Chapter 5

La programmation impérative

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

5.1 Les références

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

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

Exemple: l'expression suivante s'évalue en 4

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

En effet, le contenu de `r` est initialement 3. On calcule ensuite `!r + 1`, c'est-à-dire `3 + 1`, et on stocke cette valeur dans `r`. Enfin, on renvoie le contenu courant de `r`, qui est 4.

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

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

(La construction $a; b$ évalue a, puis b, et renvoie la valeur de b. On peut la voir comme une abrévation pour `let` $x = a$ `in` b où x n'est pas libre dans b.)

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

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

De même, une liste simplement chaînée dont on peut modifier le chaînage en place s'écrit:

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

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

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

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

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

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

5.2 Sémantique à réduction pour les références

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

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

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

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

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

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

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

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

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

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

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

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

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

$$
\begin{array}{rcll}
+\ (n_1, n_2) \mathbin{/} s & \stackrel{\varepsilon}{\to} & n \mathbin{/} s \quad \text{si } n_1, n_2 \text{ entiers et } n = n_1 + n_2 & (\delta_+) \\
\texttt{fst}\ (v_1, v_2) \mathbin{/} s & \stackrel{\varepsilon}{\to} & v_1 \mathbin{/} s & (\delta_{fst}) \\
\texttt{snd}\ (v_1, v_2) \mathbin{/} s & \stackrel{\varepsilon}{\to} & v_2 \mathbin{/} s & (\delta_{snd})
\end{array}
$$

Exemple: on a la séquence de réductions suivante:

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

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

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

5.3 Typage des références

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.4 Restreindre la généralisation aux expressions non expansives

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

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

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

Expressions non-expansives:

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

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

Exemples: l'exemple problématique de référence polymorphe:

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

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

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

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

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

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

En effet, l'expression `k 1` n'est pas non-expansive, et donc `f` reçoit le type simple $\alpha \to$ `int` où α est non généralisée, et donc `f` ne peut être utilisée de manière polymorphe par la suite. Pour faire "passer" cet exemple, le programmeur doit manuellement faire une étape d'eta-expansion sur `f`:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.5 Preuve de sûreté du typage

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

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

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

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

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

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

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

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

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

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

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

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

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

Cas des règles β_{fun}, β_{let}, et toutes les δ-règles du chapitre 2: ces règles ne font pas intervenir l'état mémoire, c'est-à-dire que $s_2 = s_1$ et de plus $a_1 \xrightarrow{\varepsilon} a_2$ est une réduction de mini-ML sans les références. Appliquant la proposition 2.3, on a donc $a_1 \sqsubseteq a_2$ (au sens de la section 2.3.1). De plus, on vérifie facilement par inspection des règles que si a_1 est non-expansive, alors a_2 l'est aussi:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1. *Si $\tau = \tau_1 \rightarrow \tau_2$, alors ou bien v est de la forme $\mathtt{fun}\ x \rightarrow a$, ou bien v est un opérateur op.*

2. *Si $\tau = \tau_1 \times \tau_2$, alors v est une paire (v_1, v_2).*

3. *Si τ est un type de base T, alors v est une constante c.*

4. *Si $\tau = \tau_1\ \mathtt{ref}$, alors v est une adresse mémoire $\ell \in \mathrm{Dom}(s)$.*

Démonstration: par examen des règles de typage qui peuvent s'appliquer suivant la forme de τ. $\square$

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

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

5.6 Autres approches

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

5.6.1 Les références monomorphes

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

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

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

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

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

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

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

5.6.2 Les variables faibles de Standard ML

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Exemples:

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

5.6.3 Systèmes d'effets et de régions

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

5.6.4 Variables dangereuses et typage des fermetures

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

5.7 Les exceptions

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

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

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

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

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

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

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

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

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

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

Contextes de réduction:

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

Contextes d'exceptions:

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

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

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

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

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

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

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

Ceci ajoute un constructeur $C : \tau \to \texttt{exn}$. Comme le constructeur de types `exn` n'a pas de paramètres, τ ne doit pas comporter de variables libres (voir la section 4.2).

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

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

5.8 Continuations et opérateurs de contrôle

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