Notes du cours du DEA :

Typage et programmation

(version provisoire)

Didier Rémy

15 janvier 1998

Table des matières

Introduction

Les langages de programmation ne sont que des approximations, rigoureuses mais incomplètes, du langage mathématique. Les langages généraux (non spécialisés à certains types de problèmes), sont complets au sens de Turing. Cela veut dire qu'ils permettent d'écrire tous les algorithmes calculables. Ils ne sont cependant pas tous équivalents, car ils diffèrent énormément par leur expressivité, c'est-à-dire leur capacité d'exprimer succinctement certains algorithmes. Nous sommes donc poussés à une recherche —sans fin ?— de nouvelles structures de programmation, plus abstraites, permettant de décrire des algorithmes de façon plus concise. Nous ne recherchons pas non plus l'expressivité à tout prix, et nous efforçons de n'introduire que des constructions essentielles, simples, bien formalisés et sures.

Donner la syntaxe d'un langage n'est qu'un préliminaire à sa définition. Pour parler de programme il faut donner un sens aux expressions du langage, c'est-à-dire définir leur sémantique. Par exemple, cela peut se faire en décrivant l'évaluation des expressions. Parce que pour être intéressant, les langages de programmation généraux sont complets au sens de Turing, il faut abandonner tout espoir de pouvoir décider de propriétés importantes telles que la terminaison d'un programme. Pour des raisons analogues, la bonne exécution d'un programme est souvent une propriété indécidable. La sûreté de l'exécution ne doit évidemment pas être abandonnée au profit de l'expressivité. Il est essentiel de conserver l'expressivité tout en se limitant à une sous-classe décidable des programmes qui s'évaluent normalement.

Le typage est un moyen de définir un sous-ensemble des expressions bien formées par un critère décidable, et en général simple. On attend du typage qu'il garantisse la sécurité de l'évaluation. Nous considérerons surtout des systèmes de typage implicites, c'est-à-dire que les types ne font pas partie des programmes, mais seront synthétisés.

Ainsi, au travers de la formalisation du langage ML, de son typage et de sa sémantique, ces notes de cours ont pour but principal de montrer l'apport du typage dans les langages de programmation et de fournir les techniques et outils nécessaires permettant d'étudier l'ajout au langage de nouvelles constructions, ainsi que d'aborder l'étude d'autres langages de façon rigoureuse.

L'unité du cours est le langage ML, et tous les aspects étudiés seront présentés à partir d'extensions de ce langage. On peut distinguer nettement trois parties. Les chapitres 1 à 4 formalisent le langage ML, en partant du noyau, puis en ajoutant de façon relativement modulaire les autres constructions. Les points importants sont le système de typage de ML, la synthèse des types, sa sémantique opérationnelle et la sûreté du calcul garantie par le typage.

Dans une seconde partie (chapitre 5 à ??) nous nous intéressons à des extensions avancées du langage, et essentiellement de son système de typage. Nous y décrivons le typage des objets enregistrements. Puis, nous ajoutons un mécanisme de sous-typage, et nous montrons que les propriétés essentielles du système de type de ML sont conservées. Nous décrivons très brièvement d'autres extensions du système de typage (types dynamiques, types existentiels et universels déclarés, types d'ordre supérieur, types intersections).

Enfin la troisième partie est consacrée à un aspect essentiel des langages de programmation : la modularité. Nous y présentons un système de module pour ML mais de façon aussi indépendante du langage ML que possible.

Tout au long de ce cours nous proposons des exercices. La plupart sont annotés de (*) à (***) selon leur niveau de difficulté, et nous en donnons un corrigé (parfois succinct) en annexe. D'autres ne sont pas annotés et ne sont pas corrigés. Ce sont soit des exercices très simples qui ne méritent pas de correction, ou bien au contraire des exercices plus long, mais pas nécessairement difficiles.

Chapitre 1

Présentation du langage ML

Note

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

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

1.1 Rappel du noyau ML

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

$$
\begin{array}{rll}
a ::= & z & \\
| & \texttt{fun}\ (x)\ a & \text{Abstractions} \\
| & a_1\ a_2 & \text{Applications} \\
| & \texttt{let}\ x = a_1\ \texttt{in}\ a_2 & \text{Déclarations} \\
z ::= & x & \text{Variables} \\
| & c & \text{Constantes} \\
c ::= & C & \text{Constructeurs} \\
| & f & \text{Primitives}
\end{array}
$$

Parmi, les constantes nous distinguons les *constructeurs* et les *primitives*. Chaque constante est donnée avec une arité. Nous donnerons plus loin une distinction précise entre constructeurs et primitives. Pour l'instant, nous pouvons nous contenter de l'idée simple suivante : les primitive complètement appliqué (i.e. autant de fois que leur arité) ne sont pas des valeurs ; les primitives partiellement appliquées ou les constructeurs, complètement ou partiellement appliqués, sont des valeurs. Par exemple, les entiers, les chaînes de caractères, le symbole de tuple sont des constructeurs ; l'addition, le successeur, la concaténation sont des primitives. Nous désignons par la lettre z une variable ou une constante, indifféremment. Nous notons également c^k, f^k ou C^k pour préciser que la constante, le constructeur ou la primitive est d'arité k. Certaines constantes sont parfois notées de façon particulières. Par exemple l'addition est notée $(\_ + \_)$; cela est simplement une convenance de notation qui permet d'écrire $(a_1 + a_2)$ au lieu de $(\_ + \_)\ a_1\ a_2$, mais nous considérerons toujours les deux notations comme équivalentes.

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

1.2 Sémantiques opérationnelles

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

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

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

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

Dans ce chapitre nous présenterons uniquement la sémantique à grands pas, plus concrète, et nous l'utiliserons pour définir un évaluateur du langage. La sémantique à petits pas, plus formelle et plus précise, sera décrite dans le chapitre 3, et utilisée pour montrer que le typage garantit le bon déroulement des calculs.

1.2.1 Sémantique à grands pas

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

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

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

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

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

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

$$\text{(App-Err-Gen)}\quad \frac{a \vdash a_1 \Rightarrow v \qquad r \neq \texttt{closure}\ (e', x, a)}{a \vdash a_1\ a_2 \Rightarrow \texttt{error}} \qquad \text{(App-Err-Prop-1)}\quad \frac{a \vdash a_1 \Rightarrow \texttt{error}}{a \vdash a_1\ a_2 \Rightarrow \texttt{error}}$$

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

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

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

Évaluateur

Évaluations infinies

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

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

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

Il est possible d'étendre la sémantique naturelle avec des variables logiques pour rendre compte de la recherche d'une dérivation infinie, et donc parler de la divergence. On peut alors montrer qu'il est toujours possible d'étendre une évaluation partielle. Pour en savoir plus à ce sujet, on pourra lire [GR93].

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

1.3 Typage

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

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

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

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

1.4 Le système de type de ML

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

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

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

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

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

$$\text{(Fun)}\quad \frac{A \oplus (x : \tau_2) \vdash a : \tau_1}{A \vdash \texttt{fun}\ (x)\ a : \tau_2 \to \tau_1} \qquad \text{(App)}\quad \frac{A \vdash a_1 : \tau_2 \to \tau_1 \qquad A \vdash a_2 : \tau_2}{A \vdash a_1\ a_2 : \tau_1}$$

$$\text{(Var-Inst)}\quad \frac{z : \forall\, \bar{\alpha}.\, \tau_0 \in A}{A \vdash z : \tau_0[\bar{\tau}/\bar{\alpha}]} \qquad \text{(Gen-Let)}\quad \frac{A \vdash a_1 : \tau_1 \qquad A \oplus (x : \mathrm{Gen}(\tau_1, A)) \vdash a_2 : \tau_2}{A \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

Figure 1.1 : Règles de typage pour le noyau ML.

1.5 Propriétés du typage

La relation de typage vérifie les propriétés suivantes :

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

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

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

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

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

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

Chapitre 3

Sémantique à réduction et correction du typage

Dans le premier chapitre, nous avons introduit le langage ML et nous l'avons défini en donnant une sémantique à grands pas ; nous en avons déduit un évaluateur pour ML. Nous avons constaté que la sémantique à grands pas ne permet pas de parler des programmes qui bouclent.

Dans ce chapitre, nous présentons une autre sémantique opérationnelle, la sémantique à réduction (ou encore à petits pas) qui est une alternative à la sémantique à grands pas. Elle est plus simple, plus précise, plus modulaire, mais elle est moins concrète, c'est-à-dire qu'elle modélise le calcul de façon plus éloignée d'un interprète.

3.1 Sémantique à petits pas

Dans le λ-calcul, une étape d'évaluation correspond à une β-réduction. À partir d'un terme, il existe en général plusieurs chemins de réduction possibles. Le λ-calcul possède la propriété Church-Rosser qui énonce qu'il est toujours possible de rejoindre deux chemins. Ainsi si un terme est fortement normalisable, c'est-à-dire si tous les calculs terminent, alors ils terminent sur une même forme normale. Le calcul est déterministe.

La réduction dans le λ-calcul est une très bonne modélisation de l'évaluation. Il n'est pas nécessaire que la réduction soit Church-Rosser pour modéliser le calcul. D'une part, il est en général possible de limiter le nombre de chemins autorisés en limitant les contextes dans lesquels la réduction peut se faire (on choisira par exemple une stratégie d'évaluation en appel par valeur), ce qui suffit bien souvent à rendre le calcul déterministe. D'autre part, dans une langage typé, il est souvent possible de modéliser un calcul non-déterministe et de montrer la conservation du typage le long de tous les chemins de réduction possibles ; cela permet non seulement de partager les preuves de correction de plusieurs stratégies, mais également d'énoncer un résultat de correction plus fort.

Dans une sémantique à réduction, les résultats de l'évaluation sont les valeurs. Dans le λ-calcul, ce sont donc les fonctions. En ML, il existe d'autres valeurs que les fonctions, par exemple les constantes, ou plus généralement les valeurs construites.

Donner une sémantique à réduction à petits pas pour le langage ML, c'est donc définir, comme pour le λ-calcul, une relation de réduction des programmes sur eux-mêmes. Lorsque c'est possible, il est commode de définir cette relation de la façon modulaire suivante.

1. Définir les valeurs

Tout d'abord nous devons définir le sous-ensemble des expressions qui sont des *valeurs* :

$$
\begin{array}{lll}
v ::= c^k\ v_1 \ldots\ v_q & (q < k) & \text{constantes partiellement appliquées} \\
\quad\mid C^k\ v_1 \ldots\ v_q & (q \leq k) & \text{valeurs construites} \\
\quad\mid \texttt{fun}\ (x)\ a & &
\end{array}
$$

Nous précisons ainsi la distinction entre constructeurs et primitives. Lorsqu'ils sont totalement appliqués à des valeurs, les constructeurs forment des valeurs, mais pas les primitives. Partiellement appliquées, les constantes sont toujours des valeurs. Par exemple, une paire $(\_,\_)\ 1\ 2$, notée $(1, 2)$ est une valeur, mais une addition $(\_+\_)$, notée $1 + 2$ n'est pas une valeur.

Nous dirons que les valeurs de la forme $C^k\ v_1 \ldots\ v_k$ sont des *valeurs construites* et que les autres valeurs sont des *valeurs fonctionnelles.*

2. Définir les radicaux et leurs réduits

Les réductions sont d'abord définies pour chaque radical. Pour simplifier la lecture, nous écrirons $a_1 \xrightarrow{\epsilon} a_2$ pour les réductions en tête, i.e. les réductions de radicaux, et plus généralement $a_1 \longrightarrow a_2$ pour une réduction à une position quelconque dans le terme. Le noyau ML sans constantes n'est qu'un sous-ensemble du λ-calcul non typé, et la seule réduction de radical est la β-réduction, qui prend toutefois deux formes syntaxiques différentes (la construction LET est une façon de marquer certains radicaux présents dans le programme initial) :

$$(\texttt{fun}\ (x)\ a)\ v \xrightarrow{\epsilon} a[v/x] \qquad (\beta_v)$$
$$\texttt{let}\ x = v\ \texttt{in}\ a \xrightarrow{\epsilon} a[v/x] \qquad (\beta_{\texttt{let}})$$

Dans l'écriture de la β-reduction, a et v sont quelconques, donc les règles s'appliquent pour toute expression a et toute valeur v. Bien entendu, le renomage $[v/x]$ ne doit substituer que les occurences de la variable x qui sont libres dans a.

Nous ajoutons également un ensemble de δ-règles qui décrivent la façon de réduire les primitives. Elles sont de la forme :

$$f^k\ v_1\ \ldots v_k \xrightarrow{\epsilon} a \qquad (\delta_f)$$

On note aussi parfois $a_1 \xrightarrow{\delta} a_2$ pour dire que $a_1 \xrightarrow{\epsilon} a_2$ par une δ-règle. Attention ! ici les v_i et a ne sont pas des paramètres formels universellement quantifiés, mais des valeurs précises qui dépendent de la δ-règle. Une δ-règle associée à une primitive f

d'arité k est donc définie par une paire $(\bar{v}, a)$ où $\bar{v}$ est un k-tuple de valeurs. Une primitive peut avoir plusieurs δ-règles associées, voir même une infinité (dénombrable). Par exemple, nous verrons ci-dessous que les paires peut être modélisées par un constructeur $(\_,\_)$ et deux primitives `fst` et `snd`. Nous avons alors pour toutes valeurs v_1 et v_2 deux δ-règles

$$\texttt{fst}\ (v_1, v_2) \xrightarrow{\delta} v_1 \qquad\qquad \texttt{snd}\ (v_1, v_2) \xrightarrow{\delta} (v_1, v_2)$$

Il existe donc une infinité de δ-règles associées à la primitive `fst`. Cependant, dans certains cas comme ci-dessus pour `fst`, il existe toutes les règles ont un même forme, et on peut les voir comme une seule règle où v_1 et v_2 ne sont plus des valeurs mais des méta-variables de valeurs universellement quantifiés (comme pour la β-règle). Ceci n'est pas toujours possible. Par exemple, la fonction successeur peut être modélisée par une infinité de réductions $\texttt{succ}\ 0 \xrightarrow{\delta} 1$, $\texttt{succ}\ 1 \xrightarrow{\delta} 2$, etc. Cependant on ne peut plus regrouper ces règles en une seule règle avec des méta-variables de valeurs comme pour `fst`.

A l'inverse, les constructeurs n'ont jamais de règles de réduction associées, puisque ce sont des valeurs.

Décrire les contextes d'évaluation

Il faut maintenant décrire les endroits où il est possible de réduire un radical. Pour cela, on définit un ensemble de *contextes d'évaluation*. Dans le λ-calcul, les contextes d'évaluation sont tous les contextes à un trou. En ML, la stratégie d'évaluation est l'appel par valeur et les contextes d'évaluation par valeur sont :

$$E ::= [] \mid E\ a \mid v\ E \mid \texttt{let}\ x = E\ \texttt{in}\ a$$

Permettre de réduire dans un contexte d'évaluation

Pour cela, on ajoute la règle d'évaluation suivante qui exprime que la réduction peut se produire également à l'intérieur d'un terme :

$$\text{si } a_1 \xrightarrow{\epsilon} a_2, \text{ alors } E[a_1] \longrightarrow E[a_2] \qquad (\textsc{Contexte})$$

Formellement la relation de réduction $\longrightarrow$ est définie comme la plus petite relation satisfaisant les règles β, β_{Let} et Contexte et contenant l'ensemble de toute les δ-règles données en hypothèses.

La plupart des langages de programmation ont une sémantique déterministe, c'est-à-dire qu'il n'y a qu'une seule évaluation possible. Toutefois, certains langages ont une sémantique non déterministe. Soit parce que la sémantique n'est pas entièrement spécifiée, par exemple l'ordre d'évaluation des arguments, soit parce que cela est essentiel, par exemple dans un langage parallèle.

Lorsque la sémantique d'un langage est déterministe, c'est souvent parce qu'il n'existe qu'un chemin de réduction possible, ce qui découle en général immédiatement de la définition des contextes d'évaluation. Il se peut aussi qu'il existe plusieurs chemins de réduction qui mènent à la même valeur, ou divergent tous simultanément. Mais ceci n'est même pas vrai pour le λ-calcul, car selon l'ordre d'évaluation des arguments, un programme peut diverger ou terminer.

Exercice 1 (*) *Dans le λ-calcul réduire le terme* `let` $f =$ `fun` $(y)\ y$ `in` $($`fun` $(z)\ z)\ f$ *de deux façons différentes, et vérifier que les deux calculs aboutissent au même résultat. Vérifier que le programme ci-dessus ne peut plus être réduit que d'une seule façon en appel par valeur.* □

Exercice 2 (*) *Plus généralement, vérifier que les contextes d'appel par valeur ci-dessus ne permettent qu'un seul chemin de réduction. Le calcul est donc trivialement déterministe.* □

Exercice 3 (*) *Quelles modifications faut-il apporter à la définition de la sémantique à petits pas pour modéliser l'appel par nom (on ignorera les constantes) ?* □

Les évaluations possibles d'un programme sont :

$$a \longrightarrow a_1 \longrightarrow \ldots \begin{cases} v & \text{le programme s'évalue en la valeur } v \\ a_n(\neq v \wedge \not\longrightarrow a_{n+1}) & \text{l'évaluation est bloquée} \\ a_n \longrightarrow \ldots & \text{l'évaluation se poursuit indéfiniment} \end{cases}$$

L'erreur est simplement modélisée par une évaluation bloquée. Pour montrer qu'un programme ne produit pas d'erreur à l'exécution, il faut montrer qu'une réduction quelconque issue de ce programme ne peut jamais être bloquée.

S'il n'y pas de constante, cela est immédiat, même pour les programmes non typables. Dès que le langage contient des constantes, il est possible que des programmes soient bloqués. Par exemple, le programme $c\ a$ si c est une constante d'ordre 0 qui ne possède pas de δ-règle.

Exercice 4 (*) *Vérifiez que pour qui ne contiennent pas de constantes, l'évaluation ne peut jamais être bloquée, que ces termes soit ou non bien typés.* □

Exercice 5 (programmation) *On se propose d'écrire un programme qui évalue les expressions du noyau ML en suivant les définitions de ce chapitre. Écrire un type concret* `expr` *représentant les expressions a du langage noyau. Les valeurs seront représentées par un prédicat* `est_une_valeur` *sur les expressions. Un contexte E pourra être représenté par l'expression* `fun` $(x)\ E[x]$. *Écrire une fonction qui retourne les décompositions possibles d'un terme a en paires (E, a') telles que $E[a']$ est égal à a et a' n'est pas une valeur. Définir les règles de réduction (c'est une fonction prenant un terme a et*

Écrire une fonction qui calcule la substitution de x par v dans a. Une règle de réduction est une fonction qui associe un programme a_2 à un programme a_1 où échoue *si a_1 n'est pas le radical correspondant. Définir les règles de réduction de radicaux indépendemment les unes des autres. Définir une étape de réduction (en prenant en paramètre une liste de règle de réduction). Enfin définir la réduction comme une succession d'étapes de réductions.*

Une constante sera représentée par un identificateur, une arité, et une information dépendant du type primitive ou constructeur. En particulier, les primitives seront données avec leurs delta-règles. (i.e. représentés par une fonction).

On se donnera les entiers et les booléens comme constantes initiales. Évaluer quelques exemples, en traçant les différentes étapes de la réduction (on pourra aussi marquer les expressions réduites). □

3.2 Correction du typage vis à vis de la sémantique

La correction du typage revient à montrer que les programmes bien typés ne sont jamais bloqués. Ceci est garanti par les deux propriétés suivantes :

1. Si $A \vdash a_1 : \tau$ et $a_1 \longrightarrow a_2$ alors $A \vdash a_2 : \tau$.

2. Si $A_0 \vdash a : \tau$ et $a \not\longrightarrow$ alors a_1 est une valeur.

où A_0 est l'environnement composé uniquement de déclarations de constantes.

La première propriété énonce que le typage est conservé par réduction. Cette propriété est en général indépendante de la stratégie d'évaluation. La seconde dit simplement que les programmes bien typés ne sont pas immédiatement erronés. Nous pouvons souligner ici l'avantage d'une sémantique à petits pas. La première propriété garantit la correction de l'évaluation, même pour les programmes qui ne terminent pas. En particulier, elle assure que l'évaluation infinie d'un programme bien typé qui diverge se déroule aussi normalement.

Pour simplifier, il est commode de définir une relation de typabilité entre les expressions : nous dirons que l'expression a_2 est au moins aussi typable que l'expression a_1, et nous noterons $a_1 \sqsubseteq a_2$, si

$$\forall A, \forall \tau, (A \vdash a_1 : \tau) \Longrightarrow (A \vdash a_2 : \tau)$$

On peut voir ici la relation de typabilité comme induite par l'inclusion des ensembles de typage d'une expression, c'est-à-dire l'ensemble des paires (A, τ) telles que $A \vdash a : \tau$.

Nous avons vu que la présence de primitives s'accompagne, en général, de δ-règles. Il est clair que ces règles ne peuvent pas être arbitraires sans invalider la correction du typage.

Hypothèse Étant donné un environnement de typage initial A_0, nous faisons les hypothèses suivantes.

H0 On suppose que le domaine de A_0 est restreint à des constantes, que $A_0(C^n)$ est de la forme $\forall\,\bar{\alpha}.\,\tau_1 \rightarrow \ldots \tau_n \rightarrow \tau_0$ et τ_0 n'est pas une flèche.

H1 si $a_1 \longrightarrow_\delta a$, alors $a_1 \sqsubseteq a_2$ et

H2 si $A_0 \vdash f^k\ v_1 \ldots v_k : \tau$, alors il existe une expression a telle que $f^k\ v_1 \ldots\ v_k \longrightarrow_\delta a$.

La première affirme que les δ-règles préservent le typage. C'est clairement nécessaire, puisque nous voulons montrer que cette propriété s'étend à une réduction quelconque. La seconde hypothèse assure qu'une application complète et bien typée d'une primitive à des valeurs peut toujours se réduire. C'est-à-dire, plus informellement, qu'il existe assez de δ-règles pour réduire les primitives.

Les deux théorèmes 1 et 2 ci-dessous expriment la sécurité du typage :

Théorème 1 (Auto-réduction) *La réduction conserve les typages (i.e. $\longrightarrow$ est une sous relation de $\sqsubseteq$).*

En utilisation la notation mathématique (une relation est un graphe) il s'agit donc de montrer $(\longrightarrow) \subset (\sqsubseteq)$. Comme la relation $\longrightarrow$ est la plus petite relation satisfaisant les règles β, δ et Contexte, il suffit en fait de montrer que

Le théorème d'auto-réduction (*subject reduction*) découle donc de immédiatement de la croissance de $\sqsubseteq$ (lemme 4) et du lemme de contraction des radicaux (lemme 5) qui suivent.

Par construction, la relation $\sqsubseteq$ est réflexive et transitive ; le lemme de remplacement énonce qu'elle est aussi croissante (avec la structure de termes).

Lemme 4 (Croissance de $\sqsubseteq$) *Pour tout contexte E, si $a_1 \sqsubseteq a_2$ alors $E[a_1] \sqsubseteq E[a_2]$.*

<u>Preuve</u> : Par récurrence structurelle sur le contexte E. Soit A un environnement de typage et τ un type tels que $A \vdash E[a_1] : \tau$ (1). Nous montrons que $A \vdash E[a_2] : \tau$.

Cas E est $[\,]$: trivial.

Cas E est `let` $x = E'$ `in` a : Une dérivation de (1) se termine par :

$$\frac{A \vdash E'[a_1] : \tau_1 \ (2) \qquad A \oplus (x : \mathrm{Gen}(\tau_1, A)) \vdash a : \tau}{A \vdash \texttt{let}\ x = E'[a_1]\ \texttt{in}\ a : \tau}\ (\textsc{Let})$$

Par l'hypothèse de récurrence appliquée à (2), nous savons que $A \vdash E' a_2 : \tau_1$, et la conclusion suit en remplaçant a_1 par a_2 dans la dérivation ci-dessus.

Cas E est $E'\ a_1$ ou $v\ E'$: Ces deux cas sont aussi simples que celui du Let. ∎

Lemme 5 (Contraction de radicaux) *Si $a_1 \longrightarrow_\epsilon a_2$ alors $a_1 \sqsubseteq a_2$.*

Chaque cas de radical peut être traité indépendamment. Pour les radicaux δ, il suffit d'appliquer l'hypothèse $(H1)$. Pour les radicaux β, on utilise le lemme de remplacement dans un contexte.

Lemme 6 (Remplacement dans un terme) *Si les formules $A \vdash v : \tau_0$ (1) et $A \oplus (x : \forall \bar\alpha_0 .\, \tau_0) \vdash a : \tau$ (2) sont prouvables et si les variables $\bar\alpha_0$ ne sont pas libres dans A, alors $A \vdash a[v/x] : \tau$ (3) est prouvable.*

En fait, on pourrait montrer ce lemme dans le cas plus général où v est remplacé par une expression b dont les variables libres ne sont pas liées dans a.

Preuve : La preuve se fait par récurrence sur la structure de a. Nous écrivons A_x pour $A \oplus (x : \forall \bar\alpha_0 .\, \tau_0)$.

Cas a est x : Les termes $a[v/x]$ et v sont égaux. De (2), nous savons que τ est τ_0 où toutes les variables $\bar\alpha_0$ ont été substituées. La stabilité par substitution du typage appliquée à (1) montre (3).

Cas a est x_1 et $x_1 \neq x$: Les termes $a[v/x]$ et a sont identiques et x_1 n'y figure pas libre. Il suffit d'appliquer le lemme d'extension de contexte à (2).

Cas a est $\texttt{let}\ x_1 = a_1\ \texttt{in}\ a_2$: Une dérivation de (2) se termine par :

$$\frac{(4)\ \ A_x \vdash a_1 : \tau_1 \qquad (5)\ \ A_x \oplus (x_1 : \mathrm{Gen}(\tau_1, A_x)) \vdash a_2 : \tau}{A_x \vdash \texttt{let}\ x_1 = a_1\ \texttt{in}\ a_2 : \tau}\ (\textsc{Let})$$

Par l'hypothèse de récurrence appliquée à (4), nous savons que $A \vdash a_1[v/x] : \tau_1$ (6).

Si x_1 est égal à x, alors $a[v/x]$ est égal à $\texttt{let}\ x = a_1[v/x]\ \texttt{in}\ a_2$. La premisse (5) s'écrit alors $A \oplus (x : \mathrm{Gen}(\tau_1, A_x)) \vdash a_2 : \tau$. Par généralisation de contexte (propriété **??**, page) nous avons $A \oplus (x : \mathrm{Gen}(\tau_1, A)) \vdash a_2 : \tau$ (70), puisque A est une sous-séquence de A_x. On conlut par l'application d'une règle Let avec les prémisses (6) et (70).

Sinon, par généralisation du contexte (propriété 3, page 13) appliqué à (5), nous déduisons $A_x \oplus (x_1 : \mathrm{Gen}(\tau_1, A)) \vdash a_2 : \tau$, puisque A est une sous-séquence de A_x. Notons A_1 le contexte $A \oplus (x_1 : \mathrm{Gen}(\tau_1, A))$. Par réarrangement du contextes (propriété 2, page 12), on obtient $A_1 \oplus (x : \forall \bar\alpha_0 .\, \tau_0) \vdash a_2 : \tau$ (8). Comme v est une valeur, x_1 n'est pas libre dans v. Donc, par extension de contexte (propriété 2, page 12) appliqué à (7), nous avons $A_1 \vdash v : \tau_0$ (9). Les variables $\bar\alpha_0$ ne sont pas libre dans A_1, car les variables libres dans A_1 sont aussi libres dans A. En appliquant l'hypothèse de récurrence à (9) et (8), nous obtenons :

$$A_1 \vdash a_2[v/x] : \tau \ (10)$$

Nous concluons par l'application de la règle Let à (10) et (6).

Cas a est $\texttt{fun}\ (x_1)\ a_1$: Si les variables de $\bar\tau_0$ apparaissent dans τ, nous pouvons les renommer en des variables prisent en dehors des variables libres de A par une substitution θ. Sinon, nous prenons θ égale à l'identité. Par application de la substitution θ (propriété 1, page 12) au jugement2, nous avons $A \vdash a : \theta(\tau)$.

Nécessairement, $\theta\tau$ est de la forme $\tau_1 \to \tau_2$, et il existe une dérivation de (2) qui se termine par :

$$\frac{(11)\ \ 1A_x \oplus (x_1 : \tau_1) \vdash a_1 : \tau_2}{1\ \ A_x \vdash \texttt{fun}\ (x_1)\ a_1 : \tau_1 \to \tau_2}\ (\textsc{Let})$$

Si x_1 est égal à x, alors $a[v/x]$ est égal à a, et il n'y a rien à montrer.

Sinon, notons A_1 le contexte $A \oplus x_1 : \tau_1$. Par réarrangement du contexte (propriété 2), appliquée à (111), nous avons $A_1 \oplus (x : \forall \alpha_0 . \tau_0) \vdash a_1[v/x] : \tau$ (122).

Comme ci-dessus pour le cas LET, nous avons $A_1 \vdash v : \tau_0$ (13). Comme nous avons supposé les variables $\bar{\tau}_0$ ne sont pas libres dans τ, donc en particulier τ_1, elles ne sont pas non plus libres dans A_1. En appliquant l'hypothèse de récurrence à (13) et (122), nous obtenons :

$$A_1 \vdash a_1[v/x] : \tau.$$

Par application de la règle FUN, nous obtenons $A \vdash a[v/x] : \theta\tau$. Nous concluons par application de la substitution θ^{-1} (propriété 1, page 12).

Cas a est $a_1\ a_2$ ou $\texttt{fun}\ (x)\ a_1$: Ces cas sont plus faciles que le cas LET.

■

<u>Preuve</u> : (de la contraction de radical) Soit A un environnement de typage et τ un type tel que $A \vdash a_1 : \tau$ (1). Nous montrons que $A_2 \vdash a_2 : \tau$ (2) par cas sur le radical a_1. Chaque cas peut être vérifié indépendamment.

Cas a_1 est $(\texttt{fun}\ (x)\ a)\ v$: Une dérivation de (1) se termine par :

$$\dfrac{\dfrac{(3)\ \ A_1 \oplus (x : \tau_1) \vdash a : \tau}{A_1 \vdash a : \tau_1 \to \tau}\ (\textsc{Fun}) \qquad (4)\ \ A_1 \vdash v : \tau_1}{A_1 \vdash a_1 : \tau}\ (\textsc{App})$$

Le lemme précédent appliqué à (4) et (3) conduit directement à la conclusion.

Cas a_1 est $\texttt{let}\ x = v\ \texttt{in}\ a$: Une dérivation de (1) se termine par :

$$\dfrac{(5)\ \ A_1 \vdash v : \tau_1 \qquad (6)\ \ A_1 \oplus (x : \mathrm{Gen}(\tau_1, A)) \vdash a : \tau}{A_1 \vdash \texttt{let}\ x = v\ \texttt{in}\ a : \tau}\ (\textsc{Let})$$

Nous concluons en appliquant le lemme de remplacement de termes à (5) et (6) avec A_1 pour A_2.

Cas a_1 est $c^k\ v_2 \ldots\ v_k$: Il s'agit donc d'une réduction $a_1 \longrightarrow_\delta a_2$. La première partie de l'hypothèse sur les δ-règles fournit directement la conclusion.

■

En utilisant une généralisation du lemme 6, il n'est pas nécessaire que l'expression liée par un LET soit une valeur. On pourrait montrer que le typage est conservé par réduction forte des expressions de liaisons, c'est-à-dire en effectuant la réduction

$$\texttt{let}\ x = a_1\ \texttt{in}\ a_2 \longrightarrow a_2[a_1/x]$$

dans un contexte quelconque. Notons $\hat{a}$ le terme obtenu, qui ne contient donc plus de liasons LET. Nous avons $a \sqsubseteq \hat{a}$, et puisque $\hat{a}$ ne comporte plus de LET, l'expression $\hat{a}$ est typable dans le calcul simplement typé.

On peut se demander si l'inverse de l'auto-réduction est également vérifié, ce qui signifierait que la réduction n'ajoute pas de typages, c'est-à-dire si

$$a_1 \longrightarrow a_2 \overset{?}{\Longrightarrow} a_2 \sqsubseteq a_1$$

Cette propriété est fausse car la réduction perd des morceaux de termes : il suffit de considérer, par exemple, le programme $(\texttt{fun}\ (x)\ a_1)\ a_2$ où le programme a_1 est typable, mais pas a_2. Toutefois, on pourrait montrer qu'en remplaçant la règle GEN-LET par la règle suivante

$$\frac{A \vdash a_1 : \tau_1 \qquad A \vdash a_2[a_1/x] : \tau_2}{A \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

on conserverait les mêmes typages.

Théorème 2 (Formes Normales) *Les expressions irréductibles et bien typées dans l'environnement initial sont des valeurs.*

Ce théorème énonce que les termes bien typés qui ne peuvent être réduits sont des valeurs, en conséquence l'évaluation des termes bien typés n'est jamais bloquée.

Preuve : Nous montrons d'abord que si $A_0 \vdash v : \tau$ alors

- Si τ est un type fonctionnel, alors v est une valeur fonctionnelle,
- Si τ est une valeur construite, alors v est une valeur construite.

Il suffit pour cela de montrer que :

- Si v est une valeur fonctionnelle, alors τ est un type fonctionnel,
- Si v est une valeur construite, alors τ est une valeur construite.

En effet 1- les valeurs fonctionnelles et les valeurs construites forment une partition de l'ensemble des valeurs, et 2- les types fonctionnnel et les types construits forment une partition (d'un sous ensemble des types). Chaque cas se vérifie très facilement par analyse de la dérivation de typage.

La preuve est une récurrence structurelle sur a. Nous supposons que $a : \tau$ (1) pour un certain type τ et que a ne peut pas être réduit.

Cas a est z : Puisque a est clos, z est une constante ; c'est donc une valeur.

Cas a est $a_1\ a_2$: Une dérivation de (1) montre qu'il existe un type τ_1 tel que $a_1 : \tau_1 \rightarrow \tau$ et $a_2 : \tau_2$. L'hypothèse de récurrence appliquée à chacune des prémisses montre que a_1 et a_2 sont des valeurs. En particulier, comme a est irréductible, a_1 ne peut pas être une fonction ; c'est donc une constante partiellement appliquée, c'est-à-dire de la forme $c^k\ v_1 \ldots v_q$; ainsi a est de la forme $c^k\ v_1 \ldots v_q\ v$. Comme a est irréductible, l'hypothèse (**H2**) implique que c n'est pas une primitive complétement appliquée, c'est donc un constructeur, ou une primitive patiellement appliquée dans a. Dans les deux cas, a est une valeur.

Cas a est $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$: Une dérivation de (1) se termine par :

$$\frac{(1)\ \ A \vdash a_1 : \tau_1 \qquad \ldots}{A \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}\ (\text{LET})$$

L'hypothèse de récurrence appliquée à (1) montre que a_1 est une valeur. Donc a pourrait être réduit.

Cas a **est** `fun` (x) a_1 **:** Alors a est une valeur.

■

3.3 Quelques extensions “gratuites” du langage

Pour illustrer l'intérêt d'avoir paramétrer la définition du langage et donc les preuves par un ensemble de primitives, nous décrivons quelques instances.

L'ajout des paires est la plus simple sans doute, et nous la traitons en détails. D'un autre coté, une extension très générale consiste à ajouter des types concrets et du filtrage. Nous laissons en exercice l'ajout de booléens. Nous la traitons dans un cas particulier. Les booléens, les paires, etc. sont des cas particulier de types concrets.

Les paires

Pour étendre le langage avec des paires, il nous faut d'abord choisir un constructeur de type pour les paires.

$$g ::= (\_ * \_)$$

puis un constructeur de valeurs $(\_, \_)$ d'arité 2 et deux fonctions de projections (i.e. deux primitives) `fst` et `snd` avec les δ-règles suivantes :

$$\begin{aligned} \texttt{fst}\ (v_1, v_2) &\longrightarrow_r v_1 && (\textsc{Fst}) \\ \texttt{snd}\ (v_1, v_2) &\longrightarrow_r v_2 && (\textsc{Snd}) \end{aligned}$$

Le typage des primitives est donné par l'environnement de typage initial suivant :

$$\begin{aligned} (\_, \_) &: \forall\, \alpha_1, \alpha_2.\, \alpha_1 \rightarrow \alpha_2 \rightarrow (\alpha_1 * \alpha_2) \\ \texttt{fst} &: \forall\, \alpha_1, \alpha_2.\, (\alpha_1 * \alpha_2) \rightarrow \alpha_1 \\ \texttt{snd} &: \forall\, \alpha_1, \alpha_2.\, (\alpha_1 * \alpha_2) \rightarrow \alpha_2 \end{aligned}$$

Pour bénéficier de l'étude précédente, et en déduire la correction du typage pour le langage avec paires, il suffit de vérifier que les hypothèses sont satisfaites.

<u>Preuve</u> : En effet,

H0 : Cette hypothèse est satisfaite par construction.

H1 : En analysant pour chaque δ-règles les dérivations de typage possibles pour le terme de gauche, et on en déduit facilement le typage du terme de droite.

H2 : On remarque que le la seule valeur construite est la paire Les valeurs bien typés dans l'environnement initial de la forme qui ont un type construit sont donc des paires.

Nous pouvons maintenant vérifier la seconde proporiété par cas sur chaque constante :

Sous-cas $(v_1, v_2)\ v_3$ **:** Ceci n'est pas possible car ce programme n'est pas typable.

Sous-cas `fst` v **:** Une dérivation de ce programme impose que $A_x \vdash v : \tau_1 * \tau_2$. Une valeur de ce type est nécessairement de la forme (v_1, v_2), donc le programme initial est réductible.

Sous-cas `snd` v **:** Ce cas est symétrique du précédent.

■

Exercice 6 *Formaliser directement le cas des booléens. Vérifier que toutes les propriétés restent valables (refaire leur démonstration, en faisant bien attention à leur structures ; on laisser de coté les branches de la preuve inchangée par rapport au noyau).* □

Exercice 7 (**) *Définir les paires de façon primitive, en étendant le langage. Vérifier que les preuves peuvent être étendues. Les nouveaux cas sont-ils des sous-cas des constantes ?* □

Exercice 8 (***) *On note ML_1 le langage où les paires sont ajoutées comme des constantes et ML_2 le langage où elle sont ajoutées de façon primitive. Proposer une traduction des programmes de ML_2 dans ML_1. Montrer que la traduction commute avec la réduction. Peut on faire un codage dans le sens inverse ? Comment montrer que les deux sont équivalents ?* □

Les tuples peuvent être ajoutés au langage de façon similaire, ou bien simplement codés à partir des paires. En fait, il est fréquent (c'est plus simple mais plus long) de formaliser les paires directement dans le langage ; on peut alors se limiter à des primitives ou des constructeurs d'artité 0 ou 1, les arguments multiples étant remplacés par des tuples.

Valeurs construites et filtrage

Une extension simple du noyau ML consiste à ajouter des valeurs construites et du filtrage superficiel (le filtrage en profondeur étant plus compliqué).

Pour simplifier la formalisation, nous ignorons les déclarations de types et nous paramétrons le langage par une collection de types concrets prédéfinis. Dans une premier temps nous ne considérons qu'un seul type concret. En fait, nous laisons en exercice le cas de plusieurs types concrets. Le cas de déclarations de types locales est plus compliqué. Par exemple, la définition suivante

$$\mathtt{type}\ \tau\ \mathtt{g} = \mathtt{C}\ \mathtt{of}\ \tau_i \mid \ldots \mathtt{C}\ \mathtt{of}\ \tau_n$$

revient à ajouter n constructeurs $\mathtt{C}_1, \ldots \mathtt{C}_n$ et une primitive $\mathtt{F_g}$ d'arité $n+1$:

$C^1 ::= \mathtt{C}_1 \mid \ldots \mathtt{C}_n$	Constructeurs
$f^{n+1} ::= \mathtt{F_g}$	Primitive de filtrage

La primitive $\mathtt{F_g}$ se réduit ainsi :

$$\mathtt{F_g}\ (\mathtt{C}_k\ \ v)\ v_1\ \ \ldots v_k \longrightarrow v_k\ v \qquad (\delta_{\mathtt{g}})$$

L'environnement de typage contient maintenant les déclarations

$$\begin{aligned}\mathtt{C}_i &: \forall\, \mathcal{V}.\, \tau_i \to \tau\ \mathtt{g} \\ \mathtt{F}_\mathtt{g} &: \forall\, \mathcal{V}.\, \tau\ \mathtt{g} \to (\tau_1 \to \tau') \to \ldots (\tau_n \to \tau') \to \tau'\end{aligned}$$

Attention ! il est important de se restreindre au cas où l'union des variables libres dans les τ_i apparaissent sont aussi libres dans τ.

On pourra ajouter le *sucre syntaxique* qui consiste à noter

$$\mathtt{match}\ a\ \mathtt{with}\ \mathtt{C}_1\ (x) \Rightarrow a_1 \ldots \mid \mathtt{C}_n\ (x) \Rightarrow a_n$$

pour

$$\mathtt{F}_\mathtt{g}\ a\ (\mathtt{fun}\ (x)\ a_1) \ldots (\mathtt{fun}\ (x)\ a_n)$$

Il s'agit juste d'une convention de notation, qui ne change rien à la formalisation ; il s'agit simplement de choisir une notation plus commode. En ce sens, le sucre syntaxique est “gratuit”.

Exercice 9 (**) *Que se passe-t-il si une variable libre de τ_i n'apparaît pas dans τ (donner un exemple) ? Commenter. Est-il possible qu'inversement une variable libre dans τ n'apparaisse dans aucun τ_i (donner un exemple) ?* □

Exercice 10 *Montrer que les hypothèses H sont satisfaites.* □

Exercice 11 (*) *Vérifier que le cas des booléens est un cas particulier des types construits* □

Exercice 12 (***) *Vérifier que le cas des paires est aussi un cas particulier d'une généralisation des types construits.* □

Exercice 13 (*) *Formaliser directement le cas des types construits. On utilisera la syntaxe habituelle pour le pattern matching :*

$$\mathtt{match}\ a\ \mathtt{with}\ \mathtt{C}_1\ (x) \Rightarrow a_1 \ldots \mid \mathtt{C}_n\ (x) \Rightarrow a_n$$

Montrer que l'on retrouve la règle de typage habituelle. en utilisant l'ajout de constantes. □

Combinateur de point-fixe

On peut ajouter une constante

$$Y : \forall\, \alpha_1, \alpha_2.\, ((\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_2$$

d'arité 2, avec la δ-règle

$$Y\ F\ v \longrightarrow F\ (Y\ F)\ v$$

Comme Y est d'arité 2, $Y\ F$ est une valeur, et l'évaluation du membre droit ci-dessus doit continuer avec la réduction de l'application externe. Il est important que Y soit d'arité deux, pour que Y calcule le point fixe "paresseusement". Par exemple, si Y était d'arité 1 avec la δ-règle

$$Y\ F \longrightarrow (\texttt{fun}\ (x)\ F\ (Y\ F)\ x)$$

alors l'évaluation de $Y\ F\ v$ partirait dans une boucle infinie (nous avons souligné le radical actif) :

$$\underline{Y\ F}\ v \longrightarrow Y\ (\underline{Y\ F})\ v \longrightarrow Y\ (F\ (\underline{F\ v}))v \longrightarrow \ldots$$

Exercice 14 (*) *Écrire la fonction factorielle à l'aide de Y.* □

Exercice 15 *Vérifier que les hypothèses* **H1** *et* **H2** *sont satisfaites pour l'ajout du combinateur Y.* □

La constante Y se comporte comme l'expression

$$\texttt{fun}\ (F)\ (\texttt{fun}\ (f\ x)\ F\ (f\ f)\ x)\ (\texttt{fun}\ (f\ x)\ F\ (f\ f)\ x)$$

qui n'est cependant pas typable en ML "pur" (il faudrait autoriser les types récursifs).

La construction `let rec` $fx = a_1$ `in` a_2 est donc simplement du sucre syntaxique pour `let` $f = Y$ (`fun` $(f\ x)\ a_1$) `in` a_2.

Exercice 16 *Les types concrets peuvent être récursifs, c'est-à-dire, dans une définition :*

$$\texttt{type}\ \tau\ \texttt{g} = \texttt{C of}\ \tau_i \mid \ldots \texttt{C of}\ \tau_n$$

le constructeur de type `g` *peut apparaître dans les types τ_i. En fait, ce constructeur ne peut apparaître qu'avec le même type τ. C'est donc τ* `g` *qui peut apparaître dans τ_i plutôt que* `g`*. Sinon, l'égalité de deux déclarations de types concrets deviendrait indécidables. Donner un exemple de type concret récursifs. Pour écrire le pointy fixe, il suffit d'un type récursif de la forme* `valeur` *égal à* `valeur` $\to$ `valeur`*. Donner une définition du type concret* `valeur`*. Donner une fonction de coercion* `deplier` *d'une* `valeur` *en une fonction de type* `valeur` $\to$ `valeur` *et son inverse replier. Les utiliser pour définir Y sans utiliser la récursion (on cherchera donc à écrire une version typée de l'expression donnée ci-dessus).* □

Exercice 17 (programmation) *Étendre le programme de l'exercice 5 avec les paires, les expressions booléennes et le combinateur de point fixe.* □

Contraintes de type

On peut également étendre le langage avec des contraintes de types. En effet, on peut voir une expression $(a : \tau)$ comme du sucre syntaxique l'application $(\_ : \tau)\ a$ où $(\_ : \tau)$ est une primitive de type principal $\forall \mathcal{V}.\, \tau \to \tau$, et avec la δ-règle $(v : \tau) \longrightarrow v$.

Exercice 18 *Vérifier que les hypothèses sont satisfaites.* □

Les contraintes de types que nous proposons ici sont locales (les variables sont quantifiés localement). En particulier, deux contraintes dans la même phrase utilisant les même variables de types sont indépendante. On peut aussi choisir une stratégie opposée qui consiste à ne jamais généraliser les variables de type dans les contraintes, et ainsi les partager entre les différentes contraintes.

Conclusion

La sémantique à petits pas est supérieure par presque tous ses aspects à la sémantique à grands pas. Elle est plus simple parce qu'elle s'exprime dans le langage même. Les valeurs sont un sous-ensemble des expressions et non pas des objets extérieurs au langage. La sémantique à petits pas est plus concise. Mais son avantage essentiel est d'être plus précise puisqu'elle permet de décrire simplement des calculs en cours qui ne termineront peut-être pas, et donc de distinguer clairement la divergence des erreurs de calculs sans avoir recours à un artifice.

On lui reproche parfois d'être trop éloignée de la sémantique réelle, implantée dans un compilateur, rendant plus difficile la vérification qu'un compilateur est correct vis à vis de la sémantique. Elle se rapproche en effet beaucoup plus d'un interpréteur naïf qui procéderait par réécriture du code source. En particulier, elle effectue des substitutions "coûteuses" pendant le calcul, alors que la sémantique à grands pas pouvait grâce aux fermetures, modéliser une machine à environnement. Mais n'est-ce pas là le bon niveau d'abstraction entre une sémantique trop près de la machine et la sémantique dénotationnelle trop éloignée de l'évaluation ?

Nous avons vu que la sémantique à réduction se prête très bien à des extensions du langage car elle est modulaire. Nous verrons dans le chapitre suivant qu'elle s'applique également, et de façon simple, aux constructions impératives.

Pour en savoir plus

Sur d'autres approches de la sémantique des langages de programmation, lire [Gun93]. La comparaison entre sémantique à petits pas et la sémantique à grand pas est détaillée dans [GR93]. Pour les propriétés générales du typage et le typage avec types récursifs, on pourra lire [Rém92a]. Le traitement des types concrets est abordé dans [Rém92b], [Rém94a] et [Läu92].

Chapitre 4

Ajout de constructions impératives.

Les variantes du noyau du langage ML étudiées précédemment ne possèdent que des constructions fonctionnelles. Dans ce chapitre nous considérons des extensions de ML avec des constructions impératives. Nous étudions en détail une extension indispensable : l'ajout d'objets mutables. Nous présentons plus brièvement les autres constructions impératives : exceptions, continuations, communications.

4.1 Constructions impératives

Dans les langages purement fonctionnels, si une expression close a_1 se réduit en une expression a_2 (en respectant la stratégie du langage, par exemple appel par valeur), alors les programmes $E[a_1]$ et $E[a_2]$ sont équivalents, c'est-à-dire qu'ils terminent sur les mêmes entrées et retournent les mêmes valeurs. Formellement cela découle de la règle de réduction Contexte.

Cette propriété implique qu'il n'y a pas de notion de partage, puisqu'il est toujours possible de remplacer une valeur liée à une variable x par un programme qui reconstruit cette valeur. Elle implique également que plusieurs évaluations d'une même expression avec les mêmes paramètres produisent toujours le même résultat. On peut alors "comprendre" une fonction du langage comme une fonction mathématique dans un certain ensemble d'objets représentants les valeurs du langage. C'est ce que fait la sémantique dénotationnelle. La sémantique opérationnelle (naturelle ou à réduction) d'un langage purement fonctionnel est elle-même très simple, parce que la sémantique d'un programme peut s'exprimer uniquement en fonction de ce programme et des valeurs de ses arguments au cours de son évaluation.

Pourtant, il est de nombreuses situations où l'on souhaite que plusieurs évaluations successives d'une même expression produisent des résultats différents : C'est le cas, par exemple, si l'on veut réaliser un compteur. De façon abstraite, un compteur est un objet que nous pouvons interroger pour connaître sa valeur ; nous pouvons aussi l'incrémenter. Son interface pourra être de la forme :

```
lire : unit → int
incrémenter : unit → unit
```

Clairement, plusieurs exécutions de la fonction `lire` séparées par des exécutions de la fonction `incrémenter` doivent retourner des résultats différents. *Il y a forcément des paramètres cachés*; ce sont des cellules mémoire *partagées* entre les fonctions `lire` et `incrémenter` qui peuvent consulter et affecter la mémoire. Nous parlerons d'objets mutables ou de références plutôt que de cellules mémoire.

Les objets mutables permettent également de simuler la liaison dynamique en la remplaçant par une liaison statique sur un objet mutable. Ils correspondent bien entendu à la notion de pointeurs des langages Pascal, C, etc. Ils permettent également de coder des structures de graphe par des graphes en mémoire, ce qui est souvent indispensable pour écrire certains algorithmes de manière efficace. Il est clair également que les communications entre un programme et le monde extérieur (la mémoire) s'effectuent de façon impérative; un fichier peut être vu comme un objet dans lequel il est possible, en général, de lire et d'écrire. Par exemple, plusieurs lectures successives retourneront les caractères consécutifs du fichier.

Enfin, il existe d'autres formes de constructions impératives que les objets mutables. Par exemple, les exceptions, et plus généralement, les opérations de contrôle. Les canaux de communications dans les langages parallèles sont également des opérations essentiellement impératives. Les objets mutables sont donc indispensables, et il n'est guère de programmes réels qui puissent aisément s'en priver.

4.2 Une sémantique à réduction pour les objets mutables

Dans le chapitre 1, nous avons décrit la sémantique opérationnelle du noyau ML par réduction des expressions du langage sur elles-mêmes jusqu'à aboutir à une forme normale, qui s'est révélée être toujours une valeur pour les programmes bien typés.

Nous limitons maintenant notre étude à l'ajout de références au langage ML, la forme la plus simple d'objets mutables. Plutôt que d'étendre la syntaxe du langage, nous essayons d'ajouter deux primitives (`ref` _) et (!_) d'arité 1 et une primitive (_ := _) d'arité 2.

Comment expliquer le comportement de `ref` v, ou plus difficile encore, celui de $v_1 := v_2$ dont le seul effet est global, plus précisément celui de changer le comportement ultérieur de toutes les lectures de v_1 ? Au début de ce chapitre, nous avons justifié l'introduction des constructions impératives dans le langage par la nécessité d'invalider parfois la règle de réduction Contexte de la partie 3.1, page 17 que nous rappelons ci-dessous :

$$\text{si } a_1 \longrightarrow a_2, \text{ alors } E[a_1] \longrightarrow E[a_2]$$

Nous ne pourrons donc pas donner une sémantique du langage simplement en ajoutant de nouvelles règles pour réduire les opérations manipulant des références.

Utilisons l'idée intuitive de cellule mémoire. Nous nous donnons un ensemble infini dénombrable de locations mémoire que nous désignerons par la lettre ℓ. Nous définissons un état mémoire comme une fonction de domaine fini des locations mémoire vers les valeurs. Nous les désignons par la lettre s. De façon informelle, une référence `ref` v s'évalue en une nouvelle location mémoire non encore utilisée ℓ, augmentant ainsi l'état mémoire s d'une liaison $\ell \mapsto v$. Nous devons donc étendre le langage afin de pouvoir y manipuler les locations. Pour cela, nous étendons la syntaxe des expressions avec les locations mémoire. Nous considérons les locations mémoire comme des valeurs, i.e. nous étendons la définition de v par (les "..." signifient que nous étendons la définition précédente) :

$$v ::= \ldots \mid \ell$$

Nous pouvons maintenant décrire la sémantique du langage. La relation de réduction est définie sur des paires notées a/s formées d'une expression a et d'un état mémoire s. Elle est définie comme la plus petite relation satisfaisant les règles de la figure 4.1. Les règles Fun et Let ne consultent ni ne modifient l'état mémoire. Pour évaluer une

$$\begin{array}{lrcll}
 & (\mathtt{fun}\ (x)\ a)\ v/s & \longrightarrow & a[v/x]/s & \textsc{Fun} \\
 & \mathtt{let}\ x = v\ \mathtt{in}\ a/s & \longrightarrow & a[v/x]/s & \textsc{Let} \\
\text{If } \ell \notin dom\,(s), & \mathtt{ref}\ v/s & \longrightarrow & \ell/s \oplus (\ell : v) & \textsc{Ref} \\
\text{If } \ell \in dom\,(s), & !\ell/s & \longrightarrow & s(\ell)/s & \textsc{Deref} \\
\text{If } \ell \in dom\,(s), & \ell := v & \longrightarrow & v/s \oplus (\ell : v) & \textsc{Assign} \\
\text{If } a_1/s_1 \longrightarrow_\epsilon a_2/s_2, & E[a_1]/s_1 & \longrightarrow & E[a_2]/s_2 & \textsc{Contexte}
\end{array}$$

Figure 4.1 : Règles de réduction avec objets mutables

expression `ref` a on évalue a en une valeur v puis on alloue une location mémoire ℓ avec la valeur initiale v et on retourne cette location en résultat. L'opération $!a$ retourne la valeur v pourvu que a s'évalue en une location de contenu v. L'opération $a_1 := a_2$ modifie, si a_1 s'évalue en une location, le contenu de cette location avec le résultat v_2 de l'évaluation de a_2 et retourne v_2. En fait, la valeur retournée par l'affectation importe peu tant que ce choix est cohérent avec les règles de typage. Par exemple, il serait possible de retourner la valeur sans information `unit`, mais il faudrait introduire cette nouvelle constante dans le langage. Les contextes d'évaluation sont inchangés.

Ainsi l'évaluation d'un programme a se fait en réduisant le programme dans un état mémoire initial vide et conduit, si l'évaluation termine, à une valeur v dans un état mémoire final s. Cet état mémoire représente toute la mémoire utilisée au cours du calcul. En général, la valeur v contient des locations liées dans l'état mémoire et ne peut pas en être séparée. En revanche, la mémoire possède sans doute des locations qui ne sont plus utilisées. La récupération de la mémoire ne peut pas être formalisée simplement en ajoutant une règle d'évaluation locale qui jetterait les locations inaccessibles. Il faudrait englober la réduction dans une relation définie sur les programmes uniquement (et non pas définie récursivement sur les sous-expressions) puis ajouter une règle globale de glanage de cellules. Il est aussi possible de rendre la notion de mémoire complètement locale, mais au prix d'une complication du formalisme [WF91].

4.3 Typage des références

Nous allons voir qu'une extension naïve n'est pas correcte. Nous étendons d'abord les types avec un symbole `Ref` :

$$\tau ::= \ \ldots \mid \tau \ \texttt{Ref}$$

Le traitement des constructions impératives comme des constantes conduit aux règles de typage dérivées suivantes :

$$\text{(REF)}\ \frac{A \vdash a : \tau}{A \vdash \texttt{ref}\ a : \tau\ \texttt{Ref}} \qquad \text{(DEREF)}\ \frac{A \vdash a : \tau\ \texttt{Ref}}{A \vdash {!a} : \tau} \qquad \text{(ASSIGN)}\ \frac{A \vdash a_1 : \tau\ \texttt{Ref} \qquad A \vdash a_2 : \tau}{A \vdash a_1 := a_2 : \tau}$$

Cependant, la combinaison de ces règles avec la règle LET déjà présente est incorrecte. En effet, considérons le programme suivant :

```
let I = ref (fun x → x);;
I := succ;;
!I (fun (x) x):;;
```

Il est facile de vérifier que ce programme est bien typé pour les règles ci-dessus. Son évaluation est cependant bloquée. En effet, l'évaluation de la dernière phrase se réduit à l'application du contenu de `I`, qui est alors la fonction qui attend un entier, à la fonction `fun` $(x)\ x$ qui n'est donc pas un entier. L'identificateur `I` est lié à une référence de type $\forall\,\alpha.\,\alpha \to \alpha$. Dans la seconde phrase, il est vu, grâce au lemme de substitution, comme liée à une référence de type `Int` $\to$ `Int`. Cette phrase est donc bien typé. Son exécution remplace le contenu de `I` par la valeur `succ`. Cependant, I dont le type n'a pas changé peut être vu dans la dernière phrase comme une référence vers une valeur de type $(\alpha_1 \to \alpha_1) \to \alpha_0$, ce qui est erroné. L'incohérence est d'avoir remplacé une référence par une valeur dont le type est moins polymorphe que celui de la valeur initiale.

Le problème est encore plus apparent si on l'analyse formellement. Ayant introduit des locations mémoire puis une mémoire, nous devons d'abord leur donner une contrepartie dans le typage. Dans un premier temps, il est nécessaire d'étendre les environnements de typage aux locations mémoire :

$$A ::= \ \ldots \mid A \oplus (\ell : \forall\,\bar{\alpha}.\,\tau)$$

Le problème décrit plus haut apparaît dès qu'une location a un type polymorphe, puisqu'il est alors possible de remplacer cette location par une valeur de type $\tau[\bar{\tau}_1/\bar{\alpha}]$ puis de la lire comme une valeur de type $\tau[\bar{\tau}_2/\bar{\alpha}]$, ce qui sera erroné au moins chaque fois que $\bar{\tau}_1$ et $\bar{\tau}_2$ sont des instances incompatibles.

Il suffit donc interdire de créer des locations mémoire avec un type polymorphe, c'est-à-dire imposer :

$$A ::= \ \dots \mid A \oplus (\ell : \tau)$$

Il ne faut pas oublier d'ajouter une règle de typage pour les locations mémoire :

$$\frac{\ell : \tau \in A}{A \vdash \ell : \tau\ \mathtt{Ref}}\ (\textsc{Loc})$$

L'ajout de cette règle change a priori le système de typage et donc les résultats associés. Cependant, il est facile de voir que $A \vdash a : \tau$ est correct dans le système avec locations si et seulement si $A' \vdash a' : \tau$ est correct dans le système sans locations, où A' est obtenu en remplaçant $\ell : \tau$ dans A par $x_\ell : \tau\ \mathtt{Ref}$ et a' en remplaçant ℓ par x_ℓ dans a. Les variables x_ℓ sont bien entendu des variables qui n'apparaissent pas dans A ni dans a. Cette correspondance, permet de transporter les propriétés du typage de ML à ML avec locations (4.3).

Le critère de correction "Ne pas créer de location mémoire de type polymorphe" fait intervenir la sémantique dynamique puisque les locations n'apparaissent pas dans le programme initial. Il faut donc trouver un critère statique qui garantit le critère dynamique. Allons plus loin dans la formalisation et relions le type des locations au type des valeurs liées par ces locations.

Nous dirons qu'un état mémoire s est en accord avec un environnement de typage A, et nous noterons $\Vdash s : A$, si s et A sont définis sur exactement les mêmes locations mémoire, et si pour toute location mémoire de ce domaine nous avons $A \vdash s(\ell) : A(\ell)$.

Il s'agit donc de ne pas donner un type polymorphe aux valeurs qui pourraient initialiser la création de références, ce que l'on peut garantir de deux façons simples et opposées :

1. Ne sachant pas quand les références seront effectivement créées, on interdit de généraliser les variables de type qui apparaissent dans le type de la construction d'une référence.

2. Pour interdire de généraliser une expression dont l'évaluation pourrait créer des références polymorphes, une approche drastique est de ne généraliser que le type des expressions qui sont des valeurs (dont l'évaluation ne fait rien, donc ne crée rien).

La lecture d'une référence et sa modification ne posent alors plus de problème. Chacune des hypothèses ci-dessus peut être garantie par une modification simple du typage. Dans le cas (1), le plus simple est de partitionner les variables de types $\mathcal{V}$ en deux classes $\mathcal{V}_N$ et $\mathcal{V}_G$ de variables non généralisables et de variables généralisables. Il faut alors modifier la règle de typage des références :

$$\frac{\overset{(\textsc{Ref})}{A \vdash a : \tau \qquad FV(\tau) \subset \mathcal{V}_N}}{A \vdash \mathtt{ref}\ a : \tau\ \mathtt{Ref}}$$

et conjointement restreindre la généralisation de τ dans A :

$$\mathrm{Gen}(\tau, A) \stackrel{def}{=\!=} (FV(\tau) \setminus FV(A)) \setminus \mathcal{V}_N$$

Dans le cas (2), les règles de typages sont inchangées, mais il faut restreindre la construction de liaison aux valeurs :

$$a ::= \ \ldots \mid \texttt{let}\ x = v\ \texttt{in}\ a$$

La première approche est celle qui a été longtemps utilisée pour ML, mais la seconde, de loin la plus simple est aussi celle qui tend à être communément admise comme le meilleur compromis. Pour cette raison, c'est celle-ci que nous retiendrons et pour laquelle nous montrerons la correction du typage. La première solution se formalise de façon similaire et ne pose pas de difficulté particulière.

Exercice 19 (*) *Trouver un exemple intéressant de programme (i.e. un programme que l'on souhaite écrire) qui n'est pas typable par la première approche. Ce programme est-il typable avec la seconde approche ? N'existe-t-il pas un programme équivalent (qui réalise le même algorithme) qui est typable avec la seconde approche ?* □

Exercice 20 (*) *Dans la seconde approche, il n'est plus possible d'écrire de nombreux programmes, par exemple* $\texttt{let}\ x = 1 + 2\ \texttt{in}\ x * x$. *Comment peut-on réintroduire la construction* $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$ *dans le cas où* a_1 *n'est pas une valeur ? Pourquoi avons nous exclu ce cas ci-dessus ?* □

Exercice 21 (**) *On suppose que la liaison est autorisée sur des expressions qui ne sont pas des valeurs comme dans l'exercice précédent. Il arrive parfois de spécialiser une fonction polymorphe* g *à plusieurs arguments sur le premier argument. On écrit alors un programme de la forme* $\texttt{let}\ g' = g\ v\ \texttt{in}\ a$. *Malheureusement* g' *ne sera pas polymorphe dans* a. *Peut-on écrire un programme équivalent qui permettent à* g' *d'être polymorphe.* □

4.4 Correction de la sémantique

La correction du typage par rapport à la sémantique se prouve en suivant exactement la même démarche que pour le noyau ML. La seule difficulté consiste à bien énoncer la correction du typage. Ensuite, il sera facile d'adapter les preuves.

Puisque le typage des constructions impératives n'est maintenant plus restreint, nous pouvons considérer celles-ci comme des applications de constantes fonctionnelles. Nous rappelons les différentes classes syntaxiques :

$$\begin{array}{rl}
a ::= & x \mid c \mid a_1\ a_2 \mid \texttt{fun}\ (x)\ a \mid \texttt{let}\ x = v\ \texttt{in}\ a_2 \\
v ::= & c^k\ v_1\ \ldots v_q \mid C^k\ v_1\ \ldots v_k \mid \texttt{fun}\ (x)\ a \qquad (q < k) \\
c ::= & (\texttt{ref}\ _) \mid (!_) \mid (_ := _) \\
E ::= & [] \mid E\ a \mid v\ E
\end{array}$$

En particulier, les contextes d'évaluation ne comprennent plus la construction de liaison, puisque celle-ci est maintenant restreinte à des liaisons à des valeurs. Nous nous limitons aux constantes ci-dessus dans le but de simplifier le formalisme, mais nous pourrions bien entendu abstraire le formalisme par rapport à un ensemble de constantes, leur typage et leurs δ-règles associées, comme dans le chapitre 1. Clairement, les constantes ($\texttt{ref}\ \_$) et ($!\_$) sont des primitives d'arité 1 et ($\_ := \_$) est une primitive d'arité 2.

L'environnement de typage initial A_0 est le suivant :

$$\begin{aligned}(\texttt{ref}\ \_) &: \forall \alpha.\, \alpha \to \texttt{Ref}(\alpha)\\ (!\_) &: \forall \alpha.\ \texttt{Ref}(\alpha) \to \alpha\\ (\_ := \_) &: \forall \alpha.\ \texttt{Ref}(\alpha) \to \alpha \to \alpha\end{aligned}$$

Nous appelons extension mémoire d'un environnement de typage A une extension de A avec un nombre quelconque d'assertions de la forme $\ell : \tau$. Nous écrivons $a_1/s_1 \sqsubseteq a_2/s_2$ si pour tout environnement de typage A_1 et tout type τ tels que $A_1 \vdash a_1 : \tau$ et $\Vdash s_1 : A_1$, il existe une extension mémoire A_2 de A_1 telle que $A_2 \vdash a_2 : \tau$ et $\Vdash s_2 : A_2$.

La sécurité du typage s'exprime en reformulant les deux théorèmes 1 et 2, page 25 du chapitre 1 :

Théorème 3 (Auto-Réduction) *La réduction préserve les typages (i.e. $\longrightarrow$ est une sous relation de $\sqsubseteq$).*

Théorème 4 (Formes Normales) *Les expressions irréductibles et bien typées relativement à une extension mémoire de l'environnement initial sont des valeurs.*

C'est-à-dire, soit A une extension mémoire de l'environnement de typage initial A_0. Si $A \vdash a : \tau$ et $\vdash s : A$ et si a/s est irréductible, alors a est une valeur.

Le théorème d'auto-réduction est une combinaison immédiate des lemmes de contraction de radical et de remplacement dans un contexte :

Lemme 7 (Remplacement dans un contexte) *Pour tout contexte E, si $a_1/s_1 \sqsubseteq a_2/s_2$ alors $E[a_1]/s_1 \sqsubseteq E[a_2]/s_2$.*

Preuve : Par induction structurelle sur E. Soit A_1 un environnement de typage et τ un type tel que $A \vdash C[a_1] : \tau$ (1) et $\Vdash s_1 : A_1$. Nous montrons qu'il existe une extension mémoire A_2 de A_1 telle que $A_2 \vdash a_2 : \tau$ (2) and $\Vdash s_2 : A_2$.

Cas a **est** $[]\ a$ **:** Une dérivation de (1) se termine par :

$$\frac{(3)\ \ A_1 \vdash a_1 : \tau_1 \to \tau \qquad (4)\ \ A_1 \vdash a : \tau_1}{A_1 \vdash a_1\ a : \tau}\ (\text{App})$$

Par hypothèse appliquée à (3), il existe une extension mémoire A_2 de A_1 telle que $A_2 \vdash a_2 : \tau$ et $\Vdash A_2 : s_2$. Par le lemme d'extension, appliqué à (3), nous avons $A_2 \vdash a : \tau$ d'où la conclusion par une règle App.

Cas a **est** $v\,[\,]$ **:** Aussi simple que le précédent. ■

Bien que la preuve soit similaire à celle du chapitre précédent, il est important de noter qu'elle n'en découle pas. En particulier le cas où a est $\mathtt{let}\ x = E\ \mathtt{in}\ a'$ ne peut pas être montré. Heureusement, nous avons exclu ce contexte des contextes d'évaluation.

Lemme 8 (Contraction de radicaux) *Si* $a_1/s_1 \longrightarrow_\epsilon a_2/s_2$ *alors* $a_1/s_1 \sqsubseteq a_2/s_2$*.*

Chaque cas de radical peut être traité indépendamment, et, après avoir prouvé le lemme suivant, les preuves sont immédiates.

Lemme 9 (Remplacement dans un terme) *Si les formules* $A \vdash v : \tau_0$ *(1) et* $A \oplus (x : \forall \bar\alpha_0.\, \tau_0) \vdash a : \tau$ *(2) sont prouvables et si les variables liées de* a *ne sont pas libres dans* v *et les variables* $\bar\alpha_0$ *ne sont pas libres dans* A*, alors* $A \vdash a[v/x] : \tau$ *(3) est prouvable.*

Preuve : Puisqu'il ne s'agit ici que d'un résultat de typage, donc ne faisant pas intervenir la sémantique, on peut cette fois-ci réutiliser la preuve du chapitre précédent. Nous reposons cependant sur la remarque donnée dans la section précédente que le système de typage avec locations est équivalent à un système de typage sans locations. Nous utilisons également le fait que la restriction syntaxique (l'ensemble des programmes ML qui n'autorise que la liason à des valeurs) est stable par substitution par une valeur.

On pourrait également faire une preuve directe, similaire à celle du chapitre précédent, dans laquelle il faudrait ajouter le cas où a est ℓ et qui se traite comme le cas a est y (ce n'est pas une surprise, puisque dans la traduction présentée ci-dessus 4.3, page 35, les locations sont remplacées par des variables). ■

Preuve : (de la contraction de radical) À nouveau, la démonstration copie celle du chapitre 1. Soit A_1 un environnement de typage et τ un type tels que $A_1 \vdash a_1 : \tau$ (1) et $\Vdash s_1 : A_1$. Nous montrons qu'il existe une extension mémoire A_2 de A_1 telle que $A_2 \vdash a_2 : \tau$ (2) et $\Vdash s_2 : A_2$ par cas sur le radical a_1. Chaque cas peut être vérifié indépendamment. Nous incorporons le cas des δ-règles.

Cas a_1 **est** $(\mathtt{fun}\ (x)\ a)\ v$ **:** Une dérivation de (1) se termine par :

$$\cfrac{\cfrac{(3)\ \ A_1 \oplus (x : \tau_1) \vdash a : \tau}{A_1 \vdash a : \tau_1 \to \tau}\ (\textsc{Fun}) \qquad (4)\ \ A_1 \vdash v : \tau_1}{A_1 \vdash a_1 : \tau}\ (\textsc{App})$$

Le lemme de remplacement appliqué à (4) et (3) montre la conclusion en choisissant A_1 pour A_2.

Cas a_1 **est** $\mathtt{let}\ x = v\ \mathtt{in}\ a$ **:** Une dérivation de (1) se termine par :

$$\frac{(3)\ \ A_1 \vdash v : \tau_1 \qquad (4)\ \ A_1[x : \mathrm{Gen}(\tau_1, A)] \vdash a : \tau}{A_1 \vdash \mathtt{let}\ x = v\ \mathtt{in}\ a : \tau}\ (\textsc{Let})$$

Nous concluons en appliquant le lemme précédent (remplacement dans un terme) à (3) et (4) en choisissant A_1 pour A_2.
Les cas suivants correspondent à la vérification de l'hypothèse **H1** sur les δ-règles.

Cas a est `ref` v : Une dérivation de (1) se termine par :

$$\frac{A_1 \vdash v : \tau_1 \ (2) \qquad A_1 \vdash \texttt{ref}\!: \tau_1 \to \tau_1 \ \texttt{Ref}}{A_1 \vdash \texttt{ref} \ v : \tau_1 \ \texttt{Ref}} \ (\text{App})$$

Nous choisissons $A_1 \oplus (\ell : \tau_1)$ pour A_2. D'une part, $A_2 \vdash \ell : \tau_1$. D'autre part, notons s' égal à $s \oplus \ell : v$. En utilisant (3), nous avons $A_1 \vdash \sigma(\ell') : A_2(\ell')$ pour chaque ℓ' du domaine de A_2. Par extension de contexte (lemme **??**), nous pouvons remplacer l'hypothèse A_1 par A_2, et donc conlure $A_2 \vdash \sigma \oplus \ell : v$.

Cas a est $\ell := v$: Nécessairement τ est de la forme τ_1 `Ref`, $\ell : \tau_1$ est dans A_1, et $A_1 \vdash v : \tau_1$. La conclusion est satisfaite, en choisissant A_2 égal à A_1.

Cas a est $!(\ell)$: Nécessairement $\ell : \tau$ est dans A_1, donc $A_1 \vdash s(\ell) : \tau$. En prenant A_1 pour A_2, nous avons bien $A_2 \vdash s(\ell) : \tau$ et $\Vdash s_2 : A_2$. ■

Le second théorème énonce que les termes bien typés qui ne peuvent être réduits sont des valeurs, en conséquence l'évaluation des termes bien typés n'est jamais bloquée. La preuve est une récurrence structurelle sur la valeur en utilisant le lemme suivant :

Lemme 10 *Soit A une extension mémoire de A_0 telle que $A \vdash v : \tau$.*

- *Si τ est un type fonctionnel alors a est une fonction ou une constante fonctionnelle partiellement appliquée.*
- *Si τ est τ_1 `Ref` alors v est une location mémoire.*

Preuve : Nous montrons séparément que pour toute valeur v et une extension mémoire A de l'environnement initial A_0 telle que $A \vdash v : \tau$ (8),

- Si v est une fonction ou une constante fonctionnelle partiellement appliquée alors τ est un type fonctionnel.
- Si v est une location mémoire alors τ est τ_1 `Ref`.

Chaque cas se vérifie facilement par analyse de la dérivation (1). Les sous-ensembles des types sont disjoints. Les sous-ensembles de valeur correspondantes forment une partition de l'ensemble des valeurs. On peut donc inverser les implications. ■

Preuve : (des formes normales) La preuve est par récurrence structurelle sur a. Soit A une extension mémoire de l'environnement initial de typage telle que $A \vdash a : \tau$ (1) pour un certain type τ et $\Vdash s : A$ (2) pour un état mémoire s. Nous supposons que a/s ne peut pas être réduit.

Cas a est x : Puisque le domaine de A est réduit à des constantes et à des locations mémoire, ce cas n'est pas possible.

Cas a est $a_1 \ a_2$: Une dérivation de (1) montre qu'il existe un type τ_1 tel que $A \vdash a_1 : \tau_1 \to \tau$ et $A \vdash a_2 : \tau_1$. L'hypothèse de récurrence appliquée à a_1/s avec l'environnement A et le type $\tau_1 \to \tau$ montre que a_1 est une valeur v. Ce ne peut pas être une fonction, car sinon a pourrait être réduit. Puisqu'elle a un type fonctionnel, c'est donc une constante fonctionnelle partiellement appliquée. L'hypothèse de récurrence appliquée à (2) montre que a_2 est une valeur. Nous raisonnons maintenant par cas sur la constante v :

Sous-cas v est `ref` : Alors a pourrait être réduite.

Sous-cas v est ! : Alors a_1 est une valeur de type τ_1 `Ref` dans A, donc une location ℓ. De (2), nous déduisons que ℓ est dans le domaine de s donc a est réductible.

Sous-cas v est $(\_ := \_)$: Alors a est effectivement une valeur.

Sous-cas v est $(v_1 := \_)$: Alors v_1 est une location mémoire ℓ liée dans s. Donc a est réductible.

Cas a est `let` $x = v$ `in` a_2 : Alors a est réductible.

Autres cas : Il ne reste donc plus que les cas où a est une constante, une fonction ou une location mémoire, c'est-à-dire une valeur. ■

Exercice 22 *Faire la preuve de correction pour l'autre solution proposée, interdisant de généraliser toute variable ayant apparu sous un constructuer* `Ref`*.* □

Exercice 23 *Peut-on proposer une solution combinée consistant à type* `let` $x = v$ `in` *aina de façon polymorphe, et* `let` $x = a_0$ `in` a *en interdisant de généraliser les variables qui apparaissent dans des expressions sous un type* `Ref` *?* □

4.5 Les autres formalismes

Nous avons proposé deux solutions pour typer correctement les références qui consistent à restreindre soit l'ensemble des variables généralisables, soit l'ensemble des termes généralisables. Aucune n'est entièrement satisfaisante.

La première solution proposée est une extension conservative de ML, car si un programme n'utilise pas de référence, il est toujours possible de le typer en n'utilisant que des variables généralisables. Mais cette solution contraint beaucoup la construction des références, et par exemple, elle ne permet pas de typer la fonction `fun` (x) `ref` x qui illustre en fait un grand nombre de cas pratiques où il est souhaitable d'écrire une fonction polymorphe qui crée des références, mais de n'appliquer cette fonction ensuite que dans des cas monomorphes.

La seconde solution qui permet de traiter l'exemple ci-dessus, et la plupart des exemples utiles en pratique, n'est cependant pas une extension conservative de ML, c'est-à-dire qu'il existe des programmes n'utilisant pas les références qui étaient auparavant typables et qui ne le sont plus. Par exemple,

```
let I = (fun x → x) (fun x → x) in I I;;
```

En effet, la définition de `I` fait intervenir une application et n'est donc pas généralisable donc le corps du programme `I I` n'est pas typable.

En pratique, la seconde solution est de loin la meilleure, même si elle restreint le polymorphisme de ML. Y a-t-il un meilleur compromis ?

Nous présentons ci-dessous une solution trop simple qui rejette trop de programmes, et deux solutions trop sophistiquées, qui permettent de typer à peu près tous les programmes souhaitables, mais qui utilisent une structure de type trop complexe pour permettre à l'utilisateur de les manipuler, et par exemple d'écrire les interfaces des modules.

Exercice 24 *Montrer que l'on peut coder le point fixe Y avec des références, et sans utiliser l'opérateur de récursion, ni les types récursifs.* □

Les variables mutables

Une restriction très forte sur les constructions mutables consiste à ne pas les autoriser dans des fermetures. Les cellules mémoire ont alors une durée de vie connue statiquement, ce qui permet de les généraliser sans restriction. La contrepartie étant bien entendu que ces cellules mémoire sont moins expressives. La restriction peut être assurée en déréférençant automatiquement les variables à chaque utilisation et en interdisant aux variables de traverser les liaisons λ. Cette solution est trop restrictive pour être suffisante, mais elle pourrait être combinée avec une des solutions précédentes.

La syntaxe de ML serait alors étendue avec les constructions :

$$a ::= \ \ldots \mid \texttt{let var}\ \ell = a_1\ \texttt{in}\ a_2 \mid \ell$$

et les règles de typage suivantes :

$$\text{(Var)}\ \frac{A \vdash a_1 : \tau_1 \qquad A \oplus (\ell : \tau_1) \vdash a_2 : \tau_2}{A \vdash \texttt{let var}\ \ell = a_1\ \texttt{in}\ a_2 : \tau_2} \qquad\qquad \text{(Loc)}\ \frac{\ell : \tau \in A}{A \vdash \ell : \tau}$$

Il convient de modifier en plus la règle d'abstraction pour interdire à une location mémoire d'être exportée dans une fermeture :

$$\text{(Fun)}\ \frac{\bar{A} \oplus (x : \tau_0) \vdash a : \tau}{A \vdash \texttt{fun}\ (x)\ a : \tau_0 \to \tau}$$

où $\bar{A}$ est l'environnement A privé de ses locations mémoire.

Les contextes d'évaluation sont simplement :

$$E ::= \ \ldots \mid \texttt{let var}\ \ell = E\ \texttt{in}\ a$$

Il faut bien remarqué qu'ici, les locations ℓ ne sont plus considérées comme des valeurs. Par exemple, elles ne peuvent pas être mémorisées dans des structures de données. Les règles de réduction sont les suivantes :

$$
\begin{array}{lrcll}
 & (\texttt{fun}\ (x)\ a)\ v/s & \longrightarrow & a[v/x]/s & \textsc{Fun} \\
 & \texttt{let}\ x = v\ \texttt{in}\ a/s & \longrightarrow & a[v/x]/s & \textsc{Let} \\
\text{If } \ell' \notin dom\,(s), & \texttt{let var}\ \ell = v\ \texttt{in}\ a/s & \longrightarrow & a[\ell'/\ell]/s \oplus (\ell' : v) & \textsc{Var} \\
\text{If } \ell \in dom\,(s), & \ell/s & \longrightarrow & s(\ell)/s & \textsc{Deref} \\
\text{If } \ell \in dom\,(s), & \ell := v & \longrightarrow & v/s \oplus (\ell : v) & \textsc{Assign} \\
\text{If } a_1/s_1 \longrightarrow_\epsilon a_2/s_2, & E[a_1]/s_1 & \longrightarrow & E[a_2]/s_2 & \textsc{Contexte}
\end{array}
$$

Le sémantique ci-dessus ne reflète cependant pas la portée syntaxique des variables mutables. Pour donner une sémantique plus précise, nous introduisons les contextes d'évaluation

$$E ::= E\ a \mid v\ E \mid \texttt{let var}\ \ell = v\ \texttt{in}\ E_\ell$$

où E_ℓ est un contexte E qui ne lie pas ℓ sur son chemin. On peut alors donner la sémantique plus élégante suivante :

$$
\begin{array}{rcll}
(\texttt{fun}\ (x)\ a)\ v & \longrightarrow & a[v/x] & \textsc{Fun} \\
\texttt{let}\ x = v\ \texttt{in}\ a & \longrightarrow & a[v/x] & \textsc{Let} \\
\texttt{let var}\ \ell = v\ \texttt{in}\ v' & \longrightarrow & v' & \textsc{Var} \\
\texttt{let var}\ \ell = v\ \texttt{in}\ E_\ell[\ell] & \longrightarrow & \texttt{let var}\ \ell = v\ \texttt{in}\ E_\ell[v] & \textsc{Deref} \\
\texttt{let var}\ \ell = v\ \texttt{in}\ E_\ell[\ell := v'] & \longrightarrow & \texttt{let var}\ \ell = v'\ \texttt{in}\ E_\ell[v'] & \textsc{Assign} \\
\text{If } a_1/s_1 \longrightarrow_\epsilon a_2/s_2, \quad E[a_1]/s_1 & \longrightarrow & E[a_2]/s_2 & \textsc{Contexte}
\end{array}
$$

En particulier la règle VAR repose sur le fait qu'une valeur ne peut pas contenir de locations.

Exercice 25 *Montrer la correction du typage vis à vis de la sémantique pour cette extension du noyau ML.* □

Les variables faibles de SML

Dans la solution que nous avons retenue, il est garanti que l'évaluation des expressions généralisables ne crée pas de références puisque ce sont des valeurs. Nous dirons qu'une expression est *non expansive* si son évaluation ne crée pas de référence. Il existe bien sûr de nombreux autres programmes non-expansifs que les valeurs. Par exemple on peut définir la classe suivante :

$$
\begin{array}{l}
b ::= v \mid \texttt{let}\ x = v\ \texttt{in}\ b \mid b_0 \\
b_0 ::= (!\_) \mid (\_ := \_) \mid b_0\ b
\end{array}
$$

Il est important que `ref _` ne soit pas dans la classe b_0, car en effet son application crée une référence.

Si b est une classe de programmes non-expansifs, on peut alors admettre la construction `let` $x = b$ `in` a dans le langage et préserver la correction forte. Mais il est difficile d'identifier de façon purement syntaxique une sous-classe de programmes non-expansifs plus grande que b.

L'idée des variables faibles est d'utiliser un mécanisme associé au typage pour identifier une classe plus large de programmes non expansifs. Par exemple, la fonction `ref` est typée comme $\forall \alpha^0 . \alpha^0 \rightarrow \alpha^0$ où l'exposant signifie que son application à un argument risque de créer une référence de type α. La variable de type α^0 est dite *faible*. On pourra se reporter à [HMT91] pour une présentation détaillée, mais cette solution autrefois utilisée dans le langage Sml-nj a été abandonnée en raison de sa complexité.

4.5.1 Le typage des fermetures

La première solution que nous avons proposée interdit de généraliser les variables qui apparaissent dans le type d'une valeur qui est mise dans une référence. Cette interdiction est globale et *ad-eternum*, c'est-à-dire qu'une variable ayant servi pour communiquer avec une référence ne peut plus être généralisée, même lorsque les locations correspondantes ne peuvent plus être atteintes.

Le typage des fermetures tente d'éviter cette contamination, en calculant au moment de la généralisation quelles sont les variables de types qui peuvent apparaître dans le type d'une référence en n'examinant que le type à généraliser.

En effet, le type d'une valeur construite décrit entièrement la structure de cette valeur. Cette propriété n'est plus vraie en général pour les fonctionnelles car les fonctions peuvent cachée des valeurs dans leur fermetures, sans en révéler le type. Par exemple, dans le programme suivant, la fonction `deux` est une valeur qui contient dans sa fermeture la valeur `un`.

```
let id x = x in let deux x = id (id x) in deux 0;;
```

L'approche consiste à typer les fonctions en montrant tous les types des valeurs qui sont libres dans leur corps, y compris le type des valeurs dans la fermeture. Ce mécanisme de typage est relativement complexe à mettre en œuvre. Se reporter à [Ler92].

4.5.2 Les régions

Cette solution consiste à déterminer, par un mécanisme lié à la synthèse des types, des régions du programme qui ne peuvent pas communiquer entre elles. L'utilisation d'une référence dans une région sera alors sans influence sur le typage d'une région disjointe. Ce mécanisme de typage est également complexe à mettre en œuvre. Se reporter à [TJ92].

Le typage des régions fournit aussi une information sur la connectivité du programme qui peut se révéler très utile pour la compilation. Par exemple cette méthode à été utiliser pour réaliser un compilateur de ML sans glanage de cellules, toutes les valeurs étant stockées dans la pile, dans des régions de taille variable, mais désallouées en bloc [BT96].

4.6 D'autres constructions impératives

Les exceptions

Par leur importance les exceptions sont sans aucun doute la seconde construction impérative dans les langages fonctionnels. Elle sont indispensables pour programmer proprement les cas pathologiques et ratrapper correctement les erreurs.

Pour ajouter les exceptions dans le langage, il nous faut d'abord introduire deux nouvelles constructions :

$$a ::= \ \ldots \mid \texttt{try}\ a\ \texttt{with}\ x \Rightarrow a \mid \texttt{Raise}\ a$$

La sémantique des exceptions est plus facile à définir que celle des objets mutables, car elle peut se faire directement sur les programmes sans devoir ajouter une construction globale analogue à la mémoire. Les contextes d'évaluation sont :

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

Nous ajoutons simplement aux constructions de ML les règles de réduction suivantes :

$$\begin{array}{c}
\texttt{try}\ v\ \texttt{with}\ x \Rightarrow a \xrightarrow{\epsilon} v \\
\texttt{try}\ E'[\texttt{Raise}\ v]\ \texttt{with}\ x \Rightarrow a \xrightarrow{\epsilon} \texttt{let}\ x = v\ \texttt{in}\ a
\end{array}$$

Dans la seconde règle, le contexte E' est un contexte d'évaluation ne contenant pas de nœud ($\texttt{try}\ \_\ \texttt{with}\ \_ \Rightarrow \_$), c'est-à-dire défini par la grammaire suivante :

$$\begin{array}{rl}
E' ::= & [] \\
& \mid E'\ a \\
& \mid v\ E' \\
& \mid \texttt{Raise}\ E'
\end{array}$$

Informellement, La seconde règle exprime que si l'évaluation de a lève une exception avec la valeur v, alors l'évaluation continue au niveau de la garde la plus proche.

Le typage des exceptions pose un problème similaire à celui des références : si le type des exceptions pouvait être un schéma σ, alors une exception pourrait être levée avec une instance τ_1 de σ et capturée avec une autre instance τ_2, ce qui conduirait à une erreur de typage si les deux instances τ_1 et τ_2 sont incompatibles. Le type des exceptions sera donc toujours monomorphe.

Privilégions un type τ_0 sans variables libres. Les règles de typage pour les exceptions sont :

$$\text{(Raise)}\ \frac{A \vdash a : \tau_0}{A \vdash \texttt{Raise}\ a : \alpha} \qquad \text{(Try)}\ \frac{A \vdash a_1 : \tau \qquad A \oplus x : \tau_0 \vdash a_2 : \tau}{A \vdash \texttt{try}\ a_1\ \texttt{with}\ x \Rightarrow a_2 : \tau}$$

Exercice 26 *Montrer la correction de cette extension.* □

Exercice 27 (***) *Peut-on coder le point fixe avec les exceptions ?* □

4.6.1 Les opérations de contrôle

Pour mémoire, se reporter à [GRR95].

Conclusion

Le formalisme de la sémantique à réduction s'adapte assez simplement aux constructions impératives du langage. Bien plus difficile est de trouver la bonne façon de typer les constructions impératives. Ce problème a des rapports très étroits avec d'autres analyses fines des programmes, par exemple avec la compilation sûre d'un plus grand ensemble de valeurs récursives. Là il ne faut plus garantir qu'une fonction n'a pas encore été appliquée, mais, ce qui est très similaire, qu'elle n'a pas encore lu les composantes de son argument.

Le typage fin des constructions impératives est un problème difficile qui n'a pas encore trouvé de solution idéale ; mais la solution minimaliste de restreindre le polymorphisme aux valeurs ne semble pas trop pénalisante, pour un gaint considérable de simplifité. Finalement, c'est peut être la meilleure solution.

Pour en savoir plus

Dans [WF91], on pourra trouver un traitement local de la mémoire dans la sémantique à réduction à petits pas.

Chapitre 5

Les enregistrements polymorphes

Dans ce chapitre nous nous intéressons à l'extension du langage ML avec des enregistrements. Les enregistrements sont des produits à champs nommés. Après avoir expliquer brièvement comment ajouter des enregistrements déclarés à ML, puis justifié la nécessité d'ajouter des opérations plus puissantes sur les enregistrements, nous découvrons le principe du typage des enregistrements extensibles, que nous formaliserons ensuite complètement.

5.1 Des enregistrements déclarés aux enregistrements extensibles

Les enregistrements déclarés sont des produits à champs nommés. Comme pour les types concrets, nous pouvons les formaliser de deux façons différentes. Nous pouvons les considérer comme une extension du formalisme, ou bien comme une application du cas général pour un choix particulier des constantes du langage.

Exercice 28 *Formaliser l'ajout d'un seul enregistrement pré-défini et correspondant à la déclaration de type suivante (en Caml) :*

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

□

Dans la définition de type de l'exercice ci-dessus, nous disons que les identificateurs `fst` et `snd` sont les étiquettes de l'enregistrement. Outre le caractère obligatoire des déclarations de types un inconvénient majeur est qu'une étiquette ne peut pas être utilisée dans plusieurs enregistrements à la fois. Pour que la fonction f égale à `fun` $(x)\ x.\ell$ soit définie, il est nécessaire qu'un type enregistrement ait été préalablement défini avec l'étiquette ℓ. La fonction f est typable, mais ne peut s'appliquer qu'aux enregistrements de ce type. L'extraction de champ est donc une opération monomorphe.

Une première extension consiste à rendre primitives les constructions d'enregistrements de façon à se passer des déclarations de types. On se donne un ensemble dénombrable d'étiquettes $\mathcal{L}$ que nous désignerons par la lettre ℓ. Nous ajoutons au langage les expressions suivantes :

$$a ::= \ \ldots \mid \{\ell_1 = a_1; \ldots \ell_n = a_n\} \mid a.\ell$$

Mais, les enregistrements ne sont dits *polymorphes* ou *flexibles* que s'il est possible également de définir une fonction d'accès polymorphe `fun` (x) $x.\ell$. Par exemple, les enregistrements de Sml-nj [HMT91] bien que non déclarés ne sont pas polymorphes.

Dans la plupart des langages, les enregistrements ne peuvent être construits qu'en une seule étape. C'est-à-dire qu'il n'est pas possible d'étendre un enregistrement déjà construit avec une nouvelle étiquette. On dit que les enregistrements sont *extensibles* lorsqu'il est possible de définir une fonction `fun` $(r\ v)\ r\ @\ \{\ell = v\}$ permettant d'étendre un enregistrement arbitraire r sur le champ ℓ avec une valeur v. Il est alors possible de construire les enregistrements à partir de l'enregistrement vide par ajout successifs de champs.

Nous étudions maintenant un langage avec enregistrements polymorphes et extensibles. La solution la plus simple est toujours d'essayer de considérer les enregistrements extensibles comme un cas particulier du cas général. Nous essayons donc de ne pas ajouter d'expressions au langage mais simplement des familles de constantes :

- un constructeur $\{\ell_1 = \_, \ldots \ell_n = \_\}$ d'arité n pour tous les tuples d'étiquettes deux à deux disjointes ;
- une primitive $(\_.\ell)$ d'arité 1 pour chaque étiquette ℓ dans $\mathcal{L}$;
- une primitive $(\_\ @\ \{\ell = \_\})$ d'arité 2 pour chaque étiquette ℓ dans $\mathcal{L}$.

Nous ajoutons les règles de réduction suivantes :

$$\begin{aligned} \{\ell_i = v_i^{i \in I}\}.\ell_j &\xrightarrow{\epsilon} v_j && (j \in I) \\ \{\ell_i = v_i^{i \in I}\}\ @\ \{\ell_j = v_j\} &\xrightarrow{\epsilon} \{l_i = v_i^{i \in I \cup \{j\}}\} && (j \notin I) \\ \{\ell_i = v_i^{i \in 1..n}\}\ @\ \{\ell_j = w_j\} &\xrightarrow{\epsilon} \{\ell_j = w_j, l_i = v_i^{j \in I \setminus \{j\}}\} && (j \in I) \end{aligned}$$

L'ajout d'un champ à un enregistrement peut se faire même si ce champ était déjà présent : on dit que l'extension est *libre*. Si l'on retire la dernière règle, cela n'est plus possible, et on dit que l'extension des enregistrements est *stricte*.

Exercice 29 (***) *Plutôt que d'ajouter des valeurs enregistrements, on aurait pu se contenter d'un enregistrement vide* $\{\}$ *et considérer les expressions* $v_1\ @\ \{\ell = v_2\}$ *comme des valeurs, c'est-à-dire* $(\_\ @\ \{\ell = \_\})$ *comme un constructeur. Suivre cette approche, et définir les règles de réduction. Les deux approches sont elles équivalentes (on n'en fera pas la preuve formelle, mais on essayera d'indiquer précisément en quel sens, et éventuellement les étapes de la preuve) ?* □

Nous pourrions considérer les valeurs enregistrements égales par permutation de champs, mais cela nous ferait sortir du formalisme des constantes. Ou bien, ayant fixé un ordre sur les étiquettes, nous pourrions nous restreindre à des primitives $\{\ell_1 = \_, \ldots \ell_k : \_\}$.

5.2 Principe du typage des enregistrements extensibles

Par opposition avec le typage des objets mutables, la difficulté n'est pas sémantique, mais purement syntaxique : comment étendre le système de typage pour assurer le polymorphisme désiré ? La première difficulté provient de la fonction d'accès `fun` (r) $r.\ell$ que l'on veut polymorphe, c'est-à-dire que l'on veut pouvoir l'appliquer à des enregistrements ayant au moins le champ ℓ mais pouvant également définir un ensemble arbitraire d'autres champs.

Pour simplifier, considérons le cas où les enregistrements sont construits à partir de deux étiquettes ℓ_1 et ℓ_2. Nous pouvons alors décrire un enregistrement en décrivant chaque champ séparément. Par exemple, soit x et y les valeurs $\{\ell_1 = 1\}$ et $\{\ell_1 = \mathit{true}; \ell_2 = 2\}$. Nous pouvons dire que l'enregistrement x est défini sur le champ ℓ_1 avec une valeur entière et indéfini sur le champ ℓ_2, et que y est défini sur le champ ℓ_1 avec une valeur booléenne et défini sur le champ ℓ_2 avec une valeur entière. Nous pouvons décrire cette information explicitement par le type suivante :

$$x : \{\ell_1 : \mathtt{Pre}\ (\mathtt{Int}); \ell_2 : \emptyset\} \qquad\qquad y : \{\ell_1 : \mathtt{Pre}\ (\mathtt{Bool}); \ell_2 : \mathtt{Pre}\ (\mathtt{Int})\}$$

La fonction d'extraction f égale à `fun` (r) $r.\ell$ prend en argument un enregistrement dont le champ ℓ_1 est défini avec une valeur de type τ retournée en résultat et dont le champ ℓ_2 est soit défini, soit indéfini ; ainsi, nous pouvons dire que son type est une instance de $\{\ell_1 : \mathtt{Pre}\ \alpha_1; \ell_2 : \alpha_2\} \to \alpha_1$, puisque α_2 peut alors être remplacé soit par $\mathtt{Pre}\ \tau$, soit par $\emptyset$. S'il y avait trois étiquettes, les types des valeurs précédentes seraient :

$$\begin{array}{l} x : \{\ell_1 : \mathtt{Pre}\ (\mathtt{Int}); \ell_2 : \emptyset; \ell_3 : \emptyset\} \\ y : \{\ell_1 : \mathtt{Pre}\ (\mathtt{Bool}); \ell_2 : \mathtt{Pre}\ (\mathtt{Int}); \ell_3 : \emptyset\} \\ f : \{\ell_1 : \mathtt{Pre}\ (\alpha_1); \ell_2 : \alpha_2; \ell_3 : \alpha_3\} \to \alpha_1 \end{array}$$

Il est immédiat d'imaginer ce que deviendrait les types ci-dessus s'il y avait une quatrième étiquette : puisque les étiquettes ℓ_3 et ℓ_4 n'apparaissent pas dans le programme, elles se comportent de la même façon, leurs types doivent donc être isomorphes. Le problème est en fait que nous ne connaissons pas par avance l'ensemble des étiquettes qui seront utilisées dans le reste du programme. En utilisant la remarque précédente, il nous suffit de donner un modèle pour les champs qui ne sont pas explicitement cités, de façon à pouvoir en prendre des instances au besoin. Le type des valeurs ci-dessus deviendra :

$$\begin{array}{l} x : \{\ell_1 : \mathtt{Pre}\ (\mathtt{Int}); \ell_2 : \emptyset; \underline{\emptyset}\} \\ y : \{\ell_1 : \mathtt{Pre}\ (\mathtt{Bool}); \ell_2 : \mathtt{Pre}\ (\mathtt{Int}); \underline{\emptyset}\} \\ f : \{\ell_1 : \mathtt{Pre}\ (\alpha_1); \ell_2 : \alpha_2; \underline{\alpha}\} \to \alpha_1 \end{array}$$

Les expressions soulignées ne sont pas attachées à des étiquettes particulières, mais elles décrivent, pour l'enregistrement dans lequel elles apparaissent, les types possibles pour les champs qui ne figurent pas explicitement à leur gauche. Ce sont, en quelque sorte, des modèles pour les champs implicites et les types de ces champs sont des instances du modèle. En fait, il faut lire $\{\ell_1 : \tau_1; \ell_2 : \tau_2; \underline{\tau_3}\}$ comme $\{\ell_1 : \tau_1; (\ell_2 : \tau_2; \underline{\tau_3})\}$. À partir du type $\{\ell_1 : \texttt{Pre}\ (\texttt{Int}); (\ell_2 : \emptyset; \underline{\emptyset})\}$, nous devons pouvoir obtenir le type $\{\ell_1 : \texttt{Pre}\ (\texttt{Int}); (\ell_2 : \emptyset; (\ell_3 : \emptyset; \underline{\emptyset}))\}$. Pour cela, il suffit de considérer les enregistrements modulo la famille d'équations

$$\underline{\emptyset} = (\ell : \emptyset; \underline{\emptyset}) \qquad \forall \ell \in \mathcal{L}$$

Le type $\{\ell_1 : \texttt{Pre}\ (\alpha_1); (\ell_2 : \alpha_2; (\ell_3; \alpha_3; \underline{\alpha'}))\} \to \alpha_1$ doit aussi pouvoir être considéré comme un type de f. Celui-ci est simplement obtenu par substitution de la variable $\underline{\alpha}$ dans le type de f par $(\ell_3 : \alpha_3; \underline{\alpha'})$. La variable $\underline{\alpha}$ est aussi appelée une variable de rangée. Enfin, il faut ajouter la famille d'équations

$$(\ell_1 : \alpha_1; (\ell_2; \alpha_2; \underline{\alpha})) = (\ell_2 : \alpha_2; (\ell_1 : \alpha_1; \underline{\alpha})) \qquad \forall \ell_1, \ell_2 \in \mathcal{L}$$

pour exprimer que l'ordre dans lequel apparaissent les champs dans un type enregistrement n'a pas d'importance.

Jusqu'à présent, nous avons ignoré le fait que les expressions `Pre` et $\emptyset$ ne désignent pas des types, mais des modèles de types. Certaines expressions, par exemple, $\texttt{Pre} \to \emptyset(\tau)$ ou $\{\underline{\{\emptyset\}}\}$ n'ont pas de sens, pas plus que $\{l_1 : \tau_1; \ell_1 : \tau_2; \underline{\emptyset}\}$ ou encore $\{\ell : \underline{\alpha}; \underline{\alpha}\}$. Pour éliminer ces types indésirables, nous utilisons des sortes. Les sortes sont aux types ce que les types sont aux termes de ML : ils permettent, par un mécanisme similaire au typage, d'éliminer parmi tous les types certains types considérés comme mal formés. Cette élimination se fait en deux étapes.

Premièrement, nous considérons deux sortes `Type` et `Field`, et nous imposons la signature (Σ_1) aux symboles de type :

$$\begin{aligned}
(_ \to _) &:: \texttt{Type} \otimes \texttt{Type} \Rightarrow \texttt{Type} \\
\{_\} &:: \texttt{Field} \Rightarrow \texttt{Type} \\
\texttt{Pre}\ (_) &:: \texttt{Type} \Rightarrow \texttt{Field} \\
\emptyset &:: \texttt{Field} \\
(\ell : _; _) &:: \texttt{Field} \otimes \texttt{Field} \Rightarrow \texttt{Field}
\end{aligned}$$

Les types $\texttt{Pre} \to \emptyset(\tau)$ et $\{\emptyset\}$ ne peuvent plus être écrits, car ils ne respectent pas les contraintes de sortes. Les types de tous les exemples qui précèdent sont heureusement bien formés pour Σ_1. Mais, il est encore possible d'écrire $\{\ell_1 : \tau_1; \ell_1 : \tau_2; \emptyset\}$.

Exercice 30 *Parmi les quatre exemples décrit une quinzaine de lignes ci-dessus, quels sont ceux que l'on ne peut plus écrire ? Comment éliminer les autres ?* □

Puis, nous filtrons à nouveau les expressions de types par une seconde signature. Nous considérons les sortes `Type` et la collection de sortes `Row` (L) où L varie dans l'ensemble des parties finies de l'ensemble des étiquettes. Cette fois-ci, l'idée est de séparer une expression de type ordinaire, de la sorte `Type`, d'une expression de rangée, de la sorte `Row` (L), l'ensemble L décrivant les étiquettes ne devant pas être redéfinies.

La signature des symboles de types est (Σ_2) :

$$
\begin{array}{rl}
(\_ \to \_) & :: \texttt{Type} \otimes \texttt{Type} \Rightarrow \texttt{Type} \\
\{\_\} & :: \texttt{Row}\ (\emptyset) \Rightarrow \texttt{Type} \\
\texttt{Pre}\ (\_) & :: \texttt{Type} \Rightarrow \texttt{Type} \\
\emptyset & :: \kappa \\
(\ell : \_; \_) & :: \texttt{Type} \otimes \texttt{Row}\ (\{\ell\} \cup L) \Rightarrow \texttt{Row}\ (L) \qquad (\ell \notin L)
\end{array}
$$

La variable κ est une variable de sorte. Cela signifie que $\emptyset$ a toutes les sortes. Les symboles $\emptyset$ et $(\ell : \_; \_)$ ont donc plusieurs sortes. En fait, il est possible de les indicer par leur sorte de façon à se ramener à des symboles mono-sortés. Dans le formalisme, il est plus simple de considérer les termes mono-sortés, mais en pratique, nous n'écrirons pas les indices des symboles. Étant donné la sorte d'un terme, il existe au plus une façon de l'annoter (c'est-à-dire d'indicer ses symboles de types par des sortes) de telle façon que le terme soit bien formé et de cette sorte. Il existe un algorithme qui décide si un terme est bien formé, et le cas échéant annote ses sous-termes par les sortes.

Les expressions de types bien formées sont celles qui respectent à la fois la signature Σ_1 et la signature Σ_2. Il est essentiel de vérifier que les deux familles d'équations sont bien sortées, c'est-à-dire que les membres gauches et droits de chacune des équations admettent simultanément les même sortes.

Exercice 31 (***) *Écrire un algorithme de synthèse de sortes. Indications : on pourra ajouter des variables de sortes au langage des sortes, puis s'inspirer de l'algorithme d'unification donné ci-dessous. Dans un premier temps on pourra chercher un semi-algorithme qui termine sur toutes les expressions de type bien formées, puis obtenir un algorithme par une petite modification simple permettant de détecter les cas de non terminaison du semi-algorithme.* □

Exercice 32 (**) *Pour chaque expression de type τ possiblement mal formée pour Σ_2 on définit un type $\bar{\tau}$ obtenu en remplaçant successivement :*

1. *toute variable de rangée α par un préfixe de la forme $\{\ell_1 : \alpha_1, \ldots \ell_k : \alpha_k; \alpha_0\}$ où les ℓ_i sont toutes les étiquettes figurant sur un chemin quelconque dans τ menant à α et les α_i sont des variables fraiches et disctinctes deux à deux.*

2. *tout terme $\ell : \tau; \tau'$ par τ' pour chaque orrurence de $\ell : \tau; \tau'$ au bout d'un chemin contenant ℓ.*

Vérifier que $\bar{\tau}$ est égal à τ si τ est bien formé pour Σ_2. À partir de la synthèse de sortes, en déduire un algorithme qui accepte une expression de type τ possiblement mal formée pour Σ_2 et retourne $\bar{\tau}$ si $\bar{\tau}$ est bien formée, ou signale une erreur sinon.

Par exemple, l'expression de type associée à $\{\ell_1 : \tau_1; \ell_1 : \tau_2; \emptyset\}$ est $\{\ell_1 : \tau_1; \emptyset\}$. Celle associée à $\{\tau\} \to \{\ell : \tau_0; \tau\}$ est $\{\ell : \alpha_1; \tau\} \to \{\ell : \tau_0; \tau\}$. Par contre, il n'y a pas d'expression de type associée à $\tau \to \{\ell : \tau_0; \tau\}$, car τ ne peut pas être à la fois de la sorte `Type` *et* `Row` (L)*.* □

Les règles de typage des enregistrements peuvent être traitées comme l'ajout de constantes avec l'environnement initial suivant :

$$\begin{array}{rll}
\{(\ell_i = _)^{i\in 1..n}\} : & \alpha_1 \to \ldots \alpha_n \to \{(\ell_i = \mathtt{Pre}\ \alpha_i)^{i\in 1..n}; \emptyset\} & (\textsc{Record}) \\
(_.\ell) : & \{\ell : \mathtt{Pre}\ \alpha; \alpha'\}) \to \alpha & (\textsc{Dot}) \\
(_\ @\ \{\ell = _\} : & \{\ell : \alpha_1; \alpha\} \to \alpha_2 \to \{\ell : \mathtt{Pre}\ \alpha_2; \alpha\} & (\textsc{With})
\end{array}$$

Il est quand même nécessaire de refaire l'étude des propriétés du typage car en considérant les types modulo une théorie équationnelle, nous avons changé l'algèbre des types. La formalisation des types enregistrements par des algèbres d'enregistrements permet de ramener le typage des enregistrements au typage dans ML, modulo une théorie équationnelle simple. Il suffira alors de généraliser les preuves faites pour le noyau ML au cas où les types sont considérés modulo une théorie équationnelle.

5.3 Algèbres d'enregistrements

Pour avoir une présentation plus générale et plus complète des types enregistrements, on pourra se reporter à [Rém91, Rém93a]. Il y est montré que les types enregistrements sont les termes d'une algèbre avec sortes engendrées par des théories équationnelles particulières, que ces théories équationnelles sont syntaxiques, et que le problème de l'unification y est décidable.

L'algèbre des types enregistrements est *syntaxique* ; cela signifie que l'unification de deux termes peut toujours se ramener à un problème d'unification entre leurs sous-termes en n'examinant les arguments que jusqu'à une profondeur bornée. Concrètement, les théories équationnelles syntaxiques sont celles pour lesquels l'agorithme d'unification peut s'écrire comme une extension naturelle de l'agorithme dans le cas non équationnel. Nous donnons ci-dessous un algorithme d'unification des types enregistrements :

```
let rec unify =
  fun α τ →
      if α ∈ 𝒱(τ) then fail else [τ/α]

    | g(τ_i)_{i∈1..n}  g(τ'_i)_{i∈1..n}  →
      let φ_0 = unify τ_0 τ'_0 in
      let φ_1 = unify φ_0(τ_1) φ_0(τ'_1) in
      ...
      let φ_n = unify φ_{n−1}(τ_n) φ_{n−1}(τ'_n) in
      φ_n

    | (ℓ_1 : τ_1; τ'_1) (ℓ_2 : τ_2; τ'_2) →
      let α = new_var() in
      let φ_0 = unify τ'_1 (ℓ_2 : τ_2; α) in
      let φ_1 = unify φ_0(τ'_2) φ_0(ℓ_1 : τ_1; α) in
```

```
  φ₁

| ∅ (ℓ : τ₁; τ₂) →
  let φ₀ = unify ∅ τ₁ in
  let φ₁ = unify ∅ φ₀(τ₂) in
  φ₁

| (ℓ : τ₁; τ₂) ∅ →
  unify ∅ (ℓ : τ₁; τ₂)

| τ α →
  unify α τ

| _ → fail
```

5.4 Synthèse de type

Le problème de la synthèse de types avec enregistrements extensibles se ramène alors à celui plus général de la synthèse de types lorsque les types sont considérés modulo une théorie équationelle pour laquelle le problème d'unification est décidable et unitaire. Les propriétés de la synthèse de types pour ML sont conservées sous certaines conditions très simples : il suffit que la théorie équationnnelle soit régulière, c'est-à-dire que les termes égaux aient toujours le même ensemble de variables libres ; que l'unification y soit décidable, et unitaire (c'est-à-dire qu'il existe un unificateur principal pour les problèmes d'unifications solvables). L'algorithme de synthèse de types est alors obtenu en remplaçant l'algorithme d'unification de ML par celui de la théorie équationelle considérée.

5.5 Correction de la sémantique

L'approche la plus naturelle consiste à reprendre tout le formalisme des chapitres 1 et 3, et l'étendre au cas où les types sont condidérés modulo une théorie équationnelle régulière, pour laquelle le problème d'unification est décidable et unitaire. En particulier, les propriétés 1, 2 et 3, page 13 sont conservées, ainsi que les résultats du chapitre 3.

Le cas des enregistrements peut ensuite être traité comme ajouts de primitives. Nous devons alors vérifier que les règles de réduction et les schémas de type des primitives donnés ci-dessus sont compatibles, dans le sens où les hypothèses H1 et H2 du chapitre 3 sont satisfaites.

Preuve : [de H1] Nous devons considérer les cas suivants :

Cas a **est** $\{\ell_i : v_i{}^{i\in 1..n}\}.\ell_j$ **:** où j est dans $1..n$. Alors une preuve du typage $A \vdash a : \tau$ impose que $A \vdash v_i : \tau_i$ pour i dans $1..n$ d'une part, et que τ soit égal à τ_j d'autre part. On a donc en particulier $A \vdash v_j : \tau$.

Cas a **est** $\{\ell_i : v_i{}^{i\in 1..n}\}||\{\ell_j = v'_j\}$ **:** . Alors une preuve du typage $A \vdash a : \tau$ impose que $A \vdash v_i : \tau_i$ pour i dans $1..n$ d'une part, et que $A \vdash v'_j : \tau'_j$ d'autre part. Nous distinguons deux cas: si j est dans $1..n$, alors le réduit a' de a est $\{\ell_i : v_i^{i\neq j\in 1..n}; l_j = v'_j\}$ et a pour type $\{\ell_i : \texttt{Pre}\ \tau_i^{i\neq j\in 1..n}; l_j = \tau'_j\}$. Sinon, le réduit de a est $\{\ell_i : v_i^{i\in 1..n}; l_j = v'_j\}$ et a pour type $\{\ell_i : \texttt{Pre}\ \tau_i^{i\in 1..n}; l_j = \tau'_j\}$. Dans les deux cas a' possède le type τ dans l'environnement A.

■

<u>Preuve</u>: [de H2] Nous raisonnons de la façon habituelle. Considérons un jugement $A \vdash v : \tau$. Alors:

- Si v est une fonction ou bien une constante fonctionnelle partiellement appliquée, alors τ est un type fonctionnel,
- Si v est un enregistrement, alors τ est un type enregistrement. De plus si v possède le champ ℓ, alors τ associe le type `Pre` τ' sur le champ ℓ.

L'égalité conserve toujours le constructeur de tête sur les types de la sorte `Type`, et préserve également la présence des champs. Les types engendrés par les différents cas ne peuvent donc pas se superposer. Les cas ci-dessus traitent tous les cas de valeurs. Nous pouvons donc inverser la propriété:

- Si τ est un type fonctionnel, alors v est une fonction ou bien une constante fonctionnelle partiellement appliquée,
- Si τ est un type enregistrement, alors v est un enregistrement. De plus si τ associe le type `Pre` τ' au champ ℓ, alors v possède le champ ℓ.

Nous pouvons maintenant montrer la seconde propriété en considérant les cas suivants de programmes irréductibles:

Cas a **est** $v.\ell$ **:** Une preuve du typage $A \vdash a : \tau$ impose que $A \vdash v : \{\ell : \texttt{pre}\ \tau'; \tau''\}$. Donc v est un enregistrement possédant le champ ℓ, et a peut se réduire.

Cas a **est** $v\ @\ \ell = v'$ **:** Une preuve du typage $A \vdash a : \tau$ impose que $A \vdash v : \{\ell : \tau'; \tau''\}$. Donc v est un enregistrement, et a peut se réduire.

■

5.6 Les variantes

Les types concrets sont des sommes déclarées. Les sommes polymorphes, encore appelées variantes sont donc aux types concrets ce que sont les enregistrements polymorphes aux enregistrementgs déclarés. Ainsi, l'idée est de pouvoir définir ou filtrer des objets construits sans devoir déclarer leur structure au préalable. Par exemple, on voudrait écrire:

```
let x = Cons (1, Nil) and let hd Cons (h,t) = h in hd x;;
```

Les variantes peuvent être formalisées par ajout de constantes : la primitive $\texttt{inject}_\ell$, d'arité 1, crée une variante avec l'étiquette ℓ ; la primitive ($\texttt{fun}\ \ell_1 \Rightarrow \_ \mid \ldots \ell_n \Rightarrow \_$) d'arité n, filtre des variantes selon l'étiquette ℓ_i avec laquelle elle a été construite. Par exemple, le programme ci-dessus pourra s'écrire :

```
let x = inject_Cons (1, inject_Nil())
and let hd = fun Cons (h,t) → h in hd x;;
```

La sémantique opérationelle est définie par la famille de δ-règles suivantes :

$$(\texttt{fun}\ \ell_i \Rightarrow {v_i}^{i \in I})\ (\texttt{inject}\ \ell_j\ v) \xrightarrow{\epsilon} v_i\ v \qquad (j \in I)$$

(On écrit aussi $\texttt{fun}\ \ell_i\ x_i \Rightarrow {a_i}^{i \in 1..i} n$ lorsque les v_i sont des fonctions $\texttt{fun}\ (x_i)\ a_i$).

Les variantes sont parfois considérées comme duales des enregistrements. En effet, on peut définir une traduction $[\![\_]\!]$ des programmes avec variantes vers les programmes avec enregistrements de la façon suivante : la primitive $\texttt{inject}_\ell$ est traduite par $\texttt{fun}\ (x\ y)\ y.\ell\ x$. Le filtrage ($\texttt{fun}\ \ell_i \Rightarrow {a_i}^{i \in 1..n}$) est traduit par $\texttt{fun}\ (x)\ x\ \{\ell_i = a_i\}$. Les autres constructions sont traduites de façon homomorphe.

Exercice 33 *Montrer que si $a_1 \longrightarrow a_2$, alors $[\![a_1]\!] \longrightarrow^* [\![a_2]\!]$.* □

Cette vision est souvent fructueuse, mais elle est aussi trompeuse. En particulier, l'usage des variantes est très différent de celui des enregistrements. Par exemple, il est naturel d'étendre les variantes avec une action par défaut qui sera appliquée lorsque la variante filtrée est construite avec une étiquette non explicite dans le filtre. La contrepartie de cette construction dans les enregistrements extensibles serait la possibilité de définir des enregistrements avec une valeur par défaut, qui sera retournée comme la projection de l'enregistrement sur un champ qui n'a pas été explicitement défini.

Nous pourrions typer les variantes en “remontant” le typage de leur traduction en objets enregistrements.

Exercice 34 *Essayer cette méthode. Pourquoi n'est-elle pas satisfaisante ?* □

Nous utilisons plutôt une méthode directe, tout en nous inspirant du typage des enregistrements. Nous notons $[\tau]$ le type des variantes décrites par un type de rangé τ (donc de la sorte $\texttt{Row}\ (\emptyset)$. Nous affectons aux primitives les schémas de type suivants :

$$\begin{aligned} \texttt{inject}_\ell &: \alpha \to [\ell : \texttt{Pre}\ \alpha; \alpha'] \\ \texttt{fun}\ \ell_1 \Rightarrow \_ \mid \ldots \ell_n \Rightarrow \_ &: (\alpha_1 \to \alpha) \to \ldots (\alpha_n \to \alpha) \to \\ &\qquad [\ell_1 : \texttt{Pre}\ (\alpha_1); \ldots \ell_n : \texttt{Pre}\ (\alpha_n); \emptyset] \to \alpha \end{aligned}$$

Dans les types de variantes, les expressions de type $\texttt{Pre}\ \tau$ et $\emptyset$ prennent un sens opposé à celui qu'elles ont dans les types d'enregistrements. Le type $[\ell_1 : \texttt{Pre}\ \tau_1; \ell_2 : \texttt{Pre}\ \tau_2; \alpha]$ est celui d'une variante qui est peut-être construite sur l'étiquette ℓ_1 avec une valeur de type τ_1 ou sur l'étiquette ℓ_2 avec une valeur de type τ_2. Un objet construit par

($\texttt{inject}_{\ell_1}\ v$) a donc un type de la forme $[\ell_1 : \texttt{Pre}\ \tau_1; \alpha]$ où τ est le type de v. En particulier, il est possible de substituer la variable α, par exemple par $(\ell_2 : \texttt{Pre}\ \tau_2; \alpha')$ et retrouver un type comme celui ci-dessus ; cela est nécessaire lorsque une variante peut provenir, par exemple de deux branches différences d'une conditionelle, et a donc pu être construit soit avec ℓ_1 soit avec ℓ_2, sans qu'il soit possible de le déterminer statiquement. Inversement, si le type $\emptyset$ est associé à une étiquette ℓ dans un type de variante τ, alors on sait qu'une variante de type τ n'a pas pu être construite avec l'étiquette ℓ. Par exemple, la construction de filtrage des variantes exige que l'objet filtré ait un type variante *fermé*, c'est-à-dire que les champs non explicitement mentionnés aient le type $\emptyset$, ce qui garantit la réussite du filtrage.

Exercice 35 *Montrer la correction du typage vis à vis de la sémantique pour les objets variantes.* □

Exercice 36 *En fait on peut ajouter un cas d'évaluation par défaut dans l'analyse par cas. Pour cela on fournit une autre collection de primitives* $\texttt{fun}\ \ell(x) \Rightarrow a_1 \ldots \mid \ell_n(y) \Rightarrow a_n\ \texttt{else}\ a_0$. *Donner les nouvelles règles de réduction. Quels schémas de type faut-il associer aux primitives. Montrer la correction du typage.* □

Conclusion

Dans ce chapitre nous avons montré comment étendre le langage ML avec des types enregistrements extensibles. Nous avons introduit les algèbres d'enregistrements, un formalisme général qui permet de traiter uniformément plusieurs variantes d'enregistrements. Nous n'avons pas décrit l'opération de concaténation sur les types enregistrements. Cette construction pose encore des problèmes de typage. Toutefois, dans de nombreux cas pratiques on peut ramener le problème de la concaténation à celui de l'extension [Rém93b]. Les variantes peuvent également être ajoutées à ML, de façon duale. Toutefois, elles sont d'un usage moins fréquent, sans doute parce leur aspect “ouvert”, c'est-à-dire la possibilité d'être mélangées entre elles de façon arbitraire, n'est pas très satisfaisante.

Pour en savoir plus

Il existe d'autres approches au typage des enregistrements. Celle utilisant des types étiquettés [Ber93] est très similaire. Une autre apporche intéressante, traitant le cas des enregistrements flexibles, mais pas celui des enregistrements extensibles repose sur l'utilisation de sortes sur les variables de type ordinaire [Oho90]. Enfin, nous verrons dans le chapitre suivant comment l'ajout de sous-typage permet également et de façon plus élégante de traiter le cas des enregistrements flexibles. Dans un cadre de typage explicite, on peut trouver plusieurs solutions utilisant également une relation de sous-typage. Par exemple, on pourra regarder [CM89, HP91].

Appendix : Simulation des enregistrements extensibles

Cette annexe décrit comment simuler les enregistrements extensibles en ML. Cela suppose que l'on connaisse à l'avance toutes les étiquettes utilisées dans tout le programme. Nous l'illustrons sur un exemple en considérant trois champs `a`, `b` et `c`. Les enregistrements extensibles seront simplement représentés par un enregistrement rigides à trois champs, certains étant marqués "absents".

```
type ('a, 'b, 'c) abc = {a:'a; b:'b; c:'c};;
```

Nous voulons garantir par le typage que l'accès à un champ autorisé réussit toujours. Nous utilisons deux types incompatibles pour représenter les champs présents et les champs absents :

```
type 'a pre = Pre of 'a;;
type abs = Abs;;
```

L'enregistrement vide est simplement

```
let nul = {a = Abs; b = Abs; c = Abs};;
```

Tous les autres enregistrements peuvent être construits par modification de champs à partir de l'enregistrement vide ; il suffit de définir trois fonctions :

```
let extend_a r x = {a = Pre x; b = r.b;   c = r.c  };;

let extend_b r x = {a = r.a;   b = Pre x; c = r.c  };;
let extend_c r x = {a = r.a;   b = r.b;   c = Pre x};;
```

Par exemple, l'enregistrement `{a = 1; b = 2}` s'écrit :

```
let deux = extend_b (extend_a nul 1) 2;;
let trois = extend_b (extend_c deux 3) 4;;
```

De façon similaire, nous définissons trois fonctions d'accès à un enregistrement :

```
let extract_a r = let (Pre v) = r.a in v;;

let extract_b r = let (Pre v) = r.b in v;;
let extract_c r = let (Pre v) = r.c in v;;
```

On vérifie que `extact_a ab` et `extract_b abc` sont typables, mais que `extract_c ab` ne l'est pas.

Le codage ci-dessus est tout à fait utilisable en pratique, soit pour une application particulière qui requiert des enregistrements extensibles alors qu'ils ne seraient pas fourni dans le langage, soit pour reprendre simplement l'idée du codage de la présence, sans les étiquettes, ou encore pour faire des exercices sur les enregistrements extensibles. Pour cela, il sera plus convivial de pré-définir un certain nombre d'étiquettes :

```
type 'a label = Label of 'a;;
```

```
let a = Label (extract_a, extend_a);;
let b = Label (extract_b, extend_b);;
let c = Label (extract_c, extend_c);;
```

Enfin, on peut utiliser des identificateurs cours et infixés pour retrouver une syntaxe plus habituelle, bien que cet exercice soit accrobatique :

```
let ( // ) r f = f r;;
let ( = ) (Label (_, extend)) x r = extend r x;;
let ( $ ) r (Label (extract, _)) = extract r;;
```

Les exemples ci-dessus s'écrivent alors très naturellement :

```
let deux = nul // (a = 1) // (b = 2) ;;
let trois = deux // (c = 3) // (b = 4);;
trois$b;;
```

Pour plus de sûreté, on pourra aussi contraindre les types des abréviations ci-dessus.

Chapitre 6

Typage avec contraintes

Dans ce chapitre, nous présentons une extension du système de typage avec des contraintes d'inclusion. Nous introduisons d'abord intuitivement puis formellement la notion de sous-typage qui nous mènera vers les types contraints. Ensuite nous présentons un système de typage avec contraintes d'inclusion et nous montrons qu'il garantit la sécurité du calcul. Nous utilisons comme exemple un langage avec enregistrements flexibles, mais le formalisme est plus général ; il s'applique à des enregistrements extensibles ainsi qu'à l'inclusion sur les types de base, les types des variantes, et bien sûr les types des objets.

La formalisation du typage avec contraintes telle qu'elle est présentée ici est très récente, et de nombreux aspects, notamment la simplification des contraintes, sont encore du domaine de la recherche.

Introduction

Les types enregistrements présentés dans le chapitre précédent ont permis de typer les opérations sur les enregistrements, notamment l'accès, de façon polymorphe. C'est-à-dire que la lecture d'un champ peut être effectuée par une même fonction, indépendemment des autres champs présents. Si l'on oublie l'extension des enregistrements, la seule opération possible sur les enregistrements est la lecture des champs. Un enregistrement peut être utilisé de façon sûr là où un enregistrement avec moins de champ peut l'être. Cela revient à donner à un enregistrement le type d'un enregistrement ayant moins de champs. Cette possibilité est souvent souhaitable, par exemple pour mémoriser dans une structure homogène telle que les listes des enregistrements ayant des domaines différents mais partageant tous une partie commune que l'on doit laisser visible.

Par exemple, l'enregistrement $a_{xy} \equiv \{x = 1; y = 2\}$ représentant un point dans le plan a pour type $\tau_{xy} \equiv \{x : \texttt{Pre Int}; y : \texttt{Pre Int}; \emptyset\}$. Il définit implicitement un point $a_x \equiv \{x = 1\}$ sur le premier axe de coordonnées qui a pour type $\tau_x \equiv \{x : \texttt{Pre Int}; \emptyset\}$. Clairement, il existe une fonction de coercion (ici une projection) qui transforme a_{xy} en a_x. Mais il est souvent plus intéressant de considérer que a_{xy} a lui-même le type τ_x et peut être utilisé partout où a_x le peut. Cela revient à supposer que la fonction de coercion est l'identité, en particulier elle ne change pas la valeur mais simplement le type de son argument. On dira alors que a_{xy} de type τ_{xy} a aussi le type τ_x parce que τ_x est un *sous-type* de τ.

La notion de sous-typage la plus simple est celle définie entre des types atomiques, comme par exemple dans le langage C, où le type des caractères est un sous-type du type des entiers, qui est lui même un sous-type du type des nombres réels. Souvent, ce genre de sous-typage n'est utilisé que pour permettre la surcharge des primitives arithmétiques et éviter ainsi à l'utilisateur de préciser complètement les opérations qu'il utilise ; il n'est alors pas essentiel. Dans la suite de ce chapitre, nous ne considérerons que le sous-typage sur les enregistrements. Il est plus utile, notamment pour son application aux objets, mais aussi plus général, et ce choix ne restreint pas l'étude du sous-typage : il sera toujours possible d'appliquer les résultats au sous-typage entre types atomiques, par exemple ceux des nombres.

Il est possible mais plus difficile de paramétrer entièrement la définition du langage par des constantes et des δ-règles comme nous l'avons fait pour ML. Pour rendre l'exposition plus simple, nous limitons notre langage à une extension de ML avec des enregistrements flexibles. Toutefois nous nous efforcerons de traiter le problème de la façon la plus générale possible, afin de pouvoir facilement transférer les résultats à d'autres constructions.

Nous reprenons donc le langage étudié dans le chapitre précédent mais en retirant la primitive d'extension des enregistrements. La syntaxe du langage est :

$$\begin{aligned} a &::= x \mid c \mid \texttt{fun}\ (x)\ a \mid a\ a \mid \texttt{let}\ x = a\ \texttt{in}\ a \\ c &::= \{\ell_1 = \_, \ldots \ell_n = \_\} \mid \_.\ell \end{aligned}$$

Les constantes $\{\ell_i = \_^{i \in 1..n}\}$ sont des constructeurs et les constantes $(\_.\ell)$ sont des primitives. Les valeurs sont donc les fonctions et les valeurs enregistrements partiellement appliquées.

$$v ::= \texttt{fun}\ (x)\ a \mid \{\ell_i = {v_i}^{i \in 1..n}, \ell_j = \_^{J \in 1..k}\}$$

La sémantique du langage est bien sûr inchangée. Les contextes d'évaluation sont donc, comme dans le chapitre 3 :

$$E ::= [] \mid E\ a \mid v\ E \mid \texttt{let}\ x = E\ \texttt{in}\ a$$

(Dans le cas d'ajout de références, la construction Let ne pourrait plus lier que des valeurs et la dernière forme du contexte d'évaluation serait donc supprimée). La relation de réduction est définie par la règle (β) et la famille de δ-règles :

$$\{\ell_i = {a_i}^{i \in 1..n}\}.\ell_j \xrightarrow{\epsilon} a_j \qquad j \in 1..n$$

Nous utiliserons les types enregistrements du chapitre précédent, mais pour simplifier nous retirons les variables de rangée. Ainsi la seule expression de rangée possible est $\underline{\emptyset}$, et nous pouvons noter $\{\ell_i : {\tau_i}^{i\in 1..n}\}$ au lieu de $\{\ell_i : {\tau_i}^{i\in 1..n}, \underline{\emptyset}\}$. Plus précisément, les expressions de types sont :

$$\tau ::= \alpha \mid \tau \rightarrow \tau \mid \{\ell : \tau^{i\in 1..n}\} \mid \emptyset \mid \texttt{Pre}\ \tau$$

Les types enregistrements sont considérés modulo permutation des champs et modulo la famille d'équations

$$\{\ell_i : {\tau_i}^{i\in 1..n}\} = \{\ell_i : \tau^{i\in 1..n}; \ell : \emptyset\} \qquad \forall \ell \notin \{{\ell_i}^{i\in 1..n}\}$$

qui exprime l'extraction de champs.

Dans la section **??**, nous étendons le système de type de ML avec une règle de sous-typage. En particulier, nous définissons une relation de sous-typage entre les types de ML. Cependant, nous montrons que cette approche pose des problèmes difficiles pour la synthèse de type. Aussi, nous présenterons dans la section 6.3 un système de typage avec contraintes d'inclusion, mieux adapté à la synthèse de type. En fait nous déduirons informellement le système avec contraintes de la relation de sous-typage introduite précédemment. Toutefois les deux formalismes ne sont pas équivalents. Par la suite, nous ne considérons plus que le formalisme des types contraints. Nous montrons dans la section 6.4 que le système de typage avec contraintes est correct vis à vis de l'évaluation.

Le typage avec contraintes est un formalisme très simple et assez général permettant d'étendre la synthèse de type en présence de sous-typage, tout en conservant les aspects essentiels du système de typage de ML. Par certains aspect, le typage avec contraintes est même plus simple que le typage de ML, du moins tant que l'on ne s'intéresse qu'à la sûreté apportée par le typage, en ignorant l'aspect de documentation qui nécessite une simplification des contraintes. Nous discuterons de ces problèmes à la fin de ce chapitre.

6.1 Sous-typage sur les types sans variable

Pour simplifier nous considérons d'abord le sous-typage pour les expressions de type sans variables.

Pour être suffisamment général (la raison apparaîtra ci-dessous), nous ajoutons aux types deux constantes $\bot$ et $\top$ avec l'idée intuitive que tout type est un sous-type de $\top$ et inversement $\bot$ est un sous-type de tout type.

Définition 1 Nous définissons une relation de sous-typage sur les types sans variables, notée $\tau <: \tau'$, comme la plus petite relation réflexive et transitive satisfaisant les règles suivantes :

$$\frac{(\text{<:-Bottom})}{\bot <: \tau} \qquad \frac{(\text{<:-Top})}{\tau <: \top} \qquad \frac{(\text{<:-Abs})}{\texttt{Pre}\ \tau <: \emptyset} \qquad \frac{\tau <: \tau'}{\texttt{Pre}\ \tau <: \texttt{Pre}\ \tau'}\ (\text{<:-Pre})$$

$$\frac{\tau_i <: \tau'_i \quad \forall i \in 1..n}{\{\ell_i : {\tau_i}^{i\in 1..n}\} <: \{\ell_i : {\tau'_i}^{i\in 1..n}\}}\ (\text{<:-Record}) \qquad \frac{\tau'_1 <: \tau_1 \qquad \tau_2 <: \tau'_2}{\tau_1 \to \tau_2 <: \tau'_1 \to \tau'_2}\ (\text{<:-Arrow})$$

□

Remarquons d'abord que les règles <:-Abs et <:-Record permettent de déduire que τ_{xy} est un sous type de τ_x et plus généralement, $\{\ell_i : \texttt{Pre}\ {\tau_i}^{i\in 1..n+k}\} <: \{\ell_i : \texttt{Pre}\ {\tau_i}^{i\in 1..n}\}$, en utilisant l'absorption des champs absents. La règle <:-Pre permet également le sous-typage en profondeur ; par exemple, on peut déduire de la relation $\tau_{xy} <: \tau_x$ que $\{z : \texttt{Pre}\ \tau_{xy}\} <: \{z : \texttt{Pre}\ \tau_x\}$. Dans la règle <:-Record chaque type doit être écrit avec les mêmes champs explicites, mais par l'équation d'extraction sur les types enregistrements, ce n'est donc pas une véritable restriction. Elle indique (comme la règle <:-Pre d'ailleurs) que le sous-typage se propage au travers de la structure, et ceci de façon covariante, i.e. le sens des inclusions entre deux types de même structure est le même que le sens des inclusions entre les sous-termes immédiats de même position. Ce n'est plus vrai pour les types fonctionnels, puisqu'une fonction de type $\tau_1 \to \tau_2$ peut aussi être considérée comme une fonction de type $\tau'_1 \to \tau'_2$ si $\tau_2 <: \tau'_2$ mais $\tau'_1 <: \tau_1$. On dit que la flèche est covariante à droite et contra-variante à gauche. Cela s'explique par la dissymétrie dans l'application entre les entrées (arguments) et la sortie (le résultat). Nous avons vu par exemple que la fonction d'extraction du champ x exige seulement que son argument soit un enregistrement possédant le champ x. Elle accepte a_x de type τ_x comme argument, et plus généralement tout enregistrement possédant plus de champs, donc tel que $\tau_{xy} <: \tau_x$; On a donc bien $\tau_x \to \tau <: \tau_{xy} \to \tau$ pour $\tau_{xy} <: \tau_x$.

Exercice 37 (*) *Supposons que l'on ajoute au système de typage la règle suivante (ce que nous ferons ci-dessous) :*

$$\frac{A \vdash a : \tau \qquad \tau <: \tau'}{A \vdash a : \tau'}\ (\text{Sub})$$

Étant donné quatre types τ_1, τ_2, τ'_1 et τ'_2 tels que $\tau'_1 <: \tau_1$ et $\tau_2 <: \tau'_2$. montrer que l'application égale à la fonction `fun` $(f\ x)\ f\ x$ *possède le type $(\tau_1 \to \tau_2) \to \tau'_1 \to \tau'_2$ sans utiliser le sous-typage sur des types flèche. En quoi cela peut-il servir à justifier la relation de sous-typage de la flèche.* □

La définition ci-dessus peut également s'appliquer aux termes avec variables ; toutefois, les seules relations possibles avec les variables sont $\bot <: \alpha$, $\alpha <: \alpha$ ou $\alpha <: \top$.

Exercice 38 (*) *Quels types peut-on donner à la fonction* `fun` (x) $\{\ell = x\}$ *? Admet-elle un type principal (le plus général) ?* □

Exercice 39 *On se pose la même question d'abord avec la fonction* `fun` (f) `fun` (x) $f\ x$ *puis avec la fonction* `fun` (f) `fun` (g) `fun` (x) $(f\ (g\ x), g\ (f\ x))$. *En quoi le second exemple est pire ?* □

L'exercice ci-dessus montre que pour typer les fonctions polymorphes, on peut être amené à supposer que les variables de types sont dans une relation de sous-typage.

Les axiomes ci-dessus ne permettent à deux variables d'être dans une relation de sous-typage que si elles sont identiques. Ceci n'est pas suffisant pour typer les fonctions polymorphes. Il est ainsi nécessaire d'abstraire la relation de sous-typage par rapport à des hypothèses sur certaines variables.

6.2 Contraintes avec variables

Nous considérons maintenant des types avec variables. On aimerait définir un jugement de la forme $C \vdash \tau_1 <: \tau_2$ où C est un ensemble d'hypothèses, i.e. de paires de types notées également $\tau_1 <: \tau_2$. Il faut lire "sous les hypothèses C, τ_1 est un sous-type de τ_2". Pour cela, nous remplaçons la relation $<:$ par un jugement de typage $C \vdash \tau_1 <: \tau_2$.

Définition 2 Nous définissons le jugement de typage $C \vdash \tau_1 <: \tau_2$ comme la plus petite relation satisfaisant les règles suivantes (nous devons maintenant inclure les règles correspondant à la transitivité, car la relation n'est plus binaire) :

$$\frac{\tau_1 <: \tau_2 \in C}{C \vdash \tau_1 <: \tau_2}\ (<:\text{-Context}) \qquad \frac{}{C \vdash \tau <: \tau}\ (<:\text{-Reflex}) \qquad \frac{C \vdash \tau_1 <: \tau_2 \qquad C \vdash \tau_2 <: \tau_3}{C \vdash \tau_1 <: \tau_3}\ (<:\text{-Trans}) \qquad \frac{}{C \vdash \tau <: \top}\ (<:\text{-Top})$$

$$\frac{}{C \vdash \bot <: \tau}\ (<:\text{-Bottom}) \qquad \frac{}{C \vdash \texttt{Pre}\ \tau <: \emptyset}\ (<:\text{-Abs}) \qquad \frac{C \vdash \tau <: \tau'}{C \vdash \texttt{Pre}\ \tau <: \texttt{Pre}\ \tau'}\ (<:\text{-Pre})$$

$$\frac{C \vdash \tau_i <: \tau_i' \quad \forall i \in 1..n}{C \vdash \{\ell_i : \tau_i^{\,i\in 1..n}\} <: \{\ell_i : \tau_i'^{\,i\in 1..n}\}}\ (<:\text{-Record}) \qquad \frac{C \vdash \tau_1' <: \tau_1 \qquad \tau_2 <: \tau_2'}{C \vdash \tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}\ (<:\text{-Arrow})$$

□

Nous pouvons en déduire également une relation $\vdash$ entre ensembles de contraintes, i.e. $C_1 \vdash C_2$ si pour toute contrainte c de C_2, on a $C_1 \vdash c$. Nous notons $\vdash C_2$ lorsque C_1 est l'ensemble vide. Nous retrouvons en particulier la définition précédente maintenant notée $\vdash \tau_1 <: \tau_2$.

Proposition 11 (Substitutivité) *Si $C_1 \vdash C_2$, alors $\theta(C_1) \vdash \theta(C_2)$ pour toute substitution θ.*

Preuve : Il suffit de voir que c'est vrai pour chaque jugement de sous-typage. ■

Proposition 12 (Transitivité) *Si $C_1 \vdash C_2$ et $C_2 \vdash C_3$ alors $C_1 \vdash C_3$.*

Preuve : Il suffit de remplacer l'utilisation des hypothèses de c_2 de C_2 dans la dérivation de $C_2 \vdash C_3$ par leur dérivation dans $C_1 \vdash c_2$. ■

Exercice 40 *Vérifier que que l'on ne changerait pas le sens de la relation $\vdash$ en limitant la règle <:-*Reflex *au cas où τ est un atome, c'est-à-dire une variable ou un des symboles $\{\emptyset, \bot, \top\}$ (les deux derniers cas étant eux mêmes inutiles.* □

Solvabilité et cohérence

Se pose immédiatement le problème de la forme des contraintes autorisées dans C. Sont-elles seulement des contraintes entre variables ? –cela ne serait pas suffisant. Doivent-elles êtres prouvables ? –mais alors elles seraient inutiles...

Certains ensembles de contraintes semblent aberrants parce qu'ils contiennent des contraintes "absurdes" telles que $\emptyset <: \texttt{Pre}\ \tau$. Une telle contrainte n'est évidemment pas valide (i.e. il est faut de dire $\emptyset \vdash \emptyset <: \texttt{Pre}\ \tau$), mais d'autres ensembles de contraintes non valides semblent plus raisonnables ; par exemple, l'ensemble réduit à la contrainte $\alpha <: \tau$ si τ n'est pas exactement α n'est pas valide ; cependant, il existe une substitution qui le rend valide, en l'occurrence $[\tau/\alpha]$.

Définition 3 (Solvabilité) Nous dirons qu'un ensemble de contraintes C est *solvable*, et on note $\models C$, s'il existe une substitution θ de $\mathcal{V}$ vers des termes sans variable telle que $\vdash \theta(C)$. □

La solvabilité est une définition sémantique. Nous alors par la suite donner une définition syntaxique et nous vérifierons que les deux coïncident exactement.

Remarque S'il existe une substitution θ vers des termes avec variables, telle que $\vdash \theta(C)$, elle peut-être spécialisée en une substitution sans variable, car on n'a pu utilisé d'autre règle sur les variables que la reflexivité, dont on peut instantier les variables par n'importe quel constante, et ainsi C est solvable. Nous avons conservé la restriction parce que le concept de solvabilité est liée à une notion sémantique, et il est usuel de donné la sémantique des termes avec variables par des ensembles de termes sans variables.

Exemple La contrainte $\emptyset <: \texttt{Pre}\ \tau$ n'est pas solvable, mais $\alpha <: \tau$ l'est.

Exercice 41 *Reprenons l'exercice 38 ci-dessus. Nous voulions donner à* $\texttt{fun}\ (x)\ \{\ell = x\}$ *le type* $\alpha \rightarrow \{\ell : \alpha'\}$ *pourvu que* $\alpha <: \alpha'$. *Vérifier que la contrainte* $\alpha <: \alpha'$ *est solvable. Montrer qu'on peut alors effectuer une dérivation dans le système de l'exercice précédent (première partie).* □

La solvabilité est une notion sémantique, difficile à manipuler : pour tester la solvabilité, il faudrait, a priori, essayer toutes les substitutions, mais aussi ensuite rechercher pour chaque contrainte une preuve de sa validité sans utiliser d'hypothèse. On cherche donc une notion syntaxique qui lui soit équivalente. C'est la notion de cohérence. On souhaite au mieux que la solvabilité corresponde à la cohérence. Au pire, on se contentera de du fait la cohérence implique la solvabilité.

Dans un premier temps, nous cherchons des propriétés des ensembles de contraintes solvables. Nous regarderons ensuite si ces propriétés sont suffisantes pour caractériser la solvabilité.

La solvabilité d'un ensemble de contrainte implique la solvabilité de chacune des contraintes. La réciproque est fausse bien sûr. Par exemple les contraintes $\tau \rightarrow \tau \rightarrow \alpha$ et $\alpha \rightarrow \{\}$ sont solvables indépendemment, mais $\{\tau \rightarrow \tau \rightarrow \alpha, \alpha \rightarrow \{\}\}$ ne l'est pas.

Remarquons que si $\models C_1$ et $C_1 \vdash C_2$, alors $\models C_2$. En effet si C_1 est solvable, alors il existe une substitution θ telle que $\vdash \theta(C_1)$. Par substitutivité, nous avons $\theta(C_1) \vdash \theta(C_2)$, puis par transitivité, nous avons $\vdash \theta(C_2)$. Donc $\models C_2$.

Nous pouvons utiliser la contra-posée comme une condition nécessaire de solvabilité.

Définition 4 (fermeture) On dit qu'un ensemble C est *fermé* (par transitivité et propagation) s'il satisfait les conditions suivantes :

$$\tau_1 <: \tau_2 \in C \wedge \tau_2 <: \tau_3 \in C \Longrightarrow \tau_1 <: \tau_3 \in C \qquad \text{(P-Trans)}$$
$$\texttt{Pre}\ \tau <: \texttt{Pre}\ \tau' \in C \Longrightarrow \tau <: \tau' \in C \qquad \text{(P-Pre)}$$
$$\tau_1 \rightarrow \tau_2 <: \tau_1' \rightarrow \tau_2' \in C \Longrightarrow \tau_2 <: \tau_2' \in C \wedge \tau_1' <: \tau_1 \in C \qquad \text{(P-Arrow)}$$
$$\{\ell_i : \tau_i{}^{i \in I}\} <: \{\ell_i : \tau_i'^{i \in I}\} \in C \Longrightarrow \forall i \in I, \tau_i <: \tau_i' \in C \qquad \text{(P-Record)}$$

□

On note C^* la fermeture de C, c'est-à-dire le plus petit ensemble fermé contenant C. Remarquer la fermeture d'un ensemble n'introduit que des relations de sous-typage entre des sous-termes de l'ensemble initiale. Elle comporte au plus n^2 termes composés de sous-termes. Elle est donc au plus de taille n^3 (n^2 avec partage). En fait, elle peut se calculer en temps $O(n^3)$ (comme la fermeture transitive simple).

Proposition 13 *Si* $\models C$, *alors* $\models C^*$.

Preuve : Il suffit de vérifier que $C^* \vdash C$. ■

Définition 5 (Incohérence immédiate) Un ensemble de contraintes est immédiatement incohérent s'il contient une contrainte $\tau_1 <: \tau_2$ tel que l'on, simultanément,

1. τ_1 et τ_2 ont des constructeurs de tête différents,
2. $\tau_1 \not\equiv \bot$ et $\tau_2 \not\equiv$,
3. $\tau_1 \not\equiv \texttt{Pre}\ \tau$ ou $\tau_2 \not\equiv \emptyset$.

□

Proposition 14 *Un ensemble de contraintes immédiatement incohérent n'est pas solvable.*

Preuve : Il suffit de le montrer pour une seule contrainte $\vdash \tau_1 <: \tau_2$. Par induction sur une dérivation du jugement précédent. Le jugement précédent n'est filtré par aucune des conclusions des règles que Trans. Mais alors l'une des deux prémisse est nécessairement immédiatement incohérente. ■

Comme corollaire, un ensemble fermé immédiatement incohérent n'est pas solvable. La réciproque n'est cependant pas vraie : une ensemble fermé qui n'est pas immédiatement cohérent n'est pas nécessairement solvable.

Une exemple est la contrainte $\{\alpha \to \bot <: \alpha, \bot <: \alpha \to \alpha\}$. On pourrait montrer que la relation d'équivalence engendrée par la relation de sous-typage est l'égalité. Donc ici, une solution devrait vérifier $\tau = \bot \to \tau$. Cela n'est possible qu'en présence de types récursifs.

Exercice 42 *Vérifier que l'ensemble de contraintes* $\{\alpha \to \bot <: \alpha, \alpha <: \bot \to \alpha\}$ *est cohérence. Montrer qu'il est solvable.*

L'ensemble de contraintes $\{\alpha \to \bot <: \alpha, \alpha <: \alpha \to \top\}$ *est-il solvable ?* □

La cohérence n'implique donc pas la solvabilité en l'absence de types récursifs. Par contre, cela devient vrai en présence de types récursifs. En fait nous ne connaissons de critère simple caractérisant la solvabilité sans types récursifs uniquement en l'absence de constructeur de type contra-variant. Ce sous-cas n'est pas intéressant.

6.2.1 Types récursifs

Présentation syntaxique On introduit une notation $\texttt{rec}\ \alpha.\tau$

$$\tau ::= \ \dots \mid \texttt{rec}\ \alpha.\tau$$

Il faut alors ajouter définir l'égalité par un système d'inférence, qui ajoute au règles structurelles (reflexivité, transitivité, congruence) les règles d'inférence suivantes :

$$\texttt{rec}\ \alpha.\tau = \tau[\texttt{rec}\ \alpha.\tau/\alpha] \qquad\qquad \frac{\tau_1 = \tau_2}{\texttt{rec}\ \alpha.\tau_1 = \texttt{rec}\ \alpha.\tau_2}$$

$$\frac{\tau_1 = \tau[\tau_1/\alpha] \qquad \tau_2 = \tau[\tau_2/\alpha] \qquad \tau \neq \alpha}{\tau_1 = \tau_2}$$

Arbre infinis rationnels Les types récursifs sont des arbres infinis rationnels, c'est-à-dire des arbres infinis ne possédant qu'un nombre fini de sous-termes. Un arbre infini est défini par une fonction qui associe à chaque occurrence (suite finie d'entiers non nuls) un symbole. Le domaine de cette fonction doit être clos par préfixes. Il doit aussi être bien formé, compte tenu de l'arité des symboles.

Un arbre rationnel peut aussi être défini implicitement par un ensemble d'équations récursives contractives, i.e. de la forme $(\alpha_i = \tau_i)^{i \in I}$ où aucun des termes τ_i ne n'est une variable.

Nous préférons cette dernière représentation.

Sous-typage sur les types récursifs On peut définir le sous-typage en étendant la présentation syntaxique, avec une nouvelle règle d'inclusion de la forme :

$$\text{(<:-}\textsc{Rec-Pos}\text{)} \quad \frac{C \oplus \alpha <: \tau \oplus \alpha <: \tau' \vdash \tau <: \tau'}{C \oplus \alpha <: \tau \vdash \tau <: \tau'} \qquad \text{(<:-}\textsc{Rec-Neg}\text{)} \quad \frac{C \oplus \tau <: \alpha \oplus \tau' <: \alpha' \vdash \tau' <: \tau}{C \oplus \tau <: \alpha \vdash \tau' <: \tau}$$

Pour les types sans variables, soit deux termes récursifs τ_1 et τ_2 définis par le système d'équations contractives C. Alors on défini $\tau_1 <: \tau_2$ comme $C \vdash \tau_1 <: \tau_2$.

On peut alors montrer que la relation de sous-typage sur les types sans variables est la plus grande relation immédiatement cohérente, transitive et fermée par propagation.

Nous admettrons la propriété suivante :

Proposition 15 *Un ensemble de contraintes C fermé et immédiatement cohérent admet une solution avec des types récursifs sans variable.*

6.2.2 Implication de contraintes

Nous avons défini une relation de sous-typage sur les types sans variable de façon syntaxique. Nous l'avons étendue à des jugements d'inclusion sous contraintes toujours de façon syntaxique.

D'un autre coté nous avons introduit une notion sémantique, la solvabilité, et nous avons montrée qu'elle coïncidait avec une notion syntaxique de cohérence.

Nous pouvons maintenant définir une notion sémantique d'implication de contrainte, afin de tester la généralité de notre définition syntaxique.

Définition 6 Nous disons qu'un ensemble de contrainte C_1 implique un ensemble de contrainte C_2, et nous notons $C_1 \supset C_2$, si toute solution de C_1 est également solution de C_2. □

Attention ! sous forme mathématique $C_1 \supset C_2$ signifie $\forall \theta, \theta(C_1) \supset \theta(C_2)$ ce qu'il ne faut pas confondre avec la propriété $(\models C_1) \implies (\models C_2)$ qui est une conséquence de la précédente. Comme cas particulier, nous avons alors un jugement sémantique d'inclusion $C \supset \tau_1 <: \tau_2$.

Nous savons, en combinant la substitutivité et la transitivité que si $C_1 \vdash C_2$, alors $C_1 \supset C_2$. Cette propriété est rassurante puisque les relations de sous-typage que l'on peut dériver syntaxiquement sont toujours correctes sémantiquement. La réciproque n'est pas vraie. Nous avons vue que $\models C$ entraîne $\models C^*$. En fait, la preuve montre que plus précisément, nous avons $C \supset C^*$. Le jugement sémantique inverse $C^* \supset C$ est vrai puisque le jugement syntaxique $C^* \vdash C$ l'est aussi. Mais, inversement, il est faut que $C \vdash C^*$. Les ensembles de contraintes C et C^* sont équivalents sémantiquement, mais pas syntaxiquement.

Nous pouvons définir une relation $\Vdash$ plus forte que la notion de déduction $\supset$. Nous écrivons $C_1 \Vdash C_2$ if $C_1^* \vdash C_2$. L'idée est que les contraintes de C_2 sont impliquées par celles de C_1, par fermeture ou par déduction. La relation $\Vdash$ est encore une notion syntaxique. Elle approche plus la relation sémantique, et possède de bonne propriété, mais elle n'est toujours pas identique à la notion sémantique. C'est encore un problème ouvert de trouver une caractérisation syntaxique de la relation sémantique.

Exercice 43 *Considérons l'ensemble de contraintes C réduit à $\alpha <: \top \to \alpha$ et $\bot \to \alpha' <: \alpha'$. Montrer que $C \models \alpha <: \alpha'$ alors que $C \not\Vdash \alpha <: \alpha'$.* □

Dans la suite nous utiliserons la relation $\Vdash$, mais on peut la remplacer par une autre relation, en particulier $\supset$, qui vérifie les propriétés suivantes :

Proposition 16 *la relation $\Vdash$ est réflexive et transitive. D'autre part, si $C_1 \Vdash C_2$, alors*

1. $C_1 \Vdash C_2^*$,
2. $(\models C_1) \Longrightarrow (\models C_2)$
3. $\forall C, C_1 \cup C \Vdash C_2 \cup C$. *(En particulier, $\forall\, \theta, \theta(C_1) \Vdash \theta(C_2)$.)*

Preuve : La réflexitivé et la transitivité découlent facilement de la définition et de la transitivité de $\vdash$. Supposons $C_1 \supset C_2$, c'est-à-dire $C_1^* \vdash C_2$.

Cas 1 : C'est-à-dire $C_1^* \vdash C_2^*$. On le montre par récurrence sur la construction de C_2^*.

Cas 2 : C'est-à-dire que si C_1^* est immédiatement cohérent. alors C_2^* l'est aussi. C'est une conséquence immédiate de $C_1^* \vdash C_2^*$ (première propriété).

Cas 3 : Il faut montrer $(C_1 \cup C)^* \Vdash C_2 \cup C$. Comme C_1^* est inclus dans $(C_1 \cup C)^*$ et $C_1^* \Vdash C_2$, on a $(C_1 \cup C)^* \supset C_2$. De même C est inclus dans $(C_1 \cup C)^*$, donc $(C_1 \cup C)^* \Vdash C$. ■

6.3 Un système de typage avec contraintes

Nous avons mentionné la nécessité de considérer des jugements de sous-typage sous des hypothèses d'inclusion. Dans l'exercice 38, page 63, nous avons également proposé d'associer à la fonction `fun` (x) $\{\ell = x\}$ le type $\alpha \to \{\ell : \texttt{Pre}\ \alpha'; \emptyset\}$ pourvu que $\alpha <: \alpha'$.

Nous devons naturellement incorporer les hypothèses d'inclusion dans les jugements de typage. Nous écrirons des jugements de typage de la forme $C, A \vdash a : \tau$ où C est un ensemble de contraintes *cohérentes*. Les contraintes C portent sur tout le jugement, donc sur A autant que sur τ.

Pour pouvoir écrire que $\texttt{fun}\ (x)\ \{\ell = x\}$ a le type $\alpha \to \{\ell : \texttt{Pre}\ \alpha'\}$ pourvu que $\alpha <: \alpha'$, et pouvoir lier cette expression, nous sommes aussi amenés à étendre les schémas de types pouvant apparaître dans un environnement de typage à des schémas de types contraints de la forme $\forall \bar\alpha \mid C.\, \tau$ où C est un ensemble de contraintes cohérentes. La notation $\forall \alpha.\, \tau$ peut être vue comme une abréviation pour $\forall \alpha \mid \emptyset.\, \tau$. Par exemple, le type de la fonction ci-dessus sera $\forall \alpha, \alpha' \mid \alpha <: \alpha'.\, \alpha \to \{\ell : \texttt{Pre}\ \alpha'\}$

Les constantes du langage ont pour schémas de types :

$$\begin{aligned} \{\ell_i : \_^{\,i \in 1..n}\} &: \forall \alpha_1, \ldots \alpha_n.\, \alpha_1 \to \ldots \alpha_n \to \{\ell_i : \texttt{Pre}\ \alpha_i^{\,i \in 1..n}\} \\ (\_.\ell) &: \forall \alpha.\, \{\ell : \texttt{Pre}\ \alpha\} \to \alpha \end{aligned}$$

Remarquons que le type de l'extraction est plus faible que le type que nous lui avions donné dans le chapitre précédent, puisque pour simplifier nous avons retiré les variables de rangée. En ML, ce type ne serait pas assez général car nous ne pourrions pas appliquer cette fonction à des enregistrements possédant d'autres champs que ℓ. Le mécanisme de sous-typage permet quand même de typer de telles applications.

Les règles de typage sont les suivantes :

$$\text{(Var-Inst)}\quad \frac{z : \forall \bar\alpha \mid C_0.\, \tau_0 \in A \qquad C \Vdash C_0[\bar\tau/\bar\alpha]}{C, A \vdash z : \tau_0[\bar\tau/\bar\alpha]} \qquad \text{(Fun)}\quad \frac{C, A \oplus (x : \tau_2) \vdash a : \tau_1}{C, A \vdash \texttt{fun}\ (x)\ a : \tau_2 \to \tau_1}$$

$$\text{(Gen-Let)}\quad \frac{C, A \vdash a_1 : \tau_1 \qquad C, A \oplus (x : \forall \bar\alpha \mid C_0.\, \tau_1) \vdash a_2 : \tau_2 \qquad \bar\alpha \notin FV(A) \cup FV(C \setminus C_0)}{C, A \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

$$\text{(App)}\quad \frac{C, A \vdash a_1 : \tau_0 \to \tau \qquad C, A \vdash a_2 : \tau_0}{C, A \vdash a_1\ a_2 : \tau} \qquad \text{(Sub)}\quad \frac{C, A \vdash a \vdash \tau \qquad C \Vdash \tau <: \tau'}{C, A \vdash a : \tau'}$$

L'abstraction et l'application sont typées comme en ML et les contraintes sont propagées des prémisses aux conclusions. Une variable dont le schéma de type est $\forall \bar\alpha.\, \tau$ est typée en substituant les variables génériques par des types pourvu que cette même substitution appliquée à C_0 soit un ensemble de contraintes impliquées par les contraintes du jugement. Le typage d'une liaison $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$ est aussi similaire à celui d'une liaison en ML, mais il faut généraliser les contraintes du typage de a_1 en même temps que le type τ_1 de a_1. En fait seule les contraintes possédant des variables génériques ont besoin d'être incluses dans le schéma de type. Il est important de laisser une copie des contraintes non généralisées dans C, car si la variable x n'est pas utilisée dans a_2, c'est la seule façon d'assurer la cohérence de C_0 (dont une instance est incluse dans C).

Enfin, la règle de sous-typage nous dit que si une expression a a un type τ dans un environnement A et sous les contraintes C, et si les contraintes (cohérentes par définition) impliquent que τ est un sous-type de τ', alors a a aussi le type τ. Sans la condition de cohérence des contraintes dans les jugements de typage, tous les programmes seraient typables.

Exercice 44 *En supposant que les constantes incluent les entiers, montrer que le non respect de la condition ci-dessus permettrait de typer le programme* $1\ (\texttt{fun}\ (x)\ x)$ *dans l'environnement vide avec le type* `Int` *pourvu qu'un certain ensemble de contraintes soit satisfaites. Montrer que ce programme n'est plus typable (en respectant la condition de cohérence)* □

Propriétés du système de typage

Par combinaison des règles Sub et App, la règle suivante est dérivable :

$$\text{(App')}\quad \frac{C, A \vdash a_1 : \tau_1 \to \tau \qquad C, A \vdash a_2 : \tau_2 \qquad C \Vdash \tau_2 <: \tau_1}{C, A \vdash a_1\ a_2 : \tau}$$

Une dérivation dans le système obtenu en remplaçant la règle App par la règle App' et où la règle Sub n'est éventuellement utilisée qu'en ultime position est dite *canonique*. On peut alors montrer le lemme suivant qui permettra de simplifier toutes les preuves en ne regardant que des dérivations canoniques.

Lemme 17 (Dérivation canoniques) *Tout jugement prouvable possède une dérivation canonique.*

Preuve : La démonstration est simple. Elle consiste à montrer que la règle Sub commute vers le bas avec toutes les autres règles (sauf App qui a été remplacée par App'), i.e. Var-Inst, Fun, App' et Let. Les seuls cas intéressants sont ceux de la règle App' et Fun, car ils utilisent les propriétés de propagation et de cohérence associées au symbole flèche.

Cas App' **:** Nous avons une dérivation de la forme

$$\dfrac{\dfrac{C, A \vdash a_1 : \tau_1 \qquad C \Vdash \tau_1 <: \tau_2 \to \tau\ (1)}{C, A \vdash a_1 : \tau_2 \to \tau}\ \text{(Sub)} \qquad C, A \vdash a_2 : \tau_2}{C, A \vdash a_1\ a_2 : \tau}\ \text{(App')}$$

La cohérence de C et (1) entraînent que τ_1 doit être de la forme $\tau_2' \to \tau'$ et la règle de propagation implique $C \Vdash \tau_2' <: \tau_2$ (2) et $C \Vdash \tau <: \tau'$ (3). On peut donc effectuer la dérivation suivante :

$$\dfrac{\dfrac{C, A \vdash a_1 : \tau_2' \to \tau \qquad \dfrac{C, A \vdash a_2 : \tau_2 \qquad (2)}{C, A \vdash a_2 : \tau_2'}\ \text{(Sub)}}{C, A \vdash a_1\ a_2 : \tau}\ \text{(App')} \qquad (3)}{C, A \vdash a_1\ a_2 : \tau'}\ \text{(Sub)}$$

Cas Fun **:** Nous avons une dérivation de la forme

$$\dfrac{\dfrac{C, A \oplus x : \tau \vdash a : \tau_1 \qquad C \Vdash \tau_1 <: \tau_2}{C, A \oplus x : \tau \vdash a : \tau_2}\ (\textsc{Sub})}{C, A \vdash \texttt{fun}\ (x)\ a : \tau \to \tau_2}\ (\textsc{Fun})$$

De $C \Vdash \tau_1 <: \tau_2$ nous pouvons déduire $C \Vdash \tau \to \tau_1 <: \tau \to \tau_2$. Ainsi, nous avons la dérivation suivante :

$$\dfrac{\dfrac{C, A \oplus x : \tau \vdash a : \tau_1}{C, A \vdash \texttt{fun}\ (x)\ a : \tau \to \tau_1}\ (\textsc{Fun}) \qquad C \Vdash \tau \to \tau_1 <: \tau \to \tau_2}{C, A \vdash \texttt{fun}\ (x)\ a : \tau \to \tau_1}\ (\textsc{Sub})$$

Cas Let **:** Il y a deux cas possibles, selon que la règle Sub peut être appliquée à la prémisse droite ou à la prémisse gauche. Dans le premier cas, la commutation est immédiate. Dans le second cas, nous avons une dérivation de la forme :

$$\dfrac{\dfrac{C, A \vdash a_1 : \tau_1 \qquad C \Vdash \tau_1 <: \tau_2\ (1)}{C, A \vdash a_1 : t_2}\ (\textsc{Sub}) \qquad C, A \oplus (x : \forall \bar{\alpha} \mid C_0 .\, \tau_2) \vdash a_2 : \tau}{C, A \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau}\ (\textsc{Let})$$

Il est alors facile de montrer, par récurrence sur la taille de la dérivation, que l'on peut dériver $C, A \oplus (x : \forall \bar{\alpha} \mid C_0 .\, \tau_2) \vdash a_2 : \tau$ en ajoutant dans la dérivation ci-dessus une règle Sub à toutes les occurrences de la règle Var-Inst appliquées à la variable x.

Cas Var-Inst **:** Comme c'est un axiome, cette règle ne peut pas être précédée d'une règle Sub.

■

Nous pouvons également montrer un lemme de substitution. Une différence par rapport à ML est qu'un jugement de typage n'est préservé par substitution, que si la substitution préserve la cohérence de ses contraintes.

Lemme 18 (Substitutivité) *Si $C, A \vdash a : \tau$ et θ est une substitution telle que $\theta(C)$ est cohérent, alors $\theta(C), \theta(A) \vdash a : \theta(\tau)$*

Les propriétés de généralisation et d'extension de contexte sont conservées : On dit que $\forall \bar{\alpha} \mid C .\, \tau$ est plus général $\forall \bar{\alpha}' \mid C' .\, \tau'$ s'il existe une substitution θ de domaine $\bar{\alpha}$ telle que $\theta(\tau) = \tau'$ et $\theta(C) \Vdash C'$.

Proposition 19 (Extension de l'environnement de typage) *Si les affectations de types A et B sont identiques partout sauf peut-être sur les variables qui ne sont pas libres dans a, alors $C, A \vdash a : \tau$ est dérivable si et seulement si $C, B \vdash a : \tau$ l'est.*

Proposition 20 (Généralisation du contexte) *Si $C, A \vdash a : \tau$ et si A et B ont même domaine, et que pour tout z de ce domaine, $B(z)$ est plus général que $A(z)$, alors $C, B \vdash a : \tau$.*

On montre également le lemme suivant qui correspond au renforcement d'hypothèses :

Proposition 21 (Renforcement des contraintes) *Si $C, A \vdash a : \tau$ et si $C' \Vdash C$ et C' est cohérent, alors $C', A \vdash a : \tau$.*

Preuve : Il suffit de remplacer chaque utilisation de $C \Vdash \tau <: \tau'$ dans une dérivation de la prémisse par $C' \Vdash \tau <: \tau'$ qui découle de la transitivité de $\Vdash$. ∎

6.4 Sûreté du typage

Nous montrons maintenant la correction du typage pour la sémantique à petits pas.

Comme pour le langage ML, la correction du typage s'exprime par deux résultats complémentaires. Le théorème 5 (d'auto-réduction) affirme que les typages sont préservés par réduction. Le théorème 6 (des termes irréductibles) certifie que les programmes bien typés irréductibles sont des valeurs. La combinaison des deux théorèmes permet donc d'affirmer que la réduction des programmes bien typés ne peut s'arrêter que sur des valeurs.

Un typage pour une expression a est un triplet (A, τ, C) tel que $C, A \vdash a : \tau$. L'inclusion des typages induit une relation $\subset$ sur les termes. Noues écrivons $a_1 \subset a_2$ pour signifier que tous les typages de a_1 sont aussi des typages de a_2. Le théorème d'auto-réduction s'écrit alors comme la préservation des typages pendant la réduction.

Théorème 5 (auto-réduction) *La réduction est une sous-relation de $\subset$ (avec des symboles, si $a_1 \longrightarrow a_2$, alors $a_1 \subset a_2$),*

La preuve est simplifiée par les lemmes suivants :

Lemme 22 *La relation $\subset$ est croissante, c'est-à-dire si $a_1 \subset a_2$, alors $E[a_1] \subset E[a_2]$.*

Preuve : Par récurrence structurelle sur E (voir le chapitre 3) ∎

Lemme 23 (Remplacement dans un terme) *Si $C, A \vdash v : \tau_0$ (1) et $C, A \oplus (x : \forall \alpha_0 \mid C_0 . \tau_0) \vdash a : \tau$ (2), et si les variables $\bar{\alpha}_0$ ne sont pas libres dans A et C_0 est un sous-ensemble de C contenant au moins toutes les coercions de C où apparaissent des variables de $\bar{\alpha}_0$, alors $C, A \vdash a[v/x] : \tau$.*

Preuve : La preuve est une récurrence structurelle sur l'expression a. Nous notons A_0 pour $A \oplus (x : \forall \alpha_0 \mid C_0 . \tau_0)$.

Dans chaque cas, nous considérons d'abord une dérivation canonique de (1) qui ne se termine pas par une règle Sub, et nous montrons qu'il est possible de construire une dérivation de (2). Si une preuve canonique de (1) se termine pas une règle Sub, alors il existe une dérivation $C, A_0 \vdash a : \tau'$ qui ne se termine pas par une règle Sub, et d'autre part $C \Vdash \tau' <: \tau$. Par hypothèse de récurrence, il existe une dérivation de $C, A \vdash a[v/x] : \tau'$, et l'on conclut par la règle Sub.

En fait, pour chaque cas le raisonnement est identique à celui du chapitre 1 en remplaçant les types par des types contraints.

Cas a **est** x **:** Les termes $a[v/x]$ et v sont égaux. De la dérivation de (2), il découle que τ est de la forme $\theta(\tau_0)$ où $dom(\theta)$ est égal à $\bar{\alpha}_0$, et $C \Vdash \theta(C_0)$ (3).

Nous pouvons appliquer le lemme de substitution 18 à (1). En effet, il découle de (3) et du fait que C est cohérent que $\theta(C_0)$ l'est aussi. On obtient donc $\theta(C), A \vdash v : \tau$. D'autre part C contient $\theta(C)$ et est cohérent. Par le lemme de renforcement des contraintes 21, $C, A \vdash v : \tau$.

Cas a **est** z **et** $z \neq x$ **:** Les termes $a[v/x]$ et a sont égaux et x n'est pas libre dans a. Donc $A(z)$ est identique à $A_0(z)$ et on peut conclure avec la même règle Var que celle utilisée pour typer a.

Cas a **est** `let` $y = a_1$ `in` a_2 **:** La dérivation canonique de (2) se termine par :

$$\frac{(3)\ \ C, A_0 \vdash a_1 : \tau_1 \qquad (4)\ \ C, A_0 \oplus (y : \sigma 1) \vdash a_2 : \tau}{A_0 \vdash \texttt{let}\ y = a_1\ \texttt{in}\ a_2 : \tau}\ (\textsc{Let})$$

où σ_1 est $\forall\, \bar{\alpha}_1 \mid C_1$. En utilisant le lemme de généralisation de contexte, nous pouvons supposé que $\bar{\alpha}_1$ inclut toutes les variables quie ne sont pas libres dans A. Ainsi l'ajout de $y : \sigma_1$ au contexte A ne modifie pas les variables libres du contexte. Par l'hypothèse de récurrence appliquée à (3), nous savons que $A \vdash a_1[v/x] : \tau_1$ (5). Par le lemme de généralisation de En réarrangeant les contextes par la propriété 21, page 72 on obtient $C, A \oplus (y : \sigma_1) \oplus (x : \forall\, \bar{\alpha}_0 \mid C_0.\, \tau_0) \vdash a_2 : \tau$ (6). Comme y n'est pas libre dans v, nous avons aussi $C, A \oplus (y : \sigma_1) \vdash v : \tau_0$ (7), puis en appliquant l'hypothèse de récurrence à (7) et (6) :

$$C, A \oplus (y : \sigma_1) \vdash a_2[v/x] : \tau\ (8)$$

Nous concluons par l'application de la règle Let à (5) et (8).

Cas a **est** `fun` $(y)\ a_1$ **:** La dérivation canonique de (1) doit se terminer par :

$$\frac{C, A_0 \oplus (y : \tau_2) \vdash a_1 : \tau_1\ (4)}{C, A_0 \vdash \texttt{fun}\ (y)\ a_1 : \tau_2 \rightarrow \tau_1}$$

où τ est $\tau_2 \rightarrow \tau_1$. En réarrangeant le contexte (4), nous obtenons $C, A \oplus (y : \tau_2) \oplus (x : \tau_0) \vdash a_1 : \tau_1$ (5). En étendant le contexte (2), nous avons $C_0, A \oplus (y : \tau_2) \vdash v : \tau_0$ (6). Par application de l'hypothèse de récurrence à (5) et (6), nous obtenons $C, A \oplus (y : \tau_2) \vdash a_1[v/x] : \tau_1$. Nous concluons en appliquant la règle Fun.

Cas a **est** $a_1\ a_2$ **:** La dérivation canonique de (1) doit se terminer par une règle App' et donc en particulier (en remplaçant la règle App' par sa définition) une règle App :

$$\frac{C, A_0 \vdash a_1 : \tau_2 \rightarrow \tau\ (1) \qquad C, A_0 \vdash a_2 : \tau_2\ (2)}{C, A_0 \vdash a_1\ a_2 : \tau}$$

En appliquant l'hypothèse de récurrence à chaque prémisse (qui sont toutes deux canoniques), nous obtenons $C, A \vdash a_1[v/x] : \tau_0 \rightarrow \tau$ (4) et $C, A \vdash a_2[v/x] : \tau_2$ (5). Puis la conclusion par une règle App.

■

Preuve : [de l'auto-réduction] En utilisant le lemme 22, il est suffisant de montrer l'auto-réduction pour chaque radical. De plus, nous pouvons montrer chaque cas indépendamment. Le lemme peut d'abord être montré pour des typages dont une dérivation canonique ne se termine pas par une règle de sous-typage. Le cas général en découle ensuite par application d'une règle de sous-typage.

Cas a_1 est `let` $x = v$ `in` a : La dérivation canonique se termine par :

$$\dfrac{C, A \vdash v : \tau_1 \ (3) \qquad C, A \oplus (x : \forall \bar\alpha \mid C'.\, \tau_1) \vdash a : \tau \ (1)}{C, A \vdash \texttt{let } x = v \texttt{ in } a : \tau} \ (\textsc{Let})$$

où, $\bar\alpha$ est inclus dans $FV(A)$ et $FV(C \setminus C')$. En particulier, les conditions du lemme 23 sont remplies, ce qui nous permet de conclure.

Cas a_1 est $(\texttt{fun } (x)\ a)\ v$: La dérivation canonique se termine par :

$$\dfrac{\dfrac{C, A \oplus (x : \tau_1) \vdash a : \tau \ (1)}{C, A \vdash \texttt{fun } (x)\ a : \tau_1 \to \tau} \ (\textsc{Fun}) \qquad \dfrac{C, A \vdash v : \tau_2 \ (2) \qquad C \Vdash \tau_2 <: \tau_0}{C, A \vdash v : \tau_1 \ (3)}}{C, A \vdash (\texttt{fun } (x)\ a)\ v : \tau} \ (\textsc{App})$$

Nous pouvons donc conclure par application du lemme 23.

Cas a est $\{\ell_i = a_i{}^{i \in 1..n}\}.\ell_j$ pour j dans $1..n$: La dérivation canonique doit se terminer comme suit :

$$\dfrac{\dfrac{C, A \vdash v_i : \tau_i \quad \forall i \in 1..n \ (1)}{C, A \vdash \{\ell_i = v_i{}^{i \in 1..n}\} : \{\ell_i : \texttt{Pre } \tau_i{}^{i \in 1..n}\}} \ (\textsc{Record}) \quad \vdots \qquad \dfrac{C, A \vdash (\ell_j.\_) : \{\ell_j : \texttt{Pre } \tau_0\} \to \tau_0 \qquad C \Vdash \{\ell : \texttt{Pre } \tau_0\} \to \tau_0 <: \{\ell_i : \texttt{Pre } \tau_i{}^{i \in 1..n}\} \to \tau \ (2)}{C, A \vdash (\ell_j.\_) : \{\ell_i : \texttt{Pre } \tau_i{}^{i \in 1..n}\} \to \tau} \ (\textsc{Sub})}{C, A \vdash \{\ell_i = v_i{}^{i \in 1..n}\}.\ell_j : \tau} \ (\textsc{App})$$

La relation (2) ne peut être satisfaite que si $C \Vdash \tau_j <: \tau_0$ et $C \Vdash \tau_0 <: \tau$ et donc en particulier $C \Vdash \tau_j <: \tau$ (3). La conclusion s'obtient en appliquant la règle Sub à (1) et (3). ■

Théorème 6 (Termes irréductibles) *Les termes bien typés irréductibles sont des valeurs.*

Preuve : La difficulté est de montrer que la forme des valeurs correspond à la forme de leur type. Ceci repose évidemment sur la cohérence des contraintes. Nous montrons d'abord que pour toute dérivation canonique du jugement $C, \vdash v : \tau$ qui ne se termine pas par une règle Sub,

- Si v est une fonction alors τ est $\tau_1 \to \tau_2$,
- Si v est de la forme $\{\ell_i = v_i{}^{i \in 1..n}\}$, alors τ est de la forme $\{\ell_i : \texttt{Pre } \tau_i{}^{i \in 1..n}\}$,

Les structures de ces types sont deux à deux disjointes ; nous avons considéré tous les cas de valeurs ; nous pouvons donc inverser les implications :

- Si τ est $\tau_1 \to \tau_2$ alors v est une fonction,
- Si τ est $\{\ell_i : \texttt{Pre } \tau_i{}^{i \in 1..n}\}$ alors v est un enregistrement de la forme $\{\ell_i = v_i{}^{i \in 1..n}\}$,

Si la dérivation canonique de $C, \vdash v : \tau$ se termine par une règle Sub, alors il y a une dérivation $C, \vdash v : \tau'$ qui ne se termine pas par une règle Sub et telle que $C \Vdash \tau <: \tau'$. Puisque τ' ne peut pas être une variable, nous pouvons déduire de la relation de cohérence :

- Si τ est $\tau_1 \to \tau_2$ alors τ' est de la forme $\tau'_1 \to \tau'_2$; ainsi v est une fonction.
- Si τ est $\{\ell_i : \texttt{Pre}\ \tau_i{}^{i\in 1..n}\}$ alors τ' est de la forme $\{\ell_i : \texttt{Pre}\ \tau'_i{}^{i\in 1..n+p}\}$, ainsi v est un enregistrement $\{\ell_i = v_i{}^{i\in 1..n+p}\}$.

Nous pouvons maintenant prouver le fait que les termes bien typés irréductibles sont des valeurs, par récurrence structurelle sur un tel terme a. Considérons une dérivation canonique d'un typage de a. Après avoir retiré la dernière règle de la dérivation si c'est une règle SUB, nous avons une dérivation du jugement $C, \vdash a : \tau$ (1) qui ne se termine pas par une règle SUB.

Cas a est x : n'est pas possible puisque a est une expression fermée.

Cas a est $\texttt{fun}\ (x)\ a_1$: alors a est une valeur.

Cas a est une application $a_1\ a_2$: La dérivation de (1) se termine par :

$$\frac{\vdash C, a_1 : \tau_2 \to \tau_1 \qquad C, \vdash a_2 : \tau_2}{\vdash C, a_1\ a_2 : \tau_1}$$

Par hypothèse de récurrence, a_1 est une valeur. Puisque a_1 a un type fonctionnel, ou bien c'est une fonction et a peut se réduire ou bien une application de primitive :

Sous-cas a est $\{\ell_i = a_i{}^{i\in 1..n}\}$: La dérivation canonique de (1) se termine par :

$$\frac{C, A \vdash a_i : \tau_i \quad \forall i \in 1..n \qquad \ldots}{C, A \vdash a : \{\ell_i : \texttt{Pre}\ \tau_i{}^{i\in 1..n}\}}\ (\text{APP})$$

Par hypothèse de récurrence, tous les a_i doivent être des valeurs donc a est également une valeur.

Sous-cas a est $a_1.\ell$: La dérivation canonique de (1) se termine par :

$$\frac{C, A \vdash a_1 : \{\ell_i : \texttt{Pre}\ \tau_i{}^{i\in 1..n}\} \qquad \ldots}{C, A \vdash a_1.\ell_i : \tau}\ (\text{APP})$$

Par hypothèse de récurrence, a_1 doit être une valeur, donc il est de la forme $\{\ell_i : v_i{}^{i\in 1..n}\}$ et a peut se réduire. ■

6.5 Discussion

Synthèse de types

Il est facile de déduire du formalisme de la section 6.3 un algorithme de synthèse de types avec contraintes. L'existence de dérivations canoniques conduit à un système de typage dirigé par la syntaxe. La seule flexibilité restante est dans l'ensemble des variables généralisables. On pourra choisir $\forall\, \bar\alpha \mid C_0.\, \tau$ pour $\text{Gen}(\tau, A, C)$ tel que $\bar\alpha$ soit le plus petit ensemble contenant toutes les variables libres de τ qui ne sont pas dans A, et toutes les variables libre de C qui ne sont pas dans A et atteignable à partir d'autres variables de α. On choisi alors pour C' le plus petit ensemble satisfaisant la condition. L'ajout ultérieur de contraintes à C' ne modifie pas ce choix.

Étant donné un programme a on cherche a trouver un type τ et un ensemble de contraintes C telles que $C, A_0 \vdash a : \tau$. L'ensemble de contraintes C devra être le même dans chaque sous jugement de typage. Il suffit donc de parcourir récursivement l'expression a en construisant le type τ et en collectionnant les contraintes C. Nous vérifions a posteriori la cohérence de C, ce qui nous permettra de garantir que chacune des étapes précédentes était correcte. Si C n'est pas cohérent, alors il n'existe pas de dérivation possible.

Pour la synthèse des types, on préferera en général effectuer des simplifications en cours de calcul comme évoqué ci-dessous. La présentation avec la règle App" décrite ci-dessous est également mieux adaptée au sous-typage.

Exercice 45 *Réaliser une maquette de la synthèse de type pour le langage de ce chapitre. Utiliser des structures de donnée simples sans trop se préoccuper de l'efficacité. L'algorithme devrait fonctionner en un temps très raisonnable sur des exemples de quelques lignes. Essayer les exemples de ce chapitre et quelques fonctions très simples.*

Il pourra être intéressant d'ajouter les variantes (voir l'exercice 46). □

Le nombre de contraintes obtenues pour une expression a sera au moins égal au nombre de nœuds d'application (c'est exactement ce nombre pour un programme sans liaison). Cet ensemble est donc au moins de la taille du programme. Il faut ensuite en vérifier la cohérence. La fermeture de l'ensemble des contraintes peut en rendre la taille proportionnelle au carré de la taille du programme. Cela pose un double problème. D'une part, les contraintes seront probablement illisibles par l'utilisateur. D'autre part, les calculs peuvent devenir très chers.

6.5.1 Simplification des contraintes

Pour cette raison, il est essentiel de pouvoir simplifier les contraintes. Il y a de nombreuses possibilités de simplification. On pourra d'abord utiliser le lemme de renforcement des contraintes21, voir même en faire une règle dérivée

$$\text{(Sub-Context)}\quad \frac{C, A \vdash a : \tau \qquad C' \Vdash C}{C', A \vdash a : \tau}$$

Le lemme de renforcement des contraintes montre que l'ajout de cette règle ne change pas l'ensemble des jugements dérivables. Par contre, elle augmente le nombre de dérivation d'un jugement, permettant de de simplifier les contraintes a posteriori.

On pourra aussi combiner cette règle avec la règle App'

$$\text{(App")}\quad \frac{C_1, A \vdash a_1 : \tau_1 \qquad C_2, A \vdash a_2 : \tau_2}{C_1 \cup C_2 \cup \{\tau_1 <: \tau_2 \to \alpha\}, A \vdash a_1\ a_2 : \alpha}$$

La règle App" permet de Cette règle est encore plus économique, permet d'ajouter les contraintes à la demande (au nœuds d'application) ; elle réduit la taille des dérivations, mais pas la taille des jugements finaux.

Pour réduire vraiment la taille des contraintes, il va falloir introduire plus de jugements de typage. On peut en fait enrichir la relation d'implication entre ensembles de contraintes, en se raprochant de la relation $\supset$.

Une autre approche est de permettre de simplifier les schémas de type. On peut définir une relation sémantique

$$C \supset \forall \bar\alpha \mid C_0.\, \tau_0 <: \forall \bar\alpha' \mid C'_0.\, \tau'_0$$

si pour toute substitution θ de $\bar\alpha$ telle que $\models \theta(C)$, pour toute substitution θ_0 de $\bar\alpha$ telle que $\Vdash (\theta_0 \circ \theta)(C_0)$, il existe une substitution θ'_0 de $\bar\alpha'$ telle que $\Vdash (\theta'_0 \circ \theta)(C'_0)$ et $\Vdash (\theta_0 \circ \theta)(\tau_0) <: (\theta'_0 \circ \theta)(\tau'_0)$. Malheureusemnt, cette relation est au moins aussi difficile que l'implication de contraintes $C \supset C'$ qui n'est pas encore bien comprise.

Toutefois, on pourrait utiliser un version syntaxique plus faible

$$C \Vdash \forall \bar\alpha \mid C_0.\, \tau_0 <: \forall \bar\alpha' \mid C'_0.\, \tau'_0$$

Cette relation sur les schémas, entraîne naturellement un relation entre les jugements de typage, donc une forme plus générale de Sub :

$$\textsc{(Sub)}\quad \frac{C, A \vdash a : \forall \alpha \mid C_0.\, \tau_0 \qquad C \Vdash \forall \alpha \mid C_0.\, \tau_0 <: \forall \alpha'_0 \mid C'_0.\, \tau'_0}{C, A \vdash a : \forall \alpha' \mid C.\, \tau'_0}$$

à condition de réintroduire les règles

$$\textsc{(Gen)}\quad \frac{C, A \vdash a : \tau_0 \qquad \bar\alpha \notin FV(A)}{C, A \vdash a : \forall \alpha \mid C.\, \tau_0} \qquad\qquad \textsc{(Inst)}\quad \frac{C, A \vdash a : \forall \alpha \mid C_0.\, \tau_0 \qquad C \Vdash C_0[\bar\tau/\bar\alpha]}{C, A \vdash a : \tau_0[\bar\tau/\bar\alpha]}$$

La simplification des contraintes est encore du domaine de la recherche, et nous ne la détaillerons pas plus ici.

Généralité de l'approche

Notre cas d'étude est un langage avec enregistrements, mais bien sûr, la méthode est générale. Elle s'applique au sous-typage sur les types de base mais aussi (et surtout) au typage des objets. Elle permet également d'améliorer le typage des variantes que nous avons traité dans le chapitre précédent.

Exercice 46 (**) *On reprend les variantes traitées dans le chapitre précédent par ajout de constantes. On fera la même simplification sur les types de variantes en retirant les variables de rangée. Quelle relation de sous-typage doit-on définir pour les types variantes (écrire les règles de propagation et de cohérence correspondantes) ? Quels schémas de type faut-il donner aux primitives de construction et destruction des variantes ? Montrer la correction du typage vis à vis de la sémantique.* □

Enfin nous avons présenté les enregistrements et les opérations associées comme des constantes de façon à montrer comment les preuves pourraient s'étendre à d'autres constantes. Il y a clairement un lien analogue à celui du chapitre 3 entre les δ-règles associées à ces constantes et leur typage (incluant les règles de propagation et les règles de cohérence), mais il est plus difficile de formaliser ce lien dans le cas général.

Nous avons considéré des enregistrements flexibles, non extensibles afin de simplifier le langage des types. Il est possible de revenir à des types enregistrements plus riches avec des variables de rangées comme dans le chapitre précédent. Le système de typage obtenu est plus expressif. Par exemple, il permet de typer l'extension des enregistrements, opération qui n'est pas typable de façon suffisamment polymorphe dans le système présenté dans ce chapitre.

Le sous-typage ne se substitue donc pas aux types enregistrements, mais il les complète de façon orthogonale, et l'on gagne à conserver les variables de rangées en présence de sous-typage.

Exercice 47 (**) *Donner un exemple de programme montrant que le sous-typage sans variable de rangée (comme dans ce chapitre) n'est pas aussi expressif que les types enregistrements polymorphes avec variables de rangées (comme dans le chapitre précédent). Donner un exemple inverse. On peut donc ré-introduire les variables de rangée dans les types d'enregistrements en présence de sous-typage. Quelles modifications faut-il apporter au formalisme de ce chapitre ? Vérifier que toutes les propriétés sont conservées.* □

Raffinement de la relation de sous-typage

Pour des raisons d'exposition, nous avons présenté ici la relation de sous-typage la plus simple entre enregistrements. Il est possible d'enrichir cette relation. Par exemple, il peut-être important de pouvoir connaître par le typage le domaine exact d'un objet enregistrement. Cela ne fonctionne pas avec la relation de sous-typage que nous avons présentée parce qu'il est toujours possible d'oublier la présence des champs dans un type-enregistrement ($\emptyset \Vdash \texttt{Pre}\ \tau <: \emptyset$). Une solution simple consiste à rejeter cette inclusion (i.e. la règle <:-Abs) et à la remplacer par les règles suivantes :

$$\text{(<:-}\textsc{Abs}\text{)}\ \frac{}{C \Vdash \emptyset <: \texttt{Either}} \qquad \text{(<:-}\textsc{Pre}\text{)}\ \frac{}{C \Vdash \texttt{Pre}\ \tau <: \texttt{Either}}$$

où `Either` est une nouvelle constante de type. Le diagramme suivant schématise la relation de sous-typage.

$$\tau ::= \ldots \mid \texttt{Either}$$

$$\begin{tikzcd} & \texttt{Either} & \\ \emptyset \arrow[ur] & & \texttt{Pre}\ \tau \arrow[ul] \end{tikzcd}$$

On pourra alors typer l'enregistrement $x \stackrel{def}{=\!=} \{x = 1\}$ avec $\{x : \texttt{Pre Int}; \emptyset\}$, mais l'expression $\texttt{if}\ x\ \texttt{then}\ \{x = 1\}\ \texttt{else}\ \{x = 2; y = 3\}$ ne pourra être typée qu'avec $\{x : \texttt{Pre Int}; y : \texttt{Either}; \emptyset\}$, se souvenant ainsi que le champ y puisse exister bien qu'innaccessible.

Exercice 48 (**) *On reprend l'exercice 46 sur les variantes. En fait on peut ajouter un cas d'évaluation par défaut dans l'analyse par cas. Pour cela on fournit une autre collection de primitives* $\texttt{fun}\ \ell(x) \Rightarrow a_1 \ldots \mid \ell_n(y) \Rightarrow a_n\ \texttt{else}\ a_0$. *Pour typer cette fonction il faut revenir à des types variantes plus explicites en faisant apparaître le terme de rangée* $\emptyset$ *et aussi autoriser des termes de rangée* `Either`.

$$\tau ::= \ \ldots [\ell_i : {\tau_i}^{i\in 1..i}, \underline{\tau}]$$
$$\underline{\tau} ::= \underline{\emptyset} \mid \underline{\texttt{Either}}$$

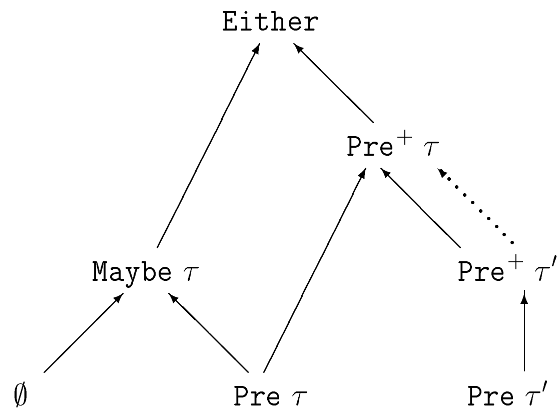
Les équations d'extraction sont, bien sûr :

$$\ell_i : {\tau_i}^{i\in 1..n}, \underline{\emptyset} \quad = [\ell_i : {\tau_i}^{i\in 1..n}, \ell : \emptyset, \underline{\emptyset}] \qquad [\ell_i : {\tau_i}^{i\in 1..n}, \underline{\texttt{Either}}] = [\ell_i : {\tau_i}^{i\in 1..n}, \ell : \texttt{Either}, \underline{\texttt{Either}}]$$

Préciser les nouvelles relations de sous-typage, les schémas de types pour les nouvelles primitives. Montrer la correction de cette extension vis à vis de l'évaluation. □

Une relation encore plus fine permet de préciser exactement sur quels champs l'inclusion a dû être utilisée :

$$\tau ::= \ldots \mid \texttt{Maybe}\ \tau \mid \texttt{Pre}^+\ \tau$$



On pourra remarquer que dans l'exemple ci-dessus le constructeur `Pre` est invariant.

De tels raffinement permettent de typer les programmes plus précisément, mais parfois aussi de typer plus de programmes, notemment dans les langages à objets.

Conclusion

Dans ce chapitre, nous avons introduit une relation d'inclusion sur les types. Nous avons d'abord montré comment elle permet de mieux typer les enregistrement. Mais une application essentielle de la relation d'inclusion est de pouvoir coercer implicitement les objets qui sont dans une relation de sous-typage, comme nous le verrons brièvement dans le chapitre **??**.

Nous avons introduit le formalisme des types contraints qui permet d'ajouter naturellement une relation de sous-typage au système de typage de ML, tout en préservant la correction du typage vis à vis du calcul.

Comme nous l'avons déjà plusieurs fois rappelé, le formalisme des types contraints est récent et de nombreux aspects sont encore du domaine de la recherche. La simplification des contraintes est un de ces aspects, et dans la messue où des algorithmes efficaces de simplification des contraintes existent, le système de typage avec contraintes pourrait peut-être remplacer avantageusement le système de typage de ML.

Pour en savoir plus

Le formalisme des types contraints que nous utilisons ici a été introduit par Pavel Curtis [Cur90], puis repris par A. Aiken [AW93] dans un cadre plus général. Une présentation plus récente, et plus proche du formalisme présenté ici a été réintroduite dans [EST95, EST] pour y être appliquée à des langages à objets. Pour une étude de la simplification des contraintes, on pourra aussi lire [Pot95].

Chapitre 7

D'autres extensions du langage ML et de son système de typage

Dans ce chapitre nous présentons d'autres extensions du language ML et de son système de typage. Dans une première partie, nous considérons des extensions du système de typage qui ne peuvent pas s'effectuer sans ajout de nouvelles primitives. L'extension du langage préserve le noyau ML, dans le sens où l'on retrouve simplement ML en retirant les nouvelles constructions du langage. On pourrait également parler d'extensions "horizontales". Nous présenterons également des extensions "verticales", où seul le système de typage est modifié, la syntaxe et la sémantique opérationelle du langage restant celles de ML, voir même du lamda-calcul.

Par exemple, l'ajout des références a nécessité des modifications importantes dans le formalisation de la sémantique, mais le système de typage est inchangé (dans la version basée sur la restriction de la liaison à des valeurs). Inversement, l'ajout d'objets enregistrements a nécessité une extension importante du système de typage, alors que leur sémantique s'écrivait très naturellement dans le formalisme existant. Nous présentons ci-dessous d'autres extensions du langage ML, nous ne les traitons pas avec le même soucis de détail que les précédentes.

7.1 Extensions horizontales

La plupart des extensions que nous avons vues jusqu'à présent rentrent dans cette catégorie. Les plus évidentes sont celles qui peuvent être traitées par simple ajout de constantes.

7.1.1 Types dynamiques

Il n'y a pas une seule bonne raison d'introduire des types dynamiques dans un langage, mais de nombreuses mauvaises raisons. Les types dynamiques sont en quelques sortes un paliatif à de nombreuses déficience du système de typage. Une valeur de type dynamique est la paire d'une valeur avec une représentation de son type ; son type statique est une nouvelle constante `Dyn` ne portant aucune information. La véritable information de type et la valeur d'origine peuvent être retrouvées à l'éxécution en explorant la représentation du type portée par la valeur dynamique.

La construction de valeurs de type dynamique permet, par exemple, de fabriquer des listes non homogènes, de sauver des données dans un fichier et de les relire de façon sure (puisque les donneées sont enregistrées avec leur type), de passer des arguments polymorphes à des fonctions (au prix de vérifications de types à l'exécution). Bien souvent, ces cas particuliers peuvent trouver des solutions mieux adaptées, mais toutes différentes.

Nous décrivons la forme la plus simple d'ajout de types dynamiques. Le langage est étendu avec deux constructions :

$$a ::= \ldots \mid \texttt{dynamic}\ (a : \sigma) \mid \texttt{typecase}\ a\ \texttt{of}\ (x, \sigma) \Rightarrow a\ \texttt{else}\ a$$

La première permet de construire des valeurs de type dynamique. La seconde extrait la "vraie" valeur contenue dans une valeur de type dynamique pourvu que son type soit suffisamment général ; elle retourne une valeur par défaut dans le cas contraire.

La sémantique des opérations sur les dynamiques est très proche de celle des types concrets, à la différence qu'elle fait intervenir les types du langage. Les contextes d'évaluation sont étendus à :

$$E ::= \ldots \mid \texttt{dynamic}\ (E, \sigma) \mid \texttt{typecase}\ E\ \texttt{of}\ (x, \sigma) \Rightarrow a\ \texttt{else}\ a \qquad (FV(\sigma) = \emptyset)$$

Dans leur formulation la plus simple, le filtrage des valeurs de type dynamique est exact, i.e. les règles de réduction sont

$$\begin{array}{r}(\texttt{typecase dynamic}\ (v, \sigma)\ \texttt{of}\ (x, \sigma) \Rightarrow a \ldots \texttt{else}\ a_0) \xrightarrow{\epsilon} (\texttt{fun}\ (x)\ a)\ v \\ (\texttt{typecase dynamic}\ (v, \sigma)\ \texttt{of}\ (x, \sigma') \Rightarrow a \ldots \texttt{else}\ a_0) \xrightarrow{\epsilon} a_0\end{array}$$

Les opérations sur les dynamiques ne peuvent pas être typées comme des ajouts de primitives, car elles font intervenir des schémas de types. Les règles de typage sont :

(Dynamic)
$$\frac{A \vdash a : \tau \qquad \bar{\alpha} \notin FV(A)}{A \vdash \texttt{dynamic}\ (a : \forall\, \bar{\alpha}.\, \tau) : \texttt{Dyn}}$$

(Typecase)
$$\frac{A \vdash a : \texttt{dynamic} \qquad A \oplus (x : \sigma) \vdash a_1 : \tau \qquad A \vdash a_2 : \tau}{A \vdash \texttt{typecase}\ a\ \texttt{of}\ (x : \sigma) \Rightarrow a_1\ \texttt{else}\ a_2 : \tau}$$

Exercice 49 *Les propriétés élémentaires du système de type de ML sont-elles conservées ?* □

Exercice 50 *Montrer la correction du typage vis à vis de la sémantique.* □

En fait on peut permettre une sémantique plus souple, et

$$\begin{aligned}&(\texttt{typecase dynamic } (v, \sigma) \texttt{ of}\\ &\qquad (x, \sigma') \Rightarrow a \texttt{ else } a_0) \xrightarrow{\epsilon} \begin{cases} (\texttt{fun } (x)\ a)\ v & \text{if } \sigma \leq \sigma' \\ a_0 & \text{otherwise} \end{cases}\end{aligned}$$

où $\sigma \leq \sigma'$ signifie que σ est plus général que σ'.

Il est aussi possible d'écrire `dynamic` (a) au lieu de `dynamic` $(a : \sigma)$; cette liberté, utile en pratique est plus délicate à formaliser. D'une part on perd la propriété de type principal. Par exemple le programme `fun` (x) `dynamic` (x) a pour types `Int` $\rightarrow$ `Dyn` et `Bool` $\rightarrow$ `Dyn` mais pas $\alpha \rightarrow$ `Dyn` car la condition de la règle Dynamic ne serait pas respectée. En pratique, on pourra par exemple échouer lorsque le programme n'admet pas de type principal. D'autre part, le type avec lequel est construite une valeur dynamique doit être fixé statiquement. Il est donc plus commode de voir cette proposition comme une synthèse de programme plutôt que comme une synthèse de types, la sémantique du langage étant définie pour les valeurs dynamiques explicitement typées.

Pour en savoir plus

Il est également possible d'étendre les schémas de types utilisés comme filtres avec des variables de motifs. On pourra consulter [ACR92, LM91] à ce sujet.

Enfin, une autre extension de ML intéressante et assez proche consiste à avoir des types dynamiques sans jamais construire de valeurs dynamiques, c'est-à-dire que les types étiquetant les valeurs dynamiques sont remplacés par des arguments supplémentaires transportant des informations de typage et ajoutés au besoin [DRW95].

7.1.2 Types existentiels

Les types dynamiques cachent complètement le type des valeurs ; ils permettent, par exemple, de contruire des listes hétérogènes. Nous proposons ici une autre façon plus systématique qui permet aussi de caché, mais partiellement, le type d'une valeur. Il s'agit des types existentiels. A la différence des types dynamiques, les types existentiels ne permettent pas de retrouver la partie cachée par un test à l'exécution.

Un exemple fréquent consiste à "empaqueter" ensemble dans une même structure une donnée avec des fonctions permettant de manipuler celle-ci. Par exemple, une paire $p_1 \equiv$ `(1, print_int)` d'un entier et d'une fonction d'impression aura le type `int * (int` $\rightarrow$ `unit)`. Dans ce cas le type révèle entièrement la structure et interdit donc de mélanger p_1 avec une autre paire $p_2 \equiv$ `(true, print_bool)`. Dans l'exemple ci-dessus, on aimerait ne retenir que le fait que la première composante de la paire est une valeur qui peut être passée en argument à la deuxième composante. On exprime cela en disant que ces paires ont le type $\exists\alpha.(\alpha \rightarrow \texttt{Unit}) * \alpha$; elles peuvent maintenant être mémorisées dans une structure homogène, par exemple une liste.

Il est facile de représenter le type $\exists\alpha.((\alpha \rightarrow \texttt{Unit}) * \alpha)$ en ML. Il suffit de définir un nouveau symbol de type, par exemple `pair`, mais d'arité 0, c'est-à-dire que la variable de type α est implicitement quantifiée et donc cachée. Il est aussi commode d'introduire un constructeur `Pair` de type $(\alpha * \texttt{Unit}) * \alpha \rightarrow \texttt{pair}$. En particulier, la création d'une valeur de type existentielle s'écrit `fun` (x) `Pair` (x).

```
type pair = Pair of ('a * ('a → unit));;
let p₁ = Pair (print_int, 1) : pair
and p₂ = Pair (print_bool, true) : pair;;
```

Mais que faire de ces objets une fois créés ? Il est possible de les déplacer, de les mémoriser bien sûr, mais comment en extraire les composantes et appliquer le second argument au premier ? Comment aussi empêcher d'appliquer le premier argument de p_1 au second argument de p_2 ? On écrira :

```
let (Pair (f,x)) = p₁ in f x;;
```

Dans le corps $f\ x$ de l'expression, la paire `(f,x)` a le type $(\omega \rightarrow \texttt{Unit} * \omega)$ où ω est un nouveau constructeur de type ω, différent de tous les autres. Le résultat `f x` est de type `Unit` dans lequel u n'apparaît plus. Plus généralement, à l'ouverture d'une valeur de type dynamique $\exists\alpha.\tau$ une nouvelle constante de type ω est introduite pour représenter la valeur inconnue de α. La règle de typage sera

$$\frac{A \vdash a_1 : \texttt{pair} \qquad (\texttt{pair} = \texttt{Pair of } \exists\alpha.\tau) \in A \qquad A \oplus (x : \tau[\omega/\alpha]) \vdash a_2 : \tau_2 \qquad \omega \notin \tau_2, A}{\texttt{let } x = \texttt{Pair } a_1 \texttt{ in } a_2 : \tau_2}$$

Nous avons affirmé plus haut les types existentiels n'avaient pas de coût à l'exécution. En effet le constructeur `Pair` est le seul dans son type, et il est donc superflu, comme le constructeur $(\_, \_)$ des produits.

Il est aussi possible de se passer de la déclaration des ces constructeurs, en considérant que tous les constructeurs sont prédéclarés : par exemple `pair` est remplacé par $(\_ : \exists\alpha.\alpha * (\alpha \rightarrow \texttt{Unit}))$. En fait, il faut faire attention à ne considérer que des déclarations de la forme $(\_ : \exists\alpha.\tau)$ pour des α-squelettes τ. Un α-squelette est un squelette de type τ (i.e. un type à trou) tels que α apparaît dans tous les sous-termes (différents de $\_$). Par exemple, $\alpha * (\_ \rightarrow \_)$ car α n'apparait pas dans $(\_ * \_)$, mais $\alpha * (\alpha \rightarrow \_)$ en est un. Le symbole pré-défini $(\exists\alpha.\alpha*(\alpha \rightarrow \_))$ d'arité 1 permet d'écrire `Unit` $(\exists\alpha.\alpha*(\alpha \rightarrow \_))$ au lieu de `pair`, ce que l'on abbrège évidemment en $\exists\alpha.\alpha * (\alpha \rightarrow \texttt{Unit})$ avec la convention usuelle.

Les règles de typage s'écrivent :

($\exists$-Intro)
$$\frac{A \vdash a_1 : \tau}{A \vdash (a_1 : \exists\alpha.\tau)}$$

($\exists$-Elim)
$$\frac{A \vdash a_1 : \exists\alpha.\tau \qquad A \oplus (x : \tau[\omega/\alpha]) \vdash a_2 : \tau_2 \qquad \omega \notin A, \tau_2}{\texttt{let } x = (a_1 : \exists\alpha.\tau) \texttt{ in } a_2 : \tau_2}$$

Pour en savoir plus

Introduits dans [LO92], les types existentiels que nous avons décrits ici sont très simples, car ils sont entièrement déclarés. Nous avons utilisé le formalisme des types concrets pour indiquer quand et comment (avec quels types) emballer ou ouvrir une structure de donnée. Ceci permet en particulier de modifier au minimum le système de type de ML (il a quand même fallu introduire la notion de constantes inconnues). Il est difficile, d'aller plus loin dans l'intégration des types existentiels et en faire des types de première classe tout en gardant la synthèse de types. On pourra se reporter à [Läu96] pour suivre une telle tentative, et découvrir ses limites !

Pour une preuve de correction on pourra se reporter à [Rém94a]. Dans le chapitre **??** nous verrons une autre façon de réaliser des types abstraits.

7.1.3 Types universels

Nous avons mentionné l'usage des types dynamiques pour passer des valeurs de types polymorphes comme argument des fonctions, comme l'illustre l'exemple ci-dessous.

```
let self_apply dx =
  typecase dx of (dx : All 'a. 'a → 'a) → x x
  else fail
in self_apply (dynamic (fun y → y : All 'a. 'a → 'a));;
```

Cette utilisation des valeurs de type dynamique est très inesthétique. En particulier, nous sommes obligés de prévoir un comportement anormal au cas où nous appliquons `xxx` a une valeur du mauvais type.

De façon analogue au types existentiels, nous écrivons :

```
type id = Id of All 'a. 'a → 'a;;

let self_apply ax =  let (Id x) = ax in x x
in self_apply (Id (fun x → x));;
```

Les types universels déclarés sont encore plus simples que les types existentiels déclarés. On pourra se reporter à [Rém94a] pour une formalisation plus précise ou [Läu96].

7.2 Extensions verticales

Nous décrivons ici des extensions essentielles du système de typage, en général au prix de perdre la synthèse des types. Nous nous limitons au noyau de ML décrit dans le premier chapitre.

7.2.1 Et si les types étaient récursifs ...

On peut étendre l'algèbre des types en remplaçant les termes de types par des expressions rationnelles. Les propriétés essentielles du langage sont conservées, en particulier, les programmes bien typés admettent des types principaux, et les programmes bien typés ne sont jamais bloqués.

Alors que dans le langage sans constante avec types récursifs, tous les termes seraient typables. C'est un peu pour cette raison que le système des types récursifs a été décrié.

Cependant, cela n'est plus vrai si le langage possède des constantes. Par exemple l'entier 1 appliqué à lui même ne peut pas être un programme bien typé car le type des entiers est incompatible avec celui des fonctions.

Le combinateur de point fixe introduit dans la section 3.3, page 29 est définissable dans cette extension de ML, car le programme

$$\texttt{let}\ Y\ F = \texttt{let}\ G\ f\ x = F\ (f\ f)\ x\ \texttt{in}\ G\ G$$

est bien typé. Donc un système de typage avec types récursifs n'a pas besoin d'une construction de récursion (ou d'un combinateur de point fixe primitif). Autoriser les types récursifs dans le langage invalide évidemment la propriété de terminaison pour les programmes bien typés. D'autres constructions comme les types récursifs déclarés, les exceptions ou bien sûr les structures mutables l'invalidaient déjà.

7.2.2 Types d'ordre supérieur

La règle Gen-Let de ML peut être remplacée par les deux règles suivantes :

$$\text{(Gen)}\ \frac{A \vdash a : \tau \qquad \alpha \notin dom\,(A)}{A \vdash a : \forall\,\alpha.\,\tau} \qquad \text{(Let)}\ \frac{A \vdash a_1 : \sigma \qquad A \oplus (x : \sigma) \vdash a_2 : \tau}{A \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau}$$

Il est maintenant possible de dériver des jugements $A \vdash a : \sigma$ mais l'ensemble des programmes typables avec chacun des deux systèmes est identique. La règle Gen signifie que si un programme possède le type τ dans un contexte A dans lequel la variable α n'apparaît pas, alors le programme possède le type polymorphe $\forall\,\alpha.\,\tau$, c'est-à-dire intuitivement qu'il possède tous les types $\tau[\tau_0/\alpha]$.

La fonction $\texttt{let}\ f = \texttt{fun}\ (x)\ x\ \texttt{in}\ f\ f$est typable en ML, en donnant à f le type polymorphe $\forall\,\alpha.\,\alpha \to \alpha$, mais on ne peut pas typer $(\texttt{fun}\ (f)\ f\ f)\ (\texttt{fun}\ (x)\ x)$. La raison est qu'il faudrait donner à l'argument f dans la première expression le type de $\forall\,\alpha.\,\alpha \to \alpha$ de f, et donc typer $\texttt{fun}\ (f)\ f\ f$ avec $(\forall\,\alpha.\,\alpha \to \alpha) \to \alpha' \to \alpha'$. Or en dans les schémas de type de ML les variables quantifiés ne peuvent à apparaître qu'en tête des types. On dit que les types de ML sont *quantifiés en tête* ou que la quantification est *prénexe*. Le système F étend le système de typage de ML en permettant des quantificateurs en possition arbitraire.

$$\tau ::= \alpha \mid \tau \to \tau \mid \forall\,\alpha.\,\tau$$

Les règles de typage sont

$$\text{(Fun)}\ \frac{A \oplus (x : \tau_2) \vdash a : \tau_1}{A \vdash \texttt{fun}\ (x)\ a : \tau_2 \to \tau_1} \qquad \text{(App)}\ \frac{A \vdash a_1 : \tau_2 \to \tau_1 \qquad A \vdash a_2 : \tau_2}{A \vdash a_1\ a_2 : \tau_1} \qquad \text{(Var)}\ \frac{z : \tau \in A}{A \vdash z : \tau}$$

$$\text{(Inst)}\ \frac{A \vdash a : \forall\,\alpha.\,\tau}{A \vdash a : \tau[t_0/\alpha]} \qquad \text{(Gen)}\ \frac{A \vdash a : \tau \qquad \alpha \notin A}{A \vdash z : \forall\,\alpha.\,\tau}$$

Exercice 51 (*) *Écrire une dérivation de typage de* `fun` $(x)\ x\ x$ *dans ce système.* □

Nous ne ferons pas une étude détaillée du système de typage ci-dessus. Dans le système F les typages sont préservés par réduction et les seuls programmes bien typés irréductibles sont les valeurs. Le typage garantit donc la sécurité de l'évaluation. Dans le système sans constantes les programmes bien typés ne peuvent pas être réduits indéfiniment. Par contre, l'inférence de type n'est pas décidable dans le système F.

Pour cette raison, nous pouvons nous intéresser à des sous-ensembles du système F. Nous définissons les types de rangs k de la façon suivante :

$$\begin{aligned} \tau_0 &::= \alpha \mid \tau \to \tau \\ \tau_{k+1} &::= \tau_k \mid \tau_k \to \tau_{k+1} \mid \forall\, \alpha.\, \tau_{k+1} \end{aligned}$$

Intuitivement, le rang mesure la profondeur des flèches traversées avant d'y trouver un quantificateur. Un type de τ_1 peut avoir des quantificateurs en tête seulement (ce sont les schémas de types de ML). Dans un type τ_2, un type τ_1 (c'est-à-dire un schéma de type de ML) peut aussi apparaître à gauche d'une flèche, etc.

La synthèse de type est possible dans le système où les contextes de typages sont de rang 1 et les types sont de rang 2, mais pas au delà.

On pourra également se reporter à [?] où est présenté plus en détail cette restriction décidable.

7.2.3 Types intersection

Dans le système F, une expression qui possède le type polymorphe $\forall\, \alpha.\, \tau$ possède aussi, en fait, tous les types $\tau[\tau_0/\alpha]$. Les types intersection permettent de donner un type, que nous notons $\tau_1 \cap \tau_2$, à une expression qui a à la fois le type τ_1 et le type τ_2. Les types intersection sont de la forme :

$$\tau ::= \alpha \mid \tau \to \tau \mid \tau \cap \tau$$

Les règles de typages sont :

$$\textsc{(Fun)}\ \frac{A \oplus (x : \tau_2) \vdash a : \tau_1}{A \vdash \texttt{fun}\ (x)\ a : \tau_2 \to \tau_1} \qquad \textsc{(App)}\ \frac{A \vdash a_1 : \tau_2 \to \tau_1 \qquad A \vdash a_2 : \tau_2}{A \vdash a_1\ a_2 : \tau_1}$$

$$\textsc{(Var)}\ \frac{z : \forall\, \bar\alpha.\, \tau_0 \in A}{A \vdash z : \tau_0[\bar\tau/\bar\alpha]} \qquad (\wedge\textsc{-Elim-L})\ \frac{A \vdash a : \tau_1 \wedge \tau_2}{A \vdash a : \tau_1} \qquad (\wedge\textsc{-Elim-R})\ \frac{A \vdash a : \tau_1 \wedge \tau_2}{A \vdash a : \tau_2} \qquad (\wedge\textsc{-Intro})\ \frac{A \vdash a : \tau_1 \qquad A \vdash a : \tau_2}{A \vdash z : \tau_1 \wedge \tau_2}$$

L'ensemble des programmes typables dans le système des types intersection est exactement l'ensemble des termes fortement normalisables. La synthèse de types pour ce système est donc indécidable, puisque savoir si un terme arbitraire est fortement normalisable est indécidable.

Il existe également des restrictions décidables intéressantes des types intersection. On définit le rang d'un type intersection de façon similaire au rang d'un type quantifié :

$$\begin{aligned} \tau_0 &::= \alpha \mid \tau \to \tau \\ \tau_{k+1} &::= \tau_k \mid \tau_k \to \tau_{k+1} \mid \forall\, \alpha.\, \tau_{k+1} \end{aligned}$$

La synthèse de type est décidable dans le système de type de rang 2 où les jugements de typage sont de rang 1 et les types de rang 2. Cette restriction décidable est détaillée dans [?].

On pourra également se reporter à [Pie91] pour un système de type qui comporte à la fois des types quanifiés et des types intersection d'ordre arbitraire.

Chapitre 8

Introduction aux Objets

Ce chapitre cours est avant tout une introduction aux notions fondamentales à la base de la programmation avec objets (auto-référence, classes, héritage, sous-typage) traitée de façon rigoureuse en se plaçant toujours dans un contexte fortement typé. C'est aussi un guide dans la découverte, par la pratique, de la programmation avec objets.

C'est aussi une application et un prolongement des différents mécanismes vus jusqu'ici, et une extension de ML avec des objets est aussi une façon de tester sa compréhension du cours.

8.1 Introduction

Dans ce chapitre nous présentons le concept de la programmation à objets, et illustrons la plupart de ses notions au travers de nombreux exemples écrits sans utiliser de constructions spécifiques aux objets. Nous espérons ainsi que cette approche progressive facilite la compréhension et aide à démystifier la notion d'objets.

La dernière partie présente les points délicats de la programmation à objets. Il n'est pas nécessaire de les comprendre dans le détail dans un premier temps, car nous reviendrons sur les difficultés dans les chapitres suivants.

8.1.1 Pourquoi la programmation à objets ?

L'idée d'objets est apparue à une époque où les langages de programmation ne fournissaient aucune aide à la programmation à grande échelle. Celle-ci est très différente de la programmation à petite échelle. Le concept essentiel de la programmation à grande échelle est la modularité, que l'on peut décliner sous différents aspects : sûreté, flexibilité, évolutivité, réutilisabilité, etc.

Une notion qui n'est pas entièrement dérivée de la modularité, mais qui lui est très liée est celle d'encapsulation, qui peut apparaître sous des formes différentes : fermeture, types abstraits, objets, etc.

Modularité

La modularité est la possibilité d'organiser un programme en plusieurs entités (modules) relativement indépendantes. Cela permet une plus grande flexibilité dans la mise au point des programmes, chaque module pouvant être implémenté séparément. L'écriture d'un programme de façon modulaire augmente aussi la sécurité, car la spécification du comportement d'un module, est plus simple que celle de l'ensemble, et sa réalisation peut être testée séparément. Le comportement du système tout entier peut être compris en se limitant à une abstraction du comportement de ses parties.

La modularité favorise également l'évolutivité des programmes. Puisqu'il suffit de comprendre une sous-partie d'un programme (et sa spécification) pour la modifier. Enfin, la programmation modulaire doit aussi faciliter la réutilisabilité des programmes. En effet, en écrivant un programme par assemblage de parties plus élémentaires, on s'attend à ce que certaines de ses parties soient en fait des bibliothèques pouvant être réutilisées dans d'autres programmes.

Encapsulation

L'encapsulation désigne la possibilité de construire des valeurs en cachant une partie de leur implémentation. Cela garantit qu'une autre partie du programme ne pourra pas directement accéder aux parties cachées de l'implémentation ; au contraire, il sera nécessaire de communiquer en utilisant les fonctions fournies dans l'interface, ce qui assure le respect de certains invariants, et permet également de faire des vérifications dynamiques de sûreté aux points de communication.

En ML, par exemple, les fonctions peuvent encapsuler des données dans leur fermeture. L'encapsulation peut aussi être réalisée par abstraction de type.

Un test de modularité

Considérons un programme typique en ML : nous allons construire des données structurées, puis les manipuler. Bien souvent, les données sont représentées par des arbres. Nous pouvons définir un type concret (récursif) :

```
type arbre = A of arbre * arbre | B of bool | C of int;;
```

Puis nous ajoutons des opérations d'impression, d'évaluation, etc...

```
let (imprime : arbre → unit) = ...
let (évalue : arbre → int) = ...
```

Plus tard, nous pouvons être amenés à modifier notre programme de deux façons différentes :

- soit en enrichissant la structure de données : par exemple, en ajoutant un nouveau type de nœuds aux arbres manipulés ; on parlera d'*extension horizontale*.

- soit en ajoutant des opérations sur la structure existante : par exemple, en ajoutant une fonction de simplification :

```
let (simplifie : arbre → arbre) = ...
```

On parlera d'*extension verticale*.

L'extension verticale est une opération modulaire en ML, alors que l'extension horizontale ne l'est pas. En effet, pour ajouter une nouvelle fonction, il suffit d'écrire celle-ci à la suite de toutes les autres. Il n'est pas nécessaire de modifier les anciennes définitions. En particulier, on peut écrire les anciennes définitions dans un fichier, et la nouvelle définition dans un nouveau fichier.

Par contre, cela n'est pas possible pour l'extension horizontale. D'une part, il n'est pas prévu de pouvoir ajouter un constructeur à un type concret déjà défini[1]. Pire, après avoir déclaré un nouveau constructeur dans le type arbre, il faut corriger les définitions des fonctions `imprime` et `évalue` pour y préciser le calcul à effectuer sur les arbres construits avec le nouveau nœud.

À l'inverse, les objets offrent une bonne modularité horizontale, mais ils rendent plus difficile la modularité verticale. Nous reviendrons sur cette bivalence de la modularité dans la conclusion de ce cours.

8.1.2 Un style de programmation

Tout comme la programmation impérative, ou la programmation fonctionnelle, la programmation à objets est avant tout un *style* de programmation. Bien sûr, nous retrouvons dans tous les exemples écrits dans ce style des constructions similaires, mais aussi des problèmes similaires et des façons analogues de les résoudre avec leurs variantes.

Le style à objets n'est donc pas réservé à des programmes écrits dans un langage à objets. Un langage à objets est simplement un langage offrant des constructions de base qui facilitent la programmation dans un style à objets.

Nous d'abord découvrir la notion d'objets au travers d'exemples écrits dans le langage ML. Nous montrerons aussi les limites de cette approche, ce qui justifiera l'utilisation de constructions primitives.

Un objet est une valeur regroupant un ensemble de données, et à laquelle sont associés un certain nombre d'informations. Un point sur la droite, par exemple, peut être représenté par un type concret et les définitions suivantes :

```
type point = Point of int ref;;
let p = Point (ref 1);;
let getx  (Point x) = !x;;
let setx  (Point x) = fun y → x := y;;
let print (Point x as self) = print_int (getx self);;
```

Dans la terminologie des objets, `ref 1` dans `p` est l'état interne de l'objet, `getx`, `setx` et `print` sont les méthodes pouvant s'appliquer à l'objet `p`. En général, beaucoup d'objets partagent le même comportement, ici tous les objets ayant le constructeur `Point`. On peut évidemment définir une fonction

[1] Nous verrons ci-dessous que cela n'est ni tout à fait vrai, ni vraiment fondamental.

```
let new_point v = Point (ref v);;
```

pour créer de nouveaux points sans avoir à se soucier de l'implémentation des points. On pourra dire que ces objets sont de la même *classe* (nous préciserons cette notion ci-dessous).

Dans la programmation avec objets, le calcul est effectué par envoi de messages aux objets. Ici, c'est simplement une application de la méthode de même nom que le message à l'objet adressé :

```
setx p (getx p + 1); print_int (getx p);;
```

La programmation par envoi de messages signifie aussi que l'on puisse adresser le même message à des objets de différentes classes, le comportement de l'objet à la réception d'un message étant déterminé en fonction de sa classe.

Dans le langage ML, on peut aussi définir des points avec une information de couleur. En suivant le modèle ci-dessus, nous obtenons :

```
type point_coloré = Point_coloré of int ref * string;;
let new_point_coloré v c = Point_coloré (ref v, c);;
let getx  (Point_coloré (x,c)) = !x;;
let setx  (Point_coloré (x,c)) y = x := y;;
let print (Point_coloré (x,c) as self) =
               print_int (getx self); print_string c;;
```

Mais alors,

```
let p = new_point_coloré 1 "gris" in setx p (getx p); print p;;
```

Par contre, il n'est plus possible d'écrire

```
let p = new_point 1 in setx p (getx p);;
```

En effet, la méthode `getx` qui a été redéfinie, requiert maintenant des points colorés. On aurait pu choisir un nouveau nom distinct du nom de l'ancienne méthode ; mais ceci limite fortement la programmation par envoi de message.

Une solution à ce problème serait que `Point` et `Point_coloré` soient les constructeurs d'un même type. Il existe en ML un type extensible permettant cela, c'est le type des exceptions (dans le langage Ocaml utiliser le mécanisme des modules pour définir de nouveaux types extensibles). Le type des objets peut être un type extensible, étendu à la définition de chaque nouvelle classe. Le type `exn` des exceptions, le seul type extensible en ML, mais il serait possible de fournir à l'utilisateur d'autres types extensibles. Reprenons les deux exemples ci-dessus :

```
exception Point of int ref;;
let new_point v = Point (ref v);;
let getx  (Point x) = !x;;
let setx  (Point x) y = x := y;;
let print (Point x as self) = print_int (getx self);;
```

```
exception Point_coloré of int ref * string;;
let new_point v c = Point_coloré (ref v, c);;
let getx =  function (Point_coloré (x,c)) → !x | p → getx p;;
let setx =  function (Point_coloré (x,c)) → fun y → x := y | p → setx p;;
let print = function (Point_coloré (x,c) as self) →
                  print_int (getx self); print_string c
              | p → print p;;
```

Cette solution présente clairement quelques aspects du style objet. Toutefois, cet exemple n'est pas satisfaisant car le typage ne garantit pas que l'envoi d'un message réussit toujours. En effet, si la méthode n'est pas définie pour l'objet auquel le message est envoyé, une exception de filtrage sera levée.

Il existe une autre approche, duale, que l'on peut également ébaucher dans le noyau ML. L'idée est d'attacher les méthodes à l'objet lui-même, ce qui évite le problème de leur surcharge :

```
let new_point x =
  let r = ref x in
  let getx = function () → !r in
  let setx = function y → r := y in
  (getx, setx);;
```

Les données sont appelées les *variables d'instance* (l'état interne étant l'ensemble de toutes les variables d'instance). Les méthodes sont les opérations pouvant s'appliquer à ces données. Dans cet exemple, seules les méthodes ont été retournées. L'état interne a été encapsulé à l'intérieur de l'objet.

En pratique, il est commode de nommer les méthodes. Un objet devient un enregistrement de méthodes. Nous pouvons réécrire l'exemple précédent :

```
type point = {getx : unit → int; setx : int → unit};;
let new_point v =
  let x = ref v in
  let getx () = !x in
  let setx y = x := y in
  {getx = getx; setx = setx};;
```

Pour créer un point, il suffit d'appliquer la fonction `new_point` à une valeur initiale.

```
let p = new_point 1 in
  p.setx (p.getx() +1); print_int (p.getx());;
```

Peut-on encore comme dans l'exemple précédent étendre les points avec une composante de couleur ? Oui, mais comme dans le cas précédent, l'envoi de message polymorphe (le même message peut être envoyé à des objets de différentes classes) ne sera pas possible.

Ci-dessus nous avions résolu le problème en utilisant des sommes extensibles. De façon similaire, on peut remplacer les enregistrements par des enregistrements extensibles, et retrouver l'envoi de messages polymorphes. Cette approche sera reprise plus loin dans ce chapitre de façon plus systématique dans un langage ML avec des enregistrements extensibles.

Deux approches Les deux exemples (très simples) ci-dessus mettent en avant deux approches des objets. Dans la première les objets transportent leurs variables d'instance et un constructeur permettant d'identifier leur classe. Les méthodes sont définies à l'extérieur des objets, et pour chaque classe auxquelles elles doivent s'appliquer. La sélection du code à utiliser se fait dynamiquement à partir du constructeur de l'objet dans notre exemple. Afin de garantir un typage statique de l'envoi de messages, la sélection sur le constructeur peut être remplacée par une sélection par filtrage sur le type de l'objet à l'exécution. Ce modèle est appelé *le modèle à surcharge*.

Dans la seconde approche les objets portent directement les méthodes qui peuvent leur être appliquées. Elles sont rangées dans un enregistrement, de telle façon que l'envoi de message consiste à appliquer la méthode de l'objet enregistrée sous ce nom. Ce modèle est appelé le modèle à enregistrements.

Dans le modèle à enregistrements, l'état interne de l'objet peut facilement être caché (dans la fermeture des méthodes, ou bien par un mécanisme de typage). Ce modèle permet donc l'encapsulation. Ce n'est pas le cas du modèle à surcharge ; la sélection étant faite sur son type, un objet doit révéler toute information qui détermine son comportement.

Bien que le modèle à surcharge semble plus naturel, il est en fait plus difficile à mettre en œuvre dans un contexte fortement typé. Nous développons plus en détail le modèle à enregistrements dans le reste de ce chapitre afin d'introduire les autres concepts de la programmation avec objets. Nous reviendrons sur le modèle à surcharge dans le chapitre **??**

8.1.3 Objets et Classes avec des enregistrements extensibles (ou des objets sans objets)

Nous poursuivons l'idée précédente que la programmation à objets peut se faire sans objets. nous reprenons le modèle à enregistrements. Pour aller plus loin dans la découverte des objets, nous supposons que le langage est équipé d'enregistrements extensibles.

Les enregistrements extensibles sont une extension des enregistrements de ML, ayant les particularités suivantes :

- Il n'est pas besoin de les déclarer ; tout identificateur peut être utilisé comme une étiquette. Par exemple,

```
let ab = {a = 1; b = 2};;
```

- Pour toute étiquette fixée, il est possible d'écrire une fonction polymorphe qui retourne le contenu du champ associé et que l'on puisse appliquer à tout enregistrement possédant ce champ. Par exemple :

```
fun x → x.a;;
```

- Pour tout étiquette fixé, il est possible d'écrire une fonction polymorphe qui, étant donné un enregistrement et une valeur quelconque, retourne un enregistrement identique au premier excepté sur ce champ où il est défini avec la valeur donnée en argument. Par exemple :

  ```
  fun r x → r || {a = x}
  ```

Le but n'est pas ici de formaliser les enregistrements extensibles, mais de les utiliser pour comprendre les mécanismes des objets. La partie suivante permet de se faire une bonne idée de leur fonctionnement, mais sa lecture n'est pas indispensable pour comprendre la suite de ce chapitre.

L'annexe 5.6 montre comment simuler les enregistrements extensibles en ML (dans le cas où toutes les étiquettes utilisés dans tout le programme sont connues à l'avance).

Objets et Classes

Reprenons rapidement l'exemple des points. Nous représentons également l'état interne par un enregistrement (dont les champs sont les variables d'instance). Un objet, par exemple un `point`, est définit par son état interne `point_R` et son enregistrement de méthodes `point_M`. Ainsi :

```
let point_R x = {x = x};;

let point_M R =
  let getx () = R.x in
  let print () = print_int (getx R) in
  {getx = getx; print = print};;

let new_point x =
  let R = point_R x in point_M R;;
```

Comme précédemment, ce codage assure l'encapsulation. Cependant, chaque objet sera créer avec un nouvel enregistrement de méthodes, puisque les méthodes contiennent l'état interne dans leur fermeture. Il est possible aussi d'écrire :

```
let point_M =
  let getx R () = R.x in
  let print R () = print_int (getx R) in
  {getx = getx; print = print};;

let new_point x = let R = point_R x in (R, point_M R);;
```

Mais l'encapsulation est perdue, puisque l'état interne est laissé visible. En fait, l'encapsulation peut aussi être effectuée par un mécanisme de typage. Aussi, nous utilisons ce second codage, plus souple. Pour plus de clarté, nous définissons :

```
type 'a objet = Objet of 'a;;
let new_point x = let R = point_R x in Objet (R, point_M R);;
```

Nous pouvons également exiger que l'utilisateur définisse les messages :

```
type 'a message = Message of 'a;;
let getx = Message (fun x → x.getx);;
let print = Message (fun x → x.print);;

let getc = Message (fun x → x.getx);;
let move = Message (fun x → x.print);;
```

L'envoi du message `m` à un objet `p` n'est plus `p.m` est noté `p⇒m`, et définit par :

```
let (⇒) (Objet (R,M)) (Message m) = m M R;;
```

les points colorés peuvent s'écrire de façon analogue.

```
let point_coloré_R (x, c) = {x = x; c = c};;

let point_coloré_M =
  let getx R () = R.x in
  let getc R () = R.c in
  let print R () = print_int (getx R); print_string (getc R) in
  {getx = getx; getc = getc; print = print};;

let new_point_coloré x c =
  let R = point_coloré_R (x, c) in point_coloré_M R;;
```

Nous pouvons envoyer le message `print` à la fois aux points et aux points colorés :

```
let p = new_point 1 in p ⇒ print;;
let q = new_point_coloré 1 "blanc" in p ⇒ print;;
```

Nous avons déjà le minimum pour pouvoir parler d'objets ; l'envoi polymorphe de messages.

Héritage

L'*héritage* est la possibilité de réutiliser les définitions d'anciens objets pour en construire de nouveaux avec un comportement différent, par modification de méthodes existantes ou ajout de nouvelles. Par exemple, les points colorés peuvent être définis à partir des points :

```
let point_coloré_R (x, c) = point_R x || {c = c};;
let point_coloré_M =
   let getc R () = R.c in
   let print R () = print_int (getx R); print_string (getc R) in
   point_coloré_M || {getc = getc; print = print};;
```

Nous obtenons évidemment le même objet. Nous dirons que les points colorés *héritent* des points. Remarquons que les variables d'instance sont hérités elles aussi. Nous pouvons aller plus loin dans la définition de `print` et la remplacer par :

```
let print R () = point_M.print (); print_string (getc R) in
```

Cette définition est préférable car une modification ultérieur du comportement des points entraînera automatiquement la même modification sur les points colorés.

Classes

Dans la pratique, les objets ne sont pas créés directement, mais à partir de classes, ou bien par duplication d'autres objets existants. Les classes sont donc avant tout des générateurs d'objets. Elle permettent également de créer d'autres classes, par héritage. En fait, ce sont simplement une façon de grouper ensemble toutes les définitions des variables d'instance et les définitions de méthodes :

```
let point_C = (point_M, point_R);;
```

On peut écrire une fonction `new` qui prend une classe en argument et des paramètres d'initialisation et retourne un nouvel objet ce cette classe.

```
let new (R, M) x = let state = R x in (r, M);;
let new_point = new point_C;;
```

On notera que tous les objets d'une même classe partagent effectivement le même enregistrement de méthodes.

De façon systématique, on pourra écrire :

```
type 'a class = Class of 'a;;
let point_C =
  let point_R = ... in
  let point_M = ... in
  Class (point_R, point_M);;
```

et avec héritage :

```
let point_coloré_C =
  let (Class (superR, superM)) = pointC in
  let point_coloré_R (x,c) = super_R x || {c = c} in
  let point_coloré_M =
    let getc R () = R.c in
    let print R () = super_M.print (); print_string (getc R) in
    super_M || {getc = getc; print = print} in
  Class (point_coloré_R, point_coloré_M);;
```

Nous disons que la classe `point_coloré` est une sous-classe de la classe `point`.

Dans les exemples ci-dessus, nous avons volontairement ignoré les points difficiles pour faire ressortir les notions les plus importantes. Toutefois le codage est relativement robuste, et quelques variations permettront de l'adapter pour rendre compte des notions plus délicates.

Exercice 52 *Testez (et corrigez) les exemples de cette partie en utilisant la simulation des enregistrements extensibles de la partie précédente.*

Ne soyez pas surpris par la longueur des expressions de types. On pourra séparer les étiquettes utilisées pour les variables d'instance de celles utilisées pour les méthodes, en utilisant deux types différents d'enregistrements. □

8.1.4 Les notions délicates

La programmation à objets serait extrêmement facile (et ce cours sans objet) s'il n'y avait pas la notion d'envoi de messages à soi-même. Cela augmente considérablement l'expressivité des langages à objets ; simultanément les programmes avec objets deviennent plus difficiles à écrire et à comprendre.

Pour pouvoir typer tous les programmes de cette partie, il faut autoriser les types récursifs implicites.

Self et liaison tardive

Dans la classe point, nous avons défini les méthodes par :

```
let point_M =
  let getx R  () = R.x in
  let print R () = print_int (getx R) in
  {getx = getx; print = print}
```

Par expansion de `getx`, cela est équivalent à :

```
let point_M =
  let getx R  () = R.x in
  let print R () = print_int R.x in
  {getx = getx; print = print}
```

Imaginons maintenant que l'on souhaite définir une sous-classe des points pour décaler l'origine. Nous voudrions que la redéfinition de la méthode `getx` dans les points suffise pour réaliser ce changement.

```
let point_décalé_C =
  let (Class (super_R, super_M)) = point_C in
  let point_décalé_R = super_R
  let point_décalé_M =
    let getx R () = R.x + 1 in
    super_M || {getx = getx}
  Class (super_R, point_décalé_M)
```

Mais cela ne produira pas l'effet attendu, car la méthode `print` continuera à utiliser l'ancienne méthode `getx`. Une solution consiste à redéfinir la méthode `print` également. Cependant, l'implémentation de la méthode `print` peut ne pas être accessible au moment de la définition de la classe `point_décalé_C`.

La solution consiste pour la méthode `print` à envoyer le message `getx` à l'objet appelant, habituellement lié à la variable `self`. Il faut donc abstraire les méthodes par rapport à la variable `self` qui à l'évaluation sera lié à l'objet appelant au moment de l'envoi de message. On modifie ainsi la définition de `point_M` dans la définition de la classe des points :

```
let point_C =
  let point_R x = {x = x} in
  let point_M =
    let getx  R self () = R.x in
    let print R self () = print_int (self ⇒ getx) in
    {getx = getx; print = print} in
  Class (point_R, point_M);;
```

Nous devons simultanément redéfinir l'envoi de message :

```
let (⇒) (Objet (état, méthodes) as self) (Message m) =
  m méthodes état self;;
```

Cependant l'envoi d'un message nécessite des types récursifs. En effet, une partie de l'objet la méthode sélectionnée est appliquée à l'objet tout entier. En ML, une récursion de type doit se faire au travers de types déclarés (c'est le cas du type des listes par exemple). Cependant si l'on effectue un déclaration de type pour cacher la récursion ici, nous rendrons l'envoi de message monomorphe. La solution consiste à autoriser les types récursifs, ce qui ne pose pas de problème en soi.

La nouvelle définition de la classe des points décalés :

```
let point_décalé_C =
  let (Class (super_R, super_M)) = point_C in
  let point_décalé_R = super_R
  let point_M =
    let getx R self () = R.x + 1 in
    super_M || {getx = getx} in
  Class (super_R, point_décalé_M)
```

produit enfin l'effet attendu.

L'envoi de message à `self` est d'un usage systématique. Il produit un effet de *liaison tardive*. En effet dans la définition de la méthode `print`, la méthode `getx` n'est plus égale à la valeur de celle définie conjointement, mais à la valeur de cette méthode dans l'objet lié à `self` à l'exécution. On parle aussi parfois, mais à tort[2], de liaison dynamique.

[2]La liaison dynamique a un autre sens, précis, signifiant que la valeur d'un identificateur à l'exécution est la dernière valeur liée à cet identificateur au moment de l'exécution ; c'est par op-

La liaison tardive pose des difficultés liées à son interaction avec l'héritage : le typage d'une classe parente doit faire des hypothèses sur le type de `self`. Mais dans une sous-classe `self` désigne des objets ayant des types différents dans la classe parente et dans la sous-classe.

Méthodes binaires

Une autre difficulté du style à objets sont les méthodes binaires. Typiquement elles apparaissent, dés que l'on veut comparer des objets entre eux. Par exemple, pour ajouter une méthode d'égalité aux points, on définira une nouvelle classe :

```
let point_comparable_C
  let (Class (super_R, super_M)) = point_C in
  let point_comparable_M =
    let equal R (self:'a) (y:'a) = (self⇒getx) = (y⇒getx) && R.c (y⇒getc)) in
    super_M || {equal = equal} in
  Class (super_R, point_comparable_C);;
```

Les méthodes binaires posent parfois des difficultés de typage, en particulier en présence de sous-typage. Nous reviendrons sur ce point délicat dans les deux chapitres suivants.

Une *méthode binaire* est une méthode qui reçoit parmi un (ou plusieurs) argument contenant un objet du même type que l'objet lié à la variable `self`.

Sélection multiple

Il ne faut pas confondre les méthodes binaires avec la sélection multiple, bien que les deux ne soient pas sans rapport. La *sélection multiple* (*multiple dispatch* en anglais) n'a de sens que pour une méthode binaire, et signifie que le choix de la méthode à appliquer à la réception du message dépend de plusieurs arguments à la fois.

La sélection multiple est une opération relativement naturelle dans le modèle à surcharge, mais plus difficile à réaliser dans le modèle à enregistrements.

Héritage Multiple

L'héritage simple permet d'étendre une définition de classe avec de nouvelles variables d'instance et de nouvelles méthodes. L'héritage multiple permet de construire une nouvelle classe à partir de plusieurs classes.

L'héritage multiple peut être réalisé de deux façons différentes. La plus naturelle repose sur la concaténation des enregistrements, mais cette opération est très difficile à typer.

Une autre approche consiste à abstraire les classes par rapport à une hypothétique classe parente. On dit quelque fois que les classes sont traitées comme des des *wrappers*

---

position à la liaison statique qui signifie que la valeur d'un identificateur à l'exécution est celle liée dans le contexte d'utilisation de l'identificateur (fonctions d'emballage). Plus précisément, cela revient à définir la classe des points comme suit :

```
let point_C =
  let point_R super_R v = super_R || {x = ref v} in
  let point_M super_M =
    let getx R self = ...
    ...
    in super_M || {getx = getx; setx = setx; print = print}
  in Class (point_R, point_M);;
```

On peut alors définir un *wrapper* qui ajoute la couleur à une classe arbitraire.

```
let coloré_C =
  let coloré_R super_R v = super_R || {c = v} in
  let point_M super_M =
    let getc R self = ...
    let print R self = super_M. print; ...
    in super_M || {getc = getc; print = print}
  in Class (point_R, point_M)
```

L'héritage de deux classes se fait de façon systématique

```
let inherit (Class (R1, M1)) (Class (R2, M2)) =
  let R super_R (x1,x2) = R2 (R1 super_R x1) x2 in
  let M super_M = M2 (M1 super_M) in
  Class (R, M);;
```

Ainsi, nous obtenons la classe des points colorés :

```
let point_coloré_C = inherit point_C coloré_C;;
```

8.1.5 Typage et sous-typage : une nécessité, des difficultés

Pour conclure ce chapitre, nous voulons insister sur le typage. Le concept d'objet est une notion relativement simple, mais elle se trouve compliquée par la combinaison de la liaison tardive et de l'héritage, qui restent deux aspects essentiels de la programmation à objets, et qu'il n'est pas question d'interdire.

De façon générale, le typage apporte énormément à la sécurité, et la rapidité de la programmation. Cet aspect est encore plus essentiel pour les langages à objets, étant donné la difficulté des constructions. Plusieurs langages à objets typés, utilisés dans l'industrie, ont été erronés pendant plusieurs années...

Un concept que nous n'avons pas encore évoqué est la possibilité pour un objet d'une sous-classe d'être utilisé comme s'il appartenait à une classe parente. En effet, il semble raisonnable de pouvoir utiliser un point coloré là où un point simple est attendu, puisqu'un point coloré implémente l'interface des points et même plus ! Mais un raisonnement trop hâtif pourrait conduire à des erreurs. Par exemple, les points colorés avec égalité ne peuvent pas être considérés comme des points simples. En effet, la méthode `equal` des points colorés exige un argument qui soit un point coloré. C'est-à-dire que son résultat risque d'être erroné si son argument n'est pas au moins un point coloré. Considérer un point coloré comme un point reviendrait à dire que la méthode `equal` des points colorés peut également prendre comme premier argument un point. Ce qui est contradictoire avec l'assertion précédente. Nous étudierons plus en détail ces problèmes dans les deux chapitres suivants.

Conclusion

En conclusion, les exemples que nous avons présentés ci-dessus souffrent de l'absence de syntaxe concrète pour les objets. C'est le problème des codages en général, compensé par l'avantage, pédagogique, d'expliquer de nouvelles constructions en termes de constructions plus simples.

La possibilité de coder les objets en ML confirme la robustesse de ML. Elle infirme aussi l'idée répandue que les objets sont des constructions trop compliquées pour être utilisables. Malgré tout, la programmation avec des objets reste un paradigme délicat d'emploi ; Le "tout objet" est à proscrire. Mais avec un peu de rigueur, et de modération, les objets sont facilement abordables et utilisables.

8.2 Objets et classes dans le langage Ocaml

Nous présentons le langage Ocaml, en reprenant chacune des notions introduites ci-dessus et en montrant sur des exemples comment les exprimer dans le langage Ocaml.

8.2.1 Présentation du langage Ocaml

Objective Caml est un *langage à classes*[3]. Les objets ne sont pas créés directement, mais à partir de classes ou par modification d'objets existants. En particulier, les objets créés par modification d'autres objets partagent les mêmes méthodes. Un objet a donc une classe associée, celle à partir de laquelle il a été créé ou celle de l'objet à partir duquel il a été dupliqué. Cette propriété est importante. Comme tout invariant, elle permet d'analyser plus facilement les programmes, mais aussi de mieux les compiler.

Toutefois, les objets existent en temps que valeurs. Il est naturel et plus simple d'autoriser leur création directe. Nous choisirons cette approche pour l'exposition, et nous rendrons la construction correspondante inaccessible à l'utilisateur.

Classes et objets

Un objet est composé d'un ensemble de variables d'instance et d'un ensemble de méthodes. Par exemple, un point sur la droite peut être défini par une variable d'instance `x` et deux méthodes `getx` et `print` :

[3]En anglais on dit aussi *class-based language* par opposition à *object-based language*

```
let p = object val x = 1 method getx = x method print = print_int x end;;
```

Son type est :

```
p : < getx : int; print : unit >
```

L'objet `p` est identique a celui que l'on aurait obtenu à partir d'une définition de classe :

```
class point x =
  val x = x
  method getx = x
  method print = print_int x
end;;
```

La définition de la classe `point` a trois effets sur l'environnement :

- ajouter la définition à la variable de classe `point` dans l'environnement des classes.
- ajouter une valeur `new point` qui permet de créer des objets de la classe correspondante.
- ajouter des abréviations de type `point` et `#point`. La première est égale au type des points `<getx : int; setx : int → unit; print : unit>`. La seconde est plus générale et s'unifie avec les types de toutes les sous-classes de `point`.

Vérifions que l'on peut créer de nouveaux points et leur envoyer des messages :

```
let p = new point 1;;
p#print;;
```

L'envoi de message est polymorphe, comme le montre l'exemple suivant :

```
fun p → p#print;;
```

Le type de cette fonction indique que l'argument doit être un objet qui possède une méthode `print` de type `'a` où `'a` est la valeur retournée par le message.

Héritage

Les points colorés peuvent être définis à partir des points par héritage :

```
class point_coloré x c =
  inherit point x as parent
  val color = c
  method getc = color
  method print = parent#print; print_string color
end;;
```

L'héritage a le même effet que de recopier dans la classe définie toutes les définitions de la classe héritée. La classe parente peut être liée à une variable comme `parent` dans l'exemple ci-dessus. Une variable liant une classe parente n'est pas une variable ordinaire ; elle ne peut être utilisée que pour envoyer des messages (comme dans `parent#print`). L'envoi d'un message m_1 à une classe parente P par une méthode m_2 définie dans une classe C revient à appliquer à l'objet exécutant la méthode m_2 la méthode m_1 définie dans la classe P, au lieu de sa valeur prise dans l'objet exécutant m_2.

L'héritage peut aussi être multiple. Par exemple, définissons une classe `couleur` :

```
class couleur c =
  val color = c
  method getc = color
  method print = print_string color
end;;
```

Les points colorés peuvent alors être défini par héritage multiple des classes `point` et `couleur`.

```
class point_coloré x c =
  inherit point x as point
  inherit couleur c as couleur
  method print = point#print; couleur#print
end;;
```

Les deux classes sont bien sûr équivalentes.

```
let p = new point_coloré 2 "blanc_cassé" in p#print;;
```

Auto-référence

L'auto-référence est la possibilité pour une méthode de référencer l'objet qui a provoqué l'exécution de cette méthode. Cet objet est en général lié à la variable `self` ; en Objective Caml la liaison de cet objet à une variable doit être explicite et utilise la construction *as*. L'auto-référence est le plus souvent utilisée pour envoyer d'autres messages à `self` ; en quelque sorte, la méthode exécutée délègue une partie du calcul à d'autres méthodes. Par exemple, on aurait pu définir la classe `point` ainsi :

```
class point x as moi_même =
  val x = 1
  method getx = x
  method print = print_int moi_même#getx
end;;
```

Les points créés par cette classe auront le même comportement que ceux créés par l'ancienne classe. Toutefois, la nouvelle classe est différente. En particulier, si une autre classe en hérite et puis modifie la définition de la méthode `getx`, alors la méthode `print` subit l'effet de cette modification, alors qu'elle resterait inchangée si la classe héritait de l'ancienne définition. Cet effet lié à l'envoi de message à l'objet auto-référencé est appelé *liaison tardive*.

Un autre usage de l'auto-référence est la possibilité de retourner l'objet lui-même en résultat. On peut en effet écrire une classe :

```
class moi_même() as moi_même =
  method moi_même = moi_même
end;;
```

Exercice 53 *Que retourne le programme suivant*

```
let p = new moi_même () in p # moi_même # moi_même;;
```

□

Objective Caml fournit une construction pour retourner une copie de l'objet dans laquelle certaines variables d'instance ont été modifiées. Par exemple, un entier peut être représenté par :

```
class entier () =
  val n = 0
  method nul () =  (n = 0)
  method add p = {< n = n + p >}
end;;
```

A chaque appel, la méthode `add` crée un nouvel entier dont la variable d'instance n est incrémenté de un. Il y a donc deux façons de créer des objets. La façon habituelle consiste à demander un nouvel objet de la classe

```
let p = new point 1;;
```

Cette solution est générale et s'applique toujours possible. On peut également créer un objet de la classe C par reproduction d'un objet déjà construit de la classe C à condition que la classe C ait elle-même définit une méthode de reproduction. Par exemple, on peut étendre la classe point avec une méthode de reproduction :

```
class point_neuf x =
  inherit point x
  method neuf y = {< x = y >}
end;;
let un_point = new point_neuf 0;;
```

Il est alors équivalent d'écrire `new point` ou `un_point.neuf`. La création d'un objet par reproduction n'est intéressante que lorsque, comme dans l'exemple des entiers, la reproduction initialise les variables d'instance à des valeurs dépendant de leur ancienne valeur.

Style impératif

Jusqu'à maintenant, nous avons défini des objets en style purement applicatif. Il est bien sûr possible de programmer avec des objets dans un style impératif. Évidemment, il est possible d'utiliser des références à l'intérieur des objets.

```
class point_mutable x as moi =
  val x = ref x
  method getx = !x
  method setx y = x := y
  method print = print_int moi#getx
end;;
```

Toutefois, Objective Caml fournit la possibilité d'annoter les variables d'instance comme mutables. On peut alors écrire la définition plus simple :

```
class point_mutable x as moi =
  val mutable x = x
  method getx = x
  method setx y = x ← y
  method print = print_int moi#getx
end;;
```

De façon plus concise, on aurait aussi pu étendre et modifier la classe `point` :

```
class point_mutable x =
  inherit point x
  val mutable x
  method setx y = x ← y
end;;
```

Il existe un module `Oo` qui fournit une fonction `copy` qui permet de dupliquer un objet : `Oo.copy p` retourne une copie superficielle de l'objet `p`. Les variables mutables ont les mêmes valeurs que celle de l'objet original, mais elles seront mutées indépendemment.

```
Oo.copy;;
```

Cette fonction peut-être utilisée pour dupliquer un objet :

Exercice 54 *Écrire une classe* `copie` *qui fournit une méthode* `copy` *qui duplique l'objet à qui elle est envoyée. Quand la méthode* `copy` *est-elle intéressante ?* □

Exercice 55 *La fonction* `Oo.copy` *n'effectue qu'une copie superficielle de l'objet. Comment peut-on effectuer une copie profonde ? Donner un exemple.* □

Exercice 56 *Il existe deux façons de représenter les points en style applicatif et en style impératif. Les comparer.* □

Exercice 57 *Pour les points écrits en style impératif, il existe deux façons de dupliquer un objet. Les comparer.* □

Classes paramétriques

Les références peuvent aussi être implémentées par une classe. Toutefois, le programme suivant échoue :

```
class ref x =
  val mutable x = x
  method get = x
  method set y = x ← y
end;;
```

En effet, les méthodes sont polymorphes, et la classe `ref` est naturellement paramétrique. Il est bien sûr possible d'écrire une version monomorphe :

```
class ref x =
  val mutable x = (x:int)
  method get = x
  method set y = x ← y
end;;
```

Pour écrire une classe paramétrique, il est nécessaire d'indiquer explicitement les variables de types dont elle dépend. Il faut également que ces variables soient liées dans le corps de la définition :

```
class 'a ref x =
  val mutable x = (x:'a)
  method get = x
  method set y = x ← y
end;;
```

Un paramètre de type peut être contraint à un type plus précis dans le programme. Cette contrainte est alors indiquée par une clause `constraint`.

```
class 'a ref (x:'a) =
  val mutable x = x + 1
  method get = x method set y = x ← y
end;;
```

Une contrainte peut également être mise explicitement :

```
class 'a ref (x:'a) =
  constraint 'a = 'b list
  val mutable x = x
  method get = x method set y = x ← y
end;;
```

Coercion

Il est parfois possible de coercer un objet vers un autre objet. Par exemple un point coloré peut aussi être considéré comme un point. On écrit cela

```
let p = new point 1 and q = new point_coloré 2 "blanc_cassé";;
let pq = [p ; (q :point_coloré :> point)];;
```

La coercion d'une expression a de type τ_1 vers le type τ_2 est possible si l'expression a peut être considéré de type τ_2 en toute sécurité. Nous verrons dans le chapitre suivant qu'il en est ainsi lorsque les types τ_1 et τ_2 sont dans une relation de sous-typage que nous définirons formellement.

Dans bien des cas, il est commode de ne pas indiquer seulement le type du domaine de la coercion. Pour cela un ensemble de primitives ($\_ :> \tau$) sont fournies, par exemple :

```
fun x → (x :> point);;
```

Cependant, le type du domaine est calculé à partir du type du co-domaine, et il n'est pas vrai que ($q :> t$) soit toujours bien typé si ($q : \tau' :> \tau$) pour un certain τ' la seconde forme étant plus générale.

Cela ne doit pas être considéré comme une coercions, mais comme une primitive avec un type fixé et la construction ($q :> \tau$) doit être considéré comme l'application de cette fonction à q. Dans le cas présent, il se trouve que cette application est bien typé si q est un point coloré.

La coercion ne modifie pas la valeur coercée. En particulier, un point coloré peut être considéré comme un point, mais l'envoi d'un message à un point coloré vu comme un point aura le même effet que l'envoi du même message au même objet vu comme un point coloré.

La coercion de type est une perte d'information. Une fois cette information oubliée, il n'est plus possible de revenir en arrière. Par exemple un point coloré vu comme un point ne plus plus être considéré comme un point coloré à nouveau, sauf à avoir gardé un pointeur sur le même objet sans perte d'information, comme le montre le programme suivant.

```
let q' = (q :> point) in (q, q');;
```

Évidemment ! puisque la liaison de q est statique.

Lorsqu'il est nécessaire de réifier ultérieurement un objet vers sa véritable classe, alors on devra utiliser des types concrets :

```
type any_point = Point of point | Point_coloré of point_coloré;;
let pq = [Point p; Point_coloré  q];;
let send_any f = function Point p → f p | Point_coloré q → f (q:>point);;
List.iter (send_any (fun x → x#print)) pq;;
```

Méthodes virtuelles

Ils est possible de déclarer une méthode d'une classe sans la définir, ni l'hériter. Elle est alors dite virtuelle. Une méthode utilisée mais non déclarée ni héritée est implicitement virtuelle. Une classe qui contient des méthodes virtuelle est elle même virtuelle ; elle doit être annotée par le mot clef `virtual`, et il n'est pas possible d'en prendre d'instance. Les méthodes virtuelles devront être définies dans une sous-classe.

Variables d'instance privées

Il est possible d'annoter les variables d'instance par la propriété `private`. Cela a pour effet de les rendre locale à la définition de la classe et donc invisibles dans le type de la classe, et a fortiori dans les les sous-classes.

8.2.2 Génération automatique de types

Les définitions de classe génèrent de abréviations de types automatiquement. Ces abréviations utilisent le partage créé dans les types par unification pendant le typage de la classe pour déterminer l'abréviation correspondante à la classe, en particulier pour séparer les occurrences de `self` et des paramètres d'autres types qui pourraient leur être accidentellement égaux.

8.3 Les Objets : encapsulation, récursion, sous-typage

Introduction

Dans les parties précédentes, nous avons expliqué comment simuler les objets dans le langage Caml. Nous avons justifié cette approche par son aspect pédagogique : elle explique les objets en termes de constructions plus élémentaires. Nous avons expliqué que l'encapsulation pouvait être assurée par fermeture, ou par abstraction de type. Nous avons ensuite abandonner le codage de l'abstraction par fermeture, qui rend certaines opérations plus difficiles, notamment la reproduction des objets, et qui n'induit pas un partage de l'enregistrement des méthodes d'une même classe. Nous avons donc choisi l'autre alternative qui s'appuyant sur une encapsulation par abstraction de type, sans toutefois réaliser cette abstraction. Il nous reste donc à expliquer en détail comment réaliser l'abstraction par un mécanisme de typage.

Dans la section précédente, nous avons présenté un langage avec des constructions primitives pour les opérations sur les objets. Nous avons modélisé l'encapsulation par abstraction. S'agit-il d'une encapsulation par fermeture ou d'une encapsulation par abstraction de type ?

Nous avons également observé que le recours aux types récursifs était inévitable : il permettent qu'une méthode retourne `self` ou le prenne en argument. Il faudra donc également revenir sur les types récursifs et expliquer leur traitement.

Enfin nous avons mentionné la possibilité de considérer parfois les objets d'une sous-classe comme des objets de leur classe parente par un mécanisme de sous-typage. Cela est-il toujours possible ? Nous verrons dans la partir suivante que la réponse est négative. Il faudra pour expliquer cela définir le sous-typage plus formellement.

Cette section est plus particulièrement consacrée aux objets ; nous donnerons un sens plus précis à ces trois aspects nouveaux : encapsulation, récursion et sous-typage. En particulier, il sera nécessaire de comprendre leur interaction pour découvrir ce que sont réellement les objets.

8.3.1 Un contre exemple

Nous décrivons brièvement un contre-exemple à la coercion des objets d'une sous-classe vers une classe parente. Ceci devrait justifier une étude approfondie des objets qui sera réalisée dans le reste de ce chapitre. Considérons les points colorés mutables :

```
class point x =
  val mutable x = (x:int)
  method getx = x
end;;

let point = new point 1;;
class color c =
  val c = (c : string)
  method getc = c
end;;

class point_coloré x c =
  inherit point x
  inherit color c
end;;

let point_coloré = new point_coloré 2 "blanc-cassé";;
```

Les points colorés peuvent être considérés comme des points. Ajoutons une méthode qui teste l'égalité de deux points.

```
class point_physique x as self : 'a =
  inherit point x
  method equal (y:'a) = self#getx = y#getx
end;;

let point_physique = new point_physique 3;;
```

Sur les points colorés l'égalité doit aussi tenir compte de la couleur :

```
class point_coloré_physique x c as self =
  inherit point_physique x as pp
  inherit color c
  method equal y = pp#equal y & self#getc = y#getc
end;;
```

```
let point_coloré_physique = new point_coloré_physique 4;;
```

Les points colorés avec égalité peuvent-ils être considérés à la fois comme des points colorés ou comme des points avec égalité ? La première réponse est affirmative, mais la seconde est négative.

En effet, considérons un point coloré p avec égalité. Si nous considérons p comme un point simple avec égalité, il posséderait la méthode `equal` qui reçoit un point avec égalité en argument et retourne un booléen. Il serait donc possible d'envoyer à p le message `equal` en passant comme argument un point avec égalité non coloré q. Cependant l'exécution de la méthode `equal` de p va envoyer le message `getc` à q qui ne pourra pas le recevoir. Par contre, p peut être considéré comme un point coloré sans égalité en toute sécurité : on peut s'en convaincre en essayant d'envoyer tous les messages des point colorés à p.

Pour savoir si un objet d'une classe peut être considéré comme un objet un objet d'une autre classe, il ne suffit donc pas de regarder la liste des méthodes auxquels il répond mais également au type de ses arguments, en particulier s'il reçoit un autre objet du même type que lui-même parmi ses arguments, la coercion n'est pas sure. Nous pouvons résumer les coercions possibles entre les différentes sortes de points dans la table suivante :

	`point`	`_coloré`	`_physique`	`_coloré_physique`
`point`	X			
`_coloré`	X	X	b	
`_physique`	X	X	X	
`_coloré_physique`	X	X	?	X

Exercice 58 *Dans l'exemple ci-dessus, la méthode d'égalité est différente pour les points colorés et pour les point simples. Il n'est donc pas trop surprenant que la coercion soit impossible.*

1. *Donner un exemple où il n'y a pas forcément d'erreur à l'exécution, mais qui soit indistingable par le typage d'un exemple qui produit une*

   erreur à l'exécution.

2. *Donner un exemple où bien que la méthode soit inchangée, la coercion produirait encore une erreur à l'exécution.*

□

Exercice 59 *A partir des points, des couleurs et points colorés, définir des cercles puis des cercles colorés. Vérifiez que certaines coercions sont possibles.*

Ajouter aux cercles une méthode qui en reçoit un autre cercle q en argument et construit un nouveau cercle concentrique à `self` *et circonscrit à q. Le nouveau cercle sera coloré seulement si les deux cercles* `self` *et q le sont. Quelles sont les coercions possibles ?* □

8.3.2 Les objets, formellement

Extension de la syntaxe pour les objets

Comme nous l'avons mentionné dans l'introduction, bien que l'implémentation Objective Caml ne permette pas à l'utilisateur de construire directement des objets, ces constructions existent dans le langage.

Nous étendons la syntaxe des expressions avec deux nouvelles constructions :

$$a ::= \ldots a\#m \mid \varsigma(x)\langle \texttt{meth}\ m_i = a_i{}^{i\in I}\rangle$$

Nous avons choisi une forme plus simple des objets sans variable d'instance.

Il nous faut également enrichir les types avec des types objets similaires aux types enregistrements.

$$\begin{aligned}\tau &::= \ldots \mid \langle\rho\rangle \mid \tau\ \texttt{as}\ \alpha\\ \rho &::= \alpha \mid \emptyset \mid (m : \tau; \rho)\\ \varphi &::= \alpha \mid \texttt{Pre}\ \tau \mid \emptyset\end{aligned}$$

Les types d'objets qui se terminent avec une variable de rangée sont dits *ouverts*, alors que les autres sont dits *fermés*.

Dans Ocaml, on impose de plus que

1. deux types objets qui se terminent par la même variable de rangée soient égaux.

2. un type objet qui contient un champ absent est fermé

Les primitives définies ci-dessous satisfont ces conditions. Les conditions resteront alors préservées par toutes les opérations. Cette restriction permet de ne jamais montrer les variables de rangées à l'utilisateur sinon par une ellipse, ni les champs absents par type objet fermé. Par exemple on écrira $(\langle m : \tau; ..\rangle\ \texttt{as}\ \alpha) \to \alpha$ pour $\langle m : \texttt{Pre}\ \tau; \alpha\rangle \to \langle m : \texttt{Pre}\ \tau; \alpha\rangle$.

Les alias τ as α permettent d'exprimer des types récursifs, indispensables pour les objets. Par exemple $(\mu\alpha.\tau_1) \to \tau_2$ peut s'écrire $(\tau_1\ \texttt{as}\ \alpha) \to \tau_2$. Il sont aussi plus généraux car α est liée à τ_1 dans τ_1 et τ_2. Ils permettent d'exprimer le partage des variables de rangées, et plus généralement le partage des expressions de types qu'il est important de conservé pour le bon fonctionnement des déclarations automatiques d'abréviations (partie 8.2.2)

Typage des objets

On ajoute les deux règles de typage :

$$\text{(OBJECT)}\quad \frac{A \oplus x : \tau \vdash a_i : \tau_i{}^{i\in I} \qquad \tau \equiv \langle i : \texttt{Pre}\ \tau_i{}^{i\in I}; \emptyset\rangle}{A \vdash \varsigma(x)\langle \texttt{meth}\ m_i = a_i{}^{i\in I}\rangle : \tau} \qquad\qquad \text{(SEND)}\quad \frac{A \vdash a : \langle m : \texttt{Pre}\ \tau; \tau'\rangle}{A \vdash a\#m : \tau}$$

Sémantique

Dans ce cas simplifié où il n'y a pas de variables d'instance, les objets sont des tuples de méthodes donc des valeurs. Il suffit d'ajouter la règle de réduction suivante.

$$\varsigma(x)\langle \texttt{meth}\ m_i = a_i{}^{i\in I}\rangle\#m_j \longrightarrow a_j[\varsigma(x)\langle m_i = a_i{}^{i\in I}\rangle/x] \qquad j \in i$$

Types récursifs

Que les objets soient simulés dans un langage avec enregistrements polymorphes ou primitifs comme dans Objective Caml, il est nécessaire que les types soient récursifs. Sans cela, il ne serait pas possible pour une méthode d'un objet p de retourner ou de prendre en argument un autre object de même type que p. Cela interdirait en particulier la duplication des objets (function `Oo.copy` en Ocaml) et les méthodes binaires qui sont deux aspects importants de la programmation à objets.

Sous-typage

L'utilisation de variables de rangée permet l'envoi de message de façon polymorphe. C'est l'essence des langages à objets. Cette possibilité est en générale réalisée par du sous-typage dans les autres langages (Java par exemple). Le sous-typage devient donc secondaire en Ocaml. Il reste cependant très utile, par exemple pour permettre à des objets ayant différentes interfaces devenir interchangeable, à condition de ne retenir que la partie de leur interface qui est communes à tous.

La solution que nous choisissons est de fournir du sous-typage explicite, sans contraintes. Cela peut se faire en supposant donné une collection de primitives $(\_ : \tau_1 \triangleright \tau_2)$ pour tous les paires (τ_1, τ_2) telles que $\tau_1 \leq \tau_2$, avec comme δ règles associée :

$$(v : \tau_1 :> \tau_2) \longrightarrow v$$

Attention ! cette règles perd l'information explicitement fournie. Il faut donc pour conserver l'auto-réduction autoriser une règle implicite de sous-typage

$$\text{(Sub)}\quad \frac{A \vdash (a : \tau_1 :> \tau_2) : \tau}{A \vdash a : \tau}$$

Ce qui détruit les dérivation principales... En résumé, on a des types principaux a condition de ne pas utiliser la règle Sub. On a l'auto-réduction à condition d'utiliser la règle Sub.

Une autre solution, mains élégante consiste à remplacer la règle ci-dessus des règles de la forme :

$$(\texttt{fun}\ (x)\ a : \tau_1 \to \tau_2 :> \tau_1' \to \tau_2')v \longrightarrow (\texttt{fun}\ (x)\ (a : \tau_2 :> \tau_2'))\ (v : \tau_1' :> \tau_1)$$

pour chaque autre rédex.

L'opération de coercion de base est entièrement explicite ; elle exige que le domaine τ_1 et le co-domaine τ_2 de la coercion soient à la fois précisés.

Coercions semi-explicite

En fait la plupart des coercions $(\_ : \tau_1 :> \tau_2)$ peuvent s'exprimer avec du polymorphisme des variables de rangées. On peut ajouter une collection de coercions semi-explicites $(\_ :> \tau)$ qui sont des abréviations de $(\_ : \mathcal{S}(t) :> \tau)$ où l'opérateur S est choisi tel que :

$$S(\tau) \leq \tau \qquad\qquad \exists\theta(\theta(S(\tau)) = \tau \wedge \theta(\tau) = \tau)$$

Il se trouve que si τ n'est pas récursif, il existe une seule solution satisfaisant ces deux conditions. Elle est obtenue en remplaçant dans τ tous les objets fermés apparaissant uniquement en position covariante par leur types ouverts correspondants.

Ce n'est en général plus le cas pour des types récursifs. Par exemple, pour $\tau \equiv \langle x : \alpha\rangle$ `as` α que on peut choisir pour $S(\tau)$ égal à $\langle x : \tau; \rho\rangle$ ou bien $\langle x : \alpha; \rho\rangle$ `as` α qui mais il n'existe pas de solution plus générale que ces deux types à la fois.

Si α apparaît négativement dans α, alors $S(\texttt{rec}\ \alpha.\tau)$ est égal à $\texttt{rec}\ \alpha.\tau$. Sinon, $S(\texttt{rec}\ \alpha.\tau)$ est égal à $S(\tau[\texttt{rec}\ \alpha.S(\tau)/\alpha^k])$. Par exemple, pour l'exemple précédent, on prendra pour $\tau \equiv \langle x : \texttt{rec}\ \alpha_1.\langle x : \alpha_1, \rho_1\rangle; \rho\rangle$.

Exercice 60 *Décrire tous les types que l'on peut choisir pour $S(\tau)$ pour le dernier exemple ci-dessus.* □

La relation de sous-typage est standard. On pourra lire [AC93]. L'algorithme simple donné ci-dessus est décrit dans [KP93]

8.3.3 Les objets en général

Encapsulation : types existentiels

L'encapsulation peut être effectuée par une opération d'abstraction de type. Nous illustrons d'abord cette notion sur des fermetures.

Une requête de service est une paire (f, v) composée d'une fonction indiquant l'opération à effectuer et d'une valeur v argument de la fonction. Supposons que l'on ne veuille pas effectuer les requêtes immédiatement, mais les faire traiter plus tard par une ou plusieurs fonctions $g_1, \ldots g_k$. Nous pouvons ranger les requêtes dans une liste. Il est raisonnable alors que les requêtes retournent toutes la même valeur, par exemple `int`. Il est nécessaire également que pour chaque requête, l'argument puisse être passé à fonction, mais il ne devrait pas être nécessaire de contraindre tous les arguments à être de même type. On peut résoudre le problème en traitant les requêtes comme des fermetures :

```
type 'a requête = Requête of (unit → 'a);;
let requête f x = Requête (fun () → f x);;
let apply (Requête f) = f();;
```

Une autre façon plus naturelle de résoudre le problème, serait de mettre simplement f et x côte à côte.

```
type ('a,'b) requête = Requête of ('a → 'b) * 'a;;
let requête f x = Requête (f, x);;
let apply (Requête (f,x)) = f x;;
```

Toutefois, cette solution révèle le type de l'argument que l'on justement veut cacher dans les requêtes. Ne pas écrire le paramètre `'a` dans la définition du type des requêtes échoue :

```
type 'b requête = Requête of ('a → 'b) * 'a;;
```

La solution à ce problème consiste à introduire des types existentiels :

```
type 'a requête = Requête of Exist ('b) ('a → 'b) * 'a;;
```

Cette déclaration dit qu'un élément du type `'a requête` construit avec le constructeur `Requête` est une paire (f, v) telle qu'il existe un type `'b` tel que `f : 'b → 'a` et `v : 'b`.

On peut voir le constructeur `Requête` comme une coercion explicite du type $(\alpha_1 \rightarrow \alpha_2) \times \alpha_1$ vers le type existentiel $\exists\, \alpha_2.\, (\alpha_1 \rightarrow \alpha_2) \times \alpha_1$ abrégé par α `requête`. Le destructeur étant la coercion inverse. Pour se passer de la définition préalable des types existentiels, nous notons $\texttt{pack}_{\exists\, \alpha.\, \tau}\ a$ au lieu du constructeur `Requête` a et $\texttt{open}_{\exists\, \alpha.\, \tau}\ a_1$ `as` x `in` a_2 au lieu de `let` $x =$ `Requête` a_1 `in` a_2. La règle de typage associée est :

$$\text{(Pack)}\quad \frac{A \vdash a : \tau}{A \vdash \texttt{pack}_{\exists\, \alpha.\, \tau}\ a : \exists\, \alpha.\, \tau}$$

$$\text{(Open)}\quad \frac{A \vdash a_1 : \exists\, \alpha.\, \tau \qquad A \oplus \omega \oplus (x : \tau[\omega/\alpha]) \vdash a_2 : \tau_2 \qquad \omega \not{\!\!in} \tau_2}{A \vdash \texttt{open}_{\exists\, \alpha.\, \tau}\ a_1\ \texttt{as}\ x\ \texttt{in}\ a_2 : \tau_2}$$

Dans la règle Open, la lettre ω désigne une constante anonyme. Elle ne peut apparaître dans la typage de a_2 mais elle ne doit pas apparaître dans le type résultant τ_2.

Qu'est-ce qu'un objet ?

L'interface d'un objet est un type record $I(Y)$ représentant les méthodes de l'objet, et pour chaque méthode, le types des arguments supplémentaires à passer à l'invocation de cette méthode ainsi que le type du résultat. Cette interface dépend du type Y de `self`.

Sans encapsulation, un objet est une paire (r, m) d'un état interne et d'un vecteur de méthodes m. Supposons pour simplifier que l'objet ne possède qu'une seule méthode a, i.e. représentée par une fonction `fun` (y) `fun` (r) a. Si R est le type de r, et Y le type de `self`, alors m est de type $R \rightarrow Y \rightarrow I(Y)$. Sans encapsulation, le type d'un objet serait de la forme $R \times (R \rightarrow Y \rightarrow I(Y))$. Nous pouvons cacher l'état interne par :

$$\texttt{pack}\ (r, m) : \exists\, R.\, R \times (R \rightarrow Y \rightarrow I(Y))$$

Cependant il faut aussi identifier `self` dans la méthode m avec l'objet lui-même :

```
let rec p = Pack (r, m p) in p
```

Nous obtenons le type :

$$\texttt{rec}\ Y.\exists\, R.\, R \times (R \to I(Y))$$

Concrètement, on peut voir l'objet $\langle \texttt{var}\ x_i = v_i, \texttt{meth}\ m_j = a_j \rangle$ comme une abréviation pour

$$\texttt{let rec}\ p = \texttt{pack}_{\texttt{rec}Y.\exists\, R.\, R\times\{m_j : R \to \tau_j[Y]\}}\ (x_i = v_i, m_j = a_j\ p)\ \texttt{in}\ p$$

L'envoi de message $(\_\#m)$ peut-être défini comme :

$$\texttt{fun}\ (p)\ \texttt{open}_{\texttt{rec}Y.\exists\, R.\, R\times\{m_j : R \to \tau_j ; ..\}}\ p\ \texttt{as}\ (r, m)\ \texttt{in}\ m.m_j\ r$$

Il serait possible de retrouver la règle de sous-typage du constructeur objet à partir des règles de sous-typage pour les enregistrements polymorphes et pour les types existentiels.

8.4 Les Classes

Introduction

8.4.1 Les classes en Objective Caml

8.4.2 Pourquoi des classes ?

Les classes sont des générateurs d'objets. Elles permettent de créer de nombreuses instances différentes d'objets partageant le même comportement. Elles permettent également de construire de nouvelles classes donc ultérieurement de nouveaux objets par héritage.

Cependant, si nous ignorons le problème de partage de code, nous pouvons voir l'aspect générateur simplement en créant de multiples objets à partir d'une même fonction.

$$\texttt{fun}\ (x)\ \varsigma(y)\langle m_1 = x; m_2 = f\ y\#m_1 \rangle$$

Le second problème lui est plus important.

Nous pourrions nous passer des classes si nous avions en contrepartie la possibilité d'étendre les objets avec de nouvelles méthodes. Oublions le sous-typage dans un premier temps.

Supposons que nous ayons déjà créé un object p de type $\langle m_i : \texttt{Pre}\ {\tau_i}^{i\in I}; \emptyset \rangle$ et que nous voulions étendre le comportement de p avec une nouvelle méthode m dont le corps est a. Nous savons par le typage que p est de la forme $\varsigma(y)\langle m_i : {a_i}^{i\in I} \rangle$. Cependant nous ne pouvons pas utiliser le typage de p pour en déduire que le nouvel objet $\varsigma(y)\langle p\ \texttt{with}\ m = a \rangle$ est bien typé.

Cela n'est malheureusement pas possible. En effet les méthodes de l'objet p on été typées en supposant que y avec le type $\langle m_i : \texttt{Pre}\ {\tau_i}^{i\in I}; \emptyset \rangle$ alors que dans le nouvel les méthodes doivent pouvoir être typées en supposant que y est de type $\langle m_i : \texttt{Pre}\ {\tau_i}^{i\in I}; m : \texttt{Pre}\ \tau; \emptyset\rangle$ pour un certain type τ. Nous saurions le montrer pour la méthode m car nous en connaissons la définition a, mais nous ne savons par le faire pour les méthodes m_i sans connaître précisément les définitions a_i.

Il aurait fallu que les méthodes m_i aient été typée de façon polymorphes par rapport aux extensions possibles de l'objet. C'est ce que nous allons faire dans les classes.

Syntaxe des classes

$$a ::= \ldots \mid \texttt{class}\ (x)\ \langle\rangle \mid \texttt{class}\ (x)\ \langle \texttt{inherit}\ a\ \texttt{with}\ m = a\rangle \mid \texttt{new}\ a \quad \text{Expressions}$$

La constante $\emptyset$ est une classe vide. La contruction suivante ajoute une méthode à une classe. La construction `new` permet d'instancier une classe en un objet.

Nous ajoutons des types de classes.

$$\tau ::= \ldots \mid \texttt{class}\ (\tau)\ \tau$$

Règles de typage

$$\text{(Nil)}\quad \frac{}{A \vdash \texttt{class}\ (x)\ \langle\rangle : \texttt{class}\ (\tau)\ \langle\emptyset\rangle}$$

$$\text{(Class)}\quad \frac{A \vdash a : \texttt{class}\ (\tau_y)\ \langle m : \tau; \rho\rangle \qquad \tau \equiv \texttt{Pre}\ \tau' \text{ or } \emptyset \qquad A \oplus (y : \tau_y) \vdash a' : \tau'}{A \vdash a\texttt{class}\ (a\ \texttt{with}\ m = a')\text{: } \texttt{class}\ (\tau_y)\ \langle m : \texttt{Pre}\ \tau'; \rho\rangle}$$

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

Quand une méthode est redéfinie, son type doit être le même que celui de l'ancienne méthode.

Bien sûr, une sous-classe peut utiliser des versions spécialisées de ses classes parentes en instantiant certaines variables.

La règle Inherits montre que le type de self dans une classe est le même que celui de la classe héritée. Ce qui garantit que les méthodes qui retournent self sont typées correctement, mais ne restreint pas la puissance, car le type de la classe héritée est une instance du type de la classe parente.

```
let copy = Oo.copy;;

class copy () as s = method copy = copy s end;;
```

Dans cette classse, `'a` est lié au type de `self`, c'est-à-dire que les objets de cette classe ont le type `<copy : 'a> as 'a` (avec la même variable `'a`). La classe `copy` peut maintenant être héritée. La méthode `copy` a bien le type attendu.

```
class point x as self =
```

```
    val x = x
    method getx = x
    method print = print_int self#getx
  end;;

  class copy_point x = inherit copy () inherit point x end;;
```

Sémantique

L'évaluation des classes consiste simplement à résoudre l'héritage. Pour cela, il faut évaluer la classe héritée puis l'extension de méthode.

$$E ::= \ \ldots \mid \texttt{new}\ E \mid \texttt{class}\ (y)\ \langle E\ \texttt{with}\ m = a \rangle$$

$$\texttt{class}\ (x)\ \langle \texttt{class}\ (y)\ \langle m_i = a_i{}^{i \in I} \rangle\ \texttt{with}\ m_j = a'_j \rangle \longrightarrow$$
$$\texttt{class}\ (y)\ \langle m_i = a_i{}^{iinI \setminus j}; m_j = a'_j[y/x] \rangle$$
$$\texttt{new class}\ (y)\ \langle m_i = a_i{}^{i \in I} \rangle \longrightarrow \varsigma(y) \langle m_i = a_i{}^{i \in I} \rangle$$

Oublis

Les variables d'instance mutables ne posent pas de difficulté particulière. Elle sont typées comme les variables non mutables. La modification en place des objets est typée par ajout de primitives, de façon similaire à la modification en style applicatif. La sémantique peut être décrite dans le formalisme décrivant les références.

La sécurité des constructions impératives est garantie par la restriction que seule les valeurs peuvent être polymorphes. En particulier, même si les classes sont paramétriques, les objets ne sont jamais polymorphes. En effet, la création d'une classe n'est pas une valeur ; elle est donc monomorphe. D'autre part, les objets produits par duplication ont le même type que leur original.

8.4.3 L'héritage n'est pas du sous-typage

Avant d'étudier plus en détail le mécanisme d'héritage, il est important de se convaincre que les deux concepts n'ont rien à voir.

L'héritage est la possibilité de construire des classes à partir d'autres classes en réutilisant globalement toutes leurs méthodes. C'est donc avant tout une propriété syntaxique. Le sous-typage est la possibilité pour un objet d'un certain type d'être considéré et donc utilisé comme un objet d'un autre type. C'est une propriété sémantique.

Nous avons montré dans la première partie du chapitre précédent que les points colorés construits à partir de points par héritage ne peuvent pas être considérés comme des points, si les points comportent une méthode testant l'égalité de deux points. Autrement dit, le type des points colorés n'est pas un sous-type des points. Cet exemple montre que l'héritage n'entraîne pas le sous-typage des types des objets des classes considérés.

Inversement, si les points colorés (sans égalité) sont aussi des points, il s'agit d'une propriété des objets qui est indépendante de la façon dont les points colorés ont été créés. On peut, bien entendu redéfinir sans hérités des points :

```
class point_coloré x c =
  val mutable x = (x:int)
  method getx = x
  val c = (c : string)
  method getc = c
end;;

let point_coloré = new point_coloré 1 "blanc−cassé";;
(point_coloré :> point);;
```

On peut aussi coercer le type d'un objet vers un type qui ne correspond pas à une définition de classe.

```
type pauvre_point = < >;;
(point_coloré :> pauvre_point);;
```

On pourrait aussi considérer deux implémentations indépendante des points dans le plan, fonctionnant en coordoonées cartésiennes pour l'une et en coordonnées polaires pour l'autre. Les deux classes devront fournir la même interface, c'est à dire que les types des objets de ces classes sont identiques donc a fortiori dans une relation de sous-typage. Pourtant les classes ne sont pas dans une relation de sous-classage. Identifier sous-classage et sous-typage interdirait cette abstraction de la représentation d'une classe. Ou forcerait à définir artificiellement une classe abstraite ayant la même interface, mais aucune implémentation dont hériteraient à la fois les deux implémentations des points.

Retenons que :

$\boxed{\text{Le sous-typage n'a rien à voir avec l'héritage}}$

La notion d'héritage est plus proche d'une notion de *filtrage* sur les types associés. La déclaration de la classe `point_coloré` ci-dessus produit la déclaration automatique de type équivalente à :

```
type 'r #point_coloré = <getx : int; getc : string; 'r>
```

En fait la variable de rangée est un paramètre implicite et elle est remplacée par une ellipse dans le type : on écrit simplement `#point_coloré` pour signifier `<getx : int; getc : string;` La #-abréviation d'une classe filtre tous le type des objets de tous les sous-classes. Cette propriété est immédiate : elle résulte du fait qu'une sous-classe ne peut pas caché une méthode définie dans une classe parente, ni changer son type, mais seulement le préciser.

Il n'y a pas de règles sans exceptions. Nous devons donc parler des nombreux langages qui pour simplifier identifient la notions d'héritage et de sous-typage. Ils sont de deux sortes.

- Certains comme Eiffel, O2 font cette identification au dépend de la sécurité. Ils permettent de considérer des points colorés avec égalité comme des points avec égalité. Ce qui bien entendu conduit à des erreurs à l'exécution. En général, les versions plus récentes de ces langages insèrent maintenant des tests dynamiques lorsque ces coercions dangereuses sont effectuées afin de rattraper les erreurs plus proprement.

- D'autres langages comme Java contourne le problème en affaiblissant les règles de typage. Il suffit d'interdire le type de `self` d'apparaître dans le type d'une méthode. Par exemple la méthode d'égalité aura le type `point` → `point` dans la classe `point` et conservera ce type lorsqu'elle sera héritée dans la classe des points colorés.

  Cet affaiblissement du système de typage force l'utilisateur à effectuer des coercions explicites qui peuvent échouées. On n'est donc pas très loin de l'approche ci-dessus, sinon que l'échec provient d'une coercion mise par l'utilisateur. Elle est donc prévisible.

  Le traitement de C++ est très similaire. La différence est qu'en C++ l'utilisateur est encouragé à effectuer une coercion sans test, c'est-à-dire forcer le type à la main avec un `cast`, ce qui peut avoir des conséquences plus graves.

8.4.4 Pré-méthodes

Le but de cette subsection est de montrer que les classes ne sont au fond que de l'assemblage des méthodes abstraites par rapport à l'objet `self` dans lequel elle seront placées.

Que ce soit dans la simulation des classes par des enregistrements extensibles dans le langage Caml, ou bien le typage des classes dans Objective Caml, les méthodes sont définies dans les classes comme des fonctions, abstraites par rapport aux variables d'instance et par rapport à `self`. Nous les appelons *pré-méthodes*.

Les pré-méthodes doivent pouvoir être utilisées comme méthodes des objets de la classe où elles sont définies (si cette classe n'est pas virtuelle), mais aussi comme méthodes des objets de ses sous-classes.

8.4.5 Liaison tardive et typage

La liaison tardive implique que dans une classe la valeur liée à `self` peut-être tout objet d'une sous-classe. En particulier, les méthodes qui dépendent de `self` doivent fonctionner aussi bien dans les classes que dans les sous-classes.

Cela est obtenu en donnant à `self` un type ouvert dans la définition d'une classe. Les pré-méthodes sont donc paramétrique par rapport à la valeur réelle de `self`. Le type dans une pré-méthode contient toutes les contraintes qui lui sont imposées dans cette pré-méthode. Le type de `self` dans est unifié dans toutes les pré-méthodes d'une classe. Il contient alors toutes les contraintes imposées à `self` dans cette classe.

Une sous-classe ne relâchera jamais des contraintes imposés à `self` dans une classe parente, car cela invaliderait le typage. En revanche, elle peuvent le contraindre d'avantage.

Une pré-méthode définie dans une classe parente et hérité est spécialisée par instantiation de type pour satisfaire les nouvelles contraintes imposées à `self`.

C'est donc le mécanisme général du polymorphisme de ML, et le polymorphisme des variables de rangées qui assure la sécurité du typage de la liaison tardive.

8.4.6 Héritage

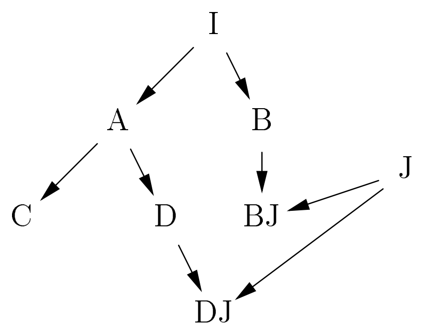
Héritage simple

L'héritage est simplement un assemblage des composantes de classes. variables d'instance et pré-méthodes, tout en gardant un contrôle précis sur la portée des variables. L'héritage simple est un ajout de composantes une par une. Il repose sur la concaténation des enregistrements. L'héritage multiple repose sur la concaténation des enregistrements. On peut ajouter les composantes en blocs.

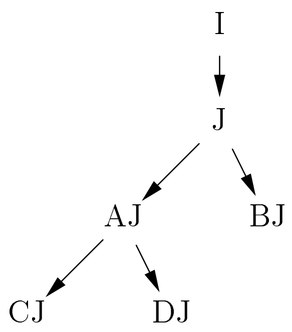
Héritage multiple

Rappelons que l'héritage multiple consiste à construire une classe à partir de plusieurs classes parentes. La construction d'héritage en Objective Caml est naturellement multiple. L'héritage multiple permet surtout d'ajouter à une classe existante les fonctionnalité d'une autre classe a posteriori. Sans l'héritage multiple, il faudrait ou bien dupliquer certaines opérations ou bien les insérer à la racine commune aux branches qui doivent être étendues. Par exemple, on peut construire la hiérarchie suivante (figure de gauche) :

Héritage multiple



Héritage simple



L'intérêt est que seule les classes B et D peuvent hériter de la classe J. Pour conserver le partage (écrire une seule fois les composantes de J) avec seulement de l'héritage simple, il est nécessaire d'introduire J très haut dans la hiérarchie (figure de droite). Il en résulte que la classe J est hérité dans toute la hiérarchie, ce qui peut être dangereux, ou créer des conflits.

De façon plus générale, l'héritage multiple permet d'organiser les classes de façon plus proche de leur dépendances réelles. Ceci accroît la sécurité bien entendu, mais renforce également tous les aspects de la modularité. Dans l'exemple ci-dessus, l'héritage multiple permet d'exprimer que BJ et DJ sont les seules classes à dépendre de J, alors que la version avec héritage simple force une dépendance artificielle de les classes en dessous de la racine commune à B et D.

8.4.7 Exemple

La difficulté dans la programmation avec des objets est souvent l'organisation des classes. Quelle sorte de modularité recherche-t-on ?

Pour montrer que ce problème n'est pas toujours simple, considérons l'exemple suivant. Imaginons que l'on manipule des figures géométriques. En commençant par des points, des segments, etc. et que l'on veuille introduire les cercles.

Qu'est-ce qu'un cercle ? Il semble naturelle de représenter un cercle par un point — son centre, et un rayon :

```
class circle p r =
  val c = (p : point)
  val r = (r : int)
  method center = c
  method radius = r
  method move dv = p#move
end;;
```

On s'aperçoit qu'un objet cercle comporte un certain nombre de méthodes telles que `move` sont identiques à celles des points ; de plus l'objet fait simplement suivre ces méthodes vers son centre. La définitions alternative suivante semble donc aussi raisonable :

```
class circle x r =
  inherits point x
  method center = (x :> point)
  val r = r
  method radius = r
end;;
```

Quelles sont les différences entre les deux définitions ? L'une est-elle préférable ? Il n'est pas évident de répondre à ces questions. D'abord, l'interface d'un objet cercle sera dans les deux cas la même ; il va de soit que l'on attend aussi un comportement identique.

La seconde définition évite de redéfinir les méthodes des points dans les cercles. D'autre l'ajout d'une méthode aux points est immédiatement répercutée vers les cercles. Enfin elle permet de considérer un cercle comme un point, et donc par exemple de définir un segment à partir de deux cercles comme le segment reliant leur centre. En bref, cette deuxième solution semble préférable.

Pourtant, elle ne permet pas de créer un point à partir d'un point déjà créé. Imaginons qu'un point soit partager par plusieurs objets géométriques et que l'on veuille maintenant créer un cercle autour de ce point. Avec la seconde définition, il faudra créer un nouveau point-cercle, c'est-à-dire que le cercle ne pourra pas partager le point existant. Ceci est embêtant si ce point est bougé physiquement, le cercle restera là où il était.

Il n'y a donc pas de meilleure solution. Il s'agit simplement de bine comprendre ce que l'on veut.

8.4.8 Conclusion

Il faut retenir que les classes sont essentiellement un ensemble de déclaration toplevel permettant de construire des objets ou bien d'être ultérieurement étendues avec d'autres déclarations.

Dans cette approche les classes sont essentiellement une collection de pré-méthodes, c'est-à-dire des méthodes abstraite par rapport à l'objet `self` dans lequel elle apparaîtront ainsi que par rapport aux variables d'instance de cet objet.

Les règles de typage des classes dans Objective Caml, font également apparaître une méticuleuse gestion de la visibilité des différentes liaisons. C'est aussi un des aspects important de la notion de classe.

Il faut retenir que la vision de `self` dans une classe est une vision ouverte, puisque nous ne connaissons alors qu'un sous-ensemble des méthodes de `self`. Dans Objective Caml cette ouverture est exprimé par un type objet ouvert, donc comportant une variable de rangée polymorphe.

Enfin, il est essentiel de distinguer la notion de sous-type et celle de sous-classe.

Pour en savoir plus

Il y a de nombreux livres sur la programmation à objets, mais hélas encore peu de bons livres. La liste ci-dessous ne se veut ni exhaustive, ni sélectionnant les meilleures ouvrages, mais quelques pointeurs illustrant des idées ou des langages précis.

Des documents directement liéés au cours

Le codage des objets et des classes avec des types enreistrements est assez bien couvert par l'article [Rém94b], toutefois plus technique. La présentation d'Objective Caml est également disponible en ligne à l'adresse
`http://pauillac.inria.fr/caml`. La formalisation d'Objective Caml, est décrite en détail dans [RV97]. La référence pour le sous-typage des types récursifs est [AC93], mais l'algorithme simple et efficace est décrit dans [KP93].

Des pointeurs pour un approfondissement plus théorique

La différence en héritage et sous-typage est le mieux décrit dans [CHC93].

Pour comprendre les objets dans un calcul plus simple (mais explicitement typés) on pourra regarder

- Un codage dans le système $F^{\Omega}_{<:}$. [PT94]
- Une théorie des objets primitifs [AC94, AC95].

Pour une très bonne comparaison entre les différentes façons de représenter les objets [BCP96].

La meilleure référence sur la théorie et le typage des objets est certainement un livre à paraître en été 1996 par M. Abadi et L. Cardelli.

Annexe A

Solutions des exercices

Exercice 1, page 20

Il y a deux radicaux qui peuvent être réduits dans un ordre arbitraire, on obtient les termes :

$$\texttt{let}\ f = \texttt{fun}\ (y)\ y\ \texttt{in}\ (\texttt{fun}\ (z)\ z)\ f \qquad \text{ou} \qquad (\texttt{fun}\ (z)\ z)\ (\texttt{fun}\ (y)\ y)$$

qui tous deux se réduisent en $(\texttt{fun}\ (y)\ y)$. En appel par valeur, seule la seconde réduction est permise, car il n'est pas possible de réduire sous une liaison.

Exercice 2, page 20

Si l'on veut être formel, on peut le montrer très facilement par une récurrence structurelle sur la taille des expressions.

Exercice 3, page 20

Dans le calcul sans constantes, il faut réduire les contextes d'évaluation à

$$E ::= [] \mid E\ a$$

pour interdire l'évaluation des arguments et autoriser la β-réduction dans toute sa généralité :

$$(\texttt{fun}\ (x)\ a_1)\ a_2 \xrightarrow{\epsilon} a_1[a_2/x] \qquad \text{Fun}$$
$$\texttt{let}\ x = a_2\ \texttt{in}\ a_1 \xrightarrow{\epsilon} a_1[a_2/x] \qquad \text{Let}$$

Avec des constantes, le plus simple serait de supposer que les constantes acceptent des arguments non évalués, mais c'est cacher la difficulté. Car les constantes doivent forcer le calcul des arguments strictement nécessaires.

Exercice 7, page 27

Pour traiter les paires de façon primitive, il faut d'abord étendre la syntaxe des expressions, sans oublier d'étendre également les valeurs :

$$\begin{aligned} a &::= \ldots \mid (a, a) \mid \texttt{fst}\ a \mid \texttt{snd}\ a \\ v &::= \ldots (v, v) \end{aligned}$$

Ensuite il faut étendre la notion de réduction, c'est-à-dire ajouter les contextes d'évaluation suivants :

$$E ::= \ldots \mid (E, a) \mid (v, E)$$

puis les deux réductions :

$$\texttt{fst}\ (v_1, v_2) \longrightarrow v_1 \qquad \text{et} \qquad \texttt{snd}\ (v_1, v_2) \longrightarrow v_2$$

Pour formuler le typage, on définit un nouveau symbol de type $(\_ * \_)$. Nous pouvons tricher en définissant le typage des paires comme celui des constantes, mais sinon, il nous faut écrire les règles

$$\frac{A \vdash a_1 : \tau_1 \qquad A \vdash a_2 : \tau_2}{A \vdash (a_1, a_2) : \tau_1 * \tau_2} \qquad \frac{A \vdash a : \tau_1 * \tau_2}{A \vdash \texttt{fst}\ a : \tau_1} \qquad \frac{A \vdash a : \tau_1 * \tau_2}{A \vdash \texttt{snd}\ a : \tau_2}$$

Pour être rigoureux, il faudrait refaire toutes les preuves sur le typage, puisque nous avons changé la relation de typage... De même il nous faut a priori refaire toutes les preuves de correction de la sémantique. Nous nous contentons de décrire les cas à ajouter dans les preuves. Dans le lemme 4, page 22 ces cas sont analogues à ceux de l'application. Dans le lemme 6, page 23, ces cas sont triviaux. Dans le théorème 1 (contraction de radical), il faut vérifier les cas suivants :

Cas A est $\texttt{fst}\ (a_1, a_2)$: En reconstruisant la dérivation de typage de a, il est facile de montrer que $A \vdash a_1 : \tau$.

Cas A est $\texttt{snd}\ (a_1, a_2)$: est symétrique du précédent.

Enfin dans le théorème 2, page 25, il faut ajouter les cas suivants :

Cas a est $\texttt{fst}\ a_1$: Comme a_1 est une valeur, il est facile de montrer à partir de la dérivation de typage de a que $A \vdash a_1 : \tau_1 * \tau_2$. Les seules valeurs de types produit sont les produits donc a_1 est de la forme (a_2, a_3) et a peut être réduit.

Cas a est $\texttt{snd}\ a_1$: Ce cas est symétrique du précédent.

Cas a est (a_1, a_2) : a est une valeur.

Les cas que nous avons explicitement traités ci-dessus correspondent exactement à la vérification des deux hypothèses sur les constantes. Mais, si nous voulons être complets, il faut également vérifier d'autres cas, similaires à l'application que nous avons ignorés ci-dessus.

Les arguments ci-dessus ne constituent pas exactement une démonstration, mais plutôt un méta-raisonnement, c'est-à-dire un discours sur comment faire une vraie démonstration. Ce discours nous rend très confiant sur la possibilité d'écrire vraiment la démonstration. Mais, pour être plus formel, il faudrait réécrire toutes les preuves, ou bien, ce qui revient au même, avoir incorporés les paires dès le début du chapitre précédent.

Exercice 8, page 27

Nous ne donnons pas un corrigé complet, mais les idées principales. On définit la traduction $[\![a]\!]$ par récurrence structurelle. Les trois nouvelles constructions se traduisent pas des applications de primitives, les autres sont homomorphes. Pour montrer que la réduction commute, il faut montrer formellement que si $a_1 \longrightarrow a_2$ alors $[\![a_1]\!] \longrightarrow [\![a_2]\!]$, par récurrence structurelle sur a. Plus simple est de remarquer que $[\![E[a]]\!]$ est égal à $[\![E]\!][[\![a]\!]]$. Il suffit alors de vérifier la propriété précédente pour les radicaux. Les cas intéressants sont les deux nouvelles règles de ML_1 qui correspondance dans ML_2 à des δ-règles.

Pour montrer l'équivalence, on rencontre le problème des expressions $(a, \_)$ bien formées dans ML_2, car ce sont des primitives partiellement appliquées (lire $(\_, \_)\ a_1$), alors que la syntaxe les interdit dans ML_1. Une solution est de se restreindre dans ML_2 aux programmes où les primitives ne sont jamais partiellement appliquées (propriété heureusement conservée par réduction). On peut alors montrer que la traduction composée avec la traduction inverse est l'identité. Nous avons mis en bijection les expressions des deux langages et leur relation de réduction. Pour éliminer la restriction, on pourrait définir dans ML_2 une transformation qui remplace les primitives $(\_, \_)$ par `fun` $(x\ y)\ (x, y)$ et montrer qu'un programme et son transformé sont observationnellement équivalents, c'est-à-dire que si placés dans le même contexte l'un se réduit en une valeur non fonctionnelle alors le second se réduit sur la même valeur. Ceci est plus difficile à formaliser.

Exercice 9, page 28

Sinon, on pourrait définir `type g` $=$ `C of` α. et écrire la fonction `fun` (x) ($\mathtt{F}_\mathtt{g}$ (`C` x)) qui se comporte comme l'identité mais à le type $\tau \rightarrow \tau'$ quelque soient τ et τ'. Pour des instances bien choisies de τ et τ' cela contredit la préservation des types.

En fait la définition ci-dessus est possible, mais il faut alors s'interdire de généraliser la variable α dans A_0.

Par contre, il est tout à fait possible que τ contiennent des variables superflu. Pour s'en assurer, il suffit de vérifier que cela se ramène au cas général en ajoutant un constructeur suplémentaire $\mathtt{C}_{n+1}$ `of` α qui rend cette variable de type non superflue et ne jamais utiliser le nouveau constructeur $\mathtt{C}_{n+1}$!

Exercice 11, page 28

On peut définir $\texttt{type bool} = \texttt{true} \mid \texttt{false}$ et choisir la primitive *if_then_else* égale à $F_{\texttt{bool}}$.

L'utilisateur doit cependant écrire $\texttt{if\_then\_else}\ a_1\ (\texttt{fun}\ ()\ a_2)\ (\texttt{fun}\ ()\ a_3)$ au lieu de $\texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ a_3$.

En fait, on peut ajouter du sucre syntaxique, c'est-à-dire écrire $\texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ a_3$ pour signifier $\texttt{if\_then\_else}\ a_1\ (\texttt{fun}\ ()\ a_2)\ (\texttt{fun}\ ()\ a_3)$.

Exercice 12, page 28

Il faut simplement généraliser les types construits à des constructeurs d'arité multiples de la forme :

$$\texttt{type}\ \bar{\tau}\ \texttt{g} = \texttt{C of}\ \bar{\tau}_i \mid \ldots \texttt{C of}\ \bar{\tau}_n$$

La généralisation est facile, et laissée en exercice. On peut alors définir :

$$\texttt{type}\ (\alpha_1, \alpha_2)\ (\_ * \_) = (\_, \_)\ \texttt{of}\ (\alpha_1,\ \alpha_2)$$

$$\texttt{fst} \stackrel{def}{=\!=} \texttt{fun}\ (z)\ (F_*\ z\ (\texttt{fun}\ (x)\ \texttt{fun}\ (y)\ x)) \qquad \texttt{snd} \stackrel{def}{=\!=} \texttt{fun}\ (z)\ (F_*\ z\ (\texttt{fun}\ (x)\ \texttt{fun}\ (y)\ y))$$

Exercice 13, page 28

On étend le langage avec les expressions ci-dessus, et les constructions $\texttt{C}_i\ a$. Les règles de typages sont

$$\frac{A \vdash a : \tau_i}{A \vdash \texttt{C}_i\ a : \tau\ \texttt{g}}\ (\texttt{C}_i) \qquad \frac{A \vdash a : \tau\ \texttt{g} \qquad A \oplus (x : \tau_i) \vdash a_i : \tau'}{\texttt{match}\ a\ \texttt{with}\ \texttt{C}_1\ (x) \Rightarrow a_1 \ldots \mid \texttt{C}_n\ (x) \Rightarrow a_n : \tau'}\ (\textsc{Match})$$

La réduction est alors une règle :

$$(\texttt{match}\ \texttt{C}_k\ (v)\ \texttt{with}\ \texttt{C}_1\ (x) \Rightarrow a_1 \ldots \mid \texttt{C}_n\ (x) \Rightarrow a_n) \longrightarrow (\texttt{fun}\ (x)\ a_k)\ v$$

Il faut montrer bien sûr que les preuves s'étendent aux nouvelles constructions, ce qui revient à montrer les hypothèses H_1 et H_2 plus d'autres cas redondants (car similaires à l'application). d'une autre façon.

Exercice 14, page 29

```
let Fact fact x = if x = 0 then 1 else x * fact (x-1)

let fact x = Y Fact x
```

Exercice 19, page 36

Si vous n'avez pas trouvé de contre-exemple, cherchez encore. Si vous avez répondu négativement à la dernière question, faites les deux exercices suivants et reposez vous la question. Si vous avez toujours une réponse négative, alors votre exemple est intéressant.

Exercice 20, page 36

On peut toujours écrire (`fun` (x) a_2) a_1. Donc il n'y a pas de problème à écrire `let` $x = a_1$ `in` a_2, mais il faut le comprendre comme du sucre syntaxique pour l'expression ci-dessus, et donc typer cette expression sans généraliser.

Exercice 21, page 36

Par η-conversion, on peut remplacer $g\ v$ par `fun` $(y)\ g\ v\ y$ qui est une valeur et permet donc à g' d'être polymorphe dans a. Si g est lui de la forme `fun` $(z\ x)\ a_0$, alors les deux programmes sont équivalents. Mais ce n'est plus nécessairement le cas si g est de la forme `fun` $(z)\ a_1\ a_2$: au mieux la sémantique sera la même (avec un calcul moins efficace), mais la sémantique peut aussi être différente si l'évaluation de $a_1\ a_2$ modifie la mémoire.

Exercice 23, page 40

Non, les deux solutions reposent sur des invariants globaux incompatibles. Voici un contre-exemple :

```
let isomorphe_a_ref x =
  let r = ref x in let ecrit y = r := y and lit () = !r in
  (ecrit, lit) in
let poly_ref = isomorphe_a_ref (fun x → x) in
  (fst poly_ref) succ; (snd poly_ref) true;;
```

La fonction `isomorphe_a_ref` crée une référence et rend les fonctions qui permettent d'écrire et de lire dans cette référence ; ceci en cache le type. Elle est polymorphe car c'est une valeur. Le reste du programme est polymorphe car le type REF n'y apparaît pas bien que des références y sont crées.

Exercice 24, page 41

Étant donné F, on veut trouver f égal à $Y\ F$ tel que $f\ x$ soit égal à $F\ f\ x$. Supposons que f soit dans le contenu d'une référence r. On a donc $f\ x$ égal à $F\ f\ x$... qu'il suffit de mettre dans le contenu de la référence r avant de retourner f. Cela donne le programme :

```
let Y F =
```

```
let r = ref (fun _ → raise (Failure "Y")) in
let f x = F !r in
r := f; !f;;
```

On peut aussi utiliser le type option pour initialiser le contenu de la référence avec *None* puis le remplacer plus tard par *None f*.

Exercice 27, page 44

Oui, car les exceptions peuvent cacher des types récursifs. L'exercice 16 a montrer comment définir Y à partir de deux fonctions `replier` et `deplier` à partir d'un type récursif à occurence négative. On peut utiliser une exception :

```
exception Masque of ((unit → unit) → (unit → unit));;
```

pour écrire les fonctions

```
let replier = fun x → fun (y:unit) → (raise (Masque x); ())
and deplier = fun x → try (x(): unit); fun x → x with Masque y → y;;
```

Les fonctions `replier` et *deplier* sont des coercions inverses :

$$U \underset{\texttt{replier}}{\overset{\texttt{deplier}}{\rightleftarrows}} (U \to U)$$

où U est égal à `unit → unit`. Elle peuvent donc servir a' emrober tout terme du lambda-calcul pour le rendre typable avec les types simples. Le codage est classique :

$$\begin{aligned} [\![x]\!] &= x \\ [\![\texttt{fun}\ (x)\ a]\!] &= \mathit{replier}(\texttt{fun}\ (x)\ [\![a]\!]) \\ [\![a_1\ a_2]\!] &= \mathit{deplier}([\![a_1]\!]\ [\![a_2]\!]) \end{aligned}$$

En particulier on peut écrire le combinateur $[\![Y]\!]$.

Exercice 29, page 48

Les règles d'évaluation sont alors

$$(v \ @\ \{\ell_0 = v_0\}).\ell_0 \xrightarrow{\epsilon} v \qquad (\delta_1)$$
$$(v \ @\ \{\ell_0 = v_0\}).\ell \xrightarrow{\epsilon} v.\ell \qquad (\delta_2)$$

Les deux approches sont équivalentes dans un langage purement fonctionnel (i.e. sans effets de bords). Cependant avec la nouvelle approche les enregistrements sont partagés, et une extension naturelle avec des champs mutables ne fournirait pas la sémantique attendue.

On distingue l'ancienne présentation (1) et la nouvelle présentation (2). La réduction dans (2) peut être abrégée en regroupant toutes les règles δ_2 suivies d'une règle δ_1 (qui ne peuvent jamais être en nombre infini). Les enregistrements de (1) peuvent être traduits en valeurs enregistrements de (2) de façon évidente et non ambigüe. On peut alors montrer qu'une étape de réduction dans (1) commute avec zéro ou une étape de réduction (regroupée) dans (2), modulo la traduction des enregistrements.

La terminaison dans (1) ou (2) sont équivalentes, et lorsque les programmes terminent, les valeurs obtenues se correspondent modulo la traduction des valeurs d'enregistrements.

Exercice 30, page 50

On ne peut plus écrire $\texttt{Pre} \rightarrow \emptyset(\tau)$ ni $\{\underline{\{\emptyset\}}\}$. On peut encore écrire $\{l_1 : \tau_1; \ell_1 : \tau_2; \underline{\emptyset}\}$ ainsi que $\{\ell : \underline{\alpha}; \underline{\alpha}\}$. La résponse à la deuxième question est décrite dans la section suivant l'énoncé de l'exercice.

Exercice 31, page 51

Comme suggéré, on ajoute des variables de sortes. On se donne des variables de sortes ϵ, et l'on construit les expressions sortes avec les constructeurs de sortes $\texttt{Type}$ and $\emptyset$ d'arité 0, et ($\texttt{Row}$ _) et (ℓ._) d'arité 1. La signature Σ_1 permet de restreindre les termes par la grammaire suivante :

$$
\begin{aligned}
\kappa &::= \epsilon \mid \texttt{Type} \mid \texttt{Row}\ (\kappa_1) \\
\kappa_1 &::= \epsilon \mid \emptyset \mid \ell \cdot \kappa_1
\end{aligned}
$$

La signature Σ_2 élimine les termes dont de la forme $\ell_1 \cdot \ldots \ell_k \cdot \kappa_1$ où au moins deux des étiquettes du préfixe sont identiques. Les sortes sont égales modulo la seule équation de commutativité gauche :

$$\ell_1 \cdot \ell_2 \cdot \epsilon = \ell_2 \cdot \ell_1 \cdot \epsilon$$

Il est facile de montrer que l'algèbre des sortes est une algèbre d'enregistrements particulière.

On en déduit un algorithme d'unification sur les sortes... pourvu que les sortes soient elles-même bien formées. Cet agorithme termine et est correct pour les expressions de sorte bien formées. En fait, commes les expressions de sorte initiales sont toujours localement bien formées, le seul cas d'expressions de sorte mal formées qui peuvent apparaître en cours de calcul sont des expressions cycliques. Dans ce cas l'agorithme de vérification des sortes boucle, en fabriquant des sortes de plus en plus grandes. Comme ces sortes sont construites à partir des étiquettes figurant dans le programme source, donc en nombre connu, il est facile de détecter ce cas et l'expression de type testée est donc mal formée.

Exercice 33, page 55

On le vérifie d'abord pour la δ-règle des variantes donnée ci-dessus, puis indépendamment, pour la β-règle. On le montre ensuite pour une réduction ayant lieu dans un contexte d'évaluation E quelconque par induction structurelle sur E.

Exercice 34, page 55

Le typage des enregistrements fournit :

$$[\![\texttt{inject}_\ell]\!] : \alpha_1 \rightarrow \ell : \texttt{Pre}\ (\alpha_1 \rightarrow \alpha_2); \alpha_3 \rightarrow \alpha_2$$

Il y a deux problèmes, cependant. D'une part, ce type est compliqué et donc peu lisible. D'autre part, appliquée à une valeur monomorphe, c'est-à-dire une fois α_1 remplacé par un type τ_1, on s'attend à avoir un type monomorphe, alors que les variables α_2 et α_3 sont libre et donc universellement quantifiées en tête.

Exercice 37, page 62

Une dérivation du jugement est

$$\dfrac{\dfrac{\dfrac{(f : \tau_1 \rightarrow \tau_2, x : \tau_1') \vdash f : \tau_1 \rightarrow \tau_2 \qquad \dfrac{(f : \tau_1 \rightarrow \tau_2, x : \tau_1') \vdash x : \tau_1' \qquad \tau_1' \leq \tau_1}{(f : \tau_1 \rightarrow \tau_2, x : \tau_1') \vdash x : \tau_1}\ (\textsc{Sub})}{\begin{array}{c}(f : \tau_1 \rightarrow \tau_2, x : \tau_1') \vdash f\ x : \tau_2 \\ \vdots\end{array}}\ (\textsc{App}) \qquad \tau_2 \leq \tau_2'}{(f : \tau_1 \rightarrow \tau_2, x : \tau_1') \vdash f\ x : \tau_2'}\ (\textsc{Sub})}{\vdash \texttt{apply}: (\tau_1 \rightarrow \tau_2) \rightarrow \tau_1' \rightarrow \tau_2'}\ (\textsc{Fun})$$

La fonction `apply` se comporte comme l'identité : elle ne calcul rien, et se contente de changer le type de ses arguments. On dit que c'est une *fonction de retypage*. On peut utiliser une telle fonction pour effectuer des coercions de types explicites. Le typage de `apply` justifie la relation de sous-typage de la flèche, car il montre qu'on peut remplacer les occurrences de la règle Sub sur les flèches par coercions explicites (en ajoutant des applications de la constante `apply`).

Exercice 38, page 62

Le type $\alpha \rightarrow \{\ell : \texttt{Pre}\ \alpha; \emptyset\}$ donné par ML (sans utilisation de la règle de sous-typage) est bien sûr un type correct. Nous pouvons en déduire d'autres en appliquant la règle de sous-typage seulement à la fin. En propageant l'inclusion par la règle structurelle pour les types fonctionnels, cette fonction a donc aussi le type $\tau \rightarrow \tau'$ pourvu que $\tau \leq \alpha$ et $\{\ell : \texttt{Pre}\ \alpha; \emptyset\} \leq \tau'$. Par exemple, on peut prendre τ' égal à $\{\emptyset\}$ et τ quelconque. Mais on peut aussi prendre τ' égal à $\{\ell : \texttt{Pre}\ \tau; \emptyset\}$ et τ quelconque. Toutes ces solutions sont incompatibles. Donc il n'y a pas de type principal.

Nous pouvons aussi prendre des instances du type ML, par exemple, en remplaçant α par $\{x : \texttt{Pre}\ I; \emptyset\}$. Il est alors possible de choisir $\{x : \texttt{Pre}\ I; y : \texttt{Pre}\ I; \emptyset\}$ pour τ et le type enregistrement vide pour τ'. Enfin, il faut aussi envisager d'appliquer également la relation de sous-typage à l'intérieur de la dérivation (mais pour cet exemple, nous n'obtiendrions pas plus de typages).

Nous aimerions dire que la fonction a le type principal $\alpha \rightarrow \{\ell : \texttt{Pre}\ (\alpha'); \emptyset\}$ tel que $\alpha \leq \alpha'$. Mais cette expression n'est pas un type.

Exercice 39, page 62

Dans le premier exemple, on peut considérer que $(\alpha \to \alpha) \to \alpha \to \alpha$ est un type principal. En effet tous les autres types possibles de la fonctions, qui sont $(\tau_1 \to \tau_1') \to \tau_2 \to \tau_2'$ tels que $\tau_2 <: \tau_1$ et $\tau_1' <: \tau_2'$, peuvent se déduire du premier par instantiation et par l'application de la règle SUB.

Ce n'est plus le cas pour le second exemple. Les types possibles sont de la forme $(\tau_1 \to \tau_1') \to (\tau_2 \to t_2') \to \tau \to (\tau_1' \times \tau_2')$ tels que

$$\tau <: \tau_1 \qquad \tau <: \tau_2 \qquad \tau_1' <: \tau \qquad \tau_2' <: \tau$$

Ils ne peuvent pas se déduire des types ML $(\tau \to \tau) \to (\tau \to \tau) \to \tau \to \tau \to (\tau \times \tau)$ simplement par application d'une relation de sous-typage.

Exercice 40, page 64

C'est une simple induction sur la structure de τ qui repose sur le fait que deux termes de même structure sont dans une relation de sous-typage si leurs sous-termes immédiats sont aussi dans une relation de sous-typage (directe ou inverse pour les positions contravariantes).

Exercice 41, page 64

Par exemple, on peut prendre la substitution $\alpha \mapsto \alpha'$. En appliquant cette substitution, partout dans la dérivation nous conservons une dérivation valide et dans laquelle les relations de sous-typage sont sans hypothèses. D'autres substitutions sont bien entendu possibles, mais le raisonnement restant le même ; elles correspondent à d'autres dérivations sans hypothèses de sous-typage.

Exercice 42, page 66

Dans le premier cas, il suffit de voir que l'ensemble est fermé et immédiatement cohérence. Une solution est $\bot \to \bot$ for α.

Dans le second cas, ni $\bot$ ni $\top$ ne sont solutions. Une solution est donc une flèche. Cependant en remplaçant α par $\alpha_1 \to \alpha_2$, on obtient après fermeture les contraintes $\{\alpha_1 \to \alpha_2 <: \alpha_1\}$ et $\alpha_1 <: \alpha_1 \to \alpha_2$ qui ne sont pas simultanément solvables.

Exercice 43, page 68

Une instance de α satisfaisant C est $\bot$ ou bien de la forme $\top \to \tau'$ telle que τ' est une instance de α' satisfaisant C. Par symétrie, une instance de α' satisfaisant C est $\top$ ou bien de la forme $\top \to \tau$ telle que τ est une instance de α' satisfaisant C. On en déduit la forme générale des instances τ et τ' de α et α' satisfaisant C.

$$\begin{aligned} \tau_n &::= \bot \mid \top \to \tau_{n-1} \mid \texttt{rec}\ \alpha.(\top \to \alpha) \\ \tau_n' &::= \top \mid \bot \to \tau_{n-1}' \mid \texttt{rec}\ \alpha.(\bot \to \alpha) \end{aligned}$$

De plus, les τ_i forment une suite croissante de limite $\texttt{rec}\ \alpha.(\top \to \alpha)$. Inversement, les τ'_i forment une suite décroissante de limite $\texttt{rec}\ \alpha.(\bot \to \alpha)$. De plus $\texttt{rec}\ \alpha.\top \to \alpha <: \texttt{rec}\ \alpha.\bot \to \alpha$. Donc on a bien $\tau <: \tau'$ pour toute solutions $\alpha \mapsto \tau, \alpha' \mapsto \tau')$ de C.

Montrons $C \Vdash \alpha <: \alpha'$ n'est pas possible. Une dérivation de la forme $C \vdash \alpha <: \tau''$ ne peut se terminer que par une règle Context ou un règle Trans. Cependant dans le dernier cas, une prémisse doit se terminer par une règle Context dérivant un jugement de la même forme. Or étant donné la forme de C, il n'est pas possible de dériver un jugement de la forme $C \vdash \alpha <: \tau''$.

Exercice 44, page 70

Il suffit d'appliquer les règles de typage de l'application et des fonctions. La seule contrainte $\texttt{Int} \leq (\tau \to \tau) \to \texttt{Int}$ suffit.

Par analyse des dérivations possibles, on peut montrer que l'ensemble de contraintes comportera au moins la contrainte $\texttt{Int} \leq \tau \to \tau'$ et ne peut donc pas être cohérent (règle C-Arrow).

Exercice 47, page 78

L'extension polymorphe des enregistrements n'est pas typable de façon aussi polymorphe. Le programme suivant n'est pas typable avec le sous-typage sans variables de rangées.

```
let extend r x = r || {a = x} in extend (extend {b = 2} 1) true;;
```

Inversement, sans le sous typage, on en peut oublier la présence des champs. Le programme suivant n'est pas typable sans inclusion.

```
if true then {a = 1} else {a = 1; b =2};;
```

Il faut modifier la règle $\leq$-Record en

$$\frac{C \Vdash \tau_i \leq \tau'_i \quad \forall i \in 0..n}{C \Vdash \{\ell_i : \tau_i^{\,i\in 1..n}; \tau_0\} \leq \{\ell_i : \tau_i'^{\,i\in 1..n}; \tau_0\}} \ (\leq\text{-Record})$$

Cela nous aménera aussi à modifier les règles de propagation et de cohérence : Il faut modifier la règle de propagation :

$$\{\ell_i : \tau_i^{\,i\in 1..n}; \tau_0\} \leq_C \{\ell_i : \tau_i'^{\,i\in 1..n}; \tau'_0\} \Longrightarrow \tau_i \leq_C \tau'_i, \forall i \in 0..n \quad (\text{P-Record})$$

Le reste du formalisme est inchangé. Toutes les propriétés restent vraies (mais il faut le vérifier).

Exercice 49, page 82

Oui. En particulier, il existe toujours des types principaux.

Exercice 53, page 105

L'objet p lui-même.

Exercice 54, page 106

```
class copy () as moi_m^^eame =
  method copy = Oo.copy moi_m^^eame
end;;
```

Cet exemple n'est guère intéressant car la fonction copy peut toujours être utilisée à la place d'un objet. Par contre si l'objet a un état interne, une copie profonde devra répercuter la copie sur les variables d'instance.

Exercice 55, page 106

On utilise la construction $\{\langle\_\rangle\}$. Par exemple, on peut réécrire la classe précédente :

```
class copy () =
  method copy = {< >}
end;;
```

Si dans une sous-classe, une variable d'instance est un objet qui doit également être dupliqué, on pourra redéfinir la méthode `copy` :

```
class deep_copy x as self =
  inherit copy () as super
  val x = x
  method local_copy = {< x = x#copy >}
  method copy = (super#copy)#local_copy
```

L'utilisation de `local_copy` permet l'héritage de la méthode `super#copy` qui effectue une copie profonde de la classe parente, puis une copie locale. Cette méthode n'est pas efficace

Un exemple plus concret, est d'avoir un vecteur dans une des variables d'instance.

```
class vector f n =
  val m =
     let v = Vect.create (f k) p in
     for i = 0 to p-1 do v.(i) ← f i done; v
  method deep_copy = {< m = Vect.copy m >}
end;;
```

Pour en savoir plus

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *Transactions on Programming Languages and Systems. ACM*, 15(4) :575–631, 1993.

[AC94] Martín Abadi and Luca Cardelli. A theory of primitive objects : Untyped and first-order systems. In *Theoretical Aspects of Computer Software*, pages 296–320. Springer-Verlag, April 1994.

[AC95] Martín Abadi and Luca Cardelli. A theory of primitive objects : Second-order systems. *Science of Computer Programming*, 25(2-3) :81–116, December 1995. Preliminary version appeared in D. Sanella, editor, Proceedings of European Symposium on Programming, pages 1-24. Springer-Verlag, April 1994.

[ACR92] Martín Abadi, Luca Cardelli, and Benjamin Pierce Didier Rémy. Dynamic typing in polymorphic languages. In *Proceedings of the 1992 workshop on ML and its applications*, page 92, San Francisco, USA, June 1992.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture*, pages 31–41. ACM press, 1993.

[Bar84] Hans P. Barendregt. *The Lambda-Calulus. Its Syntax and Semantics*, volume 103 of *Studies in Logic and The Foundations of Mathematics*. North-Holland, 1984.

[BCP96] Kim Bruce, Luca Cardelli, and Benjamin Pierce. Comparing objects encodings. Presented at the FOOL'3 workshop. Available electronically in informal workshop proceedings, July 1996.

[Ber93] Bernard Berthomieu. Tagged types, a theory of order sorted types for tagged expressions. Research Report 93083, LAAS, 7, Avenue du Colonnel Roche, 31077 Toulouse, France, March 1993.

[BT96] Lars Birkedal and Mads Tofte. From region inference to von neumann machines via region representation inference. In *Proceedings of the 23rd ACM Conference on Principles of Programming Languages*, January 1996.

[CHC93] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[CM89] Luca Cardelli and John C. Mitchell. Operations on records. In *Fifth International Conference on Mathematical Foundations of Programming Semantics*, 1989.

[Cur90] Pavel Curtis. Constrained quantification in polymorphic type analysis. Technical Report CSL-90-1, Xerox Palo Alto Research Center, February 1990.

[DRW95] Catherine Dubois, François Rouaix, and Pierre Weis. Extensional polymorphism. In *Proceedings of the 22th ACM Conference on Principles of Programming Languages*, January 1995.

[EST] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. To appear in OOPSLA 1995.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics, New Orleans*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. To Appear. Currently available as `ftp://ftp.cs.jhu.edu/pub/scott/ooinfer.ps.Z`.

[GR93] C. A. Gunter and R. Rémy. A proof-theoretic assessment of runtime type errors. Research Report 11261-921230-43TM, AT&T Bell Laboratories, 600 Mountain Ave, Murray Hill, NJ 07974-2070, 1993.

[GRR95] C. A. Gunter, R. Rémy, and Jon G. Riecke. A generalization of exceptions and control in ml. In *Proc. ACM Conf. on Functional Programming and Computer Architecture*, 1995.

[Gun93] Carl Gunter. *Semantics of programming languages.* ? ? ?, 1993 ?

[HMT91] Robert Harper, Robin Milner, and Mads Tofte. *The definition of Standard ML.* The MIT Press, 1991.

[HP91] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando FL*, pages 131–142. ACM, January 1991. Extended version available as Carnegie Mellon Technical Report CMU-CS-90-157.

[KP93] Dexter Kozen and Jens Palsberg. Efficient recursive subtyping. In *Proc. 20th symp. Principles of Programming Languages*, pages 419–428. ACM press, 1993.

[Läu92] Konstantin Läufer. *Polymorphic Type Inference and Abstract Data Types.* PhD thesis, New York University, 1992.

[Läu96] Konstantin Läufer. Putting type annotations to work. In *Proceedings of the 23th ACM Conference on Principles of Programming Languages*, January 1996. Preliminary version at `http://wwwipd.ira.uka.de/~odersky/`.

[Ler92] Xavier Leroy. *Typage polymorphe d'un langage algorithmique.* Doctoral dissertation (in French) Thèse de doctorat, Université Paris 7, 1992.

[LM91] Xavier Leroy and Michel Mauny. Dynamics in ML. In John Hughes, editor, *Conference on Functional Porgramming and Computer Achitecture*, volume 523 of *Lecture Notes in Computer Science*, pages 406–426. Springer-Verlag, 1991.

[LO92] Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, 1992.

[Oho90] Atsushi Ohori. Extending ML polymorphism to record structure. Technical Report CSC 90/R24, University of Glasgow, Department of Computer Science, September 1990.

[Pie91] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, Carnegie Mellon University, December 1991. Available as School of Computer Science technical report CMU-CS-91-205.

[Pot95] François Pottier. Simplifying constrained types. Draft, August 1995.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[Rém91] Didier Rémy. Type inference for records in a natural extension of ML. Research Report 1431, Institut National de Recherche en Informatique et Automatisme, BP 105, F-78 153 Le Chesnay Cedex, May 1991.

[Rém92a] Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatisme, BP 105, F-78 153 Le Chesnay Cedex, 1992.

[Rém92b] Didier Rémy. Projective ML. In *1992 ACM Conference on Lisp and Functional Programming*, pages 66–75, New-York, 1992. ACM press.

[Rém93a] Didier Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, Institut National de Recherche en Informatique et Automatisme, BP 105, F-78 153 Le Chesnay Cedex, 1993.

[Rém93b] Didier Rémy. Typing record concatenation for free. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1993.

[Rém94a] Didier Rémy. Programming objects with ML-ART : An extension to ml with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 321–346. Springer-Verlag, April 1994.

[Rém94b] Didier Rémy. Programming objects with ML-ART : An extension to ml with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 321–346. Springer-Verlag, April 1994.

[RV97] Didier Rémy and Jérôme Vouillon. Objective ML : A simple object-oriented extension to ML. In *Proceedings of the 24th ACM Conference on Principles of Programming Languages*, January 1997. To appear. Also presented at the FOOL'3 workshop, available electronically in informal workshop proceedings.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Proc. IEEE Symp. on Logic in Computer Science*, 1992.

[WF91] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Rice University, 1991.