeurs sont du coup elles-mêmes appelées *exceptions.*) L'opérateur `raise` et la construction `try`/`with` sont alors typées de la façon suivante :

$$TC(\texttt{raise}) : \forall\alpha.\ \texttt{exn} \to \alpha$$

$$\frac{\Gamma \vdash a_1 : \tau \qquad \Gamma; x : \texttt{exn} \vdash a_2 : \tau}{\Gamma \vdash \texttt{try}\ a_1\ \texttt{with}\ x \to a_2 : \tau}$$

On note que le domaine de `raise` est bien `exn` ; de même, la variable x reçoit le type `exn` dans la construction `try`/`with`. Le codomaine de `raise` est arbitraire, car l'opérateur `raise` ne renvoie aucune valeur ; il lève une exception. On peut donc donner n'importe quel type à son (hypothétique) résultat. La règle de typage associée à `try`/`with` exige que l'expression a_1 et le gestionnaire d'exceptions a_2 (*handler* en anglais) renvoient des résultats de même type. Cela est nécessité par le fait qu'on ne sait pas, statiquement, lequel sera exécuté ; la situation est la même que pour la construction `match`, dont toutes les branches doivent avoir le même type.

Exercice 5.5 *(**) Montrer que le typage est préservé par la réduction et qu'aucun programme bien typé n'est bloqué.*

Le fait de fixer le type des exceptions comme étant obligatoirement `exn` simplifie considérablement le système. Si on pouvait fournir à `raise` un argument de type arbitraire, alors une expression a_1 pourrait lever des exceptions de type arbitraire. Pour que la construction $\texttt{try}\ a_1\ \texttt{with}\ x \rightarrow a_2$ soit sûre, il faudrait alors attribuer à x, lorsqu'on type le *handler* a_2, le type des exceptions levées par a_1. Le système devrait donc attribuer à chaque expression non seulement un type, mais aussi le type des exceptions qu'elle peut lever. De plus, il est relativement aisé de constater que cela nécessite que le type de chaque fonction soit annoté par le type des exceptions qu'elle peut lever. Le constructeur de types $\rightarrow$ deviendrait ainsi un constructeur ternaire. Cette construction est possible et a été étudiée, mais a semblé trop lourde pour être adoptée en ML. Notons que Java adopte ce système, mais la flexibilité en est diminuée par l'absence de polymorphisme.

Cela dit, fixer le type des exceptions n'est acceptable que si ce type est *extensible*, c'est-à-dire si sa définition peut être formulée de façon *incrémentale* ou *modulaire*. Pour cette raison, en ML, le type `exn` est en fait un *type concret extensible*, c'est-à-dire un type concret auquel le programmeur peut, à tout moment, ajouter de nouveaux constructeurs, à l'aide d'une déclaration de la forme

$$\texttt{exception}\ C\ \texttt{of}\ \tau$$

Ceci ajoute un constructeur $C : \tau \rightarrow \texttt{exn}$. Comme le constructeur de types `exn` n'a pas de paramètres, τ ne doit pas comporter de variables libres. Le fait que le type `exn` soit extensible implique qu'un filtrage sur une exception ne peut être exhaustif.

La construction `try`/`with` de ML combine le rattrapage d'une exception et le filtrage, c'est-à-dire qu'on peut écrire $\texttt{try}\ a_1\ \texttt{with}\ p \rightarrow a_2$, où p est un motif. L'exception ne sera rattrapée que si elle correspond au motif p. Dans le cas contraire, elle continuera à se propager vers le haut.

Exercice 5.6 *(***) Montrer qu'il existe des programmes qui ne terminent pas dans mini-ML avec exceptions (mais sans* `fix`*, sans types concrets, et sans références). (Indication : on pourrait bien sûr définir* $\texttt{exception C of (exn} \rightarrow \texttt{exn)}$ *et utiliser* `C` *comme dans l'exercice 4.8. Il est plus intéressant de considérer la déclaration* $\texttt{exception M of ((int} \rightarrow \texttt{int)} \rightarrow \texttt{(int} \rightarrow \texttt{int))}$ *et de définir à l'aide de cette exception deux fonctions mettant en correspondance les type* $(\texttt{int} \rightarrow \texttt{int}) \rightarrow (\texttt{int} \rightarrow \texttt{int})$ *et* $\texttt{int} \rightarrow \texttt{int}$*.)*

Chapitre 6

Les enregistrements extensibles

Les enregistrements déclarés à la Caml, tels que nous les avons décrits dans la section 4.4.2, souffrent de plusieurs limitations :

- Les types enregistrements doivent être déclarés avant usage.
- Une étiquette e ne peut appartenir à plusieurs types enregistrements. Sinon, la fonction $\texttt{fun}\ x \to x.e$ aurait plusieurs types incomparables (un pour chaque type enregistrement déclaré avec une étiquette e), et cela détruit la propriété de types principaux.
- On ne peut pas construire un enregistrement "incrémentalement", c'est-à-dire en ajoutant un ou plusieurs champs à un enregistrement existant.

Nous allons maintenant étudier un système de types pour enregistrements avec les traits suivants :

- Enregistrements *polymorphes* (ou encore *flexibles*) : on peut définir et typer une fonction d'accès $\texttt{fun}\ x \to x.e$ capable de s'appliquer à tout enregistrement possédant un champ nommé e.
- Enregistrements *extensibles* : on peut définir et typer une fonction d'extension $\texttt{fun}\ x \to \texttt{fun}\ v \to x@\{e = v\}$ qui renvoie un enregistrement identique à x, mais auquel on a ajouté un champ e contenant la valeur v.

6.1 Sémantique à réduction

Soit $\mathcal{E}$ un ensemble dénombrable d'étiquettes. Nous utiliserons la méta-variable L pour dénoter un ensemble fini d'étiquettes.

Nous ajoutons à mini-ML les constructions suivantes :

Expressions : $a ::= \ldots \mid \{(e = a_e)_{e \in L}\}$

Opérateurs : $op ::= \ldots \mid \texttt{proj}_e \mid \texttt{exten}_e$

Valeurs : $v ::= \ldots \mid \{(e = v_e)_{e \in L}\}$

L'expression $\{(e = a_e)_{e \in L}\}$ construit un enregistrement dont les champs sont ceux contenus dans L et où la valeur du champ e est la valeur de l'expression a_e.

On note $a.e$ pour l'application d'opérateur $\texttt{proj}_e(a)$. Cette expression renvoie la valeur associée à e dans l'enregistrement a.

On note $a_1@\{e = a_2\}$ pour l'application d'opérateur $\texttt{exten}_e(a_1, a_2)$. Cette expression renvoie un enregistrement identique à a_1, sauf pour le champ e qui devient associé à a_2.

Les règles de réduction pour ces opérateurs sont :

$$\begin{array}{rcll} \{(e = v_e)_{e \in L}\}.e & \xrightarrow{\varepsilon} & v_e & \text{si } e \in L \\ \{(e = v_e)_{e \in L}\}@\{e' = w\} & \xrightarrow{\varepsilon} & \{(e = v'_e)_{e \in L \cup \{e'\}}\} & \text{où } v' = v[e' \mapsto w] \end{array}$$

La seconde règle s'applique que e' soit présente ou non dans l'enregistrement qu'on étend, c'est-à-dire dans le domaine de la famille v : si oui, la valeur w remplace la valeur précédemment liée à e' ; si non, l'enregistrement résultat a une étiquette de plus. On parle dans ce cas d'extension *libre* d'enregistrement. On pourrait également définir une opération d'extension *stricte*, où l'étiquette ajoutée ne doit pas être déjà présente dans l'enregistrement initial.

Les contextes d'évaluation E s'étendent naturellement aux enregistrements :

Contextes d'évaluation : $E ::= \ldots \mid \{(e = a_e)_{e \in L}; e = E\}$

La définition ci-dessus ne spécifie pas l'ordre d'évaluation des champs. On pourrait souhaiter le faire, par exemple en supposant fixé un ordre arbitraire sur les étiquettes. (Dans la mesure où nous avons défini une expression enregistrement comme une *fonction* qui à chaque étiquette associe une sous-expression, l'ordre dans lequel les champs apparaissaient dans la version textuelle du programme a été perdu.)

6.2 Typage simplifié des enregistrements extensibles

Pour introduire le typage des enregistrements, nous allons supposer que l'ensemble des étiquettes est fini, connu, et suffisamment petit pour qu'il soit raisonnable de les énumérer tous dans un type d'enregistrement. Supposons par exemple trois étiquettes `e`, `f`, `g`. On définit l'algèbre de types suivante :

Types :

$\tau ::= \alpha \mid T \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$	comme précédemment
$\mid \{\mathtt{e} : \tau_1;\ \mathtt{f} : \tau_2;\ \mathtt{g} : \tau_3\}$	type d'enregistrement
$\mid \mathtt{Abs}$	le champ est absent (indéfini)
$\mid \mathtt{Pre}\ \tau$	le champ est présent (défini) avec le type τ

Un type d'enregistrement $\{\mathtt{e} : \tau_1;\ \mathtt{f} : \tau_2;\ \mathtt{g} : \tau_3\}$ liste pour chaque étiquette `e`, `f`, `g` une indication de présence τ pour cette étiquette. Si $\tau_i = \mathtt{Abs}$, l'étiquette correspondante est indéfinie dans l'enregistrement. Si $\tau_i = \mathtt{Pre}\ \tau$, l'étiquette est définie dans l'enregistrement et contient une valeur de type τ. Enfin, τ_i peut également être une variable de type α, ce qui nous offre une flexibilité essentielle : cela permet en effet d'utiliser un enregistrement sans nécessairement connaître exactement la liste de ses champs présents. (Notons que d'autres valeurs pour τ_i, p.ex. $\tau_i = \mathtt{int}$, n'auraient aucun sens ; nous verrons plus loin comment les éviter par introduction de *sortes*.)

Exemples :
$\{\mathtt{e} : \mathtt{Pre\ int};\ \mathtt{f} : \mathtt{Abs};\ \mathtt{g} : \mathtt{Abs}\}$ est le type des enregistrements à un champ `e` de type `int`.
$\{\mathtt{e} : \mathtt{Pre\ bool};\ \mathtt{f} : \mathtt{Abs};\ \mathtt{g} : \mathtt{Pre\ int}\}$ est le type des enregistrements à deux champs, `e` de type `bool` et `g` de type `int`.
$\{\mathtt{e} : \mathtt{Abs};\ \mathtt{f} : \mathtt{Abs};\ \mathtt{g} : \mathtt{Abs}\}$ est le type de l'enregistrement vide.
$\{\mathtt{e} : \alpha_1;\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \alpha_3\} \to \{\mathtt{e} : \mathtt{Pre\ int};\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \alpha_3\}$ est le type d'une fonction qui prend n'importe quel enregistrement en paramètre et l'étend avec un champ `e` de type `int`.

Règles de typage : Nous pouvons maintenant exprimer la règle de typage qui gouverne les expressions enregistrements ainsi que les schémas de types associés aux nouveaux opérateurs :

$$\frac{\forall e \in L \quad \Gamma \vdash a_e : \tau_e}{\Gamma \vdash \{(e = a_e)_{e \in L}\} : \{(e : \mathtt{Pre}\ \tau_e)_{e \in L}; (e : \mathtt{Abs})_{e \notin L}\}}$$

$$\begin{array}{rcl}
\mathtt{proj_e} & : & \forall \alpha, \alpha_1, \alpha_2.\ \{\mathtt{e} : \mathtt{Pre}\ \alpha;\ \mathtt{f} : \alpha_1;\ \mathtt{g} : \alpha_2\} \to \alpha \\
\mathtt{proj_f} & : & \forall \alpha, \alpha_1, \alpha_2.\ \{\mathtt{e} : \alpha_1;\ \mathtt{f} : \mathtt{Pre}\ \alpha;\ \mathtt{g} : \alpha_2\} \to \alpha \\
\mathtt{proj_g} & : & \forall \alpha, \alpha_1, \alpha_2.\ \{\mathtt{e} : \alpha_1;\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \mathtt{Pre}\ \alpha\} \to \alpha \\
\mathtt{exten_e} & : & \forall \alpha, \alpha_1, \alpha_2, \alpha_3.\ \{\mathtt{e} : \alpha_1;\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \alpha_3\} \times \alpha \to \{\mathtt{e} : \mathtt{Pre}\ \alpha;\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \alpha_3\} \\
\mathtt{exten_f} & : & \forall \alpha, \alpha_1, \alpha_2, \alpha_3.\ \{\mathtt{e} : \alpha_1;\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \alpha_3\} \times \alpha \to \{\mathtt{e} : \alpha_1;\ \mathtt{f} : \mathtt{Pre}\ \alpha;\ \mathtt{g} : \alpha_3\} \\
\mathtt{exten_g} & : & \forall \alpha, \alpha_1, \alpha_2, \alpha_3.\ \{\mathtt{e} : \alpha_1;\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \alpha_3\} \times \alpha \to \{\mathtt{e} : \alpha_1;\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \mathtt{Pre}\ \alpha\}
\end{array}$$

Les types donnés ci-dessus pour l'extension correspondent à l'extension libre. Ainsi, $\mathtt{exten_e}$ peut être utilisé aussi bien avec le type

$$\{\mathtt{e} : \mathtt{Abs} \ldots\} \times \tau \to \{\mathtt{e} : \mathtt{Pre}\ \tau;\ \ldots\}$$

qu'avec le type

$$\{\mathtt{e} : \mathtt{Pre}\ \tau' \ldots\} \times \tau \to \{\mathtt{e} : \mathtt{Pre}\ \tau;\ \ldots\}$$

Le premier correspond à une extension d'un enregistrement qui n'a pas de champ `e` ; le second, au remplacement d'un champ `e` existant. Si l'on veut se restreindre à l'extension stricte, il faut utiliser des types moins polymorphes de la forme suivante :

$$\mathtt{exten_e} \quad : \quad \forall \alpha, \alpha_2, \alpha_3.\ \{\mathtt{e} : \mathtt{Abs};\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \alpha_3\} \times \alpha \to \{\mathtt{e} : \mathtt{Pre}\ \alpha;\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \alpha_3\}$$

Exercice 6.1 *(*) Pourquoi avons-nous exigé qu'un type enregistrement contienne une information à propos de* toutes *les étiquettes existantes ? Si l'on s'était contenté de donner, dans un type enregistrement, l'ensemble des champs effectivement présents ainsi que leur type, aurions-nous pu associer un schéma de types satisfaisant aux opérateurs d'accès et d'extension ?*

Exercice 6.2 *(*) On peut coder l'approche décrite ci-dessus en ML "de base" en définissant les types concrets suivants :*

```
type ('a, 'b, 'c) enreg = { e : 'a ; f : 'b ; g : 'c }
type 'a pre = Present of 'a
type abs = Absent
```

Montrer comment définir l'enregistrement vide ; l'enregistrement $\{e = a\}$; les fonctions d'accès ; les fonctions d'extension.

6.3 Typage avec rangées

L'approche de la section 6.2 est simple mais fort limitée. En effet, chaque type enregistrement a une taille proportionnelle au nombre d'étiquettes employées dans l'ensemble du programme, et non proportionnelle à la taille de l'enregistrement qu'il représente. De plus, cette approche exige que l'ensemble des étiquettes potentiellement employées soit connu, ce qui empêche de typer séparément chaque unité de compilation. L'approche est donc lourde et non modulaire.

La difficulté provient du fait que nous avons représenté des fonctions des étiquettes vers les types *in extenso*, c'est-à-dire comme une liste où chaque étiquette doit apparaître explicitement avec son type associé. Pour résoudre la difficulté, il nous faut imaginer une autre représentation de ces fonctions. Les *rangées* en sont une possible.

6.3.1 L'algèbre de types

La principale innovation des rangées est d'introduire une notion de *modèle* pour les champs qui ne sont pas explicitement mentionnés dans un type enregistrement. Ce modèle peut être $\partial\mathtt{Abs}$ pour dire que tous les autres champs sont absent, ou bien une variable α qui représente un ensemble arbitraire de noms de champs et d'infos de présence. Un type d'enregistrement est alors de la forme générale $\{$*rangée*$\}$ ou *rangée* se compose de zéro, une ou plusieurs déclarations de champs teminées par un modèle. Ainsi, $\{e : \mathtt{Pre\ int};\ \partial\mathtt{Abs}\}$ représente les enregistrements dont le champ e contient un entier et dont tous les autres champs sont indéfinis. L'algèbre de types devient :

Types :	$\tau ::= \alpha \mid T \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$	comme précédemment
	$\mid \{\tau\}$	type d'enregistrement
	$\mid \partial\mathtt{Abs}$	la rangée vide
	$\mid e : \tau_1;\ \tau_2$	la rangée contenant $e : \tau_1$ plus ce que contient la rangée τ_2
	$\mid \mathtt{Abs}$	le champ est absent (indéfini)
	$\mid \mathtt{Pre}\ \tau$	le champ est présent (défini) avec le type τ

Les types sont identifiés modulo les deux équations suivantes :

$$\begin{array}{rcll} e_1 : \tau_1;\ e_2 : \tau_2;\ \tau & = & e_2 : \tau_2;\ e_1 : \tau_1;\ \tau & \text{(commutativité)} \\ \partial\mathtt{Abs} & = & e : \mathtt{Abs};\ \partial\mathtt{Abs} & \text{(absorption)} \end{array}$$

L'équation de commutativité exprime que l'ordre dans lequel les étiquettes apparaissent dans une rangée n'a pas d'importance. L'équation d'absoption capture l'intuition que $\partial\mathtt{Abs}$ représente une infinité d'étiquettes, toutes absentes.

Exemple : les deux types d'enregistrement suivants sont égaux :

$$\{e_1 : \mathtt{Pre\ int};\ \partial\mathtt{Abs}\} \text{ et } \{e_2 : \mathtt{Abs};\ e_1 : \mathtt{Pre\ int};\ \partial\mathtt{Abs}\}$$

6.3.2 Règles de typage

La construction d'enregistrements se type comme suit :

$$\frac{\forall e \in L \quad \Gamma \vdash a_e : \tau_e}{\Gamma \vdash \{(e = a_e)_{e \in L}\} : \{(e : \mathtt{Pre}\ \tau_e)_{e \in L};\ \partial\mathtt{Abs}\}}$$

Chaque champ $e \in L$ est présent avec le type τ_e ; tout autre champ est absent, d'où le $\partial\mathtt{Abs}$ à la fin de la rangée. Notons que la notation $\{(e = a_e)_{e \in L}\} : \{(e : \mathtt{Pre}\ \tau_e)_{e \in L};\ \partial\mathtt{Abs}\}$ ne spécifie pas l'ordre d'apparition des champs de L dans le type ; mais cet ordre n'a pas d'importance, puisque les types sont considérés comme égaux modulo commutation.

Voici à présent le schéma de types associé à l'opérateur de projection :

$$\mathtt{proj}_e \quad : \quad \forall \alpha, \beta.\ \{e : \mathtt{Pre}\ \alpha;\ \beta\} \to \alpha$$

Par instanciation de β et application de l'axiome de commutativité, on peut appliquer $\mathtt{proj}_e$ à tout enregistrement dont le type contient un champ e marqué présent : un tel enregistrement a un type de la forme $\{\ldots;\ e : \mathtt{Pre}\ \tau;\ \ldots\}$, qui par commutativité est aussi de la forme $\{e : \mathtt{Pre}\ \tau;\ \tau'\}$, qui est une instance du type argument de $\mathtt{proj}_e$ (avec $\alpha \leftarrow \tau$ et $\beta \leftarrow \tau'$).

Enfin, voici le schéma de types associé à l'opérateur d'extension :

$$\mathtt{exten}_e \quad : \quad \forall \alpha, \beta, \gamma.\ \{e : \alpha;\ \beta\} \times \gamma \to \{e : \mathtt{Pre}\ \gamma;\ \beta\}$$

Pour l'extension stricte, on prendrait :

$$\mathtt{exten}_e \quad : \quad \forall \beta, \gamma.\ \{e : \mathtt{Abs};\ \beta\} \times \gamma \to \{e : \mathtt{Pre}\ \gamma;\ \beta\}$$

Exemples : la fonction

```
fun r → (r.a, r.b)
```

a le type $\{a : \mathtt{Pre}\ \alpha;\ b : \mathtt{Pre}\ \beta;\ \gamma\} \to \alpha \times \beta$. En effet, on a $\{a : \mathtt{Pre}\ \alpha;\ b : \mathtt{Pre}\ \beta;\ \gamma\} = \{b : \mathtt{Pre}\ \beta;\ a : \mathtt{Pre}\ \alpha;\ \gamma\}$ par l'équation de commutativité, donc les deux projections sont bien typées.

Voici maintenant quelques exemples de typage de l'extension libre. Premier exemple : ajout d'un nouveau champ.

$$\underbrace{\{\mathtt{a} = 1;\ \mathtt{b} = \mathtt{true}\}}_{r} @ \{\mathtt{c} = \text{"}\mathtt{foo}\text{"}\}$$

L'enregistrement r a le type $\{\mathtt{a} : \mathtt{Pre\ int};\ \mathtt{b} : \mathtt{Pre\ bool};\ \partial\mathtt{Abs}\}$. Utilisant l'absorption puis la commutativité, on transforme ce type en le type équivalent

$$\{\mathtt{c} : \mathtt{Abs};\ \mathtt{a} : \mathtt{Pre\ int};\ \mathtt{b} : \mathtt{Pre\ bool};\ \partial\mathtt{Abs}\}$$

Ce type est une instance du type argument de $\mathtt{exten}_\mathtt{c}$, et on obtient comme type du résultat

$$\{\mathtt{c} : \mathtt{Pre\ string};\ \mathtt{a} : \mathtt{Pre\ int};\ \mathtt{b} : \mathtt{Pre\ bool};\ \partial\mathtt{Abs}\}$$

Second exemple : redéfinition d'un champ existant.

$$\underbrace{\{\mathtt{a} = 1;\ \mathtt{b} = \mathtt{true}\}}_{r} @ \{\mathtt{b} = \text{"}\mathtt{foo}\text{"}\}$$

Il faut voir r avec le type $\{\mathtt{b} : \mathtt{Pre\ bool};\ \mathtt{a} : \mathtt{Pre\ int};\ \partial\mathtt{Abs}\}$, et on obtient comme type du résultat de l'extension

$$\{\mathtt{b} : \mathtt{Pre\ string};\ \mathtt{a} : \mathtt{Pre\ int};\ \partial\mathtt{Abs}\}$$

Dernier exemple : extension dans une fonction polymorphe.

$$\mathtt{fun}\ r \to r @ \{\mathtt{a} = 1\}$$

Il faut instancier τ par $\mathtt{int}$ dans le schéma de type pour $\mathtt{exten}_\mathtt{a}$. On obtient le type suivant pour la fonction :

$$\{\mathtt{a} : \alpha;\ \beta\} \to \{\mathtt{a} : \mathtt{Pre\ int};\ \beta\}$$

6.3.3 Sortes

L'algèbre de types enregistrements que nous venons d'introduire contient un certain nombre de types absurdes, comme par exemple $\partial\texttt{Abs} \to \partial\texttt{Abs}$ ou $\texttt{Abs} \times \texttt{Pre}\ \tau$ ou $\alpha \to \texttt{Pre}\ \alpha$. Pour les éviter, il faut s'imposer une certaine discipline dans l'utilisation des expressions de types, afin de ne pas confondre :

- les types "normaux", qui peuvent apparaître comme types d'expressions du langage, p.ex. `int` ou $\texttt{int} \to \texttt{bool}$;
- les rangées de types, qui peuvent apparaître à l'intérieur d'un type enregistrement $\{\dots\}$, p.ex. $\partial\texttt{Abs}$ ou $(e : \texttt{Abs};\ \dots)$.
- les infos de présence `Abs` et $\texttt{Pre}\ \tau$, qui peuvent apparaître comme annotation d'une étiquette dans une rangée de type.

Plus subtilement, l'algèbre de types contient aussi des types contradictoires, comme par exemple $\{a : \texttt{Pre int};\ a : \texttt{Abs};\ \partial\texttt{Abs}\}$ (a ne peut pas être à la fois absent et présent), ou $\{a : \texttt{Pre int};\ a : \texttt{Pre bool};\ \partial\texttt{Abs}\}$ (a ne peut pas être présent avec deux types différents). Nous souhaitons considérer de tels types comme mal formés. (Notons toutefois qu'on pourrait imaginer une approche différente, dans laquelle ces types auraient un sens ; il faudrait par exemple limiter la commutation à des étiquettes distinctes, de sorte que si une étiquette est répétée, c'est l'occurrence de gauche qui fait foi.) Nous allons donc faire en sorte d'imposer l'invariant suivant : *une étiquette e ne peut apparaître plus d'une fois dans une même rangée.* Ainsi, on pourra parler sans ambiguïté de l'information associée à l'étiquette e dans une rangée τ ; ce sera l'unique type τ_1 tel que la rangée τ puisse s'écrire $e : \tau_1;\ \tau_2$.

Cet invariant est cependant difficile à maintenir, en particulier par substitution de variables de rangées. Exemple : le type $\tau = \{a : \texttt{Pre int};\ \rho\}$ satisfait l'invariant, ainsi que la rangée $\varphi = a : \texttt{Pre bool};\ \partial\texttt{Abs}$. Cependant, la substitution $\tau[\rho \leftarrow \varphi]$ ne satisfait pas l'invariant. Une manière rigoureuse de garantir cet invariant, et plus généralement d'empêcher l'apparition de types absurdes, est d'utiliser des *sortes* (*sorts* ou *kinds* en anglais). Les sortes sont aux types ce que les types sont aux programmes : de même que les types éliminent des programmes absurdes tels que 1 2, les sortes éliminent des types absurdes tels que $\partial\texttt{Abs} \to \partial\texttt{Abs}$ ou $\{a : \texttt{Pre int};\ a : \texttt{Pre bool};\ \partial\texttt{Abs}\}$.

On va donc définir une relation de "sortage" (*kinding*) $\vdash \tau :: \kappa$, qui signifie "le type τ est bien formé et de la sorte κ". L'algèbre des sortes pour les enregistrements est :

Sortes : $\kappa ::= \texttt{TYPE} \mid \texttt{PRE} \mid \texttt{R}(\{e_1, \dots, e_n\})$

`TYPE` est la sorte des types (d'expressions) bien formés. `PRE` est celle des infos de présence bien formées. Enfin, $\texttt{R}(L)$, où L est un ensemble d'étiquettes, est la sorte des rangées bien formées et qui *n'associent pas* d'information aux étiquettes $e \in L$. En d'autres termes, une rangée de sorte $\texttt{R}(L)$ représente une fonction dont le domaine est $\mathcal{E} \setminus L$. Les règles de sortage sont les suivantes :

$$\vdash \alpha :: K(\alpha) \qquad \vdash T :: \texttt{TYPE} \qquad \frac{\vdash \tau_1 :: \texttt{TYPE} \quad \vdash \tau_2 :: \texttt{TYPE}}{\vdash \tau_1 \to \tau_2 :: \texttt{TYPE}} \qquad \frac{\vdash \tau_1 :: \texttt{TYPE} \quad \vdash \tau_2 :: \texttt{TYPE}}{\vdash \tau_1 \times \tau_2 :: \texttt{TYPE}}$$

$$\frac{\vdash \tau :: \texttt{R}(\emptyset)}{\vdash \{\tau\} :: \texttt{TYPE}} \qquad \vdash \partial\texttt{Abs} :: \texttt{R}(L) \qquad \frac{e \notin L \quad \vdash \tau_1 :: \texttt{PRE} \quad \vdash \tau_2 :: \texttt{R}(L \cup \{e\})}{\vdash (e : \tau_1;\ \tau_2) :: \texttt{R}(L)}$$

$$\vdash \texttt{Abs} :: \texttt{PRE} \qquad \frac{\vdash \tau :: \texttt{TYPE}}{\vdash \texttt{Pre}\ \tau :: \texttt{PRE}}$$

Pour l'axiome $\vdash \alpha :: K(\alpha)$, on suppose donnée une fonction K qui associe à chaque variable α sa sorte $K(\alpha)$. Ainsi, chaque variable de type a une sorte unique quelle que soit l'expression de type dans laquelle elle apparaît. On suppose bien sûr qu'il existe une infinité de variables de chaque sorte. (Cet artifice technique simplifie les règles de sortage, puisque l'association entre variables et sortes devient ainsi implicite, au lieu d'apparaître explicitement, sous forme d'un environnement, dans chaque règle.)

La constante $\partial\mathtt{Abs}$ est polymorphe : elle admet toutes les sortes de la forme $\mathtt{R}(L)$. Du coup, l'équation d'absorption $\partial\mathtt{Abs} = e : \mathtt{Abs};\ \partial\mathtt{Abs}$ pose un problème potentiel : son membre gauche a toutes les sortes $\mathtt{R}(L)$, alors que son membre droit admet la sorte $\mathtt{R}(L)$ seulement si L ne contient pas e. Cela signifie qu'on doit être prudent et n'effectuer la réécriture de gauche à droite que lorsque le contexte le permet, c'est-à-dire que lorsque cette réécriture préserve les sortes. Le lecteur pourra vérifier qu'en effet l'algorithme d'unification n'appliquera cette équation que dans des contextes qui le permettent.

Le système de sortes garantit l'invariant qu'une étiquette apparaît au plus une fois dans une même rangée. En effet, supposons que l'étiquette e apparaisse deux fois dans une rangée τ bien sortée de sorte $\mathtt{R}(L)$. Par commutativité, on aurait $\tau = e : \tau_1;\ e : \tau_2;\ \tau'$. Comme $\vdash \tau :: \mathtt{R}(L)$, il faudrait $(e : \tau_2;\ \tau') :: \mathtt{R}(L \cup \{e\})$, mais ceci est impossible car $e \in L \cup \{e\}$.

Pour assurer que tous les types intervenant dans une dérivation de typage sont bien sortés, il faut modifier légèrement la définition du système. D'abord, toutes les substitutions considérées doivent *préserver les sortes*. On dit qu'une substitution θ préserve les sortes si et seulement si, pour toute variable α, on a $\vdash \theta(\alpha) :: K(\alpha)$. Il est facile de voir que si θ préserve les sortes, alors $\vdash \tau :: \kappa$ implique $\vdash \theta(\tau) :: \kappa$. Ensuite, tout schéma de types $\forall\vec{\alpha}.\ \tau$ doit vérifier $\vdash \tau :: \mathtt{TYPE}$.

Une première conséquence de ces restrictions est que, si $\sigma \leq \tau$, alors $\vdash \tau :: \mathtt{TYPE}$. Cela provient du fait que le corps du schéma σ admet la sorte TYPE et du fait que la définition de la relation d'instanciation $\leq$ fait appel à une substitution, laquelle devra maintenant préserver les sortes. Une seconde conséquence est que tout type apparaissant dans un environnement de typage doit admettre la sorte TYPE. Cela provient du fait qu'un environnement est en réalité une collection de *schémas* de types ; les types y sont considérés comme des schémas (non quantifiés) et doivent donc adhérer à la même restriction quant aux sortes.

Exercice 6.3 *(*) Vérifier que, une fois adoptées les restrictions ci-dessus, tout jugement dérivable* $\Gamma \vdash a : \tau$ *vérifie* $\vdash \tau :: \mathtt{TYPE}$. *On n'oubliera pas de considérer le cas où a est une expression enregistrement et celui où a est un opérateur d'accès ou d'extension.*

6.4 Sûreté du typage

Pour montrer la sûreté du typage des enregistrements, il faut d'abord prouver des lemmes techniques sur la relation de typage $E \vdash a : \tau$ analogues aux lemmes 1.2, 1.3, 1.4 et 2.2. Il ne s'agit pas juste d'ajouter un cas aux preuves pour traiter la nouvelle règle (record). En effet, nous considérons maintenant les types modulo une théorie équationnelle (les axiomes de commutativité et d'absorption), et pour être tout à fait rigoureux, il faut vérifier soigneusement que tous nos définitions et énoncés sont compatibles avec ces axiomes. Nous ne le ferons pas explicitement dans ces notes. Contentons-nous de noter que ces axiomes préservent l'ensemble des variables libres dans un type, ce qui garantit que la fonction $\mathcal{L}$ reste bien définie. C'est bien sûr d'une grande importance, puisque la règle de généralisation dépend crucialement de cette notion.

Une fois ces résultats obtenus, la sûreté du typage se montre comme d'habitude en "paramétrant" la preuve du chapitre 2 pour l'adapter aux nouveaux opérateurs et à la nouvelle algèbre de valeurs. Les étapes de la preuve sont :

– Montrer que l'hypothèse (H1) est vérifiée pour $\texttt{proj}_e$ et $\texttt{exten}_e$.
– Montrer que si $\emptyset \vdash v : \{\tau\}$, alors v est une valeur enregistrement, et que si de plus $\tau = e : \texttt{Pre}\ \tau_1;\ \tau_2$, alors v contient un champ e associé à une valeur de type τ_1.
– Montrer que l'hypothèse (H2) est vérifiée pour $\texttt{proj}_e$ et $\texttt{exten}_e$.

Exercice 6.4 *(*)/(**) Rédiger ces trois étapes.*

6.5 Inférence de types

Pour l'inférence de types, nous adaptons l'approche du chapitre 3 (algorithme W de Damas-Milner-Tofte et unification entre types) à la nouvelle algèbre de types.

6.5.1 Unification

De manière générale, l'ajout d'une théorie équationnelle à une algèbre libre de termes (telle que l'algèbre des types de mini-ML) peut changer radicalement la nature et les propriétés des problèmes d'unification. Ainsi, l'ajout d'axiomes d'associativité et de commutativité pour un opérateur binaire + transforme l'unification en résolution d'équations entre mots, et fait perdre l'existence d'unificateurs principaux.

Le cas des axiomes de commutativité et d'absorption dans les types des enregistrements est heureusement plus simple, quoique non trivial. En particulier, il ne suffit plus d'examiner les symboles de tête de deux types à unifier et de déclarer qu'ils ne sont pas unifiables si ces deux symboles sont différents.

Exemples : les deux types $\partial\texttt{Abs}$ et $(e : \alpha;\ \beta)$ n'ont pas le même symbole de tête ($\partial\texttt{Abs}$ pour l'un, $(e : \cdot;\ \cdot)$ pour l'autre), mais sont pourtant unifiables en prenant $\alpha \leftarrow \texttt{Abs}$ et $\beta \leftarrow \partial\texttt{Abs}$, et en appliquant l'axiome d'absorption.

De même, les types $e : \texttt{Pre int};\ \alpha$ et $f : \texttt{Pre bool};\ \beta$ peuvent sembler non-unifiables au premier abord (car l'un commence par $e : \dots$ et l'autre par $f : \dots$), mais sont pourtant unifiables par la substitution

$$\alpha \leftarrow f : \texttt{Pre bool};\ \tau \qquad \beta \leftarrow e : \texttt{Pre int};\ \tau$$

où τ est un type arbitraire. En effet, en appliquant cette substitution aux deux types, on obtient les types

$$e : \texttt{Pre int};\ f : \texttt{Pre bool};\ \tau \qquad \text{et} \qquad f : \texttt{Pre bool};\ e : \texttt{Pre int};\ \tau$$

qui sont bien égaux modulo commutativité.

Le point délicat, dans ce second exemple, est le fait que le type τ peut être choisi de façon arbitraire ; il ne provient pas de l'équation que l'on cherchait à résoudre. Pour obtenir un unificateur principal, il faudra donc choisir pour τ une *variable fraîche*, c'est-à-dire une variable non présente jusqu'ici dans le système d'équations. L'algorithme d'unification pourra donc dorénavant faire apparaître de nouvelles variables. (Techniquement, nous utiliserons la variable globale V déjà introduite pour l'algorithme W pour représenter l'ensemble des variables encore non utilisées.) La preuve de terminaison de l'algorithme devient alors plus complexe : le nombre de variables du problème pouvant augmenter au cours de la résolution, on sera obligé de s'appuyer sur le fait que le système d'équations est *bien sorté* pour garantir la terminaison de l'algorithme. Un système est bien sorté si et seulement si chaque équation $\tau_1 \stackrel{?}{=} \tau_2$ admet l'existence d'une sorte κ telle que $\vdash \tau_1 :: \kappa$ et $\vdash \tau_2 :: \kappa$.

Algorithme d'unification : Soit C un système d'équations bien sorté. L'unificateur principal $\mathtt{mgu}(C)$ se calcule de la façon suivante. Les premiers cas sont identiques à ceux de la section 3.2.3, où nous avons décrit l'unification dans une algèbre libre, c'est-à-dire en l'absence de théorie équationnelle :

$$\begin{array}{rcl}
\mathtt{mgu}(\emptyset) & = & id \\
\mathtt{mgu}(\{\alpha \stackrel{?}{=} \alpha\} \cup C & = & \mathtt{mgu}(C) \\
\mathtt{mgu}(\{\alpha \stackrel{?}{=} \tau\} \cup C) & = & \mathtt{mgu}(C[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau] \text{ si } \alpha \text{ n'est pas libre dans } \tau \\
\mathtt{mgu}(\{\tau \stackrel{?}{=} \alpha\} \cup C) & = & \mathtt{mgu}(C[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau] \text{ si } \alpha \text{ n'est pas libre dans } \tau \\
\mathtt{mgu}(\{\tau_1 \to \tau_2 \stackrel{?}{=} \tau_1' \to \tau_2'\} \cup C) & = & \mathtt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_1'; \tau_2 \stackrel{?}{=} \tau_2'\} \cup C) \\
\mathtt{mgu}(\{\tau_1 \times \tau_2 \stackrel{?}{=} \tau_1' \times \tau_2'\} \cup C) & = & \mathtt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_1'; \tau_2 \stackrel{?}{=} \tau_2'\} \cup C) \\
\mathtt{mgu}(\{\ \{\tau_1\} \stackrel{?}{=} \{\tau_2\}\ \} \cup C) & = & \mathtt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_2\} \cup C) \\
\mathtt{mgu}(\{\mathtt{Abs} \stackrel{?}{=} \mathtt{Abs}\} \cup C) & = & \mathtt{mgu}(C) \\
\mathtt{mgu}(\{\mathtt{Pre}\ \tau_1 \stackrel{?}{=} \mathtt{Pre}\ \tau'\} \cup C) & = & \mathtt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_2\} \cup C)
\end{array}$$

Les cas correspondant à l'unification de deux rangées sont plus intéressants :

$$\begin{array}{rcl}
\mathtt{mgu}(\{\partial\mathtt{Abs} \stackrel{?}{=} \partial\mathtt{Abs}\} \cup C) & = & \mathtt{mgu}(C) \\
\mathtt{mgu}(\{e : \tau;\ \tau' \stackrel{?}{=} \partial\mathtt{Abs}\} \cup C) & = & \mathtt{mgu}(\{\tau \stackrel{?}{=} \mathtt{Abs};\ \tau' \stackrel{?}{=} \partial\mathtt{Abs}\} \cup C) \\
\mathtt{mgu}(\{\partial\mathtt{Abs} \stackrel{?}{=} e : \tau;\ \tau'\} \cup C) & = & \mathtt{mgu}(\{\tau \stackrel{?}{=} \mathtt{Abs};\ \tau' \stackrel{?}{=} \partial\mathtt{Abs}\} \cup C) \\
\mathtt{mgu}(\{e : \tau_1;\ \tau_1' \stackrel{?}{=} e : \tau_2;\ \tau_2'\} \cup C) & = & \mathtt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_2;\ \tau_1' \stackrel{?}{=} \tau_2'\} \cup C) \\
\mathtt{mgu}(\{(e : \tau_1;\ \tau_1') \stackrel{?}{=} (f : \tau_2;\ \tau_2')\} \cup C) & = & \mathtt{do}\ \alpha = \mathtt{fresh} \\
(e \neq f) & & \mathtt{mgu}(\{\tau_1' \stackrel{?}{=} (f : \tau_2;\ \alpha);\ \tau_2' \stackrel{?}{=} (e : \tau_1;\ \alpha)\} \cup C)
\end{array}$$

Dans le dernier cas, on fait appel à la fonction auxiliaire `fresh`, définie dans la section 3.3.1, pour obtenir une variable fraîche α. Techniquement, il faut de plus que la sorte de α soit choisie de façon appropriée pour que le système résultant reste bien sorté ; plus précisément, si $\mathtt{R}(L)$ est la sorte commune à $(e : \tau_1;\ \tau_1')$ et $(f : \tau_2;\ \tau_2')$, on choisit α de la sorte $\mathtt{R}(L \cup \{e; f\})$. Cette remarque est purement formelle, car lorsqu'on implémente l'algorithme, il n'est pas nécessaire de connaître la sorte des variables manipulées. Aucune décision ne dépend en effet de la sorte d'une variable ou d'un type.

Le seul cas où une vérification de sortes sera nécessaire concerne les types fournis par le programmeur, p.ex. dans des contraintes de types $(a : \tau)$ ou dans des déclarations de types algébriques. Il faut vérifier que ceux-ci sont bien sortés pour pouvoir les introduire dans les équations engendrées.

Exercice 6.5 *(**)/(***) Proposer un algorithme de vérification de sortes. (Indication : l'algorithme prend en entrée un type τ, une sorte κ attendue pour ce type, et un sortage partiel K pour les variables déjà rencontrées ; il produit en sortie un sortage de variables K' qui est K étendu avec les sortes des variables rencontrées pour la première fois dans τ.)*

Le fait que le système C est bien sorté garantit que la substitution $\mathtt{mgu}(C)$ préserve les sortes.

Exercice de programmation 6.1 *Implémenter la fonction* `mgu`.

6.5.2 Inférence de types

La modification apportée à l'algorithme d'unification est suffisante ; l'algorithme d'inférence proprement dit, à savoir l'algorithme W, reste inchangé, à condition bien sûr d'ajouter un cas pour traiter les expressions enregistrements. La preuve en sera cependant légèrement modifiée, du fait que les propriétés de l'algorithme `mgu` ont changé : celui-ci modifie la variable globale V et renvoie une substitution dont le domaine peut inclure plus de variables que celles présentes dans le système d'équations original.

Exercice de programmation 6.2 *Mettre à jour l'implémentation de l'algorithme W de l'exercice 3.5 pour l'adapter aux enregistrements.*

6.5.3 Une présentation efficace de l'unification

L'algorithme d'unification donné dans la section 6.5.1 est simple. Néanmoins, il a deux inconvénients : d'une part, il manipule et compose explicitement des substitutions, et il n'est pas clair, d'après cette spécification, comment on peut obtenir une implémentation efficace ; d'autre part, il fait appel à une variable globale V pour allouer des variables fraîches, ce qui, d'un point de vue purement formel, n'est pas la façon la plus élégante de procéder, et ne correspond pas nécessairement non plus à ce que l'on fera en machine. Nous allons maintenant développer une seconde présentation de l'algorithme d'unification, plus élégante et plus proche de ce qu'on implantera réellement.

Le problème soumis à l'algorithme reste une conjonction d'équations à résoudre. Cependant, le résultat de l'algorithme change de forme : au lieu de produire une substitution, l'algorithme va *réécrire* le problème initial en un nouveau problème, lequel sera équivalent d'un point de vue logique et *en forme résolue*. Nous démontrerons que tout problème en forme résolue admet une solution, ce qui garantit que l'algorithme, s'il termine sans échec, a prouvé que le problème initial est soluble.

Nous distinguerons deux sortes d'équations : d'une part celles qui figurent dans le problème initial, et que nous noterons comme précédemment $\tau_1 \stackrel{?}{=} \tau_2$, où τ_1 et τ_2 sont des types ; d'autre part, des équations *résolues* produites par l'algorithme, et qui vérifient des invariants plus forts. Pour cette seconde catégorie, nous parlerons en fait de *multi-équations*, car nous aurons besoin de regrouper plus de deux types au sein d'une même équation. Une multi-équation e sera donc tout simplement un ensemble de types. Pour des raisons de confort, nous écrirons toutefois τ au lieu de $\{\tau\}$ et $e_1 = e_2$ au lieu de $e_1 \cup e_2$. Il en découle que nous pourrons écrire $\tau_1 = \ldots = \tau_n$ au lieu de $\{\tau_1, \ldots, \tau_n\}$ et $\tau = e$ au lieu de $\{\tau\} \cup e$.

Nous noterons $\hat{\tau}$ un type qui n'est pas une variable et dont les sous-termes immédiats (s'il en a) sont des variables. Les termes `int` et $\alpha \to \beta$ en sont des exemples.

Voici la syntaxe des problèmes ou *contraintes* :

$$C \quad ::= \quad \mathbf{true} \mid \mathbf{false} \mid \alpha \stackrel{?}{=} \alpha \mid \hat{\tau} \stackrel{?}{=} \hat{\tau} \mid e \mid C \wedge C \mid \exists \alpha . C$$

Une contrainte est construite à partir d'équations, de multi-équations, et des contraintes élémentaires **true** et **false** à l'aide de la conjonction et de la quantitication existentielle. La variable α est liée dans C par la construction $\exists \alpha . C$. Nous identifierons les contraintes modulo une relation d'*équivalence structurelle*, que l'on peut définir comme la plus petite *congruence* contenant l'α-conversion, la commutativité et l'associativité de la conjonction, le fait que **true** est élément neutre pour la conjonction, le fait que **false** est absorbant pour la conjonction et la quantification existentielle, et l'*extrusion* du quantificateur existentiel (*scope extrusion* en anglais) :

$$(\exists \alpha . C_1) \wedge C_2 \approx \exists \alpha . (C_1 \wedge C_2) \qquad \text{si } \alpha \notin \mathcal{L}(C_2)$$

(Le mot *congruence* indique que l'équivalence structurelle est préservée par la conjonction et par la quantification existentielle.) Cette dernière loi, usuelle en logique, permet de mettre toute contrainte sous forme prénexe, c'est-à-dire sous une forme où tous les quantificateurs existentiels apparaissent en tête. (Cela peut bien sûr nécessiter également des étapes d'α-conversion, en cas de conflit entre un nom libre et un nom lié.) Ainsi, la présence des quantificateurs existentiels n'ajoute en réalité aucune puissance à notre langage de contraintes, mais nous fournit un moyen élégant de disposer de variables locales (c'est-à-dire "d'allouer des variables fraîches") lorsque nous le souhaiterons.

Nous exigeons que les équations à résoudre soient constituées soit de deux variables (p.ex. $\alpha \stackrel{?}{=} \beta$), soit de deux termes construits ($\hat{\tau}_1 \stackrel{?}{=} \hat{\tau}_2$). Cette restriction simplifie la présentation et joue également un rôle dans l'argument de terminaison de l'algorithme. Si on souhaite résoudre une équation entre une variable α et un type $\hat{\tau}$, on placera dans le problème initial la contrainte $\exists\beta.(\alpha \stackrel{?}{=} \beta \wedge \beta = \hat{\tau})$. Cette contrainte, dont la signification logique est équivalente, respecte la restriction que nous avons imposée.

Nous noterons $\bar{\alpha}$ une multi-équation composée uniquement de variables. (Notons que α et $\bar{\alpha}$ sont deux méta-variables distinctes, sans aucun lien.)

L'algorithme exige (et préserve) l'invariant suivant :

> Toute multi-équation e est de la forme $\bar{\alpha}$ ou $\hat{\tau} = \bar{\alpha}$. (Nous appellerons alors $\bar{\alpha}$ les *variables de* e. Notons que ce ne sont pas exactement les variables *libres* de e, car $\hat{\tau}$ peut contenir des variables libres.)
>
> Les variables de deux multi-équations distinctes forment des ensembles distincts.
>
> Si α est libre dans une équation à résoudre, alors il existe une multi-équation de la forme $\alpha = e$.

Le troisième et dernier point de l'invariant n'est pas nécessairement satisfait initialement : typiquement, le problème initial ne contient que des équations à résoudre, et aucune multi-équation. On doit alors allouer une multi-équation à un élément α pour chaque variable α. Cela satisfait l'invariant sans altérer l'ensemble des solutions du problème.

L'algorithme se présente sous forme de règles de réécriture, de la forme $C_1 \to C_2$, où C_1 et C_2 peuvent contenir des méta-variables. La relation de réécriture qui en découle se définit comme la plus petite relation contenant toutes les instances des règles de réécriture, préservée par la conjonction et la quantification existentielle, et close vis-à-vis de l'équivalence structurelle.

Les premières règles de réécriture, caractéristiques de tout algorithme d'unification, sont les suivantes :

$$\alpha_1 \stackrel{?}{=} \alpha_2 \wedge \alpha_1 = \alpha_2 = e \quad \to \quad \alpha_1 = \alpha_2 = e \tag{1}$$
$$\alpha_1 \stackrel{?}{=} \alpha_2 \wedge \alpha_1 = \bar{\alpha}_1 \wedge \alpha_2 = e \quad \to \quad \alpha_1 = \bar{\alpha}_1 = \alpha_2 = e \tag{2}$$
$$\alpha_1 \stackrel{?}{=} \alpha_2 \wedge \alpha_1 = e \wedge \alpha_2 = \bar{\alpha}_2 \quad \to \quad \alpha_1 = e = \alpha_2 = \bar{\alpha}_2 \tag{3}$$
$$\alpha_1 \stackrel{?}{=} \alpha_2 \wedge \alpha_1 = \bar{\alpha}_1 = \hat{\tau}_1 \wedge \alpha_2 = \bar{\alpha}_2 = \hat{\tau}_2 \quad \to \quad \alpha_1 = \bar{\alpha}_1 = \alpha_2 = \bar{\alpha}_2 = \hat{\tau}_1 \wedge \hat{\tau}_1 \stackrel{?}{=} \hat{\tau}_2 \tag{4}$$

La règle (1) indique que si les variables α_1 et α_2 sont déjà connues pour être égales, comme l'indique le fait qu'elles apparaissent dans une même multi-équation, alors l'équation $\alpha_1 \stackrel{?}{=} \alpha_2$ peut être supprimée. Les règles (2) et (3), qui sont symétriques, s'appliquent lorsque α_1 et α_2 ne sont pas encore connues pour être égales, c'est-à-dire lorsqu'elles apparaissent dans deux multi-équations distinctes. On réalise alors la *fusion* de ces deux multi-équations ; l'invariant de l'algorithme est maintenu du fait que l'on suppose l'une des deux multi-équations initiales composées de variables uniquement. La règle (4) s'applique dans le cas où chacune des deux multi-équations initiales contient un terme construit (ici $\hat{\tau}_1$ et $\hat{\tau}_2$). Dans ce cas, on effectue également une fusion, mais on ne peut inclure ces deux termes dans la nouvelle multi-équation sans briser l'invariant ; on en mémorise donc un seul ($\hat{\tau}_1$ est choisi arbitrairement), et on émet un nouveau sous-problème à résoudre, à savoir $\tau_1 \stackrel{?}{=} \tau_2$.

Les règles suivantes, caractéristiques de l'unification du premier ordre dans une algèbre libre, sont les règles de décomposition et d'échec. Leur nombre dépend de l'algèbre de types que l'on utilise, mais leur structure est régulière. Nous donnons ci-dessous les règles correspondant à l'algèbre engendrée par les constructeurs $\rightarrow$ et $\times$ binaires :

$$\begin{array}{rcl}
\alpha_1 \rightarrow \alpha_2 \stackrel{?}{=} \alpha'_1 \rightarrow \alpha'_2 & \rightarrow & \alpha_1 \stackrel{?}{=} \alpha'_1 \wedge \alpha_2 \stackrel{?}{=} \alpha'_2 \\
\alpha_1 \times \alpha_2 \stackrel{?}{=} \alpha'_1 \times \alpha'_2 & \rightarrow & \alpha_1 \stackrel{?}{=} \alpha'_1 \wedge \alpha_2 \stackrel{?}{=} \alpha'_2 \\
\alpha_1 \times \alpha_2 \stackrel{?}{=} \alpha'_1 \rightarrow \alpha'_2 & \rightarrow & \mathbf{false} \\
\alpha_1 \rightarrow \alpha_2 \stackrel{?}{=} \alpha'_1 \times \alpha'_2 & \rightarrow & \mathbf{false}
\end{array}$$

La restriction concernant la forme des types $\hat{\tau}$ – à savoir que leurs sous-termes sont des variables – est utilisée ici ; elle garantit que les nouvelles équations à résoudre sont des équations entre variables. Ainsi, nous n'avons jamais à traiter d'équations entre une variable et un terme non variable, cas qui pourrait poser un problème de terminaison. Supposons pour l'instant que nous travaillions dans l'algèbre engendrée par $\rightarrow$ et $\times$ – en particulier, en l'absence de rangées. On peut démontrer, à l'aide de l'invariant, que l'algorithme termine, c'est-à-dire qu'il n'existe aucune séquence de réécriture infinie, et qu'un problème en forme normale ne contient aucune équation à résoudre. Un problème en forme normale est donc en forme *résolue*, au sens où il ne contient plus que des multi-équations, dont la forme est sévèrement régie par l'invariant. Il ne reste plus qu'à démontrer que tout problème en forme résolue admet une solution.

Si l'on raisonne dans un univers de types finis *ou réguliers* (récursifs), alors on peut démontrer que c'est bien le cas. En d'autres termes, bien que ces multi-équations soient potentiellement récursives (par exemple $\alpha = \beta \rightarrow \gamma \wedge \beta = \alpha \rightarrow \gamma$), elles admettent toujours une solution.

Si en revanche l'on se situe dans un univers de types finis, ce qui est le cas du langage ML habituel, alors une condition supplémentaire est nécessaire pour garantir que toute forme normale est soluble. Disons que α_1 *domine* α_2 s'il existe une multi-équation $\alpha_1 = e = \hat{\tau}$ et α_2 apparaît libre dans $\hat{\tau}$. Disons qu'une conjonction de multi-équations est *cyclique* si la relation de domination qu'elles induisent est cyclique. On peut alors exprimer la condition souhaitée sous forme d'une règle supplémentaire, la règle *d'occur-check* :

$$C \rightarrow \mathbf{false} \qquad \text{si } C \text{ est cyclique}$$

Il est intéressant de noter qu'on peut choisir de n'appliquer cette règle qu'à la fin du processus ; il n'est donc pas nécessaire que l'algorithme échoue dès que le problème devient cyclique. La détection des cycles étant coûteuse, cela affecte (en théorie du moins) la complexité de l'algorithme.

Exercice 6.6 *(*)/(***) Démontrer, dans le cas où les types sont finis, que toute contrainte en forme normale admet une solution. Dans les cas des types infinis, définir l'univers des types réguliers ainsi que la notion de solution, puis montrer que toute contrainte en forme normale (pour le système privé de l'occur-check) admet une solution.*

Lorsque l'on ajoute les rangées, les règles ci-dessus restent inchangées, mais on doit ajouter de nouvelles règles de décomposition, comme nous l'avons fait pour `mgu` dans la section 6.5.1.

$$
\begin{array}{rcl}
\partial\mathtt{Abs} \stackrel{?}{=} \partial\mathtt{Abs} & \rightarrow & \mathbf{true} \\
(\ell : \alpha;\ \alpha') \stackrel{?}{=} \partial\mathtt{Abs} & \rightarrow & \alpha \stackrel{?}{=} \mathtt{Abs} \wedge \alpha' \stackrel{?}{=} \partial\mathtt{Abs} \\
\partial\mathtt{Abs} \stackrel{?}{=} (\ell : \alpha;\ \alpha') & \rightarrow & \alpha \stackrel{?}{=} \mathtt{Abs} \wedge \alpha' \stackrel{?}{=} \partial\mathtt{Abs} \\
(\ell : \alpha_1;\ \alpha'_1) \stackrel{?}{=} (\ell : \alpha_2;\ \alpha'_2) & \rightarrow & \alpha_1 \stackrel{?}{=} \alpha_2 \wedge \alpha'_1 \stackrel{?}{=} \alpha'_2 \\
(\ell : \alpha_1;\ \alpha'_1) \stackrel{?}{=} (\ell' : \alpha_2;\ \alpha'_2) & \rightarrow & \exists\alpha.(\alpha'_1 \stackrel{?}{=} (\ell' : \alpha_2;\ \alpha) \wedge \alpha'_2 \stackrel{?}{=} (\ell : \alpha_1;\ \alpha))
\end{array}
$$

Ces règles sont fort similaires à celle de la section 6.5.1, à la différence près que, en ce qui concerne la dernière règle, au lieu d'allouer une variable "fraîche" à l'aide de V, nous utilisons la quantification existentielle. Par convention, α doit être choisie hors des variables libres du membre gauche.

Exercice 6.7 *(***) Formuler et établir un énoncé d'équivalence entre l'algorithme* `mgu` *de la section 6.5.1 et l'algorithme ci-dessus. On pourra commencer par raisonner en l'absence de rangées.*

Comment implanter cet algorithme ? Les équations à résoudre et les multi-équations seront représentées en mémoire de façon très différente. Les multi-équations formeront une structure de données (allouée dans le tas) qui sera modifiée *en place* par les règles de réécriture. Les opérations dont nous avons besoin sont : (i) à partir d'une variable, trouver la multi-équation associée ; (ii) à partir d'une multi-équation, déterminer si elle contient un terme construit $\hat{\tau}$, et si oui l'exhiber ; (iii) fusionner deux multi-équations. En machine, ces opérations sont rendues possibles de façon extrêmement efficace par l'algorithme *d'union-find* de Tarjan. L'idée consiste à représenter chaque variable comme un nœud dans un graphe, et à distinguer, dans chaque multi-équation, une variable arbitraire. Chaque variable porte un pointeur ; si la variable est distinguée, elle porte un pointeur vers le terme $\hat{\tau}$ de la multi-équation, s'il en existe un ; si elle n'est pas distinguée, elle porte un pointeur vers une autre variable de la même multi-équation. Ces pointeurs ne devront pas former de cycle ; c'est un invariant que l'on maintient facilement. Ainsi, toutes les variables d'une même multi-équation forment un arbre dont la racine est la variable distinguée. À partir d'une variable arbitraire, il suffit de suivre les pointeurs pour parvenir à la racine de l'arbre, donc à la variable distinguée, et de là au terme $\hat{\tau}$ éventuellement associé à la multi-équation. De plus, la fusion de deux multi-équations s'effectue aisément en ajoutant un pointeur de la racine de l'une vers la racine de l'autre. On greffe ainsi l'un des arbres sur l'autre, pour obtenir un seul arbre. On a un choix quant à la direction de cette greffe ; on peut effectuer ce choix de sorte que tout arbre soit équilibré, ce qui du coup garantit un temps d'accès à la racine logarithmique vis-à-vis de la taille de l'arbre.

Exercice 6.8 *(***) Définir la représentation en machine de cette structure de données. Écrire une fonction* `unify` *qui, étant donnée une équation à résoudre (c'est-à-dire une paire de termes, dans lesquels les variables sont des nœuds dans cette structure), modifie la structure en conséquence et produit éventuellement de nouvelles équations à résoudre. Le code de cette fonction sera constitué d'une série de cas reflétant les règles de réécriture ci-dessus.*

Quant aux équations à résoudre, on peut les représenter de deux façons. Voici la première. On peut les représenter explicitement par des paires de termes et les placer dans une liste d'attente (c'est-à-dire une queue). On répète alors le processus suivant : tant que la queue n'est pas vide, on y prend une équation à résoudre, on lui applique `unify`, et on place dans la queue les équations produites par `unify`. La seconde approche représente la liste d'attente implicitement à l'aide de la pile du langage. Elle consiste simplement à rendre la fonction `unify` récursive, de sorte qu'au lieu de renvoyer de nouveaux buts en résultat, elle les résoud immédiatement à l'aide d'un appel récursif.

Exercice 6.9 *(*) Implémenter les deux approches ci-dessus.*

6.6 Les sommes ouvertes

De même que les enregistrements polymorphes et extensibles généralisent les enregistrements déclarés de la section 4.4.2, on peut généraliser les types concrets de la section 4.4.1 de façon à ne plus nécessiter la déclaration préalable du type concret et à pouvoir écrire des fonctions qui s'appliquent à n'importe quel type somme contenant au plus certains constructeurs avec certains types. On parle alors de *sommes ouvertes*, ou de *variantes polymorphes*, etc. (La terminologie est assez floue.) Ceci fait l'objet de l'exercice suivant.

Exercice 6.10 *(***) On se donne une famille de constructeurs $C, C', C_1, C_2, \ldots$ ainsi que pour chaque constructeur C un opérateur de projection P_C et un opérateur de filtrage ouvert F_C. Les règles de réduction pour P_C et F_C sont :*

$$\begin{array}{rcl} P_C(C(v)) & \stackrel{\varepsilon}{\to} & v \\ F_C(C(v), v_1, v_2) & \stackrel{\varepsilon}{\to} & v_1\ v \\ F_C(C'(v), v_1, v_2) & \stackrel{\varepsilon}{\to} & v_2\ (C'(v))\ \textit{si}\ C' \neq C \end{array}$$

Remarquez que P_C ne se réduit pas s'il est appliqué à une valeur $C'(v)$ avec $C' \neq C$. Autrement dit, le typage de P_C doit garantir que son argument porte le constructeur C et aucun autre. En revanche, F_C est un véritable filtrage en ce sens qu'il teste le constructeur de son premier argument, et appelle la fonction donnée en second argument ou bien celle donnée en troisième argument suivant que le constructeur de son premier argument est C ou pas.

Proposer un système de typage aussi flexible que possible pour cette présentation des types concrets. (Indication : on donnera aux valeurs de types concrets des types de la forme $[\tau]$, où τ est une rangée construite avec $C : \_ ; \_$ et ∂`Abs` *qui décrit tous les constructeurs pouvant apparaître dans cette valeur, avec les types de leur argument.) Donner les types des opérateurs C, P_C et F_C dans votre système. Quels types votre système donne-t'il aux expressions suivantes ?*

```
C(1)
if cond then C(1) else D(true)
fun x → F_C(x, fun y → y, fun z → 0)
fun x → F_C(x, fun y → y, P_D)
```

Chapitre 7

Programmation par objets et classes

La conception d'un système de typage statique sûr pour la programmation par objets est un problème délicat. Outre les garanties habituelles de sûreté, on se fixe comme objectif de garantir statiquement l'absence d'erreurs “message not understood” correspondant à évaluer $obj\#m$ où obj est un objet ne possédant pas la méthode m. La difficulté provient des nombreux traits originaux des objets et des classes, dont il faut rendre compte :

- l'emploi de “self” et la liaison tardive ;
- l'encapsulation des variables d'instance, qui sont accessibles seulement aux méthodes de l'objet ;
- le sous-typage entre types d'objets et ses interactions avec l'inférence de types ;
- l'héritage entre classes et ses liens avec le sous-typage.

En particulier, l'inférence de types en présence de sous-typage implicite pose des problèmes délicats. Le *sous-typage* est la possibilité de remplacer à tout instant le type d'un objet par un type moins précis ; cela permet, par exemple, d'oublier qu'un objet possède une certaine méthode, en retirant celle-ci de la liste des méthodes qui figurent dans le type de l'objet. Le sous-typage est dit *implicite* si ce changement de type n'a pas à être signalé dans le texte du programme, et *explicite* dans le cas contraire. Sans prétention à être exhaustif, on peut distinguer plusieurs approches de la question :

1. Sous-typage implicite, absence d'inférence de types. On déclare alors les types des paramètres des fonctions et des variables locales. C'est l'approche suivie par la plupart des langages orientés-objets classiques : Java, Eiffel, Modula-3, C++.

2. Sous-typage explicite et inférence de types. C'est l'approche suivie en Objective Caml. Le sous-typage y est explicite : pour changer le type d'un objet, on doit utiliser une *coercion* de la forme $(a : \tau :> \tau')$. L'avantage de cette approche est que la présence de sous-typage n'a alors aucun impact sur l'algorithme d'inférence, qui reste celui de ML.

3. Sous-typage implicite et inférence de types. On doit alors généraliser l'inférence de types et adopter un algorithme d'inférence à base de contraintes. Cette approche, étudiée pendant les années 1990, est bien comprise d'un point de vue théorique, mais pose quelques problèmes pratiques : elles est en général moins efficace que l'inférence de types de ML, et produit des types moins lisibles. Elle est néanmoins intéressante de par sa généralité.

Dans ce chapitre, nous étudions l'approche (2), car c'est une application directe des rangées de types que nous avons déjà utilisées au chapitre 6. L'approche (3) sera étudiée au chapitre 8.

7.1 Un calcul d'objets sans classes

L'approche d'Objective Caml est de traiter les objets comme des enregistrements polymorphes dont chaque champ correspond à une méthode de l'objet. Les variables d'instance et le paramètre `self` sont traités par une sémantique d'auto-application (*self-application semantics*).

7.1.1 Syntaxe

On commence par étendre mini-ML avec des objets mais pas de classes. Nous omettons également, pour le moment, la notion de variable d'instance. Un objet est donc constitué uniquement de méthodes. La construction d'un objet se fait en listant ses méthodes. Nos objets sont donc fort similaires à des enregistrements, à une différence près : nos méthodes ont un paramètre, lequel représente l'objet lui-même. On l'appelle typiquement "self". Il est représenté dans la grammaire ci-dessous par la variable x, laquelle est liée dans le corps de chaque méthode a_m. Ce paramètre permet à une méthode d'appeler d'autres méthodes du même objet, en envoyant un message à "self", c'est-à-dire à x. Les noms de méthodes m sont choisis dans un ensemble dénombrable fixé.

Expressions :	$a ::= \ldots$	
	$\mid \texttt{obj}(x)\langle (m = a_m)_{m \in M} \rangle$	construction d'un objet
Opérateurs :	$op ::= \ldots \mid \#m$	sélection de la méthode m

On note $a\#m$ pour l'application d'opérateur $\#m\ a$. Ceci correspond à l'appel de la méthode m de l'objet a.

7.1.2 Règles de réduction

La règle de réduction pour $\#m$ est la suivante :

$$v\#m \xrightarrow{\varepsilon} a_m[x \leftarrow v] \quad \text{si } v = \texttt{obj}(x)\langle (m = a_m)_{m \in M} \rangle$$

L'auto-application est visible ici par le remplacement de l'identificateur x par l'objet v lui-même dans le corps a_m de la méthode m. De la sorte, les appels $x\#m'$ présents dans a_m feront bien référence à l'objet v lui-même. Remarquons que l'auto-application est une forme de récursion. Par exemple, il est facile de définir des fonctions récursives ou mutuellement récursives à l'aide d'un objet :

```
let o = obj(s) < method fact = fun n →
                                 if n = 0 then 1 else n * s#fact (n-1) >
in o#fact 10
```

Les valeurs sont étendues de la façon suivante :

Valeurs :

$$v ::= \ldots \mid \texttt{obj}(x)\langle (m = a_m)_{m \in M} \rangle$$

En d'autres termes, tout objet est une valeur, puisque le corps des méthodes n'a pas être évalué. Les méthodes sont en cela semblables à des fonctions dont le paramètre est x ("self"). Les contextes d'évaluation sont inchangés, puisqu'on n'évalue jamais à l'intérieur d'un objet.

7.1.3 L'algèbre de types

Le type d'un objet est identique à celui d'un enregistrement : il liste les noms des méthodes de l'objet, avec pour chaque méthode le type de son résultat. (Notons que le type du paramètre "self" n'apparaît pas, puisque c'est en fait le type de l'objet lui-même.) Nous conservons donc exactement la même grammaire de types que dans le cas des types enregistrements, à la différence près que le constructeur des types objets est noté $\langle\cdot\rangle$ au lieu de $\{\cdot\}$.

Types :	$\tau \ ::= \alpha \mid T \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$	comme précédemment
	$\mid \langle\tau\rangle$	type d'objet
	$\mid \partial\mathtt{Abs}$	la rangée vide
	$\mid m : \tau_1;\ \tau_2$	la rangée contenant $m : \tau_1$ plus ce que contient la rangée τ_2
	$\mid \mathtt{Abs}$	le champ est absent (indéfini)
	$\mid \mathtt{Pre}\ \tau$	le champ est présent (défini) avec le type τ

La théorie équationnelle sur les types est la même que dans le cas des enregistrements, ainsi que le système de sortes.

Exemples : le type $\langle m : \mathtt{Pre\ int};\ n : \mathtt{Pre\ string};\ \partial\mathtt{Abs}\rangle$ est le type des objets possédant une méthode m renvoyant un entier, une méthode n renvoyant une chaîne, et aucune autre méthode. De tels types "fermés" apparaissent lorsque l'on type la création d'un objet $\mathtt{obj}(s)\langle\ldots\rangle$.

Le type $\langle m : \mathtt{Pre\ int};\ \alpha\rangle$ est le type des objets possédant une méthode m renvoyant un entier, et éventuellement d'autres méthodes, selon la valeur que prendra α. De tels types "ouverts" apparaissent naturellement pour les paramètres de fonctions ; ainsi,

```
fun obj → 1 + obj#m
```

admet le schéma de types $\forall\alpha.\langle m : \mathtt{Pre\ int};\ \alpha\rangle \to \mathtt{int}$.

7.1.4 Règles de typage

Pour étendre le système de types, il faut donner la règle de typage associée à la construction `obj` :

$$\frac{\tau = \langle (m : \mathtt{Pre}\ \tau_m)_{m\in M};\ \partial\mathtt{Abs}\rangle \qquad \forall m \in M \quad \Gamma; x : \tau \vdash a_m : \tau_m}{\Gamma \vdash \mathtt{obj}(x)\langle (m = a_m)_{m\in M}\rangle : \tau}$$

On notera que le type attribué à x lorsqu'on type le corps de chaque méthode doit coïncider avec le type de l'objet lui-même. Une telle structure, où le type τ apparaît à la fois en hypothèse de la prémisse et en conclusion de la règle, est typique des constructions permettant les définitions récursives. On comparera par exemple avec la règle de typage de la construction `let rec`.

Enfin, le schéma de types pour l'opérateur $\#m$ est exactement le même que celui de l'opération d'accès à un enregistrement :

$$\#m \quad : \quad \forall\alpha,\beta.\ \langle m : \mathtt{Pre}\ \alpha;\ \beta\rangle \to \alpha$$

7.1.5 Sûreté du typage

La sûreté du typage ci-dessus se montre comme au chapitre 2. En particulier, il faut montrer les hypothèses (H1) et (H2) pour l'opérateur $\#m$. Pour (H1), supposons $v\#m \xrightarrow{\varepsilon} a_m[x \leftarrow v]$, où

$v = \mathtt{obj}(x)\langle (m = a_m)_{m \in M}\rangle$, et $\Gamma \vdash v\#m : \tau$. La dérivation de typage doit être de la forme :

$$\dfrac{\#m : \tau_v \to \tau \qquad \dfrac{\begin{array}{c}\tau_v = \langle (m : \mathtt{Pre}\ \tau_m)_{m\in M};\ \partial\mathtt{Abs}\rangle \\ \forall m \in M \quad \Gamma; x : \tau_v \vdash a_m : \tau_m\end{array}}{\Gamma \vdash v : \tau_v}}{\Gamma \vdash v\#m : \tau}$$

De ceci, on déduit que $\tau_v \to \tau$ est instance du schéma de types associé à $\#m$, d'où il découle que τ n'est autre que τ_m. Or, en appliquant le lemme de substitution (lemme 2.2) à $\Gamma; x : \tau_v \vdash a_m : \tau_m$ et à $\Gamma \vdash v : \tau_v$, on obtient $\Gamma \vdash a_m[x \leftarrow v] : \tau_m$, c'est-à-dire $\Gamma \vdash a_m[x \leftarrow v] : \tau$.

La preuve de (H2) est du même style ; nous l'omettons.

7.1.6 Inférence de types

L'algorithme W s'étend aisément à cette nouvelle construction :

$$\begin{array}{rcl} W(\Gamma \vdash \mathtt{obj}(x)\langle (m = a_m)_{m\in M}\rangle) & = & \mathtt{do}\,\alpha = \mathtt{fresh} \\ & & \mathtt{do}\,\forall m \in M \quad \tau_m = W(\Gamma; x : \alpha \vdash a_m) \\ & & \mathtt{do}\,\varphi \leftarrow \mathtt{mgu}(\varphi(\alpha) \stackrel{?}{=} \langle (m : \mathtt{Pre}\ \varphi(\tau_m))_{m\in M};\ \partial\mathtt{Abs}\rangle) \circ \varphi \\ & & \mathtt{return}\,\alpha \end{array}$$

Notons que l'ordre dans lequel les différentes méthodes sont analysées n'a pas d'importance. L'ordre dans lequel elles apparaissent dans le type $\langle (m : \mathtt{Pre}\ \varphi(\tau_m))_{m\in M};\ \partial\mathtt{Abs}\rangle$ est également sans importance.

7.2 Sous-typage explicite

Le calcul d'objets ci-dessus n'a aucun mécanisme de sous-typage. En particulier, il n'est pas possible d'oublier des méthodes dans un objet : si $a : \langle \mathtt{m} : \mathtt{Pre\ int};\ \partial\mathtt{Abs}\rangle$ et $b : \langle \mathtt{m} : \mathtt{Pre\ int};\ \mathtt{n} : \mathtt{Pre\ string};\ \partial\mathtt{Abs}\rangle$, l'expression `if` *cond* `then` a `else` b n'est pas typable, et en particulier n'a pas le type "naturel" $\langle \mathtt{m} : \mathtt{Pre\ int};\ \partial\mathtt{Abs}\rangle$. Ceci peut être considéré comme une faiblesse ; les langages orientés objets courants, comme Java, utilisent couramment le sous-typage pour former des listes d'objets "hétérogènes", en ne retenant que le dénominateur commun de leurs méthodes. (En Java, le type d'un objet est le nom d'une classe ou d'une interface, et non la liste de ses méthodes ; retenir le dénominateur commun consiste alors à remplacer le type des objets de la liste par une super-classe ou super-interface commune aussi précise que possible.)

Pour pallier cette faiblesse, on ajoute une famille d'opérateurs de coercion $\mathtt{coerce}_{\tau,\tau'}$ qui permettent de transformer explicitement un objet de type τ en un objet du super-type τ'.

Opérateurs : *op* $::= \mathtt{coerce}_{\tau,\tau'}$ pour tous τ, τ' tels que $\tau <: \tau'$

La construction $(a : \tau :> \tau')$ d'Objective Caml est bien sûr du sucre syntaxique pour $\mathtt{coerce}_{\tau,\tau'}(a)$. Le type des opérateurs de coercion et leur règle de réduction sont :

$$\begin{array}{rcl} \mathtt{coerce}_{\tau,\tau'} & : & \forall \vec{\alpha}.\ \tau \to \tau' \quad \text{si } \tau <: \tau' \text{ et } \vec{\alpha} = \mathcal{L}(\tau) \cup \mathcal{L}(\tau') \\ \mathtt{coerce}_{\tau,\tau'}(v) & \xrightarrow{\varepsilon} & v \end{array}$$

Le fait que la règle de réduction soit celle de l'identité indique que ces coercions n'ont aucun effet sur la sémantique du programme. Elles constituent simplement une indication à l'intention du typeur.

Relation de sous-typage : La relation de sous-typage $<:$ qui définit l'ensemble des coercions légales est définie par les règles d'inférence suivantes :

$$T <: T \qquad \alpha <: \alpha \qquad \frac{\tau' <: \tau \quad \varphi <: \varphi'}{(\tau \to \varphi) <: (\tau' \to \varphi')} \qquad \frac{\tau <: \tau' \quad \varphi <: \varphi'}{(\tau \times \varphi) <: (\tau' \times \varphi')}$$

$$\frac{\tau <: \tau'}{\langle \tau \rangle <: \langle \tau' \rangle} \qquad \tau <: \partial\texttt{Abs} \qquad \frac{\tau <: \tau' \quad \varphi <: \varphi'}{(m : \tau;\ \varphi) <: (m : \tau';\ \varphi')}$$

$$\texttt{Abs} <: \texttt{Abs} \qquad \frac{\tau <: \tau'}{\texttt{Pre}\ \tau <: \texttt{Pre}\ \tau'}$$

Pour commencer, notons que l'axiome le plus intéressant est certainement $\tau <: \partial\texttt{Abs}$, puisqu'en son absence, la relation $<:$ ne serait autre que l'égalité. Cet axiome autorise une rangée quelconque τ à être remplacée par la rangée uniformément absente $\partial\texttt{Abs}$, ce qui correspond à oublier la présence de certaines méthodes.

Les autres règles se contentent de propager le sous-typage à travers la structure des types, de façon à ce que l'axiome ci-dessus soit applicable non seulement à la racine d'un type, mais également à l'intérieur de celui-ci. La seule subtilité provient de ce que dans certaines prémisses, le sous-typage est propagé de façon *covariante*, tandis que dans d'autres, il est propagé de façon *contravariante*. Considérons par exemple la règle de sous-typage entre types flèches. Celle-ci est covariante vis-à-vis du codomaine et contravariante vis-à-vis du domaine. En d'autres termes, la fonction $\cdot \to \tau$ est décroissante vis-à-vis de l'ordre $<:$, tandis que la fonction $\tau \to \cdot$ est croissante. Ceci peut se comprendre en imaginant les types comme des ensembles de valeurs, et la relation de sous-typage comme l'inclusion entre ensembles de valeurs. En théorie des ensembles, $A \to B$ (l'ensemble des fonctions de A dans B) est inclus dans $A' \to B'$ (l'ensemble des fonctions de A' dans B') si et seulement si $A' \subseteq A$ et $B \subseteq B'$.

Exemples : les jugements suivants sont corrects :

$$\begin{array}{rcl}
\langle m : \texttt{Pre int};\ \partial\texttt{Abs}\rangle & <: & \langle \partial\texttt{Abs}\rangle \\
\langle o : \langle m : \texttt{Pre int};\ \partial\texttt{Abs}\rangle\rangle & <: & \langle o : \langle \partial\texttt{Abs}\rangle\rangle \\
\langle m : \texttt{Pre int};\ \alpha\rangle & \not<: & \langle \alpha \rangle \\
\texttt{int} \to \langle m : \texttt{Pre int};\ \partial\texttt{Abs}\rangle & <: & \texttt{int} \to \langle \partial\texttt{Abs}\rangle \\
\langle m : \texttt{Pre int};\ \partial\texttt{Abs}\rangle \to \texttt{int} & \not<: & \langle \partial\texttt{Abs}\rangle \to \texttt{int} \\
\langle \partial\texttt{Abs}\rangle \to \texttt{int} & <: & \langle m : \texttt{Pre int};\ \partial\texttt{Abs}\rangle \to \texttt{int}
\end{array}$$

Le troisième jugement est mal sorté. Le cinquième est faux du fait que le constructeur flèche est contravariant vis-à-vis de son domaine, donc le sous-typage permet *d'ajouter* des méthodes à l'argument d'une fonction, non d'en *oublier*.

Sous-typage et sûreté du typage : la règle de réduction pour $\texttt{coerce}_{\tau,\tau'}$ compromet la préservation du typage pendant la réduction : en effet, si v a le type τ, l'expression $\texttt{coerce}_{\tau,\tau'}(v)$ a le type τ', mais se réduit en v qui a le type $\tau \neq \tau'$.

Une manière de retrouver la préservation du typage, donc de montrer la sûreté du typage, est d'ajouter – uniquement pour les besoins de la preuve – une règle de sous-typage implicite :

$$\frac{\Gamma \vdash a : \tau \qquad \tau <: \tau'}{\Gamma \vdash a : \tau'} \text{ (sub)}$$

Avec cette règle, on peut démontrer que la réduction $\texttt{coerce}_{\tau,\tau'}(v) \xrightarrow{\varepsilon} v$ préserve le typage, puisque v, qui a le type τ par hypothèse, a alors aussi le type τ', par application de la règle (sub).

Bien sûr, cette règle (sub) rend l'inférence de types problématique – c'est justement pour éviter cela que nous avons choisi d'employer des coercions explicites ! Il faut donc typer les programmes source sans la règle (sub), ce qui permet de faire de l'inférence de types, et n'ajouter la règle (sub) que pour raisonner sur la préservation du typage pendant la réduction.

7.3 Classes

Nous ajoutons maintenant au calcul d'objets une notion de classes avec un mécanisme d'héritage. Les classes sont des collections de définitions de méthodes. Une construction `new` explicite permet de prendre une instance d'une classe, c'est-à-dire de construire un objet ayant les méthodes indiquées dans la classe.

Quelle est alors la différence entre classes et objets ? (La question est d'autant plus valide que cette distinction n'existe pas dans certains langages.) Ici, la réponse est qu'on peut *hériter* d'une classe, c'est-à-dire définir une nouvelle classe en apportant une modification incrémentale à une classe existante ; cette opération n'est pas permise sur les objets. Cette distinction sera reflétée au niveau des types : les types de classes devront comporter plus d'information que les types d'objets.

Pour simplifier, nous présentons les classes comme faisant partie des expressions ; dans un langage réaliste, on a tendance à distinguer un sous-langage de classes distinct du langage des expressions (dans un langage comme Objective Caml, cela simplifie l'inférence de types).

Toujours pour simplifier, nous ne traitons pas l'héritage multiple. Nous nous limitons à un héritage simple permettant la redéfinition d'exactement une méthode (au lieu de zéro, une ou plusieurs). On se référera à l'article de Rémy et Vouillon cité à la fin de ce chapitre pour un traitement plus complet.

Expressions :	$a ::= \ldots$	
	$\mid \texttt{new}$	création d'un objet à partir d'une classe
	$\mid \texttt{class}(x)\langle (m = a_m)_{m \in M} \rangle$	classe
	$\mid \texttt{class}(x)\langle \texttt{inherit}\ a;\ m = a \rangle$	héritage et ajout ou redéfinition d'une méthode

On précise que, dans une définition de classe, le paramètre x ("self") est lié dans le corps des méthodes, mais pas dans la clause `inherit`.

7.3.1 Évaluation des classes

Nous considérerons qu'une classe est entièrement évaluée si elle ne contient aucune clause `inherit` :

Valeurs : v ::= … | $\texttt{class}(x)\langle(m = a_m)_{m\in M}\rangle$

L'évaluation de `new` transforme une classe entièrement évaluée en un objet portant les mêmes méthodes. L'évaluation d'une classe consiste à résoudre les clauses `inherit` qu'elle contient, par un mécanisme qui ressemble à celui de l'extension des enregistrements.

$$\begin{array}{rcl} \texttt{new class}(x)\langle(m = a_m)_{m\in M}\rangle & \xrightarrow{\varepsilon} & \texttt{obj}(x)\langle(m = a_m)_{m\in M}\rangle \\ \texttt{class}(x)\langle\texttt{inherit class}(x)\langle(m = a_m)_{m\in M}\rangle;\ n = b\rangle & \xrightarrow{\varepsilon} & \texttt{class}(x)\langle(m = a'_m)_{m\in M\cup\{m'\}}\rangle \\ & & \text{où } a' = a[n \mapsto b] \end{array}$$

Notons que, dans la seconde règle de réduction ci-dessus, on suppose que le nom de "self" est le même dans la classe courante et dans la super-classe (à savoir x). Cela peut toujours être obtenu par α-conversion, donc cette condition n'est pas restrictive. Les contextes d'évaluation sont les suivants :

Contextes d'évaluation : E ::= … | $\texttt{new}\ E$ | $\texttt{class}(x)\langle\texttt{inherit}\ E;\ m = a\rangle$

En réalité, en Objective Caml, le corps d'une classe n'est évalué qu'au moment où on lui applique l'opérateur `new`. Si on désire obtenir ce comportement lorsqu'on programme dans notre langage noyau, il suffit d'abstraire chaque classe vis-à-vis de `()`, c'est-à-dire écrire $\texttt{fun}\ () \to \texttt{class}(x)\langle\ldots\rangle$ au lieu de $\texttt{class}(x)\langle\ldots\rangle$ et écrire $\texttt{new}\ (c())$ au lieu de $\texttt{new}\ c$.

7.3.2 Typage des classes

Types : τ ::= … | $\texttt{class}(\tau_1)\ \tau_2$ type de classe

Le type d'une classe est $\texttt{class}(\tau_1)\ \tau_2$, où τ_1 est le type du paramètre "self" des méthodes, et τ_2 est un type d'objet représentant les types des méthodes définies dans la classe. Si τ_1 et τ_2 sont égaux, alors la classe est "autosuffisante", c'est-à-dire que toutes les méthodes utilisées dans des appels à "self" sont effectivement définies. Il est légal d'appliquer la construction `new` à une telle classe pour obtenir un objet. Si, à l'inverse, τ_1 mentionne des méthodes qui n'apparaissent pas dans τ_2, alors la classe est "virtuelle" : certaines méthodes sont mentionnées dans des appels à "self" mais ne sont pas définies. Il est alors illégal d'appliquer `new` à cette classe ; il faut d'abord utiliser l'héritage pour ajouter les méthodes manquantes. Par exemple, la classe OCaml

```
class virtual c =
  object(self)
    method virtual m : int
    method n = 1 + self#m
  end
```

admet le type $\texttt{class}(\langle\texttt{m : Pre int; n : Pre int};\ \alpha\rangle)\ \langle\texttt{n : Pre int};\ \partial\texttt{Abs}\rangle$. Dans un type de classe comme celui-ci, le type de "self" (τ_1) est typiquement ouvert, c'est-à-dire terminé par une variable de rangée, et pourra donc être spécialisé (par instanciantion) dans une sous-classe, tandis que le type des méthodes fournies (τ_2) est toujours fermé, puisque les méthodes fournies sont en nombre fini et sont connues. Le compilateur Objective Caml adopte une syntaxe plus légère pour les types de classes, où τ_1 et τ_2 sont superposés, les méthodes présentes dans τ_1 et non dans τ_2 étant alors marquées comme "virtuelles" :

```
class virtual c : object method virtual m : int method n : int end
```

Ceci est rendu possible par le fait que si une méthode apparaît à la fois dans τ_1 et dans τ_2, alors c'est avec le même type ; cela est garanti par les règles ci-dessous. Les règles garantissent également que toute méthode présente dans τ_2 est présente dans τ_1.

Voici les règles de typage associées aux nouvelles constructions.

$$\frac{\Gamma \vdash a : \texttt{class}(\tau)\ \tau}{\Gamma \vdash \texttt{new}\ a : \tau}\ \text{(new)}$$

$$\frac{\tau = \langle (m : \texttt{Pre}\ \tau_m)_{m\in M};\ \tau' \rangle \qquad \forall m \in M \quad \Gamma; x : \tau \vdash a_m : \tau_m}{\Gamma \vdash \texttt{class}(x)\langle (m = a_m)_{m\in M} \rangle : \texttt{class}(\tau)\ \langle (m : \texttt{Pre}\ \tau_m)_{m\in M};\ \partial\texttt{Abs} \rangle}\ \text{(classe)}$$

$$\frac{\Gamma \vdash a : \texttt{class}(\tau)\ \langle m : \tau_m^0;\ \tau' \rangle \qquad \Gamma; x : \tau \vdash b : \tau_m \qquad \tau = \langle m : \texttt{Pre}\ \tau_m;\ \tau'' \rangle}{\Gamma \vdash \texttt{class}(x)\langle \texttt{inherit}\ a;\ m = b \rangle : \texttt{class}(\tau)\ \langle m : \texttt{Pre}\ \tau_m;\ \tau' \rangle}\ \text{(inherit)}$$

La première règle n'autorise l'application de `new` à une classe que si celle-ci n'est pas virtuelle, c'est-à-dire si les méthodes définies et les méthodes nécessaires coïncident. La seconde règle donne le type d'une classe définie *in extenso* ; on remarque que les types entrée et sortie de la classe coïncident en ce qui concerne les méthodes définies ; ils diffèrent en ce que le premier est ouvert (il se termine par un τ' arbitraire) tandis que le second est fermé (il se termine par $\partial\texttt{Abs}$). La troisième règle donne le type d'une classe définie par héritage. On remarque d'abord que le typage de la classe initiale a et le type de la classe obtenue ne diffèrent que par le fait que m est à présent définie, alors qu'elle pouvait être définie ou non dans a ; c'est ce qu'exprime le type τ_m^0, dont la sorte est `PRE`, et qui peut être choisi arbitrairement. Le type de "self" est le même dans les deux classes, ce qui assure la cohérence du typage. La troisième prémisse exige que la méthode m soit mentionnée dans le type de "self", exactement comme dans le cas des classes définies *in extenso*. On peut être surpris par l'exigence que le type de "self" soit le même dans les deux classes ; n'est-ce pas restrictif, dans le cas où m n'était pas initialement définie ? En fait non, car le type de "self" dans la classe initiale est en général ouvert, c'est-à-dire qu'il se termine par une variable, et si cette classe a été introduite par une construction `let`, alors cette variable a été généralisée. La classe initiale a donc typiquement un schéma de types de la forme

$$\forall\alpha.\ \texttt{class}(\langle (m : \tau_m)_{m\in M\cup M'};\ \alpha \rangle)\ \langle (m : \tau_m)_{m\in M};\ \partial\texttt{Abs} \rangle$$

En instanciant correctement la variable de rangée α, on peut satisfaire les contraintes imposées par la règle (inherit). Si la méthode m' qu'on souhaite ajouter est déjà définie ($m' \in M'$), alors on devra respecter son type ; sinon, on aura le choix de son type, puisqu'on pourra instancier α arbitrairement.

Pour simplifier les types affichés, Objective Caml cache toujours les variables de rangées, qui n'apparaissent donc jamais dans les types de classes et qui apparaissent sous forme anonyme uniquement ("..") dans les types d'objets. Si une classe admet un schéma de types polymorphe, Objective Caml demande que cela soit spécifié en paramétrant explicitement la déclaration de classe par des variables de types, et en instanciant explicitement ces paramètres lorsque la classe est utilisée. Cela est illustré par la classe `comparable` ci-dessous.

On remarquera qu'il n'y a, dans notre approche, aucun lien entre héritage et sous-typage. Le sous-typage n'intervient nullement dans les règles de typage liées aux classes. On pourra vérifier que si la classe B est définie par héritage à partir de la classe A, alors le type des objets de classe B est parfois, mais pas nécessairement, un sous-type du type des objets de classe A. Pour plus de détails sur la distinction entre héritage et sous-type, on pourra se référer à

> W. R. Cook, W. L. Hill, and P. S. Canning. *Inheritance is not subtyping.* ACM Press, Proceedings of POPL'90. `http://www.gamboas.org/ruben/classes/2002-fall/cosc5000/inherit-subclass.pdf`

Exemple : Voici un fragment d'Objective Caml qui illustre plusieurs points délicats : déclaration et emploi d'une classe `comparable` paramétrée par un type `'a` ; utilisation de ce type dans la spécification du type de la méthode virtuelle `get` ; méthode virtuelle `leq` acceptant un paramètre du même type que l'objet courant ; méthode `min` renvoyant un résultat du même type que l'objet courant ; définition, par héritage, d'une sous-classe `int_comparable` paramétrée par un entier `n`. Nous avons utilisé ici un minimum d'annotations de types, mais on pourrait annoter `self`, `leq` et `min` par leurs types.

```
class virtual ['a] comparable =
  object (self)
    method virtual get : 'a
    method virtual leq : _
    method min y = if self#leq y then self else y
  end
class int_comparable n =
  object(self)
    inherit [int] comparable
    method get = n
    method leq obj = self#get <= obj#get
  end
```

Une traduction dans notre petit langage de classes donne approximativement :

```
let comparable =
  class(self : < get : Pre 'a ; leq : Pre 'b ; 'c>)
  < min = fun y → if self#leq y then self else y > in
let int_comparable =
  fun n →
    class(self) < inherit comparable ;
                  get = n ;
                  leq = fun obj → self#get <= obj#get >
```

La contrainte sur le type de `self` dans la définition de `comparable` provient des deux déclarations de méthodes virtuelles. Cette contrainte va diminuer la généralité du schéma de types attribué à `comparable` : le type de `self` mentionnera la méthode `get`, tandis qu'elle n'aurait pas été mentionnée dans le type inféré en l'absence de contrainte, puisqu'elle n'est pas appelée par la méthode `min`. Voici le schémas de types inféré pour `comparable` :

$$\forall\alpha\beta\gamma[\gamma = \langle \texttt{get} : \texttt{Pre}\ \alpha;\ \texttt{leq} : \texttt{Pre}\ (\gamma \to \texttt{bool});\ \texttt{min} : \texttt{Pre}\ (\gamma \to \gamma);\ \beta\rangle].$$
$$\texttt{class}(\gamma)\ \langle \texttt{min} : \texttt{Pre}\ (\gamma \to \gamma);\ \partial\texttt{Abs}\rangle$$

Il est quantifié vis-à-vis de trois variables de types : α représente le type de `get` ; β est la variable qui termine le type de "self" ; γ est le type de "self". Nous avons écrit ce schéma sous la forme d'un schéma de types *contraint*, de la forme $\forall\alpha\beta\gamma[C].\tau$ où C est une *contrainte* : ici, une équation portant sur γ. Cette notion sera formellement définie au chapitre 8. La raison pour laquelle nous avons employé cette notation tient au fait que l'équation portant sur γ est *récursive*.

Ainsi, ce programme n'est typable qu'en présence de types récursifs ; en leur absence, cette équation n'aurait aucune solution, et le programme serait rejeté. En fait, dès que le type de "self" apparaît dans le type d'une méthode, l'emploi de types récursifs devient nécessaire.

Nous aurions pu résoudre cette équation en substituant les deux occurrences de droite de γ par l'unique solution de l'équation, à savoir $\mu\gamma.\langle \texttt{get} : \texttt{Pre}\ \alpha;\ \texttt{leq} : \texttt{Pre}\ (\gamma \to \texttt{bool});\ \texttt{min} : \texttt{Pre}\ (\gamma \to$

γ); $\beta\rangle$. Le résultat serait cependant assez lourd syntaxiquement ; c'est pourquoi nous avons préféré employer un schéma de types contraint.

Le schéma de types inféré pour `int_comparable` est le suivant :

$$\forall\beta\gamma[\gamma = \langle \texttt{get} : \texttt{Pre int};\ \texttt{leq} : \texttt{Pre}\ (\gamma \rightarrow \texttt{bool});\ \texttt{min} : \texttt{Pre}\ (\gamma \rightarrow \gamma);\ \beta\rangle].$$
$$\texttt{int} \rightarrow \texttt{class}(\gamma)\ \langle \texttt{get} : \texttt{Pre int};\ \texttt{leq} : \texttt{Pre}\ (\gamma \rightarrow \texttt{bool});\ \texttt{min} : \texttt{Pre}\ (\gamma \rightarrow \gamma);\ \partial\texttt{Abs}\rangle$$

En instanciant β par ∂`Abs`, on constate que l'application `int_comparable 1` admet le type $\texttt{class}(\gamma)\ \gamma$, où γ est sujet à l'équation $\gamma = \langle \texttt{get} : \texttt{Pre int};\ \texttt{leq} : \texttt{Pre}\ (\gamma \rightarrow \texttt{bool});\ \texttt{min} : \texttt{Pre}\ (\gamma \rightarrow \gamma);\ \beta\rangle$. Cette classe constitue donc un argument acceptable pour l'opérateur `new`. On peut écrire (en syntaxe Objective Caml) :

```
let o1 = new int_comparable 1 ; ;
let o2 = new int_comparable 2 ; ;
o1#leq o2 ; ;
```

On constatera que le type inféré pour `o1` par Objective Caml est `int_comparable` ; ce type est en réalité une abréviation pour le type récursif γ. Le système d'abréviations d'Objective Caml est trop complexe pour être abordé ici ; on voit toutefois que son importance est essentielle, car le type abrégé est beaucoup plus compréhensible que l'équation récursive portant sur γ que nous avons inférée.

7.3.3 Variables d'instance

En Objective Caml, comme dans la plupart des langages orientés objets, un objet contient non seulement des méthodes, mais également des champs, souvent appelés *variables d'instance*. En Objective Caml, ces champs sont accessibles à toutes les méthodes de l'objet, mais invisibles de l'extérieur. En termes techniques, lorsqu'on type une méthode, on fait figurer chaque champ dans l'environnement de typage, ce qui le rend accessible à la méthode ; mais le type de l'objet, lui, ne mentionne pas les champs, ce qui les rend inaccessibles de l'extérieur.

On peut tenter de simuler la notion de variable d'instance dans notre langage noyau. Considérons la classe Objective Caml suivante, qui implémente une cellule de référence, dont le contenu a le type `'a`, dotée de deux méthodes `get` et `set`.

```
class ['a] cellule (init : 'a) =
  object
    val mutable v = init
    method get = v
    method set x = v <- x
  end
```

Si v est une valeur de type τ, tout appel à `new cellule` v produit un nouvel objet, dont le champ `v` contient initialement v, et dont le type est $[\tau]$ `cellule`. On peut imaginer coder cet idiome dans notre langage noyau de la façon suivante :

```
let cellule init =
  let v = ref init in
  class(self)
    get = !v
    set x = v := x
  end
```

L'effet obtenu est sensiblement le même : une nouvelle cellule de référence contenant v est allouée à chaque appel de `new (cellule` v`)`. L'identificateur `v` est accessible depuis les méthodes `get` et `set`, mais n'apparaît bien sûr pas dans le type de l'objet, puisqu'il s'agit là d'une déclaration `let` tout-à-fait standard.

La limite de ce codage réside dans le fait que, en Objective Caml, les variables d'instance d'une classe sont également accessibles aux méthodes de ses sous-classes. Ainsi, on peut écrire :

```
class ['a] cellule' (init : 'a) =
  object(self)
    inherit ['a] cellule (init)
    method incrémente = v <- v + 1
  end
```

Pour permettre cela, il faut naturellement que les variables d'instance apparaissent dans les types de classes (ici, que `v` apparaisse dans le type de la classe `cellule`), ce que notre langage noyau ne permet pas. Dans le codage ci-dessus, l'identificateur `v` est lié par `let`. Il n'est donc pas accessible à l'extérieur de la définition de `cellule`, y compris dans les sous-classes de `cellule`.

Chapitre 8

Systèmes de types à base de contraintes

La section 3.2 nous a montré que le problème de l'inférence de types pour le λ-calcul simplement typé peut être vu comme un problème de résolution d'équations entre types. La raison pour laquelle ces équations apparaissent tient au fait que certaines règles imposent une forme spécifique aux types qui apparaissent dans leurs prémisses ou dans leur conclusion ; par exemple, la règle associée à l'application exige que le type du membre gauche soit (égal à) un type flèche, dont le domaine coïncide avec le (est égal au) type du membre droit.

Lorsque nous avons étudié l'inférence de types pour ML, section 3.3, nous avons proposé une formulation – l'algorithme W – dans laquelle la notion de système d'équations disparaît. En effet, au lieu de construire un système d'équations C, l'algorithme en calcule directement et incrémentalement une solution principale φ. La notion de système d'équations passe donc au second plan, mais on pourrait imaginer une formulation de l'algorithme dans laquelle ces équations resteraient explicites.

Dans la section 7.2, nous avons introduit l'idée d'une relation d'ordre entre types (appelée *sous-typage*). Celle-ci est rendue exploitable à l'aide d'une règle de typage supplémentaire, la règle (sub), qui permet à tout instant de remplacer le type associé à une expression par un super-type quelconque, c'est-à-dire un type moins général. Le sous-typage permet ainsi d'oublier une partie des opérations permises sur cette expression ; par exemple, dans le cas des objets, on peut oublier la présence de certaines méthodes dans un objet. Cela permet de construire des structures de données hétérogènes, par exemple une liste d'objets n'ayant que certaines méthodes en commun. La règle (sub) pouvant être utilisée n'importe où, un algorithme d'inférence de types devra en tenir compte lorsqu'il cherche à produire un type principal. Par exemple, lorsqu'il considérera une application, il ne pourra plus exiger que le type du membre gauche soit *égal à* un type flèche dont le domaine est le type du membre droit ; ce pourra être un *sous-type* arbitraire d'un tel type. L'algorithme sera donc naturellement modifié de façon à engendrer un système d'*inéquations* ou *contraintes de sous-typage*.

Un système d'équations entre types du premier ordre, s'il est soluble, admet un unificateur principal. Nous nous sommes appuyés sur cette propriété lorsque nous avons défini l'algorithme W, pour travailler directement en termes de solutions principales et non en termes de systèmes d'équation. L'existence de types principaux en ML est donc liée de façon essentielle à cette propriété. (Rappelons que l'algorithme W et ses propriétés constituent notre preuve de l'existence de types principaux en ML.) Or, il est clair qu'un système d'inéquations n'admet pas de solution principale. En effet, quelle serait la solution principale φ de l'inéquation $\alpha \leq \beta$? Les termes $\varphi(\alpha)$ et $\varphi(\beta)$ doivent être des variables, sinon φ ne peut être une solution principale. S'ils sont égaux, alors φ n'est pas principale, car $\alpha \leq \beta$ n'implique pas $\alpha = \beta$, à moins que la relation de sous-typage soit réduite à l'égalité. À l'inverse, si $\varphi(\alpha)$ et $\varphi(\beta)$ sont des variables distinctes, alors pour que φ soit solution de $\alpha \leq \beta$, il faudrait que la relation de sous-typage soit universellement vraie. Il découle de cette remarque que nous ne pourrons plus raisonner en termes de solutions principales, mais bien en termes de systèmes d'inéquations, c'est-à-dire de *contraintes*. En particulier, en présence de polymorphisme, l'existence de (schémas de) types principaux ne pourra être obtenue qu'en incorporant des contraintes au sein même des schémas de types. En d'autres termes, les contraintes devront être généralisées et instanciées, exactement comme les types précédemment.

Ceci va nous amener à formuler un système de types (et un algorithme d'inférence associé) *à base de contraintes*. Cette formulation est plus complexe que celle de ML. En contrepartie, elle est beaucoup plus générale, car la *syntaxe* et *l'interprétation logique* des contraintes sont flexibles. En d'autres termes : (i) on peut définir des formes de contraintes autres que les équations et les contraintes de sous-typage évoquées plus haut ; (ii) on peut interpréter ces contraintes dans divers modèles logiques, par exemple dans un univers de types finis, dans un univers de types récursifs, dans un univers de types avec rangées, etc. Toutes ces variations sont ainsi regroupées dans un même cadre théorique, et partagent une unique preuve de sûreté du typage.

Le paragraphe précédent insiste sur l'intérêt de varier la syntaxe et l'interprétation des contraintes. Pour rendre cela possible, il faudra que nos définitions et nos preuves soient *paramétrées* par ces deux éléments. Bien sûr, nous devrons formuler quelques exigences, sans lesquelles nous ne pourrions rien construire ; par exemple, nous exigerons que les inégalités de sous-typage figurent dans le langage des contraintes, et soient effectivement interprétées par une relation d'ordre dans le modèle. Nous exigerons également que la conjonction et la quantification existentielle soient disponibles dans la syntaxe des contraintes, et soient interprétées de façon standard, c'est-à-dire comme dans toute logique du premier ordre.

8.1 Syntaxe et interprétation des contraintes

Supposons donné un ensemble T, dont les éléments, dénotés par t, sont appelés *monotypes* ou *types ground*. On suppose T muni d'un ordre partiel $\leq$. On suppose donnée une fonction totale, notée $\rightarrow$, de $T \times T$ dans T, telle que $t_0 \rightarrow t_1 \leq t'_0 \rightarrow t'_1$ est équivalent à $t'_0 \leq t_0$ and $t_1 \leq t'_1$.

De plus, supposons donné un langage de types et de contraintes, défini de la façon suivante :

$$\begin{array}{rcll}
\tau & ::= & \alpha & \text{variable de types} \\
& | & \tau \rightarrow \tau & \text{type flèche} \\
& | & \ldots & \\
C & ::= & \mathbf{true} & \text{contrainte vraie} \\
& | & \tau \leq \tau & \text{contrainte de sous-typage} \\
& | & C \wedge C & \text{conjonction} \\
& | & \exists \alpha.C & \text{quantification existentielle} \\
& | & \ldots &
\end{array}$$

(Les points de suspension indiquent la présence possible d'autres formes de types et de contraintes.) Ce langage doit être muni d'une interprétation dans T, c'est-à-dire d'un prédicat binaire $\vdash$, dont le premier argument est une fonction totale ρ de l'ensemble des variables de types dans T, et dont le second argument est une contrainte C. Cette interprétation doit satisfaire les lois suivantes :

$$\begin{array}{rcl}
\rho(\tau_1 \rightarrow \tau_2) = \rho(\tau_1) \rightarrow \rho(\tau_2) && \\
\rho \vdash \mathbf{true} && \\
\rho \vdash \alpha_0 \leq \alpha_1 & \iff & \rho(\alpha_0) \leq \rho(\alpha_1) \\
\rho \vdash C_0 \wedge C_1 & \iff & (\rho \vdash C_0) \wedge (\rho \vdash C_1) \\
\rho \vdash \exists \alpha.C & \iff & \exists t \in T \quad \rho[\alpha \mapsto t] \vdash C
\end{array}$$

La donnée du modèle, du langage de types et de contraintes, et de l'interprétation qui les relie, forme le paramètre X dont dépend le système de types que nous allons maintenant présenter. Ce système, proposé par Odersky, Sulzmann et Wehr, est baptisé HM(X), les lettres H et M rappelant Hindley et Milner (à qui on attribue le système ML) et la partie (X) rappelant qu'il s'agit en fait d'une famille de systèmes de types.

8.2 Le système de types HM(X)

Un *schéma de types* σ est de la forme $\forall\bar{\alpha}[C].\tau$; les variables de types $\bar{\alpha}$ sont liées dans C et dans τ. Comme précédemment, on considère les schémas de types comme équivalents modulo α-conversion, et on identifie parfois le type τ avec le schéma trivial $\forall\emptyset[\mathbf{true}].\tau$. Un environnement de typage Γ est, comme précédemment, une séquence de paires de la forme $x : \sigma$.

Comme dans le cas de ML, il existe plusieurs variantes de présentation du système HM(X), toutes équivalentes entre elles. Celle présentée ici emploie deux formes de jugements de typage, à savoir $C, \Gamma \vdash a : \tau$ et $C, \Gamma \vdash a : \sigma$. Ces jugements signifient que, *sous l'hypothèse C concernant les variables de types libres dans le jugement*, et sous l'hypothèse Γ concernant les variables libres dans a, l'expression a admet le type τ, ou (respectivement) le schéma de types σ.

Les règles de typage font fréquemment appel au prédicat *d'implication* entre contraintes, défini comme suit : $C_1 \Vdash C_2$ (C_1 implique C_2) si et seulement si $\rho \vdash C_1$ implique $\rho \vdash C_2$, c'est-à-dire si et seulement si toute solution de C_1 est solution de C_2. L'équivalence de contraintes $\approx$ est la clôture réflexive de l'implication. Dans la suite, nous considérerons les contraintes comme équivalentes modulo $\approx$, c'est-à-dire que nous permettrons de remplacer à tout instant une contrainte par une contrainte équivalente.

On dira qu'un programme a est *bien typé* dans l'environnement Γ si $C, \Gamma \vdash a : \tau$ est dérivable pour une contrainte C *satisfiable*, c'est-à-dire qui admet une solution.

Voici à présent les règles de typage qui définissent le système.

Var
$$\frac{\Gamma(x) = \forall\bar{\alpha}[D].\tau \qquad C \Vdash D}{C, \Gamma \vdash x : \tau}$$

Abs
$$\frac{C, \Gamma; x : \tau \vdash a : \tau'}{C, \Gamma \vdash \texttt{fun}\ x \to a : \tau \to \tau'}$$

App
$$\frac{C, \Gamma \vdash a_1 : \tau_2 \to \tau \qquad C, \Gamma \vdash a_2 : \tau_2}{C, \Gamma \vdash a_1\, a_2 : \tau}$$

∀ Intro
$$\frac{C, \Gamma \vdash a : \tau \qquad \bar{\alpha} \cap \mathcal{L}(\Gamma) = \emptyset}{\exists\bar{\alpha}.C, \Gamma \vdash a : \forall\bar{\alpha}[C].\tau}$$

Let
$$\frac{C, \Gamma \vdash a_1 : \sigma \qquad C, \Gamma; x : \sigma \vdash a_2 : \tau}{C, \Gamma \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau}$$

Sub
$$\frac{C, \Gamma \vdash a : \tau \qquad C \Vdash \tau \leq \tau'}{C, \Gamma \vdash a : \tau'}$$

∃ Intro
$$\frac{C, \Gamma \vdash a : \tau \qquad \bar{\alpha} \cap \mathcal{L}(\Gamma, \tau) = \emptyset}{\exists\bar{\alpha}.C, \Gamma \vdash a : \tau}$$

Les règles Abs et App sont presque inchangées vis-à-vis de ML ; on a simplement ajouté l'hypothèse C, qui n'est pas explicitement utilisée par ces règles.

La règle Var permet de prendre une instance arbitraire d'un schéma de types $\forall\bar{\alpha}[D].\tau$. Dans le cas de ML, prendre une instance d'un schéma consistait à appliquer une substitution de domaine $\bar{\alpha}$ au corps τ. Ici, pour des raisons essentiellement esthétiques, on préfère donner une version de la règle dans laquelle aucune substitution n'apparaît. Cela peut surprendre – l'expressivité de la règle est-elle la même ? En fait, oui. D'abord, n'oublions pas que les schémas sont considérés équivalents modulo α-conversion, ce qui permet d'appliquer à D et à τ un renommage quelconque de domaine $\bar{\alpha}$. Par exemple, voici une instance correcte de la règle Var :

$$\dfrac{\Gamma(x) = \forall\alpha\beta[\alpha \leq \beta].\alpha \to \beta \qquad \alpha' \leq \beta' \Vdash \alpha' \leq \beta'}{\alpha' \leq \beta', \Gamma \vdash x : \alpha' \to \beta'}\ \textsc{Var}$$

Ensuite, la seconde prémisse de la règle Var permet à la conclusion de mentionner une contrainte *plus forte* que celle qui apparaît dans le schéma de types. Ainsi, en supposant $\texttt{int} \leq \texttt{float}$, voici une seconde instance correcte de la règle Var :

$$\dfrac{\Gamma(x) = \forall\alpha\beta[\alpha \leq \beta].\alpha \to \beta \qquad \texttt{int} \leq \alpha' \wedge \beta' \leq \texttt{float} \Vdash \alpha' \leq \beta'}{\texttt{int} \leq \alpha' \wedge \beta' \leq \texttt{float}, \Gamma \vdash x : \alpha' \to \beta'}\ \textsc{Var}$$

Ici, la seconde prémisse est vraie du fait que $\leq$ est interprété comme une relation d'ordre, et du fait que l'interprétation de la conjonction est standard.

Or, la contrainte $\texttt{int} \leq \alpha' \wedge \beta' \leq \texttt{float}$ implique la contrainte $\alpha' \to \beta' \leq \texttt{int} \to \texttt{float}$. Cela découle du fait que la flèche est contravariante dans le modèle et du fait que l'interprétation des contraintes est standard. La règle Sub permet alors de dériver, à partir du jugement ci-dessus :

$$\dfrac{\texttt{int} \leq \alpha' \wedge \beta' \leq \texttt{float}, \Gamma \vdash x : \alpha' \to \beta'}{\texttt{int} \leq \alpha' \wedge \beta' \leq \texttt{float}, \Gamma \vdash x : \texttt{int} \to \texttt{float}}\ \textsc{Sub}$$

Enfin, on note que les variables de types α' et β' utilisées dans cette dérivation sont "internes", au sens où leur nom peut être choisi de façon arbitraire, et en particulier hors de $\mathcal{L}(\Gamma)$. La règle $\exists$ Intro permet alors de les éliminer :

$$\dfrac{\texttt{int} \leq \alpha' \wedge \beta' \leq \texttt{float}, \Gamma \vdash x : \texttt{int} \to \texttt{float}}{\mathbf{true}, \Gamma \vdash x : \texttt{int} \to \texttt{float}}\ \exists\ \textsc{Intro}$$

Dans la conclusion, nous avons remplacé $\exists\alpha'\beta'.(\texttt{int} \leq \alpha' \wedge \beta' \leq \texttt{float})$ par la contrainte équivalente **true**. Une façon possible de lire la règle $\exists$ Intro est : "si une variable de types n'apparaît ni dans l'environnement, ni dans le type associé à l'expression, alors on peut considérer qu'elle est locale à la sous-dérivation". Une autre façon de comprendre cette règle est de la considérer comme une règle de généralisation. En effet, lisons (informellement) le jugement $C, \Gamma \vdash a : \tau$ comme "C et Γ *impliquent* que a admet le type τ". Les noms de variables de types qui apparaissent dans un tel jugement n'ont aucune importance – on peut toujours les renommer – donc on peut généraliser un tel jugement *en tête* : "pour tous $\bar{\alpha}$, C et Γ *impliquent* que a admet le type τ". Si de plus les variables $\bar{\alpha}$ n'apparaissent libres que dans C, et non dans Γ ou τ, alors (toujours informellement) il semble naturel que cette assertion soit équivalente à "s'il existe $\bar{\alpha}$ satisfaisant la contrainte C, alors Γ implique que a admet le type τ". En d'autres termes, le quantificateur universel externe commute alors avec l'implication et devient un quantificateur existentiel en hypothèse – une règle usuelle en mathématiques.

Pour conclure cette digression de façon plus formelle, on peut formuler deux résultats. D'abord, la règle d'instanciation suivante, qui utilise une substitution φ des variables vers les types :

$$\dfrac{\Gamma(x) = \forall\bar{\alpha}[D].\tau \qquad \mathrm{Dom}(\varphi) = \bar{\alpha} \qquad C \Vdash \varphi(D)}{C, \Gamma \vdash x : \varphi(\tau)}$$

est dérivable à l'aide de VAR, SUB et $\exists$ INTRO. Nous l'avons suggéré plus haut ; le lecteur intéressé pourra le démontrer formellement. Ensuite, la règle $\exists$ INTRO n'est qu'une commodité formelle : tout jugement dérivable admet une dérivation dans laquelle $\exists$ INTRO est utilisée une fois et une seule, à la racine. Le lecteur intéressé pourra vérifier cela en démontrant que, moyennant des renommages de variables de types, $\exists$ INTRO commute (vers le bas) avec toutes les règles, c'est-à-dire que dans toute dérivation de typage, les instances de $\exists$ INTRO peuvent être "poussées vers le bas" jusqu'à se retrouver toutes à la racine, où on peut les regrouper en une seule. Il découle de cette remarque que les programmes bien typés sont en fait les mêmes dans le système privé de $\exists$ INTRO et dans le système doté de $\exists$ INTRO. (Les jugements dérivables, cependant, ne sont pas les mêmes ; $\exists$ INTRO permet de dériver plus de jugements, en cachant des variables internes, comme on l'a vu plus haut ; le jugement $\mathbf{true}, \Gamma \vdash x : \mathtt{int} \rightarrow \mathtt{float}$ ne serait pas dérivable en l'absence de cette règle.)

La règle SUB, que nous avons vue plus haut à l'œuvre, permet de remplacer le type de l'expression courante par un supertype. La notion de supertype n'est pas absolue, mais relative à l'hypothèse C. Cela permet d'avoir recours à des hypothèses de sous-typage sur les variables de types. Voici par exemple une instance valide de SUB :

$$\textsc{Sub}\quad \frac{\alpha \leq \beta, \Gamma \vdash a : \beta \rightarrow \mathtt{int} \qquad \alpha \leq \beta \Vdash \beta \rightarrow \mathtt{int} \leq \alpha \rightarrow \mathtt{int}}{\alpha \leq \beta, \Gamma \vdash a : \alpha \rightarrow \mathtt{int}}$$

Si on pense en termes d'inférence de types, plutôt qu'en termes de vérification, et si on voit la contrainte C comme un résultat du typeur, et non comme une donnée, alors on constate que la règle SUB a pour effet d'accumuler dans C toutes les hypothèses sur les variables de types libres dans le jugement dont on a eu besoin pour construire la dérivation de typage. Cette intuition sera confirmée par la description de l'algorithme d'inférence.

Pour conclure notre description du système, restent à examiner $\forall$ INTRO et LET. Ces deux règles forment l'équivalent de la fonction *Gen* et de la règle (let-gen) de ML. On les a séparées ici pour plus de lisibilité, mais on pourrait les fondre en une seule règle LETGEN si on le souhaitait. La règle $\forall$ INTRO est chargée de généraliser un jugement de typage. Elle applique la même condition qu'en ML, à savoir que toutes les variables qui n'apparaissent pas dans l'environnement sont généralisables. La principale nouveauté tient au fait que la contrainte C qui apparaît dans la prémisse devient partie intégrante du schéma de types ainsi créé. En effet, cette contrainte était nécessaire pour que la prémisse soit valide. Il serait donc incorrect de l'oublier ; il faut que toute instance du schéma satisfasse cette contrainte. Voici un exemple d'emploi de la règle $\forall$ INTRO :

$$\frac{\beta \leq \alpha, \emptyset \vdash \mathtt{fun}\ f \rightarrow \mathtt{fun}\ x \rightarrow f\, x : (\alpha \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma)}{\mathbf{true}, \emptyset \vdash \mathtt{fun}\ f \rightarrow \mathtt{fun}\ x \rightarrow f\, x : \forall \alpha\beta\gamma[\beta \leq \alpha].(\alpha \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma)}$$

Le lecteur pourra vérifier que la prémisse est elle-même un jugement valide, et que la contrainte $\beta \leq \alpha$ exprime le fait que x doit être un argument valide pour la fonction f, i.e. son type doit être sous-type de celui attendu par f. Les variables α, β et γ n'apparaissant pas dans l'environnement (qui est vide), on peut les généraliser ; mais on doit bien sûr mémoriser la contrainte $\alpha \leq \beta$ au sein du schéma de types. Ainsi, le schéma admet pour instance $(\mathtt{float} \rightarrow \mathtt{float}) \rightarrow (\mathtt{int} \rightarrow \mathtt{float})$, par exemple, mais pas $(\mathtt{int} \rightarrow \mathtt{float}) \rightarrow (\mathtt{float} \rightarrow \mathtt{float})$, lequel serait certainement incorrect.

On peut se demander pourquoi la conclusion de la règle $\forall$ INTRO mentionne $\exists \bar{\alpha}.C$, et non simplement C. En fait, si elle mentionnait C, on pourrait faire apparaître $\exists \bar{\alpha}.C$ par application de $\exists$ INTRO, puisque les variables $\bar{\alpha}$ ne sont libres ni dans Γ ni dans le schéma de types $\forall \bar{\alpha}[C].\tau$. À l'inverse, si elle mentionne $\exists \bar{\alpha}.C$, on peut faire apparaître C, en vertu du fait que $C \Vdash \exists \bar{\alpha}.C$ est une tautologie et grâce au principe de *renforcement des hypothèses* suivant :

$C, \Gamma \vdash a : \tau$ et $C' \Vdash C$ impliquent $C', \Gamma \vdash a : \tau$.

(Le lecteur intéressé pourra démontrer cette propriété par induction sur la dérivation de $C, \Gamma \vdash a : \tau$.) Du coup, ces deux variantes possibles de la règle $\forall$ INTRO sont équivalentes : chacune est dérivable en termes de l'autre. Le choix est donc purement une question de goût.

Enfin, pour terminer, mentionnons que la règle $\forall$ INTRO donnée ci-dessus est correcte, mais légèrement simplifiée. Elle oblige toutes les hypothèses contenues dans C à être introduites dans le schéma de types créé (donc copiées), alors qu'en réalité il suffit de copier celles qui concernent les variables généralisées $\bar{\alpha}$ uniquement, les autres n'étant pas affectées. Cela peut s'exprimer de la façon suivante, où la contrainte qui figure en prémisse est divisée en deux parties : l'une (C) ne concerne pas les variables généralisées, et reste inchangée ; l'autre (D) les concerne potentiellement et doit être généralisée et incluse dans le schéma.

$$\frac{\begin{array}{l}\forall\ \textsc{Intro}\\ C \wedge D, \Gamma \vdash a : \tau \qquad \bar{\alpha} \cap \mathcal{L}(C, \Gamma) = \emptyset\end{array}}{C \wedge \exists \bar{\alpha}.D, \Gamma \vdash a : \forall \bar{\alpha}[D].\tau}$$

Les énoncés de sûreté de typage, que nous ne démontrerons pas ici, sont les suivants :

Théorème 8.1 (Subject reduction) *$C, \Gamma \vdash a : \tau$ et $\tau \to \tau'$ impliquent $C, \Gamma \vdash a' : \tau$.*

Théorème 8.2 (Progress) *Toute expression close et bien typée est soit une valeur, soit réductible.*

Nous n'avons décrit ici que le fragment le plus élémentaire du langage : le λ-calcul avec `let`. Dans ce langage, aucune erreur de réduction ne peut se produire ; il va de soi qu'il faudrait ajouter au minimum une autre construction (par exemple les paires) pour que le théorème de progression ait un véritable intérêt.

Pour plus d'information, on pourra se référer à

> Martin Odersky, Martin Sulzmann et Martin Wehr. *Type inference with constrained types.* Theory and Practice of Object Systems, 5(1) :35-55, 1999. `http://www.comp.nus.edu.sg/~sulzmann/publications/tapos.ps`
>
> Christian Skalka et François Pottier. *Syntactic Type Soundness for HM(X)*. Proceedings of the 2002 Workshop on Types in Programming (TIP'02), Dagstuhl, Germany, July 2002. `http://pauillac.inria.fr/~fpottier/publis/skalka-fpottier-tip-02.ps.gz`

Le premier contient une preuve de correction basée sur une sémantique dénotationnelle ; le second propose une preuve syntaxique basée sur une sémantique opérationnelle à petits pas, comme dans ce cours.

8.3 Inférence de types

Commençons par une définition auxiliaire, à savoir la relation d'instanciation $\sigma \leq \tau$ entre un schéma de types σ et un type τ. Dans le cas de ML, cette relation était un simple prédicat, c'est-à-dire vraie ou fausse. Ici, les choses sont plus complexes : par exemple, le type `int` $\to \gamma$ est-il instance du schéma $\forall \alpha\beta[\alpha \leq \beta].\alpha \to \beta$? On voit que la réponse ne peut être "oui" ou "non" mais plutôt "oui si la contrainte `int` $\leq \gamma$ est satisfaite". Par conséquent, dans le système HM(X), nous dénoterons par $\sigma \leq \tau$ la contrainte nécessaire et suffisante pour que τ soit instance de σ. Elle est définie de la façon suivante : si $\bar{\alpha} \cap \mathcal{L}(\tau') = \emptyset$, alors $(\forall \bar{\alpha}[C].\tau) \leq \tau'$ est la contrainte $\exists \bar{\alpha}.(C \wedge \tau \leq \tau')$. La condition portant sur $\bar{\alpha}$ peut toujours être satisfaite par α-conversion, et garantit que les variables de τ' ne sont pas capturées lorsqu'on introduit τ' sous la quantification existentielle $\exists \bar{\alpha}$. La condition peut être lue : “pour une certaine valeur des variables $\bar{\alpha}$ satisfaisant C, le corps du schéma τ est un sous-type de τ'”. En d'autres termes, les instances du schéma sont obtenues par instanciation des variables quantifiées universellement, dans la limite où C est satisfaite, et par sous-typage. Pour reprendre l'exemple ci-dessus, la contrainte $\forall\alpha\beta[\alpha \leq \beta].\alpha \to \beta \leq \texttt{int} \to \gamma$ est, par définition, $\exists\alpha\beta.(\alpha \leq \beta \wedge \alpha \to \beta \leq \texttt{int} \to \gamma)$. D'après nos hypothèses sur l'interaction entre la flèche et le sous-typage, cette contrainte est équivalente à $\exists\alpha\beta.(\alpha \leq \beta \wedge \texttt{int} \leq \alpha \wedge \beta \leq \gamma)$. D'après nos hypothèses, la relation de sous-typage est transitive, donc cette contrainte est équivalente à $\texttt{int} \leq \gamma$, comme nous le souhaitions. Comment prouver que cette définition est appropriée ? Cela fera partie des preuves de correction et de complétude de l'algorithme d'inférence, décrit ci-dessous, et où cette définition est utilisée.

L'algorithme d'inférence pour HM(X) peut être présenté sous forme d'une fonction qui à un environnement Γ, une expression a et un type τ, associe une contrainte C nécessaire et suffisante pour que le jugement $C, \Gamma \vdash a : \tau$ soit valide ; nous noterons cette contrainte $[\![\Gamma \vdash a : \tau]\!]$.

L'algorithme est défini de la façon suivante.

$$
\begin{array}{rcl}
[\![\Gamma \vdash x : \tau]\!] & = & \Gamma(x) \leq \tau \\
[\![\Gamma \vdash \texttt{fun}\ x \to a : \tau]\!] & = & \exists\alpha_1\alpha_2.([\![\Gamma; x : \alpha_1 \vdash a : \alpha_2]\!] \wedge \alpha_1 \to \alpha_2 \leq \tau) \\
[\![\Gamma \vdash a_1\, a_2 : \tau]\!] & = & \exists\alpha.([\![\Gamma \vdash a_1 : \alpha \to \tau]\!] \wedge [\![\Gamma \vdash a_2 : \alpha]\!]) \\
[\![\Gamma \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau]\!] & = & [\![\Gamma; x : \forall\alpha[[\![\Gamma \vdash a_1 : \alpha]\!]].\alpha \vdash a_2 : \tau]\!]
\end{array}
$$

Avant de commenter l'algorithme, on pourra en admirer la simplicité. Celle-ci est due en partie au fait qu'il ne s'agit là que d'un algorithme de *génération* de contraintes – la *résolution* étant un problème indépendant.

Les trois premières lignes rappellent beaucoup l'inférence de types pour le λ-calcul monomorphe de la section 3.2, à quelques différences près. D'abord, nous avons remplacé les équations par des contraintes d'instanciation ou de sous-typage. Lorsque l'expression est une variable x, nous exigeons que le type attendu τ soit une instance du schéma de types $\Gamma(x)$. Lorsque l'expression est une fonction, nous demandons que le type effectif $\alpha_1 \to \alpha_2$ soit un sous-type du type attendu τ. Dans le cas de l'application, aucune contrainte de sous-typage n'apparaît explicitement, car les contraintes appropriées sont engendrées directement par les appels récursifs à l'algorithme. Par ailleurs, nous avons utilisé la quantification existentielle pour formaliser l'emploi de variables de types fraîches, comme dans la section 6.5.3. Dans la section 3.2, nous avions supposé qu'une variable de types α_a choisie à l'avance était associée à chaque sous-expression a, ce qui est moins élégant. Ici, on suppose implicitement que les noms des variables fraîches (α_1, α_2 ligne 2 ; α lignes 3 et 4) sont choisis non libres dans Γ ou τ ; cette convention usuelle est appelée convention de Barendregt.

La quatrième ligne analyse d'abord a_1, puis a_2, comme le montre l'imbrication des crochets $[\![\cdot]\!]$. Pour la comprendre, il faut d'abord formuler la remarque suivante :

> Les variables libres dans $[\![\Gamma \vdash a : \tau]\!]$ sont libres dans Γ ou dans τ.

En d'autres termes, la contrainte produite par l'algorithme n'implique aucune variable de types autre que celles présentes dans les données de l'algorithme. Cela est dû au fait que toutes les variables internes employées par l'algorithme ont été quantifiées (existentiellement, lignes 2 et 3 ; universellement, ligne 4) dans la contrainte produite. Cette propriété est naturelle ; si la contrainte produite devait mentionner d'autres variables, le choix de leurs noms serait arbitraire, ce qui rendrait l'algorithme déterministe modulo α-conversion seulement ; c'est le cas de l'algorithme W tel que nous l'avons présenté. Ici, nous avons les moyens techniques d'éviter cette inélégance.

L'intérêt de cette propriété, outre son élégance intrinsèque, tient au fait que, dans la contrainte $[\![\Gamma \vdash a_1 : \alpha]\!]$, il y a une et une seule variable généralisable, à savoir α. En effet, toutes les variables libres dans ce jugement sont libres dans Γ ou sont α ; donc, aucune variable autre que α n'est généralisable. Par ailleurs, α n'est pas libre dans Γ, d'après la convention de Barendregt, donc α est généralisable. Cela explique pourquoi, ligne 4, nous généralisons uniquement α, et formons le schéma de types $\forall\alpha[[\![\Gamma \vdash a_1 : \alpha]\!]].\alpha$. Ce schéma de types exprime exactement tous les types de l'expression a_1, à savoir tous les α satisfaisant la contrainte inférée $[\![\Gamma \vdash a_1 : \alpha]\!]$. Une fois ce schéma obtenu, nous l'associons à la variable x dans l'environnement utilisé pour typer a_2, comme en ML.

La correction de l'algorithme d'inférence est relativement aisée à vérifier, et s'énonce de la façon suivante :

Théorème 8.3 $[\![\Gamma \vdash a : \tau]\!], \Gamma \vdash a : \tau$.

En d'autres termes, la contrainte inférée est effectivement suffisante pour garantir que a admet le type τ dans l'environnement Γ. Nous invitons le lecteur à vérifier cet énoncé, par induction sur la structure de l'expression a. On constatera en particulier que l'introduction des quantificateurs existentiels est justifiée par la règle $\exists$ Intro.

La complétude (ou principalité) de l'algorithme peut s'énoncer comme suit :

Théorème 8.4 *Si* $C, \Gamma \vdash a : \tau$*, alors* $C \Vdash [\![\Gamma \vdash a : \tau]\!]$.

Ainsi, toute contrainte C suffisante pour garantir que a admet le type τ dans l'environnement Γ contient la contrainte inférée. En d'autres termes, la contrainte inférée est nécessaire. Cet énoncé ce démontre par induction sur la dérivation du jugement de typage, mais il est nécessaire, pour cela, de le généraliser légèrement. On pourra se référer à une note de Sulzmann (*Constraint-based type inference*, 14 octobre 2000, disponible électroniquement).

8.4 Quelques mots de résolution

Une fois les contraintes engendrées, reste à les résoudre, et c'est en fait le plus délicat, car il est difficile de concevoir et de mettre au point des algorithmes de résolution efficaces. De plus, la nature de l'algorithme pourra varier largement en fonction de la nature (syntaxe et interprétation) des contraintes ; elle est donc fonction du paramètre X.

Dans le cas où les contraintes sont des équations, c'est-à-dire en l'absence de sous-typage ou lorsque le sous-typage est interprété par l'égalité, on peut utiliser l'algorithme d'unification de la section 6.5.3. On obtient alors un algorithme d'inférence équivalent à l'algorithme W – on peut le démontrer – mais présenté de façon assez différente.

Dans le cas où le sous-typage est non trivial, c'est-à-dire non réduit à l'égalité, l'algorithme de résolution de contraintes aura typiquement recours à un calcul de clôture transitive – c'est-à-dire qu'à chaque fois que $\tau_1 \leq \alpha \wedge \alpha \leq \tau_2$ apparaît dans la contrainte, on y ajoutera $\tau_1 \leq \tau_2$ – combiné avec des règles de décomposition structurelle – comme celle qui, à chaque fois que $\tau_0 \to \tau_1 \leq \tau'_0 \to \tau'_1$ apparaît, ajoute $\tau'_0 \leq \tau_0 \wedge \tau_1 \leq \tau'_1$. Les détails dépendent de la nature de l'ordre de sous-typage (par exemple, est-ce ou non un treillis ?), mais ces deux principes forment la base de la plupart des algorithmes de résolution de contraintes de sous-typage. La difficulté provient du fait que cette combinaison de clôture transitive et décomposition structurelle ne semble pas pouvoir être effectuée en une complexité inférieure à $O(n^3)$, où n est la taille initiale de la contrainte. Cela est beaucoup plus coûteux que l'unification, dont le comportement est linéaire en pratique, et justifie la nécessité d'algorithmes capables d'effectuer des *simplifications* au vol de façon à limiter le coût de la résolution.

8.5 Quelques mots de simplification

Le système permet de remplacer à tout instant une contrainte par une contrainte équivalente. On peut tirer parti de cette propriété en pour *simplifier* les contraintes ; toutes les contraintes équivalentes n'ont pas la même représentation textuelle, et on peut faire en sorte que la nouvelle contrainte soit de taille inférieure à celle de l'ancienne, ou bien soit plus lisible (selon un critère subjectif) que l'ancienne. Par exemple, du fait que le sous-typage est transitif, les contraintes $\exists\beta.(\alpha \leq \beta \wedge \beta \leq \gamma)$ et $\alpha \leq \gamma$ sont équivalentes ; on pourra donc remplacer la première par la seconde. On notera que $\alpha \leq \beta \wedge \beta \leq \gamma$ n'est pas équivalente à $\alpha \leq \gamma$: il est essentiel que β soit locale. On comprend mieux alors le rôle de la règle $\exists$ INTRO : en signalant explicitement la présence de variables locales, elle autorise certaines simplifications.

Si on le souhaite, on peut définir, outre la relation d'équivalence entre contraintes, une relation d'équivalence entre schémas de types. On dira que σ_1 et σ_2 sont équivalents si et seulement si, pour toute variable de types α non libre dans σ_1 et σ_2, les contraintes d'instanciation $\sigma_1 \leq \alpha$ et $\sigma_2 \leq \alpha$ sont équivalentes. En d'autres termes, deux schémas de types sont équivalents si et seulement s'ils ont les mêmes instances. Le lecteur pourra vérifier par exemple que, si $\bar{\beta} \cap \mathcal{L}(\tau) = \emptyset$, alors les schémas $\forall\bar{\alpha}[\exists\bar{\beta}.C].\tau$ et $\forall\bar{\alpha}\bar{\beta}[C].\tau$ sont équivalents. En d'autres termes, dans un schéma de types contraint, les variables internes à la contrainte et les variables universellement quantifiées jouent le même rôle. Cela ne doit pas être une surprise ; nous avons indiqué plus haut que l'emploi de la quantification existentielle dans les contraintes est essentiellement une commodité, laquelle permet une gestion élégante des noms de variables de types, mais n'ajoute pas de puissance au langage.

8.6 Quels paramètres X ?

Si l'ensemble T est une algèbre libre de types finis, et si le sous-typage est réduit à l'égalité, nous obtenons un système équivalent à ML, comme indiqué plus haut.

Si – sans changer la syntaxe des types ni des contraintes – l'ensemble T est remplacé par une algèbre de types *réguliers*, nous obtenons un système équivalent à ML avec types récursifs. La syntaxe ne contenant aucune construction particulière pour dénoter les types récursifs, ceux-ci doivent être codés à l'aide de contraintes ; par exemple, $\alpha = \alpha \rightarrow \alpha, \emptyset \vdash \texttt{fun}\ x \rightarrow x\,x : \alpha$ est un jugement valide ; la contrainte est satisfiable $\alpha = \alpha \rightarrow \alpha$ dans un tel T, donc le programme $\texttt{fun}\ x \rightarrow x\,x$ est bien typé.

Si on ajoute les constructions $(\ell : \tau; \tau)$ et $\partial\tau$ à la syntaxe des types, et si, pour les interpréter, on ajoute à T des fonctions totales des étiquettes vers les types (des "rangées ground"), alors on obtient un système doté de rangées, équivalent à celui présenté au chapitre 6. La différence est que, ici, nous n'introduisons aucune théorie équationnelle sur les types "syntaxiques" τ. Une équation telle que $(\ell : \tau; \partial\tau) = \partial\tau$ étant "vraie dans le modèle" (c'est-à-dire satisfaite par toute valuation ρ), elle est impliquée par toute contrainte C. La règle SUB permet donc d'utiliser cette équation librement. Notons qu'au chapitre 6, prouver la correction de ML avec rangées demandait en principe de revoir tous les lemmes précédents, à cause de l'introduction de cette théorie équationnelle ; ici, la correction découle de la correction de HM(X).

Enfin, on peut imaginer d'autres formes de contraintes. La quantification universelle $\forall\alpha.C$ peut être utile lorsque l'on souhaite ajouter au langage une forme de polymorphisme plus puissante que celle de ML. La disjonction $C \vee C$ et/ou la négation $\neg C$ permettent, en principe, une grande expressivité, mais conduisent souvent à des problèmes de résolution d'une complexité importante (exponentielle), voire indécidables. Les *contraintes conditionnelles* $c \Rightarrow C$, où c est une condition exprimée dans un langage très restreint, peuvent être considérées comme une forme faible de disjonction, dont l'expressivité reste intéressante, et dont la complexité reste faible (polynomiale).

Chapitre 9

Systèmes de modules

[Avertissement : ce chapitre n'a pas été revu cette année, et les notations sont par endroits incohérentes vis-à-vis des chapitres précédents. En particulier, les environnements de typage sont notés E et les contextes d'évaluation Γ.]

Pour conclure ce cours, nous étendons les systèmes de types vers la programmation modulaire (programmation à grande échelle). Nous suivons l'approche ML où le système de modules se présente comme un petit langage fonctionnel typé opérant au-dessus du langage de base. Le langage de base considéré ici est mini-ML, mais l'approche s'étend facilement à une classe très vaste de langages de base typés.

9.1 Un calcul de modules

La syntaxe des expressions de modules et des types de modules est la suivante :

Chemins d'accès :	p	$::= X$	identificateur de module
		$\mid p.X$	accès à un sous-module d'un chemin
Modules :	m	$::= p$	module nommé (identificateur ou chemin)
		$\mid \texttt{struct}\ d_1; \dots; d_n\ \texttt{end}$	définition de structure
		$\mid \texttt{functor}\ (X : M) \to m$	définition de foncteur
		$\mid m_1\ m_2$	application de foncteur
		$\mid (m : M)$	contrainte de signature
Définitions :	d	$::= \texttt{let}\ x = a$	définition de valeur
		$\mid \texttt{type}\ t = \tau$	définition de type
		$\mid \texttt{module}\ X = m$	définition de sous-module
Types de modules :	M	$::= \texttt{sig}\ S_1; \dots; S_n\ \texttt{end}$	signature
		$\mid \texttt{functor}\ (X : M) \to M'$	type de foncteur
Spécifications :	S	$::= \texttt{val}\ x : \sigma$	spécification de valeur
		$\mid \texttt{type}\ t$	spécification de type abstrait
		$\mid \texttt{type}\ t = \tau$	spécification de type manifeste
		$\mid \texttt{module}\ X : M$	spécification de sous-module

Le langage de base (expressions a, types τ et schémas σ) est mini-ML auquel on ajoute la possibilité de faire référence à des noms ou des composantes de valeurs ou de types :

Expressions :	a	$::= p.x$	composante de valeur d'une structure
		$\mid x \mid c \mid op \mid \texttt{fun}\ x \to a \mid a_1\ a_2$	
		$\mid (a_1, a_2) \mid \texttt{let}\ x = a_1\ \texttt{in}\ a_2$	comme d'habitude
Types :	τ	$::= t$	type nommé
		$\mid p.t$	composante de type d'une structure
		$\mid \alpha \mid T \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$	comme d'habitude
Schémas de types :	σ	$::= \forall \alpha_1 \ldots \alpha_n.\ \tau$	

9.2 Évaluation

9.2.1 Sémantique par traduction

L'évaluation des modules s'exprime très facilement par une traduction dans mini-ML avec enregistrements : on efface les composantes de types des structures, on transforme les structures en enregistrements, et les foncteurs en fonctions. Cette traduction, notée $[\![\cdots]\!]$, est définie par :

$$
\begin{array}{rcl}
[\![X]\!] & = & X \\
[\![p.X]\!] & = & [\![p]\!].X \\
[\![\texttt{struct}\ d^*\ \texttt{end}]\!] & = & L(d^*, \{\ldots x_i = x_i; \ldots ; X_j = X_j; \ldots\}) \\
 & & \text{où les } x_i \text{ et } X_j \text{ sont les identificateurs de valeurs} \\
 & & \text{et de modules définis dans } d^* \\
[\![\texttt{functor}\ (X : M) \to m]\!] & = & \texttt{fun}\ X \to [\![m]\!] \\
[\![m\ m']\!] & = & [\![m]\!]\ [\![m']\!] \\
\\
L(\varepsilon, a) & = & a \\
L(\texttt{let}\ x = a;\ d^*, b) & = & \texttt{let}\ x = a\ \texttt{in}\ L(d^*, b) \\
L(\texttt{type}\ t = T;\ d^*, b) & = & L(d^*, b) \\
L(\texttt{module}\ X = m;\ d^*, b) & = & \texttt{let}\ X = [\![m]\!]\ \texttt{in}\ L(d^*, b)
\end{array}
$$

Exemple : l'expression de module

```
functor (X : sig type t ; val v : t end) →
    struct type u = X.t ; val y = X.v ; val z = (y, y) end
```

a pour traduction

```
fun X → let y = X.v in let z = (y,y) in { y = y ; z = z }
```

9.2.2 Sémantique à réduction

Le calcul de modules défini ci-dessus ne se prête pas bien à une sémantique par réduction. La raison est que la classe syntaxique p des chemins d'accès n'est pas stable par substitution (d'un identificateur de module par une expression de module). Par exemple, l'expression de type $X.t$, où X est un identificateur de module, n'est plus bien formée une fois que X est substitué par une expression de module qui n'est pas un chemin d'accès, p.ex. $\texttt{struct} \ldots \texttt{end}$: on obtient l'expression de type $(\texttt{struct} \ldots \texttt{end}).t$, qui est mal formée.

Une solution simple à ce problème est d'autoriser toute expression de module m dans les projections $.x$, $.t$ et $.X$. Autrement dit, on prendrait $p ::= m$ dans la grammaire ci-dessus. Le problème de cette approche est que les expressions de types $p.t$ deviennent extrêmement compliquées : p peut être une expression de module arbitrairement complexe. En particulier, il devient difficile de décider si deux expressions de types $p.t$ et $p'.t$ représentent le même type : il faudrait réduire (évaluer) p et p' *pendant le typage* pour voir s'ils se réduisent sur des valeurs de modules identiques. Une telle évaluation pourrait ne pas terminer, rendant le typage indécidable. De manière générale, nous voulons préserver une certaine *distinction de phase* entre le typage (pendant la compilation) et l'évaluation (pendant l'exécution) ; devoir réduire des expressions pendant le typage brouille cette distinction.

Une solution intermédiaire, proposée par Harper et Lillibridge, consiste à ajouter à la classe des chemins d'accès les valeurs de modules, c'est-à-dire les expressions de modules entièrement évaluées :

Chemins d'accès :	p	$::= X \mid p.X \mid V$
Valeurs de modules :	V	$::= \texttt{struct}\ d_V^*\ \texttt{end} \mid \texttt{functor}(X : M) \rightarrow a$
Définitions entièrement évaluées : :	d_V	$::= \texttt{let}\ x = v \mid \texttt{type}\ t = \tau \mid \texttt{module}\ X = V$
Valeurs du langage de base :	v	$::= c \mid op \mid \texttt{fun}\ x \rightarrow a \mid (v_1, v_2)$

Les valeurs de modules, étant déjà entièrement réduites, se comparent simplement par égalité syntaxique. Cela permet de décider facilement si deux types $V_1.t$ et $V_2.t$ sont identiques.

Dans l'approche Harper-Lillibridge, la grammaire du langage n'est toujours pas stable par substitution $[X \leftarrow m]$ où m est une expression de module arbitraire ; mais elle est stable par substitution $[X \leftarrow V]$ où V est une valeur de module. C'est suffisant pour donner une sémantique à réduction en appel par valeur :

$$\begin{array}{rcll} (\texttt{struct}\ d_V^*; \texttt{val}\ x = v; d_V^{*\prime}\ \texttt{end}) & \stackrel{\varepsilon}{\rightarrow} & v & \text{(proj-val)} \\ (\texttt{struct}\ d_V^*; \texttt{module}\ X = V; d_V^{*\prime}\ \texttt{end}) & \stackrel{\varepsilon}{\rightarrow} & V & \text{(proj-mod)} \\ (\texttt{functor}\ (X : M) \rightarrow m)\ V & \stackrel{\varepsilon}{\rightarrow} & m[X \leftarrow V] & \beta_{\text{functor}} \\ (V : M) & \stackrel{\varepsilon}{\rightarrow} & V & \text{(contrainte)} \end{array}$$

Pour tenir compte du fait qu'une composante de structure peut faire référence aux noms des composantes précédentes (p.ex. `struct val x = 1 ; val y = x + 1 end`), il faut ajouter les deux règles de "propagation" de valeurs suivantes :

$$\begin{array}{rcll} \texttt{struct}\ d_V^*; \texttt{let}\ x = v; d^*\ \texttt{end} & \stackrel{\varepsilon}{\rightarrow} & \texttt{struct}\ d_V^*; \texttt{let}\ x = v; d^*[x \leftarrow v]\ \texttt{end} & \text{(prop-val)} \\ \texttt{struct}\ d_V^*; \texttt{module}\ X = V; d^*\ \texttt{end} & \stackrel{\varepsilon}{\rightarrow} & \texttt{struct}\ d_V^*; \texttt{module}\ X = V; d^*[X \leftarrow V]\ \texttt{end} & \text{(prop-mod)} \end{array}$$

Enfin, pour les réductions en profondeur, on utilise les contextes suivants :

Contextes de modules :

$$\begin{array}{rl} \Gamma_m & ::= [\,] \mid \Gamma_m.X \\ & \mid \texttt{struct}\ d_V^*; \texttt{let}\ x = \Gamma_a; d^*\ \texttt{end} \\ & \mid \texttt{struct}\ d_V^*; \texttt{module}\ X = \Gamma_m; d^*\ \texttt{end} \\ & \mid \Gamma_m\ m \mid V\ \Gamma_m \mid (\Gamma : M) \end{array}$$

Contextes d'expressions de base :

$$\Gamma_a \ ::= [\,] \mid \Gamma_m.x \mid \Gamma_a\ a \mid v\ \Gamma_a \mid (\Gamma_a, a) \mid (v, \Gamma_a) \mid \texttt{let}\ x = \Gamma_a\ \texttt{in}\ a$$

9.3 Règles de typage

Le typage des modules soulève plusieurs difficultés. Premièrement, il faut prendre en compte les équivalences entre types de base induites par les types manifestes dans les signatures. Par exemple, si X : $\texttt{sig type } t = \texttt{ int end}$, alors le type $X.t$ est équivalent à `int`. Dans les chapitres précédents, nous avons vu plusieurs exemples de théories équationnelles ajoutées à l'algèbre de types. La différence dans le cas des modules est que les équations ne sont pas fixées une fois pour toute, mais dépendent des types qui peuvent être donnés aux modules.

La seconde difficulté est de prendre en compte l'oubli d'informations concernant les signatures : on peut oublier la présence de certains composants ; on peut aussi oublier des égalités de types en voyant un type manifeste comme un cas particulier de type abstrait. Ceci est reflété dans une relation de sous-typage entre types de modules, avec une règle de subsomption implicite (nous ne faisons pas d'inférence de types sur le langage de modules).

La dernière difficulté est la prise en compte des dépendances entre composantes de structures, d'une part, et de l'autre entre l'argument et le résultat d'un foncteur. Par exemple, le foncteur

```
functor (X : sig type t end) → struct type u = X.t * X.t end
```

reçoit le type

```
functor (X : sig type t end) → sig type u = X.t * X.t end
```

dans lequel la signature du résultat fait intervenir le nom `X` du paramètre formel.

9.3.1 Équivalence entre types de base

L'équivalence entre types du langage de base est définie par le prédicat $E \vdash \tau_1 \approx \tau_2$. Comme dans tous les prédicats de typage de cette section, l'environnement E contient des hypothèses de typage sur les identificateurs de valeurs, de types, et de modules :

$$E = \ldots; \texttt{val } v : \sigma; \ldots; \texttt{type } t; \ldots; \texttt{type } t = \tau; \ldots; \texttt{module } X : M; \ldots$$

L'environnement E est en fait une séquence de spécifications $S_1; \ldots S_n$, tout comme l'intérieur d'une signature `sig...end`.

Les règles définissant le prédicat $E \vdash \tau_1 \approx \tau_2$ sont :

$$\frac{E = E_1;\ \texttt{type } t = \tau;\ E_2}{E \vdash t \approx \tau}\ \text{(eq-manifeste)} \qquad \frac{E \vdash p : \texttt{sig } S_1^*;\ \texttt{type } t = \tau;\ S_2^* \texttt{ end}}{E \vdash p.t \approx \tau\{z \leftarrow p.z \mid z \text{ lié dans } S_1^*\}}\ \text{(eq-proj)}$$

$$E \vdash \tau \approx \tau\ \text{(eq-réflexivité)} \qquad \frac{E \vdash \tau_1 \approx \tau_2}{E \vdash \tau_2 \approx \tau_1}\ \text{(eq-symétrie)}$$

$$\frac{E \vdash \tau_1 \approx \tau_2 \qquad E \vdash \tau_2 \approx \tau_3}{E \vdash \tau_1 \approx \tau_3}\ \text{(eq-transitivité)}$$

$$\frac{E \vdash \tau[\alpha_i \leftarrow \tau_i] \approx \tau'}{E \vdash \forall \alpha_1 \ldots \alpha_n.\ \tau \geq \forall \beta_1 \ldots \beta_p.\ \tau'}\ \text{(schémas)}$$

La règle (eq-projection) est compliquée par le fait que les composantes de la signature affectée à p peuvent dépendre les unes des autres. Par conséquent, le type τ manifestement égal à la composante t peut faire référence à des identificateurs de types et de modules liés au début de la signature. Il n'est donc pas correct en général de dire que $E \vdash p.t \approx \tau$, car certains identificateurs de τ vont être sortis de leur portée et devenir libre. Il faut au contraire préfixer par p les identificateurs X et t liés précédemment dans la signature.

Exemple : supposons $E \vdash p : \texttt{sig type } t; \texttt{type } u; \texttt{type } v = u \times t \texttt{ end}$. Il n'est pas correct de dire $E \vdash p.v \approx u \times t$, car ce type fait référence à des identificateurs de types u et t inconnus. En revanche, nous pouvons déduire $E \vdash p.v \approx (u \times t)[t \leftarrow p.t; u \leftarrow p.u]$, c'est-à-dire $E \vdash p.v \approx p.u \times p.t$. Le type qui s'appelle u à l'intérieur de la signature est référencé à l'extérieur de la signature par $p.u$, et de même pour t.

9.3.2 Typage du langage de base

Les règles de typage du langage de base sont essentiellement celles de mini-ML sans modules, avec prise en compte de l'équivalence $\approx$ entre types.

$$\frac{E = E_1; \texttt{val } x : \sigma; E_2 \qquad \tau \leq \sigma}{E \vdash x : \tau} \text{ (var-inst)}$$

$$\frac{E \vdash p : \texttt{sig } S_1^*; \texttt{ val } x : \sigma; S_2^* \texttt{ end} \qquad \tau \leq \sigma\{z \leftarrow p.z \mid z \text{ lié dans } S_1^*\}}{E \vdash p.x : \tau} \text{ (val-projection)}$$

$$\frac{E \vdash a : \tau \qquad E \vdash \tau \approx \tau'}{E \vdash a : \tau'} \text{ (equiv)}$$

Pour l'accès aux composantes de valeurs de structures (règle (proj-val)), on effectue la même opération de "préfixage" par p décrite plus haut à propos de la règle (eq-proj).

Les autres règles de typage sont reprises directement de celles du chapitre 1 :

$$\frac{TC(c) \leq \tau}{E \vdash c : \tau} \text{ (const-inst)} \qquad \frac{TC(op) \leq \tau}{E \vdash op : \tau} \text{ (op-inst)} \qquad \frac{E; \texttt{val } x : \tau_1 \vdash a : \tau_2}{E \vdash (\texttt{fun } x \rightarrow a) : \tau_1 \rightarrow \tau_2} \text{ (fun)}$$

$$\frac{E \vdash a_1 : \tau' \rightarrow \tau \qquad E \vdash a_2 : \tau'}{E \vdash a_1\ a_2 : \tau} \text{ (app)} \qquad \frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2} \text{ (paire)}$$

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

Remarquons que pendant le typage d'une expression a, on n'ajoute pas d'hypothèses `type t`... ou `module` X : ... à l'environnement. Par conséquent, la relation d'équivalence entre types $\approx$ est constante pendant le typage d'une expression. Cela permet de reprendre des résultats généraux concernant le typage modulo une théorie équationnelle. Par exemple, l'inférence du type principal d'une expression a apparaissant comme composante de valeur d'une structure `struct` ... `val` $x = a \ldots$ `end` est possible, en utilisant l'algorithme W et l'unification modulo la relation $\approx$.

9.3.3 Sous-typage entre types de modules

Nous définissons maintenant une relation de sous-typage entre types de modules $E \vdash M_1 <: M_2$ qui reflète les deux possibilités d'oubli d'information dans les signatures mentionnées plus haut (oubli de composantes et transformation de types manifestes en types abstraits). Au passage, nous définissons également le sous-typage entre deux spécifications $E \vdash S_1 <: S_2$.

$$\frac{\rho : \{1 \dots n\} \mapsto \{1 \dots m\} \qquad E; S_1; \dots ; S_m \vdash S_{\rho(i)} <: S'_i \text{ for } i = 1, \dots , n}{E \vdash (\texttt{sig}\ S_1; \dots ; S_m\ \texttt{end}) <: (\texttt{sig}\ S'_1; \dots ; S'_n\ \texttt{end})} \text{ (sub-sig)}$$

$$\frac{E \vdash P_2 <: P_1 \qquad E; \texttt{module}\ X : P_2 \vdash R_1 <: R_2}{E \vdash (\texttt{functor}\ (X : P_1) \to R_1) <: (\texttt{functor}\ (X : P_2) \to R_2)} \text{ (sub-foncteur)}$$

$$\frac{E \vdash \sigma \geq \sigma'}{E \vdash (\texttt{val}\ v : \sigma) <: (\texttt{val}\ v : \sigma')} \text{ (sub-val)}$$

$$E \vdash (\texttt{type}\ t) <: (\texttt{type}\ t) \text{ (sub-abstr-abstr)} \qquad E \vdash (\texttt{type}\ t = \tau) <: (\texttt{type}\ t) \text{ (sub-mani-abstr)}$$

$$\frac{E \vdash \tau \approx \tau'}{E \vdash (\texttt{type}\ t = \tau) <: (\texttt{type}\ t = \tau')} \text{ (sub-mani-mani)}$$

$$\frac{E \vdash \tau \approx t}{E \vdash (\texttt{type}\ t) <: (\texttt{type}\ t = \tau)} \text{ (sub-abstr-mani)}$$

$$\frac{E \vdash M <: M'}{E \vdash (\texttt{module}\ X : M) <: (\texttt{module}\ X : M')} \text{ (sub-mod)}$$

Sous-typage entre signatures

La règle (sub-sig) exprime qu'une signature $M = \texttt{sig}\ S_1; \dots ; S_m\ \texttt{end}$ est sous-type d'une autre signature $M' = \texttt{sig}\ S'_1; \dots ; S'_n\ \texttt{end}$ si, à chaque composante S'_i de M' (la moins précise des deux), on peut associer une composante $S_{\rho(i)}$ de M (la plus précise des deux signatures) telle que $S_{\rho(i)} <: S'_i$. Remarquons que M peut très bien contenir plus de composantes que M' ; ces composantes ne sont simplement pas associées à aucune composante de M'. Cela correspond à l'oubli de ces composantes de M, ou encore au sous-typage "en largeur" entre signatures.

La relation de sous-typage entre composantes de signatures $E \vdash S <: S'$ est définie de telle sorte que les deux composantes S et S' doivent être du même "genre" (deux spécifications de valeurs, ou deux spécifications de types, ou deux spécification de modules, mais pas une valeur et un type, par exemple) et spécifier des identificateurs de même nom (deux valeurs nommées x, mais pas une valeur nommée x et une nommée y par exemple). Par conséquent, pour déterminer l'existence de la correspondance ρ entre les composantes des deux signatures, il suffit de regarder les genres et les noms de ces composantes. Si nous supposons de plus que les composantes d'une signature ont toutes des noms distincts (on s'arrangera pour que ce soit le cas dans la règle de typage des structures), il y a au plus une correspondance ρ qui convient, et elle est entièrement déterminée par les genres et les noms des composantes des deux signatures.

Le sous-typage entre composantes de signatures $E \vdash S <: S'$ se comprend facilement en voyant les composantes de signatures comme des spécifications : S est "sous-type" de S' si toute déclaration (composante de structure) satisfaisant la spécification S satisfait aussi S'.

- Pour deux spécifications de valeurs $\texttt{val}\ x : \sigma$ et $\texttt{val}\ x : \sigma'$, il faut que le schéma σ soit plus général que σ' (i.e. σ au moins aussi polymorphe que σ'), de manière à ce que toute instance de σ soit aussi instance de σ'.

– Pour deux spécifications de modules `module` $X : M$ et `module` $X : M'$, il faut que M soit sous-type de M'.
– Pour deux spécifications de types, on a quatre cas :
  – abstrait, abstrait : (`type` t) <: (`type` t) est toujours vrai ;
  – manifeste, abstrait : (`type` $t = \tau$) <: (`type` t) est toujours vrai ;
  – manifeste, manifeste : (`type` $t = \tau$) <: (`type` $t = \tau'$) est vrai si et seulement si τ et τ' sont équivalents ;
  – abstrait, manifeste : (`type` t) <: (`type` $t = \tau'$) est vrai si on peut prouver l'équivalence $t \approx \tau'$ à partir des autres hypothèses.

Comme les composantes des signatures peuvent faire référence à des identificateurs liés par d'autres composantes, on ne peut pas les comparer dans l'environnement initial E : il faut ajouter à E des hypothèses de typage sur les identificateurs liés par les signatures. Comme tous les identificateurs liés par M' (la plus grande des deux signatures) le sont aussi par M (la plus petite), et ont a priori des types plus précis dans M, il suffit d'ajouter à E toutes les composantes $S_1 \ldots S_n$ de M. C'est pourquoi la règle (sub-sig) compare les composantes deux à deux dans $E; S_1; \ldots; S_n$.

Exemples de sous-typage entre signatures :
Oubli d'une composante :

```
(sig type t = int ; val v : t end) < : (sig type t = int end) < : (sig end)
```

En particulier, la signature vide `sig end` est supertype de toutes les signatures.
Oubli d'une égalité de type :

```
(sig type t = int ; val v : t end) < : (sig type t ; val v : t end)
```

Utilisation d'une égalité de type fournie par une composante oubliée pour sous-typer les autres composantes :

```
(sig type t = int ; val v : t end) < : (sig val v : int end)
```

Permutation de composantes indépendantes :

```
(sig type t = int ; val v : int end) < : (sig val v : int ; type t = int end)
```

Permutation de composantes dépendantes (c'est ici que la règle (sub-abstr-mani) se révèle nécessaire) :

```
(sig type t ; type u = t end) < : (sig type u ; type t = u end)
```

Exemples de non sous-typage entre signatures :
Il manque une composante dans M :

```
(sig type t = int end) ≮: (sig type t = int val v : t end)
```

Erreur sur le nom d'une composante :

```
(sig type t = int val n : t end) ≮: (sig type t = int val v : t end)
```

Désaccord sur la définition d'un type manifeste :

```
(sig type t = int end) ≮: (sig type t = bool end)
```

Un type abstrait n'est pas prouvablement égal à un type manifeste :

```
(sig type t end) ≮: (sig type t = bool end)
```

Sous-typage entre types de foncteurs

Comme pour les types de fonctions, le sous-typage entre types de foncteurs est contravariant en le type argument des foncteurs, et covariant en leur type résultat : $E \vdash (\texttt{functor}\ (X : P_1) \rightarrow R_1) <: (\texttt{functor}\ (X : P_2) \rightarrow R_2)$ si $E \vdash P_2 <: P_1$ (contravariance) et $E; \texttt{module}\ X : P_2 \vdash R_1 <: R_2$ (covariance). Comme les types résultat R_1 et R_2 peuvent dépendre de X, il serait incorrect de les comparer dans l'environnement E ; il faut ajouter une hypothèse sur X. On a le choix entre les hypothèses $X : P_1$ et $X : P_2$, qui sont toutes deux sémantiquement correctes. Cependant, l'hypothèse $X : P_2$ est plus précise (car P_2 est plus petit que P_1) et permet de dériver davantage de sous-typages intéressants. Par exemple, on a

$$
\begin{aligned}
&(\texttt{functor}\ (X : \texttt{sig type}\ t\ \texttt{end}) \rightarrow \texttt{sig type}\ t = X.t\ \texttt{end})\\
&\quad <: \quad (\texttt{functor}\ (X : \texttt{sig type}\ t = \texttt{int end}) \rightarrow \texttt{sig type}\ t = \texttt{int end})
\end{aligned}
$$

car sous l'hypothèse $X : \texttt{sig type}\ t = \texttt{int end}$ on peut prouver que $X.t \approx \texttt{int}$, et donc que $\texttt{sig type}\ t = X.t\ \texttt{end} <: \texttt{sig type}\ t = \texttt{int end}$. Ce ne serait pas possible sous l'hypothèse $X : \texttt{sig type}\ t\ \texttt{end}$.

9.3.4 Typage du langage de modules

Les règles de typage pour le langage de modules sont de la forme $E \vdash m : M$ (le module m a le type de module M) et $E \vdash d^* : S^*$ (la séquence de déclarations d^* a pour spécification de types S^*).

$$\frac{E = E_1;\ \texttt{module}\ X : M;\ E_2}{E \vdash X : M}\ \text{(mod-var)}$$

$$\frac{E \vdash p : \texttt{sig}\ S_1^*;\ \texttt{module}\ X : M;\ S_2^*\ \texttt{end}}{E \vdash p.X : M\{z \leftarrow p.z \mid z \text{ lié dans } S_1^*\}}\ \text{(mod-projection)}$$

$$\frac{E \vdash m : M}{E \vdash (m : M) : M}\ \text{(mod-contrainte)} \qquad \frac{E \vdash m : M \qquad E \vdash M <: M'}{E \vdash m : M'}\ \text{(mod-sub)}$$

Typage des structures

$$\frac{E \vdash d^* : S^*}{E \vdash (\texttt{struct}\ d^*\ \texttt{end}) : (\texttt{sig}\ S^*\ \texttt{end})}\ \text{(struct)}$$

$$E \vdash \varepsilon : \varepsilon\ \text{(struct-vide)} \qquad \frac{E \vdash a : \tau \qquad E;\ \texttt{val}\ v : \tau \vdash d^* : S^*}{E \vdash (\texttt{let}\ v = a;\ d^*) : (\texttt{val}\ v : \tau;\ S^*)}\ \text{(struct-val)}$$

$$\frac{t \notin \mathrm{Dom}(E) \qquad E;\ \texttt{type}\ t = \tau \vdash d^* : S^*}{E \vdash (\texttt{type}\ t = \tau;\ d^*) : (\texttt{type}\ t = \tau;\ S^*)}\ \text{(struct-type)}$$

$$\frac{E \vdash m : M \qquad X \notin \mathrm{Dom}(E) \qquad E;\ \texttt{module}\ X : M \vdash d^* : S^*}{E \vdash (\texttt{module}\ X = m;\ d^*) : (\texttt{module}\ X : M;\ S^*)}\ \text{(struct-mod)}$$

Lorsqu'on type une struture, les définitions de types reçoivent des types manifestes, et sont donc transparentes par défaut. Par exemple, la structure

```
module S = struct type t = int ; let x = 0 end
```

reçoit la signature `sig type t = int ; val x : int end`, ou de manière équivalente `sig type t = int ; val x : t end`. Dans les deux cas, nous pouvons typer `1 + S.f 0` par exemple.

Pour rendre abstrait un type, il faut utiliser une contrainte de signature :

```
module S = (struct type t = int ; let x = 0 end : sig type t ; val x: t end)
```

Les conditions $t \notin \mathrm{Dom}(E)$ et $X \notin \mathrm{Dom}(E)$ dans les règles (struct-type) et (struct-mod) évitent des incohérences de typage lorsqu'on redéfinit un nom de type ou de module existant. Par exemple :

```
struct
  type t = int                t est équivalent à int
  let x = (1 : t)             x a le type t
  type t = bool               t est équivalent à int
  let y = (true : t)          y a le type t et x aussi
  ... if x = y then ...       x et y ont le même type t
end
```

La condition $t \notin \mathrm{Dom}(E)$ dans la règle (struct-type) et la condition $X \notin \mathrm{Dom}(E)$ dans (struct-mod) évitent ce problème, mais sont un peu trop restrictives. En particulier, on ne peut pas définir deux fois le même nom de type à des niveaux d'emboîtement de structures différents :

```
struct
  type t = int
  module X = struct type t = bool ... end
end
```

Pour traiter plus souplement les redéfinitions, on peut distinguer dans les structures les noms des composantes (qui servent à faire référence aux composantes dans les projections et ne peuvent donc pas être renommées) et les identificateurs liés par les définitions de composantes (qui servent à faire référence aux composantes dans le reste de la structure et peuvent être renommés par alpha-conversion). Ainsi, l'exemple ci-dessus devient (les identificateurs sont marqués en italiques) :

```
struct
  type t/t1 = int
  let x/x = (1 : t1)            x a le type t1
  type t/t2 = int
  let y/y = (true : t2)         y a le type t2
  ... if x = y then ...         x et y ont des types différents
end
```

Foncteurs et applications de foncteurs

$$\frac{X \notin \mathrm{Dom}(E) \qquad E;\, X : M \vdash m : M'}{E \vdash (\texttt{functor}\ (X : M) \to m) : (\texttt{functor}\ (X : M) \to M')}\ \text{(mod-foncteur)}$$

$$\frac{E \vdash m_1 : (\texttt{functor}\ (X : M) \to M') \qquad E \vdash m_2 : M}{E \vdash m_1(m_2) : M'[X \leftarrow m_2]}\ \text{(mod-appl)}$$

Dans la règle (mod-appl), le remplacement du paramètre formel X par l'argument effectif m_2 dans le type M' du résultat du foncteur assure la propagation des égalités de types à travers les applications de foncteurs. Exemple :

```
module Id : functor(X : sig type t end) → sig type t = X.t end
module E = struct type t = int end
module B = Id(E)
```

Le type de `B` est (`sig type t` $=$ `X.t end`)[`X` $\leftarrow$ `E`], c'est-à-dire `sig type t = E.t`. Il s'ensuit que `B.t` $\approx$ `E.t` par la règle (eq-projection), et comme d'autre part `E.t` $\approx$ `int` d'après la signature de `E`, on a bien `B.t` $\approx$ `int` comme attendu.

Dans le cas où le paramètre formel X n'apparaît pas dans le type M', on retrouve la règle d'application de fonction habituelle :

$$\frac{E \vdash m_1 : (\texttt{functor}\ (X : M) \to M') \qquad E \vdash m_2 : M \qquad X \notin \mathcal{L}(M')}{E \vdash m_1(m_2) : M'} \ \text{(mod-appl-nondep)}$$

Si l'argument m_2 du foncteur n'est pas un chemin d'accès et que X apparaît dans le type M' du résultat du foncteur, la substitution $M'[X \leftarrow m_2]$ n'est pas un type bien formé. La règle (mod-appl) n'est alors pas applicable directement. Exemple :

```
module B = Id(struct type t = int end)
```

Pour appliquer (mod-appl), il faudrait dans le type `sig type t` $=$ `X.t end` remplacer `X` par `struct type t = int end`, obtenant le type mal formé

```
sig type t = (struct type t = int).t end
```

Une solution simple à ce problème est de toujours nommer les arguments de foncteurs qui ne sont pas des chemins :

`module A = F(`m`)` $\quad\Rightarrow\quad$ `module B = `m` ; module A = F(B)`

Une solution plus subtile consiste à utiliser la règle de sous-typage (mod-sub) pour faire disparaître la dépendance entre argument et résultat dans le type du foncteur. Dans l'exemple `Id` ci-dessus, remarquons que

```
functor(X : sig type t end) → sig type t = X.t end <:
      functor(X : sig type t = int end) → sig type t = int end
```

Le super-type est un type non dépendant : le paramètre formel `X` n'apparaît plus dans le type du résultat. On peut donc construire la dérivation suivante :

$$\frac{\dfrac{E \vdash \texttt{Id} : \texttt{functor(X : sig type t end)} \to \texttt{sig type t} = \texttt{X.t end}}{E \vdash \texttt{Id} : \texttt{functor(X : sig type t} = \texttt{int end)} \to \texttt{sig type t} = \texttt{int end}} \qquad E \vdash (\texttt{struct type t} = \texttt{int end}) : (\texttt{sig type t} = \texttt{int end})}{E \vdash \texttt{Id(struct type t} = \texttt{int end)} : (\texttt{sig type t} = \texttt{int end})}$$

La règle de renforcement

Soit p un chemin désignant une structure avec un type t abstrait :

$$p : \texttt{sig}\ \ldots ; \texttt{type}\ t ; \ldots\ \texttt{end}$$

L'implémentation du type $p.t$ est inconnue. Donc, le type $p.t$ est a priori incompatible avec tout autre type. Cependant, la composante t de la structure désignée par p n'est pas, elle, n'importe quel type : nous savons qu'elle est forcément égale à $p.t$. Le chemin p a donc aussi le type suivant :

$$p : \texttt{sig } \ldots ; \texttt{type } t = p.t ; \ldots \texttt{ end}$$

Plus généralement, nous appelons renforcement cette opération de remplacement de types abstraits par des types manifestement égaux à eux-même dans la signature d'un chemin. Si M est un type de module et p un chemin, nous définissons le type renforcé M/p comme suit :

$$\begin{array}{rcl}
(\texttt{sig } d_1; \ldots ; d_n \texttt{ end})/p & = & \texttt{sig } d_1/p; \ldots ; d_n/p \texttt{ end} \\
(\texttt{val } v : \tau)/p & = & \texttt{val } v : \tau \\
(\texttt{type } t)/p & = & \texttt{type } t = p.t \\
(\texttt{type } t = \tau)/p & = & \texttt{type } t = \tau \\
(\texttt{module } X : M)/p & = & \texttt{module } X : (M/(p.X)) \\
(\texttt{functor}(X : M) \to M')/p & = & \texttt{functor}(X : M) \to M'
\end{array}$$

La règle de typage correspondant à l'opération de renforcement est alors :

$$\frac{E \vdash p : M}{E \vdash p : (M/p)} \ \text{(mod-renforcement)}$$

Le renforcement est nécessaire dans plusieurs situations. La première est lorsqu'on prend une vue restreinte d'un module contenant des types abstraits. Supposons par exemple

```
E : sig type t ; val x: t ; val y: t → t end
```

Nous voulons prendre une vue de `E` qui cache la composante `y`. Si nous faisons

```
module B = (E : sig type t ; val x: t end)
```

nous cachons `y`, mais rendons le type `B.t` abstrait, et donc différent de `E.t`. Pour conserver la compatibilité entre `B.t` et `E.t`, il faut écrire

```
module B = (E : sig type t = E.t ; val x: t end)
```

Pour typer cette contrainte, il faut montrer que le type de `E` est sous-type de `sig type t = E.t ; val x : t end`. Si l'on part de `E : sig type t`; ..., ce n'est pas possible. Il faut renforcer au préalable le type de `E`, obtenant `sig type t = E.t`; ..., pour montrer que la contrainte de signature est bien typée.

La seconde utilité de la règle de renforcement est pour détecter la propagation des types à travers les foncteurs. Reprenons le foncteur identité vu plus haut :

```
module Id = functor(X: sig type t end) → struct type t = X.t end
```

Sans règle de renforcement, le type le plus précis que l'on puisse lui attribuer est

```
Id : functor(X: sig type t end) → sig type t end
```

et ce type ne traduit pas que la composante `t` du résultat est égale à la composante `t` de l'argument. En appliquant la règle de renforcement au type de `X`, nous obtenons le type plus précis :

```
Id : functor(X: sig type t end) → sig type t = X.t end
```

Une troisième application du renforcement est pour vérifier des contraintes de partage. Un foncteur à plusieurs arguments peut avoir à imposer que des composantes de types de ses arguments soient égales afin que le corps du foncteur soit bien typé. C'est ce qu'on appelle une contrainte de partage entre les arguments d'un foncteur, et cette contrainte de partage s'exprime facilement avec des types manifestes en position d'argument de foncteur. Par exemple, voici un foncteur qui compose des opérations entre types abstraits :

```
module Compose =
  functor(X : sig type t ; val f: t → t end) →
  functor(Y : sig type t = X.t ; val f: t → t end) →
    struct
      type t = X.t
      let f = fun x → X.f(Y.f(x))
    end
```

Grâce à la spécification `Y : sig type t = X.t ; ...`, le corps du foncteur est typé sous l'hypothèse `X.t` = `Y.t`, et cela assure que la composition des fonctions `X.f` et `Y.f` est bien typée. Si on avait déclaré `Y : sig type t ; ...`, `X.t` et `Y.t` auraient été considérés comme incompatibles, et la définition de la fonction `f` comme mal typée.

La vérification des contraintes de partage lors de l'application du foncteur s'effectue automatiquement par les règles de typage de l'application et de sous-typage. Par exemple, supposons

```
module A : sig type t = τ1 ; val f: t → t end
module B : sig type t = τ2 ; val f: t → t end
```

Le typage de l'application `Compose(A)(B)` se déroule comme suit. Tout d'abord, on type `Compose(A)`, obtenant

```
Compose(A) :
  functor (Y : sig type t = A.t end ; val f : t → t end) →
    sig type t = A.t ;  val f : t → t end
```

Maintenant, l'application de `Compose(A)` à `B` est bien typée si et seulement si `B` a le type de l'argument de `Compose(A)`, c'est-à-dire ssi τ_2 est équivalent à `A.t`, ou de manière équivalente ssi τ_1 et τ_2 sont équivalents.

Dans le cas où `B.t` est abstrait mais `A.t` est prouvablement égal à `B.t`, la règle de renforcement est nécessaire pour vérifier la contrainte de partage. Supposons

```
module A : sig type t = B.t ; val f: t → t end
module B : sig type t ; val f: t → t end
```

Pour que `Compose(A)(B)` soit bien typé, il faut que `B` possède la signature `sig type t = B.t ; val f : t → t`, et cela nécessite d'appliquer la règle de renforcement au type de `B`.

Pour aller plus loin

Sémantique statique à stamps : la définition de Standard ML utilise une approche différente du typage des modules de ML, s'appuyant sur des *stamps* uniques pour représenter l'identité des types abstraits. Cette approche est présentée dans *A type discipline for program modules*, R. Harper, R. Milner, M. Tofte, actes TAPSOFT 87, LNCS 250, et étendue au langage Standard ML tout entier dans *The definition of Standard ML*, R. Milner, M. Tofte, R. Harper et D.MacQueen, MIT Press, 1997. Une présentation plus simple ainsi qu'une preuve d'équivalence entre la sémantique statique à stamps et les règles de typage présentées dans ce cours se trouve dans *A syntactic theory of* *type generativity and sharing*, X. Leroy, Journal of Functional Programming 6(5), 1996, `http ://pauillac.inria.fr/~xleroy/publi/syntactic-generativity.dvi.gz`.

Implémentation du typage des modules et adaptation à d'autres langages de base : le rapport technique suivant développe une implémentation simple d'un typeur pour le langage des modules qui est paramétrée par un typeur pour le langage de base : *A modular module system*, X. Leroy, rapport de recherche 2866, INRIA, 1996, `http ://pauillac.inria.fr/~xleroy/publi/modular-modules.ps.gz`.

Preuve de sûreté du typage : la thèse suivante étudie les propriétés formelles d'un calcul de modules proche de celui présenté dans ce chapitre, en particulier la sûreté du typage : *Translucent sums : a foundation for higher-order module systems*, M. Lillibridge, PhD thesis, Carnegie-Mellon University, 1997, `http ://www.research.digital.com/SRC/personal/Mark_Lillibridge/Papers/Thesis/thesis.PS`.

Exercices

Exercice 9.1 *(*)/(**) Montrer que la relation de sous-typage $<:$ est réflexive (*) et transitive (**).*

Exercice 9.2 *(*) On dit que deux signatures S_1 et S_2 sont équivalentes dans l'environnement A, et on note $A \vdash S_1 \approx S_2$, si $A \vdash S_1 <: S_2$ et $A \vdash S_2 <: S_1$. Donner une axiomatisation directe de $A \vdash S_1 \approx S_2$ sous forme de règles d'inférence.*

Exercice de programmation 9.1 *Implémenter la relation de sous-typage $<:$.*

Exercice 9.3 *(***) Montrer qu'on ne perd pas d'expressivité si on restreint les règles de projection (eq-projection), (val-projection) et (mod-projection) au cas où le type de la composante extraite ne dépend pas du début S_1^* de la signature :*

$$\frac{E \vdash p : \texttt{sig}\ S_1^*;\ \texttt{type}\ t = \tau;\ S_2^*\ \texttt{end} \qquad \mathcal{L}(\tau) \cap \mathcal{B}(S_1^*) = \emptyset}{E \vdash p.t \approx \tau} \text{ (eq-projection')}$$

$$\frac{E \vdash p : \texttt{sig}\ S_1^*;\ \texttt{val}\ x : \sigma;\ S_2^*\ \texttt{end} \qquad \mathcal{L}(\sigma) \cap \mathcal{B}(S_1^*) = \emptyset \qquad \tau \leq \sigma}{E \vdash p.x : \tau} \text{ (val-projection')}$$

$$\frac{A \vdash p : \texttt{sig}\ S_1^*;\ \texttt{module}\ X : M;\ S_2^*\ \texttt{end} \qquad \mathcal{L}(M) \cap \mathcal{B}(S_1^*) = \emptyset}{A \vdash p.X : M} \text{ (mod-projection')}$$

On a noté $\mathcal{L}(\tau)$ les identificateurs de types et de modules libres dans le type τ et $\mathcal{B}(S^*)$ les identificateurs de types et de modules liés par la signature S^*. (Indication : on peut toujours appliquer la règle de renforcement à p avant de typer la projection.)

Annexe A

Corrigés des exercices du chapitre 1

Exercice 1.1 Soit $\bullet$ une valeur quelconque, par exemple l'entier 0 ; elle nous servira de "terminateur". La construction d'un n-uplet $(a_1, a_2, \ldots, a_n)$ peut être codée par la construction de n paires imbriquées et terminées par $\bullet$, c'est-à-dire $(a_1, (a_2, \ldots (a_n, \bullet) \ldots))$. L'accès à la k-ième composante d'un n-uplet ainsi codé se fera donc par une série de projections successives : $\pi_k\ a$ sera codé par $\mathtt{snd}^{k-1}\ (\mathtt{fst}\ a)$.

L'emploi d'un terminateur permet de donner un sens au codage même lorsque n vaut 0 ou 1, et en simplifie la définition. On peut imaginer d'autres codages légèrement, voire sensiblement plus efficaces, mais l'efficacité n'est pas recherchée ici, l'intérêt étant simplement l'existence d'un codage.

Exercice 1.2 La construction ML `let rec` $f\ x = a_1$ `in` a_2 peut être vue comme du "sucre syntaxique" pour

$$\mathtt{let}\ f = \mathtt{fix}(\mathtt{fun}\ f \to \mathtt{fun}\ x \to a_1)\ \mathtt{in}\ a_2$$

Exercice 1.3 Le `let rec` multiple peut toujours se ramener à un `let rec` simple en paramétrisant l'une des définitions par-rapport à l'autre. Autrement dit, `let rec` $f\ x = a_1$ `and` $g\ y = a_2$ `in` a_3 peut se transformer en

```
let rec f g x = a1 in
let rec g y = let f = f g in a2 in
let f = f g in
a3
```

Ensuite, on encode ces deux `let rec` simples en termes de `fix` comme dans l'exercice 1.2.

Exercice 1.4 Considérons la fonction `fun` $x \to x+y$, qui à x associe $x+y$. Cette expression a une variable libre, à savoir y ; la variable x est liée dans cette expression. Si la restriction concernant l'α-conversion était levée, on pourrait renommer x en y, ce qui conduirait à la fonction `fun` $y \to y+y$. Il est intuitivement clair qu'il ne s'agit pas de la même fonction. En particulier, cette dernière fonction n'a aucune variable libre, donc ne fait plus référence à la variable y ; on pourrait aussi bien l'écrire `fun` $x \to x + x$. Ainsi, il est clair que lever cette restriction conduit à une définition qui n'a plus le sens souhaité.

Exercice 1.5 À rédiger...

Exercice 1.6 La substitution $(x+1)[x \leftarrow 2]$ est égale à $(2+1)$. L'unique occurrence de x dans le terme $(x+1)$ est libre, et est donc substituée par 2.

Pour calculer la substitution $(\texttt{fun}\ x \to x)[x \leftarrow 2]$, il faut appliquer une étape d'α-conversion, de façon à ce que la variable liée ne soit pas celle que l'on substitue. On peut donc d'abord écrire $(\texttt{fun}\ y \to y)[x \leftarrow 2]$, où y est une variable fraîche, et on constate alors que le résultat est $(\texttt{fun}\ y \to y)$, que l'on peut également écrire, si on le souhaite, $(\texttt{fun}\ x \to x)$. En fait, le résultat était prévisible de façon plus directe : il est conséquence de l'énoncé établi par l'exercice 1.7.

Pour calculer la substitution $(\texttt{fun}\ x \to x+y)[y \leftarrow x]$, il faut appliquer une étape d'α-conversion, car la variable liée x apparaît libre dans l'expression que l'on souhaite substituer à y. On peut donc d'abord écrire $(\texttt{fun}\ z \to z+y)[y \leftarrow x]$, où z est une variable fraîche, et on constate alors que le résultat est $(\texttt{fun}\ z \to z+x)$. Notons que ce résultat semble dépendre du choix de z. C'est vrai, mais si on varie z, tous les termes obtenus sont α-convertibles. La substitution est donc bien définie, à α-conversion près.

Exercice 1.7 À rédiger...

Exercice 1.8 On montre facilement que si deux prédicats P et Q satisfont les règles, alors leur conjonction $P \wedge Q$ les satisfait aussi. En effet, si P et Q satisfont les axiomes, alors $P(a_i(x))$ est vrai pour tout x, ainsi que $Q(a_i(x))$, et donc $(P \wedge Q)(a_i(x))$ est vrai. Pour ce qui est des règles, supposons que $(P \wedge Q)(b_j^k(x))$ est vrai pour tout $k = 1 \dots n$. Alors, $P(b_j^k(x))$ est vrai pour tout k, et donc $P(c_j(x))$ est vrai puisque P satisfait la règle j. De même, $Q(b_j^k(x))$ est vrai pour tout k, et donc $Q(c_j(x))$ est vrai puisque Q satisfait la règle j. Il s'ensuit que l'implication

$$(P \wedge Q)(b_j^1(x)) \wedge \dots \wedge (P \wedge Q)(b_j^n(x)) \Rightarrow (P \wedge Q)(c_j(x))$$

est vraie, et donc $P \wedge Q$ satisfait la règle j.

Le résultat précédent s'étend trivialement à des conjonctions sur un nombre arbitraire de prédicats satisfaisant les règles : si on a une famille de prédicats $(P_a)_{a \in A}$ qui satisfont les règles, alors leur conjonction $\bigwedge_{a \in A} P_a$ les satisfait aussi.

On considère alors $P_{min} = \bigwedge\{P \mid P \text{ satisfait les règles}\}$ (c'est-à-dire, la conjonction de tous les prédicats satisfaisant les règles). Par le résultat précédent, P_{min} satisfait les règles, et par construction il est plus petit que tout autre prédicat P satisfaisant les règles.

Exercice 1.9 On note $D(x)$ le prédicat "il existe une dérivation de l'énoncé $P(x)$ dans le système de règles". La preuve que D est le plus petit prédicat satisfaisant les règles est en deux temps : (1) on montre que D satisfait les règles, et (2) on montre que pour tout prédicat Q satisfaisant les règles, $D(x)$ implique $Q(x)$.

Pour (1), il est vrai que $\forall x.\, D(a_i(x))$, puisque pour tout x donné, $P(a_i(x))$ est une dérivation valide réduite à une feuille. De même, si pour un certain x, nous avons $D(b_j^1(x)) \wedge \dots \wedge D(b_j^n(x))$, cela signifie que nous avons des dérivations de $P(b_j^1(x)) \dots P(b_j^n(x))$; on peut donc construire une dérivation de noeud racine $P(c_j(x))$ et de fils les dérivations de $P(b_j^1(x)) \dots P(b_j^n(x))$, et c'est une dérivation valide. Donc, $D((c_j(x))$ est vrai. Il s'ensuit que D satisfait les règles.

Pour (2), on se donne un prédicat Q satisfaisant les règles, et on montre par récurrence structurelle sur la dérivation que pour tout x, si l'on a une dérivation de l'énoncé $P(x)$, alors $Q(x)$ est vrai. Il s'ensuit $D(x)$ implique $Q(x)$ pour tout x, et donc D est plus petit que Q. Prenant $Q = P_{min}$, par minimalité de P_{min}, il s'ensuit $D = P_{min}$ comme annoncé.

Exercice 1.10 Pour $a = 1\ 2$, une dérivation de $a \twoheadrightarrow v$ devrait se terminer par une des règles 4, 8 ou 6. Mais pour que ces règles s'appliquent, il faudrait que 1 s'évalue en une fonction (pour la règle 4) ou en une opération (pour les autres règles). Cependant, la seule valeur en laquelle 1 peut s'évaluer est 1, qui n'est rien de tout cela. Donc, il n'y a pas de dérivation de $a \twoheadrightarrow v$ pour tout v.

Pour $a' = (\texttt{fun}\ f \to f\ f)\ (\texttt{fun}\ f \to f\ f)$, notons $b = (\texttt{fun}\ f \to f\ f)$. Une dérivation de $a' \twoheadrightarrow v$ doit nécessairement se terminer ainsi :

$$\frac{b \twoheadrightarrow b \qquad b \twoheadrightarrow b \qquad (f\ f)[f \leftarrow b] \twoheadrightarrow v}{b\ b \twoheadrightarrow v}$$

Mais $(f\ f)[f \leftarrow b] = b\ b = a'$, donc toute dérivation de $a' \twoheadrightarrow v$ doit contenir une sous-dérivation de $a' \twoheadrightarrow v$; il n'existe bien sûr pas de dérivation finie satisfaisant cette propriété.

La différence entre a et a' est que a est un terme essentiellement mal formé, alors que a' est un terme bien formé mais dont l'évaluation ne termine pas.

Exercice 1.11 **Cas** $a = \texttt{let}\ x = a_1\ \texttt{in}\ a_2$. La seule règle qui s'applique est 7, et donc D est de la forme

$$\frac{\begin{matrix}(D_1) \\ \vdots \\ a_1 \twoheadrightarrow v_1\end{matrix} \qquad \begin{matrix}(D_2) \\ \vdots \\ a_2[x \leftarrow v_1] \twoheadrightarrow v_2\end{matrix}}{a_1\ a_2 \twoheadrightarrow v_2}$$

Vu la forme de a, D' se termine nécessairement par la règle 7 elle aussi. Donc, D' contient des sous-dérivations $D'_1 : a_1 \twoheadrightarrow v'_1$ et $D'_2 : a_2[x \leftarrow v'_1] \twoheadrightarrow v'_2$ pour certaines valeurs v'_1 et v'_2. 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 $v_1 = v'_1$. Par conséquent, $a_2[x \leftarrow v_1] = a_2[x \leftarrow v'_1]$ et nous pouvons appliquer l'hypothèse de récurrence à D_2 et D'_2. Il vient $v_2 = v'_2$, ce qui entraîne le résultat attendu $v = v'$.

Exercice 1.12 D'après la règle d'évaluation (7), l'assertion $\texttt{let}\ x = a_1\ \texttt{in}\ a_2 \twoheadrightarrow v$ (*) est équivalente à $\exists v_1.(a_1 \twoheadrightarrow v_1 \wedge a_2[x \leftarrow v_1] \twoheadrightarrow v)$. De plus, d'après la règle (3), l'assertion $\texttt{fun}\ x \to a_2 \twoheadrightarrow \texttt{fun}\ x \to a_2$ est vraie. L'assertion (*) est donc équivalente à $\exists v_1.(\texttt{fun}\ x \to a_2 \twoheadrightarrow \texttt{fun}\ x \to a_2 \wedge a_1 \twoheadrightarrow v_1 \wedge a_2[x \leftarrow v_1] \twoheadrightarrow v)$ qui, d'après la règle (4), est elle-même équivalente à $(\texttt{fun}\ x \to a_2)\ a_1 \twoheadrightarrow v$.

D'après la règle de typage (let), l'assertion $\Gamma \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2$ est équivalente à $\exists \tau_1.(\Gamma \vdash a_1 : \tau_1 \wedge \Gamma; x : \tau_1 \vdash a_2 : \tau_2)$. D'après la règle (fun), on peut écrire cela $\exists \tau_1.(\Gamma \vdash a_1 : \tau_1 \wedge \Gamma \vdash \texttt{fun}\ x \to a_2 : \tau_1 \to \tau_2)$. Enfin, d'après la règle (app), ceci est équivalent à $\Gamma \vdash (\texttt{fun}\ x \to a_2)\ a_1 : \tau_2$.

Exercice 1.13 Pour typer 1 2, il faudrait pouvoir attribuer à 1 un type flèche $\tau_1 \to \tau_2$, ce qui est bien sûr impossible car 1 a le type `int` dans tous les environnements de typage.

Pour typer $\texttt{fun}\ f \to f\ f$, il faudrait construire une dérivation de la forme suivante :

$$\frac{\dfrac{\Gamma; f : \tau_1 \vdash f : \tau_1 \to \tau_2 \qquad \Gamma; f : \tau_1 \vdash f : \tau_2}{\Gamma; f : \tau_1 \vdash f\ f : \tau_2}}{\Gamma \vdash \texttt{fun}\ f \to f\ f : \tau_1 \to \tau_2}$$

Pour que les feuilles de la dérivation soient justifiées par l'axiome (var), il faudrait que $\tau_1 = \tau_1 \to \tau_2$ et $\tau_1 = \tau_2$. La première de ces égalités est impossible, car τ_1 serait alors un sous-terme strict de lui-même, ce qui est impossible pour tout terme τ_1 fini.

Enfin, dans le cas de $\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$, nous pouvons attribuer à $\texttt{fun}\ x \to x$ le type $\tau \to \tau$ pour n'importe quel τ. Mais pour que $f\ 1$ soit bien typé, il faudrait prendre $\tau = \texttt{int}$, et pour que $f\ \texttt{true}$ soit bien typé, il faudrait prendre $\tau = \texttt{bool}$, et il est impossible de satisfaire ces deux contraintes simultanément.

Exercice 1.14 Notons φ la substitution $[\alpha_1 \leftarrow \beta_1, \dots, \alpha_n \leftarrow \beta_n]$. On montre $\mathcal{L}(\varphi(\tau)) = \varphi(\mathcal{L}(\tau))$ par récurrence structurelle sur τ. Le cas de base est τ est une variable γ. Si $\gamma = \alpha_i$ pour un certain i, on a

$$\mathcal{L}(\varphi(\alpha_i)) = \mathcal{L}(\beta_i) = \{\beta_i\} = \varphi(\{\alpha_i\}) = \varphi(\mathcal{L}(\alpha_i)).$$

Si $\gamma \notin \{\alpha_1, \dots, \alpha_n\}$, on a

$$\mathcal{L}(\varphi(\gamma)) = \mathcal{L}(\gamma) = \{\gamma\} = \varphi(\{\gamma\}) = \varphi(\mathcal{L}(\gamma)).$$

Les cas inductifs sont immédiats par application de l'hypothèse de récurrence.

Soient maintenant deux schémas $\sigma = \forall \alpha_1, \dots, \alpha_n.\, \tau$ et $\sigma' = \forall \beta_1, \dots, \beta_n.\, \tau'$ égaux à α-conversion près. On a donc $\tau' = \tau[\alpha_1 \leftarrow \beta_1, \dots, \alpha_n \leftarrow \beta_n]$, et de plus $\beta_i \notin \mathcal{L}(\sigma)$ pour tout i. Par ce qui précède,

$$\mathcal{L}(\tau') = \mathcal{L}(\tau)[\alpha_1 \leftarrow \beta_1, \dots, \alpha_n \leftarrow \beta_n]$$

Écrivons $\mathcal{L}(\tau) = \{\alpha_{i_1}, \dots, \alpha_{i_p}, \gamma_1, \dots, \gamma_q\}$ où les α_{i_j} sont les α_i libres dans τ, et les γ_j les variables libres dans τ qui ne sont pas les α_i. Calculons maintenant les variables libres de σ et σ'.

$$\begin{array}{rcl}
\mathcal{L}(\sigma) & = & \mathcal{L}(\tau) \setminus \{\alpha_1, \dots, \alpha_n\} \\
& = & \{\gamma_1, \dots, \gamma_q\} \\
\mathcal{L}(\sigma') & = & \mathcal{L}(\tau') \setminus \{\beta_1, \dots, \beta_n\} \\
& = & (\mathcal{L}(\tau)[\alpha_1 \leftarrow \beta_1, \dots, \alpha_n \leftarrow \beta_n]) \setminus \{\beta_1, \dots, \beta_n\} \\
& = & (\{\alpha_{i_1}, \dots, \alpha_{i_p}, \gamma_1, \dots, \gamma_q\}[\alpha_1 \leftarrow \beta_1, \dots, \alpha_n \leftarrow \beta_n]) \setminus \{\beta_1, \dots, \beta_n\} \\
& = & \{\beta_{i_1}, \dots, \beta_{i_p}, \gamma_1, \dots, \gamma_q\} \setminus \{\beta_1, \dots, \beta_n\} \\
& = & \{\gamma_1, \dots, \gamma_q\}
\end{array}$$

Pour la dernière égalité, on s'appuie sur le fait que $\{\gamma_1 \dots \gamma_q\} \cap \{\beta_1 \dots \beta_n\} = \emptyset$ car sinon l'une des β_i serait libre dans σ. D'où $\mathcal{L}(\sigma) = \mathcal{L}(\sigma')$. CQFD.

Exercice 1.15 Pour le terme $\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ f\ f$, on donne à f le schéma $\forall \alpha.\, \alpha \to \alpha$, puis on donne à la première occurrence de f un type $(\tau \to \tau) \to (\tau \to \tau)$, et à la seconde occurrence $\tau \to \tau$. Le terme entier a donc le type $\tau \to \tau$ pour τ arbitraire.

Pour $\texttt{fun}\ f \to f\ f$, ce terme n'est toujours pas typable. En effet, une dérivation de typage de ce terme devrait se terminer par :

$$\dfrac{\dfrac{\dfrac{\tau_1 \leq \tau \to \tau_2}{f : \tau_1 \vdash f : \tau \to \tau_2} \qquad \dfrac{\tau_1 \leq \tau}{f : \tau_1 \vdash f : \tau}}{f : \tau_1 \vdash f\ f : \tau_2}}{\emptyset \vdash \texttt{fun}\ f \to f\ f : \tau_1 \to \tau_2}$$

pour des types τ, τ_1, τ_2 bien choisis. Cependant, $\tau_1 \leq \tau \rightarrow \tau_2$ implique $\tau \rightarrow \tau_2 = \tau_1$, et de même $\tau_1 \leq \tau$ implique $\tau = \tau_1$. Donc, il faudrait que $\tau \rightarrow \tau_2 = \tau$, et ceci est impossible pour tout τ (un terme fini ne peut pas se contenir comme sous-terme).

Exercice 1.16 Considérons $a = \texttt{fun}\ x \rightarrow \texttt{let}\ y = x\ \texttt{in}\ y$. Sous l'hypothèse $x : \alpha$, avec la définition plus simple de *Gen*, on obtiendrait $y : \forall\alpha.\,\alpha$, et donc l'occurrence de y après le `in` pourrait recevoir le type β. a aurait donc le type $\alpha \rightarrow \beta$ où α et β sont deux variables de types distinctes. Ce n'est bien sûr pas un type correct pour la fonction identité.

Exercice 1.17 Commençons par définir l'image $\varphi(\sigma)$ d'un schéma $\sigma = \forall\alpha_1 \ldots \alpha_n.\ \tau$ par une substitution φ. Ensuite, $\varphi(\Gamma)$ sera simplement l'application point à point de φ sur les schémas contenus dans Γ, c'est-à-dire $\varphi(\Gamma) = \Gamma'$ est tel que $\Gamma'(x) = \varphi(\Gamma(x))$ pour tout $x \in \mathrm{Dom}(\Gamma)$. Naïvement, on prendrait

$$\varphi(\forall\alpha_1 \ldots \alpha_n.\ \tau) = \forall\alpha_1 \ldots \alpha_n.\ \varphi(\tau)$$

mais cela pose des problèmes de capture de variables liées par le quantificateur $\forall$. Par exemple, si $\varphi = [\alpha \leftarrow \beta]$, cela donnerait :

$$\varphi(\forall\alpha.\alpha) = \forall\alpha.\beta \qquad \varphi(\forall\beta.\alpha) = \forall\beta.\beta$$

et on voit que ces deux résultats sont incorrects en se rappelant que les variables liées α et β dans les deux schémas de types peuvent être renommées en toute autre variable γ sans changer la signification du schéma de types. Or,

$$\varphi(\forall\gamma.\gamma) = \forall\gamma.\gamma \qquad \varphi(\forall\gamma.\alpha) = \forall\gamma.\beta.$$

Autrement dit, la définition naïve ci-dessus ne passe pas au quotient par α-conversion (renommage des variables liées) : suivant les renommages que l'on effectue sur les variables liées du schéma argument, on obtient des schémas résultats différents, même à α-conversion près.

L'idée est d'exiger un renommage approprié du schéma afin d'éviter toute interférence entre les variables liées et la substitution. On pose donc :

$$\varphi(\forall\alpha_1 \ldots \alpha_n.\ \tau) = \forall\alpha_1 \ldots \alpha_n.\ \varphi(\tau) \text{ si } \alpha_1, \ldots, \alpha_n \text{ sont hors de portée de } \varphi.$$

Dans cette définition, on dit qu'une variable α est hors de portée d'une substitution φ si

1. $\varphi(\alpha) = \alpha$ (c'est-à-dire, φ ne modifie pas α)
2. si α n'est pas libre dans τ, alors α n'est pas libre dans $\varphi(\tau)$ (c'est-à-dire, φ n'introduit pas α dans son résultat).

Par exemple, prenant $\varphi = [\alpha \leftarrow \beta]$, on voit que α n'est pas hors de portée de φ (car $\varphi(\alpha) = \beta$, donc la condition 1 n'est pas vraie), et β n'est pas hors de portée de φ non plus (car β n'est pas libre dans α, mais β est libre dans $\varphi(\alpha) = \beta$, donc la condition 2 n'est pas vraie). En revanche, γ est hors de portée de φ, car $\varphi(\gamma) = \gamma$, et de plus si γ n'est pas libre dans τ, alors τ ne contient que des variables $\alpha, \beta, \delta, \ldots$, que φ transforme en $\beta, \beta, \delta, \ldots$ respectivement, donc γ n'est pas non plus libre dans $\varphi(\gamma)$.

Notez que dans la définition de $\varphi(\forall\alpha_1 \ldots \alpha_n.\ \tau)$, la condition "$\alpha_1, \ldots, \alpha_n$ sont hors de portée de φ" peut toujours être satisfaite en renommant préalablement les $\alpha_1 \ldots \alpha_n$. (Il y a un nombre infini de variables hors de portée d'une substitution donnée.) On a donc défini $\varphi(\sigma)$ pour tout schéma σ.

Passons à la preuve de la proposition 1.2. La preuve est par récurrence sur la dérivation de typage de $\Gamma \vdash a : \tau$ et par cas sur la dernière règle de typage utilisée.

Cas règle (var-inst). Nous avons $\Gamma \vdash x : \tau$ avec $\Gamma(x) \leq \tau$. Écrivons $\Gamma(x) = \forall \alpha_1 \ldots \alpha_n.\tau_x$. Nous avons donc $\tau = \tau_x[\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n]$. Quitte à renommer les α_i, nous pouvons de plus supposer les α_i hors de portée de φ. Donc :

$$(\varphi(\Gamma))(x) = \varphi(\Gamma(x)) = \forall \alpha_1 \ldots \alpha_n.\varphi(\tau_x)$$

d'une part, et d'autre part

$$\varphi(\tau) = \varphi(\tau_x[\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n]) = \varphi(\tau_x)[\alpha_1 \leftarrow \varphi(\tau_1), \ldots, \alpha_n \leftarrow \varphi(\tau_n)].$$

Ceci montre que $(\varphi(\Gamma))(x) \leq \varphi(\tau)$. Par conséquent, la règle (var-inst) nous permet de conclure que $\varphi(\Gamma) \vdash x : \varphi(\tau)$, ce qui est le résultat désiré.

Cas règle (const-inst) ou (op-inst). Comme les schémas $TC(c)$ et $TC(op)$ sont clos (sans variables libres) pour tous c et op, on a $\varphi(TC(c)) = TC(c)$ et de même $\varphi(TC(op)) = TC(op)$. On conclut alors par le même raisonnement que pour la règle (var-inst).

Cas règle (fun). Nous avons $\Gamma \vdash (\texttt{fun}\ x \to a) : \tau_1 \to \tau_2$ en conséquence de la prémisse $\Gamma; x : \tau_1 \vdash a : \tau_2$. Appliquant l'hypothèse de récurrence à cette prémisse, nous obtenons une dérivation de $\varphi(\Gamma; x : \tau_1) \vdash a : \varphi(\tau_2)$, c'est-à-dire $\varphi(\Gamma); x : \varphi(\tau_1) \vdash a : \varphi(\tau_2)$. Par application de la règle (fun), nous concluons $\varphi(\Gamma) \vdash (\texttt{fun}\ x \to a) : \varphi(\tau_1) \to \varphi(\tau_2)$, ce qui est le résultat désiré.

Cas règle (app), (paire), (proj). Même raisonnement que pour la règle (fun).

Cas règle (let). Nous avons $\Gamma \vdash (\texttt{let}\ x = a_1\ \texttt{in}\ a_2) : \tau_2$ en conséquence des prémisses $\Gamma \vdash a_1 : \tau_1$ et $\Gamma; x : Gen(\tau_1, \Gamma) \vdash a_2 : \tau_2$. Il faut être prudent, car les variables généralisées par Gen, c'est-à-dire $\{\alpha_1, \ldots, \alpha_n\} = \mathcal{L}(\tau_1) \setminus \mathcal{L}(\Gamma)$, ne sont pas forcément hors de portée de φ, et on n'a donc pas, en général,

$$\varphi(Gen(\tau_1, \Gamma)) = Gen(\varphi(\tau_1), \varphi(\Gamma)).$$

Par exemple, on peut avoir $y : \alpha \vdash a_1 : \beta \to \beta$, et β est généralisable, mais après application de la substitution $[\beta \leftarrow \alpha]$, on obtient $y : \alpha \vdash a_1 : \alpha \to \alpha$, dans lequel α n'est pas généralisable ! Nous allons donc d'abord montrer que nous pouvons nous ramener au cas où $\alpha_1, \ldots, \alpha_n$ sont hors de portée de φ.

Donnons-nous pour cela des variables distinctes $\beta_1, \ldots, \beta_n$ non libres dans Γ et hors de portée de φ, et considérons la substitution

$$\psi = [\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n]$$

Appliquant l'hypothèse de récurrence à la première prémisse, nous obtenons $\psi(\Gamma) \vdash a_1 : \psi(\tau_1)$, c'est-à-dire, puisqu'aucun des α_i n'est libre dans Γ, $\Gamma \vdash a_1 : \psi(\tau_1)$. On observe alors que les schémas $Gen(\psi(\tau_1), \Gamma)$ et $Gen(\tau_1, \Gamma)$ sont identiques à α-conversion près, puisque le premier est $\forall \beta_1, \ldots, \beta_n.\tau_1[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n]$, tandis que le second est $\forall \alpha_1, \ldots, \alpha_n.\tau_1$, et aucun β_i n'est libre dans $\forall \alpha_1, \ldots, \alpha_n.\tau_1 = Gen(\tau_1, \Gamma)$, sans quoi il serait libre dans Γ. Le jugement de typage qui nous intéresse, à savoir $\Gamma \vdash (\texttt{let}\ x = a_1\ \texttt{in}\ a_2) : \tau_2$, peut donc également être obtenu à partir des prémisses $\Gamma \vdash a_1 : \psi(\tau_1)$ et $\Gamma; x : Gen(\tau_1, \Gamma) \vdash a_2 : \tau_2$. Nous sommes donc ramenés à un problème identique au problème initial, à la différence près que les variables généralisables dans la première prémisse, à savoir $\beta_1, \ldots, \beta_n$, sont hors de portée de φ.

Nous avons donc démontré qu'on peut supposer, *sans perte de généralité*, que les variables $\alpha_1, \ldots, \alpha_n$ sont hors de portée de φ, et c'est ce que nous allons faire dans la suite.

Par application de l'hypothèse de récurrence aux deux prémisses, on obtient $\varphi(\Gamma) \vdash a_1 : \varphi(\tau_1)$ et $\varphi(\Gamma); x : \varphi(Gen(\tau_1, \Gamma)) \vdash a_2 : \varphi(\tau_2)$. Pour conclure le résultat attendu par application de la règle (let-gen), il reste donc à montrer que

$$Gen(\varphi(\tau_1), \varphi(\Gamma)) = \varphi(Gen(\tau_1, \Gamma)).$$

Les variables libres dans $\varphi(\tau_1)$ mais pas dans $\varphi(\Gamma)$ sont exactement $\{\alpha_1, \ldots, \alpha_n\}$. En effet, les variables libres dans τ_1 sont

$$\alpha_1, \ldots, \alpha_n, \text{et des variables libres dans } \Gamma$$

Lorsqu'on applique φ, ces variables libres deviennent

$$\alpha_1, \ldots, \alpha_n, \text{et des variables libres dans } \varphi(\Gamma)$$

Aucun α_i n'est libre dans $\varphi(\Gamma)$ car, étant donné que les α_i sont hors de portée de φ, l'inverse impliquerait qu'un α_i est libre dans Γ, ce qui n'est pas le cas, par définition de $\{\alpha_1, \ldots, \alpha_n\}$. Par conséquent,

$$Gen(\varphi(\tau_1), \varphi(\Gamma)) = \forall \alpha_1, \ldots, \alpha_n. \varphi(\tau_1)$$

De plus, les α_i sont hors de portée de φ, donc :

$$\varphi(Gen(\tau_1, \Gamma)) = \forall \alpha_1, \ldots, \alpha_n. \varphi(\tau_1)$$

D'où le résultat.

Annexe B

Corrigés des exercices du chapitre 2

Exercice 2.1 L'évaluation de droite à gauche s'obtient en définissant les contextes d'évaluation par

Contextes d'évaluation (droite-gauche) :

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

L'évaluation sans ordre imposé d'évaluation, mais toujours en appel par valeur, s'obtient par :

Contextes d'évaluation (sans ordre imposé) :

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

On notera qu'alors la sémantique du langage n'est plus déterministe, au sens strict, mais il est facile de constater qu'elle reste confluente.

Exercice 2.2 On montre d'abord l'implication $(a \twoheadrightarrow v) \Rightarrow (a \rightarrow^\star v)$ par récurrence sur la dérivation de l'évaluation $a \twoheadrightarrow v$ et par cas sur la dernière règle utilisée. Si c'est un des axiomes 1, 2, 3, le résultat est immédiat car $v = a$. Si c'est la règle 4, on a :

$$\frac{a \twoheadrightarrow (\texttt{fun}\ x \rightarrow c) \qquad b \twoheadrightarrow v_2 \qquad c[x \leftarrow v_2] \twoheadrightarrow v}{a\ b \twoheadrightarrow v}$$

Par hypothèse de récurrence appliquée aux trois sous-dérivations des prémisses, nous savons qu'il existe des séquences de réductions de la forme

$$
\begin{array}{l}
a \to a_1 \to \ldots \to a_n \to v_1 = (\texttt{fun}\ x \to c) \\
b \to b_1 \to \ldots \to b_p \to v_2 \\
c[x \leftarrow v_2] \to c_1 \to \ldots \to c_q \to v
\end{array}
$$

En appliquant la règle (contexte) à chaque étape des deux premières séquences (avec les contextes $[\,]\ b$ et $v_1\ [\,]$), nous obtenons

$$
\begin{array}{l}
a\ b \to a_1\ b \to \ldots \to a_n\ b \to v_1\ b \\
v_1\ b \to v_1\ b_1 \to \ldots \to v_1\ b_p \to v_1\ v_2
\end{array}
$$

En enchaînant toutes ces réductions et en glissant au milieu une étape (β), il vient

$$a\ b \to \ldots \to v_1\ b \to \ldots \to v_1\ v_2 = (\texttt{fun}\ x \to c)\ v_2 \to c[x \leftarrow v_2] \to \ldots \to v$$

D'où le résultat attendu $a\ b \to^\star v$. Le raisonnement est le même pour les autres règles d'évaluation (5, 7, 8, 6).

Montrons maintenant l'implication inverse $(a \to^\star v) \Rightarrow (a \twoheadrightarrow v)$. Il suffit de montrer les deux lemmes suivants :

1. Pour toute valeur v, $v \twoheadrightarrow v$.
2. Si $a \to a'$ et $a' \twoheadrightarrow v$, alors $a \twoheadrightarrow v$.

En effet, si l'on a $a \to a_1 \to \ldots a_n \to v$, il s'ensuit que $v \twoheadrightarrow v$ par (1), puis $a_n \twoheadrightarrow v$ par (2), puis $a_{n-1} \twoheadrightarrow v$ par (2), et ainsi de suite jusqu'à $a \twoheadrightarrow v$ par applications répétées de (2).

Le lemme (1) est immédiat par récurrence structurelle sur v.

Pour (2), on montre d'abord le résultat pour une réduction de tête $a \xrightarrow{\varepsilon} a'$, en examinant les axiomes de réduction. Prenons comme exemple (β) : on a $a = (\texttt{fun}\ x \to a_1)\ v_2$ et $a' = a_1[x \leftarrow v_2]$. On peut construire la dérivation suivante :

$$\frac{(\texttt{fun}\ x \to a_1) \twoheadrightarrow (\texttt{fun}\ x \to a_1) \qquad v_2 \twoheadrightarrow v_2 \qquad a_1[x \leftarrow v_2] \twoheadrightarrow v}{(\texttt{fun}\ x \to a_1)\ v_2 \twoheadrightarrow v}$$

(La prémisse de gauche est l'axiome 3 ; celle du milieu vient du lemme (2) ; celle de droite de l'hypothèse $a_1[x \leftarrow v_2] \twoheadrightarrow v$.) On a donc bien montré $a \twoheadrightarrow v$. La preuve pour les autres axiomes de réduction en tête est similaire.

Pour finir la preuve de (2), il faut montrer que tout cela passe bien au contexte : si $a \xrightarrow{\varepsilon} a'$ et $E[a'] \twoheadrightarrow v$, alors $E[a] \twoheadrightarrow v$. Cela se fait par récurrence structurelle sur E. Le cas de base $E = [\,]$ est le résultat précédent sur les réductions de tête. Faisons le cas $E = E'\ a_2$ par exemple. Supposons donc $E[a'] \twoheadrightarrow v$. Comme $E[a'] = E'[a']\ a_2$, nous avons donc une dérivation de la forme

$$\frac{E'[a'] \twoheadrightarrow \texttt{fun}\ x \to a_3 \qquad a_2 \twoheadrightarrow v_2 \qquad a_3[x \leftarrow v_2] \twoheadrightarrow v}{E'[a']\ a_2 \twoheadrightarrow v}$$

Par hypothèse de récurrence, on a $E'[a] \twoheadrightarrow \texttt{fun}\ x \to a_3$. On peut donc construire la dérivation

$$\frac{E'[a] \twoheadrightarrow \texttt{fun}\ x \to a_3 \qquad a_2 \twoheadrightarrow v_2 \qquad a_3[x \leftarrow v_2] \twoheadrightarrow v}{E'[a]\ a_2 \twoheadrightarrow v}$$

qui conclut $E[a] \twoheadrightarrow v$ comme attendu. La preuve pour les autres formes de contextes est similaire.

Exercice 2.3 H0 est trivialement vérifiée. Pour H1, on regarde chaque δ-règle $a \xrightarrow{\varepsilon} a'$ et à partir d'une dérivation de typage de $\Gamma \vdash a : \tau$ (à gauche), on construit une dérivation de typage de $\Gamma \vdash a' : \tau$ (à droite) :

$$\dfrac{\Gamma \vdash + : \mathtt{int} \times \mathtt{int} \to \mathtt{int} \qquad \dfrac{\Gamma \vdash n_1 : \mathtt{int} \qquad \Gamma \vdash n_2 : \mathtt{int}}{\Gamma \vdash (n_1, n_2) : \mathtt{int}}}{\Gamma \vdash +(n_1, n_2) : \mathtt{int}} \qquad \Gamma \vdash n_1 + n_2 : \mathtt{int}$$

$$\dfrac{\Gamma \vdash \mathtt{fix} : (\tau \to \tau) \to \tau \qquad \dfrac{\Gamma; x : \tau \vdash a : \tau}{\Gamma \vdash \mathtt{fun}\ x \to a : \tau \to \tau}}{\Gamma \vdash \mathtt{fix}\ (\mathtt{fun}\ x \to a) : \tau} \qquad \Gamma \vdash a[x \leftarrow \mathtt{fix}(\mathtt{fun}\ x \to a)] : \tau$$

Dans le dernier cas, règle (δ_{fix}), on a utilisé le lemme de substitution 2.2.

Pour H2, on considère les applications $op\ v$ qui sont bien typées, et on montre que v est forcément de la forme exigée par la δ-règle. Par exemple, si $+\ v$ est bien typée, v est une valeur de type $\mathtt{int} \times \mathtt{int}$. Par examen des règles de typage et d'après H0, v est forcément une paire d'entiers. De même, dans le cas de $\mathtt{fix}\ v$, nous savons que v est une valeur ayant un type flèche $\tau \to \tau$. Ainsi, v est ou bien une fonction $\mathtt{fun}\ x \to a$, soit un opérateur op. Mais le cas $v = op$ est impossible, car aucun des opérateurs qui nous intéressent n'a un type de la forme $\tau \to \tau$. (Notons au passage qu'un tel argument est un peu "fragile" : c'est un peu un hasard si aucun opérateur n'a un type de cette forme... quelle remarque peut-on en tirer ?) Donc v est une fonction, et la règle (δ_{fix}) s'applique.

Annexe C

Corrigés des exercices du chapitre 5

Exercice 5.1 La fonction `f` calcule la fonction factorielle. En effet, une fois que l'on a évalué l'affectation

```
r := (fun x → if x = 0 then 1 else x * ( !r)(x-1))
```

la fonction $g =$ `!r` est égale à `fun x → if x = 0 then 1 else x * ( !r)(x-1)`, c'est-à-dire `fun x → if x = 0 then 1 else x *` g`(x-1)`.

La fonction `fun x → 0` n'est jamais appelée, car elle n'est accessible initialement qu'à travers la référence `r`, mais dès la seconde ligne, le contenu de `r` est remplacé par une autre fonction, et elle devient donc inaccessible. Elle joue simplement le rôle de valeur initiale (arbitraire, mais de type `int` → `int`) pour la référence. Il est en effet impossible de créer une référence sans fournir sa valeur initiale.

Exercice 5.2 Voici l'opérateur de point fixe pour les fonctions des entiers dans les entiers :

```
let fix = fun f →
            let r = ref (fun x → 0) in
            r := (fun x → f ( !r) x) ;
           !r
```

On peut le généraliser pour qu'il opère sur toutes les fonctions à condition de se donner une expression ω de type $\forall\alpha.\alpha$ et qui bien sûr ne termine pas (p.ex. une boucle infinie ou encore une levée d'exception `raise` E) :

```
let fix = fun f →
            let r = ref (fun x → ω) in
            r := (fun x → f ( !r) x) ;
           !r
```

Le `fix` ci-dessus a le type $\forall\alpha,\beta.\ ((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta)$.

L'opérateur `fix` sur des types non-fonctionnels (p.ex. (`int` → `int`) → `int`) n'est pas définissable avec des références.

Exercice 5.3

$$c/s \twoheadrightarrow c/s \qquad op/s \twoheadrightarrow op/s \qquad (\mathtt{fun}\ x \to a)/s \twoheadrightarrow (\mathtt{fun}\ x \to a)/s$$

$$\frac{a_1/s \twoheadrightarrow (\mathtt{fun}\ x \to a)/s_1 \qquad a_2/s_1 \twoheadrightarrow v_2/s_2 \qquad a[x \leftarrow v_2]/s_2 \twoheadrightarrow v/s_3}{a_1\ a_2/s \twoheadrightarrow v/s_3}$$

$$\frac{a_1/s \twoheadrightarrow v_1/s_1 \quad a_2/s_1 \twoheadrightarrow v_2/s_2}{(a_1, a_2)/s \twoheadrightarrow (v_1, v_2)/s_2} \qquad \frac{a/s \twoheadrightarrow (v_1, v_2)/s'}{\texttt{fst}\ a/s \twoheadrightarrow v_1/s' \quad \texttt{snd}\ a/s \twoheadrightarrow v_2/s'}$$

$$\frac{a_1/s \twoheadrightarrow v_1/s_1 \quad a_2[x \leftarrow v_1]/s_1 \twoheadrightarrow v/s_2}{(\texttt{let}\ x = a_1\ \texttt{in}\ a_2)/s \twoheadrightarrow v/s_2} \qquad \frac{a/s \twoheadrightarrow v/s_1 \quad \ell \notin \mathrm{Dom}(s_1)}{\texttt{ref}\ a/s \twoheadrightarrow \ell/s_1\{\ell \mapsto v\}} \qquad \frac{a/s \twoheadrightarrow \ell/s_1}{!a/s \twoheadrightarrow s_1(\ell)/s_1}$$

$$\frac{a_1/s \twoheadrightarrow \ell/s_1 \quad a_2/s_1 \twoheadrightarrow v_2/s_2}{s, (a_1 := a_2) \twoheadrightarrow (\,)s_2\{\ell \mapsto v_2\}}$$

Exercice 5.4 L'η-expansion a pour effet de changer l'ordre dans lequel une fonction entrelace passage d'arguments et calculs internes. Par exemple :

```
let f = fun x →
          print_string "f";
          fun y → (x, y) in
let g = f 1 in
(g true, g "hello")
```

La chaîne `f` est affichée une seule fois, alors que si l'on fait une η-expansion dans la définition de `g` (afin de la rendre non-expansive), on obtient :

```
let f = fun x →
          print_string "f";
          fun y → (x, y) in
let g = fun y → f 1 y in
(g true, g "hello")
```

et maintenant `f` est affichée deux fois. Toute fonction qui effectue des effets de bords entre le passage de deux arguments permet donc d'observer une différence de comportement lorsqu'on eta-expanse une application partielle de cette fonction. De tels exemples n'apparaissent cependant que très rarement en pratique. Un peu plus plausible est l'exemple de fonctions qui font une grande quantité de calculs (sans effets de bord) entre le passage de deux arguments. Là, l'η-expansion va changer non pas la sémantique du programme, mais son temps d'exécution. Exemple : on considère une fonction qui trie des tables (clé, valeurs) par ordre croissant des clés. On peut vouloir l'écrire sous la forme d'une fonction qui prend d'abord le tableau des clés, calcule la permutation qui trie ces clés, puis prend un tableau de valeurs et y applique la permutation :

```
let tri_table = fun table →
                  let perm = trier table in
                  fun données → appliquer_permutation perm données
```

Cette écriture est avantageuse si l'on s'attend à avoir plusieurs tables de valeurs qui partagent les mêmes clés : on peut alors calculer la permutation de tri une seule fois.

```
let fonction_de_tri = tri_table clés in
... fonction_de_tri données1 ... fonction_de_tri données2 ...
```

Avec la restriction de la généralisation aux expressions non-expansives, `fonction_de_tri` devient monomorphe, et donc l'écriture ci-dessus n'est possible que si `données1` et `données2` sont du même type. Mais sinon, il faut faire une η-expansion sur `fonction_de_tri` :

```
let fonction_de_tri = fun d → tri_table clés d in
... fonction_de_tri données1 ... fonction_de_tri données2 ...
```

et la permutation de tri est recalculée à chaque appel de `fonction_de_tri`.

Exercice 5.5 On démontre d'abord que le typage est préservé chacune des nouvelles règles de réduction. Pour $(\texttt{try}\ v\ \texttt{with}\ x \to a) \xrightarrow{\varepsilon} v$, il est clair que si $\Gamma \vdash (\texttt{try}\ v\ \texttt{with}\ x \to a) : \tau$, alors $\Gamma \vdash v : \tau$ également. Pour $(\texttt{try raise}\ v\ \texttt{with}\ x \to a) \xrightarrow{\varepsilon} a[x \leftarrow v]$, une dérivation de typage du membre gauche est nécessairement de la forme

$$\dfrac{\dfrac{\Gamma \vdash \texttt{raise} : \texttt{exn} \to \tau \qquad \Gamma \vdash v : \texttt{exn}}{\Gamma \vdash \texttt{raise}\ v : \tau} \qquad \Gamma; x : \texttt{exn} \vdash a : \tau}{\Gamma \vdash (\texttt{try raise}\ v\ \texttt{with}\ x \to a) : \tau}$$

En appliquant le lemme de substitution (proposition 2.2) à $\Gamma \vdash v : \texttt{exn}$ et $\Gamma; x : \texttt{exn} \vdash a : \tau$, il vient $\Gamma \vdash a[x \leftarrow v] : \tau$ comme désiré. Enfin, en ce qui concerne la règle $\Delta[\texttt{raise}\ v] \xrightarrow{\varepsilon} \texttt{raise}\ v$, le résultat découle immédiatement du fait que $\texttt{raise}\ v$ a tous les types.

Il faut ensuite montrer une propriété de progression similaire à la proposition 2.5. Le problème, ici, est que l'évaluation peut s'arrêter à cause d'une exception non rattrapée, c'est-à-dire lorsqu'on évalue une sous-expression $\texttt{raise}\ v$ qui n'est contenue dans aucun `try...with`. La propriété de progression est donc :

> Si $\emptyset \vdash a : \tau$, alors ou bien a est une valeur, ou bien $a = \texttt{raise}\ v$ pour une certaine valeur v, ou bien il existe un terme a' tel que $a \to a'$.

La preuve de ce lemme est semblable à celle de la proposition 2.5, avec des cas supplémentaires montrant que si une sous-expression est $\texttt{raise}\ v$, alors l'expression tout entière se réduit, soit par la règle pour $\texttt{try raise}\ v\ \texttt{with}\ldots$, soit par la règle pour $\Delta[\texttt{raise}\ v]$.

Finalement, on obtient qu'une expression a close et bien typée se réduit en une valeur, ou se réduit en $\texttt{raise}\ v$, ou diverge.

Exercice 5.6 Avec les exceptions, on peut écrire une fonction qui a plusieurs résultats : son résultat "normal" (la valeur du corps de la fonction) plus un autre résultat qui est renvoyé en levant une exception contenant cet autre résultat dedans. Autant le type du résultat "normal" est apparent dans le type de la fonction (et ne peut donc être plus complexe que le type de la fonction elle-même), autant le type du résultat "exceptionnel" n'est pas visible dans le type de la fonction et peut être plus complexe que ce dernier. En appliquant cette idée, on peut "cacher" une fonctionnelle de type $(\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$ à l'intérieur d'une humble fonction de type $\texttt{int} \to \texttt{int}$:

```
exception M of (int → int) → (int → int)

let cacher f = fun n → (raise (M f) ; 0)
```

La fonction f cachée dans le résultat de `cacher` f peut ensuite être extraite par la fonction suivante :

```
let retrouver g = try (g 0 ; fun z → z) with M f → f
```

Ceci permet d'exprimer l'exemple bien connu de λ-terme qui boucle $(\lambda f.f\ f)\ (\lambda f.f\ f)$ de la manière suivante :

```
let delta = fun f → (retrouver f) f in delta (cacher delta)
```

On peut aussi définir l'opérateur de point fixe $\texttt{fix} = \lambda f.\ (\lambda g.\lambda x.f\ (g\ g)\ x)\ (\lambda g.\lambda x.f\ (g\ g)\ x)$ comme suit :

```
let fix = fun f →
            let d = fun g → fun x → f (retrouver g g) x in
            d (cacher d)
```

Essayez en Caml ; ça marche !

```
let fact = fix (fun f → fun n → if n = 0 then 1 else n * f(n-1))
in fact 5
```

Annexe D

Corrigés des exercices du chapitre 6

Exercice 6.1 Supposons qu'un type enregistrement contienne uniquement l'ensemble des champs effectivement présents, ainsi que leur type. La grammaire des types sera donc

$$\tau ::= \ldots \mid \{(e : \tau_e)_{e \in L}\}$$

Quels schémas de types peut-on alors attribuer à l'opérateur d'accès $\texttt{proj}_e$? On peut en trouver une infinité :

$$\begin{gathered} \forall\alpha.\{e : \alpha\} \to \alpha \\ \forall\alpha,\alpha'.\{e : \alpha; e' : \alpha'\} \to \alpha' \\ \ldots \end{gathered}$$

Tous sont sûrs, mais ces schémas sont incomparables deux à deux. En fait, ces schémas sont trop spécifiques : ils indiquent si les champs autres que e doivent être absents ou présents, alors qu'on voudrait ignorer ce détail, puisqu'on ne s'intéresse qu'au champ e. Il nous faut donc un langage de types plus général, dans lequel l'absence ou la présence d'un champ puisse être représentée par une *variable*. C'est ce que permet l'introduction des constructeurs de types `Abs` et `Pre`.

Exercice 6.2

```
type ('a, 'b, 'c) enreg = { e : 'a ; f : 'b ; g : 'c }
type 'a pre = Pre of 'a
type abs = Abs

let vide = { e = Abs ; f = Abs ; g = Abs }

let enreg_e a = { e = Pre a ; f = Abs ; g = Abs }

let proj_e r = match r.e with Pre v -> v
let proj_f r = match r.f with Pre v -> v
let proj_g r = match r.g with Pre v -> v

let exten_e r x = { e = Pre x ; f = r.f ; g = r.g }
let exten_f r x = { e = r.e ; f = Pre x ; g = r.g }
let exten_g r x = { e = r.e ; f = r.f ; g = Pre x }
```

Exercice 6.4

Hypothèse (H1) pour $\texttt{proj}_e$: supposons $\Gamma \vdash \texttt{proj}_e\{(e = v_e)_{e\in L}\} : \tau$. Une dérivation de ce jugement est forcément de la forme :

$$\dfrac{\Gamma \vdash \texttt{proj}_e : \{e : \texttt{Pre}\ \tau;\ \tau'\} \to \tau \qquad \dfrac{\forall e \in L \quad \Gamma \vdash v_e : \tau_e}{\Gamma \vdash \{(e = v_e)_{e\in L}\} : \{(e : \texttt{Pre}\ \tau_e)_{e\in L};\ \partial\texttt{Abs}\}}}{\Gamma \vdash \texttt{proj}_e\{(e = v_e)_{e\in L}\} : \tau}$$

D'où $\tau = \tau_e$, et le résultat attendu puisque nous avons une sous-dérivation de $\Gamma \vdash v_e : \tau_e$ et que le résultat de la δ-réduction est v_e.

Hypothèse (H1) pour $\texttt{exten}_e$: supposons $\Gamma \vdash \texttt{exten}_{e'}(\{(e = v_e)_{e\in L}\}, w) : \tau$. La dérivation de ce typage est de la forme :

$$\dfrac{\Gamma \vdash \texttt{exten}_{e'} : \tau' \times \tau_w \to \tau \qquad \dfrac{\dfrac{\forall e \in L \quad \Gamma v_e : \tau_e}{\Gamma \vdash \{(e = v_e)_{e\in L}\} : \tau'} \quad \Gamma \vdash w : \tau_w}{\Gamma \vdash (\{(e = v_e)_{e\in L}\}, w) : \tau' \times \tau_w}}{\Gamma \vdash \texttt{exten}_{e'}(\{(e = v_e)_{e\in L}\}, w) : \tau}$$

où $\tau' = \{(e : \texttt{Pre}\ \tau_e)_{e\in L};\ \partial\texttt{Abs}\}$. Si e' n'est pas dans L, alors nous avons $\tau = \{(e : \texttt{Pre}\ \tau_e)_{e\in L};\ e' : \texttt{Pre}\ \tau_w;\ \partial\texttt{Abs}\}$. D'autre part, le résultat de la δ-réduction est $\{(e = v_e)_{e\in L};\ e' = w\}$. Comme e' n'est pas élément de L, la règle (record) appliquée aux hypothèses $\Gamma \vdash w : \tau_w$ et $\Gamma \vdash v_e : \tau_e$ montre que le résultat de la delta-réduction a le type τ, comme désiré. Si au contraire e' appartient à L, alors nous avons $\tau = \{e' : \texttt{Pre}\ \tau_w; (e : \texttt{Pre}\ \tau_e)_{e\in L\setminus\{e'\}};\ \partial\texttt{Abs}\}$. Le résultat de la δ-réduction est $\{e' = w;\ (e = v_e)_{e\in L\setminus\{e'\}}\}$. On conclut que ce résultat est de type τ à partir des hypothèses $\Gamma \vdash w : \tau_w$ et $\Gamma \vdash v_e : \tau_e$ pour $e \neq e'$.

Forme des valeurs selon leur type : on examine les règles de typage qui peuvent s'appliquer à des valeurs (const-inst, op-inst, fun, paire, record) et on remplit la matrice suivante des combinaisons (valeur, type) possibles :

	c	op	$\texttt{fun}\ x \to a$	(v_1, v_2)	$\{(e = v_e)_{e\in L}\}$
T	X				
$\tau_1 \to \tau_2$		X	X		
$\tau_1 \times \tau_2$				X	
$\{\tau\}$					X

(Rappelons que par l'hypothèse H0 un opérateur a toujours un type flèche). De plus, le type d'une valeur enregistrement $\{(e = v_e)_{e\in L}\}$ est nécessairement de la forme $\{(e : \texttt{Pre}\ \tau_e)_{e\in L};\ \partial\texttt{Abs}\}$ avec $\emptyset \vdash v_e : \tau_e$. Donc, toutes les étiquettes marquées présentes dans ce type enregistrement sont bien présentes dans la valeur.

Hypothèse (H2) pour $\texttt{proj}_e$: supposons $\emptyset \vdash \texttt{proj}_e(v) : \tau$. Vu le schéma de types de $\texttt{proj}_e$, nous avons $\emptyset \vdash v : \{e : \texttt{Pre}\ \tau_1;\ \tau_2\}$. Donc, v est un enregistrement et contient un champ étiqueté e. Par conséquent, l'application de $\texttt{proj}_e$ se réduit par la δ-règle pour les projections.

Hypothèse (H2) pour $\texttt{exten}_e$: si $\emptyset \vdash \texttt{exten}_e(v) : \tau$, v a nécessairement un type enregistrement. C'est donc un enregistrement, et l'application $\texttt{exten}_e(v)$ se réduit par la δ-règle pour $\texttt{exten}_e$.

Exercice 6.5
Algorithme de sortage : $S(\tau, \kappa, K) = K'$ si τ est de la sorte κ, ou échec sinon.

- Si $\tau = \alpha$ et $\alpha \notin Dom(K)$:
  prendre $K' = K[\alpha \mapsto \kappa]$.
- Si $\tau = \alpha$ et $\alpha \in Dom(K)$:
  si $\kappa = K(\alpha)$, prendre $K' = K$, sinon échec.
- Si $\tau = T$:
  si $\kappa = \texttt{TYPE}$, prendre $K' = K$, sinon échec.
- Si $\tau = \tau_1 \to \tau_2$ ou $\tau = \tau_1 \times \tau_2$:
  si $\kappa = \texttt{TYPE}$, prendre $K' = S(\tau_2, \texttt{TYPE}, S(\tau_1, \texttt{TYPE}, K))$, sinon échec.
- Si $\tau = \{\tau'\}$:
  si $\kappa = \texttt{TYPE}$, prendre $K' = S(\tau', \texttt{R}(\emptyset), \tau')$, sinon échec.
- Si $\tau = \emptyset$:
  si $\kappa = \texttt{R}(L)$ pour un certain L, prendre $K' = K$, sinon échec.
- Si $\tau = (e : \tau_1;\ \tau_2)$:
  Si $\kappa = \texttt{TYPE}$ ou $\kappa = \texttt{PRE}$, échec.
  Si $\kappa = \texttt{R}(L)$ et $e \in L$, échec.
  Si $\kappa = \texttt{R}(L)$ et $e \notin L$, prendre $K' = S(\tau_2, \texttt{R}(L \cup \{e\}), S(\tau_1, \texttt{TYPE}, K))$.
- Si $\tau = \texttt{Abs}$:
  si $\kappa = \texttt{PRE}$, prendre $K' = K$, sinon échec.
- Si $\tau = \texttt{Pre}\ \tau'$:
  si $\kappa = \texttt{PRE}$, prendre $K' = S(\tau', \texttt{TYPE}, K)$, sinon échec.

Pour typer une contrainte $(a : \tau)$ ou une déclaration de constructeur C **of** τ, on appelle $S(\tau, \texttt{TYPE}, K)$ et l'algorithme S vérifier et propage la contrainte $\tau :: \texttt{TYPE}$ vers les variables de types de τ, déterminant au passage leurs sortes.

Exercice 6.10
L'idée est d'interpréter la rangée τ dans le type somme $[\tau]$ comme un "ou", et non plus comme un "et" comme dans le cas des enregistrements. Ainsi, le type enregistrement $\{e : \texttt{Pre int};\ f : \texttt{Pre bool};\ \partial\texttt{Abs}\}$ signifie "il y a un champ e de type `int` *et* un champ f de type `bool` *et* rien d'autre". Le type somme $[C : \texttt{Pre int};\ D : \texttt{Pre bool};\ \partial\texttt{Abs}]$ signifie, lui, "il y a un constructeur C qui porte un argument de type `int` *ou* un constructeur D d'argument `bool` *ou* rien d'autre". Si la rangée se termine par une variable α au lieu de $\partial\texttt{Abs}$, cela signifie "… ou d'autres constructeurs, représentés par α" au lieu de "… ou rien d'autre". Avec ces intuitions, on obtient les types suivants pour les opérateurs :

$$\begin{array}{rcl} C & : & \forall \alpha, \beta.\ \alpha \to [C : \texttt{Pre}\ \alpha;\ \beta] \\ P_C & : & \forall \alpha.\ [C : \texttt{Pre}\ \alpha;\ \partial\texttt{Abs}] \to \alpha \\ F_C & : & \forall \alpha, \beta, \gamma.\ [C : \texttt{Pre}\ \alpha;\ \beta] \times (\alpha \to \gamma) \times ([C : \texttt{Abs};\ \beta] \to \gamma) \to \gamma \end{array}$$

Le type de C indique que le résultat contient le constructeur C avec un argument de type α, mais peut aussi être vu comme contenant d'autres constructeurs si le contexte le demande. Ainsi, $C(1)$ a le type $[C : \texttt{Pre int};\ \tau]$ pour toute rangée τ, et **if** *cond* **then** $C(1)$ **else** $D(\texttt{true})$ a le type $[C : \texttt{Pre int};\ D : \texttt{Pre bool};\ \tau]$, puisque le résultat est soit un C soit un D.

Le type de P_C exprime que l'argument doit posséder le constructeur C et aucun autre, c.à.d. que l'on est sûr statiquement que la projection ne peut pas échouer.

Enfin, le type de F_C indique que le premier argument doit contenir le constructeur C et peut-être d'autres constructeurs. Le second argument doit s'appliquer à l'argument de C. Quant au troisième argument, il doit s'appliquer à tous les constructeurs qui peuvent être dans le premier argument, *sauf* C. En effet, le troisième argument ne sera jamais appliqué si le premier est de constructeur C. On obtient en particulier les types suivants :

$$
\begin{array}{rcl}
\mathtt{fun}\ x \rightarrow F_C(x, \mathtt{fun}\ y \rightarrow y, \mathtt{fun}\ z \rightarrow 0) & : & [C : \mathtt{Pre\ int};\ \alpha] \rightarrow \mathtt{int} \\
\mathtt{fun}\ x \rightarrow F_C(x, \mathtt{fun}\ y \rightarrow y, P_D) & : & [C : \mathtt{Pre}\ \alpha;\ D : \mathtt{Pre}\ \alpha;\ \partial\mathtt{Abs}] \rightarrow \alpha
\end{array}
$$

La première fonction correspond, en ML, à un filtrage “ouvert” (avec un cas par défaut à la fin) : elle peut s'appliquer à n'importe quelle somme contenant au moins le constructeur C d'argument `int`. La seconde fonction correspond à un filtrage “fermé” (sans cas attrape-tout à la fin) : elle ne peut s'appliquer qu'à des sommes qui contiennent au plus les constructeurs C et D.