Notes du cours de DEA
“Typage et programmation”

Xavier Leroy
Xavier.Leroy@inria.fr

Version du December 13, 2001
http://pauillac.inria.fr/~xleroy/dea/

Contents

Chapter 1

Mini-ML: évaluation et typage

1.1 Syntaxe de mini-ML

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

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

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

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

Exemples d'expressions:

`+ (3, 2)`

le calcul de 3 plus 2

`3 + 2`

le même, avec notation infixe pour le `+`

`fun x → + (x, 1)`

la fonction "successeur"

`fun f → fun g → fun x → f(g x)`

la composition de fonctions

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

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

la fonction "fois 16"

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

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

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

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

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

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

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

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

1.2.1 Les valeurs

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

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

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

1.2.2 Rappels sur les règles d'inférences

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

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

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

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

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

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

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

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

Il faut les lire comme les conditions suivantes portant sur les prédicats $\texttt{Pair}$ et $\texttt{Impair}$:

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

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 $\texttt{Pair}$ et $\texttt{Impair}$ comme étant ces deux prédicats.

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

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

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

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

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

Les dérivations caractérisent exactement les plus petits prédicats vérifiant un ensemble de règles d'inférences. 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 $\texttt{Pair}(n)$.

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

1.2.3 Les règles d'évaluation

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

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

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

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

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

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

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

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

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

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

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

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

Exercice 1.5 *(*) On considère l'expression $a = 1\ 2$. Existe-t'il une valeur v telle que $a \stackrel{v}{\to} v$? Même question avec l'expression $a' = (\texttt{fun}\ f \to (f\ f))\ (\texttt{fun}\ f \to (f\ f))$. 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 1.5?*

1.2.4 Déterminisme de l'évaluation

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

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

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

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

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

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

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

et de même D' est de la forme

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

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

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

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

et D' est aussi de la forme

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

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

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

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

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

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

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

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

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

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 réellement aléatoire entre 0 et n. Sa règle d'évaluation est:

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

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

1.3 Typage de mini-ML

Le but du typage statique est de détecter et de rejeter à la compilation un certain nombre de programmes absurdes, comme $1\ 2$ ou $(\texttt{fun}\ x \rightarrow x) + 1$. 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. Pour être effctif, un système de types statique doit être décidable: il ne s'agit pas d'exécuter entièrement le programme lors de la compilation.

1.3.1 L'algèbre de types

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

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

1.3.2 Les règles de typage – typage monomorphe

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

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

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

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

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

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

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

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

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

Autres exemples de typages que l'on peut dériver:

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

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

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

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

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

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

1.3.3 Schémas de types

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.3.4 Les règles de typage – typage polymorphe à la ML

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

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

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

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

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

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

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

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

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

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

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

Exemple:

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

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

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

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

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

1.3.5 Quelques propriétés des règles de typage

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

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

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

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

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

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

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

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

si et seulement si on peut dériver

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

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

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

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

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

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

Chapter 2

Sûreté du typage et sémantique à réduction

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

2.1 Distinguer erreurs d'exécution et non-terminaison

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

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

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

La sémantique opérationnelle structurée du chapitre 1 n'a pas de règles d'évaluation qui s'applique à ces expressions erronées. Autrement dit, si a est une expression dont l'évaluation provoque une de ces erreurs, il n'existe pas de valeur v telle que $a \stackrel{v}{\to} v$. 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 $a \stackrel{v}{\to} v$.

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

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

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

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

est fausse. Par exemple,

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

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

2.1.2 Règles d'erreurs en sémantique opérationnelle structurelle

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

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

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

$$\frac{a_1 \overset{v}{\to} \mathtt{fst} \qquad a_2 \overset{v}{\to} v_2 \qquad v_2 \text{ n'est pas une paire}}{a_1\ a_2 \overset{v}{\to} \mathtt{err}} \quad (13)$$

$$\frac{a_1 \overset{v}{\to} \mathtt{snd} \qquad a_2 \overset{v}{\to} v_2 \qquad v_2 \text{ n'est pas une paire}}{a_1\ a_2 \overset{v}{\to} \mathtt{err}} \quad (14)$$

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

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

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

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

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

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

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

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

2.2 Sémantique à réductions

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

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

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

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

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

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

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

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

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

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

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

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

Contextes d'évaluation:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Tester votre code sur les exemples vus jusqu'ici.

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

2.3 Sûreté du typage

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

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

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

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

2. a se réduit à l'infini

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

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

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

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

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

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

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

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

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

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

2.3.1 La relation "être moins typable"

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

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

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

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

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

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

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

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

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

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

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

2.3.2 Hypothèses sur les opérateurs

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

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

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

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

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

2.3.3 Typage et substitution

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

Proposition 2.2 (Lemme de substitution) *Supposons*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Si $y = x$, nous avons $a[x \leftarrow a'] = \texttt{let}\ x = a_1[x \leftarrow a']\ \texttt{in}\ a_2$, et $E_x + \{x : Gen(\tau_1, E_x)\} = E + \{x : Gen(\tau_1, E_x)\}$. Remarquons que $Gen(\tau_1, E) \geq Gen(\tau_1, E_x)$ puisque $\mathcal{L}(E) \subseteq \mathcal{L}(E_x)$. Par stabilité du typage par renforcement des hypothèses (proposition 1.4), $E + \{x : Gen(\tau_1, E_x)\} \vdash a_2 : \tau$ implique $E + \{x : Gen(\tau_1, E)\} \vdash a_2 : \tau$. D'où la dérivation du résultat attendu:

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

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

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

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

2.3.4 Préservation du typage par réduction

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

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

Démonstration: par cas sur la règle de réduction utilisée.

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

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

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

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

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

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

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

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

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

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

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

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

2.3.5 Les formes normales bien typées sont des valeurs

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

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

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

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

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

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

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

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

Cas $a = c$ ou $a = op$ ou $a = \mathtt{fun}\ x \to a$. a est une valeur.

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

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

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

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

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

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

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

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

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

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

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

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

Démonstration: conséquence immédiate de la proposition 2.6. □

2.3.6 Pour finir

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

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

Chapter 3

Inférence de types

3.1 Introduction à l'inférence de types

Pour un langage statiquement typé, un *typeur* est un algorithme qui prend un programme en entrée et détermine si ce programme est typable ou non. En général, il va aussi déterminer un type τ pour ce programme.

Si plusieurs types τ sont possibles, on essaye de renvoyer comme résultat du typeur un *type principal*, c'est-à-dire un type plus précis que tous les autres types possibles (pour une notion de "plus précis" qui dépend du système de types considéré). Par exemple, en mini-ML, `fun` $x \to x$ a les types $\tau \to \tau$ pour n'importe quel type τ, mais le type $\alpha \to \alpha$ est principal, puisque tous les autres types possibles s'en déduisent par substitution.

La quantité de travail fournie par le typeur est inversement proportionnelle à la quantité de déclarations de types présentes dans le langage source, et donc à la quantité d'information de typage écrite par le programmeur:

Vérification pure: Dans le source, toutes les sous-expressions du programme, ainsi que tous les identificateurs, sont annotés par leur type.

```
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 programme source contient déjà autant d'informations de typage que la dérivation de typage correspondante. La patience du programmeur est mise à rude épreuve par la quantité d'annotations de types à fournir. Aucun langage réaliste ne suit cette approche.

Déclaration des types des identificateurs et propagation des types: Le programmeur déclare les types des paramètres de fonctions et des variables locales. Le typeur infère les types des sous-expressions en "propageant" les types des feuilles de l'expression 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` $\rightarrow$ `int` pour cette expression. Cette approche est suivie par la plupart des langages impératifs: Pascal, C, Java, ... (En fait, ces langages exigent un peu plus d'annotations de types; par exemple, il faut aussi déclarer le type du résultat des fonctions.)

Déclaration des types des paramètres de fonction et propagation des types: La seule différence par-rapport à l'approche précédente est que les variables locales (p.ex. les identificateurs liés par `let`) ne sont pas annotées par leur type, ce dernier étant déterminé par le type de l'expression liée à la variable. Exemple:

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

Ayant déterminé que `+(x,1)` est de type `int`, le typeur déduit que `y` est de type `int` dans le reste de la fonction.

Inférence complète de types: Le programme source ne contient aucune déclaration de type sur les paramètres de fonctions ni sur les variables locales. Le typeur détermine le type des paramètres de fonctions d'après l'utilisation qui en est faite dans le reste du programme. Exemple:

```
fun x → let y = +(x,1) in y
```

Puisque l'addition `+` n'opère que sur des paires d'entiers, `x` est forcément de type `int`. C'est l'approche suivie dans les langages de la famille ML.

Dans ce cours, nous allons nous concentrer sur la dernière approche (inférence complète), car les autres sont techniquement très simples. On pourra faire l'exercice suivant pour se familiariser avec l'avant-dernière approche.

Exercice de programmation 3.1 *Écrire un typeur pour mini-ML avec typage monomorphe (section 1.3.2) et dans lequel les paramètres de fonctions sont annotés dans le programme source par leur type:*

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

Il s'agit donc de la troisième approche dans la liste ci-dessus. On écrira le typeur sous la forme d'une fonction prenant une expression a et un environnement de typage E en arguments, et renvoyant un type τ pour a dans E s'il existe, ou bien échoue (en levant une exception) si a n'est pas typable dans E.

Quels problèmes se posent si l'on veut étendre ce typeur à mini-ML avec typage polymorphe (en gardant les paramètres de fonctions annotés par leurs types)?

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 1.3.2. 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 au système d'équation; cela nous donne le type principal du programme.

En combinant ces deux phases, on obtient un algorithme de typage qui détermine si un programme est bien typé et si oui, calcule son type principal.

3.2.1 Construction du système d'équations

On se donne un programme (une expression close) a_0 dans laquelle tous les identificateurs liés par `fun` ou `let` ont des noms différents. À chaque identificateur x dans a_0, on associe une variable de type α_x. De même, à chaque sous-expression a de a_0, on associe une variable de type α_a.

Le système d'équations $C(a_0)$ associé à a_0 est construit en parcourant l'expression a_0 et en ajoutant des équations pour chaque sous-expression a de a_0, comme suit:

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

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 de l'ensemble d'équations $C(a)$.

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

Proposition 3.1 (Correction des solutions vis-à-vis du typage) *Si φ est une solution de $C(a)$, alors $E \vdash a : \varphi(\alpha_a)$ où E est l'environnement de typage $\{x : \varphi(\alpha_x) \mid x$ libre dans $a\}$.*

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 $E + \{x : \varphi(\alpha_x)\} \vdash b : \varphi(\alpha_b)$. On peut donc construire la dérivation suivante:

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

c'est-à-dire $E \vdash a : \varphi(\alpha_a)$, 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 $E \vdash b : \varphi(\alpha_b)$ et $E \vdash c : \varphi(\alpha_c)$. Comme $\varphi(\alpha_b) = \varphi(\alpha_c) \to \varphi(\alpha_a)$, on a la dérivation suivante:

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

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

Proposition 3.2 (Complétude des solutions vis-à-vis du typage) *Soit a une expression. S'il existe un environnement E et un type τ tels que $E \vdash a : \tau$, alors le système d'équations $C(a)$ admet une solution.*

Démonstration: on construit une solution φ par récurrence sur la dérivation de $E \vdash a : \tau$ et par cas sur la dernière règle de typage utilisée. Cette solution φ vérifie de plus les propriétés suivantes:

1. $\varphi(\alpha_a) = \tau$
2. $\varphi(\alpha_x) = E(x)$ pour tout $x \in \mathrm{Dom}(E)$
3. le domaine de φ est α_x pour $x \in \mathrm{Dom}(E)$ et α_b pour toute sous-expression b de a.

On montre deux cas représentatifs de la preuve; les autres sont similaires.

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

$$\frac{E + \{x : \tau_1\} \vdash b : \tau_2}{E \vdash \texttt{fun}\ x \to b : \tau_1 \to \tau_2}$$

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 de plus (1), (2) et (3). On prend $\varphi = \varphi' + [\alpha_a \leftarrow \tau]$. 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'. Enfin, les propriétés (1), (2) et (3) sont vérifiées pour φ.

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

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

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 les propriétés (1)–(3). Par la propriété (2), il vient $\varphi_1(\alpha_x) = \varphi_2(\alpha_x)$ pour tout $x \in \mathrm{Dom}(E)$. On peut donc prendre $\varphi = \varphi_1 + \varphi_2 + [\alpha_a \leftarrow \tau]$. C'est une substitution qui prolonge φ_1 et φ_2. Donc, φ 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 $C(a)$, et de plus vérifie (1)–(3). CQFD. □

3.2.3 Résolution des équations

L'ensemble $C(a)$ peut être vu comme un problème d'unification du premier ordre: c'est un ensemble d'équations entre équations de types, qui sont des termes du premier ordre construits sur la signature $T \cup \{\to, \times\}$. Il existe donc un algorithme mgu qui, étant donné un ensemble d'équations C, a l'un des deux comportements suivants:

- $\texttt{mgu}(C)$ échoue, signifiant que C n'a pas de solution.
- $\texttt{mgu}(C)$ renvoie une substitution φ (un unificateur principal de C) qui est une solution de C, et de plus telle que toute autre solution ψ de C peut s'écrire $\psi = \theta \circ \varphi$ pour une certaine substitution θ.

(Voir le cours de J.-P. Jouannaud pour plus de détails.) Un algorithme qui convient est l'algorithme d'unification de Robinson, que nous rappelons ci-dessous.

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

Dans tous les autres cas, $\texttt{mgu}(C)$ échoue et C n'a pas de solutions.

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.

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

3.2.4 L'algorithme d'inférence

En recollant les morceaux, on obtient l'algorithme d'inférence de types $I(a)$ suivant:
Entrée: une expression close a.
Sortie: ou bien `err`, ou bien un type τ.
Calcul: calculer $\varphi = \texttt{mgu}(C(a))$. Si `mgu` échoue, renvoyer `err`. Sinon, renvoyer $\varphi(\alpha_a)$.

Proposition 3.3 (Propriétés de l'algorithme I)

1. *Correction: si $I(a)$ est un type τ, alors $\emptyset \vdash a : \tau$.*
2. *Complétude: si $I(a)$ est* `err`*, alors a n'est pas typable dans $\emptyset$.*
3. *Principalité du type inféré: s'il existe un type τ' tel que $\emptyset \vdash a : \tau'$, alors $I(a)$ n'est pas* `err`*; au contraire, c'est un type τ, et de plus il existe une substitution θ telle que $\tau' = \theta(\tau)$.*

Démonstration: (1) est conséquence du lemme 3.1 et de la correction de l'algorithme `mgu`: le résultat de $\texttt{mgu}(C(a))$, s'il existe, est une solution de $C(a)$.

Pour (3), supposons $\emptyset \vdash a : \tau'$. Par le lemme 3.2, cela signifie qu'il existe une solution φ' de $C(a)$ avec de plus $\tau' = \varphi'(\alpha_a)$. Puisque $C(a)$ admet une solution, $\varphi = \texttt{mgu}(C(a))$ existe, et de plus $\varphi' = \theta \circ \varphi$ pour une certaine substitution θ (par principalité de l'unificateur calculé par `mgu`). Donc, $\tau' = \varphi'(\alpha_a) = \theta(\varphi(\alpha_a)) = \theta(\tau')$.

Pour (2), la contraposée de (2), "si a est typable, alors $I(a) \neq \texttt{err}$", est un corollaire immédiat de (3). □

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

3.3 Inférence de types pour mini-ML avec typage polymorphe

L'approche de la section 3.2 ne s'étend pas facilement au typage polymorphe de la section 1.3.4. En effet, les contraintes de typages $C(a)$ devraient contenir non seulement des équations d'unification entre types, mais aussi des contraintes d'instanciation (pour les règles (var-inst), (const-inst) et (op-inst)) et de généralisation (pour la règle (let-gen)):

- Si a est une variable x: $C(a) = \{\alpha_a \leq \alpha_x\}$.
- Si a est `let` $x = b$ `in` c: $C(a) = \{\alpha_x \stackrel{?}{=} \texttt{Gen}(\alpha_b, E);\ \alpha_a \stackrel{?}{=} \alpha_c\} \cup C(b) \cup C(c)$
  où E est l'environnement $\{y : \alpha_y\}$ pour tout y lié à l'endroit où apparaît l'expression `let`.

Remarquons que les α_x sont maintenant des schémas et non plus des types simples.

La résolution des contraintes $C(a)$ est maintenant beaucoup plus difficile qu'un problème d'unification du premier ordre. En particulier, on ne peut plus résoudre les contraintes d'unification et les contraintes de généralisation dans n'importe quel ordre.

Exemple: on considère

$$a = \texttt{let}\ x = \underbrace{\texttt{fun}\ y \to \underbrace{y}_{c}}_{b}\ \texttt{in}\ x\ 1$$

On obtient les contraintes suivantes (entre autres):

$$\begin{aligned}
\alpha_x &= \texttt{Gen}(\alpha_b, \emptyset) \\
\alpha_b &= \alpha_y \to \alpha_c \\
\alpha_c &= \alpha_y
\end{aligned}$$

Si l'on résout la première immédiatement, on obtient $\alpha_x = \forall \alpha_b.\alpha_b$, ce qui n'est clairement pas correct puisque b a forcément un type fonctionnel. Il faut donc avoir résolu au préalable les contraintes sur α_b et α_c avant de calculer α_x.

Pour résoudre ce problème, nous allons voir un algorithme qui entremêle construction de contraintes et résolution de ces contraintes par unification, en une seule passe sur le programme d'entrée.

3.3.1 L'algorithme W de Damas-Milner-Tofte

On commence par définir la notion d'instance triviale $\texttt{Inst}(\sigma, V)$ d'un schéma de types σ par-rapport à un ensemble de variables "nouvelles" V:

$$\texttt{Inst}(\forall \alpha_1, \ldots, \alpha_n.\tau,\ V) = (\tau[\alpha_i \leftarrow \beta_i],\ V \setminus \{\beta_1 \ldots \beta_n\})$$

où $\beta_1, \ldots, \beta_n$ sont n variables distinctes choisies dans V.

L'algorithme W est alors le suivant:

Entrée: un environnement de typage E, une expression a, et un ensemble de variables V.

Sortie: une erreur ou bien un triplet (τ, φ, V').
τ est le type inféré pour l'expression a.
φ est la substitution à effectuer sur les variables libres de E afin que a soit typable dans E.
V' est V privé des variables "nouvelles" que W a utilisées.

Calcul:

- Si a est une variable x avec $x \in \mathrm{Dom}(E)$:
  prendre $(\tau, V') = \texttt{Inst}(E(x), V)$ et $\varphi = id$.

- Si a est une constante c ou un opérateur op:
  prendre $(\tau, V') = \texttt{Inst}(TC(a), V)$ et $\varphi = id$.

- Si a est `fun` $x \to a_1$:
  soit α une nouvelle variable prise dans V
  soit $(\tau_1, \varphi_1, V_1) = W(E + \{x : \alpha\}, a_1, V \setminus \{\alpha\})$
  prendre $\tau = \varphi_1(\alpha) \to \tau_1$ et $\varphi = \varphi_1$ et $V' = V_1$.

- Si a est une application $a_1\ a_2$:
  soit $(\tau_1, \varphi_1, V_1) = W(E, a_1, V)$
  soit $(\tau_2, \varphi_2, V_2) = W(\varphi_1(E), a_2, V_1)$
  soit α une nouvelle variable prise dans V_2
  soit $\mu = \mathtt{mgu}\{\varphi_2(\tau_1) \stackrel{?}{=} \tau_2 \to \alpha\}$
  prendre $\tau = \mu(\alpha)$ et $\varphi = \mu \circ \varphi_2 \circ \varphi_1$ et $V' = V_2 \setminus \{\alpha\}$.

- Si a est une paire (a_1, a_2):
  soit $(\tau_1, \varphi_1, V_1) = W(E, a_1, V)$
  soit $(\tau_2, \varphi_2, V_2) = W(\varphi_1(E), a_2, V_1)$
  prendre $\tau = \varphi_2(\tau_1) \times \tau_2$ et $\varphi = \varphi_2 \circ \varphi_1$ et $V' = V_2$.

- Si a est `let` $x = b$ `in` c:
  soit $(\tau_1, \varphi_1, V_1) = W(E, a_1, V)$
  soit $(\tau_2, \varphi_2, V_2) = W(\varphi_1(E) + \{x : \mathtt{Gen}(\tau_1, \varphi_1(E))\}, a_2, V_1)$
  prendre $\tau = \tau_2$ et $\varphi = \varphi_2 \circ \varphi_1$ et $V' = V_2$.

- Dans tous les cas non couverts par les cas ci-dessus, et en particulier si l'un des appels à `mgu` échoue, ou si V est vide lorsqu'on veut prendre une nouvelle variable dedans, on prend $W(E, a, V) = \mathtt{err}$.

Exemple on considère $a = \mathtt{fun}\ x \to +(x, 1)$. Le déroulement de $W(\emptyset, a, V)$ est le suivant:

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

Exercice de programmation 3.5 *Implémenter l'algorithme W et le faire tourner sur des exemples. (On pourra se dispenser du paramètre V et du résultat V', et à la place générer des variables "nouvelles" en utilisant un compteur.)*

3.3.2 Propriétés de l'algorithme W

Théorème 3.1 (Correction de l'algorithme W) *Si $W(E, a, V) = (\tau, \varphi, V')$, alors on peut dériver $\varphi(E) \vdash a : \tau$.*

La preuve utilise le lemme technique suivant:

Proposition 3.4 (Commutation entre $\mathtt{Gen}$ et substitution) *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 $\varphi(\beta)$. Soit alors un environnement E, un type τ et une substitution φ tels que les variables généralisables $\mathcal{L}(\tau) \setminus \mathcal{L}(E)$ sont toutes hors de portée de φ. Alors, $\mathtt{Gen}(\varphi(\tau), \varphi(E)) = \varphi(\mathtt{Gen}(\tau, E))$.*

Démonstration: on remarque qu'une variable α hors de portée de φ est libre dans un type $\varphi(\tau)$ si et seulement si elle est libre dans le type d'origine τ. Il s'ensuit $\mathcal{L}(\varphi(\tau)) \setminus \mathcal{L}(\varphi(E)) = \mathcal{L}(\tau) \setminus \mathcal{L}(E)$, puis le résultat annoncé. □

Démonstration: (du théorème 3.1) par récurrence structurelle sur a. La preuve utilise de manière essentielle la stabilité du typage par substitution (proposition 1.2). On donne un cas de base, et deux cas qui utilisent l'hypothèse de récurrence; les autres cas sont similaires. On reprend les notations de l'algorithme.

Cas $a = x$ avec $x \in \mathrm{Dom}(E)$. On a $(\tau, V') = \mathtt{Inst}(E(x), V)$ et $\varphi = id$. Par définition de $\mathtt{Inst}$, on a $\tau \leq E(x)$. On peut donc bien dériver $E \vdash x : \tau$ par la règle (inst-var).

Cas $a = a_1\ a_2$. Appliquant l'hypothèse de récurrence aux deux appels récursifs de $\mathtt{W}$, on obtient des dérivations de

$$\varphi_1(E) \vdash a_1 : \tau_1 \qquad \text{et} \qquad \varphi_2(\varphi_1(E)) \vdash a_2 : \tau_2.$$

On applique la substitution $\mu \circ \varphi_2$ à la dérivation de gauche, et μ à celle de droite. Par la proposition 1.2, il vient:

$$\varphi(E) \vdash a_1 : \mu(\varphi_2(\tau_1)) \qquad \text{et} \qquad \varphi(E) \vdash a_2 : \mu(\tau_2).$$

Comme μ est un unificateur de $\{\varphi_2(\tau_1) \stackrel{?}{=} \tau_2 \to \alpha\}$, on a $\mu(\varphi_2(\tau_1)) = \mu(\tau_2) \to \mu(\alpha)$. On peut donc dériver par la règle (app)

$$\varphi(E) \vdash a_1\ a_2 : \mu(\alpha)$$

C'est le résultat attendu.

Cas $a = \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2$. On applique l'hypothèse de récurrence aux deux appels récursifs de W. Il vient des preuves de

$$\varphi_1(E) \vdash a_1 : \tau_1 \qquad \text{et} \qquad \varphi_2(\varphi_1(E) + \{x \leftarrow \mathtt{Gen}(\tau_1, \varphi_1(E))\} \vdash a_2 : \tau_2.$$

Si nécessaire, on renomme dans la dérivation de gauche les variables généralisées pour qu'elles soient hors de portée de φ_2. On a alors par le lemme 3.4

$$\mathtt{Gen}(\varphi_2(\tau_1), \varphi_2(\varphi_1(E))) = \varphi_2(\mathtt{Gen}(\tau_1, \varphi_1(E)))$$

Notant $\varphi = \varphi_2 \circ \varphi_1$, on a donc des preuves de:

$$\varphi(E) \vdash a_1 : \varphi_2(\tau_1) \qquad \text{et} \qquad \varphi(E) + \{x : \mathtt{Gen}(\varphi_2(\tau_1), \varphi(E))\} \vdash a_2 : \tau_2.$$

On conclut, par la règle (let-gen),

$$\varphi(E) \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau_2.$$

C'est le résultat annoncé. □

Définition: étant données deux substitutions φ et ψ et un ensemble de variables V, on dit que $\varphi = \psi$ *hors de* V si $\varphi(\alpha) = \psi(\alpha)$ pour toute variable $\alpha \notin V$. On voit facilement que si $\mathcal{L}(\tau) \cap V = \emptyset$ et si $\varphi = \psi$ hors de V, alors $\varphi(\tau) = \psi(\tau)$.

Théorème 3.2 (Complétude et principalité de l'algorithme W) *Soit V un ensemble de variables infini et tel que $V \cap \mathcal{L}(E) = \emptyset$. S'il existe un type τ' et une substitution φ' tels que $\varphi'(E) \vdash a : \tau'$, alors $W(E, a, V)$ n'est pas* `err`*; au contraire, il existe τ, φ, V' et une substitution θ tels que*

$$W(E, a, V) = (\tau, \varphi, V') \qquad \textit{et} \qquad \tau' = \theta(\tau) \qquad \textit{et} \qquad \varphi' = \theta \circ \varphi \textit{ hors de } V.$$

Démonstration: on commence par remarquer que, avec les hypothèses de la proposition, si $(\tau, \varphi, V') = W(a, E, V)$ est défini, alors $V' \subseteq V$, V' est infini, et les variables de V' ne sont pas libres dans τ et sont hors de portée de φ. En conséquence, $V' \cap \mathcal{L}(\varphi(E)) = \emptyset$.

La preuve du théorème 3.2 est par récurrence structurelle sur a. On donne un cas de base et trois cas de récurrence; les autres cas sont similaires.

Cas $a = x$. Puisque $\varphi(E) \vdash x : \tau$, on a $x \in \mathrm{Dom}(\varphi(E))$ et $\tau \leq \varphi(E)(x)$. Ceci entraîne $x \in \mathrm{Dom}(E)$. Écrivons $E(x) = \forall \alpha_1 \ldots \alpha_n . \tau_x$, avec les α_i choisies dans V' et hors de portée de φ'. Nous avons donc que $W(E, x, V)$ est défini et renvoie

$$\tau = \tau_x[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n] \qquad \text{et} \qquad \varphi = id \qquad \text{et} \qquad V' = V \setminus \{\beta_1 \ldots \beta_n\}$$

pour certaines variables $\beta_1 \ldots \beta_n \in V$. Par choix des α_i, nous avons $\varphi'(E(x)) = \forall \alpha_1 \ldots \alpha_n . \varphi'(\tau_x)$. On note ρ la substitution sur les α_i telle que $\tau' = \rho(\varphi'(\tau_x))$. On prend

$$\psi = \rho \circ \varphi' \circ [\beta_1 \leftarrow \alpha_1, \ldots, \beta_n \leftarrow \alpha_n].$$

On a $\psi(\tau) = \rho(\varphi'(\tau_x)) = \tau'$. D'autre part, toute variable $\alpha \notin V$ n'est ni une des α_i, ni une des β_i, d'où $\psi(\alpha) = \rho(\varphi'(\alpha)) = \varphi'(\alpha)$. C'est le résultat annoncé, puisque $\varphi = id$ ici.

Cas $a = \texttt{fun}\ x \to a_1$. La dérivation initiale se termine par

$$\frac{\varphi'(E) + \{x \leftarrow \tau_2'\} \vdash a_1 : \tau_1'}{\varphi'(E) \vdash \texttt{fun}\ x \to a_1 : \tau_2' \to \tau_1'}$$

Prenons α dans V, comme dans l'algorithme. On définit l'environnement E_1 et la substitution φ_1' par

$$E_1 = E + \{x \leftarrow \alpha\} \qquad \text{et} \qquad \varphi_1' = \varphi' + \{\alpha \leftarrow \tau_2'\}.$$

On a $\varphi_1'(E_1) = \varphi'(E) + \{x \leftarrow \tau_2'\}$. On applique l'hypothèse de récurrence à a_1, E_1, $V \setminus \{\alpha\}$, φ_1' et τ_2'. Il vient

$$(\tau_1, \varphi_1, V_1) = W(a_1, E_1, V \setminus \{\alpha\}) \qquad \text{et} \qquad \tau_1' = \psi_1(\tau_1) \qquad \text{et} \qquad \varphi_1' = \psi_1 \circ \varphi_1 \text{ hors de } V \setminus \{\alpha\}.$$

Il s'ensuit que $W(E, a, V)$ est bien défini. On prend alors $\psi = \psi_1$. Montrons que cette substitution ψ convient. On a:

$$\begin{array}{rcll}
\psi(\tau) & = & \psi(\varphi_1(\alpha) \to \tau_1) & \text{par définition de } \tau \text{ dans l'algorithme} \\
& = & \psi_1(\varphi_1(\alpha) \to \tau_1) & \text{par définition de } \psi \\
& = & \varphi_1'(\alpha) \to \psi_1(\tau_1) & \text{parce que } \alpha \notin V \setminus \{\alpha\} \\
& = & \tau_2' \to \psi_1(\tau_1) & \text{par construction de } \varphi_1' \\
& = & \tau_2' \to \tau_1' & \text{par construction de } \psi_1
\end{array}$$

D'autre part, pour toute variable γ hors de V,

$$\begin{array}{rcll} \psi(\varphi(\gamma)) & = & \psi_1(\varphi_1(\gamma)) & \text{par définition de } \psi \text{ et } \varphi \\ & = & \varphi'_1(\gamma) & \text{puisque } \gamma \notin V \\ & = & \varphi_1(\gamma) & \text{puisque } \gamma \notin V \text{ implique } \gamma \neq \alpha \end{array}$$

D'où le résultat annoncé.

Cas $a = a_1(a_2)$. La dérivation initiale est de la forme

$$\frac{\varphi'(E) \vdash a_1 : \tau'' \rightarrow \tau' \qquad \varphi'(E) \vdash a_2 : \tau''}{\varphi'(E) \vdash a_1(a_2) : \tau'}$$

On applique l'hypothèse de récurrence à a_1, E, V, $\tau'' \rightarrow \tau'$ et φ'. Il vient

$$(\tau_1, \varphi_1, V_1) = W(a_1, E, V) \qquad \text{et} \qquad \tau'' \rightarrow \tau' = \psi_1(\tau_1) \qquad \text{et} \qquad \varphi' = \psi_1 \circ \varphi_1 \text{ hors de } V.$$

En particulier, $\varphi'(E) = \psi_1(\varphi_1(E))$. On applique l'hypothèse de récurrence à a_2, $\varphi_1(E)$, V_1, τ et ψ_1. On a bien $\mathcal{L}(\varphi_1(E)) \cap V_1 = \emptyset$ par la remarque du début de la preuve. Il vient:

$$(\tau_2, \varphi_2, V_2) = W(a_2, \varphi_1(E), V_1) \qquad \text{et} \qquad \tau'' = \psi_2(\tau_2) \qquad \text{et} \qquad \psi_1 = \psi_2 \circ \varphi_2 \text{ hors de } V_1.$$

On a $\mathcal{L}(\tau_1) \cap V_1 = \emptyset$, d'où $\psi_1(\tau_1) = \psi_2(\varphi_2(\tau_1))$. Posons $\psi_3 = \psi_2 + \{\alpha \leftarrow \tau'\}$. (La variable α, choisie dans V_2, est hors de portée de ψ_2, et donc ψ_3 prolonge ψ_2.) On a:

$$\begin{array}{rclclcl} \psi_3(\varphi_2(\tau_1)) & = & \psi_2(\varphi_2(\tau_1)) & = & \psi_1(\tau_1) & = & \tau'' \rightarrow \tau' \\ \psi_3(\tau_2 \rightarrow \alpha) & = & \psi_2(\tau_2) \rightarrow \tau'' & = & \tau'' \rightarrow \tau' & & \end{array}$$

La substitution ψ_3 est donc un unificateur de $\varphi_2(\tau_1)$ et $\tau_2 \rightarrow \alpha$. L'unificateur principal de ces deux types, μ, existe donc, et $W(a_1(a_2), E, V)$ est bien défini. De plus, on a $\psi_3 = \psi_4 \circ \mu$ pour une certaine substitution ψ_4. On montre maintenant que $\psi = \psi_4$ convient. Avec les notations de l'algorithme, on a bien

$$\psi(\tau) = \psi_4(\mu(\alpha)))) = \psi_3(\alpha) = \tau',$$

d'une part, et d'autre part pour tout $\beta \notin V$ (et donc a fortiori $\beta \notin V_1$, $\beta \notin V_2$, $\beta \neq \alpha$):

$$\begin{array}{rcll} \psi(\varphi(\beta)) & = & \psi_4(\mu(\varphi_2(\varphi_1(\beta)))) & \text{par définition de } \varphi \\ & = & \psi_3(\varphi_2(\varphi_1(\beta))) & \text{par définition de } \psi_4 \\ & = & \psi_2(\varphi_2(\varphi_1(\beta))) & \text{parce que } \beta \neq \alpha \text{ et } \alpha \text{ hors de portée de } \varphi_1 \text{ et de } \varphi_2 \\ & = & \psi_1(\varphi_1(\beta)) & \text{parce que } \varphi_1(\beta) \notin V_1 \\ & = & \varphi'(\beta) & \text{parce que } \beta \notin V. \end{array}$$

C'est le résultat annoncé.

Cas $a = (\texttt{let } x = a_1 \texttt{ in } a_2)$. La dérivation initiale se termine par

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

On applique l'hypothèse de récurrence à a_1, E, V, τ' et φ'. Il vient

$$(\tau_1, \varphi_1, V_1) = W(a_1, E, V) \qquad \text{et} \qquad \tau' = \psi_1(\tau_1) \qquad \text{et} \qquad \varphi' = \psi_1 \circ \varphi_1 \text{ hors de } V.$$

En particulier, $\varphi'(E) = \psi_1(\varphi_1(E))$. On vérifie facilement que $\psi_1(\mathtt{Gen}(\tau_1, \varphi_1(E)))$ est plus général que $\mathtt{Gen}(\psi_1(\tau_1), \psi_1(\varphi_1(E)))$, c'est-à-dire que $\mathtt{Gen}(\tau', \varphi'(E))$. Puisqu'on peut prouver

$$\varphi'(E) + \{x \leftarrow \mathtt{Gen}(\tau', \varphi'(E))\} \vdash a_2 : \tau'',$$

le lemme 1.4 dit qu'on peut a fortiori prouver

$$\varphi'(E) + \{x \leftarrow \psi_1(\mathtt{Gen}(\tau_1, \varphi_1(E)))\} \vdash a_2 : \tau'',$$

c'est-à-dire

$$\psi_1(\varphi_1(E) + \{x \leftarrow \mathtt{Gen}(\tau_1, \varphi_1(E))\}) \vdash a_2 : \tau''.$$

On applique l'hypothèse de récurrence à a_2, dans l'environnement $\varphi_1(E) + \{x \leftarrow \mathtt{Gen}(\tau_1, \varphi_1(E))\}$, avec les variables V_1, le type τ'' et la substitution ψ_1. Il vient

$$(\tau_2, \varphi_2, V_2) = W(a_2,\ \varphi_1(E) + \{x \leftarrow \mathtt{Gen}(\tau_1, \varphi_1(E))\},\ V_1)$$

et $\tau'' = \psi_2(\tau_2)$ et $\psi_1 = \psi_2 \circ \varphi_2$ hors de V_1. L'algorithme prend $\tau = \tau_2$ et $\varphi = \varphi_2 \circ \varphi_1$ et $V' = V_2$. Montrons que $\psi = \psi_2$ convient. On a bien $\psi(\tau) = \tau''$. Et si $\alpha \notin V$, a fortiori $\alpha \notin V_1$, et donc:

$$\begin{array}{rcll} \psi(\varphi(\alpha)) & = & \psi_2(\varphi_2(\varphi_1(\alpha)))) & \text{par définition de } \varphi \\ & = & \psi_1(\varphi_1(\alpha)) & \text{parce que } \varphi_1(\alpha) \notin V_1\text{, puisque } \alpha \text{ hors de portée de } \varphi_1 \\ & = & \varphi'(\alpha) & \text{parce que } \alpha \notin V. \end{array}$$

D'où $\varphi' = \psi \circ \varphi$ hors de V, comme annoncé. □

3.3.3 Typage polymorphe de ML par expansion des `let`

Une autre approche du typage de ML avec polymorphisme provient de la remarque suivante. Supposons que $\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2$ est bien typée, de type τ. Nous avons donc:

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

Le lemme de substitution 2.2 du cours précédent nous dit qu'alors $E \vdash a_2[x \leftarrow a_1] : \tau$. Autrement dit, au lieu de généraliser le type de a_1 et de typer a_2 sous l'hypothèse $x : \mathtt{Gen}(\tau_1, E)$, 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:

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

La raison pour laquelle il faut quand même typer a_1, et non pas uniquement $a_2[x \leftarrow a_1]$, est que sinon nous laisserions passer des expressions de la forme

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

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 $\emptyset \vdash \texttt{fun}\ x \rightarrow x : \texttt{string} \rightarrow \texttt{string}$ (ou tout autre type à la place de `string`), d'une part, et de l'autre $\emptyset \vdash ((\texttt{fun}\ x \rightarrow x)\ 1, (\texttt{fun}\ x \rightarrow x)\ \texttt{true}) : \texttt{int} \times \texttt{bool}$.

Une fois (let-gen) remplacée par (let-subst), l'environnement de typage E ne contient plus que les identificateurs liés par `fun` (ceux liés par `let` ne sont jamais ajoutés à E). Donc, nous n'avons plus besoin de schémas de types: il suffit de dire que E 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:

$$E \vdash x : E(x) \quad \text{(var)}$$

Nous sommes donc ramenés à un système de types essentiellement monomorphe (le système de la section 1.3.2 plus la règle (let-subst)), auquel nous pouvons appliquer les techniques de la section 3.2: génération d'équations entre types simples et résolution par unification. En particulier, les équations à générer pour une construction `let` sont les suivantes:

$$\text{si } a = \texttt{let}\ x = b\ \texttt{in}\ c,\ C(a) = C(b) \cup C(c[x \leftarrow b]) \cup \{\alpha_a \stackrel{?}{=} \alpha_{c[x \leftarrow b]}\}$$

(Remarque: ceci n'est pas tout à fait exact, car nous avions supposé pour définir $C(a)$ que tous les identificateurs liés dans a avaient des noms différents, et ce n'est certainement pas le cas pour $a = c[x \leftarrow b]$. Par exemple, si $b = \texttt{fun}\ y \rightarrow y$ et $c = x\ x$, nous avons $a = (\texttt{fun}\ y \rightarrow y)\ (\texttt{fun}\ y \rightarrow y)$, dans laquelle y est liée deux fois. Pour être tout à fait correct, il faut considérer les expressions de mini-ML à α-conversion près, en s'autorisant à renommer les identificateurs liés par `let` et `fun` comme dans le λ-calcul. Puis il faut interpréter $c[x \leftarrow b]$ dans la formule ci-dessus comme "l'expression c dans laquelle chaque occurrence de x est remplacée par une copie de b dans laquelle on a renommé tous les identificateurs liés par de nouveaux identificateurs". Dans l'exemple, cela donnerait $c[x \leftarrow b] = (\texttt{fun}\ y' \rightarrow y')\ (\texttt{fun}\ y'' \rightarrow y'')$.)

Pour justifier complètement l'approche décrite ci-dessus, il faut encore montrer que le système de type ML monomorphe + la règle (let-subst) type exactement les mêmes programmes que ML polymorphe (tout programme bien typé dans l'un des systèmes est bien typé avec le même type dans l'autre). C'est une conséquence du théorème suivant:

Théorème 3.3 (Expansion du `let`) *Dans ML polymorphe,* $E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau$ *si et seulement s'il existe* τ_1 *tel que* $E \vdash a_1 : \tau_1$ *et* $E \vdash a_2[x \leftarrow a_1] : \tau$.

Démonstration: (esquissée). La partie "seulement si" est une 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 E: c'est-à-dire, $E \vdash a : \tau_0$, et de plus pour tout type τ' tel que $E \vdash a : \tau'$, il existe une substitution ψ telle que $\tau' = \psi(\tau_0)$ et $\mathrm{Dom}(\psi) \subseteq \mathcal{L}(\tau) \setminus \mathcal{L}(E)$. On prend alors $\sigma = \texttt{Gen}(\tau_0, E)$, et on montre par récurrence structurelle sur a que si $E + E' \vdash a_2[x \leftarrow a_1] : \tau$ avec $\mathrm{Dom}(E) \cap \mathrm{Dom}(E') = \emptyset$, alors $E + \{x : \sigma\} + E' \vdash a_2 : \tau$. $\square$

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: sur `let` $x = a_1$ `in` a_2, l'algorithme W type a_1 une seule fois, alors que l'algorithme avec (let-subst) re-type a_1 autant de fois que x est utilisé dans a_2. L'algorithme W est également préférable en pratique car il fournit des messages d'erreurs qui sont directement reliés au source du programme.

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 se reportera à Kanellakis, P.C., Mairson, H.G. and Mitchell, J.C., *Unification and ML type reconstruction.* In *Computational Logic: Essays in Honor of Alan Robinson*, ed. J.-L. Lassez and G.D. Plotkin, MIT Press, 1991, pages 444–478. `ftp://theory.stanford.edu/pub/jcm/papers/complexity-type-inf.dvi.Z`. Les principaux résultats 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-types 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 $O(2^n)$ si représenté par un arbre:

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

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.

Chapter 4

Extensions simples de mini-ML

Nous décrivons dans ce chapitre quelques traits du "vrai" langage ML qui s'ajoutent facilement à 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, il suffit de généraliser le constructeur de valeurs $(\_, \_)$ et le constructeur de types $\_ \times \_$ comme suit:

Expressions:	$a ::= \ldots \mid (a_1, \ldots, a_n)$
Valeurs:	$v ::= \ldots \mid (v_1, \ldots, v_n)$
Types:	$\tau ::= \ldots \mid \tau_1 \times \ldots \times \tau_n$
Contextes:	$\Gamma ::= \ldots \mid (\Gamma, a_2, \ldots, a_n) \mid (v_1, \Gamma, a_3, \ldots, a_n) \mid \ldots \mid (v_1, v_2, \ldots, v_{n-1}, \Gamma)$

On se donne alors une famille d'opérateurs de projection $\texttt{proj}_{i,n}$ $(1 \leq i \leq n)$ pour extraire la $i^{\text{ième}}$ composante d'un n-uplet. Leurs règles de typage et de réduction sont:

$$\frac{E \vdash a_1 : \tau_1 \qquad \ldots \qquad E \vdash a_n : \tau_n}{E \vdash (a_1, \ldots, a_n) : \tau_1 \times \ldots \times \tau_n}$$

$$TC(\texttt{proj}_{i,n}) = \forall \alpha_1 \ldots \alpha_n.\ (\alpha_1 \times \ldots \times \alpha_n) \to \alpha_i$$

$$\texttt{proj}_{i,n}(v_1, \ldots, v_n) \overset{\varepsilon}{\to} v_i$$

Les résultats des chapitres 2 et 3 s'étendent sans problèmes aux n-uplets. En particulier, les hypothèses H0, H1, H2 du chapitre 2 sont vérifiées.

Exercice 4.1 *(*)/(**) 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 $\texttt{proj}_{i,n}$ 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

$a \rightarrow a'$ dans ML_t, alors $T(a) \xrightarrow{*} T(a')$ dans ML_p. Réciproquement, est-il vrai que si $T(a) \rightarrow a''$, alors il existe a' tel que $a \rightarrow a'$ et $a'' = T(a')$?

4.2 Les types concrets

Les types concrets, aussi appelés types sommes ou types variants, jouent un rôle crucial pour définir de nouvelles structures de données, en particulier des structures récursives (listes, arbres, expressions, etc.). Un type concret se présente sous la forme d'un certain nombre de cas, appelés *constructeurs*, portant éventuellement un *argument*.

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

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

Le type 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
```

Un autre exemple est la syntaxe abstraite des expressions mini-ML, comme dans les exercices de programmation.

Les types concrets peuvent être paramétrés par un ou plusieurs types, ainsi les types `option` et `list`:

```
type 'a option = None | Some of 'a
type 'a list = Nil | Cons of 'a * 'a list
```

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

$$\texttt{type}\ (\alpha_1, \ldots, \alpha_n)\ t = C_1\ \texttt{of}\ \tau_1 \mid \ldots \mid C_p\ \texttt{of}\ \tau_p$$

$\alpha_1 \ldots \alpha_n$ sont les paramètres du type t. (Dans le cas fréquent $n = 0$, on écrit juste `type` $t = \ldots$.)
$C_1 \ldots C_m$ sont les constructeurs du type t.
$\tau_1 \ldots \tau_n$ sont les types des arguments des constructeurs.
(On convient de représenter les constructeurs constants comme des constructeurs `of unit`, où `unit` est un type muni d'une seule valeur notée `()`.)
On impose que $\mathcal{L}(\tau_i) \subseteq \{\alpha_1, \ldots, \alpha_n\}$ pour tout i.

La déclaration ci-dessus étend le langage de la manière suivante:

Opérateurs:	$op ::= \ldots \mid C_1 \mid \ldots \mid C_p \mid F_t$
Expressions de types:	$\tau ::= \ldots \mid (\tau_1, \ldots, \tau_n)\ t$
Valeurs:	$v ::= \ldots \mid C_1(v) \mid \ldots \mid C_p(v)$

Contextes: $\Gamma ::= \ldots \mid C_1(\Gamma) \mid \ldots \mid C_p(\Gamma)$

L'opérateur F_t est l'opérateur de filtrage associé au type t. Il permet de discriminer sur une valeur de type t suivant son constructeur de tête. Le filtrage qui s'écrit en ML

$$\texttt{match}\ a\ \texttt{with}\ C_1(x_1) \to a_1 \mid \ldots \mid C_p(x_p) \to a_p$$

est vu comme l'application suivante de l'opérateur F_t:

$$F_t(a, (\texttt{fun}\ x_1 \to a_1), \ldots, (\texttt{fun}\ x_p \to a_p))$$

La règle de réduction de F_t est:

$$F_t(C_k(v), v_1, \ldots, v_n) \quad \xrightarrow{\varepsilon} \quad v_k\ v \quad \text{si } C_k \text{ est le } k^{\text{ième}} \text{ constructeur du type } t$$

Les types des opérateurs C_k et F_t sont:

$$\begin{array}{rcl} C_k & : & \forall \alpha_1 \ldots \alpha_n.\ \tau_i \to (\alpha_1, \ldots, \alpha_n)\ t \\ F_t & : & \forall \alpha_1, \ldots, \alpha_n, \beta.\ ((\alpha_1, \ldots, \alpha_n)\ t \times (\tau_1 \to \beta) \times \ldots \times (\tau_n \to \beta)) \to \beta \end{array}$$

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

$$\begin{array}{rcl} \texttt{Entier} & : & \texttt{int} \to \texttt{num} \\ \texttt{Flottant} & : & \texttt{float} \to \texttt{num} \\ F_{\texttt{num}} & : & \forall \beta.\ (\texttt{num} \times (\texttt{int} \to \beta) \times (\texttt{float} \to \beta)) \to \beta \\ F_{\texttt{num}}(\texttt{Entier}(v), v_1, v_2) & \xrightarrow{\varepsilon} & v_1\ v \\ F_{\texttt{num}}(\texttt{Flottant}(v), v_1, v_2) & \xrightarrow{\varepsilon} & v_2\ v \end{array}$$

Pour le type α `list`, on a de même:

$$\begin{array}{rcl} \texttt{Nil} & : & \forall \alpha.\ \texttt{unit} \to \alpha\ \texttt{list} \\ \texttt{Cons} & : & \forall \alpha.\ (\alpha \times \alpha\ \texttt{list}) \to \alpha\ \texttt{list} \\ F_{\texttt{list}} & : & \forall \alpha, \beta.\ (\alpha\ \texttt{list} \times (\texttt{unit} \to \beta) \times ((\alpha \times \alpha\ \texttt{list}) \to \beta)) \to \beta \\ F_{\texttt{list}}(\texttt{Nil}(v), v_1, v_2) & \xrightarrow{\varepsilon} & v_1\ v \\ F_{\texttt{list}}(\texttt{Cons}(v), v_1, v_2) & \xrightarrow{\varepsilon} & v_2\ v \end{array}$$

Les résultats des chapitres 2 et 3 s'appliquent presque immédiatement à cette extension de mini-ML. En particulier, l'hypothèse H1 de la page 22 est satisfaite, ce qui garantit la préservation des types pendant la réduction.

Le seul point délicat est que nous avons maintenant des applications d'opérateurs qui sont des valeurs et donc ne se réduisent pas: les applications de constructeurs $C_k(v)$. Il faut donc modifier l'hypothèse H2 de la page 22 de la manière suivante:

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

Il est facile de voir que cette modification de H2 n'invalide pas la preuve du lemme de progression (proposition 2.6).

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

Exercice 4.3 *(*) Justifier la restriction $\mathcal{L}(\tau_i) \subseteq \{\alpha_1, \ldots, \alpha_n\}$ sur les types des arguments de constructeurs dans la déclaration* `type`. *(On montrera que le typage n'est pas sûr si cette restriction est levée.)*

Exercice 4.4 *(**) On considère le type concret suivant:*

```
type t = C of t → t
```

Quelles sont les 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 4.5 *(***) Le langage ML offre des mécanismes de filtrage plus puissants que l'opérateur de filtrage F_t 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 $F_{\texttt{list}}$ comme suit:

F_{list}`(l, (fun v → 0), (fun l1 →` F_{list}`(l1, (fun v' → 0), (fun l2 → 2))))`

On considère mini-ML étendu comme suit:

Expressions: $a ::= \ldots \mid (\texttt{match}\ a_1\ \texttt{with}\ p \rightarrow a_2 \mid \_ \rightarrow a_3)$
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 `match` a_1 `with` $p \rightarrow a_2 \mid \_ \rightarrow a_3$ *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 F_t.*

4.3 Les enregistrements déclarés

Les enregistrements (*records*) sont des n-uplets dont les composantes sont nommées (par des étiquettes) au lieu d'être repérées par position. En Caml, ils sont traités de manière similaire aux types concrets.

Exemple: le type des points dans l'espace s'écrit:

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

Le type des paires peut se définir comme suit:

```
type ('a, 'b) paire = { fst : 'a; snd : 'b }
```

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

$$\texttt{type}\ (\alpha_1, \ldots, \alpha_n)\ t = \{\textit{étiq}_1 : \tau_1; \ldots; \textit{étiq}_p : \tau_p\}$$

La déclaration ci-dessus étend le langage de la manière suivante:

Opérateurs:	$op ::= \ldots \mid M_t \mid .\textit{étiq}_1 \mid \ldots \mid .\textit{étiq}_p$
Expressions de types:	$\tau ::= \ldots \mid (\tau_1, \ldots, \tau_n)\ t$
Valeurs:	$v ::= \ldots \mid M_t(v)$
Contextes:	$\Gamma ::= \ldots \mid M_t(\Gamma)$

Les opérateurs $.\textit{étiq}$ sont les fonctions d'accès aux champs: l'expression Caml $a.\textit{étiq}$ est vue comme l'application d'opérateur $.\textit{étiq}(a)$.

L'opérateur M_t est l'opérateur de construction de l'enregistrement: l'expression Caml $\{\textit{étiq}_1 = a_1; \ldots; \textit{étiq}_p = a_p\}$ est vue comme l'application d'opérateur $M_t(a_1, \ldots, a_p)$ si t est le type enregistrement dont les étiquettes sont $\textit{étiq}_1, \ldots, \textit{étiq}_p$.

Les règles de typage et de réduction pour ces opérateurs sont:

$$\begin{array}{rcl} M_t & : & \forall \alpha_1 \ldots \alpha_n.\ \tau_1 \times \tau_2 \times \ldots \times \tau_p \rightarrow (\alpha_1, \ldots, \alpha_n)\ t \\ .\textit{étiq}_k & : & \forall \alpha_1 \ldots \alpha_n.\ (\alpha_1, \ldots, \alpha_n)\ t \rightarrow \tau_k \\ .\textit{étiq}_k(M_t(v_1, \ldots, v_p)) & \stackrel{\varepsilon}{\rightarrow} & v_k \end{array}$$

Remarque: en fait, l'ordre des étiquettes dans l'expression Caml $\{\textit{étiq}_1 = a_1; \ldots; \textit{étiq}_p = a_p\}$ n'est pas forcément le même que dans la déclaration du type t. Pour refléter ce fait, il faut déterminer le type t et la permutation σ sur $\{1, \ldots, p\}$ tels que t est déclaré comme $\{\textit{étiq}_{\sigma(1)} : \tau_1; \ldots; \textit{étiq}_{\sigma(p)} : \tau_p\}$ et traduire l'expression Caml ci-dessus en $M_t(a_{\sigma^{-1}(1)}, \ldots, a_{\sigma^{-1}(p)})$.

Remarque: ce traitement des enregistrements fait qu'une étiquette donnée ne peut appartenir à plusieurs types enregistrement simultanément. Nous verrons au chapitre 6 un traitement beaucoup plus souple des enregistrements.

4.4 Les contraintes de types

En ML, l'expression $(a : \tau)$ s'évalue comme a, mais force a à avoir le type τ. En mini-ML, on peut voir $(a : \tau)$ comme une application d'opérateur $\mathtt{contr}_\tau(a)$. Les opérateurs $\mathtt{contr}_\tau$ (un par type τ) se typent et s'évaluent comme suit:

$$\begin{array}{rcl} \mathtt{contr}_\tau & : & \forall \alpha_1 \ldots \alpha_n.\ \tau \rightarrow \tau \text{ si } \alpha_1 \ldots \alpha_n = \mathcal{L}(\tau) \\ \mathtt{contr}_\tau(v) & \stackrel{\varepsilon}{\rightarrow} & v \end{array}$$

Si τ contient des variables de types, cette présentation ne force pas a à avoir exactement le type τ, mais assure que le type de a est une instance $\varphi(\tau)$. Ainsi, $(1 : \alpha)$ est correct et a le type `int`. Ce comportement est celui adopté en Caml, mais Standard ML par exemple exige que a ait exactement le type τ. Ce dernier comportement ne peut s'exprimer en terme d'applications d'opérateurs; il faut une règle de typage spéciale.

4.5 Les références et autres structures de données mutables

Voir le chapitre 5.

4.6 Les exceptions

Voir le chapitre 5.

Chapter 5

La programmation impérative

Dans ce chapitre, nous étendons mini-ML avec plusieurs traits caractéristiques des langages impératifs: la modification en place de variables et de structures de données, et les exceptions.

5.1 Les références

Le premier trait impératif que nous considérons est la possibilité de modifier en place, par affectation, une variables ou une structure de données. Pour ce faire, nous ajoutons à mini-ML la notion de *référence*. Une référence est une cellule d'indirection modifiable en place; on peut aussi la voir comme un tableau à un seul élément.

On crée une référence par la construction `ref`(a), qui renvoie une nouvelle référence contenant initialement la valeur de a. Une référence a un contenu courant, qui s'obtient par l'opération de déréférencement $!a$. Enfin, on peut changer le contenu de la référence a_1 en y stockant a_2 par l'opération d'affectation $a_1 := a_2$.

Exemple: l'expression suivante s'évalue en 4

```
let r = ref(3) in let x = r := !r + 1 in !r
```

En effet, le contenu de `r` est initialement 3. On calcule ensuite `!r + 1`, c'est-à-dire `3 + 1`, et on stocke cette valeur dans `r`. Enfin, on renvoie le contenu courant de `r`, qui est 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 construction $a; b$ évalue a, puis b, et renvoie la valeur de b. On peut la voir comme une abrévation pour `let` $x = a$ `in` b où x n'est pas libre dans b.)

Une structure de données (liste, arbres, etc) contenant des références modélise une structure de donnée mutable (modifiable en place). Par exemple, on peut représenter les tableaux (mutables) par des listes (immuables) dont chaque élément est une référence (mutable).

```
type α tableau = α ref list
let rec nième l n = match l with Cons(x, l') → if n = 0 then x else nième l' (n-1)
let lire_élément t i = !(nième t i)
let écrire_élément t i v = (nième t i) := v
```

De même, une liste simplement chaînée dont on peut modifier le chaînage en place 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(x, r) → concat !r l2
```

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

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

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 applicatif (aussi appelé purement fonctionnel), la valeur d'une expression ne dépend que de la valeur de ses variables libres. Lorsque ces dernières ont des valeurs connues, l'évaluation des sous-expressions de l'expression peut s'effectuer indépendamment les unes des autres. Ce n'est plus vrai dans un langage impératif: l'évaluation d'une sous-expression peut modifier des références, et donc affecter les évaluations d'autres sous-expressions mentionnant ces mêmes références.

Une seconde difficulté introduite par les références est la notion de partage de références, qui entre en conflit avec le dé-partage effectué par l'étape de substitution dans la β-réduction classique. Prenons comme exemple le terme

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

Les deux occurrences de `r` correspondent à la même référence, allouée une fois pour toute par le `ref 1` en partie gauche du `let`. Si nous effectuons naïvement une étape de β-réduction sur cette expression, nous obtenons:

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

Ce terme a un comportement tout à fait différent du premier: il alloue deux références distinctes initialisées à 1, modifie la première, et lit la seconde. Il faut donc trouver une sémantique plus fine que la simple β-réduction sur les termes du langage source.

Pour étendre aux références la sémantique à réduction de la section 2.2, nous formalisons la notion d'adresse mémoire et d'état mémoire. On se donne un ensemble infini d'adresses mémoires (*locations* en anglais), notées ℓ. Un état mémoire (*store*) s est une fonction partielle des adresses mémoires dans les valeurs. Les expressions et leurs valeurs possibles sont:

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

En particulier, une référence s'évalue en son adresse mémoire ℓ associée. Les programmes initiaux ne contiennent pas d'adresses mémoire ℓ; ces dernières apparaissent lorsqu'on évalue une création de référence $\mathtt{ref}(a)$.

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:

$$
\begin{array}{rcl}
(\mathtt{fun}\ x \to a)\ v\ /\ s & \xrightarrow{\varepsilon} & a\{x \leftarrow v\}\ /\ s \qquad (\beta_{fun}) \\
(\mathtt{let}\ x = v\ \mathtt{in}\ a)\ /\ s & \xrightarrow{\varepsilon} & a\{x \leftarrow v\}\ /\ s \qquad (\beta_{let}) \\
\mathtt{ref}(v)\ /\ s & \xrightarrow{\varepsilon} & \ell\ /\ s + \{\ell \leftarrow v\} \quad \text{si } \ell \notin \mathrm{Dom}(s) \qquad (\delta_{ref}) \\
!\ell\ /\ s & \xrightarrow{\varepsilon} & s(\ell)\ /\ s \quad \text{si } \ell \in \mathrm{Dom}(s) \qquad (\delta_{deref}) \\
:=(\ell, v)\ /\ s & \xrightarrow{\varepsilon} & (\,)\ /\ s + \{\ell \leftarrow v\} \quad \text{si } \ell \in \mathrm{Dom}(s) \qquad (\delta_{aff})
\end{array}
$$

$$
\frac{a_1\ /\ s_1 \xrightarrow{\varepsilon} a_2\ /\ s_2}{\Gamma(a_1)\ /\ s_1 \to \Gamma(a_2)\ /\ s_2}\ \text{(contexte)}
$$

Pour les opérateurs qui proviennent de mini-ML "pur" (arithmétique, `fst`, `snd`, `fix`, ...), il suffit de transformer leurs δ-règles $a_1 \xrightarrow{\varepsilon} a_2$ en $a_1\ /\ s \xrightarrow{\varepsilon} a_2\ /\ s$. En effet, la réduction de ces opérateurs ne dépend pas de l'état mémoire, et ne modifie pas non plus l'état mémoire. On a ainsi, par exemple:

$$
\begin{array}{rcl}
+\ (n_1, n_2)\ /\ s & \xrightarrow{\varepsilon} & n\ /\ s \quad \text{si } n_1, n_2 \text{ entiers et } n = n_1 + n_2 \qquad (\delta_+) \\
\mathtt{fst}\ (v_1, v_2)\ /\ s & \xrightarrow{\varepsilon} & v_1\ /\ s \qquad (\delta_{fst}) \\
\mathtt{snd}\ (v_1, v_2)\ /\ s & \xrightarrow{\varepsilon} & v_2\ /\ s \qquad (\delta_{snd})
\end{array}
$$

Exemple: on a la séquence de réductions suivante:

$$\begin{array}{rl}
& \mathtt{let}\ r = \mathtt{ref}(3)\ \mathtt{in}\ \mathtt{let}\ x = r := {!r} + 1\ \mathtt{in}\ {!r} \,/\, \emptyset \\
\rightarrow & \mathtt{let}\ r = \ell\ \mathtt{in}\ \mathtt{let}\ x = r := {!r} + 1\ \mathtt{in}\ {!r} \,/\, \{\ell \leftarrow 3\} \\
\rightarrow & \mathtt{let}\ x = \ell := {!\ell} + 1\ \mathtt{in}\ {!\ell} \,/\, \{\ell \leftarrow 3\} \\
\rightarrow & \mathtt{let}\ x = \ell := 3 + 1\ \mathtt{in}\ {!\ell} \,/\, \{\ell \leftarrow 3\} \\
\rightarrow & \mathtt{let}\ x = \ell := 4\ \mathtt{in}\ {!\ell} \,/\, \{\ell \leftarrow 3\} \\
\rightarrow & \mathtt{let}\ x = (\,)\ \mathtt{in}\ {!\ell} \,/\, \{\ell \leftarrow 4\} \\
\rightarrow & {!\ell} \,/\, \{\ell \leftarrow 4\} \\
\rightarrow & 4
\end{array}$$

Exercice 5.3 *(*) Donner une sémantique opérationnelle "grands pas" (dans le style de la section 1.2) pour mini-ML + références. (Indication: la relation d'évaluation est de la forme* $a \,/\, s \stackrel{v}{\rightarrow} v \,/\, s'$*.)*

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

5.3 Typage des références

Pour typer les références, nous suivons la même approche qu'au chapitre 4: on étend l'algèbre des types par les type τ `ref` (le type des références dont le contenu est de type τ), et on donne les types "évidents" aux opérateurs `ref`, ! et := ainsi qu'à la constante ( )

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

$$\begin{array}{rcl}
\mathtt{ref} & : & \forall\alpha.\ \alpha \rightarrow \alpha\ \mathtt{ref} \\
! & : & \forall\alpha.\ \alpha\ \mathtt{ref} \rightarrow \alpha \\
:= & : & \forall\alpha.\ \alpha\ \mathtt{ref} \times \alpha \rightarrow \mathtt{unit} \\
(\,) & : & \mathtt{unit}
\end{array}$$

Malheureusement, ce typage "évident" n'est pas sûr en conjonction avec le polymorphisme de ML (il est sûr en conjonction avec le typage monomorphe de la section 1.3.2, cependant). Exemple:

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

`r` reçoit le type polymorphe $\forall\alpha.\ (\alpha \rightarrow \alpha)$ `ref`. L'affectation `r := (fun x → +(x,1))` est donc bien typée (on utilise `r` avec l'instance (`int` → `int`) `ref`), ainsi que l'application `(!r) true` (on utilise `r` avec l'instance (`bool` → `bool`) `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 E associe des types aux adresses mémoire. Ces types peuvent être des schémas ou des types simples.

Si E 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:

$$\frac{\tau \leq E(\ell)}{E \vdash \ell : \tau} \text{ (loc-inst)}$$

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

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

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

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 $E(\ell)$. Le typage des opérations ! et := redevient sûr. En revanche, la généralisation des types au moment du `let` pose problème: le typage n'est pas préservé par la réduction δ_{ref}. Considérons

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

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

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

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 qui sont *non-expansives*, c'est-à-dire dont la forme même garantit que leur évaluation ne crée pas de références:

$$\frac{E \vdash a_1 : \tau' \qquad \sigma = \begin{cases} \mathtt{Gen}(\tau', E) & \text{si } a_1 \text{ non expansive;} \\ \tau' & \text{sinon} \end{cases} \qquad E + \{x : \sigma\} \vdash a_2 : \tau}{E \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau}$$

Les expressions non-expansives a_{ne} sont décrites par la grammaire suivante:

Expressions non-expansives:

$a_{ne} ::= x$	identificateurs
$\mid c$	constantes
$\mid op$	opérateurs
$\mid \mathtt{fun}\ x \to a$	fonctions
$\mid (a'_{ne}, a''_{ne})$	paires d'expressions non-expansives
$\mid op(a_{ne})$	si $op \neq \mathtt{ref}$
$\mid \mathtt{let}\ x = a'_{ne}\ \mathtt{in}\ a''_{ne}$	liaison `let`

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, car `ref(fun x → x)` n'est pas non-expansive, et donc `r` reçoit le type simple $(\alpha \to \alpha)$ `ref` avec α non généralisé. α est unifié avec `int` lorsqu'on type la seconde ligne de l'exemple, et la troisième ligne `(!r) true` est donc mal typée.

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, et donc `f` reçoit le type simple $\alpha \to$ `int` où α est non généralisée, et donc `f` ne peut être utilisée de manière polymorphe par la suite. Pour faire "passer" cet exemple, le programmeur doit manuellement faire une étape d'eta-expansion sur `f`:

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

De manière générale, une étape d'eta-expansion permet de rendre non-expansive (et donc généralisable) toute définition de fonction résultant d'un calcul (comme l'application `k 1` ci-dessus).

La raison pour laquelle toute application de fonction est considérée comme potentiellement expansive est qu'elle peut cacher (de manière plus ou moins évidente) la 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 $\forall\alpha.\ \alpha \rightarrow (\alpha \rightarrow \mathtt{unit}) \times (\mathtt{unit} \rightarrow \alpha)$. 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.

Remarque: les expressions qui sont des valeurs sont également non-expansives. Une présentation plus simple mais plus restrictive de la règle de typage du `let` est donc:

$$\frac{E \vdash a_1 : \tau' \qquad \sigma = \begin{cases} \mathtt{Gen}(\tau', E) & \text{si } a_1 \text{ est une valeur;} \\ \tau' & \text{sinon} \end{cases} \qquad E + \{x : \sigma\} \vdash a_2 : \tau}{E \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau}$$

Cette approche s'appelle le polymorphisme restreint aux valeurs (*value restriction on polymorphism*) dans la littérature. Nous préférons introduire une notion distincte d'expression non expansive, car cela permet de typer un peu plus de programmes, comme par exemple:

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

`fst((fun x → x), 1)` est non-expansive, mais n'est pas une valeur.

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

Exercice 5.4 *(**) Trouvez un exemple où l'eta-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 $E \vdash a : \tau$ (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 E. 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 E, et on écrit $E \vdash s$, si $\ell \in \mathrm{Dom}(s) \Leftrightarrow \ell \in \mathrm{Dom}(E)$ et pour toute adresse $\ell \in \mathrm{Dom}(s)$, il existe τ tel que $E(\ell) = \tau$ `ref` et $E \vdash s(\ell) : \tau$.

Définition: on dit qu'un environnement E' étend un environnement E si E' est E auquel on a ajouté zéro, une ou plusieurs hypothèses de typage d'adresses mémoire $\ell : \tau$ (avec $\ell \notin \mathrm{Dom}(E)$).

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 E et tout type τ,

- Si a_1 est non-expansive: a_2 est non-expansive, et de plus $E \vdash a_1 : \tau$ et $E \vdash s_1$ implique $E \vdash a_2 : \tau$ et $E \vdash s_2$.
- Si a_1 est expansive: $E \vdash a_1 : \tau$ et $E \vdash s_1$ implique qu'il existe E' étendant E tel que $E' \vdash a_2 : \tau$ et $E' \vdash s_2$.

Remarquons que cette notion d'être "moins typable que" étend celle de la section 2.3.1, comme le montre la proposition suivante.

Proposition 5.1 ($\sqsubseteq$ sans changement d'état mémoire) *Supposons $a_1 \sqsubseteq a_2$ au sens de la section 2.3.1. Supposons de plus que si a_1 est non-expansive, alors a_2 est non-expansive. Alors pour tout s nous avons $a_1 \,/\, s \sqsubseteq a_2 \,/\, s$.*

Démonstration: soient E et τ tels que $E \vdash a_1 : \tau$ et $E \vdash s$. Par hypothèse $a_1 \sqsubseteq a_2$, nous avons $E \vdash a_2 : \tau$. Si a_1 est non-expansive, a_2 l'est aussi, et donc $a_1 \,/\, s \sqsubseteq a_2 \,/\, s$ par définition de $\sqsubseteq$. Si a_1 est expansive, nous prenons $E' = E$ et nous avons bien $E' \vdash a_2 : \tau$ et $E' \vdash s$; donc, $a_1 \,/\, s \sqsubseteq a_2 \,/\, s$ par définition de $\sqsubseteq$. □

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

Le théorème de sûreté se prouve par une séquence de lemmes analogues à ceux utilisés au chapitre 2.

Proposition 5.2 (Préservation du typage par réduction de tête) *Si $a_1 \,/\, s_1 \xrightarrow{\varepsilon} a_2 \,/\, s_2$, alors $a_1 \,/\, s_1 \sqsubseteq a_2 \,/\, s_2$.*

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

Cas des règles β_{fun}, β_{let}, et 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 $a_1 \xrightarrow{\varepsilon} a_2$ est une réduction de mini-ML sans les références. Appliquant la proposition 2.3, on a donc $a_1 \sqsubseteq a_2$ (au sens de la section 2.3.1). De plus, on vérifie facilement par inspection des règles que si a_1 est non-expansive, alors a_2 l'est aussi:

$(\mathtt{fun}\ x \rightarrow a)\ v$	(expansive)	$\xrightarrow{\varepsilon}$	$a[x \leftarrow v]$	(indifférent)	(β_{fun})
$\mathtt{let}\ x = v\ \mathtt{in}\ a_{ne}$	(non-expansive)	$\xrightarrow{\varepsilon}$	$a_{ne}[x \leftarrow v]$	(non-expansive)	(β_{let})
$\mathtt{let}\ x = v\ \mathtt{in}\ a_e$	(expansive)	$\xrightarrow{\varepsilon}$	$a_e[x \leftarrow v]$	(indifférent)	(β_{let})
$+\ (n_1, n_2)$	(non-expansive)	$\xrightarrow{\varepsilon}$	n	(non-expansive)	(δ_+)
$\mathtt{fst}\ (v_1, v_2)$	(non-expansive)	$\xrightarrow{\varepsilon}$	v_1	(non-expansive)	(δ_{fst})
$\mathtt{snd}\ (v_1, v_2)$	(non-expansive)	$\xrightarrow{\varepsilon}$	v_2	(non-expansive)	(δ_{snd})
$\mathtt{fix}\ (\mathtt{fun}\ x \rightarrow a)$	(expansive)	$\xrightarrow{\varepsilon}$	$a\{x \leftarrow \mathtt{fix}\ (\mathtt{fun}\ x \rightarrow a)\}$	(indifférent)	(δ_{fix})

(Pour le second cas, nous utilisons le fait que l'ensemble des expressions non-expansives est clos par substitution de variables par des expressions non-expansives, et a fortiori par des valeurs.)

D'où $a_1 \,/\, s_1 \sqsubseteq a_2 \,/\, s_1$ par la proposition 5.1, et le résultat annoncé car $s_2 = s_1$.

Cas de la règle δ_{ref}: on a $a_1 = \mathtt{ref}(v)$ et $a_2 = \ell$ et $s_2 = s_1 + \{\ell \leftarrow v\}$ avec $\ell \notin \mathrm{Dom}(E)$. Remarquons que a_1 est expansive. Nous avons:

$$\frac{\dfrac{\tau_1 \rightarrow \tau_1\ \mathtt{ref} \leq TC(\mathtt{ref})}{E \vdash \mathtt{ref} : \tau_1 \rightarrow \tau_1\ \mathtt{ref}} \qquad E \vdash v : \tau_1}{E \vdash \mathtt{ref}(v) : \tau_1\ \mathtt{ref}}$$

D'où $E \vdash v : \tau_1$. Prenons $E' = E + \{\ell \leftarrow \tau_1\ \mathtt{ref}\}$. Puisque $E \vdash s_1$, on a bien $E' \vdash s_2$. De plus, $E' \vdash \ell : \tau_1\ \mathtt{ref}$ par la règle de typage des adresses mémoire.

Cas de la règle δ_{deref}: on a $a_1 = !\ell$ et $a_2 = s_1(\ell)$ et $\ell \in \mathrm{Dom}(s_1)$ et $s_2 = s_1$. Par hypothèse $E \vdash a_1 : \tau$, nous avons

$$\frac{\dfrac{\tau\ \mathtt{ref} \rightarrow \tau \leq TC(!)}{E \vdash\ ! : \tau\ \mathtt{ref} \rightarrow \tau} \qquad E \vdash \ell : \tau\ \mathtt{ref}}{E \vdash\ !\ell : \tau}$$

Comme $E \vdash s_1$ et $E \vdash \ell : \tau\ \texttt{ref}$, il s'ensuit que $E(\ell) = \tau\ \texttt{ref}$, et donc $E \vdash s_1(\ell) : \tau$. On a bien $E \vdash a_2 : \tau$ et $E \vdash s_2$ comme désiré (car a_1 est non-expansive).

Cas de la règle δ_{aff}: on a $a_1 =:= (\ell, v)$ et $a_2 = (\,)$ et $\ell \in \mathrm{Dom}(s_1)$ et $s_2 = s_1 + \{\ell \leftarrow v\}$. Puisque a_1 est bien typée dans E, nous avons la dérivation suivante:

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

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

Proposition 5.3 (Croissance de $\sqsubseteq$) *Pour tout contexte d'évaluation* Γ, $a_1 / s_1 \sqsubseteq a_2 / s_2$ *implique* $\Gamma(a_1) \,/\, s_1 \sqsubseteq \Gamma(a_2) \,/\, s_2$.

C'est ce lemme qui ne serait pas vrai sans la restriction de la généralisation (prendre $\Gamma = \texttt{let}\ r = [\,]\ \texttt{in}\ a$ et $a_1 = \texttt{ref}(\texttt{fun}\ x \to x)$ et $a_2 = \ell$).

Démonstration: par récurrence structurelle sur Γ. Le seul cas intéressant est $\Gamma = \texttt{let}\ x = \Gamma'\ \texttt{in}\ a$. Soient donc E et τ tels que $E \vdash \Gamma(a_1) : \tau$ et $E \vdash s_1$.

Si $\Gamma'(a_1)$ est non-expansive: on a la dérivation de typage suivante:

$$\dfrac{E \vdash \Gamma'(a_1) : \tau_1 \qquad \sigma = \texttt{Gen}(\tau_1, E) \qquad E + \{x : \sigma\} \vdash a : \tau}{E \vdash \texttt{let}\ x = \Gamma'(a_1)\ \texttt{in}\ a : \tau}$$

Appliquant l'hypothèse de récurrence à $\Gamma'(a_1)$, il vient $\Gamma'(a_1) \,/\, s_1 \sqsubseteq \Gamma'(a_2) \,/\, s_2$. Comme $\Gamma'(a_1)$ est non-expansive, cela signifie que $E \vdash \Gamma'(a_2) : \tau_1$ et $E \vdash s_2$, et de plus $\Gamma'(a_2)$ est non-expansive. Par conséquent, on peut construire la dérivation suivante:

$$\dfrac{E \vdash \Gamma'(a_2) : \tau_1 \qquad \sigma = \texttt{Gen}(\tau_1, E) \qquad E + \{x : \sigma\} \vdash a : \tau}{E \vdash \texttt{let}\ x = \Gamma'(a_2)\ \texttt{in}\ a : \tau}$$

D'où $E \vdash \Gamma(a_2) : \tau$ et $E \vdash s_2$, ce qui entraîne le résultat désiré $\Gamma(a_1) \,/\, s_1 \sqsubseteq \Gamma(a_2) \,/\, s_2$.

Si $\Gamma'(a_1)$ est expansive: on a la dérivation de typage suivante:

$$\dfrac{E \vdash \Gamma'(a_1) : \tau_1 \qquad E + \{x : \tau_1\} \vdash a : \tau}{E \vdash \texttt{let}\ x = \Gamma'(a_1)\ \texttt{in}\ a : \tau}$$

Appliquant l'hypothèse de récurrence à $\Gamma'(a_1)$, il vient $\Gamma'(a_1) \,/\, s_1 \sqsubseteq \Gamma'(a_2) \,/\, s_2$. Comme $\Gamma'(a_1)$ est expansive, cela signifie qu'il existe E' étendant E tel que $E' \vdash \Gamma'(a_2) : \tau_1$ et $E' \vdash s_2$. Par le lemme 1.3, $E + \{x : \tau_1\} \vdash a : \tau$ implique $E' + \{x : \tau_1\} \vdash a : \tau$. On peut donc construire la dérivation suivante:

$$\dfrac{E' \vdash \Gamma'(a_2) : \tau_1 \qquad E' + \{x : \tau_1\} \vdash a : \tau}{E' \vdash \texttt{let}\ x = \Gamma'(a_2)\ \texttt{in}\ a : \tau}$$

D'où $E' \vdash \Gamma(a_2) : \tau$ et $E' \vdash s_2$, ce qui établit que $\Gamma(a_1) \,/\, s_1 \sqsubseteq \Gamma(a_2) \,/\, s_2$. □

Proposition 5.4 (Préservation du typage par réduction) *Si $a_1 \,/\, s_1 \rightarrow a_2 \,/\, s_2$, alors $a_1 \,/\, s_1 \sqsubseteq a_2 \,/\, s_2$.*

Démonstration: conséquence des lemmes 5.2 et 5.3. $\square$

Proposition 5.5 (Forme des valeurs selon leur type) *Soit E un environnement qui ne lie aucun identificateur x mais seulement des adresses ℓ. Supposons $E \vdash v : \tau$ et $E \vdash s$.*

1. *Si $\tau = \tau_1 \rightarrow \tau_2$, alors ou bien v est de la forme $\texttt{fun}\ x \rightarrow a$, 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 qui peuvent s'appliquer suivant la forme de τ. $\square$

Proposition 5.6 (Lemme de progression) *Soit E un environnement qui ne lie aucun identificateur x mais seulement des adresses ℓ. Supposons $E \vdash a : \tau$ et $E \vdash s$. Alors, ou bien a est une valeur, ou bien il existe a' et s' tels que $a \,/\, s \rightarrow a' \,/\, s'$.*

Démonstration: semblable à celle du lemme 2.6. $\square$

5.6 Autres approches

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.

5.6.1 Les références monomorphes

L'approche suivie dans les premières version de Caml est de typer spécialement l'opérateur `ref` de manière à ce qu'il ne soit employé que de manière monomorphe:

$$\frac{\tau \text{ ne contient pas de variables}}{E \vdash \texttt{ref} : \tau \rightarrow \tau\ \texttt{ref}}$$

Ainsi, les adresses mémoire ℓ reçoivent toujours des types sans variables, et on peut lever les restrictions sur la généralisation au moment du `let`.

Cette approche présente deux problèmes. Tout d'abord, il est impossible d'écrire des fonctions polymorphes qui allouent des structures mutables, soit pour les renvoyer en résultat, soit même pour les utiliser en interne comme des temporaires. Par exemple, la fonction Caml qui transpose une matrice

```
let transpose m dimx dimy =
  let tm = Array.make_matrix dimy dimx in
  (* remplir tm *); tm
```

ne peut recevoir son type naturel $\forall\alpha.\ \alpha$ `array array` $\rightarrow \alpha$ `array array`, et doit être spécialisée à un type sans variable particulier, comme `float array array` $\rightarrow$ `float array array`. Pour transposer des matrices d'entiers, il faudra alors réécrire une autre fonction `transpose`.

L'autre problème posé par cette approche est que le système de types n'admet plus de types principaux. Par exemple, `fun` $x \rightarrow$ `ref`(x) admet tous les types $\tau \rightarrow \tau$ `ref` pour τ sans variables, mais aucun de ces types ne résume tous les autres. En particulier, leur borne supérieure $\alpha \rightarrow \alpha$ `ref` où α est une variable de type n'est pas un type correct pour cette fonction.

5.6.2 Les variables faibles de Standard ML

L'approche suivie dans Standard ML 90 est d'avoir 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:	$\tau ::= \alpha_a \mid \alpha_i \mid T \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1\ \texttt{ref}$
Types impératifs:	$\bar{\tau} ::= \alpha_i \mid T \mid \bar{\tau}_1 \rightarrow \bar{\tau}_2 \mid \bar{\tau}_1 \times \bar{\tau}_2 \mid \bar{\tau}_1\ \texttt{ref}$

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

Toutes les constantes et les opérateurs ont des schémas de types "applicatifs" (où les variables quantifiées sont applicatives, leur permettant d'être instanciées par n'importe quel type ensuite):

$$\begin{array}{rcl} \texttt{fst} & : & \forall\alpha_a,\beta_a.\ \alpha_a \times \beta_a \rightarrow \alpha_a \\ \texttt{snd} & : & \forall\alpha_a,\beta_a.\ \alpha_a \times \beta_a \rightarrow \beta_a \\ ! & : & \forall\alpha_a.\ \alpha_a\ \texttt{ref} \rightarrow \alpha_a \\ := & : & \forall\alpha_a.\ \alpha_a\ \texttt{ref} \times \alpha_a \rightarrow \texttt{unit} \end{array}$$

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):

$$\frac{E \vdash a_1 : \tau' \qquad \sigma = \texttt{GenAppl}(\tau', E) \qquad E + \{x : \sigma\} \vdash a_2 : \tau}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau}$$

avec $\texttt{GenAppl}(\tau, E) = \forall\alpha_{a,1}\ldots\alpha_{a,n}.\ \tau$ où $\{\alpha_{a,1},\ldots,\alpha_{a,n}\} = \mathcal{L}_a(\tau)\backslash\mathcal{L}_a(E)$ (les variables applicatives libres dans τ mais pas dans E).

En fait, SML 90 va plus loin et permet la généralisation des variables impératives lorsque l'expression liée par `let` est non-expansive.

$$\frac{E \vdash a_1 : \tau' \qquad \sigma = \begin{cases} \texttt{Gen}(\tau', E) & \text{si } a_1 \text{ non expansive;} \\ \texttt{GenAppl}(\tau', E) & \text{sinon} \end{cases} \qquad E + \{x : \sigma\} \vdash a_2 : \tau}{E \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau}$$

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.

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

5.6.3 Systèmes d'effets et de régions

Pour aller plus loin. Voir *The type and effect discipline*, Jean-Pierre Talpin et Pierre Jouvelot, *Information and Computation* 111(2), 1994.

5.6.4 Variables dangereuses et typage des fermetures

Pour aller plus loin. Voir *Polymorphic type inference and assignment*, Xavier Leroy et Pierre Weis, *Principles of Programming Languages* 1991.

5.7 Les exceptions

Les exceptions sont un mécanisme très souple pour signaler les erreurs dans une fonction, propager cette erreur à travers les fonctions appelantes, et se brancher sur le code de traitement de l'erreur. On peut aussi s'en servir comme d'une structure générale de contrôle, p.ex. pour programmer des sorties de boucles.

Expressions: $a ::= \ldots \mid \texttt{try } a_1 \texttt{ with } x \to a_2$

Opérations: $op ::= \ldots \mid \texttt{raise}$

Les exceptions se présentent comme l'opérateur `raise` qui interrompt l'évaluation courante et lève une exception, et comme la construction `try` a_1 `with` $x \to a_2$ qui évalue a_1 et renvoie sa valeur si aucune exception n'est levée par a_1, ou bien évalue a_2 et renvoie sa valeur si l'évaluation de a_1 déclenche une exception. Les exceptions portent un "argument" (p.ex. la cause de l'erreur) qui est donné en argument à `raise` et lié ensuite à x dans a_2 par la construction `try` a_1 `with` $x \to a_2$.

Exemple: `try 1 + (raise "Hello") with x` $\rightarrow$ `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`:

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

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

$$\Delta(\texttt{raise}(v)) \quad \rightarrow \quad \texttt{raise}(v) \text{ si } \Delta \neq [\,]$$

Dans cette dernière règle, Δ est un contexte de réduction ne contenant pas de `try...with`:

Contextes de réduction:

Γ ::= $[\,] \mid \Gamma\ a \mid v\ \Gamma \mid \texttt{let } x = \Gamma \texttt{ in } a \mid (\Gamma, a) \mid (v, \Gamma) \mid \texttt{try } \Gamma \texttt{ with } x \rightarrow a$

Contextes d'exceptions:

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

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.

Typage des exceptions: ML introduit un type concret spécial `exn` pour le type des valeurs d'exceptions (les valeurs passées en argument à `raise` et récupérées par le `try...with`). On a les typages suivants:

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

$$\frac{E \vdash a_1 : \tau \qquad E + \{x : \texttt{exn}\} \vdash a_2 : \tau}{E \vdash \texttt{try } a_1 \texttt{ with } x \rightarrow a_2 : \tau}$$

`exn` est un type concret qui comporte un certain nombre de constructeurs prédéfinis, et auquel le programmeur peut ajouter des constructeurs par la déclaration

`exception` C `of` τ

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 (voir la section 4.2).

Exercice 5.5 *(**) Montrer la sûreté de ce typage. (Indication: on montrera qu'un programme bien typé ou bien se réduit en une valeur, ou bien se réduit en* `raise` (v)*, ou bien ne termine pas.)*

Exercice 5.6 *(***) Montrer qu'il existe des programmes qui ne terminent pas dans mini-ML + exceptions (mais sans* `fix`*, sans types concrets, et sans références). (Indication: on pourrait bien sûr définir* `exception C of (exn` $\rightarrow$ `exn)` *et utiliser* `C` *comme dans l'exercice 4.4. Il est plus intéressant de considérer la déclaration* `exception M of ((int` $\rightarrow$ `int)` $\rightarrow$ `(int` $\rightarrow$ `int))` *et de définir à l'aide de cette exception deux fonctions mettant en correspondance les type* `(int` $\rightarrow$ `int)` $\rightarrow$ `(int` $\rightarrow$ `int)` *et* `int` $\rightarrow$ `int`*.)*

5.8 Continuations et opérateurs de contrôle

Pour aller plus loin. Voir *Typing first-class continuations in ML*, R. Harper, B. Duba, D. MacQueen, *Journal of Functional Programming*, 3(4), 1993, et *A Generalization of Exceptions and Control in ML*, C. Gunter, D. Rémy, J. Riecke, *ACM Conf. on Functional Programming and Computer Architecture*, 1995.

Chapter 6

Les enregistrements extensibles

Les enregistrements déclarés à la Caml (comme décrits section 4.3) 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 $\mathtt{fun}\ x \rightarrow x.e$ 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", en ajoutant une ou plusieurs étiquettes 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 $\mathtt{fun}\ x \rightarrow x.e$ qui s'applique à tout type enregistrement possédant un champ de nom e.
- Enregistrements *extensibles*: on peut définir et typer une fonction d'extension $\mathtt{fun}\ x \rightarrow v \rightarrow x@\{e = v\}$ qui renvoie un enregistrement identique à x, mais auquel on a ajouté un champ e contenant la valeur v.

6.1 Sémantique à réduction

Nous ajoutons à mini-ML les constructions suivantes:

Expressions: $a ::= \ldots \mid \{e_1 = a_1; \ldots; e_n = a_n\}$
Opérateurs: $op ::= \ldots \mid \mathtt{proj}_e \mid \mathtt{exten}_e$
Valeurs: $v ::= \ldots \mid \{e_1 = v_1; \ldots; e_n = v_n\}$

L'expression $\{\ldots; e_i = a_i; \ldots\}$ construit un enregistrement de champs e_i associés aux valeurs a_i.

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

On note $a_1@\{e = a_2\}$ pour l'application d'opérateur $\mathtt{exten}_e(a_1, a_2)$. 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:

$$\begin{array}{rcll}(\{e_i = v_i\}_{i\in I}).e_j & \xrightarrow{\varepsilon} & v_j & \text{si } j \in I \\ \{e_i = v_i\}_{i\in I}@\{e_j = w\} & \xrightarrow{\varepsilon} & \{e_j = w; e_i = v_i\}_{i\in I\setminus\{j\}} & \end{array}$$

La seconde règle s'applique que e_j soit ou non déjà liée dans l'enregistrement qu'on étend: si oui, la valeur w remplace la valeur précédemment liée à e_j); si non, l'enregistrement résultat a une étiquette de plus. On parle d'extension *libre* d'enregistrement. L'extension *stricte*, où l'étiquette ajoutée ne doit pas être déjà présente dans l'enregistrement initial, s'obtient par la règle:

$$\{e_i = v_i\}_{i\in I}@\{e_j = w\} \xrightarrow{\varepsilon} \{e_j = w; e_i = v_i\}_{i\in I} \qquad \text{si } j \notin I$$

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

Contextes: $\Gamma ::= \ldots \mid \{e_1 = v_1;\ \ldots;\ e_{n-1} = v_n;\ e_n = \Gamma;\ e_{n+1} = a_{n+1};\ e_m = a_m\}$

6.2 Typage simplifié des enregistrements extensibles

Pour introduire le typage des enregistrements, nous allons supposer que l'ensemble des étiquettes est fini 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 rend le type polymorphe par-rapport à la présence ou l'absence d'un champ. (D'autres valeurs pour τ_i, p.ex. $\tau_i = \mathtt{int}$, ne font pas sens dans le contexte d'un type d'enregistrement; 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:

$$\frac{E \vdash a_1 : \tau_1 \qquad \ldots \qquad E \vdash a_n : \tau_n \qquad \{m_1 \ldots m_k\} = \{\mathtt{e}, \mathtt{f}, \mathtt{g}\} \setminus \{e_1 \ldots e_n\}}{E \vdash \{e_1 = a_1;\ \ldots;\ e_n = a_n\} : \{e_1 : \mathtt{Pre}\ \tau_1;\ \ldots;\ e_n : \mathtt{Pre}\ \tau_n;\ m_1 : \mathtt{Abs};\ \ldots;\ m_k : \mathtt{Abs}\}}$$

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

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:

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

Exercice 6.1 *(*) 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

6.3.1 L'algèbre de types

Pour étendre l'approche de la section 6.2 au cas d'un nombre infini (ou même simplement très grand) d'étiquettes, l'idée est d'introduire une notion de *modèle* pour les champs qui ne sont pas explicitement mentionnés dans un type enregistrement: ce modèle peut être $\emptyset$ 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 $\{\textit{rangée}\}$ ou *rangée* se compose de zéro, une ou plusieurs déclarations de champs teminées par un modèle. Ainsi, $\{e : \mathtt{Pre\ int};\ \emptyset\}$ représente les enregistrements dont le champ e contient un entier et dont tous les autres champs sont indéfinis. L'algèbre de types devient:

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

Les types sont identifiés modulo les deux équations suivantes:

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

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 $\emptyset$ représente une infinité d'étiquettes, toutes absentes.

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

$$\{e_1 : \texttt{Pre int};\ \emptyset\} \text{ et } \{e_2 : \texttt{Abs};\ e_1 : \texttt{Pre int};\ \emptyset\}$$

6.3.2 Règles de typage

La construction d'enregistrements se type comme suit:

$$\frac{E \vdash a_1 : \tau_1 \qquad \ldots \qquad E \vdash a_n : \tau_n}{E \vdash \{e_1 = a_1;\ \ldots;\ e_n = a_n\} : \{e_1 : \texttt{Pre } \tau_1;\ \ldots;\ e_n : \texttt{Pre } \tau_n;\ \emptyset\}}$$

Les champs $e_1 \ldots e_n$ sont présents avec les types $\tau_1, \ldots, \tau_n$; tous les autres champs sont absents, d'où le $\emptyset$ à la fin de la rangée.

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

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

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

Pour l'extension stricte, on prendrait:

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

Exemples: la fonction

```
fun r → (r.a, r.b)
```

a le type $\{a : \texttt{Pre}\ \alpha;\ b : \texttt{Pre}\ \beta;\ \gamma\} \to \alpha \times \beta$. En effet, on a $\{a : \texttt{Pre}\ \alpha;\ b : \texttt{Pre}\ \beta;\ \gamma\} = \{b : \texttt{Pre}\ \beta;\ a : \texttt{Pre}\ \alpha;\ \gamma\}$ 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 $\{\mathtt{a} : \texttt{Pre int};\ \mathtt{b} : \texttt{Pre bool};\ \emptyset\}$. Utilisant l'absorption puis la commutativité, on transforme ce type en le type équivalent

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

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

$$\{\mathtt{c} : \texttt{Pre string};\ \mathtt{a} : \texttt{Pre int};\ \mathtt{b} : \texttt{Pre bool};\ \emptyset\}$$

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 $\{\mathtt{b} : \texttt{Pre bool};\ \mathtt{a} : \texttt{Pre int};\ \emptyset\}$, et on obtient comme type du résultat de l'extension

$$\{\mathtt{b} : \texttt{Pre string};\ \mathtt{a} : \texttt{Pre int};\ \emptyset\}$$

Dernier exemple: extension dans une fonction polymorphe.

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

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

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

6.3.3 Sortes

L'algèbre de types enregistrements que nous venons d'introduire contient un certain nombre de types absurdes, comme par exemple $\emptyset \to \emptyset$ 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 $\{\ldots\}$, p.ex. $\emptyset$ ou $(e : \texttt{Abs};\ \ldots)$.
- 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 $\{a : \texttt{Pre int};\ a : \texttt{Abs};\ \emptyset\}$ (a ne peut pas être à la fois absent et présent), ou $\{a : \texttt{Pre int};\ a : \texttt{Pre bool};\ \emptyset\}$ (a ne peut pas être présent avec deux types différents). De tels types permettent de typer des programmes incorrects, comme par exemple

```
let f = fun r → r.x + 1 in f { x = true }
```

Cet exemple serait typable si l'on pouvait attribuer à l'argument de `f` le type $\{x : \texttt{Pre int};\ x : \texttt{Pre bool};\ \emptyset\}$.

Pour éviter les types contradictoires, il faut s'imposer de respecter l'invariant suivant dans tous les typages:

> Une même étiquette e doit apparaître au plus une fois dans un type enregistrement $\{\varphi\}$.

De la sorte, on peut parler sans ambiguïté de l'information associée à l'étiquette e dans une rangée τ (p.ex. en écrivant τ de manière non ambiguë sous la forme $e : \tau_1;\ \tau_2$).

L'invariant ci-dessus 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 $\varphi = a : \texttt{Pre bool};\ \emptyset$. Cependant, la substitution $\tau[\rho \leftarrow \varphi]$ ne satisfait pas l'invariant.

La manière rigoureuse d'assurer l'invariant ci-dessus, et aussi d'empêcher l'apparition de types absurdes, est d'utiliser des *sortes* (*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 $\emptyset \to \emptyset$ ou $\{a : \texttt{Pre int};\ a : \texttt{Pre bool};\ \emptyset\}$.

On va donc définir par des règles d'inférence 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: $\kappa ::= \texttt{TYPE} \mid \texttt{PRE} \mid \texttt{R}(\{e_1, \ldots, e_n\})$

TYPE est la sorte des types (d'expressions) bien formés. PRE est celle des infos de présence bien formées. Enfin, $\texttt{R}(E)$, où E est un ensemble d'étiquettes, est la sorte des rangées bien formées et qui n'associent pas d'information aux étiquettes $e \in E$. Les règles de "sortage" sont les suivantes:

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

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

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

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.

L'équation d'absorption $\emptyset = e : \texttt{Abs};\ \emptyset$ pose problème car elle ne préserve pas les sortes. En effet, le membre gauche a toutes les sortes $\texttt{R}(E)$ et peut donc apparaître dans tout contexte de rangée, alors que le membre droit a les sortes $\mathtt{R}(E)$ pour tout E ne contenant pas e, et ne peut donc pas apparaître dans une rangée contenant déjà e. Une solution simple est d'annoter $\emptyset$ par sa sorte E:

$$\vdash \emptyset_E :: \mathtt{R}(E)$$

et de réécrire l'équation d'absorption comme suit:

$$\emptyset_E \quad = \quad e : \mathtt{Abs};\ \emptyset_{E \cup \{e\}} \quad \text{si } e \notin E \qquad \text{(absorption)}$$

Proposition 6.1 (Les sortes passent au quotient) *Soient τ_1 et τ_2 deux types et κ une sorte. Si $\vdash \tau_1 :: \kappa$ et τ_1 et τ_2 sont égaux modulo les équations, alors $\vdash \tau_2 :: \kappa$.*

Démonstration: il suffit de prouver le résultat pour les membres gauches et droits des deux équations; il s'étend ensuite à toute expression de type par une récurrence immédiate. Pour l'axiome de commutativité, supposons $\vdash e_1 : \tau_1;\ e_2 : \tau_2;\ \tau :: \kappa$. Vu les règles de sortage, on a $\kappa = \mathtt{R}(E)$ et la dérivation suivante:

$$\dfrac{e_1 \notin E \qquad \vdash \tau_1 :: \mathtt{PRE} \qquad \dfrac{e_2 \notin E \cup \{e_1\} \qquad \vdash \tau_2 :: \mathtt{PRE} \qquad \vdash \tau :: \mathtt{R}(E \cup \{e_1, e_2\})}{\vdash e_2 : \tau_2;\ \tau :: \mathtt{R}(E \cup \{e_1\})}}{\vdash e_1 : \tau_1;\ e_2 : \tau_2;\ \tau :: \mathtt{R}(E)}$$

On a donc $e_1 \neq e_2$ et $e_1 \notin E$ et $e_2 \notin E$. En permutant les étapes finales de cette dérivation, on obtient:

$$\dfrac{e_2 \notin E \qquad \vdash \tau_2 :: \mathtt{PRE} \qquad \dfrac{e_1 \notin E \cup \{e_2\} \qquad \vdash \tau_1 :: \mathtt{PRE} \qquad \vdash \tau :: \mathtt{R}(E \cup \{e_1, e_2\})}{\vdash e_1 : \tau_1;\ \tau :: \mathtt{R}(E \cup \{e_2\})}}{\vdash e_2 : \tau_2;\ e_1 : \tau_1;\ \tau :: \mathtt{R}(E)}$$

C'est le résultat attendu. Pour l'axiome d'absorption, supposons $\vdash \emptyset_E :: \kappa$. Nécessairement, $\kappa = \mathtt{R}(E)$, et $e \notin E$. Donc, on peut dériver $\vdash e : \mathtt{Abs};\ \emptyset_{E \cup \{e\}} :: \kappa$ de la manière suivante:

$$\dfrac{e \notin E \qquad \vdash \mathtt{Abs} :: \mathtt{PRE} \qquad \vdash \emptyset_{E \cup \{e\}} :: \mathtt{R}(E \cup \{e\})}{\vdash e : \mathtt{Abs};\ \emptyset_E :: \mathtt{R}(E)}$$

Enfin, si $\vdash e : \mathtt{Abs};\ \emptyset_{E \cup \{e\}} :: \kappa$, on a $\kappa = \mathtt{R}(E)$ pour un certain E, et $\vdash \emptyset_E :: \mathtt{R}(E)$ se dérive par l'axiome sur $\emptyset$. □

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

Substitutions et sortes: On dit qu'une substitution θ préserve les sortes 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 $\vdash \theta(\tau) :: \kappa$.

Schémas et sortes: un schéma de types $\forall\vec{\alpha}.\ \tau$ est bien sorté si et seulement si $\vdash \tau :: \mathtt{TYPE}$.

6.3.4 Règles de typages avec sortes

Pour assurer que tous les types intervenant dans une dérivation de typage sont bien sortés, il faut modifier légèrement les règles de typage de la manière suivante. Tout d'abord, on redéfinit la notion d'instance $\tau \leq \sigma$ comme $\tau \leq \forall\alpha_1 \ldots \alpha_n.\ \tau'$ si et seulement s'il existe une substitution θ préservant les sortes telle que $\tau = \theta(\tau')$ et $\mathrm{Dom}(\theta) \subseteq \{\alpha_1 \ldots \alpha_n\}$. Ensuite, on ajoute une hypothèse de bon "sortage" pour le type de l'argument d'une fonction (règle (fun)), et on s'assure que toutes les étiquettes d'une expression enregistrement sont distinctes (règle (record)).

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

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

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

$$\frac{E \vdash a_1 : \tau_1 \qquad \ldots \qquad E \vdash a_n : \tau_n \qquad i \neq j \Rightarrow e_i \neq e_j}{E \vdash \{e_1 = a_1;\ \ldots;\ e_n = a_n\} : \{e_1 : \mathtt{Pre}\ \tau_1;\ \ldots;\ e_n : \mathtt{Pre}\ \tau_n;\ \emptyset_{\{e_1,\ldots,e_n\}}\}}\ \text{(record)}$$

Proposition 6.2 (Le typage respecte les sortes) *Supposons $TC(c)$ bien sorté pour toute constante ou opérateur c, et $E(x)$ bien sorté pour tout identificateur $x \in \mathrm{Dom}(E)$. Alors, $E \vdash a : \tau$ implique $\vdash \tau :: \mathtt{TYPE}$.*

Démonstration: récurrence facile sur la dérivation de $E \vdash a : \tau$. Pour les règles (var-inst), (const-inst) et (op-inst), le résultat découle des hypothèses sur E et TC, et sur le fait que les substitutions d'instanciation préservent les sortes. Pour la règle (record), le résultat découle de l'hypothèse de récurrence et du fait que les étiquettes e_i sont deux à deux disjointes. Les autres règles se traitent par application directe de l'hypothèse de récurrence. □

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 (le lemme de substitution). 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 les définitions et les énoncés "passent au quotient" par ces axiomes. Nous ne le ferons pas dans ces notes.

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 un lemme de forme des valeurs selon leur type dans le style du lemme 2.5. En particulier, 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 6.2 *(*)/(**) 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 $\emptyset$ et $e : \alpha;\ \beta$ n'ont pas le même symbole de tête ($\emptyset$ pour l'un, ";" pour l'autre), mais sont pourtant unifiables en prenant $\alpha \leftarrow \texttt{Abs}$ et $\beta \leftarrow \emptyset$, 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 $e : \ldots$ et l'autre par $f : \ldots$), mais sont pourtant unifiables par la substitution

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

où γ est une nouvelle variable. En effet, en appliquant cette substitution aux deux types, on obtient les types

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

qui sont bien égaux modulo commutativité. Plus généralement, pour unifier deux types enregistrement se terminant par des variables différentes $\{\ldots;\ \alpha\}$ et $\{\ldots;\ \beta\}$, on commute les étiquettes de manière à mettre en premier les étiquettes communes aux deux types:

$$\{e_1 : \tau_1;\ \ldots;\ e_n : \tau_n;\ f_1 : \varphi_1;\ \ldots;\ f_k : \varphi_k;\ \alpha\}$$
$$\{e_1 : \tau'_1;\ \ldots;\ e_n : \tau'_n;\ g_1 : \psi_1;\ \ldots;\ g_m : \psi_m;\ \beta\}$$

Ensuite, on effectue la substitution "croisée"

$$\begin{array}{rcl} \alpha & \leftarrow & g_1 : \psi_1;\ \ldots;\ g_m : \psi_m;\ \gamma \\ \beta & \leftarrow & f_1 : \varphi_1;\ \ldots;\ f_k : \varphi_k;\ \gamma \end{array}$$

et on finit en unifiant les paires (τ_i, τ'_i).

Algorithme d'unification: soit C un ensemble d'équations entre types bien sortées (c'est-à-dire, pour toute équation $\tau_1 \stackrel{?}{=} \tau_2$ dans C, il existe une sorte κ telle que $\vdash \tau_1 :: \kappa$ et $\vdash \tau_2 :: \kappa$). L'unificateur principal $\mathtt{mgu}(C)$ se calcule par l'algorithme suivant. Les premiers cas sont identiques à ceux de la section 3.2.3 (unification entre types "normaux", sans équations):

$$\begin{array}{rcl} \mathtt{mgu}(\emptyset) & = & id \\ \mathtt{mgu}(\{\alpha \stackrel{?}{=} \alpha\} \cup C & = & \mathtt{mgu}(C) \\ \mathtt{mgu}(\{\alpha \stackrel{?}{=} \tau\} \cup C) & = & \mathtt{mgu}(C[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau] \text{ si } \alpha \text{ n'est pas libre dans } \tau \\ \mathtt{mgu}(\{\tau \stackrel{?}{=} \alpha\} \cup C) & = & \mathtt{mgu}(C[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau] \text{ si } \alpha \text{ n'est pas libre dans } \tau \\ \mathtt{mgu}(\{\tau_1 \to \tau_2 \stackrel{?}{=} \tau'_1 \to \tau'_2\} \cup C) & = & \mathtt{mgu}(\{\tau_1 \stackrel{?}{=} \tau'_1; \tau_2 \stackrel{?}{=} \tau'_2\} \cup C) \\ \mathtt{mgu}(\{\tau_1 \times \tau_2 \stackrel{?}{=} \tau'_1 \times \tau'_2\} \cup C) & = & \mathtt{mgu}(\{\tau_1 \stackrel{?}{=} \tau'_1; \tau_2 \stackrel{?}{=} \tau'_2\} \cup C) \\ \mathtt{mgu}(\{\ \{\tau_1\} \stackrel{?}{=} \{\tau_2\}\ \} \cup C) & = & \mathtt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_2\} \cup C) \end{array}$$

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

$$\begin{array}{rcl} \mathtt{mgu}(\{\emptyset \stackrel{?}{=} \emptyset\} \cup C) & = & \mathtt{mgu}(C) \\ \mathtt{mgu}(\{e : \tau;\ \tau' \stackrel{?}{=} \emptyset\} \cup C) & = & \mathtt{mgu}(\{\tau \stackrel{?}{=} \mathtt{Abs};\quad \tau' \stackrel{?}{=} \emptyset\} \cup C) \\ \mathtt{mgu}(\{\emptyset \stackrel{?}{=} e : \tau;\ \tau'\} \cup C) & = & \mathtt{mgu}(\{\tau \stackrel{?}{=} \mathtt{Abs};\ \tau' \stackrel{?}{=} \emptyset\} \cup C) \\ \mathtt{mgu}(\{e : \tau_1;\ \tau'_1 \stackrel{?}{=} e : \tau_2;\ \tau'_2\} \cup C) & = & \mathtt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_2;\quad \tau'_1 \stackrel{?}{=} \tau'_2\} \cup C) \\ \mathtt{mgu}(\{(e : \tau_1;\ \tau'_1) \stackrel{?}{=} (f : \tau_2;\ \tau'_2)\} \cup C) & = & \mathtt{mgu}(\{\tau'_1 \stackrel{?}{=} (f : \tau_2;\ \alpha);\quad \tau'_2 \stackrel{?}{=} (e : \tau_1;\ \alpha)\} \cup C) \\ & & \text{si } e \neq f \text{ et } \alpha \text{ est une nouvelle variable} \end{array}$$

Dans le dernier cas, α est choisie non libre dans le système d'équations initiale et de la sorte qui va bien pour préserver le bon "sortage" des équations. C'est-à-dire, si $\mathtt{R}(E)$ est la sorte commune à $(e : \tau_1;\ \tau'_1)$ et $(f : \tau_2;\ \tau'_2)$, on choisit α de la sorte $\mathtt{R}(E \cup \{e; f\})$.

Enfin, les cas d'unification entre drapeaux de présence sont immédiats:

$$\begin{array}{rcl} \mathtt{mgu}(\{\mathtt{Abs} \stackrel{?}{=} \mathtt{Abs}\} \cup C) & = & \mathtt{mgu}(C) \\ \mathtt{mgu}(\{\mathtt{Pre}\ \tau_1 \stackrel{?}{=} \mathtt{Pre}\ \tau'\} \cup C) & = & \mathtt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_2\} \cup C) \end{array}$$

L'hypothèse que les équations de C sont bien sortées garantit que l'unificateur $\mathtt{mgu}(C)$ préserve les sortes.

Exercice de programmation 6.1 *Implémenter la fonction* mgu.

6.5.2 Inférence de types

Munis de cet algorithme d'unification, il ne nous reste plus qu'à adapter l'algorithme W de la section 3.3.1 comme suit (ce qui change est souligné):

- Si a est une variable x avec $x \in \mathrm{Dom}(E)$:
  prendre $(\tau, V') = \texttt{Inst}(E(x), V)$ et $\varphi = id$.

- Si a est une constante c ou un opérateur op:
  prendre $(\tau, V') = \texttt{Inst}(TC(a), V)$ et $\varphi = id$.

- Si a est $\texttt{fun}\ x \to a_1$:
  soit α une nouvelle variable $\underline{\text{de sorte } \texttt{TYPE}}$ prise dans V
  soit $(\tau_1, \varphi_1, V_1) = W(E + \{x : \alpha\}, a_1, V \setminus \{\alpha\})$
  prendre $\tau = \varphi_1(\alpha) \to \tau_1$ et $\varphi = \varphi_1$ et $V = V_1$.

- Si a est une application $a_1\ a_2$:
  soit $(\tau_1, \varphi_1, V_1) = W(E, a_1, V)$
  soit $(\tau_2, \varphi_2, V_2) = W(\varphi_1(E), a_2, V_1)$
  soit α une nouvelle variable $\underline{\text{de sorte } \texttt{TYPE}}$ prise dans V_2
  soit $\mu = \texttt{mgu}\{\varphi_2(\tau_1) \stackrel{?}{=} \tau_2 \to \alpha\}$
  prendre $\tau = \mu(\alpha)$ et $\varphi = \mu \circ \varphi_2 \circ \varphi_1$ et $V = V_2 \setminus \{\alpha\} \underline{\setminus \mathrm{Dom}(\mu)}$.

- Si a est une paire (a_1, a_2):
  soit $(\tau_1, \varphi_1, V_1) = W(E, a_1, V)$
  soit $(\tau_2, \varphi_2, V_2) = W(\varphi_1(E), a_2, V_1)$
  prendre $\tau = \varphi_2(\tau_1) \times \tau_2$ et $\varphi = \varphi_2 \circ \varphi_1$ et $V = V_2$.

- Si a est $\texttt{let}\ x = b\ \texttt{in}\ c$:
  soit $(\tau_1, \varphi_1, V_1) = W(E, a_1, V)$
  soit $(\tau_2, \varphi_2, V_2) = W(\varphi_1(E) + \{x : \texttt{Gen}(\tau_1, \varphi_1(E))\}, a_2, V_1)$
  prendre $\tau = \tau_2$ et $\varphi = \varphi_2 \circ \varphi_1$ et $V = V_2$.

- $\underline{\text{Si } a \text{ est } \{e_1 = a_1;\ \ldots;\ e_n = a_n\}\text{:}}$
  $\underline{\text{vérifier que les étiquettes } e_i \text{ sont deux à deux distinctes}}$
  soit $(\tau_1, \varphi_1, V_1) = W(E, a_1, V)$
  soit $(\tau_2, \varphi_2, V_2) = W(\varphi_1(E), a_2, V_1)$
  soit ...
  soit $(\tau_n, \varphi_n, V_n) = W((\varphi_{n-1} \circ \cdots \circ \varphi_1)(E), a_n, V_{n-1})$
  prendre $\tau = \{e_1 : \texttt{Pre}\ (\varphi_n \circ \cdots \circ \varphi_2)(\tau_1);\ \ldots;\ e_n : \texttt{Pre}\ \tau_n;\ \emptyset\}$ et $\varphi = \varphi_n \circ \cdots \circ \varphi_1$ et $V = V_n$

Dans le cas de l'application $a_1\ a_2$, on prend $V = V_2 \setminus \{\alpha\} \setminus \mathrm{Dom}(\mu)$ et non pas juste $V = V_2 \setminus \{\alpha\}$ comme dans la section 3.3.1 car maintenant l'algorithme `mgu` peut introduire de nouvelles variables de types (dans le cas de deux rangées commençant par des étiquettes différentes), et ces nouvelles variables doivent être enlevées du résultat V. C'est pour cela que nous enlevons de V toutes les variables de $\mathrm{Dom}(\mu)$.

Pour finir, il faut s'assurer que la fonction d'instance triviale respecte les sortes:

$$\texttt{Inst}(\forall \alpha_1, \ldots, \alpha_n.\tau,\ V) = (\tau[\alpha_i \leftarrow \beta_i],\ V \setminus \{\beta_1 \ldots \beta_n\})$$

où $\beta_1, \ldots, \beta_n$ sont n variables distinctes choisies dans V et telles que $K(\beta_i) = K(\alpha_i)$ pour tout i.

On admettra les résultats de correction et de principalité suivants (analogues aux théorèmes 3.1 et 3.2):

Théorème 6.1 (Correction de l'algorithme W) *Soit E un environnement bien sorté. Si $W(E, a, V) = (\tau, \varphi, V')$, alors on peut dériver $\varphi(E) \vdash a : \tau$, et de plus φ préserve les sortes.*

Théorème 6.2 (Complétude et principalité de l'algorithme W) *Soit V un ensemble de variables comprenant une infinité de variable de chaque sorte (c.à.d. tel que $V \cap K^{-1}(\kappa)$ est infini pour toute sorte κ). Supposons E bien sorté et $V \cap \mathcal{L}(E) = \emptyset$. S'il existe un type τ' et une substitution φ' préservant les sortes tels que $\varphi'(E) \vdash a : \tau'$, alors $W(E, a, V)$ n'est pas* `err`*; au contraire, il existe τ, φ, V' et une substitution θ préservant les sortes tels que*

$$W(E, a, V) = (\tau, \varphi, V') \qquad \text{et} \qquad \tau' = \theta(\tau) \qquad \text{et} \qquad \varphi' = \theta \circ \varphi \text{ hors de } V.$$

Remarque sur le sortage: l'algorithme W n'effectue aucune vérification de sortes à proprement parler, car les conditions de bon sortage sont garanties "par construction". Par exemple, il n'est pas nécessaire de vérifier que le type de l'argument d'un `fun` est de la sorte `TYPE`, comme le fait la règle de typage (fun): l'algorithme utilise pour ce type une nouvelle variable α de la sorte `TYPE`, initialement, variable qui se trouve ensuite instanciée par des substitutions φ préservant les sortes, et donc telles que $\varphi(\alpha)$ est toujours de la sorte `TYPE`.

La seule contrainte de sortage qui apparaît dans les algorithmes W et `mgu` est dans le choix des nouvelles variables, qui doivent être choisies avec la sorte qui convient pour le contexte dans lequel elles vont être utilisées. Là non plus, il n'y a pas besoin de vérifier des sortes pendant le déroulement de l'algorithme: au lieu de se donner à l'avance un sortage des variables K, il suffit de construire K incrémentalement pendant le déroulement de l'algorithme, en attribuant aux nouvelles variables les sortes qui vont bien. Autrement dit, au lieu de

> soit α une nouvelle variable de sorte κ

on peut lire

> soit α une nouvelle variable
> prendre $K(\alpha) = \kappa$

Le seul cas où une vérification de sortes est nécessaire est pour les types fournis par le programmeur, p.ex. dans des contraintes de types $(a : \tau)$ ou des déclarations de types concrets.

Exercice 6.3 *(**)/(***) 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.)*

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

6.6 Les sommes ouvertes

De même que les enregistrements polymorphes et extensibles généralisent les enregistrements déclarés de la section 4.3, on peut généraliser les types concrets de la section 4.2 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 (type concret) contenant au moins certains constructeurs avec certains types. Ceci fait l'objet de l'exercice suivant.

Exercice 6.4 *(***) 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:*

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

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 $\emptyset$ 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)
```

Chapter 7

Programmation par objets et classes

La conception d'un système de typage statique sûr pour la programmation par objets pose de sérieux problèmes. 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. (Ceci s'ajoute bien sûr aux garanties habituelles de sûreté, comme p.ex. qu'on élimine `1(2)` ou `1 + "foo"`.) Les difficultés proviennent des nombreux traits originaux des objets et des classes, dont il faut rendre compte:

- "self" et la liaison tardive;
- encapsulation des variables d'instance (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.

Une difficulté majeure est de combiner inférence de types et subsomption implicite. La *subsomption* est l'acte de considérer une expression a de type τ comme étant d'un super-type τ' de τ. Elle est *implicite* si le changement de type (de τ à τ') n'est pas marqué dans le texte du programme, et *explicite* sinon — comme par exemple en Objective Caml où la subsomption doit être écrite sous forme d'une coercion $(a : \tau :> \tau')$. On distingue les trois combinaisons suivantes:

1. Subsomption implicite, typage explicite, pas d'inférence de types. On déclare 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. Subsomption explicite, inférence de types à la ML. C'est l'approche suivie en Objective Caml.

3. Subsomption implicite, inférence de types par contraintes de sous-typage (généralisant l'inférence à la ML). Cette approche est le résultat de travaux récents de recherche et n'a pas encore été intégrée dans un langage complet. On ne sait pas encore si elle est réaliste (les types sont excessivement gros, p.ex.).

Dans ce cours, nous étudions (2) en détails, car c'est une application directe des rangées de types que nous avons déjà utilisées au chapitre 6. Nous verrons aussi l'approche (3) dans un cadre simplifié. Les approches (1) et (3) sont détaillées dans le cours d'option de G. Castagna et F. Pottier.

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. La construction d'un objet se fait donc en listant ses méthodes et ses variables d'instance.

Expressions:	$a ::= x \mid c \mid op \mid \texttt{fun}\ x \rightarrow a \mid a_1\ a_2$	
	$\mid (a_1, a_2) \mid \texttt{let}\ x = a_1\ \texttt{in}\ a_2$	comme d'habitude
	$\mid \texttt{obj}(x)\langle \ldots ; \texttt{val}\ x_i = a_i; \ldots ; \texttt{method}\ m_j = a'_j; \ldots \rangle$	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.

L'identificateur x dans la construction $\texttt{obj}(x)\langle \ldots \rangle$ correspond au paramètre "self" des méthodes. Une méthode de cet objet peut donc faire $x\#m$ pour rappeler une autre méthode du même objet.

Les variables d'instance sont considérées comme immuables. On peut y mettre des références comme au chapitre 4 pour modéliser les variables d'instance mutables.

7.1.2 Règles de réduction

La règle de réduction pour $\#m$ est la suivante:

$$\begin{array}{lll} v\#m_j & \xrightarrow{\varepsilon} & a_j[s \leftarrow v,\ x_1 \leftarrow v_1,\ \ \ldots, x_n \leftarrow v_n] \\ & & \text{si } v = \texttt{obj}(s)\langle \texttt{val}\ x_i = v_i; \ldots ; \texttt{val}\ x_n = v_n; \texttt{method}\ m_1 = a_1; \ldots ; \texttt{method}\ m_k = a_k \rangle \end{array}$$

L'auto-application est visible ici par le remplacement de l'identificateur s par l'objet v lui-même dans le corps a_j de la méthode m_j. De la sorte, les appels $s\#m'$ présents dans a_j 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 et les contextes d'évaluation sont:

Valeurs:

$$\begin{array}{ll} v ::= & c \mid op \mid \texttt{fun}\ x \rightarrow a \mid (v_1, v_2) \\ & \mid \texttt{obj}(s)\langle \texttt{val}\ x_1 = v_1; \ldots ; \texttt{val}\ x_n = v_n; \texttt{method}\ m_1 = a_1; \ldots ; \texttt{method}\ m_k = a_k \rangle \end{array}$$

Contextes:

$$\Gamma ::= [\,] \mid \Gamma\ a \mid v\ \Gamma \mid \texttt{let}\ x = \Gamma\ \texttt{in}\ a \mid (\Gamma, a) \mid (v, \Gamma)$$
$$\mid \texttt{obj}(s)\langle \ldots \texttt{val}\ x_{i-1} = v_i; \texttt{val}\ x_i = \Gamma; \texttt{val}\ x_{i+1} = a_{i+1}; \ldots; \texttt{method}\ m_j = a'_j; \ldots; \rangle$$

Notons que dans un objet complètement évalué, les variables d'instance sont complètement évaluées, mais pas les corps des méthodes. En ce sens, la partie "variables d'instance" d'un objet se comporte comme un n-uplet ou un enregistrement, alors que la partie "méthode" se comporte comme un corps de fonction (de paramètre s).

7.1.3 L'algèbre de types

Le type d'un objet est essentiellement identique à celui d'un enregistrement: il liste les noms des méthodes de l'objet, avec pour chaque méthode le type de son résultat. Nous conservons les mêmes mécanismes de rangées et d'annotations de présence que pour les types d'enregistrements.

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
	$\mid \emptyset$	la rangée vide
	$\mid m : \tau_1;\ \tau_2$	la rangée contenant $m : \tau_1$ plus ce que contient la rangée τ_2
	$\mid \texttt{Abs}$	le champ est absent (indéfini)
	$\mid \texttt{Pre}\ \tau$	le champ est présent (défini) avec le type τ

Exemples: le type $\langle m : \texttt{Pre int}; n : \texttt{Pre string}; \emptyset \rangle$ 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 $\texttt{obj}(s)\langle \ldots \rangle$.

Le type $\langle m : \texttt{Pre int};\ \alpha \rangle$ est le type des objets possédant une méthode m renvoyant un entier, et éventuellement d'autres méthodes (dont le type va instancier α). De tels types "ouverts" apparaissent naturellement pour les paramètres de fonctions; ainsi,

```
fun obj → 1 + obj#m
```

a le type $\langle m : \texttt{Pre int};\ \alpha \rangle \to \texttt{int}$.

Théorie équationnelle: comme pour les enregistrements, on considère les types modulo les équations de commutativité et d'absorption:

$$\begin{array}{rcll} m_1 : \tau_1;\ m_2 : \tau_2;\ \tau & = & m_2 : \tau_2;\ m_1 : \tau_1;\ \tau & \text{(commutativité)} \\ \emptyset & = & m : \texttt{Abs};\ \emptyset & \text{(absorption)} \end{array}$$

Sortes: comme pour les enregistrements, on utilise des sortes pour éviter les types absurdes $\langle m : \texttt{int}; \texttt{bool} \rangle$ ou contradictoires $\langle m : \texttt{Pre int};\ m : \texttt{Pre bool};\ \emptyset \rangle$. Le système de sortage est identique à celui des enregistrements (section 6.3.3).

7.1.4 Règles de typage

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

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

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

$$\frac{\begin{array}{c} i \neq j \Rightarrow m_i \neq m_j \qquad E \vdash a_i : \tau_i \\ E + \{x : \langle m_1 : \texttt{Pre}\ \tau'_1;\ \ldots;\ m_k : \texttt{Pre}\ \tau'_k;\ \emptyset\rangle;\ x_i : \tau_i\} \vdash a'_j : \tau'_j \end{array}}{E \vdash \texttt{obj}(x)\langle \ldots; \texttt{val}\ x_i = a_i; \ldots; \texttt{method}\ m_j = a'_j; \ldots\rangle : \langle m_1 : \texttt{Pre}\ \tau'_1;\ \ldots;\ m_k : \texttt{Pre}\ \tau'_k;\ \emptyset\rangle}\ \text{(objet)}$$

Enfin, le schéma de types pour l'opérateur $\#m$ est bien sûr:

$$\#m \quad : \quad \forall \alpha, \beta.\ \langle m : \texttt{Pre}\ \alpha;\ \beta\rangle \to \alpha$$

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 $v\#m \xrightarrow{\varepsilon} a'$ et $E \vdash v\#m : \tau$, avec $v = \texttt{obj}(s)\langle \ldots; \texttt{val}\ x_i = v_i; \ldots; \texttt{method}\ m_j = a_j; \ldots\rangle$, $m = m_j$, et $a' = a_j[s \leftarrow v,\ x_i \leftarrow v_i]$. On a forcément une dérivation de typage de la forme:

$$\frac{\dfrac{i \neq j \Rightarrow m_i \neq m_j \qquad E \vdash v_i : \tau'_i \qquad E + \{s : \tau_s;\ x_i : \tau'_i\} \vdash a_k : \tau_k}{E \vdash v : \tau_s}}{E \vdash v\#m_j : \tau_j}$$

avec $\tau = \tau_j$ et $\tau_s = \langle \ldots; m_i : \texttt{Pre}\ \tau_i; \ldots; \emptyset\rangle$. On a donc $E \vdash v_i : \tau'_i$ pour tout i, et de plus $E \vdash v : \tau_s$. Appliquant un lemme de substitution semblable au lemme 2.2 à $E + \{s : \tau_s;\ x_i : \tau'_i\} \vdash a_j : \tau_j$, il vient $E \vdash a_j[s \leftarrow v,\ x_i \leftarrow v_i] : \tau_j$, c'est-à-dire $E \vdash a' : \tau_j$.

La preuve de (H2) est du même style; nous l'omettons.

7.1.6 Inférence de types

L'inférence de types pour le système de types de cette section est très proche de celle pour mini-ML avec enregistrements (section 6.5). On utilise un algorithme d'unification dans le style de la section 6.5.1, et un algorithme W modifié avec le cas suivant pour la construction d'objets:

- Si a est $\texttt{obj}(x)\langle \texttt{val}\ x_i = a_i; \texttt{method}\ m_j = a'_j\rangle$:
  vérifier que les noms de méthodes m_i sont deux à deux distincts
  soit $(\vec{\tau}, \varphi_1, V_1) = \vec{W}(E, \vec{a}, V)$
  soit $\alpha \in V_1$ une variable de sorte $\texttt{TYPE}$

soit $(\vec{\tau'}, \varphi_2, V_2) = W(\varphi_1(E) + \{\mathtt{self}\ x : \alpha; \mathtt{val}\ x_i : \tau_i\}, \vec{a'}, V_1 \setminus \{\alpha\})$
soit $\mu = \mathtt{mgu}\{\varphi_2(\alpha) \stackrel{?}{=} \langle m_1 : \mathtt{Pre}\ \tau'_1;\ \ldots;\ m_k : \mathtt{Pre}\ \tau'_k;\ \emptyset\rangle\}$
prendre $\tau = \mu(\varphi_2(\alpha))$ et $\varphi = \mu \circ \varphi_2 \circ \varphi_1$ et $V = V_2 \setminus \mathrm{Dom}(\mu)$.

On a noté $\vec{W}$ l'inférence de types pour une séquence d'expressions: $(\vec{\tau}, \varphi, V') = \vec{W}(E, \vec{a}, V)$ est défini par:

soit $(\tau'_1, \varphi_1, V_1) = W(E, a_1, V)$
soit $(\tau'_2, \varphi_2, V_2) = W(\varphi_1(E), a_2, V_1)$
soit ...
soit $(\tau'_n, \varphi_n, V_n) = W((\varphi_{n-1} \circ \cdots \circ \varphi_1)(E), a_n, V_{n-1})$
prendre $\tau_i = (\varphi_n \circ \cdots \circ \varphi_{i+1})(\tau'_i)$ pour $i = 1 \ldots n$, et $\varphi = \varphi_n \circ \cdots \circ \varphi_1$ et $V' = V_n$.

7.2 Sous-typage et subsomption 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 $a : \langle \mathtt{m} : \mathtt{Pre\ int};\ \emptyset\rangle$ et $b : \langle \mathtt{m} : \mathtt{Pre\ int};\ \mathtt{n} : \mathtt{Pre\ string};\ \emptyset\rangle$, l'expression `if` *cond* `then` a `else` b n'est pas typable, et en particulier n'a pas le type "naturel" $\langle \mathtt{m} : \mathtt{Pre\ int};\ \emptyset\rangle$.

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:

$$\begin{array}{rcl} \mathtt{coerce}_{\tau,\tau'} & : & \forall \vec{\alpha}.\ \tau \to \tau' \quad \text{si } \tau <: \tau' \text{ et } \vec{\alpha} = \mathcal{L}(\tau) \cup \mathcal{L}(\tau') \\ \mathtt{coerce}_{\tau,\tau'}(v) & \stackrel{\varepsilon}{\to} & v \end{array}$$

Remarquez que ces opérateurs `coerce` généralisent très naturellement les contraintes de types de ML comme présentées section 4.4.

Relation de sous-typage: La relation de sous-typage $<:$ qui caractérise les opérateurs `coerce` valides est définie par les règles d'inférence suivantes:

$$T <: T \qquad \alpha <: \alpha \qquad \frac{\tau' <: \tau \quad \varphi <: \varphi'}{(\tau \to \varphi) <: (\tau' \to \varphi')} \qquad \frac{\tau <: \tau' \quad \varphi <: \varphi'}{(\tau \times \varphi) <: (\tau' \times \varphi')}$$

$$\frac{\tau <: \tau'}{\langle\tau\rangle <: \langle\tau'\rangle} \qquad \tau <: \emptyset \qquad \frac{\tau <: \tau' \quad \varphi <: \varphi'}{(m : \tau;\ \varphi) <: (m : \tau';\ \varphi')}$$

$$\mathtt{Abs} <: \mathtt{Abs} \qquad \frac{\tau <: \tau'}{\mathtt{Pre}\ \tau <: \mathtt{Pre}\ \tau'}$$

Deux types de base ou deux variables de types sont en relation de sous-typage si et seulement si ils sont égaux.

Un type objet $\langle\tau\rangle$ est sous-type d'un autre $\langle\tau'\rangle$ si toutes les méthodes mentionnées dans la rangée τ' sont également mentionnées dans τ, et leur type dans τ est sous-type de celui dans τ'. Si τ' se termine par $\emptyset$, la rangée τ peut aussi mentionner des méthodes supplémentaires qui n'apparaissent pas dans τ' (en raison de l'axiome $\tau <: \emptyset$). En revanche, si τ' se termine par une variable de rangée, τ doit se terminer par la même variable et mentionner les mêmes méthodes que τ'.

Pour les types produit, il y a sous-typage si les types des premières composantes sont sous-types, ainsi que les types des secondes composantes. On dit que le produit est un constructeur de type *covariant* en ses deux arguments. Autrement dit, les deux fonctions des types dans les types $\_ \times \tau$ et $\tau \times \_$ sont des fonctions croissantes pour l'ordre $<:$.

Enfin, pour le sous-typage entre types flèches, on a covariance en le type du résultat, mais *contravariance* en le type de l'argument: les types arguments doivent être en sous-typage dans l'ordre inverse du sous-typage entre les types flèche. Autrement dit, la fonction $\_ \rightarrow \tau$ est décroissante, alors que la fonction $\tau \rightarrow \_$ est croissante. Ceci se comprend mieux en pensant aux types comme à des ensembles de valeurs, et à la relation de sous-typage comme à l'inclusion entre ensembles de valeurs. En théorie des ensembles, on a aussi que $A \rightarrow B$ (l'ensemble des fonctions de A dans B) est inclus dans $A' \rightarrow B'$ (l'ensemble des fonctions de A' dans B') si et seulement si $A' \subseteq A$ et $B \subseteq B'$.

Exemples: on a les sous-typages et les non-sous-typages suivants:

$$\begin{array}{rcl}
\langle m : \texttt{Pre int};\, \emptyset\rangle & <: & \langle\emptyset\rangle \\
\langle o : \langle m : \texttt{Pre int};\, \emptyset\rangle\rangle & <: & \langle o : \langle\emptyset\rangle\rangle \\
\langle m : \texttt{Pre int};\, \alpha\rangle & \not<: & \langle\alpha\rangle \\
\texttt{int} \rightarrow \langle m : \texttt{Pre int};\, \emptyset\rangle & <: & \texttt{int} \rightarrow \langle\emptyset\rangle \\
\langle m : \texttt{Pre int};\, \emptyset\rangle \rightarrow \texttt{int} & \not<: & \langle\emptyset\rangle \rightarrow \texttt{int} \\
\langle\emptyset\rangle \rightarrow \texttt{int} & <: & \langle m : \texttt{Pre int};\, \emptyset\rangle \rightarrow \texttt{int}
\end{array}$$

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 et donc de montrer la sûreté du typage est d'ajouter (pour les besoins de la preuve de sûreté uniquement) une règle de subsomption implicite:

$$\frac{E \vdash a : \tau \qquad \tau <: \tau'}{E \vdash a : \tau'}\ (\text{sub})$$

Avec cette règle, on a bien que la réduction $\texttt{coerce}_{\tau,\tau'}(v) \xrightarrow{\varepsilon} v$ préserve le typage, puisque v qui a le type τ par hypothèse a aussi le type τ' par application de la règle (sub).

Bien sûr, cette règle (sub) rend l'inférence de types problématique — c'est pour éviter cela que nous mettons 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 et de variables d'instance. Une construction `new` explicite permet de prendre une instance d'une classe, c'est-à-dire de construire un objet ayant les méthodes et les variables indiquées dans la classe. 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).

Autres simplifications: nous traiterons seulement les classes ne contenant pas de variables d'instances, uniquement des méthodes. Nous ne traitons pas non plus l'héritage multiple. (Voir l'article de Rémy et Vouillon référencé à la fin de ce chapitre pour un traitement plus complet.)

Expressions:	$a ::= \ldots$	
	$\mid \mathtt{new}(a)$	création d'un objet
	$\mid \mathtt{class}(x)\langle \ldots m_i = a_i \ldots\rangle$	définition de classe
	$\mid \mathtt{class}(x)\langle \mathtt{inherit}(a);\ m = a\rangle$	héritage et ajout ou redéfinition d'une méthode
Types:	$\tau ::= \ldots \mid \mathtt{class}(\tau_1)\ \tau_2$	type de classe

7.3.1 Évaluation des classes

L'évaluation des classes consiste à résoudre l'héritage. Pour ce faire, on évalue la classe héritée, et on remplace la méthode redéfinie si nécessaire.

$$\begin{aligned}\mathtt{new}(\mathtt{class}(x)\langle m_i = a_i\rangle) &\xrightarrow{\varepsilon} \mathtt{obj}(x)\langle \mathtt{method}\ m_i = a_i\rangle\\ \mathtt{class}(x)\langle \mathtt{inherit}(\mathtt{class}(y)\langle m_i = a_i\rangle_{i=1\ldots n});\ m = a\rangle &\xrightarrow{\varepsilon} \mathtt{class}(y)\langle (m_i = a_i)_{i=1\ldots n, m_i \neq m};\ m = a[x \leftarrow y]\rangle\end{aligned}$$

Valeurs: $v ::= \ldots \mid \mathtt{class}(x)\langle m_i = a_i\rangle$

Contextes: $\Gamma ::= \ldots \mid \mathtt{new}(\Gamma) \mid \mathtt{class}(x)\langle \mathtt{inherit}\ \Gamma;\ m = a\rangle$

7.3.2 Typage des classes

Le type d'une classe est $\mathtt{class}(\tau_1)\ \tau_2$, 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. Le cas $\tau_1 = \tau_2$ correspond à une classe "autosuffisante", c'est-à-dire qui définit toutes les méthodes qu'elle utilise dans des appels via "self". On peut donc faire `new` sur une telle classe pour construire un objet avec ses méthodes. Le cas où τ_1 mentionne des méthodes qui n'apparaissent pas dans τ_2 correspond à une classe virtuelle en Objective Caml: certaines méthodes restent à définir. Par exemple, la classe OCaml

```
class virtual c =
  object(self)
```

```
    method virtual m : int
    method n = 1 + self#m
  end
```

reçoit ici le type $\texttt{class}(\langle \texttt{m} : \texttt{Pre int};\ \alpha\rangle)\ \langle \texttt{n} : \texttt{Pre int}; \emptyset\rangle$.

$$\frac{E \vdash a : \texttt{class}(\tau)\ \tau}{E \vdash \texttt{new}(a) : \tau}\ \text{(new)}$$

$$\frac{E + \{x : \tau\} \vdash a_i : \tau_i}{E \vdash \texttt{class}(x)\langle m_i = a_i\rangle : \texttt{class}(\tau)\ \langle m_1 : \texttt{Pre}\ \tau_1;\ \ldots;\ m_k : \texttt{Pre}\ \tau_k;\ \emptyset\rangle}\ \text{(classe)}$$

$$\frac{E \vdash a_1 : \texttt{class}(\tau_x)\ \langle m : \texttt{Abs};\ \tau\rangle \qquad E + \{x : \tau_x\} \vdash a_2 : \tau'}{E \vdash \texttt{class}(x)\langle \texttt{inherit}(a_1);\ m = a_2\rangle : \texttt{class}(\tau_x)\ \langle m : \texttt{Pre}\ \tau';\ \tau\rangle}\ \text{(exten)}$$

$$\frac{E \vdash a_1 : \texttt{class}(\tau_x)\ \langle m : \texttt{Pre}\ \tau';\ \tau\rangle \qquad E + \{x : \tau_x\} \vdash a_2 : \tau'}{E \vdash \texttt{class}(x)\langle \texttt{inherit}(a_1);\ m = a_2\rangle : \texttt{class}(\tau_x)\ \langle m : \texttt{Pre}\ \tau';\ \tau\rangle}\ \text{(redéf)}$$

Pour l'héritage, on a deux règles suivant que l'on ajoute une nouvelle méthode ou que l'on redéfinit une méthode existant dans la classe héritée. Dans le second cas, la nouvelle définition de la méthode doit avoir le même type que dans la super-classe.

Dans tous les cas d'héritage, le type de "self" doit être le même dans la nouvelle classe et dans la classe héritée. Cela garantit que les méthodes héritées qui retournent "self" en résultat restent bien typées. Ceci ne restreint pas l'expressivité du langage, car en général la classe dont on hérite est liée à un identificateur, et a donc un schéma de type de la forme

$$\forall\alpha.\ \texttt{class}(\langle m_i : \tau_i;\ \alpha\rangle)\ \langle m_j : \tau_j;\ \emptyset\rangle$$

En instanciant correctement la variable de rangée α, on peut facilement satisfaire les contraintes des règles (exten) et (redéf).

Le même effet de polymorphisme sur les types de classes s'applique aussi à la spécialisation de "selftype" entre une classe et la classe dont elle hérite.

Exemple: voici un fragment d'Objective Caml qui illustre plusieurs points délicats: classe paramétrée par un type; spécialisation de ce paramètre dans une sous-classe; utilisation de "selftype" pour typer une méthode qui renvoie "self".

```
class ['a] comparable =
  object (self : 'selftype)
    method virtual leq : 'selftype -> bool
    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
```

La traduction dans notre petit langage de classes donne:

```
let comparable =
  class(self) < min = fun y → if self#leq y then self else y > in
let int_comparable =
  fun n →
    class(self) < inherit(comparable);
                  method get = n;
                  method leq = fun obj → self#get <= obj#get >
```

Cet exemple est typable avec les schémas de types suivants:

$$\begin{array}{rcl}
\texttt{comparable} & : & \forall\beta.\ \texttt{class}(\tau)\ \langle \texttt{min} : \texttt{Pre}\ (\tau \to \tau);\ \emptyset \rangle \\
 & & \text{avec } \tau = \mu\alpha.\ \langle \texttt{leq} : \texttt{Pre}(\alpha \to \texttt{bool});\ \beta \rangle \\
\texttt{int_comparable} & : & \forall\beta.\ \texttt{class}(\tau')\ \langle \texttt{min} : \texttt{Pre}(\tau' \to \tau');\ \texttt{get} : \texttt{Pre int};\ \texttt{leq} : \texttt{Pre}\ (\tau' \to \texttt{bool});\ \emptyset \rangle \\
 & & \text{avec } \tau' = \mu\alpha.\ \langle \texttt{leq} : \texttt{Pre}(\alpha \to \texttt{bool});\ \texttt{get} : \texttt{Pre int};\ \beta \rangle
\end{array}$$

Une instance du schéma de `int_comparable` est $\texttt{class}(\tau'')\ \tau''$, avec $\tau'' = \langle \texttt{min} : \texttt{Pre}(\tau'' \to \tau''); \texttt{get} : \texttt{Pre int}; \texttt{leq} : \texttt{Pre}(\tau'' \to \texttt{bool}); \emptyset \rangle$, ce qui montre que l'on peut faire `new(int_comparable)`.

7.4 Les types récursifs

La programmation par objets fait rapidement apparaître le besoin de types *récursifs* (infinis, cycliques). Par exemple, considérons une méthode `min` qui renvoie le plus petit de l'objet lui-même et d'un autre objet passé en paramètre:

```
obj(s) < val clé = une expression entière;
         method rang = clé;
         method min o = if s#rang <= o#rang then s else o >
```

Cet objet n'est pas typable dans le système de la section 7.1. En effet, le type de l'objet doit avoir la forme suivante:

$$\tau_s = \langle \texttt{rang} : \texttt{Pre int};\ \texttt{min} : \texttt{Pre}\ (\tau_o \to \tau_o);\ \ldots \rangle$$

mais pour que le `if...then...else` soit bien typé, il faut $\tau_s = \tau_o$, et c'est impossible avec notre algèbre de types car τ_o est un sous-terme strict de τ_s.

Pour contourner cette restriction, on peut introduire des types récursifs dans l'algèbre de types. Intuitivement, un type récursif est une expression de type infinie, comme par exemple $\texttt{int} \to \texttt{int} \to \texttt{int} \to \ldots$. Avec les types récursifs, une équation comme $\alpha \stackrel{?}{=} \tau[\alpha]$ admet une solution qui est le type infini $\tau[\tau[\tau[\ldots]]]$.

7.4.1 Présentations des types récursifs

Il y a plusieurs manières de formaliser rigoureusement les types récursifs:

Limites de suites de types finis: on voit les types récursifs comme des limites de suites d'approximations finies. Ainsi, $\texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int} \rightarrow \ldots$ est la limite de la suite $\texttt{int}$, $(\texttt{int} \rightarrow \texttt{int})$, $(\texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int})$, ... L'exercice 7.5 explore cette approche.

Arbres infinis rationnels: de même que les expressions de types normales peuvent être vues comme des arbres finis (avec des nœuds étiquetés par T, $\rightarrow$, $\times$, α, ...), on peut voir les types récursifs comme des arbres infinis. On impose que ces arbres soient rationnels, c'est-à-dire comportent un nombre fini de sous-arbres différents.

Graphes: on représente les types comme des graphes finis dont les nœuds sont étiquetés par T, $\rightarrow$, $\times$, α, ... Les types normaux correspondent à des graphes acycliques; les types récursifs, à des graphes comportant des cycles. L'exercice 7.4 explore cette approche.

Présentation syntaxique avec des μ-types: on enrichit les expressions de types comme suit:

Types: $\tau ::= \ldots \mid \mu\alpha.\ \tau \quad \text{si } \tau \neq \alpha$

Le type $\mu\alpha.\ \tau$ est une représentation finie du seul type τ' (fini ou infini) qui vérifie l'égalité $\tau' = \tau[\alpha \leftarrow \tau']$.

Par exemple, $\mu\alpha.\texttt{int} \rightarrow \alpha$ est le type infini $\texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int} \rightarrow \ldots$. C'est la seule solution de l'équation $\tau' = \texttt{int} \rightarrow \tau'$.

Dans l'exemple `min` ci-dessus, un type correct pour l'objet est

$$\mu\alpha.\langle \texttt{rang} : \texttt{Pre int};\ \texttt{min} : \texttt{Pre}(\alpha \rightarrow \alpha);\ \emptyset\rangle$$

Les types μ sont identifiés modulo la théorie équationnelle suivante:

$$\mu\alpha.\tau = \mu\beta.\tau[\alpha \leftarrow \beta] \ \text{(renommage)} \qquad\qquad \mu\alpha.\tau = \tau[\alpha \leftarrow \mu\alpha.\tau] \ \text{(déroulage)}$$

$$\frac{\tau[\alpha \leftarrow \tau_1] = \tau[\alpha \leftarrow \tau_2] \qquad \tau \neq \alpha}{\tau_1 = \tau_2} \ \text{(unicité)}$$

L'axiome (renommage) exprime que α dans $\mu\alpha.\ \tau$ est une variable liée et peut être renommée à volonter. L'axiome (déroulage) permet d'"enrouler" et de "dérouler" un type μ à volonté pour satisfaire des égalités de types. Elle capture l'idée que $\mu\alpha.\ \tau$ est une solution de l'équation $\alpha \stackrel{?}{=} \tau$. Enfin, la règle d'inférence (unicité) exprime que $\mu\alpha.\ \tau$ est la seule solution de l'équation $\alpha \stackrel{?}{=} \tau$. Donc, si τ_1 et τ_2 satisfont cette équation, ils sont forcément égaux.

Exemple: les deux types $\mu\alpha.\ \texttt{int} \rightarrow \alpha$ et $\mu\alpha.\ \texttt{int} \rightarrow \texttt{int} \rightarrow \alpha$ sont égaux, car ils sont tous deux solution de $\alpha \stackrel{?}{=} \texttt{int} \rightarrow \texttt{int} \rightarrow \alpha$: en déroulant le premier type deux fois, on a bien

$$\mu\alpha.\ \texttt{int} \rightarrow \alpha = \texttt{int} \rightarrow (\mu\alpha.\ \texttt{int} \rightarrow \alpha) = \texttt{int} \rightarrow \texttt{int} \rightarrow (\mu\alpha.\ \texttt{int} \rightarrow \alpha)$$

et en déroulant le second une fois,

$$\mu\alpha.\ \texttt{int} \rightarrow \texttt{int} \rightarrow \alpha = \texttt{int} \rightarrow \texttt{int} \rightarrow (\mu\alpha.\ \texttt{int} \rightarrow \texttt{int} \rightarrow \alpha)$$

Présentation syntaxique sous forme de types avec équations: on manipule syntaxiquement à la fois des expressions de types et des équations entre variables de types et types. Par exemple, $\mathtt{int} \rightarrow \mathtt{int} \rightarrow \ldots$ est vu comme le type α dans le contexte de l'équation $\alpha = \mathtt{int} \rightarrow \alpha$.

Quelle que soit la présentation des types récursifs retenue, on conserve exactement les mêmes règles de typage que dans le cas non récursif. Simplement, certaines contraintes d'égalité imposées par les règles sont maintenant satisfiables par des types récursifs, alors qu'elles n'avaient pas de solution avec les types normaux.

7.4.2 Sous-typage et types récursifs

Le sous-typage en présence de types récursifs est délicat. Pour déterminer $\tau <: \tau'$, si p.ex. τ est un type μ mais pas τ', il faut bien sûr dérouler le type μ pour faire apparaître le constructeur de type qui est en dessous et le comparer avec le constructeur de tête de τ'. Mais si les deux types τ, τ' sont des types μ, dérouler systématiquement les deux types mène à des dérivations de sous-typage infinies, et donc incorrectes. Par exemple, pour déterminer si

$$\mu\alpha.\langle m : \mathtt{Pre}\ \alpha;\ \beta\rangle <: \mu\alpha.\langle m : \mathtt{Pre}\ \alpha;\ \emptyset\rangle,$$

on peut en déroulant les deux types se ramener à

$$\langle m : (\mu\alpha.\langle m : \mathtt{Pre}\ \alpha;\ \beta\rangle);\ \beta\rangle <: \langle m : (\mu\alpha.\langle m : \mathtt{Pre}\ \alpha;\ \emptyset\rangle);\ \emptyset\rangle,$$

mais pour prouver ce sous-typage, il faut avoir une dérivation de

$$\mu\alpha.\langle m : \mathtt{Pre}\ \alpha;\ \beta\rangle <: \mu\alpha.\langle m : \mathtt{Pre}\ \alpha;\ \emptyset\rangle$$

et nous voilà ramenés au problème de départ.

La solution est d'ajouter une règle de sous-typage supplémentaire pour le cas de deux types μ, disant que $\mu\alpha.\ \tau <: \mu\beta.\ \tau'$ si, en faisant l'hypothèse que $\alpha <: \beta$, on peut montrer que $\tau <: \tau'$. C'est une forme de preuve par *co-induction*: si, en supposant que les deux types μ après expansion sont sous-types, on peut montrer qu'ils sont sous-types, alors ils sont sous-types. La relation de sous-typage devient $H \vdash \tau <: \tau'$, où l'environnement H est un ensemble d'hypothèses de la forme $\alpha <: \beta$. Les règles définissant le sous-typage sont:

$$H \vdash T <: T \qquad H \vdash \alpha <: \alpha \qquad \frac{H \vdash \tau' <: \tau \quad H \vdash \varphi <: \varphi'}{H \vdash \tau \rightarrow \varphi <: \tau' \rightarrow \varphi'} \qquad \frac{H \vdash \tau <: \tau' \quad H \vdash \varphi <: \varphi'}{H \vdash \tau \times \varphi <: \tau' \times \varphi'}$$

$$\frac{H \vdash \tau <: \tau'}{H \vdash \langle\tau\rangle <: \langle\tau'\rangle} \qquad H \vdash \tau <: \emptyset \qquad \frac{H \vdash \tau <: \tau' \quad H \vdash \varphi <: \varphi'}{H \vdash (m : \tau;\ \varphi) <: (m : \tau';\ \varphi')}$$

$$\mathtt{Abs} <: \mathtt{Abs} \qquad \frac{\tau <: \tau'}{Pre\ \tau <: \mathtt{Pre}\ \tau'}$$

$$\frac{(\alpha <: \beta) \in H}{H \vdash \alpha <: \beta} \qquad \frac{H \cup \{(\alpha <: \beta)\} \vdash \tau <: \tau'}{H \vdash \mu\alpha.\ \tau <: \mu\beta.\ \tau'}$$

7.4.3 Inférence en présence de types récursifs

L'introduction de types récursifs ne modifie pas l'algorithme W, mais nécessite de remplacer l'unification entre termes finis par de l'unification entre termes rationnels, c'est-à-dire de l'unification entre graphes.

L'idée de base est de faire comme dans l'algorithme d'unification entre termes, sauf qu'on supprime le test d'occurrence dans le cas $\alpha \stackrel{?}{=} \tau$: au lieu de prendre $[\alpha \leftarrow \tau]$ si $\alpha \notin \mathcal{L}(\tau)$ et d'échouer sinon, on prend dans tous les cas $[\alpha \leftarrow \mu\alpha.\ \tau]$. Par l'équation de déroulement, on a bien $\mu\alpha.\ \tau = \tau[\alpha \leftarrow \mu\alpha.\ \tau]$. Remarquez que si $\alpha \notin \mathcal{L}(\tau)$, on retrouve la même solution que dans le cas des termes finis, car alors $\mu\alpha.\ \tau = \tau$.

La difficulté est d'assurer la terminaison de l'algorithme d'unification. Pour ce faire, il faut introduire des formalismes plus complexes que ceux que nous avons employé jusqu'ici: unification entre graphes, ou bien multi-équations. L'exercice 7.4 est une introduction à l'unification entre graphes.

7.5 Inférence par contraintes de sous-typage

Dans cette partie, nous introduisons l'approche 3 (subsomption implicite et inférence de types par contraintes de sous-typage) dans un cadre simplifié où nous n'avons pas de `let` polymorphe et pas de variables de rangées dans les types d'objets.

7.5.1 Règles de typage

Algèbre de types: on considère les types suivants:

Types:	$\tau ::= \alpha \mid T \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$	comme précédemment
	$\mid \langle m_1 : \tau_1; \ldots; m_k : \tau_k \rangle$	type d'objet
	$\mid \mu\alpha.\ \tau$	type récursif
	$\mid \top$	le super-type de tous les types
	$\mid \bot$	le type vide (sous-type de tous les types)

On suppose que dans un type d'objet, les noms de méthodes m_i sont tous différents. L'ordre des composantes $m_i : \tau_i$ dans un type d'objet n'est pas significatif.

Relation de sous-typage: La relation de sous-typage est une simplification de celle de la section 7.2, avec les types récursifs traités comme expliqué dans la section 7.4.2:

$$H \vdash \bot <: \tau \qquad H \vdash \tau <: \top \qquad H \vdash T <: T \qquad H \vdash \alpha <: \alpha$$

$$\frac{H \vdash \tau' <: \tau \quad H \vdash \varphi <: \varphi'}{H \vdash (\tau \to \varphi) <: (\tau' \to \varphi')} \qquad \frac{H \vdash \tau <: \tau' \quad H \vdash \varphi <: \varphi'}{H \vdash (\tau \times \varphi) <: (\tau' \times \varphi')}$$

$$\frac{H \vdash \tau_1 <: \varphi_1 \quad \ldots \quad H \vdash \tau_k <: \varphi_k}{H \vdash \langle m_1 : \tau_1; \ldots; m_k : \tau_k; \ldots; m_n : \tau_n \rangle <: \langle m_1 : \varphi_1; \ldots; m_k : \varphi_k \rangle}$$

$$\frac{(\alpha <: \beta) \in H}{H \vdash \alpha <: \beta} \qquad \frac{H \cup \{(\alpha <: \beta)\} \vdash \tau <: \tau'}{H \vdash \mu\alpha.\ \tau <: \mu\beta.\ \tau'}$$

Proposition 7.1 *La relation $H \vdash \_ <: \_$ est transitive.*

Règles de typage: les règles de typage sont celles de mini-ML monomorphe (section 1.3.2), plus une règle pour les objets, une pour les appels de méthodes, et une règle de subsomption implicite.

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

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

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

$$\frac{i \neq j \Rightarrow m_i \neq m_j \quad E^* \vdash a_i : \tau_i \quad E^* + \{x : \langle m_1 : \tau'_1;\ \ldots;\ m_k : \tau'_k\rangle;\ \texttt{val}\ x_i : \tau_i\} \vdash a'_j : \tau'_j}{E \vdash \texttt{obj}(x)\langle \ldots; \texttt{val}\ x_i = a_i; \ldots; \texttt{method}\ m_j = a'_j; \ldots\rangle : \langle m_1 : \tau'_1;\ \ldots;\ m_k : \tau'_k\rangle} \ (\text{objet})$$

$$\frac{E \vdash a : \langle m : \tau\rangle}{E \vdash a\#m : \tau} \ (\text{meth}) \qquad \frac{E \vdash a : \tau \quad \emptyset \vdash \tau <: \tau'}{E \vdash a : \tau'} \ (\text{sub})$$

7.5.2 Construction du système de contraintes

Comme pour l'inférence de types de ML monomorphe (section 3.2), la première phase de l'inférence de types est d'associer au programme un système de contraintes entre types qui caractérise tous les typages possibles pour le programme. Dans la section 3.2, ces contraintes étaient des contraintes d'égalité; ici, ce sont des contraintes de sous-typage.

On se donne un programme (une expression close) a_0 dans laquelle tous les identificateurs liés par `fun` ou `let` ont des noms différents. À chaque identificateur x dans a_0, on associe une variable de type α_x. De même, à chaque sous-expression a de a_0, on associe une variable de type α_a.

Le système d'équations $C(a_0)$ associé à a_0 est construit en parcourant l'expression a_0 et en ajoutant des équations pour chaque sous-expression a de a_0, comme suit:

- Si a est une variable x: $C(a) = \{\alpha_x \overset{?}{<:} \alpha_a\}$.
- Si a est une constante c ou un opérateur op:
  $C(a) = \{TC(c) \overset{?}{<:} \alpha_a\}$ ou $C(a) = \{TC(op) \overset{?}{<:} \alpha_a\}$.
- Si a est $\texttt{fun}\ x \to b$: $C(a) = \{\alpha_x \to \alpha_b \overset{?}{<:} \alpha_a\} \cup C(b)$.
- Si a est une application $b\ c$: $C(a) = \{\alpha_b \overset{?}{<:} \alpha_c \to \alpha_a\} \cup C(b) \cup C(c)$.
- Si a est une paire (b, c): $C(a) = \{\alpha_b \times \alpha_c \overset{?}{<:} \alpha_a\} \cup C(b) \cup C(c)$.
- Si a est $\texttt{let}\ x = b\ \texttt{in}\ c$: $C(a) = \{\alpha_b \overset{?}{<:} \alpha_x;\ \alpha_c \overset{?}{<:} \alpha_a\} \cup C(b) \cup C(c)$.
- Si a est $\texttt{obj}(x)\langle \ldots; \texttt{val}\ x_i = a_i; \ldots; \texttt{method}\ m_j = b_j; \ldots\rangle$:
  $C(a) = \{\langle m_j : \alpha_{b_j}\rangle \overset{?}{<:} \alpha_a;\ \langle m_j : \alpha_{b_j}\rangle \overset{?}{<:} \alpha_x;\ \alpha_{a_i} \overset{?}{<:} \alpha_{x_i}\} \cup C(a_i) \cup C(b_j)$.

- Si a est $b\#m$: $C(a) = \{\alpha_b \overset{?}{<:} \langle m : \alpha_a \rangle\} \cup C(b)$.

Remarquons que $C(a)$ est un ensemble d'inéquations entre types finis (il ne contient pas de types récursifs $\mu\alpha.\ \tau$).

Exemple: on considère le programme

$$a = \texttt{fun}\ x \to \underbrace{(\underbrace{\underbrace{x}_{d}\#m}_{c}, \underbrace{\underbrace{x}_{f}\#m'}_{e})}_{b}$$

On a:

$$\begin{aligned} C(a) = \{\ & \alpha_x \to \alpha_b \overset{?}{<:} \alpha_a; \\ & \alpha_c \times \alpha_e \overset{?}{<:} \alpha_b; \\ & \alpha_d \overset{?}{<:} \langle m : \alpha_c \rangle; \\ & \alpha_x \overset{?}{<:} \alpha_d; \\ & \alpha_f \overset{?}{<:} \langle m' : \alpha_e \rangle; \\ & \alpha_x \overset{?}{<:} \alpha_f \} \end{aligned}$$

7.5.3 Lien entre typages et solutions des équations

Une solution de l'ensemble de contraintes $C(a)$ est une substitution φ (des variables de types dans les types finis ou infinis) telle que pour toute inéquation $\tau_1 \overset{?}{<:} \tau_2$ dans $C(a)$, on ait $\varphi(\tau_1) <: \varphi(\tau_2)$. Le deux propositions suivantes montrent que les solutions de $C(a)$ caractérisent exactement les typages possibles pour a.

Proposition 7.2 (Correction des solutions vis-à-vis du typage) *Si φ est une solution de $C(a)$, alors $E \vdash a : \varphi(\alpha_a)$ où E est l'environnement de typage $\{x : \varphi(\alpha_x) \mid x$ libre dans $a\}$.*

Proposition 7.3 (Complétude des solutions vis-à-vis du typage) *Soit a une expression. S'il existe un environnement E et un type τ tels que $E \vdash a : \tau$, alors le système d'équations $C(a)$ admet une solution.*

7.5.4 Cohérence d'un système de contraintes

Dans la section 3.2, nous étins capables de résoudre le système d'équations $C(a)$ par unification et d'en fournir une solution principale (qui résume toutes les solutions possibles). Ce n'est plus possible avec les systèmes d'inéquations de sous-typage.

Exemple: $\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ f(\texttt{obj}(s)\langle\texttt{method}\ m = 1\rangle)$. On peut attribuer à `f` les types $\langle \texttt{m : int} \rangle \to \langle \texttt{m : int} \rangle$ et $\langle\,\rangle \to \langle\,\rangle$ (entre autres). Cependant, aucun de ces deux types n'est plus général que l'autre. En particulier, ils sont incomparables par la relation de sous-typage.

Au lieu d'essayer de résoudre l'ensemble de contraintes $C(a)$, nous allons simplement établir qu'il existe une solution de $C(a)$ sans la calculer entièrement. Cela suffit à garantir que le programme a est bien typé et s'exécute sans erreurs.

On établit la solvabilité (l'existence d'une solution) d'un ensemble de contraintes C en deux temps: d'abord on calcule la fermeture C^* de C; ensuite, on vérifie que C^* ne contient pas d'incohérences immédiates.

Fermeture: on dit qu'un ensemble de contraintes C entre types finis est fermé (par transitivité et par propagation) s'il satisfait les conditions suivantes:

- Si $\tau_1 \overset{?}{<:} \tau_2 \in C$ et $\tau_2 \overset{?}{<:} \tau_3 \in C$, alors $\tau_1 \overset{?}{<:} \tau_3 \in C$.
- Si $(\tau_1 \to \tau_2) \overset{?}{<:} (\varphi_1 \to \varphi_2) \in C$, alors $\varphi_1 \overset{?}{<:} \tau_1 \in C$ et $\tau_2 \overset{?}{<:} \varphi_2 \in C$.
- Si $(\tau_1 \times \tau_2) \overset{?}{<:} (\varphi_1 \times \varphi_2) \in C$, alors $\tau_1 \overset{?}{<:} \varphi_1 \in C$ et $\tau_2 \overset{?}{<:} \varphi_2 \in C$.
- Si $\langle m_i : \tau_i \rangle \overset{?}{<:} \langle n_j : \varphi_j \rangle \in C$, alors $\tau_i <: \varphi_j \in C$ pour tous i, j tels que $m_i = n_j$ (i.e. pour tous les noms de méthodes qui apparaissent à la fois dans les deux types objet).

On note C^* la fermeture de C, c'est-à-dire le plus petit ensemble fermé contenant C. On l'obtient à partir de C en ajoutant à C toutes les contraintes nécessaires pour satisfaire les conditions ci-dessus. Par exemple, si $\tau_1 \overset{?}{<:} \tau_2 \in C$ et $\tau_2 \overset{?}{<:} \tau_3 \in C$ mais $\tau_1 \overset{?}{<:} \tau_3 \notin C$, on ajoute $\tau_1 \overset{?}{<:} \tau_3$ à C. Le processus termine forcément, car les contraintes ajoutées sont des contraintes entre types qui sont des sous-termes de types apparaissant dans l'ensemble initial C, et ces sous-termes sont en nombre fini.

Proposition 7.4 *Un ensemble de contraintes C est solvable si et seulement si sa fermeture C^* est solvable*

Démonstration: Si C^* est solvable, comme $C \subseteq C^*$, toute solution de C^* est aussi solution de C, donc C est solvable. Réciproquement, toute solution de C satisfait également les inéquations de C^*, car celles-ci sont des conséquences d'inéquations contenues dans C. Par exemple, si $\tau_1 \overset{?}{<:} \tau_3 \in C^*$ avec $\tau_1 \overset{?}{<:} \tau_2 \in C$ et $\tau_2 \overset{?}{<:} \tau_3 \in C$ (transitivité), si une substitution φ est telle que $\varphi(\tau_1) <: \varphi(\tau_2)$ et $\varphi(\tau_2) <: \varphi(\tau_3)$, alors par transitivité de $<:$ on a $\varphi(\tau_1) <: \varphi(\tau_3)$. □

Cohérence immédiate: un ensemble de contraintes C est immédiatement cohérent si toutes les contraintes $\tau_1 \overset{?}{<:} \tau_2$ dans C tombent dans l'un des cas suivants:

- τ_1 ou τ_2 sont des variables de types;
- $\tau_1 = \bot$;
- $\tau_2 = \top$;
- τ_1 et τ_2 sont des types flèches;
- τ_1 et τ_2 sont des types produits;

- $\tau_1 = \langle m_i : \varphi_i \rangle$, $\tau_2 = \langle n_j : \psi_j \rangle$, et l'ensemble de noms de méthodes $\{n_j\}$ est inclus dans l'ensemble des noms de méthodes $\{m_i\}$.

Un ensemble qui n'est pas immédiatement cohérent est dit immédiatement incohérent.

Proposition 7.5 *Un ensemble de contraintes immédiatement incohérent n'est pas solvable.*

Démonstration: si $\tau_1 \overset{?}{<:} \tau_2$ est une équation incohérente de C, pour toute substitution φ, on ne peut pas dériver $H \vdash \varphi(\tau_1) <: \varphi(\tau_2)$ car cet énoncé n'a la forme d'aucune conclusion d'une des règles d'inférence définissant $<:$. $\square$

La réciproque de la proposition précédente n'est pas vraie si C n'est pas fermé. Par exemple, $\{\texttt{int} \rightarrow \texttt{int} \overset{?}{<:} \alpha;\ \alpha \overset{?}{<:} \texttt{int} \times \texttt{int}\}$ est immédiatement cohérent, mais n'admet pas de solutions. Cependant, si C est fermé, nous avons le théorème suivant (que nous admettrons):

Proposition 7.6 *Un ensemble de contraintes C fermé et immédiatement cohérent est solvable.*

7.5.5 Algorithme d'inférence de types

En combinant les résultats précédents, nous obtenons l'algorithme d'inférence de types $I(a)$ suivant:

Entrée: une expression close a.
Sortie: "bien typé" ou "non typable".
Calcul: si $C(a)^*$ est immédiatement incohérent, renvoyer "non typable". Sinon, renvoyer "bien typé".

Proposition 7.7 (Correction et complétude de l'inférence) $I(a)$ *renvoie "bien typé" si et seulement si il existe τ tel que $\emptyset \vdash a : \tau$.*

Démonstration: la partie "si" découle des propositions 7.2, 7.4 et 7.5. La partie "seulement si" est corollaire des propositions 7.3, 7.4 et 7.6. $\square$

Lectures pour aller plus loin

- Martín Abadi et Luca Cardelli, *A theory of objects*, Springer-Verlag Monographs in Computer Science, 1996.
  Un livre entier sur les calculs d'objets et leurs systèmes de types (sans inférence).

- Didier Rémy et Jérôme Vouillon, *Objective ML: An effective object-oriented extension to ML*, Theory And Practice of Object Systems, 4(1):27–50, 1998, `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/objective-ml!tapos98.ps.gz`
  L'approche Objective Caml.

- Roberto Amadio et Luca Cardelli, *Subtyping recursive types*, ACM Transactions on Programming Languages and Systems, 15(4), 1993, `http://research.microsoft.com/Users/luca/Papers/SRT.A4.ps`
  La référence sur le sous-typage entre types récursifs.

Exercices

Exercice 7.1 *(*)/(**) Montrer que la relation de sous-typage (sans types récursifs) est une relation d'ordre: réflexive ($\tau <: \tau$), transitive ($\tau_1 <: \tau_2$ et $\tau_2 <: \tau_3$ implique $\tau_1 <: \tau_3$) et antisymétrique ($\tau <: \tau'$ et $\tau' <: \tau$ impliquent $\tau = \tau'$).*

Exercice de programmation 7.1 *Implémenter le prédicat $<:$ de la section 7.2. (Indication: attention à l'équation de commutativité sur les rangées.)*

Exercice 7.2 *(**)/(***) Montrer le cas β_{fun} de la préservation du typage par réduction de tête (l'analogue de la proposition 2.3) lorsqu'on ajoute la règle (sub). (Indication: attention, il y a un piège.)*

Exercice 7.3 *(*) Montrer que les types récursifs permettent de typer en ML tous les termes du λ-calcul pur. Quel est le type qu'ont tous les λ-termes?*

Exercice 7.4 *(**)/(***) Le but de cet exercice est de comprendre la représentation des types récursifs par des graphes et de programmer l'algorithme d'unification correspondant.*

Un graphe de types est un graphe orienté. Chaque noeud est étiqueté ou bien par le symbole V (signifiant que ce noeud représente une variable de type), ou bien par un constructeur de type comme `int`, `bool`, $\rightarrow$, $\times$. *(Pour simplifier, on ne considère pas les types objets.) Chaque noeud doit avoir un nombre de noeuds fils égal à l'arité $A(c)$ de son étiquette c, avec bien sûr $A(V) = A(\texttt{int}) = A(\texttt{bool}) = A(\emptyset) = 0$ (pas de fils), et $A(\rightarrow) = A(\times) = 2$ (deux fils). Un noeud du graphe représente donc une expression de type dont le constructeur de tête est donné par l'étiquette du type et dont les sous-expressions sont représentées par les fils éventuels de ce noeud. Les types récursifs apparaissent naturellement sous forme de cycles dans le graphe.*

Si n est un noeud du graphe, on note $C(n)$ son constructeur de type et $F_i(n)$ le noeud du i^e fils de n (avec $1 \leq i \leq A(C(n))$).

Une substitution, dans ce formalisme, est une relation d'équivalence R entre les noeuds du graphe qui vérifie les propriétés suivantes:

- *(Compatibilité des constructeurs) Si $n \ R \ n'$ et $C(n) \neq V$ et $C(n') \neq V$, alors $C(n) = C(n')$. (Autrement dit, une subsitution ne peut rendre égaux que des noeuds qui ont le même constructeur, sauf si l'un des deux est une variable; un noeud variable peut être rendu égal à n'importe quel noeud.)*
- *(Fermeture) Si $n \ R \ n'$ et $C(n) \neq V$ et $C(n') \neq V$, alors $F_i(n) \ R \ F_i(n')$ pour $1 \leq i \leq A(C(n))$. (Autrement dit, une substitution qui égalise deux noeuds non variables doit aussi égaliser leurs fils deux à deux.)*

Un ensemble d'équations sur un graphe de types est un ensemble de paires de noeuds $\{n \stackrel{?}{=} n'; n_1 \stackrel{?}{=} n'_1; \ldots\}$. Un unificateur d'un ensemble d'équations E est une substitution R telle que $n \ R \ n'$ pour tout $(n \stackrel{?}{=} n') \in E$. L'unificateur R est principal si tout autre unificateur R' est une relation moins fine que R, c'est-à-dire $n_1 \ R \ n_2 \Rightarrow n_1 \ R' \ n_2$.

L'algorithme suivant calcule l'unificateur principal d'un ensemble d'équations E. Il prend en entrée une relation d'équivalence R qui représente des identifications entre noeuds déjà effectuées, *et retourne une relation d'équivalence qui est l'unificateur principal.*

$$\begin{array}{rcl}
\mathtt{mgu}(\emptyset, R) & = & R \\
\mathtt{mgu}(\{n \stackrel{?}{=} n'\} \cup E, R) & = & \mathtt{mgu}(E, R) \text{ si } n\ R\ n' \\
\mathtt{mgu}(\{n \stackrel{?}{=} n'\} \cup E, R) & = & \mathtt{mgu}(E, R + \{n = n'\}) \text{ si } C(n) = V \text{ ou } C(n') = V \\
\mathtt{mgu}(\{n \stackrel{?}{=} n'\} \cup E, R) & = & \mathtt{mgu}(E \cup \{F_1(n) \stackrel{?}{=} F_1(n'); \ldots; F_k(n) \stackrel{?}{=} F_k(n')\}, R + \{n = n'\}) \\
& & \text{si } C(n) \neq V \text{ et } C(n') = C(n) \text{ et } k = A(C(n)) \\
\mathtt{mgu}(\{n \stackrel{?}{=} n'\} \cup E, R) & = & \text{échec si } C(n) \neq V \text{ et } C(n') \neq C(n)
\end{array}$$

On a noté $R + \{n = n'\}$ la plus fine relation d'équivalence qui contient R et qui relie n et n'. Elle s'obtient à partir de R en fusionnant les classes d'équivalence de n et n' dans R.

1) Dessiner les graphes représentant les types suivants:
$\mathtt{int} \rightarrow \mathtt{bool}$; $\alpha \times \beta$; $\alpha \times \alpha$; $\mu\alpha.\ \mathtt{int} \rightarrow \alpha$; $\mu\alpha.\ \beta \rightarrow \gamma \rightarrow \alpha$.

2) Faire tourner à la main l'algorithme sur la représentation sous forme de graphe du problème d'unification $\mu\alpha.\ \mathtt{int} \rightarrow \alpha \stackrel{?}{=} \mu\alpha.\ \beta \rightarrow \gamma \rightarrow \alpha$. (On partira de la relation identité comme second paramètre initial de `mgu`*.) Quelle est la forme de la substitution renvoyée?*

3) Montrer que si $R = \mathtt{mgu}(E, id)$ (où id est la relation identité) est définie, alors R est une substitution. Montrer que R est un unificateur de E. Montrer que R est un unificateur principal de E.

4) Montrer que l'algorithme `mgu` *termine toujours.*

Exercice de programmation 7.2 *Implémenter l'algorithme d'unification de graphes de l'exercice précédent. (Indication: on représentera la relation d'équivalence entre noeuds par une structure de type* union-find*. C'est-à-dire, on placera dans chaque noeud un champ mutable de type* `noeud option`*, dont la signification est la suivante: si* `None`*, cela veut dire que le noeud n'est encore pas identifié à un autre noeud; si* `Some`(n)*, cela veut dire que le noeud est dans la même classe d'équivalence que le noeud n. Au lieu de renvoyer une relation d'équivalence comme résultat de l'unification, on modifiera en place ces champs mutables types pour représenter cette relation.)*

Exercice 7.5 *(***, pour mathématiciens) Le but de cet exercice est de justifier mathématiquement l'existence des types $\mu\alpha.\ \tau$. L'idée est de construire les types infinis comme limites de suites convergentes de types finis, de même que Cantor construisit les réels comme limites de suites convergentes de rationnels.*

1) Pour simplifier, on considère uniquement l'algèbre de types $\tau ::= \alpha \mid \tau_1 \rightarrow \tau_2$. On définit la distance $d(\tau_1, \tau_2)$ entre deux types (finis) τ_1 et τ_2 de la façon suivante:

$$\begin{array}{rcl}
d(\tau_1 \rightarrow \varphi_1, \tau_2 \rightarrow \varphi_2) & = & \frac{1}{2} \max(d(\tau_1, \tau_2), d(\varphi_1, \varphi_2)) \\
d(\alpha, \alpha) & = & 0 \\
d(\alpha, \beta) & = & 1 \text{ si } \alpha \neq \beta \\
d(\alpha, \tau_2 \rightarrow \varphi_2) & = & 1 \\
d(\tau_1 \rightarrow \varphi_1, \beta) & = & 1
\end{array}$$

Montrer que d est une distance ultramétrique, c'est-à-dire qu'elle satisfait l'inégalité du triangle ultramétrique $d(\tau_1, \tau_3) \leq \max(d(\tau_1, \tau_2), d(\tau_2, \tau_3))$, qu'elle est symétrique, et que de plus $d(\tau_1, \tau_2) = 0$ si et seulement si $\tau_1 = \tau_2$.

2) (Complétion d'un espace métrique.) On considère l'ensemble $\mathcal{S}$ dont les éléments sont des suites de Cauchy de types $(\tau_n)_{n\in\mathbf{N}}$. On rappelle qu'une suite (τ_n) est de Cauchy si

$$\forall\varepsilon.\exists n.\forall p,q \geq n.\ d(\tau_p,\tau_q) \leq \varepsilon.$$

On définit la distance $d(s,s')$ entre deux telles suites $s = (\tau_n)$ et $s' = (\tau'_n)$ par

$$d(s,s') = \lim_{n\to\infty} d(\tau_n,\tau'_n).$$

a) Montrer que cette limite existe toujours. b) Montrer que la relation entre suites $\cong$ définie par $s \cong s' \Leftrightarrow d(s,s') = 0$ est une relation d'équivalence. c) Montrer que l'ensemble quotient $\mathcal{T} = \mathcal{S}/\cong$ muni de la distance $d/\cong$ est un espace ultramétrique. d) Montrer que les types simples se plongent naturellement dans $\mathcal{T}$ et que les distances sont préservées par le plongement. e) Montrer ou admettre que $\mathcal{T}$ est complet (toute suite de Cauchy d'éléments de $\mathcal{T}$ converge vers un élement de $\mathcal{T}$).

3) (Théorème de Banach-Tarski.) Soit (E,d) un espace métrique complet. Une fonction F de E dans E est dite contractive s'il existe une constante $k\in\,]0,1[$ telle que $d(F(x),F(y)) \leq k\,d(x,y)$ pour tous $x,y\in E$. Montrer que toute fonction contractive admet un point fixe et que ce point fixe est unique.

4) Soit α une variable de type et $\tau\in\mathcal{T}$ tel que $\tau\neq\alpha$. Montrer qu'il existe un et un seul $\tau'\in\mathcal{T}$ tel que $\tau' = \tau[\alpha\leftarrow\tau']$.

Chapter 8

Systèmes de modules

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.

8.1 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)
	$\mid \texttt{struct}\ d_1; \ldots; d_n\ \texttt{end}$	définition de structure
	$\mid \texttt{functor}\ (X : M) \rightarrow m$	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:	$M ::= \texttt{sig}\ S_1; \ldots; S_n\ \texttt{end}$	signature
	$\mid \texttt{functor}\ (X : M) \rightarrow M'$	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
	$\mid x \mid c \mid op \mid \texttt{fun}\ x \rightarrow a \mid a_1\ a_2$	
	$\mid (a_1, a_2) \mid \texttt{let}\ x = a_1\ \texttt{in}\ a_2$	comme d'habitude

Types:	$\tau ::= t$	type nommé
	$\mid p.t$	composante de type d'une structure
	$\mid \alpha \mid T \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2$	comme d'habitude
Schémas de types:	$\sigma ::= \forall \alpha_1 \ldots \alpha_n.\ \tau$	

8.2 Évaluation

8.2.1 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:

$$\begin{array}{rcl}
[\![X]\!] & = & X \\
[\![p.X]\!] & = & [\![p]\!].X \\
[\![\texttt{struct}\ d^*\ \texttt{end}]\!] & = & L(d^*, \{\ldots x_i = x_i; \ldots; X_j = X_j; \ldots\}) \\
& & \text{où les } x_i \text{ et } X_j \text{ sont les identificateurs de valeurs} \\
& & \text{et de modules définis dans } d^* \\
[\![\texttt{functor}\ (X : M) \rightarrow m]\!] & = & \texttt{fun}\ X \rightarrow [\![m]\!] \\
[\![m\ m']\!] & = & [\![m]\!]\ [\![m']\!]
\end{array}$$

$$\begin{array}{rcl}
L(\varepsilon, a) & = & a \\
L(\texttt{let}\ x = a;\ d^*, b) & = & \texttt{let}\ x = a\ \texttt{in}\ L(d^*, b) \\
L(\texttt{type}\ t = T;\ d^*, b) & = & L(d^*, b) \\
L(\texttt{module}\ X = m;\ d^*, b) & = & \texttt{let}\ X = [\![m]\!]\ \texttt{in}\ L(d^*, b)
\end{array}$$

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 }
```

8.2.2 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. `struct ... end`: on obtient l'expression de type (`struct ... 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:

$$
\begin{array}{rcll}
(\texttt{struct}\ d_V^*; \texttt{val}\ x = v; d_V^{*\prime}\ \texttt{end}) & \xrightarrow{\varepsilon} & v & \text{(proj-val)} \\
(\texttt{struct}\ d_V^*; \texttt{module}\ X = V; d_V^{*\prime}\ \texttt{end}) & \xrightarrow{\varepsilon} & V & \text{(proj-mod)} \\
(\texttt{functor}\ (X : M) \rightarrow m)\ V & \xrightarrow{\varepsilon} & m[X \leftarrow V] & \beta_{\text{functor}} \\
(V : M) & \xrightarrow{\varepsilon} & V & \text{(contrainte)}
\end{array}
$$

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:

$$
\begin{array}{rcll}
\texttt{struct}\ d_V^*; \texttt{let}\ x = v; d^*\ \texttt{end} & \xrightarrow{\varepsilon} & \texttt{struct}\ d_V^*; \texttt{let}\ x = v; d^*[x \leftarrow v]\ \texttt{end} & \text{(prop-val)} \\
\texttt{struct}\ d_V^*; \texttt{module}\ X = V; d^*\ \texttt{end} & \xrightarrow{\varepsilon} & \texttt{struct}\ d_V^*; \texttt{module}\ X = V; d^*[X \leftarrow V]\ \texttt{end} & \text{(prop-mod)}
\end{array}
$$

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:

$$\Gamma_a ::= [\,] \mid \Gamma_m.x \mid \Gamma_a\ a \mid v\ \Gamma_a \mid (\Gamma_a, a) \mid (v, \Gamma_a) \mid \texttt{let}\ x = \Gamma_a\ \texttt{in}\ a$$

8.3 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 : `sig type` $t =$ `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.

8.3.1 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.

8.3.2 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:

$$\frac{\tau \leq TC(c)}{E \vdash c : \tau}\ \text{(const-inst)} \qquad \frac{\tau \leq TC(op)}{E \vdash op : \tau}\ \text{(op-inst)} \qquad \frac{E;\ \texttt{val}\ x : \tau_1 \vdash a : \tau_2}{E \vdash (\texttt{fun}\ x \rightarrow a) : \tau_1 \rightarrow \tau_2}\ \text{(fun)}$$

$$\frac{E \vdash a_1 : \tau' \rightarrow \tau \qquad E \vdash a_2 : \tau'}{E \vdash a_1\ a_2 : \tau}\ \text{(app)} \qquad \frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2}\ \text{(paire)}$$

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

Remarquons que pendant le typage d'une expression a, on n'ajoute pas d'hypothèses `type t`... ou `module` $X : \ldots$ 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` $\ldots$ `val` $x = a \ldots$ `end` est possible, en utilisant l'algorithme W et l'unification modulo la relation $\approx$.

8.3.3 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 `val` $x : \sigma$ et `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:

$$\texttt{(sig type t = int end)} \not<: \texttt{(sig type t = int val v : t end)}$$

Erreur sur le nom d'une composante:

$$\texttt{(sig type t = int val n : t end)} \not<: \texttt{(sig type t = int val v : t end)}$$

Désaccord sur la définition d'un type manifeste:

$$\texttt{(sig type t = int end)} \not<: \texttt{(sig type t = bool end)}$$

Un type abstrait n'est pas prouvablement égal à un type manifeste:

$$\texttt{(sig type t end)} \not<: \texttt{(sig type t = bool end)}$$

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: $E \vdash (\texttt{functor } (X : P_1) \to R_1) <: (\texttt{functor } (X : P_2) \to R_2)$ 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

$$\begin{aligned}&(\texttt{functor } (X : \texttt{sig type } t \texttt{ end}) \to \texttt{sig type } t = X.t \texttt{ end})\\ &\quad <: \quad (\texttt{functor } (X : \texttt{sig type } t = \texttt{int end}) \to \texttt{sig type } t = \texttt{int end})\end{aligned}$$

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 $X : \texttt{sig type } t \texttt{ end}$.

8.3.4 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` ← `E`], c'est-à-dire `sig type t = E.t`. Il s'ensuit que `B.t` ≈ `E.t` par la règle (eq-projection), et comme d'autre part `E.t` ≈ `int` d'après la signature de `E`, on a bien `B.t` ≈ `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)     ⇒    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:

$$\dfrac{\dfrac{E \vdash \mathtt{Id} : \mathtt{functor}(\mathtt{X} : \mathtt{sig\ type\ t\ end}) \to \mathtt{sig\ type\ t} = \mathtt{X.t\ end}}{E \vdash \mathtt{Id} : \mathtt{functor}(\mathtt{X} : \mathtt{sig\ type\ t} = \mathtt{int\ end}) \to \mathtt{sig\ type\ t} = \mathtt{int\ end}} \quad E \vdash (\mathtt{struct\ type\ t} = \mathtt{int\ end}) : (\mathtt{sig\ type\ t} = \mathtt{int\ end})}{E \vdash \mathtt{Id}(\mathtt{struct\ type\ t} = \mathtt{int\ end}) : (\mathtt{sig\ type\ t} = \mathtt{int\ end})}$$

La règle de renforcement

Soit p un chemin désignant une structure avec un type t abstrait:

$$p : \mathtt{sig}\ \ldots ; \mathtt{type}\ t ; \ldots\ \mathtt{end}$$

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:

$$p : \mathtt{sig}\ \ldots ; \mathtt{type}\ t = p.t ; \ldots\ \mathtt{end}$$

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:

$$\begin{array}{rcl}
(\mathtt{sig}\ d_1; \ldots ; d_n\ \mathtt{end})/p & = & \mathtt{sig}\ d_1/p; \ldots ; d_n/p\ \mathtt{end} \\
(\mathtt{val}\ v : \tau)/p & = & \mathtt{val}\ v : \tau \\
(\mathtt{type}\ t)/p & = & \mathtt{type}\ t = p.t \\
(\mathtt{type}\ t = \tau)/p & = & \mathtt{type}\ t = \tau \\
(\mathtt{module}\ X : M)/p & = & \mathtt{module}\ X : (M/(p.X)) \\
(\mathtt{functor}(X : M) \to M')/p & = & \mathtt{functor}(X : M) \to M'
\end{array}$$

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 8.1 *(*)/(**) Montrer que la relation de sous-typage* $<:$ *est réflexive (*) et transitive (**).*

Exercice 8.2 *(*) 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 8.1 *Implémenter la relation de sous-typage* $<:$.

Exercice 8.3 *(***) 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.)*

Appendix A

Corrigés des exercices du chapitre 1

Exercice 1.1 La construction ML `let rec` $f\ x = a_1$ `in` a_2 peut être vue comme du "sucre syntaxique" pour

$$\texttt{let}\ f = \texttt{fix}(\texttt{fun}\ f \rightarrow \texttt{fun}\ x \rightarrow a_1)\ \texttt{in}\ a_2$$

Exercice 1.2 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 1.1.

Exercice 1.3 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 $k = 1 \ldots n$. 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

$$(P \wedge Q)(b_j^1(x)) \wedge \ldots \wedge (P \wedge Q)(b_j^n(x)) \Rightarrow (P \wedge Q)(c_j(x))$$

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 1.4 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 $D(b_j^1(x)) \wedge \ldots \wedge D(b_j^n(x))$, cela signifie que nous avons des dérivations de $P(b_j^1(x)) \ldots P(b_j^n(x))$; on peut donc construire une dérivation de noeud racine $P(c_j(x))$ et de fils les dérivations de $P(b_j^1(x)) \ldots P(b_j^n(x))$, 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 1.5 Pour $a = 1\ 2$, une dérivation de $a \stackrel{v}{\rightarrow} v$ devrait se terminer par une des règles 4, 7, 8 ou 9. 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 $a \stackrel{v}{\rightarrow} v$ pour tout v.

Pour $a' = (\texttt{fun}\ f \rightarrow f\ f)\ (\texttt{fun}\ f \rightarrow f\ f)$, notons $b = (\texttt{fun}\ f \rightarrow f\ f)$. Une dérivation de $a' \stackrel{v}{\rightarrow} v$ doit nécessairement se terminer ainsi:

$$\frac{b \stackrel{v}{\rightarrow} b \qquad b \stackrel{v}{\rightarrow} b \qquad (f\ f)[f \leftarrow b] \stackrel{v}{\rightarrow} v}{b\ b \stackrel{v}{\rightarrow} v}$$

Mais $(f\ f)[f \leftarrow b] = b\ b = a'$, donc toute dérivation de $a' \stackrel{v}{\rightarrow} v$ doit contenir une sous-dérivation de $a' \stackrel{v}{\rightarrow} v$; 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 1.6 **Cas** $a = \texttt{let}\ x = a_1\ \texttt{in}\ a_2$. La seule règle qui s'applique est 6, et donc D est de la forme

$$\frac{\begin{matrix}(D_1)\\ \vdots \\ a_1 \stackrel{v}{\rightarrow} v_1\end{matrix} \qquad \begin{matrix}(D_2)\\ \vdots \\ a_2[x \leftarrow v_1] \stackrel{v}{\rightarrow} v_2\end{matrix}}{a_1\ a_2 \stackrel{v}{\rightarrow} v_2}$$

Vu la forme de a, D' se termine nécessairement par la règle 6 elle aussi. Donc, D' contient des sous-dérivations $D'_1 : a_1 \stackrel{v}{\rightarrow} v'_1$ et $D'_2 : a_2[x \leftarrow v'_1] \stackrel{v}{\rightarrow} v'_2$ 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 1.7 Pour typer 1 2, il faudrait pouvoir attribuer à 1 un type flèche $\tau_1 \rightarrow \tau_2$, ce qui est bien sûr impossible car 1 a le type `int` dans tous les environnements de typage.

Pour typer `fun` $f \rightarrow f\ f$, il faudrait construire une dérivation de la forme suivante:

$$\dfrac{\dfrac{E + \{f : \tau_1\} \vdash f : \tau_1 \rightarrow \tau_2 \qquad E + \{f : \tau_1\} \vdash f : \tau_2}{E + \{f : \tau_1\} \vdash f\ f : \tau_2}}{E \vdash \texttt{fun}\ f \rightarrow f\ f : \tau_1 \rightarrow \tau_2}$$

Pour que les feuilles de la dérivation soient justifiées par l'axiome (var), il faudrait que $\tau_1 = \tau_1 \rightarrow \tau_2$ 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 `let` $f =$ `fun` $x \rightarrow x$ `in` $(f\ 1, f\ \texttt{true})$, nous pouvons attribuer à `fun` $x \rightarrow x$ le type $\tau \rightarrow \tau$ pour n'importe quel τ. Mais pour que f 1 soit bien typé, il faudrait prendre $\tau =$ `int`, et pour que f `true` soit bien typé, il faudrait prendre $\tau =$ `bool`, et il est impossible de satisfaire ces deux contraintes simultanément.

Exercice 1.8 Notons φ la substitution $[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n]$. 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 $\gamma \notin \{\alpha_1, \ldots, \alpha_n\}$, 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 $\sigma = \forall \alpha_1, \ldots, \alpha_n. \tau$ et $\sigma' = \forall \beta_1, \ldots, \beta_n. \tau'$ égaux à alpha-conversion près. On a donc $\tau' = \tau[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n]$, et de plus $\beta_i \notin \mathcal{L}(\sigma)$ pour tout i. Par ce qui précède,

$$\mathcal{L}(\tau') = \mathcal{L}(\tau)[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n]$$

Écrivons $\mathcal{L}(\tau) = \{\alpha_{i_1}, \ldots, \alpha_{i_p}, \gamma_1, \ldots, \gamma_q\}$ 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'.

$$\begin{array}{rcl}
\mathcal{L}(\sigma) & = & \mathcal{L}(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\} \\
& = & \{\gamma_1, \ldots, \gamma_q\} \\
\mathcal{L}(\sigma') & = & \mathcal{L}(\tau') \setminus \{\beta_1, \ldots, \beta_n\} \\
& = & (\mathcal{L}(\tau)[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n]) \setminus \{\beta_1, \ldots, \beta_n\} \\
& = & (\{\alpha_{i_1}, \ldots, \alpha_{i_p}, \gamma_1, \ldots, \gamma_q\}[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n]) \setminus \{\beta_1, \ldots, \beta_n\} \\
& = & \{\beta_{i_1}, \ldots, \beta_{i_p}, \gamma_1, \ldots, \gamma_q\} \setminus \{\beta_1, \ldots, \beta_n\} \\
& = & \{\gamma_1, \ldots, \gamma_q\}
\end{array}$$

Pour la dernière égalité, on s'appuie sur le fait que $\{\gamma_1 \ldots \gamma_q\} \cap \{\beta_1 \ldots \beta_n\} = \emptyset$ car sinon l'une des β_i serait libre dans σ. D'où $\mathcal{L}(\sigma) = \mathcal{L}(\sigma')$. CQFD.

Exercice 1.9 Pour le terme `let` $f =$ `fun` $x \to x$ `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 `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:

$$\dfrac{\dfrac{\dfrac{\tau \to \tau_2 \leq \tau_1}{\{f : \tau_1\} \vdash f : \tau \to \tau_2} \qquad \dfrac{\tau \leq \tau_1}{\{f : \tau_1\} \vdash f : \tau}}{\{f : \tau_1\} \vdash f\ f : \tau_2}}{\emptyset \vdash \texttt{fun}\ f \to f\ f : \tau_1 \to \tau_2}$$

pour des types τ, τ_1, τ_2 bien choisis. Cependant, $\tau \to \tau_2 \leq \tau_1$ implique $\tau \to \tau_2 = \tau_1$, et de même $\tau \leq \tau_1$ implique $\tau = \tau_1$. Donc, il faudrait que $\tau \to \tau_2 = \tau$, et ceci est impossible pour tout τ (un terme fini ne peut pas se contenir comme sous-terme).

Exercice 1.10 Considérons $a =$ `fun` $x \to$ `let` $y = x$ `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 $\alpha \to \beta$ où α et β sont deux variables de types distinctes. Ce n'est bien sûr pas un type correct pour la fonction identité.

Exercice 1.11 Commençons par définir l'image $\varphi(\sigma)$ d'un schéma $\sigma = \forall\alpha_1 \ldots \alpha_n.\ \tau$ par une substitution φ. Ensuite, $\varphi(E)$ sera simplement l'application point à point de φ sur les schémas contenus dans E, c'est-à-dire $\varphi(E) = E'$ est tel que $E'(x) = \varphi(E(x))$ pour tout $x \in \mathrm{Dom}(E)$. 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 alpha-conversion (renommage des variables liées): suivant les rennomages que l'on effectue sur les variables liées du schéma argument, on obtient des schémas résultats différents, même à alpha-conversion près.

L'idée est de forcer le rennomage "qui va bien" dans le schéma argument afin d'éviter toute interférence entre les variables liées et la substitution. On prend 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 $E \vdash a : \tau$ et par cas sur la dernière règle de typage utilisée.

Cas règle (var-inst). Nous avons $E \vdash x : \tau$ avec $\tau \leq E(x)$. Écrivons $E(x) = \forall \alpha_1 \ldots \alpha_n . \tau_x$. 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:

$$(\varphi(E))(x) = \varphi(E(x)) = \forall \alpha_1 \ldots \alpha_n . \varphi(\tau_x)$$

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 $\varphi(\tau) \leq (\varphi(E))(x)$. Par conséquent, la règle (var-inst) nous permet de conclure que $\varphi(E) \vdash x : \varphi(\tau)$, 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 $E \vdash (\texttt{fun}\ x \rightarrow a) : \tau_1 \rightarrow \tau_2$ en conséquence de la prémisse $E + \{x : \tau_1\} \vdash a : \tau_2$. Appliquant l'hypothèse de récurrence à cette prémisse, nous obtenons une dérivation de $\varphi(E + \{x : \tau_1\}) \vdash a : \varphi(\tau_2)$, c'est-à-dire $\varphi(E) + \{x : \varphi(\tau_1)\} \vdash a : \varphi(\tau_2)$. Par application de la règle (fun), nous concluons $\varphi(E) \vdash (\texttt{fun}\ x \rightarrow a) : \varphi(\tau_1) \rightarrow \varphi(\tau_2)$, ce qui est le résultat désiré.

Cas règle (app) ou (paire). Même raisonnement que pour la règle (fun).

Cas règle (let). C'est là que les choses se compliquent. Nous avons donc $E \vdash (\texttt{let}\ x = a_1\ \texttt{in}\ a_2) : \tau_2$ en conséquence des prémisses $E \vdash a_1 : \tau_1$ et $E + \{x : Gen(\tau_1, E)\} \vdash a_2 : \tau_2$. Le problème est que les variables généralisées par Gen, c'est-à-dire $\{\alpha_1, \ldots, \alpha_n\} = \mathcal{L}(\tau_1) \setminus \mathcal{L}(E)$, ne sont pas forcément hors de portée de φ, et donc on n'a pas, en général,

$$\varphi(Gen(\tau_1, E)) = Gen(\varphi(\tau_1), \varphi(E)).$$

(Exemple: on peut avoir $\{y : \alpha\} \vdash a_1 : \beta \rightarrow \beta$, et β est généralisable, mais après application de la substitution $[\beta \leftarrow \alpha]$, on se retrouve avec $\{y : \alpha\} \vdash a_1 : \alpha \rightarrow \alpha$, dans lequel α n'est pas généralisable!)

L'astuce est d'arriver à renommer les variables généralisées α_i afin qu'elles soient hors de portée de φ. Pour ce faire, on va appliquer l'hypothèse de récurrence à $E \vdash a_1 : \tau_1$ et non pas à la substitution φ, mais à une substitution ψ "proche" de φ mais contournant les problèmes de capture.

Plus précisément, on se donne des variables $\beta_1, \ldots, \beta_n$ non libres dans E et hors de portée de φ, et on considère la substitution

$$\psi = \varphi \circ [\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n]$$

Par application de l'hypothèse de récurrence, on a $\psi(E) \vdash a_1 : \psi(\tau_1)$. Comme les β_i ne sont pas libres dans E, on a $[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n](E) = E$ et donc $\psi(E) = \varphi(E)$. La dérivation obtenue par récurrence prouve donc

$$\varphi(E) \vdash a_1 : \psi(\tau_1)$$

On applique également l'hypothèse de récurrence à la seconde prémisse, avec la substitution φ cette fois-ci. On obtient une dérivation de

$$\varphi(E) + \{x : \varphi(Gen(\tau_1, E))\} \vdash a_2 : \varphi(\tau_2)$$

Pour conclure le résultat attendu par application de la règle (let-gen), il reste donc à montrer que

$$Gen(\psi(\tau_1), \varphi(E)) = \varphi(Gen(\tau_1, E)).$$

Les variables libres dans $\psi(\tau_1)$ mais pas dans $\varphi(E)$ sont exactement $\{\beta_1, \ldots, \beta_n\}$. En effet, les variables libres dans τ_1 sont

$$\alpha_1, \ldots, \alpha_n, \text{et des variables libres dans } E$$

Lorsque l'on applique le renommage $[\alpha_i \leftarrow \beta_i]$, ces variables libres deviennent

$$\beta_1, \ldots, \beta_n, \text{et des variables libres dans } E$$

Enfin, lorsqu'on applique φ, ces variables libres deviennent

$$\beta_1, \ldots, \beta_n, \text{et des variables libres dans } \varphi(E)$$

Les β_i ne sont pas libres dans $\varphi(E)$, car par hypothèse β_i hors de portée de φ, cela impliquerait β_i libre dans E, contredisant l'hypothèse β_i non libre dans E. Par conséquent,

$$Gen(\psi(\tau_1), \varphi(E)) = \forall \beta_1, \ldots, \beta_n.\psi(\tau_1)$$

Or, $Gen(\tau_1, E) = \forall \alpha_1, \ldots, \alpha_n.\tau_1 = \forall \beta_1, \ldots, \beta_n.\tau_1[\alpha_i \leftarrow \beta_i]$ à alpha-conversion près. De plus, les β_i sont hors de portée de φ, donc:

$$\varphi(Gen(\tau_1, E)) = \forall \beta_1, \ldots, \beta_n.\varphi(\tau_1[\alpha_i \leftarrow \beta_i]) = \forall \beta_1, \ldots, \beta_n.\psi(\tau_1)$$

Nous avons donc montré $\varphi(Gen(\tau_1, E)) = Gen(\psi(\tau_1), \varphi(E))$, et le résultat attendu en découle.

Appendix 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):

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

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

Exercice 2.2 On montre d'abord l'implication $(a \stackrel{v}{\to} v) \Rightarrow (a \stackrel{*}{\to} v)$ par récurrence sur la dérivation de l'évaluation $a \stackrel{v}{\to} v$ 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:

$$\frac{a \stackrel{v}{\to} (\texttt{fun}\ x \to c) \qquad b \stackrel{v}{\to} v_2 \qquad c[x \leftarrow v_2] \stackrel{v}{\to} v}{a\ b \stackrel{v}{\to} v}$$

Par hypothèse de récurrence appliquée aux trois sous-dérivations des prémisses, nous savons qu'il existe des séquences de réductions de la forme

$$\begin{array}{l} a \rightarrow a_1 \rightarrow \ldots \rightarrow a_n \rightarrow v_1 = (\texttt{fun}\ x \rightarrow c) \\ b \rightarrow b_1 \rightarrow \ldots \rightarrow b_p \rightarrow v_2 \\ c[x \leftarrow v_2] \rightarrow c_1 \rightarrow \ldots \rightarrow c_q \rightarrow v \end{array}$$

En appliquant la règle (contexte) à chaque étape des deux premières séquences (avec les contextes $[\,]\ b$ et $v_1\ [\,]$), nous obtenons

$$\begin{array}{l} a\ b \rightarrow a_1\ b \rightarrow \ldots \rightarrow a_n\ b \rightarrow v_1\ b \\ v_1\ b \rightarrow v_1\ b_1 \rightarrow \ldots \rightarrow v_1\ b_p \rightarrow v_1\ v_2 \end{array}$$

En enchaînant toutes ces réductions et en glissant au milieu une étape β_{fun}, il vient

$$a\ b \rightarrow \ldots \rightarrow v_1\ b \rightarrow \ldots \rightarrow v_1\ v_2 = (\texttt{fun}\ x \rightarrow c)\ v_2 \rightarrow c[x \leftarrow v_2] \rightarrow \ldots \rightarrow v$$

D'où le résultat attendu $a\ b \overset{*}{\rightarrow} v$. Le raisonnement est le même pour les autres règles d'évaluation (5, 6, 7, 8, 9).

Montrons maintenant l'implication inverse $(a \overset{*}{\rightarrow} v) \Rightarrow (a \overset{v}{\rightarrow} v)$. Il suffit de montrer les deux lemmes suivants:

1. Pour toute valeur v, $v \overset{v}{\rightarrow} v$.

2. Si $a \rightarrow a'$ et $a' \overset{v}{\rightarrow} v$, alors $a \overset{v}{\rightarrow} v$.

En effet, si l'on a $a \rightarrow a_1 \rightarrow \ldots a_n \rightarrow v$, il s'ensuit que $v \overset{v}{\rightarrow} v$ par (1), puis $a_n \overset{v}{\rightarrow} v$ par (2), puis $a_{n-1} \overset{v}{\rightarrow} v$ par (2), et ainsi de suite jusqu'à $a \overset{v}{\rightarrow} v$ 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 $a \overset{\varepsilon}{\rightarrow} a'$, en examinant les axiomes de réduction. Prenons comme exemple β_{fun}: on a $a = (\texttt{fun}\ x \rightarrow a_1)\ v_2$ et $a' = a_1[x \leftarrow v_2]$. On peut construire la dérivation suivante:

$$\frac{(\texttt{fun}\ x \rightarrow a_1) \overset{v}{\rightarrow} (\texttt{fun}\ x \rightarrow a_1) \qquad v_2 \overset{v}{\rightarrow} v_2 \qquad a_1[x \leftarrow v_2] \overset{v}{\rightarrow} v}{(\texttt{fun}\ x \rightarrow a_1)\ v_2 \overset{v}{\rightarrow} v}$$

(La prémisse de gauche est l'axiome 3; celle du milieu vient du lemme (2); celle de droite de l'hypothèse $a_1[x \leftarrow v_2] \overset{v}{\rightarrow} v$.) On a donc bien montré $a \overset{v}{\rightarrow} v$. 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 $a \overset{\varepsilon}{\rightarrow} a'$ et $\Gamma(a') \overset{v}{\rightarrow} v$, alors $\Gamma(a) \overset{v}{\rightarrow} v$. Cela se fait par récurrence structurelle sur Γ. Le cas de base $\Gamma = [\,]$ est le résultat précédent sur les réductions de tête. Faisons le cas $\Gamma = \Delta\ a_2$ par exemple. Supposons donc $\Gamma(a') \overset{v}{\rightarrow} v$. Comme $\Gamma(a') = \Delta(a')\ a_2$, nous avons donc une dérivation de la forme

$$\frac{\Delta(a') \overset{v}{\rightarrow} \texttt{fun}\ x \rightarrow a_3 \qquad a_2 \overset{v}{\rightarrow} v_2 \qquad a_3\{x \leftarrow v_2\} \overset{v}{\rightarrow} v}{\Delta(a')\ a_2 \overset{v}{\rightarrow} v}$$

Par hypothèse de récurrence, on a $\Delta(a) \xrightarrow{v} \texttt{fun}\ x \to a_3$. On peut donc construire la dérivation

$$\frac{\Delta(a) \xrightarrow{v} \texttt{fun}\ x \to a_3 \qquad a_2 \xrightarrow{v} v_2 \qquad a_3\{x \leftarrow v_2\} \xrightarrow{v} v}{\Delta(a)\ a_2 \xrightarrow{v} v}$$

qui conclut $\Gamma(a) \xrightarrow{v} v$ comme attendu. La preuve pour les autres types 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 $E \vdash a : \tau$ (à gauche), on construit une dérivation de typage de $E \vdash a' : \tau$ (à droite):

$$\frac{E \vdash + : \texttt{int} \times \texttt{int} \to \texttt{int} \qquad \dfrac{E \vdash n_1 : \texttt{int} \qquad E \vdash n_2 : \texttt{int}}{E \vdash (n_1, n_2) : \texttt{int}}}{E \vdash +(n_1, n_2) : \texttt{int}} \qquad E \vdash n : \texttt{int} \qquad \text{si } n = n_1 + n_2$$

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

$$\frac{E \vdash \texttt{snd} : \tau_1 \times \tau_2 \to \tau_2 \qquad \dfrac{E \vdash v_1 : \tau_1 \qquad E \vdash v_2 : \tau_2}{E \vdash (v_1, v_2) : \tau_1 \times \tau_2}}{E \vdash \texttt{snd}(v_1, v_2) : \tau_2} \qquad E \vdash v_2 : \tau_2$$

$$\frac{E \vdash \texttt{fix} : (\tau \to \tau) \to \tau \qquad \dfrac{E + \{x : \tau\} \vdash a : \tau}{E \vdash \texttt{fun}\ x \to a : \tau \to \tau}}{E \vdash \texttt{fix}(\texttt{fun}\ x \to a) : \tau} \qquad E \vdash a[x \leftarrow \texttt{fix}(\texttt{fun}\ x \to a)] : \tau$$

(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 $\texttt{int} \times \texttt{int}$. Par le lemme 2.5, v est forcément une paire d'entiers. De même, si $\texttt{fst}\ v$ ou $\texttt{snd}\ v$ est bien typée, v est forcément une paire de valeurs. Enfin pour $\texttt{fix}\ v$, nous savons que v est une valeur ayant un type fonctionnel $\tau \to \tau$. Par le lemme 2.5, v est ou bien une fonction $\texttt{fun}\ x \to a$, soit un opérateur op. Mais le cas $v = op$ est impossible, car aucun des opérateurs qui nous intéressent ont un type de la forme $\tau \to \tau$. Donc v est une fonction et la règle δ_{fix} s'applique.

Appendix C

Corrigés des exercices du chapitre 4

Exercice 4.1 La projection $\texttt{proj}_{i,n}$ se définit comme:

$$\begin{array}{rcl} \texttt{proj}_{n,n} & = & \texttt{fun}\ x \to \texttt{snd}^{n-1}(x) \\ \texttt{proj}_{i,n} & = & \texttt{fun}\ x \to \texttt{fst}(\texttt{snd}^{i-1}(x)) \quad \text{si } i < n \end{array}$$

La traduction T de ML_t dans ML_p:

$$\begin{array}{rcl} T(x) & = & x \\ T(c) & = & c \\ T(\texttt{proj}_{n,n}) & = & \texttt{fun}\ x \to \texttt{snd}^{n-1}(x) \\ T(\texttt{proj}_{i,n}) & = & \texttt{fun}\ x \to \texttt{fst}(\texttt{snd}^{i-1}(x)) \quad \text{si } i < n \\ T(op) & = & op \quad \text{pour les autres opérateurs } op \\ T((v_1, \dots, v_n)) & = & (T(v_1), (T(v_2), \dots (T(v_{n-1}), T(v_n)))) \\ T(\texttt{fun}\ x \to a) & = & \texttt{fun}\ x \to T(a) \\ T(a_1(a_2)) & = & T(a_1)(T(a_2)) \\ T(\texttt{let}\ x = a_1\ \texttt{in}\ a_2) & = & \texttt{let}\ x = T(a_1)\ \texttt{in}\ T(a_2) \end{array}$$

La commutation entre T et la réduction se montre d'abord pour chaque règle de réduction en tête. La plus intéressante est bien sûr la δ-règle pour les tuples: $\texttt{proj}_{i,n}(v_1, \dots, v_n) \xrightarrow{\varepsilon} v_i$. Si $i = n$, on a:

$$\begin{array}{l} T(\texttt{proj}_{n,n}(v_1, \dots, v_n)) \\ \quad = \quad (\texttt{fun}\ x \to \texttt{snd}^{n-1}(x))(T(v_1), (T(v_2), \dots (T(v_{n-1}), T(v_n)))) \\ \quad \to \quad \texttt{snd}^{n-1}(T(v_1), (T(v_2), \dots (T(v_{n-1}), T(v_n)))) \text{ par } \beta_{fun} \\ \quad \to \quad \texttt{snd}^{n-2}(T(v_2), \dots (T(v_{n-1}), T(v_n))) \text{ par } \delta_{snd} \\ \quad \xrightarrow{*} \quad T(v_n) \text{ par } \delta_{snd} \end{array}$$

De même, si $i < n$, on a:

$$\begin{array}{l} T(\texttt{proj}_{n,n}(v_1, \dots, v_n)) \\ \quad = \quad (\texttt{fun}\ x \to \texttt{fst}(\texttt{snd}^{i-1}(x)))(T(v_1), (T(v_2), \dots (T(v_{n-1}), T(v_n)))) \end{array}$$

$$
\begin{array}{ll}
\to & \mathtt{fst}(\mathtt{snd}^{i-1}(T(v_1),(T(v_2),\ldots(T(v_{n-1}),T(v_n))))) \text{ par } \beta_{fun}\\
\to & \mathtt{fst}(\mathtt{snd}^{i-2}(T(v_2),\ldots(T(v_{n-1}),T(v_n)))) \text{ par } \delta_{snd}\\
\overset{*}{\to} & \mathtt{fst}(T(v_i),\ldots) \text{ par } \delta_{snd}\\
\to & T(v_i) \text{ par } \delta_{fst}
\end{array}
$$

Le résultat est immédiat pour β_{fun} et β_{let} car T commute avec la substitution: $T(a[x \leftarrow b]) = T(a)[x \leftarrow T(b)]$. Enfin, pour la règle (contexte), on utilise la commutation de T avec l'application de contexte: $T(\Gamma(a)) = T(\Gamma)(T(a))$. D'où le résultat: si $a \to a'$ dans ML_t, alors $T(a) \overset{*}{\to} T(a')$ dans ML_p.

Réciproquement, si $T(a) \to a''$, a'' n'est pas nécessairement la traduction d'un terme a' tel que $a \to a'$. Exemple:

$$(\mathtt{fun}\ x \to \mathtt{snd}(\mathtt{snd}(x)))(1,(2,3)) \to \mathtt{snd}(\mathtt{snd}(1,(2,3))) \to \mathtt{snd}(2,3) \to 3$$

Le terme de gauche est la traduction de $\mathtt{proj}_{3,3}(1,2,3)$, mais les deux étapes suivantes de la réduction ne sont pas des traductions de termes de ML_t. Si l'on ignore les projections non appliquées et que l'on prend une traduction plus directe des applications de projections:

$$
\begin{array}{rcll}
T(\mathtt{proj}_{n,n}(a)) & = & \mathtt{snd}^{n-1}(a) & \\
T(\mathtt{proj}_{i,n}(a)) & = & \mathtt{fst}(\mathtt{snd}^{i-1}(a)) & \text{si } i < n
\end{array}
$$

l'exemple devient plus intéressant:

$$
\begin{array}{ccccc}
\mathtt{proj}_{3,3}(1,2,3) & & \mathtt{proj}_{2,2}(2,3) & & 3\\
\downarrow T & & \downarrow T & & \downarrow T\\
\mathtt{snd}(\mathtt{snd}(1,(2,3))) & \to & \mathtt{snd}(2,3) & \to & 3
\end{array}
$$

Les étapes intermédiaires de la réduction dans ML_p sont maintenant des traductions de termes de ML_t, mais la réduction intermédiaire

$$\mathtt{proj}_{3,3}(1,2,3) \to \mathtt{proj}_{2,2}(2,3)$$

n'est pas une réduction valide de ML_t.

Exercice 4.2 Il suffit de définir `type bool = false of unit | true of unit`. L'opérateur de filtrage $F_{\mathtt{bool}}$ permet de définir la conditionnelle comme suit:

$$\mathtt{if}\ a\ \mathtt{then}\ b\ \mathtt{else}\ c = F_{\mathtt{bool}}(a,(\mathtt{fun}\ x \to b),(\mathtt{fun}\ x \to c))$$

où x est une variable non libre dans b et c.

Exercice 4.3 Supposons qu'on laisse passer la déclaration suivante:

```
type bug = B of 'a
```

Si l'on donne à `B` le type $\forall \alpha.\alpha \to \mathtt{bug}$, alors le code suivant est bien typé:

```
match B(true) with B(x) -> x+1
```

et pourtant il finit par évaluer $\mathtt{true} + 1$.

Exercice 4.4 Voici quelques valeurs du type `t`:

```
C(fun x → x)
C(fun y → C(fun x → x))
C(fun y → C(fun x → y))
```

Les fonctions `enrouler` et `dérouler`:

```
let enrouler = fun f → C f
let dérouler = fun t → match t with C f → f
```

Le codage du λ-calcul pur:

$$\begin{array}{rcl} [\![x]\!] & = & x \\ [\![\lambda x.M]\!] & = & \texttt{enrouler}(\texttt{fun}\ x \rightarrow [\![M]\!]) \\ [\![M\ N]\!] & = & \texttt{dérouler}[\![M]\!]\ [\![N]\!] \end{array}$$

En corollaire, on voit qu'il existe des expressions de type `t` (la traduction de $(\lambda f.f\ f)(\lambda f.f\ f)$ p.ex.) dont l'évaluation ne termine pas, bien qu'elles soient construites sans un seul `let rec`. Un exemple qui ne passe même pas par le codage du λ-calcul pur est:

```
let delta t = dérouler t t in delta(enrouler delta)
```

Exercice 4.5
1) On définit la substitution $F(p, v)$ comme suit:

$$\begin{array}{rcl} F(\_, v) & = & id \\ F(x, v) & = & [x \leftarrow v] \\ F(c, c) & = & id \\ F(C(p), C(v)) & = & F(p, v) \\ F((p_1, \ldots, p_n), (v_1, \ldots, v_n)) & = & F(p_1, v_1) + \cdots + F(p_n, v_n) \end{array}$$

(Pour le dernier cas, on suppose qu'une même variable x n'apparait jamais deux fois dans un motif p.) Si aucun des cas ci-dessus ne s'applique, $F(p, v)$ est indéfini. La réduction du `match` est alors:

$$\begin{array}{rcl} (\texttt{match}\ v\ \texttt{with}\ p \rightarrow a_2 \mid \_ \rightarrow a_3) & \stackrel{\varepsilon}{\rightarrow} & \sigma(a_2) \quad \text{si } \sigma = F(p, v) \text{ est défini} \\ (\texttt{match}\ v\ \texttt{with}\ p \rightarrow a_2 \mid \_ \rightarrow a_3) & \stackrel{\varepsilon}{\rightarrow} & a_3 \quad \text{si } F(p, v) \text{ est indéfini} \end{array}$$

2) On définit l'énoncé de typage auxiliaire $\vdash p : \tau, E$ ("le motif p filtre des valeurs de type τ, et ce faisant il lie les variables $x \in \mathrm{Dom}(E)$ à des valeurs de type $E(x)$").

$$\vdash \_ : \tau, \emptyset \qquad\qquad \vdash x : \tau, \{x : \tau\} \qquad\qquad \frac{\tau' \rightarrow \tau \leq TC(C) \qquad \vdash p : \tau', E}{\vdash C(p) : \tau, E}$$

$$\frac{\vdash p_1 : \tau_1, E_1 \qquad \ldots \qquad \vdash p_n : \tau_n, E_n}{\vdash (p_1, \ldots, p_n) : \tau_1 \times \ldots \times \tau_n, E_1 + \ldots + E_n}$$

La règle de typage du `match` est alors:

$$\frac{E \vdash a_1 : \tau' \qquad \vdash p : \tau', E' \qquad E + E' \vdash a_2 : \tau \qquad E \vdash a_3 : \tau}{E \vdash (\texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid \_ \to a_3) : \tau}$$

3) On traduit $(\texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid \_ \to a_3)$ en $\texttt{let}\ x = a_1\ \texttt{in}\ C(x, p, a_2, a_3)$, où le schéma de compilation C est défini comme suit:

$$\begin{array}{rcl}
C(a, _, b, c) & = & b \\
C(a, x, b, c) & = & \texttt{let}\ x = a\ \texttt{in}\ b \\
C(a, C(p), b, c) & = & F_t(a, \underbrace{f_n, f_n, \ldots, f_n}_{i-1 \text{ fois}}, f_o, f_n, \ldots, f_n) \\
& & \text{si } C \text{ est le } i^{\text{e}} \text{ constructeur du type } t \\
& & \text{et } f_n = \texttt{fun}\ x \to c\ \ x \text{ non libre dans } c \\
& & \text{et } f_o = \texttt{fun}\ x \to C(x, p, b, c)\ \ x \text{ non libre dans } b \text{ et } c \\
C(a, (p_1, \ldots, p_n), b, c) & = & C(\texttt{proj}_{1,n}(a), p_1, C(\texttt{proj}_{2,n}(a), p_2, \ldots, C(\texttt{proj}_{n,n}(a), p_n, b, c)))
\end{array}$$

Pour plus d'explications, on se reportera aux chapitres 4 et 5 du livre de S.L.Peyton-Jones, *The implementation of functional programming languages*, Prentice-Hall, 1987.

Appendix D

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)
```

on a bien que 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)`.

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 $\forall\alpha,\beta.\ ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)) \rightarrow (\alpha \rightarrow \beta)$.

L'opérateur `fix` sur des types non-fonctionnels (p.ex. $(\texttt{int} \rightarrow \texttt{int}) \rightarrow \texttt{int}$) n'est pas définissable avec des références.

Exercice 5.3

$$c/s \overset{v}{\rightarrow} c/s \qquad op/s \overset{v}{\rightarrow} op/s \qquad (\texttt{fun}\ x \rightarrow a)/s \overset{v}{\rightarrow} (\texttt{fun}\ x \rightarrow a)/s$$

$$\frac{a_1/s \overset{v}{\rightarrow} (\texttt{fun}\ x \rightarrow a)/s_1 \qquad a_2/s_1 \overset{v}{\rightarrow} v_2/s_2 \qquad a[x \leftarrow v_2]/s_2 \overset{v}{\rightarrow} v/s_3}{a_1\ a_2/s \overset{v}{\rightarrow} v/s_3}$$

$$\frac{a_1/s \stackrel{v}{\to} v_1/s_1 \qquad a_2/s_1 \stackrel{v}{\to} v_2/s_2}{(a_1, a_2)/s \stackrel{v}{\to} (v_1, v_2)/s_2} \qquad \frac{a_1/s \stackrel{v}{\to} v_1/s_1 \qquad a_2[x \leftarrow v_1]/s_1 \stackrel{v}{\to} v/s_2}{(\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2)/s \stackrel{v}{\to} v/s_2}$$

$$\frac{a/s \stackrel{v}{\to} v/s_1 \qquad \ell \notin \mathrm{Dom}(s_1)}{(\mathtt{ref}(a))/s \stackrel{v}{\to} \ell/s_1 + [\ell \leftarrow v]} \qquad \frac{a/s \stackrel{v}{\to} \ell/s_1 \qquad \ell \in \mathrm{Dom}(s_1)}{!a/s \stackrel{v}{\to} s_1(\ell)/s_1}$$

$$\frac{a_1/s \stackrel{v}{\to} \ell/s_1 \qquad \ell \in \mathrm{Dom}(s_1) \qquad a_2/s_1 \stackrel{v}{\to} v_2/s_2}{s, (a_1 := a_2) \stackrel{v}{\to} (\,)s_2 + [\ell \leftarrow v_2]}$$

Exercice 5.4 L'eta-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 eta-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'eta-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 eta-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 montre d'abord la conservation du typage par réductions. Pour $(\texttt{try}\ v\ \texttt{with}\ x \to a) \xrightarrow{\varepsilon} v$, il est clair que si $E \vdash (\texttt{try}\ v\ \texttt{with}\ x \to a) : \tau$, alors $E \vdash v : \tau$ également. Pour $(\texttt{try}\ \Delta(\texttt{raise}(v))\ \texttt{with}\ x \to a) \xrightarrow{\varepsilon} a[x \leftarrow v]$, une dérivation de typage du membre gauche est nécessairement de la forme

$$\dfrac{\dfrac{\dfrac{E \vdash \texttt{raise} : \texttt{exn} \to \tau' \qquad E \vdash v : \texttt{exn}}{E \vdash \texttt{raise}(v) : \tau'} \\ \vdots}{E \vdash \Delta(\texttt{raise}(v)) : \tau} \qquad E + \{x : \texttt{exn}\} \vdash a : \tau}{E \vdash (\texttt{try}\ \Delta(\texttt{raise}(v))\ \texttt{with}\ x \to a) : \tau}$$

En appliquant le lemme de substitution (proposition 2.2) à $E \vdash v : \texttt{exn}$ et $E + \{x : \texttt{exn}\} \vdash a : \tau$, il vient $E \vdash a[x \leftarrow v] : \tau$ comme désiré.

Il faut ensuite montrer une propriété de progression similaire à la proposition 2.6. 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 `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 2.6, 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 `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
```

Appendix E

Corrigés des exercices du chapitre 6

Exercice 6.1

```
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 6.2

Hypothèse (H1) pour $\texttt{proj}_e$: supposons $E \vdash \texttt{proj}_e(\{e_1 = v_1;\ \ldots;\ e_n = v_n\}) : \tau$ et $e = e_i$. Une dérivation de ce typage est forcément de la forme:

$$\dfrac{E \vdash \texttt{proj}_e : \{e_i : \texttt{Pre}\ \tau;\ \ldots\} \to \tau \qquad \dfrac{E \vdash v_1 : \tau_1 \qquad \ldots \qquad E \vdash e_n : \tau_n}{E \vdash \{e_1 = v_1;\ \ldots;\ e_n = v_n\} : \{e_1 : \texttt{Pre}\ \tau_1;\ \ldots;\ e_n : \texttt{Pre}\ \tau_n;\ \emptyset\}}}{E \vdash \texttt{proj}_e(\{e_1 = v_1;\ \ldots;\ e_n = v_n\}) : \tau}$$

D'où $\tau = \tau_i$, et le résultat attendu puisque nous avons une sous-dérivation de $E \vdash v_i : \tau_i$ et que le résultat de la delta-réduction est v_i.

Hypothèse (H1) pour $\mathtt{exten}_e$: supposons $E \vdash \mathtt{exten}_e(\{e_1 = v_1; \ldots; e_n = v_n\}, v) : \tau$. Si e n'est pas l'une des e_i, la dérivation de ce typage est de la forme:

$$\dfrac{E \vdash \mathtt{exten}_e : \tau' \times \tau_v \to \tau \qquad \dfrac{\dfrac{E \vdash v_1 : \tau_1 \quad \ldots \quad E \vdash e_n : \tau_n}{E \vdash \{e_1 = v_1;\ \ldots;\ e_n = v_n\} : \tau'} \qquad E \vdash v : \tau_v}{E \vdash (\{e_1 = v_1;\ \ldots;\ e_n = v_n\}, v) : \tau' \times \tau_v}}{E \vdash \mathtt{exten}_e(\{e_1 = v_1;\ \ldots;\ e_n = v_n\}, v) : \tau}$$

avec $\tau' = \{e_1 : \mathtt{Pre}\ \tau_1;\ \ldots;\ e_n : \mathtt{Pre}\ \tau_n;\ \emptyset\}$ et $\tau = \{e : \mathtt{Pre}\ \tau_v;\ e_1 : \mathtt{Pre}\ \tau_1;\ \ldots;\ e_n : \mathtt{Pre}\ \tau_n;\ \emptyset\}$. D'autre part, le résultat de la delta-réduction est $\{e = v;\ e_1 = v_1;\ \ldots;\ e_n = v_n\}$. Comme e et les e_i sont deux à deux différentes, la règle (record) appliquée aux hypothèses $E \vdash v : \tau_v$ et $E \vdash v_i : \tau_i$ montre que le résultat de la delta-réduction a le type τ, comme désiré.

Si $e = e_j$, on a une dérivation de la même forme mais avec $\tau = \{e_1 : \mathtt{Pre}\ \tau_1;\ \ldots;\ e_j : \mathtt{Pre}\ \tau_v; \ldots; e_n : \mathtt{Pre}\ \tau_n; \emptyset\}$. Le résultat de la delta-réduction est $\{e_1 = v_1; \ldots; e_j = v; \ldots; e_n = v_n\}$. On conclut que ce résultat est de type τ à partir des hypothèses $E \vdash v : \tau_v$ et $E \vdash v_i : \tau_i$, $i \not= j$.

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	$\mathtt{fun}\ x \to a$	(v_1, v_2)	$\{e_i = v_i\}$
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 $\{\ldots e_i = v_i \ldots\}$ est nécessairement de la forme $\{\ldots e_i : \mathtt{Pre}\ \tau_i \ldots; \emptyset\}$ avec $\emptyset \vdash v_i : \tau_i$. Donc, toutes les étiquettes marquées présentes dans ce type enregistrement sont bien présentes dans la valeur.

Hypothèse (H2) pour $\mathtt{proj}_e$: supposons $\emptyset \vdash \mathtt{proj}_e(v) : \tau$. Vu le schéma de type de $\mathtt{proj}_e$, nous avons $\emptyset \vdash v : \{e : \mathtt{Pre}\ \tau_1;\ \tau_2\}$. Donc, v est un enregistrement et contient un champ étiqueté e. Par conséquent, l'application de $\mathtt{proj}_e$ se réduit par la delta-règle pour les projections.

Hypothèse (H2) pour $\mathtt{exten}_e$: si $\emptyset \vdash \mathtt{exten}_e(v) : \tau$, v a nécessairement un type enregistrement. C'est donc un enregistrement, et l'application $\mathtt{exten}_e(v)$ se réduit par la delta-règle pour $\mathtt{exten}_e$.

Exercice 6.3
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 $K' = K + \{\alpha \leftarrow \kappa\}$.

- Si $\tau = \alpha$ et $\alpha \in Dom(K)$:
  si $\kappa = K(\alpha)$, prendre $K' = K$, sinon échec.

- Si $\tau = T$:
  si $\kappa = \mathtt{TYPE}$, prendre $K' = K$, sinon échec.

- Si $\tau = \tau_1 \rightarrow \tau_2$ ou $\tau = \tau_1 \times \tau_2$:
  si $\kappa = \mathtt{TYPE}$, prendre $K' = S(\tau_2, \mathtt{TYPE}, S(\tau_1, \mathtt{TYPE}, K))$, sinon échec.

- Si $\tau = \{\tau'\}$:
  si $\kappa = \mathtt{TYPE}$, prendre $K' = S(\tau', \mathtt{R}(\emptyset), \tau')$, sinon échec.

- Si $\tau = \emptyset$:
  si $\kappa = \mathtt{R}(E)$ pour un certain E, prendre $K' = K$, sinon échec.

- Si $\tau = (e : \tau_1;\ \tau_2)$:
  Si $\kappa = \mathtt{TYPE}$ ou $\kappa = \mathtt{PRE}$, échec.
  Si $\kappa = \mathtt{R}(E)$ et $e \in E$, échec.
  Si $\kappa = \mathtt{R}(E)$ et $e \notin E$, prendre $K' = S(\tau_2, \mathtt{R}(E \cup \{e\}), S(\tau_1, \mathtt{TYPE}, K))$.

- Si $\tau = \mathtt{Abs}$:
  si $\kappa = \mathtt{PRE}$, prendre $K' = K$, sinon échec.

- Si $\tau = \mathtt{Pre}\ \tau'$:
  si $\kappa = \mathtt{PRE}$, prendre $K' = S(\tau', \mathtt{TYPE}, K)$, sinon échec.

Pour typer une contrainte $(a : \tau)$ ou une déclaration de constructeur C `of` τ, on appelle $S(\tau, \mathtt{TYPE}, K)$ et l'algorithme S vérifier et propage la contrainte $\tau :: \mathtt{TYPE}$ vers les variables de types de τ, déterminant au passage leurs sortes.

Exercice 6.4
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 $\{e : \mathtt{Pre\ int};\ f : \mathtt{Pre\ bool};\ \emptyset\}$ signifie "il y a un champ e de type `int` *et* un champ f de type `bool` *et* rien d'autre". Le type somme $[C : \mathtt{Pre\ int};\ D : \mathtt{Pre\ bool};\ \emptyset]$ 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 $\emptyset$, cela signifie "...ou d'autres constructeurs" au lieu de "...ou rien d'autre". Avec ces intuitions, on obtient les types suivants pour les opérateurs:

$$\begin{array}{rcl} C & : & \forall \alpha, \beta.\ \alpha \rightarrow [C : \mathtt{Pre}\ \alpha;\ \beta] \\ P_C & : & \forall \alpha.\ [C : \mathtt{Pre}\ \alpha;\ \emptyset] \rightarrow \alpha \\ F_C & : & \forall \alpha, \beta, \gamma.\ [C : \mathtt{Pre}\ \alpha;\ \beta] \times (\alpha \rightarrow \gamma) \times ([\beta] \rightarrow \gamma) \rightarrow \gamma \end{array}$$

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 $[C : \mathtt{Pre\ int};\ \tau]$ pour toute rangée τ, et `if` *cond* `then` $C(1)$ `else` $D(\mathtt{true})$ a le type $[C : \mathtt{Pre\ int};\ D : \mathtt{Pre\ bool};\ \tau]$, 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:

$$\begin{array}{rcl} \texttt{fun}\ x \to F_C(x, \texttt{fun}\ y \to y, \texttt{fun}\ z \to 0) & : & [C : \texttt{Pre int};\ \alpha] \to \texttt{int} \\ \texttt{fun}\ x \to F_C(x, \texttt{fun}\ y \to y, P_D) & : & [C : \texttt{Pre int};\ D : \texttt{Pre int};\ \emptyset] \to \texttt{int} \end{array}$$

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.

Appendix F

Corrigés des exercices du chapitre 7

Exercice 7.1 La réflexivité $\tau <: \tau$ est immédiate par récurrence structurelle sur τ. Pour la transitivité ($\tau_1 <: \tau_2$ et $\tau_2 <: \tau_3$ implique $\tau_1 <: \tau_3$) on procède par récurrence structurelle sur τ_1, τ_2, τ_3 et par cas sur le constructeur de tête de ces trois types. On est forcément dans l'un des cas suivants:

τ_1	τ_2	τ_3	
T	T	T	On a bien $T <: T$ par axiome
α	α	α	Là aussi, $\alpha <: \alpha$ est un axiome
$\varphi_1 \to \psi_1$	$\varphi_2 \to \psi_2$	$\varphi_3 \to \psi_3$	On a $\varphi_3 <: \varphi_2$ et $\varphi_2 <: \varphi_1$ et $\psi_1 <: \psi_2$ et $\psi_2 <: \psi_3$. Par hypothèse de récurrence, $\varphi_3 <: \varphi_1$ et $\psi_1 <: \psi_3$. D'où $\varphi_1 \to \psi_1 <: \varphi_3 \to \psi_3$ par la règle de sous-typage des types flèche.
$\varphi_1 \times \psi_1$	$\varphi_2 \times \psi_2$	$\varphi_3 \times \psi_3$	Même raisonnement que pour les types flèche.
$\langle\varphi_1\rangle$	$\langle\varphi_2\rangle$	$\langle\varphi_3\rangle$	On a $\varphi_1 <: \varphi_2$ et $\varphi_2 <: \varphi_3$, d'où $\varphi_1 <: \varphi_3$ par hypothèse de récurrence, et $\langle\varphi_1\rangle <: \langle\varphi_3\rangle$ par la règle de sous-typage des types objets.
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset <: \emptyset$ par axiome.
τ_1	$\emptyset$	$\emptyset$	$\tau_1 <: \emptyset$ par axiome.
τ_1	τ_2	$\emptyset$	$\tau_1 <: \emptyset$ par axiome.
$m : \varphi_1;\ \psi_1$	$m : \varphi_2;\ \psi_2$	$m : \varphi_3;\ \psi_3$	Même raisonnement que pour les types produit.

Enfin, pour l'antisymétrie ($\tau <: \tau'$ et $\tau' <: \tau$ impliquent $\tau = \tau'$), on raisonne par récurrence structurelle sur τ et τ'. Les cas où τ et τ' ont le même constructeur de tête sont immédiats par application de l'hypothèse de récurrence. Reste le cas $\tau' = \emptyset$ où $\tau <: \tau'$ est vrai pour tout τ. Cependant, on a aussi $\emptyset <: \tau$ et cela n'est possible que si $\tau = \emptyset$, d'où $\tau = \tau'$ comme désiré.

Exercice 7.2 Supposons $E \vdash (\texttt{fun}\ x \rightarrow a)\ v : \tau$. Une dérivation de ce typage est nécessairement de la forme suivante:

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{E + \{x : \varphi_1\} \vdash a : \tau_1}{E \vdash \texttt{fun}\ x \rightarrow a : \varphi_1 \rightarrow \tau_1} \quad \varphi_1 \rightarrow \tau_1 <: \varphi_2 \rightarrow \tau_2
}{\vdots}
}{E \vdash \texttt{fun}\ x \rightarrow a : \varphi_n \rightarrow \tau_n} \quad \varphi_n \rightarrow \tau_n <: \varphi \rightarrow \tau
}{E \vdash \texttt{fun}\ x \rightarrow a : \varphi \rightarrow \tau} \quad E \vdash v : \varphi
}{E \vdash (\texttt{fun}\ x \rightarrow a)\ v : \tau}
$$

En effet, les règles de typage ne sont plus dirigées par la syntaxe: pour tout terme a, il y a deux règles qui peuvent dériver $E \vdash a : \tau$, la règle (sub) et la règle propre à la forme de a (p.ex. (fun) si a est une fonction). Donc, la forme générale d'une dérivation de $E \vdash \texttt{fun}\ x \rightarrow a : \varphi \rightarrow \tau$ est une étape de règle (fun) suivie par zéro, une ou plusieurs étapes de (sub).

Par transitivité du sous-typage (exercice 7.1), nous avons $\varphi_1 \rightarrow \tau_1 <: \varphi \rightarrow \tau$, ce qui implique $\varphi <: \varphi_1$ et $\tau_1 <: \tau$. Appliquant la règle (sub), on a donc $E \vdash v : \varphi_1$. Comme de plus $E + \{x : \varphi_1\} \vdash a : \tau_1$, un lemme de substitution analogue au lemme 2.2 montre que $E \vdash a[x \leftarrow v] : \tau_1$. Appliquant une dernière fois la règle (sub), il vient $E \vdash a[x \leftarrow v] : \tau$; c'est le résultat attendu.

Exercice 7.3 Tout terme du lambda-calcul pur peut être vu comme une expression de mini-ML ayant le type $\tau = \mu\alpha.\ \alpha \rightarrow \alpha$. Considérons par exemple $\texttt{fun}\ x \rightarrow a$. Si sous l'hypothèse $x : \tau$ le type de a est τ, alors $\texttt{fun}\ x \rightarrow a$ a le type $\tau \rightarrow \tau$ qui par enroulage est égal à τ. De même, si a_1 et a_2 ont le type τ, alors par déroulage a_1 a aussi le type $\tau \rightarrow \tau$, donc l'application $a_1\ a_2$ est bien typée et a le type τ.

Exercice 7.4
1)

```
                                        ------          ---------
                                       |      |        |         |
   ->          *        *        ->           |        ->        |
  /  \        / \      / \      /  \          |       /  \       |
int   bool   V   V     \ /    int   \____|    V      ->      |
                        V                                 /  \     |
                                                         V    \____|
```

2) La substitution renvoyée a deux classes d'équivalence: l'une contient les trois noeuds $\rightarrow$ du graphe de départ, et l'autre les trois noeuds représentant `int`, α et β dans le graphe de départ.

3) Tout d'abord, il est clair que si R est une relation d'équivalence et $R' = \texttt{mgu}(E, R)$ est définie, alors R' est une relation d'équivalence, et de plus les classes d'équivalence de R sont incluses dans celles de R'.

Ensuite, considérons un appel $\texttt{mgu}(E, R)$ qui, récursivement, appelle $\texttt{mgu}(E', R')$. Dans tous les cas de l'algorithme, pour tout $(n_1 \stackrel{?}{=} n_2) \in E$, on a ou bien $(n_1 \stackrel{?}{=} n_2) \in E'$, ou bien $n_1\ R\ n_2$.

Donc, par récurrence sur le déroulement de l'algorithme, il vient que $R = \mathtt{mgu}(E, id)$ satisfait les équations de E: pour tout $(n_1 \stackrel{?}{=} n_2) \in E$, on a $n_1 \; R \; n_2$.

Il reste à vérifier que $R = \mathtt{mgu}(E, id)$ vérifie les conditions de compatibilité et de fermeture. Si $n \; R \; n'$ et $C(n) \neq V$ et $C(n') \neq V$, alors forcément une des étapes de l'algorithme a été

$$\begin{aligned} \mathtt{mgu}(\{n \stackrel{?}{=} n'\} \cup E, R) &= \mathtt{mgu}(E \cup \{F_1(n) \stackrel{?}{=} F_1(n'); \ldots; F_k(n) \stackrel{?}{=} F_k(n')\}, R + \{n = n'\}) \\ & \quad \text{si } C(n) \neq V \text{ et } C(n') = C(n) \text{ et } k = A(C(n)) \end{aligned}$$

Donc $C(n) = C(n')$, et de plus la relation R satisfait les équations de $E \cup \{F_1(n) \stackrel{?}{=} F_1(n'); \ldots; F_k(n) \stackrel{?}{=} F_k(n')\}$, d'où $F_i(n) \; R \; F_i(n')$ pour tout i. Par conséquent, R est une substitution, et comme elle satisfait toutes les équations de E, c'est un unificateur de E.

Soit maintenant R' un unificateur de E. On montre par récurrence sur le déroulement de l'algorithme que si R_1 est plus fine que R' et $R_2 = \mathtt{mgu}(E, R_1)$, alors R_2 est plus fine que R'. Faisons par exemple le cas

$$\begin{aligned} \mathtt{mgu}(\{n \stackrel{?}{=} n'\} \cup E, R_1) &= \mathtt{mgu}(E \cup \{F_1(n) \stackrel{?}{=} F_1(n'); \ldots; F_k(n) \stackrel{?}{=} F_k(n')\}, R_1 + \{n = n'\}) \\ & \quad \text{si } C(n) \neq V \text{ et } C(n') = C(n) \text{ et } k = A(C(n)) \end{aligned}$$

Puisque R' est un unificateur de $\{n \stackrel{?}{=} n'\} \cup E$, on a nécessairement $n \; R' \; n'$. Comme R' est une relation d'équivalence et qu'elle est moins fine que R_1, elle est aussi moins fine que $R_1 + \{n = n'\}$. De plus, R' est un unificateur de $E \cup \{F_1(n) \stackrel{?}{=} F_1(n'); \ldots; F_k(n) \stackrel{?}{=} F_k(n')\}$ puisque c'est un unificateur de $E \cup \{n \stackrel{?}{=} n'\}$ et puisque R' satisfait la condition de fermeture. Appliquant l'hypothèse de récurrence, il vient que $R_2 = \mathtt{mgu}(E \cup \{F_1(n) \stackrel{?}{=} F_1(n'); \ldots; F_k(n) \stackrel{?}{=} F_k(n')\}, R_1 + \{n = n'\})$ est plus fine que R'. C'est le résultat attendu, puisque $R_2 = \mathtt{mgu}(\{n \stackrel{?}{=} n'\} \cup E, R_1)$.

4) À chaque appel récursif de $\mathtt{mgu}(E, R)$, ou bien le nombre de classes d'équivalences de R diminue d'un, ou bien R est inchangé mais le nombre d'équations de E diminue d'un. Ceci garantit que `mgu` ne peut pas boucler.

Exercice 7.5

1) La symétrie est évidente par récurrence sur les deux types.

Si $d(\tau_1, \tau_2) = 0$, ou bien τ_1 et τ_2 sont la même variable de type, ou bien τ_1 et τ_2 sont deux types flèche $\varphi_1 \to \psi_1$ et $\varphi_2 \to \psi_2$, et $d(\varphi_1, \varphi_2) = 0$ et $d(\psi_1, \psi_2) = 0$. Par récurrence, il vient $\varphi_1 = \varphi_2$ et $\psi_1 = \psi_2$, d'où $\tau_1 = \tau_2$.

Pour l'inégalité du triangle ultramétrique, on raisonne par récurrence et par cas sur les trois types τ_1, τ_2, τ_3. On dit que deux types sont en désaccord si ce sont deux variables différentes, ou si l'un est un type flèche et l'autre une variable. La distance entre deux types en désaccord est toujours 1.

Si τ_1 et τ_2 sont en désaccord, ou τ_2 et τ_3 en désaccord: alors $\max(d(\tau_1, \tau_2), d(\tau_2, \tau_3)) = 1$ et l'inégalité est vérifiée, car $d(\tau_1, \tau_3) \leq 1$ quels que soient τ_1 et τ_3.

Si $\tau_1 = \tau_2 = \tau_3 = \alpha$, les trois distances sont nulles et l'inégalité est vérifiée.

Enfin, si $\tau_1 = \varphi_1 \to \psi_1$ et $\tau_2 = \varphi_2 \to \psi_2$ et $\tau_3 = \varphi_3 \to \psi_3$: par hypothèse de récurrence, on a $d(\varphi_1, \varphi_3) \leq \max(d(\varphi_1, \varphi_2), d(\varphi_2, \varphi_3))$ et de même $d(\psi_1, \psi_3) \leq \max(d(\psi_1, \psi_2), d(\psi_2, \psi_3))$. D'où:

$$d(\varphi_1 \to \psi_1, \varphi_3 \to \psi_3) \quad = \quad 1/2 \max(d(\varphi_1, \varphi_3), d(\psi_1, \psi_3))$$

$$
\begin{aligned}
&\leq \quad 1/2 \max(d(\varphi_1, \varphi_2), d(\varphi_2, \varphi_3), d(\psi_1, \psi_2), d(\psi_2, \psi_3)) \\
&= \quad \max(1/2 \max(d(\varphi_1, \varphi_2), d(\psi_1, \psi_2)), 1/2 \max(d(\varphi_2, \varphi_3), d(\psi_2, \psi_3))) \\
&= \quad \max(d(\varphi_1 \to \psi_1, \varphi_2 \to \psi_2), d(\varphi_2 \to \psi_2, \varphi_3 \to \psi_3))
\end{aligned}
$$

2a) Soient (τ_n) et (τ'_n) deux suites de Cauchy. On montre que la suite $(d(\tau_n, \tau'_n))$ à valeurs dans $\mathbf{R}$ est de Cauchy. En effet, pour tous p, q, on a:

$$d(\tau_p, \tau'_p) \leq \max(d(\tau_p, \tau_q), d(\tau_q, \tau'_q), d(\tau'_q, \tau'_p))$$

d'où

$$d(\tau_p, \tau'_p) - d(\tau_q, \tau'_q) \leq \max(d(\tau_p, \tau_q), d(\tau'_q, \tau'_p))$$

et, en intervertissant les rôles de p et q,

$$d(\tau_q, \tau'_q) - d(\tau_p, \tau'_p) \leq \max(d(\tau_q, \tau_p), d(\tau'_p, \tau'_q))$$

d'où

$$|d(\tau_p, \tau'_p) - d(\tau_q, \tau'_q)| \leq \max(d(\tau_q, \tau_p), d(\tau'_p, \tau'_q)).$$

Soit $\varepsilon > 0$. Soient N_1 et N_2 tels que $p, q \geq N_1 \Rightarrow d(\tau_p, \tau_q) \leq \varepsilon$ et $p, q \geq N_2 \Rightarrow d(\tau'_p, \tau'_q) \leq \varepsilon$. Pour tous $p, q \geq \max(N_1, N_2)$, on a donc $|d(\tau_p, \tau'_p) - d(\tau_q, \tau'_q)| \leq \max(\varepsilon, \varepsilon) = \varepsilon$. Donc la suite $(d(\tau_n, \tau'_n))$ est de Cauchy dans $\mathbf{R}$. Elle converge donc.

2b) La réflexivité de $\cong$ découle de $d(\tau, \tau) = 0$ pour tout τ. La symétrie de $\cong$ découle de celle de d. Pour la transitivité de $\cong$, supposons $(\tau_n) \cong (\tau'_n)$ et $(\tau'_n) \cong (\tau''_n)$. Pour tout n, on a:

$$d(\tau_n, \tau''_n) \leq \max(d(\tau_n, \tau'_n), d(\tau'_n, \tau''_n))$$

Passant à la limite $n \to \infty$, il vient

$$d(\tau, \tau'') \leq \max(d(\tau, \tau'), d(\tau', \tau'')) = 0$$

d'où $(\tau_n) \cong (\tau''_n)$.

2c) Par passage à la limite sur l'inégalité du triangle ultramétrique, on a $d(s, s'') \leq \max(d(s, s'), d(s', s''))$ pour toutes suites s, s', s''. De même, $d(s, s') = d(s', s)$.

On vérifie maintenant que d passe au quotient par $\cong$. Si $s \cong s'$ et $u \cong u'$, on a

$$d(s', u') \leq \max(d(s', s), d(s, u), d(u, u')) = d(s, u)$$

et de même

$$d(s, u) \leq \max(d(s, s'), d(s', u'), d(u', u)) = d(s', u').$$

Donc $d(s, u) = d(s', u')$, et d passe au quotient par $\cong$.

Par définition de $\cong$, $d(s, u) = 0$ si et seulement si $s \cong u$, c'est-à-dire si et seulement si s et u sont égales dans $\mathcal{S}/\cong$.

2d) À tout type simple τ on associe la suite constante $(\tau, \tau, \ldots)$. La distance entre deux telles suites constantes (τ) et (τ') est bien sûr $d(\tau, \tau')$.

2e) Soit (s_n) une suite de Cauchy à valeurs dans $\mathcal{T}$, c'est-à-dire une suite de Cauchy de suites de Cauchy. On note $s_{n,p}$ le $p^{\text{ième}}$ élément de la suite s_n.

Par définition de $\mathcal{T}$, la suite $(s_{n,p})_{p\in\mathbf{N}}$, vue comme suite à valeurs dans $\mathcal{T}$, converge vers s_n. Donc, pour tout $n > 0$, il existe $N(n)$ tel que

$$p \geq N(n) \Rightarrow d(s_{n,p}, s_n) \leq \frac{1}{n}$$

On définit la suite diagonale u par $u_n = s_{n,N(n)}$. Montrons que u est de Cauchy et que (s_n) converge vers u. Pour tous p, q, on a:

$$d(u_p, u_q) = d(s_{p,N(p)}, s_{q,N(q)}) \leq \max(d(s_{p,N(p)}, s_p),\ d(s_p, s_q),\ d(s_q, s_{q,N(q)})) \leq \max(\frac{1}{p}, d(s_p, s_q), \frac{1}{q})$$

Soit $\varepsilon > 0$. Soit N_0 tel que

$$p, q \geq N_0 \Rightarrow d(s_p, s_q) \leq \varepsilon.$$

Soit N_1 tel que $1/N_1 \leq \varepsilon$. Pour tous $p, q \geq \max(N_0, N_1)$, on a

$$d(u_p, u_q) \leq \max(\frac{1}{p}, d(s_p, s_q), \frac{1}{q}) \leq \max(\frac{1}{N_1}, \varepsilon, \frac{1}{N_1}) \leq \varepsilon$$

Donc la suite u est bien de Cauchy.

Montrons que (s_n) converge vers u. Pour tous p et q,

$$d(u_p, s_q) = d(s_{p,N(p)}, s_q) \leq \max(d(s_{p,N(p)}, s_p),\ d(s_p, s_q)) \leq \max(\frac{1}{p}, d(s_p, s_q))$$

Soit $\varepsilon > 0$. Soit N_0 tel que

$$p, q \geq N_0 \Rightarrow d(s_p, s_q) \leq \varepsilon.$$

Soit N_1 tel que $1/N_1 \leq \varepsilon$. Pour tous $p, q \geq \max(N_0, N_1)$, on a

$$d(u_p, s_q) \leq \max(\frac{1}{p}, d(s_p, s_q)) \leq \max(\frac{1}{N_1}, \varepsilon) \leq \varepsilon$$

Faisant tendre p vers ∞, il vient $d(u, s_q) \leq \varepsilon$. Donc, pour tout $\varepsilon > 0$, il existe $N = \max(N_0, N_1)$ tel que $q \geq N \Rightarrow d(u, s_q) \leq \varepsilon$. Ceci montre que u est la limite de (s_q).

3) Unicité du point fixe: si x et y sont deux points fixes de F, on a $x = F(x)$ et $y = F(y)$, d'où

$$d(x, y) = d(F(x), F(y)) \leq k\, d(x, y)$$

et comme $k < 1$, ceci entraîne $d(x, y) = 0$, d'où $x = y$.

Existence du point fixe: on construit la suite (x_n) par $x_{n+1} = F(x_n)$ et x_0 quelconque. On a

$$d(x_{n+1}, x_n) \leq k\, d(x_n, x_{n-1}) \leq \ldots \leq k^n d(x_1, x_0)$$

Montrons que la suite (x_n) est de Cauchy. On a

$$d(x_{n+p}, x_n) \leq d(x_{n+p}, x_{n+p-1}) + \cdots + d(x_{n+1}, x_n) \leq (k^{n+p} + \cdots + k^n) d(x_1, x_0) \leq \frac{k^n}{1-k} d(x_1, x_0)$$

Soit alors $\varepsilon > 0$. Soit N suffisamment grand pour que $n \geq N$ entraîne $\frac{k^N}{1-k} d(x_1, x_0) \leq \varepsilon/2$. Pour tous $p, q \geq N$, on a

$$d(x_p, x_q) \leq d(x_p, x_N) + d(x_N, x_q) \leq \frac{k^N}{1-k} d(x_1, x_0) + \frac{k^N}{1-k} d(x_1, x_0) \leq \varepsilon$$

Donc la suite (x_n) est de Cauchy. Soit x sa limite. Pour tout n, on a $d(x_{n+1}, F(x)) \leq k\, d(x_n, x)$. Faisant tendre n vers ∞, il vient que $F(x)$ est la limite de la suite (x_n). Par unicité de la limite, il s'ensuit $F(x) = x$, et x est un point fixe de F.

4) Pour tous types finis τ, τ', τ'', on montre par une récurrence facile sur τ que

$$d(\tau[\alpha \leftarrow \tau'], \tau[\alpha \leftarrow \tau'']) \leq d(\tau', \tau'')$$

Si de plus $\tau \neq \alpha$, on a

$$d(\tau[\alpha \leftarrow \tau'], \tau[\alpha \leftarrow \tau'']) \leq \frac{1}{2} d(\tau', \tau'').$$

En effet, ou bien τ est une variable de type différente de α, et alors $\tau[\alpha \leftarrow \tau'] = \tau[\alpha \leftarrow \tau'']$, ou bien τ est un type flèche $\tau_1 \to \tau_2$ et alors

$$d(\tau[\alpha \leftarrow \tau'], \tau[\alpha \leftarrow \tau'']) \leq \frac{1}{2} \max(d(\tau_1[\alpha \leftarrow \tau'], \tau_1[\alpha \leftarrow \tau'']), d(\tau_2[\alpha \leftarrow \tau'], \tau_2[\alpha \leftarrow \tau''])) \leq \frac{1}{2} d(\tau', \tau'')$$

Cette inégalité s'étend ensuite à des types τ, τ', τ'' infinis par passage à la limite.

Par conséquent, l'opérateur $F(\tau') = \tau[\alpha \leftarrow \tau']$ est contractif: $d(F(\tau', \tau'')) \leq \frac{1}{2} d(\tau', \tau'')$. Appliquant le théorème de Banach-Tarski, il vient qu'il existe un et un seul point fixe τ' de F. Donc, $\tau' = \tau[\alpha \leftarrow \tau']$ et τ' est le seul type qui vérifie cette égalité.

Appendix G

Corrigés des exercices du chapitre 8

Exercice 8.1

Proposition G.1 (Réflexivité) *Pour tous environnements E et types de modules M, on a $E \vdash M <: M$.*

Démonstration: par récurrence structurelle sur le type M.

Cas $M = \texttt{sig}\ S_1; \ldots; S_n\ \texttt{end}$. On cherche à obtenir le résultat désiré en appliquant la règle (sub-sig) avec comme injection ρ l'identité. Il suffit donc de montrer

$$E;\ S_1;\ \ldots;\ S_n \vdash S_i <: S_i \tag{1}$$

pour tout $i = 1, \ldots, n$.

Si $S_i = (\texttt{val}\ v : \sigma)$, on a $E;\ S_1;\ \ldots;\ S_n \vdash \sigma \approx \sigma$ (axiome de transitivité sur $\approx$ plus règle (schémas)), d'où (1) par application de la règle (sub-val).

Si $S_i = (\texttt{type}\ t)$, (1) s'ensuit de la règle (sub-abstr-abstr).

Si $S_i = (\texttt{type}\ t = \tau)$, on a $E;\ S_1;\ \ldots;\ S_n \vdash \tau \approx \tau$ par la règle (eq-réflexivité), d'où (1) par la règle (sub-mani-mani).

Si $S_i = (\texttt{module}\ X : M_1)$, on a $E;\ S_1;\ \ldots;\ S_n \vdash M_1 <: M_1$ par application de l'hypothèse de récurrence (M_1 est un sous-terme strict de M). Utilisant la règle (sub-mod), on conclut (1).

D'où le résultat attendu, $E \vdash M <: M$, par application de la règle (8).

Cas $M = \texttt{functor}(X : M_1) \to M_2$. En appliquant l'hypothèse de récurrence à M_1, on obtient

$$E \vdash M_1 <: M_1$$

et en l'appliquant à M_2,

$$E;\ \texttt{module}\ X : M_1 \vdash M_2 <: M_2$$

D'où $E \vdash M <: M$ en appliquant la règle (sub-foncteur). □

Pour prouver la transitivité de $<:$, il faut introduire une relation de sous-typage entre environnements de typage. Un environnement de typage peut être vu comme l'intérieur d'une signature $\texttt{sig} \ldots \texttt{end}$: les deux sont des séquences de spécifications. On dit par conséquent que deux environnements E et A' sont en relation de sous-typage, et on note $\vdash E <: A'$, si

$\emptyset \vdash$ `sig` E `end` $<:$ `sig` A' `end`; c'est-à-dire, notant $E = S_1; \ldots; S_m$ et $A' = S'_1; \ldots; S'_n$, il existe une injection ρ telle que $E \vdash S_{\rho(i)} <: S'_i$ pour tout $i = 1, \ldots, n$.

L'intérêt de cette notion est que, si $\vdash E <: E'$, tout résultat de typage qui peut être dérivé sous les hypothèses E' peut également être dérivé sous les hypothèses E (hypothèses plus fortes). Plus précisément, on a le lemme suivant:

Proposition G.2 (Sous-typage sur l'environnement) *Supposons $\vdash E <: E'$.*

1. *Si $E' \vdash \tau \approx \tau'$, alors $E \vdash \tau \approx \tau'$.*
2. *Si $E' \vdash \sigma \geq \sigma'$, alors $E \vdash \sigma \geq \sigma'$.*
3. *Si $E' \vdash S <: S'$, alors $E \vdash S <: S'$.*
4. *Si $E' \vdash M <: M'$, alors $E \vdash M <: M'$.*

Démonstration: on commence par prouver (1) par récurrence sur la dérivation de $E' \vdash \tau \approx \tau'$. Le seul cas intéressant est le cas de base correspondant à l'application de la règle (1): $E' \vdash t \approx \tau'$ parce que E' contient l'hypothèse `type` $t = \tau'$. Par définition du sous-typage entre environnements, E doit forcément contenir une hypothèse S telle que $E \vdash S <:$ (`type` $t = \tau'$). Il n'y a que deux possibilités: $S =$ (`type` $t = \tau$) et $E \vdash \tau \approx \tau'$, ou bien $S =$ (`type` t) et $E \vdash t \approx \tau'$. Le second cas nous donne directement la preuve de $E \vdash t \approx \tau'$ désirée. Dans le premier cas, on l'obtient par la dérivation suivante:

$$\dfrac{\dfrac{E = E_1;\ \texttt{type}\ t = \tau;\ E_2}{E \vdash t \approx \tau} \qquad E \vdash \tau \approx \tau'}{E \vdash t \approx \tau'}$$

On prouve ensuite (2) comme corollaire immédiat de (1), puis (3) et (4) simultanément par récurrence structurelle sur M et S. □

Proposition G.3 (Transitivité) *Si $E \vdash M <: M'$ et $E \vdash M' <: M''$, alors $E \vdash M <: M''$.*

Démonstration: par récurrence structurelle sur les types M, M', M''. Vu les règles (sub-sig) et (sub-foncteur), ces trois types sont ou bien trois signatures, ou bien trois types de foncteurs.

Cas $M =$ `sig` $S_1; \ldots; S_m$ `end` et $M' =$ `sig` $S'_1; \ldots; S'_n$ `end` et $M'' =$ `sig` $S''_1; \ldots; S''_p$ `end`. La seule règle ayant pu conclure $E \vdash M <: M'$ et $E \vdash M' <: M''$ est la règle (sub-sig). Ses prémisses sont donc nécessairement vraies. Il existe donc deux injections $\varphi : \{1 \ldots n\} \mapsto \{1 \ldots m\}$ et $\psi : \{1 \ldots p\} \mapsto \{1 \ldots n\}$ telles que

$$E;\ S_1;\ \ldots;\ S_m \vdash S_{\varphi(i)} <: S'_i \quad \text{pour } i = 1, \ldots, n \tag{2}$$

$$E;\ S'_1;\ \ldots;\ S'_n \vdash S'_{\psi(i)} <: S''_i \quad \text{pour } i = 1, \ldots, p \tag{3}$$

Notons $B = E;\ S_1;\ \ldots;\ S_m$ et $B' = E;\ S'_1;\ \ldots;\ S'_n$. Il est facile de voir que $\vdash B <: B'$. Par la proposition G.2, on peut donc prouver (3) sous les hypothèses B au lieu de B':

$$E;\ S_1;\ \ldots;\ S_n \vdash S'_{\psi(i)} <: S''_i \quad \text{pour } i = 1, \ldots, p \tag{4}$$

Considérons alors $\rho = \varphi \circ \psi$. C'est une injection de $\{1 \dots p\}$ dans $\{1 \dots m\}$. Fixons i dans $\{1 \dots p\}$. Par (2) et (4), nous avons

$$B \vdash S_{\rho(i)} <: S'_{\psi(i)} \qquad \text{et} \qquad B \vdash S'_{\psi(i)} <: S''_i$$

Nous montrons maintenant que $B \vdash S_{\rho(i)} <: S''_i$ en discutant sur la forme de $S_{\rho(i)}$, $S'_{\psi(i)}$ et S''_i. Les cas suivants sont à considérer:

$S_{\rho(i)}$	$S'_{\psi(i)}$	S''_i	
`val` $v : \sigma$	`val` $v : \sigma'$	`val` $v : \sigma''$	On a $B \vdash \sigma \geq \sigma'$ et $B \vdash \sigma' \geq \sigma''$, d'où $B \vdash \sigma \geq \sigma''$ car $\geq$ est transitive, et le résultat attendu par la règle (sub-val).
-	-	`type` t	Trivial par (sub-mani-abstr) ou (sub-abstr-abstr).
`type` t	`type` t	`type` $t = \tau''$	De (4) il vient $B \vdash t \approx \tau''$, d'où le résultat par la règle (sub-abstr-mani).
`type` t	`type` $t = \tau'$	`type` $t = \tau''$	De (4) il vient $B \vdash t \approx \tau'$ et $B \vdash \tau' \approx \tau''$, d'où $B \vdash t \approx \tau''$ par transitivité de $\approx$ et le résultat par la règle (sub-abstr-mani).
`type` $t = \tau$	`type` t	`type` $t = \tau''$	On a $B \vdash t \approx \tau''$, et d'autre part $B \vdash t \approx \tau$ trivialement puisque B contient l'hypothèse `type` $t = \tau$. D'où le résultat par transitivité de $\approx$ et la règle (sub-mani-mani).
`type` $t = \tau$	`type` $t = \tau'$	`type` $t = \tau''$	Transitivité de $\approx$ et règle (sub-mani-mani).
`module` $X : M_1$	`module` $X : M'_1$	`module` $X : M''_1$	On a $B \vdash M_1 <: M'_1$ et $B \vdash M'_1 <: M''_1$. Les types M_1, M'_1, M''_1 étant sous-termes stricts de M, M', M'' respectivement, on peut appliquer l'hypothèse de récurrence, obtenant $B \vdash M_1 <: M''_1$ et le résultat attendu via la règle (sub-mod).

Ayant ainsi montré $B \vdash S_{\rho(i)} <: S''_i$ pour tout $i = 1, \dots, p$, nous pouvons conclure

$$E \vdash \texttt{sig}\ S_1; \dots ; S_m\ \texttt{end} <: \texttt{sig}\ S''_1; \dots ; S''_p\ \texttt{end}$$

par la règle (sub-sig); c'est le résultat attendu.

Cas $M = \texttt{functor}(X : P) \to Q$ et $M' = \texttt{functor}(X : P') \to Q'$ et $M'' = \texttt{functor}(X : P'') \to Q''$. La seule règle ayant pu conclure $E \vdash M <: M'$ et $E \vdash M' <: M''$ est la règle (sub-foncteur). On a donc:

$$E \vdash P' <: P \qquad E; \texttt{module}\ X : P' \vdash Q <: Q' \qquad E \vdash P'' <: P' \qquad E; \texttt{module}\ X : P'' \vdash Q' <: Q''$$

On a clairement $\vdash (E;\ \texttt{module}\ X : P'') <: (E;\ \texttt{module}\ X : P')$, d'où par la proposition G.2,

$$E;\ \texttt{module}\ X : P'' \vdash Q' <: Q''$$

Appliquant deux fois l'hypothèse de récurrence, on obtient des preuves de

$$E \vdash P'' <: P \qquad E;\ \texttt{module}\ X : P'' \vdash Q <: Q''$$

D'où le résultat attendu par application de la règle (sub-foncteur). □

Exercice 8.2 On définit le prédicat $E \vdash S_1 \approx S_2$ par les règles d'inférence suivantes:

$$\frac{\begin{array}{c}\rho \text{ permutation de } \{1 \dots n\} \\ E; S_1; \dots; S_n \vdash S_{\rho(i)} \approx S'_i \text{ pour } i = 1, \dots, n\end{array}}{E \vdash (\texttt{sig}\ S_1; \dots; S_n\ \texttt{end}) \approx (\texttt{sig}\ S'_1; \dots; S'_n\ \texttt{end})} \qquad \frac{E \vdash \sigma \approx \sigma'}{E \vdash (\texttt{val}\ v : \sigma) \approx (\texttt{val}\ v : \sigma')}$$

$$E \vdash (\texttt{type}\ t) \approx (\texttt{type}\ t) \qquad \frac{E \vdash \tau \approx t}{E \vdash (\texttt{type}\ t = \tau) \approx (\texttt{type}\ t)} \qquad \frac{E \vdash \tau \approx \tau'}{E \vdash (\texttt{type}\ t = \tau) \approx (\texttt{type}\ t = \tau')}$$

$$\frac{E \vdash \tau \approx t}{E \vdash (\texttt{type}\ t) \approx (\texttt{type}\ t = \tau)} \qquad \frac{E \vdash M \approx M'}{E \vdash (\texttt{module}\ X : M) \approx (\texttt{module}\ X : M')}$$

$$\frac{E \vdash P_1 \approx P_2 \qquad E;\ \texttt{module}\ X : P_2 \vdash R_1 \approx R_2}{E \vdash (\texttt{functor}\ (X : P_1) \to R_1) \approx (\texttt{functor}\ (X : P_2) \to R_2)}$$

Exercice 8.3 Informellement, toute signature qui ne contient pas de spécification de type abstrait possède toujours une signature équivalente (au sens de l'exercice 8.2) dans laquelle il n'y a pas de dépendances entre les composantes. On l'obtient en expansant de manière répétée les égalités sur les types manifestes dans le reste de la signature. Par exemple, si on part de la signature

```
module type S =
  sig
    type t = int
    type u = t
    module X : sig val v : u end
  end
```

et qu'on remplace chaque utilisation de `u` par `t`, puis chaque utilisation de `t` par `int`, on obtient la signature non-dépendante équivalente

```
  sig
    type t = int
    type u = int
    module X : sig val v : int end
  end
```

D'autre part, si un chemin p a une signature M, on peut toujours lui attribuer la signature M/p (règle de renforcement), dans laquelle toute les spécifications de types sont manifestes.

En combinant les deux remarques, l'idée est donc de remplacer chaque utilisation des règles de projections non restreintes (eq-projection), (val-projection) et (mod-projection) par une étape de renforcement (mod-renforcement), une étape de sous-typage vers un type non dépendant équivalent (mod-sub), une projection restreinte (eq-projection'), (val-projection'), (mod-projection'), et enfin une étape d'équivalence de type ou de sous-typage (equiv), (mod-sub) pour revenir au type initialement obtenu comme conclusion de (eq-projection), (val-projection), (mod-projection). Faisons le cas (val-projection) plus en détails. Considérons une occurrence de

$$\frac{E \vdash p : \texttt{sig}\ S_1^*;\ \texttt{val}\ v : \sigma;\ S_2^*\ \texttt{end} \qquad \tau \leq \sigma\{z \leftarrow p.z \mid z \text{ lié dans } S_1^*\}}{E \vdash p.v : \tau}$$

dans une dérivation. Notons $M = \texttt{sig}\ S_1^*;\ \texttt{val}\ v : \sigma;\ S_2^*\ \texttt{end}$ et M' la signature non dépendante équivalente à M/p. Par construction, M' est de la forme $\texttt{sig}\ S_1'^*;\ \texttt{val}\ v : \sigma';\ S_2'^*\ \texttt{end}$. On peut donc construire la dérivation suivante:

$$\frac{\dfrac{\dfrac{\dfrac{E \vdash p : M}{E \vdash p : M/p} \qquad E \vdash M/p <: M'}{E \vdash p : M'} \qquad \mathcal{L}(\sigma') \cap \mathcal{B}(S_1'^*) = \emptyset \qquad \tau' \leq \sigma'}{E \vdash p.v : \tau'} \qquad E \vdash \tau' \approx \tau}{E \vdash p.v : \tau}$$

Il reste bien sûr à montrer que $E \vdash \tau' \approx \tau$. Ceci découle de $E \vdash \sigma' \approx \sigma$, qui est conséquence des lemmes suivants:

1. Si $E \vdash p : M$, alors $E \vdash M \approx M/p$.

2. L'équivalence entre types de modules est transitive. D'où $E \vdash M \approx M'$.

3. Si $E \vdash (\texttt{sig}\ S_1^*;\ \texttt{val}\ v : \sigma;\ S_2^*\ \texttt{end}) <: (\texttt{sig}\ S_1'^*;\ \texttt{val}\ v : \sigma';\ S_2'^*\ \texttt{end})$, alors

$$E \vdash \sigma\{z \leftarrow p.z \mid z \text{ lié dans } S_1^*\} \approx \sigma'\{z \leftarrow p.z \mid z \text{ lié dans } S_1'^*\}$$