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 $\texttt{fun}\ x \rightarrow x.e$ 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 $\texttt{fun}\ x \rightarrow x.e$ qui s'applique à tout type enregistrement possédant un champ de nom e.
- Enregistrements *extensibles*: on peut définir et typer une fonction d'extension $\texttt{fun}\ x \rightarrow v \rightarrow x@\{e = v\}$ 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: $op ::= \ldots \mid \texttt{proj}_e \mid \texttt{exten}_e$
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 $\texttt{proj}_e(a)$. Cette expression renvoie la valeur associée à e dans l'enregistrement a.

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

Les règles de réduction pour ces opérateurs sont:

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

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:

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

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:

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

$$\begin{array}{rcl}
\mathtt{proj}_{\mathtt{e}} & : & \forall \alpha, \alpha_1, \alpha_2.\ \{\mathtt{e} : \mathtt{Pre}\ \alpha;\ \mathtt{f} : \alpha_1;\ \mathtt{g} : \alpha_2\} \rightarrow \alpha \\
\mathtt{proj}_{\mathtt{f}} & : & \forall \alpha, \alpha_1, \alpha_2.\ \{\mathtt{e} : \alpha_1;\ \mathtt{f} : \mathtt{Pre}\ \alpha;\ \mathtt{g} : \alpha_2\} \rightarrow \alpha \\
\mathtt{proj}_{\mathtt{g}} & : & \forall \alpha, \alpha_1, \alpha_2.\ \{\mathtt{e} : \alpha_1;\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \mathtt{Pre}\ \alpha\} \rightarrow \alpha \\
\mathtt{exten}_{\mathtt{e}} & : & \forall \alpha, \alpha_1, \alpha_2, \alpha_3.\ \{\mathtt{e} : \alpha_1;\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \alpha_3\} \times \alpha \rightarrow \{\mathtt{e} : \mathtt{Pre}\ \alpha;\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \alpha_3\} \\
\mathtt{exten}_{\mathtt{f}} & : & \forall \alpha, \alpha_1, \alpha_2, \alpha_3.\ \{\mathtt{e} : \alpha_1;\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \alpha_3\} \times \alpha \rightarrow \{\mathtt{e} : \alpha_1;\ \mathtt{f} : \mathtt{Pre}\ \alpha;\ \mathtt{g} : \alpha_3\} \\
\mathtt{exten}_{\mathtt{g}} & : & \forall \alpha, \alpha_1, \alpha_2, \alpha_3.\ \{\mathtt{e} : \alpha_1;\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \alpha_3\} \times \alpha \rightarrow \{\mathtt{e} : \alpha_1;\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \mathtt{Pre}\ \alpha\}
\end{array}$$

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

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

qu'avec le type

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

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:

$$\mathtt{exten}_{\mathtt{e}} \ : \ \forall \alpha, \alpha_2, \alpha_3.\ \{\mathtt{e} : \mathtt{Abs};\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \alpha_3\} \times \alpha \rightarrow \{\mathtt{e} : \mathtt{Pre}\ \alpha;\ \mathtt{f} : \alpha_2;\ \mathtt{g} : \alpha_3\}$$

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, $\{e : \mathtt{Pre}\ \mathtt{int};\ \emptyset\}$ représente les enregistrements dont le champ e contient un entier et dont tous les autres champs sont indéfinis. L'algèbre de types devient:

Types:	$\tau ::= \alpha \mid T \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$	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
	$\mid \texttt{Abs}$	le champ est absent (indéfini)
	$\mid \texttt{Pre}\ \tau$	le champ est présent (défini) avec le type τ

Les types sont identifiés modulo les deux équations suivantes:

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

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.

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

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 $\beta \leftarrow \tau'$).

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

Pour l'extension stricte, on prendrait:

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

Exemples: la fonction

```
fun r → (r.a, r.b)
```

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

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

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

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

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

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

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

Second exemple: redéfinition d'un champ existant.

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

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

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

Dernier exemple: extension dans une fonction polymorphe.

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

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

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

6.3.3 Sortes

L'algèbre de types enregistrements que nous venons d'introduire contient un certain nombre de types absurdes, comme par exemple $\emptyset \to \emptyset$ ou $\mathtt{Abs} \times \mathtt{Pre}\ \tau$ ou $\alpha \to \mathtt{Pre}\ \alpha$. 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 $\mathtt{int} \to \mathtt{bool}$;
- les rangées de types, qui peuvent apparaître à l'intérieur d'un type enregistrement $\{\ldots\}$, p.ex. $\emptyset$ ou $(e : \mathtt{Abs};\ \ldots)$.
- les infos de présence `Abs` et $\mathtt{Pre}\ \tau$, 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:

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

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:

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

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:

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

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 $\vdash \tau :: \texttt{TYPE}$.

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 $\tau \le \sigma$ comme $\tau \le \forall\alpha_1 \ldots \alpha_n.\ \tau'$ 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)).

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

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

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

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

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 $\vdash \tau :: \texttt{TYPE}$.*

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):

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

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 $(\tau, V') = \mathtt{Inst}(E(x), V)$ et $\varphi = id$.

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

- Si a est $\mathtt{fun}\ x \to a_1$:
  soit α une nouvelle variable <u>de sorte `TYPE`</u> 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 <u>de sorte `TYPE`</u> 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 $V = V_2 \setminus \{\alpha\} \underline{\setminus \mathrm{Dom}(\mu)}$.

- 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 $\mathtt{let}\ x = b\ \mathtt{in}\ c$:
  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 $V = V_2$.

- <u>Si a est $\{e_1 = a_1;\ \ldots;\ e_n = a_n\}$:</u>
  <u>vérifier que les étiquettes e_i sont deux à deux distinctes</u>
  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 $\tau = \{e_1 : \mathtt{Pre}\ (\varphi_n \circ \cdots \circ \varphi_2)(\tau_1);\ \ldots;\ e_n : \mathtt{Pre}\ \tau_n;\ \emptyset\}$ 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:

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

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*

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

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:*

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

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

Proposer un système de typage aussi flexible que possible pour cette présentation des types concrets. (Indication: on donnera aux valeurs de types concrets des types de la forme $[\tau]$, où τ est une rangée construite avec $C : \_; \_$ et $\emptyset$ 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)
```