Chapitre 8

Systèmes de modules

Pour conclure ce cours, nous étendons les systèmes de types vers la programmation modulaire (programmation à grande échelle). Nous suivons l'approche ML où le système de modules se présente comme un petit langage fonctionnel typé opérant au-dessus du langage de base. Le langage de base considéré ici est mini-ML, mais l'approche s'étend facilement à une classe très vaste de langages de base typés.

8.1 Un calcul de modules

La syntaxe des expressions de modules et des types de modules est la suivante :

Chemins d'accès :	p	$::= X$	identificateur de module
		$\mid p.X$	accès à un sous-module d'un chemin
Modules :	m	$::= p$	module nommé (identificateur ou chemin)
		$\mid \texttt{struct}\ d_1; \ldots; d_n\ \texttt{end}$	définition de structure
		$\mid \texttt{functor}\ (X : M) \to m$	définition de foncteur
		$\mid m_1\ m_2$	application de foncteur
		$\mid (m : M)$	contrainte de signature
Définitions :	d	$::= \texttt{let}\ x = a$	définition de valeur
		$\mid \texttt{type}\ t = \tau$	définition de type
		$\mid \texttt{module}\ X = m$	définition de sous-module
Types de modules :	M	$::= \texttt{sig}\ S_1; \ldots; S_n\ \texttt{end}$	signature
		$\mid \texttt{functor}\ (X : M) \to M'$	type de foncteur
Spécifications :	S	$::= \texttt{val}\ x : \sigma$	spécification de valeur
		$\mid \texttt{type}\ t$	spécification de type abstrait
		$\mid \texttt{type}\ t = \tau$	spécification de type manifeste
		$\mid \texttt{module}\ X : M$	spécification de sous-module

Le langage de base (expressions a, types τ et schémas σ) est mini-ML auquel on ajoute la possibilité de faire référence à des noms ou des composantes de valeurs ou de types :

Expressions :	a	$::= p.x$	composante de valeur d'une structure
		$\mid x \mid c \mid op \mid \texttt{fun}\ x \to a \mid a_1\ a_2$	
		$\mid (a_1, a_2) \mid \texttt{let}\ x = a_1\ \texttt{in}\ a_2$	comme d'habitude

Types :	$\tau ::= t$	type nommé
	$\mid p.t$	composante de type d'une structure
	$\mid \alpha \mid T \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$	comme d'habitude
Schémas de types :	$\sigma ::= \forall \alpha_1 \ldots \alpha_n.\ \tau$	

8.2 Évaluation

8.2.1 Sémantique par traduction

L'évaluation des modules s'exprime très facilement par une traduction dans mini-ML avec enregistrements : on efface les composantes de types des structures, on transforme les structures en enregistrements, et les foncteurs en fonctions. Cette traduction, notée $[\![\cdots]\!]$, est définie par :

$$\begin{array}{rcl}
[\![X]\!] & = & X \\
[\![p.X]\!] & = & [\![p]\!].X \\
[\![\texttt{struct}\ d^*\ \texttt{end}]\!] & = & L(d^*, \{\ldots x_i = x_i; \ldots; X_j = X_j; \ldots\}) \\
 & & \text{où les } x_i \text{ et } X_j \text{ sont les identificateurs de valeurs} \\
 & & \text{et de modules définis dans } d^* \\
[\![\texttt{functor}\ (X : M) \to m]\!] & = & \texttt{fun}\ X \to [\![m]\!] \\
[\![m\ m']\!] & = & [\![m]\!]\ [\![m']\!]
\end{array}$$

$$\begin{array}{rcl}
L(\varepsilon, a) & = & a \\
L(\texttt{let}\ x = a;\ d^*, b) & = & \texttt{let}\ x = a\ \texttt{in}\ L(d^*, b) \\
L(\texttt{type}\ t = T;\ d^*, b) & = & L(d^*, b) \\
L(\texttt{module}\ X = m;\ d^*, b) & = & \texttt{let}\ X = [\![m]\!]\ \texttt{in}\ L(d^*, b)
\end{array}$$

Exemple : l'expression de module

```
functor (X : sig type t ; val v : t end) →
    struct type u = X.t ; val y = X.v ; val z = (y, y) end
```

a pour traduction

```
fun X → let y = X.v in let z = (y,y) in { y = y ; z = z }
```

8.2.2 Sémantique à réduction

Le calcul de modules défini ci-dessus ne se prête pas bien à une sémantique par réduction. La raison est que la classe syntaxique p des chemins d'accès n'est pas stable par substitution (d'un identificateur de module par une expression de module). Par exemple, l'expression de type $X.t$, où X est un identificateur de module, n'est plus bien formée une fois que X est substitué par une expression de module qui n'est pas un chemin d'accès, p.ex. `struct ... end` : on obtient l'expression de type (`struct ... end`)$.t$, qui est mal formée.

Une solution simple à ce problème est d'autoriser toute expression de module m dans les projections $.x$, $.t$ et $.X$. Autrement dit, on prendrait $p ::= m$ dans la grammaire ci-dessus. Le problème de cette approche est que les expressions de types $p.t$ deviennent extrêmement compliquées : p peut être une expression de module arbitrairement complexe. En particulier, il devient difficile de décider si deux expressions de types $p.t$ et $p'.t$ représentent le même type : il faudrait réduire (évaluer) p et p' *pendant le typage* pour voir s'ils se réduisent sur des valeurs de modules identiques. Une telle évaluation pourrait ne pas terminer, rendant le typage indécidable. De manière générale, nous voulons préserver une certaine *distinction de phase* entre le typage (pendant la compilation) et l'évaluation (pendant l'exécution) ; devoir réduire des expressions pendant le typage brouille cette distinction.

Une solution intermédiaire, proposée par Harper et Lillibridge, consiste à ajouter à la classe des chemins d'accès les valeurs de modules, c'est-à-dire les expressions de modules entièrement évaluées :

Chemins d'accès :	p	$::= X \mid p.X \mid V$
Valeurs de modules :	V	$::= \texttt{struct}\ d_V^*\ \texttt{end} \mid \texttt{functor}(X : M) \to a$
Définitions entièrement évaluées : :	d_V	$::= \texttt{let}\ x = v \mid \texttt{type}\ t = \tau \mid \texttt{module}\ X = V$
Valeurs du langage de base :	v	$::= c \mid op \mid \texttt{fun}\ x \to a \mid (v_1, v_2)$

Les valeurs de modules, étant déjà entièrement réduites, se comparent simplement par égalité syntaxique. Cela permet de décider facilement si deux types $V_1.t$ et $V_2.t$ sont identiques.

Dans l'approche Harper-Lillibridge, la grammaire du langage n'est toujours pas stable par substitution $[X \leftarrow m]$ où m est une expression de module arbitraire ; mais elle est stable par substitution $[X \leftarrow V]$ où V est une valeur de module. C'est suffisant pour donner une sémantique à réduction en appel par valeur :

$$\begin{array}{rcll}
(\texttt{struct}\ d_V^*; \texttt{val}\ x = v; d_V^{*\prime}\ \texttt{end}) & \stackrel{\varepsilon}{\to} & v & \text{(proj-val)} \\
(\texttt{struct}\ d_V^*; \texttt{module}\ X = V; d_V^{*\prime}\ \texttt{end}) & \stackrel{\varepsilon}{\to} & V & \text{(proj-mod)} \\
(\texttt{functor}\ (X : M) \to m)\ V & \stackrel{\varepsilon}{\to} & m[X \leftarrow V] & \beta_{\text{functor}} \\
(V : M) & \stackrel{\varepsilon}{\to} & V & \text{(contrainte)}
\end{array}$$

Pour tenir compte du fait qu'une composante de structure peut faire référence aux noms des composantes précédentes (p.ex. `struct val x = 1 ; val y = x + 1 end`), il faut ajouter les deux règles de "propagation" de valeurs suivantes :

$$\begin{array}{rcll}
\texttt{struct}\ d_V^*; \texttt{let}\ x = v; d^*\ \texttt{end} & \stackrel{\varepsilon}{\to} & \texttt{struct}\ d_V^*; \texttt{let}\ x = v; d^*[x \leftarrow v]\ \texttt{end} & \text{(prop-val)} \\
\texttt{struct}\ d_V^*; \texttt{module}\ X = V; d^*\ \texttt{end} & \stackrel{\varepsilon}{\to} & \texttt{struct}\ d_V^*; \texttt{module}\ X = V; d^*[X \leftarrow V]\ \texttt{end} & \text{(prop-mod)}
\end{array}$$

Enfin, pour les réductions en profondeur, on utilise les contextes suivants :

Contextes de modules :

$$\begin{array}{rl}
\Gamma_m & ::= [\,] \mid \Gamma_m.X \\
& \mid \texttt{struct}\ d_V^*; \texttt{let}\ x = \Gamma_a; d^*\ \texttt{end} \\
& \mid \texttt{struct}\ d_V^*; \texttt{module}\ X = \Gamma_m; d^*\ \texttt{end} \\
& \mid \Gamma_m\ m \mid V\ \Gamma_m \mid (\Gamma : M)
\end{array}$$

Contextes d'expressions de base :

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

8.3 Règles de typage

Le typage des modules soulève plusieurs difficultés. Premièrement, il faut prendre en compte les équivalences entre types de base induites par les types manifestes dans les signatures. Par exemple, si X : `sig type` $t =$ `int end`, alors le type $X.t$ est équivalent à `int`. Dans les chapitres précédents, nous avons vu plusieurs exemples de théories équationnelles ajoutées à l'algèbre de types. La différence dans le cas des modules est que les équations ne sont pas fixées une fois pour toute, mais dépendent des types qui peuvent être donnés aux modules.

La seconde difficulté est de prendre en compte l'oubli d'informations concernant les signatures : on peut oublier la présence de certains composants ; on peut aussi oublier des égalités de types en voyant un type manifeste comme un cas particulier de type abstrait. Ceci est reflété dans une relation de sous-typage entre types de modules, avec une règle de subsomption implicite (nous ne faisons pas d'inférence de types sur le langage de modules).

La dernière difficulté est la prise en compte des dépendances entre composantes de structures, d'une part, et de l'autre entre l'argument et le résultat d'un foncteur. Par exemple, le foncteur

```
functor (X : sig type t end) → struct type u = X.t * X.t end
```

reçoit le type

```
functor (X : sig type t end) → sig type u = X.t * X.t end
```

dans lequel la signature du résultat fait intervenir le nom `X` du paramètre formel.

8.3.1 Équivalence entre types de base

L'équivalence entre types du langage de base est définie par le prédicat $E \vdash \tau_1 \approx \tau_2$. Comme dans tous les prédicats de typage de cette section, l'environnement E contient des hypothèses de typage sur les identificateurs de valeurs, de types, et de modules :

$$E = \ldots; \texttt{val}\ v : \sigma; \ldots; \texttt{type}\ t; \ldots; \texttt{type}\ t = \tau; \ldots; \texttt{module}\ X : M; \ldots$$

L'environnement E est en fait une séquence de spécifications $S_1; \ldots S_n$, tout comme l'intérieur d'une signature `sig`...`end`.

Les règles définissant le prédicat $E \vdash \tau_1 \approx \tau_2$ sont :

$$\frac{E = E_1;\ \texttt{type}\ t = \tau;\ E_2}{E \vdash t \approx \tau}\ \text{(eq-manifeste)} \qquad \frac{E \vdash p : \texttt{sig}\ S_1^*;\ \texttt{type}\ t = \tau;\ S_2^*\ \texttt{end}}{E \vdash p.t \approx \tau\{z \leftarrow p.z \mid z \text{ lié dans } S_1^*\}}\ \text{(eq-proj)}$$

$$E \vdash \tau \approx \tau\ \text{(eq-réflexivité)} \qquad \frac{E \vdash \tau_1 \approx \tau_2}{E \vdash \tau_2 \approx \tau_1}\ \text{(eq-symétrie)}$$

$$\frac{E \vdash \tau_1 \approx \tau_2 \qquad E \vdash \tau_2 \approx \tau_3}{E \vdash \tau_1 \approx \tau_3}\ \text{(eq-transitivité)}$$

$$\frac{E \vdash \tau[\alpha_i \leftarrow \tau_i] \approx \tau'}{E \vdash \forall \alpha_1 \ldots \alpha_n.\ \tau \geq \forall \beta_1 \ldots \beta_p.\ \tau'}\ \text{(schémas)}$$

La règle (eq-projection) est compliquée par le fait que les composantes de la signature affectée à p peuvent dépendre les unes des autres. Par conséquent, le type τ manifestement égal à la composante t peut faire référence à des identificateurs de types et de modules liés au début de la signature. Il n'est donc pas correct en général de dire que $E \vdash p.t \approx \tau$, car certains identificateurs de τ vont être sortis de leur portée et devenir libre. Il faut au contraire préfixer par p les identificateurs X et t liés précédemment dans la signature.

Exemple : supposons $E \vdash p : \texttt{sig type } t; \texttt{type } u; \texttt{type } v = u \times t \texttt{ end}$. Il n'est pas correct de dire $E \vdash p.v \approx u \times t$, car ce type fait référence à des identificateurs de types u et t inconnus. En revanche, nous pouvons déduire $E \vdash p.v \approx (u \times t)[t \leftarrow p.t; u \leftarrow p.u]$, c'est-à-dire $E \vdash p.v \approx p.u \times p.t$. Le type qui s'appelle u à l'intérieur de la signature est référencé à l'extérieur de la signature par $p.u$, et de même pour t.

8.3.2 Typage du langage de base

Les règles de typage du langage de base sont essentiellement celles de mini-ML sans modules, avec prise en compte de l'équivalence $\approx$ entre types.

$$\frac{E = E_1;\ \texttt{val}\ x : \sigma;\ E_2 \qquad \tau \leq \sigma}{E \vdash x : \tau} \text{ (var-inst)}$$

$$\frac{E \vdash p : \texttt{sig}\ S_1^*;\ \texttt{val}\ x : \sigma;\ S_2^*\ \texttt{end} \qquad \tau \leq \sigma\{z \leftarrow p.z \mid z \text{ lié dans } S_1^*\}}{E \vdash p.x : \tau} \text{ (val-projection)}$$

$$\frac{E \vdash a : \tau \qquad E \vdash \tau \approx \tau'}{E \vdash a : \tau'} \text{ (equiv)}$$

Pour l'accès aux composantes de valeurs de structures (règle (proj-val)), on effectue la même opération de "préfixage" par p décrite plus haut à propos de la règle (eq-proj).

Les autres règles de typage sont reprises directement de celles du chapitre 1 :

$$\frac{\tau \leq TC(c)}{E \vdash c : \tau} \text{ (const-inst)} \qquad \frac{\tau \leq TC(op)}{E \vdash op : \tau} \text{ (op-inst)} \qquad \frac{E;\ \texttt{val}\ x : \tau_1 \vdash a : \tau_2}{E \vdash (\texttt{fun}\ x \rightarrow a) : \tau_1 \rightarrow \tau_2} \text{ (fun)}$$

$$\frac{E \vdash a_1 : \tau' \rightarrow \tau \qquad E \vdash a_2 : \tau'}{E \vdash a_1\ a_2 : \tau} \text{ (app)} \qquad \frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2} \text{ (paire)}$$

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

Remarquons que pendant le typage d'une expression a, on n'ajoute pas d'hypothèses `type t`... ou `module` X : ... à l'environnement. Par conséquent, la relation d'équivalence entre types $\approx$ est constante pendant le typage d'une expression. Cela permet de reprendre des résultats généraux concernant le typage modulo une théorie équationnelle. Par exemple, l'inférence du type principal d'une expression a apparaissant comme composante de valeur d'une structure `struct` ... `val` $x = a \ldots$ `end` est possible, en utilisant l'algorithme W et l'unification modulo la relation $\approx$.

8.3.3 Sous-typage entre types de modules

Nous définissons maintenant une relation de sous-typage entre types de modules $E \vdash M_1 <: M_2$ qui reflète les deux possibilités d'oubli d'information dans les signatures mentionnées plus haut (oubli de composantes et transformation de types manifestes en types abstraits). Au passage, nous définissons également le sous-typage entre deux spécifications $E \vdash S_1 <: S_2$.

$$\frac{\rho : \{1 \ldots n\} \mapsto \{1 \ldots m\} \qquad E; S_1; \ldots; S_m \vdash S_{\rho(i)} <: S'_i \text{ for } i = 1, \ldots, n}{E \vdash (\texttt{sig } S_1; \ldots; S_m \texttt{ end}) <: (\texttt{sig } S'_1; \ldots; S'_n \texttt{ end})} \text{ (sub-sig)}$$

$$\frac{E \vdash P_2 <: P_1 \qquad E; \texttt{module } X : P_2 \vdash R_1 <: R_2}{E \vdash (\texttt{functor } (X : P_1) \to R_1) <: (\texttt{functor } (X : P_2) \to R_2)} \text{ (sub-foncteur)}$$

$$\frac{E \vdash \sigma \geq \sigma'}{E \vdash (\texttt{val } v : \sigma) <: (\texttt{val } v : \sigma')} \text{ (sub-val)}$$

$$E \vdash (\texttt{type } t) <: (\texttt{type } t) \text{ (sub-abstr-abstr)} \qquad E \vdash (\texttt{type } t = \tau) <: (\texttt{type } t) \text{ (sub-mani-abstr)}$$

$$\frac{E \vdash \tau \approx \tau'}{E \vdash (\texttt{type } t = \tau) <: (\texttt{type } t = \tau')} \text{ (sub-mani-mani)}$$

$$\frac{E \vdash \tau \approx t}{E \vdash (\texttt{type } t) <: (\texttt{type } t = \tau)} \text{ (sub-abstr-mani)}$$

$$\frac{E \vdash M <: M'}{E \vdash (\texttt{module } X : M) <: (\texttt{module } X : M')} \text{ (sub-mod)}$$

Sous-typage entre signatures

La règle (sub-sig) exprime qu'une signature $M = \texttt{sig } S_1; \ldots; S_m \texttt{ end}$ est sous-type d'une autre signature $M' = \texttt{sig } S'_1; \ldots; S'_n \texttt{ end}$ si, à chaque composante S'_i de M' (la moins précise des deux), on peut associer une composante $S_{\rho(i)}$ de M (la plus précise des deux signatures) telle que $S_{\rho(i)} <: S'_i$. Remarquons que M peut très bien contenir plus de composantes que M' ; ces composantes ne sont simplement pas associées à aucune composante de M'. Cela correspond à l'oubli de ces composantes de M, ou encore au sous-typage "en largeur" entre signatures.

La relation de sous-typage entre composantes de signatures $E \vdash S <: S'$ est définie de telle sorte que les deux composantes S et S' doivent être du même "genre" (deux spécifications de valeurs, ou deux spécifications de types, ou deux spécification de modules, mais pas une valeur et un type, par exemple) et spécifier des identificateurs de même nom (deux valeurs nommées x, mais pas une valeur nommée x et une nommée y par exemple). Par conséquent, pour déterminer l'existence de la correspondance ρ entre les composantes des deux signatures, il suffit de regarder les genres et les noms de ces composantes. Si nous supposons de plus que les composantes d'une signature ont toutes des noms distincts (on s'arrangera pour que ce soit le cas dans la règle de typage des structures), il y a au plus une correspondance ρ qui convient, et elle est entièrement déterminée par les genres et les noms des composantes des deux signatures.

Le sous-typage entre composantes de signatures $E \vdash S <: S'$ se comprend facilement en voyant les composantes de signatures comme des spécifications : S est “sous-type” de S' si toute déclaration (composante de structure) satisfaisant la spécification S satisfait aussi S'.

- Pour deux spécifications de valeurs `val` $x : \sigma$ et `val` $x : \sigma'$, il faut que le schéma σ soit plus général que σ' (i.e. σ au moins aussi polymorphe que σ'), de manière à ce que toute instance de σ soit aussi instance de σ'.
- Pour deux spécifications de modules `module` $X : M$ et `module` $X : M'$, il faut que M soit sous-type de M'.
- Pour deux spécifications de types, on a quatre cas :
  - abstrait, abstrait : (`type` t) $<:$ (`type` t) est toujours vrai ;
  - manifeste, abstrait : (`type` $t = \tau$) $<:$ (`type` t) est toujours vrai ;
  - manifeste, manifeste : (`type` $t = \tau$) $<:$ (`type` $t = \tau'$) est vrai si et seulement si τ et τ' sont équivalents ;
  - abstrait, manifeste : (`type` t) $<:$ (`type` $t = \tau'$) est vrai si on peut prouver l'équivalence $t \approx \tau'$ à partir des autres hypothèses.

Comme les composantes des signatures peuvent faire référence à des identificateurs liés par d'autres composantes, on ne peut pas les comparer dans l'environnement initial E : il faut ajouter à E des hypothèses de typage sur les identificateurs liés par les signatures. Comme tous les identificateurs liés par M' (la plus grande des deux signatures) le sont aussi par M (la plus petite), et ont a priori des types plus précis dans M, il suffit d'ajouter à E toutes les composantes $S_1 \ldots S_n$ de M. C'est pourquoi la règle (sub-sig) compare les composantes deux à deux dans $E; S_1; \ldots; S_n$.

Exemples de sous-typage entre signatures :
Oubli d'une composante :

```
(sig type t = int ; val v : t end) < : (sig type t = int end) < : (sig end)
```

En particulier, la signature vide `sig end` est supertype de toutes les signatures.
Oubli d'une égalité de type :

```
(sig type t = int ; val v : t end) < : (sig type t ; val v : t end)
```

Utilisation d'une égalité de type fournie par une composante oubliée pour sous-typer les autres composantes :

```
(sig type t = int ; val v : t end) < : (sig val v : int end)
```

Permutation de composantes indépendantes :

```
(sig type t = int ; val v : int end) < : (sig val v : int ; type t = int end)
```

Permutation de composantes dépendantes (c'est ici que la règle (sub-abstr-mani) se révèle nécessaire) :

```
(sig type t ; type u = t end) < : (sig type u ; type t = u end)
```

Exemples de non sous-typage entre signatures :
Il manque une composante dans M :

```
(sig type t = int end) ≮: (sig type t = int val v : t end)
```

Erreur sur le nom d'une composante :

```
(sig type t = int val n : t end) ≮: (sig type t = int val v : t end)
```

Désaccord sur la définition d'un type manifeste :

`(sig type t = int end)` $\not<:$ `(sig type t = bool end)`

Un type abstrait n'est pas prouvablement égal à un type manifeste :

`(sig type t end)` $\not<:$ `(sig type t = bool end)`

Sous-typage entre types de foncteurs

Comme pour les types de fonctions, le sous-typage entre types de foncteurs est contravariant en le type argument des foncteurs, et covariant en leur type résultat : $E \vdash (\texttt{functor}\ (X : P_1) \rightarrow R_1) <: (\texttt{functor}\ (X : P_2) \rightarrow R_2)$ si $E \vdash P_2 <: P_1$ (contravariance) et $E; \texttt{module}\ X : P_2 \vdash R_1 <: R_2$ (covariance). Comme les types résultat R_1 et R_2 peuvent dépendre de X, il serait incorrect de les comparer dans l'environnement E ; il faut ajouter une hypothèse sur X. On a le choix entre les hypothèses $X : P_1$ et $X : P_2$, qui sont toutes deux sémantiquement correctes. Cependant, l'hypothèse $X : P_2$ est plus précise (car P_2 est plus petit que P_1) et permet de dériver davantage de sous-typages intéressants. Par exemple, on a

$$\begin{array}{l}(\texttt{functor}\ (X : \texttt{sig type}\ t\ \texttt{end}) \rightarrow \texttt{sig type}\ t = X.t\ \texttt{end})\\ \quad <: \quad (\texttt{functor}\ (X : \texttt{sig type}\ t = \texttt{int end}) \rightarrow \texttt{sig type}\ t = \texttt{int end})\end{array}$$

car sous l'hypothèse $X : \texttt{sig type}\ t = \texttt{int end}$ on peut prouver que $X.t \approx \texttt{int}$, et donc que $\texttt{sig type}\ t = X.t\ \texttt{end} <: \texttt{sig type}\ t = \texttt{int end}$. Ce ne serait pas possible sous l'hypothèse $X : \texttt{sig type}\ t\ \texttt{end}$.

8.3.4 Typage du langage de modules

Les règles de typage pour le langage de modules sont de la forme $E \vdash m : M$ (le module m a le type de module M) et $E \vdash d^* : S^*$ (la séquence de déclarations d^* a pour spécification de types S^*).

$$\frac{E = E_1; \texttt{module}\ X : M;\ E_2}{E \vdash X : M}\ (\text{mod-var})$$

$$\frac{E \vdash p : \texttt{sig}\ S_1^*; \texttt{module}\ X : M;\ S_2^*\ \texttt{end}}{E \vdash p.X : M\{z \leftarrow p.z \mid z \text{ lié dans } S_1^*\}}\ (\text{mod-projection})$$

$$\frac{E \vdash m : M}{E \vdash (m : M) : M}\ (\text{mod-contrainte}) \qquad \frac{E \vdash m : M \qquad E \vdash M <: M'}{E \vdash m : M'}\ (\text{mod-sub})$$

Typage des structures

$$\frac{E \vdash d^* : S^*}{E \vdash (\texttt{struct}\ d^*\ \texttt{end}) : (\texttt{sig}\ S^*\ \texttt{end})}\ (\text{struct})$$

$$E \vdash \varepsilon : \varepsilon \text{ (struct-vide)} \qquad \frac{E \vdash a : \tau \qquad E;\, \texttt{val}\ v : \tau \vdash d^* : S^*}{E \vdash (\texttt{let}\ v = a;\ d^*) : (\texttt{val}\ v : \tau;\ S^*)} \text{ (struct-val)}$$

$$\frac{t \notin \mathrm{Dom}(E) \qquad E;\, \texttt{type}\ t = \tau \vdash d^* : S^*}{E \vdash (\texttt{type}\ t = \tau;\ d^*) : (\texttt{type}\ t = \tau;\ S^*)} \text{ (struct-type)}$$

$$\frac{E \vdash m : M \qquad X \notin \mathrm{Dom}(E) \qquad E;\, \texttt{module}\ X : M \vdash d^* : S^*}{E \vdash (\texttt{module}\ X = m;\ d^*) : (\texttt{module}\ X : M;\ S^*)} \text{ (struct-mod)}$$

Lorsqu'on type une struture, les définitions de types reçoivent des types manifestes, et sont donc transparentes par défaut. Par exemple, la structure

```
module S = struct type t = int ; let x = 0 end
```

reçoit la signature `sig type t = int ; val x : int end`, ou de manière équivalente `sig type t = int ; val x : t end`. Dans les deux cas, nous pouvons typer `1 + S.f 0` par exemple.

Pour rendre abstrait un type, il faut utiliser une contrainte de signature :

```
module S = (struct type t = int ; let x = 0 end : sig type t ; val x: t end)
```

Les conditions $t \notin \mathrm{Dom}(E)$ et $X \notin \mathrm{Dom}(E)$ dans les règles (struct-type) et (struct-mod) évitent des incohérences de typage lorsqu'on redéfinit un nom de type ou de module existant. Par exemple :

```
struct
  type t = int                t est équivalent à int
  let x = (1 : t)             x a le type t
  type t = bool               t est équivalent à int
  let y = (true : t)          y a le type t et x aussi
  ... if x = y then ...       x et y ont le même type t
end
```

La condition $t \notin \mathrm{Dom}(E)$ dans la règle (struct-type) et la condition $X \notin \mathrm{Dom}(E)$ dans (struct-mod) évitent ce problème, mais sont un peu trop restrictives. En particulier, on ne peut pas définir deux fois le même nom de type à des niveaux d'emboîtement de structures différents :

```
struct
  type t = int
  module X = struct type t = bool ... end
end
```

Pour traiter plus souplement les redéfinitions, on peut distinguer dans les structures les noms des composantes (qui servent à faire référence aux composantes dans les projections et ne peuvent donc pas être renommées) et les identificateurs liés par les définitions de composantes (qui servent à faire référence aux composantes dans le reste de la structure et peuvent être renommés par alpha-conversion). Ainsi, l'exemple ci-dessus devient (les identificateurs sont marqués en italiques) :

```
struct
  type t/t1 = int
  let x/x = (1 : t1)          x a le type t1
  type t/t2 = int
  let y/y = (true : t2)       y a le type t2
  ... if x = y then ...       x et y ont des types différents
end
```

Foncteurs et applications de foncteurs

$$\frac{X \notin \mathrm{Dom}(E) \qquad E;\, X : M \vdash m : M'}{E \vdash (\texttt{functor}\ (X : M) \to m) : (\texttt{functor}\ (X : M) \to M')}\ \text{(mod-foncteur)}$$

$$\frac{E \vdash m_1 : (\texttt{functor}\ (X : M) \to M') \qquad E \vdash m_2 : M}{E \vdash m_1(m_2) : M'[X \leftarrow m_2]}\ \text{(mod-appl)}$$

Dans la règle (mod-appl), le remplacement du paramètre formel X par l'argument effectif m_2 dans le type M' du résultat du foncteur assure la propagation des égalités de types à travers les applications de foncteurs. Exemple :

```
module Id : functor(X : sig type t end) → sig type t = X.t end
module E = struct type t = int end
module B = Id(E)
```

Le type de `B` est (**sig type** `t` = `X.t` **end**)[`X` ← `E`], c'est-à-dire `sig type t = E.t`. Il s'ensuit que `B.t` ≈ `E.t` par la règle (eq-projection), et comme d'autre part `E.t` ≈ **`int`** d'après la signature de `E`, on a bien `B.t` ≈ **`int`** comme attendu.

Dans le cas où le paramètre formel X n'apparaît pas dans le type M', on retrouve la règle d'application de fonction habituelle :

$$\frac{E \vdash m_1 : (\texttt{functor}\ (X : M) \to M') \qquad E \vdash m_2 : M \qquad X \notin \mathcal{L}(M')}{E \vdash m_1(m_2) : M'}\ \text{(mod-appl-nondep)}$$

Si l'argument m_2 du foncteur n'est pas un chemin d'accès et que X apparaît dans le type M' du résultat du foncteur, la substitution $M'[X \leftarrow m_2]$ n'est pas un type bien formé. La règle (mod-appl) n'est alors pas applicable directement. Exemple :

```
module B = Id(struct type t = int end)
```

Pour appliquer (mod-appl), il faudrait dans le type **sig type** `t` = `X.t` **end** remplacer `X` par `struct type t = int end`, obtenant le type mal formé

```
sig type t = (struct type t = int).t end
```

Une solution simple à ce problème est de toujours nommer les arguments de foncteurs qui ne sont pas des chemins :

`module A = F(`m`)` $\quad\Rightarrow\quad$ `module B = ` m `; module A = F(B)`

Une solution plus subtile consiste à utiliser la règle de sous-typage (mod-sub) pour faire disparaître la dépendance entre argument et résultat dans le type du foncteur. Dans l'exemple `Id` ci-dessus, remarquons que

```
functor(X : sig type t end) → sig type t = X.t end <:
       functor(X : sig type t = int end) → sig type t = int end
```

Le super-type est un type non dépendant : le paramètre formel X n'apparaît plus dans le type du résultat. On peut donc construire la dérivation suivante :

$$\frac{\dfrac{E \vdash \texttt{Id} : \texttt{functor(X : sig type t end)} \to \texttt{sig type t} = \texttt{X.t end}}{E \vdash \texttt{Id} : \texttt{functor(X : sig type t} = \texttt{int end)} \to \texttt{sig type t} = \texttt{int end}} \quad E \vdash (\texttt{struct type t} = \texttt{int end}) : (\texttt{sig type t} = \texttt{int end})}{E \vdash \texttt{Id(struct type t} = \texttt{int end)} : (\texttt{sig type t} = \texttt{int end})}$$

La règle de renforcement

Soit p un chemin désignant une structure avec un type t abstrait :

$$p : \texttt{sig } \ldots ; \texttt{type } t ; \ldots \texttt{ end}$$

L'implémentation du type $p.t$ est inconnue. Donc, le type $p.t$ est a priori incompatible avec tout autre type. Cependant, la composante t de la structure désignée par p n'est pas, elle, n'importe quel type : nous savons qu'elle est forcément égale à $p.t$. Le chemin p a donc aussi le type suivant :

$$p : \texttt{sig } \ldots ; \texttt{type } t = p.t ; \ldots \texttt{ end}$$

Plus généralement, nous appelons renforcement cette opération de remplacement de types abstraits par des types manifestement égaux à eux-même dans la signature d'un chemin. Si M est un type de module et p un chemin, nous définissons le type renforcé M/p comme suit :

$$\begin{array}{rcl}
(\texttt{sig } d_1 ; \ldots ; d_n \texttt{ end})/p & = & \texttt{sig } d_1/p ; \ldots ; d_n/p \texttt{ end} \\
(\texttt{val } v : \tau)/p & = & \texttt{val } v : \tau \\
(\texttt{type } t)/p & = & \texttt{type } t = p.t \\
(\texttt{type } t = \tau)/p & = & \texttt{type } t = \tau \\
(\texttt{module } X : M)/p & = & \texttt{module } X : (M/(p.X)) \\
(\texttt{functor}(X : M) \to M')/p & = & \texttt{functor}(X : M) \to M'
\end{array}$$

La règle de typage correspondant à l'opération de renforcement est alors :

$$\frac{E \vdash p : M}{E \vdash p : (M/p)} \ (\text{mod-renforcement})$$

Le renforcement est nécessaire dans plusieurs situations. La première est lorsqu'on prend une vue restreinte d'un module contenant des types abstraits. Supposons par exemple

```
E : sig type t ; val x: t ; val y: t → t end
```

Nous voulons prendre une vue de E qui cache la composante y. Si nous faisons

```
module B = (E : sig type t ; val x: t end)
```

nous cachons y, mais rendons le type B.t abstrait, et donc différent de E.t. Pour conserver la compatibilité entre B.t et E.t, il faut écrire

```
module B = (E : sig type t = E.t ; val x: t end)
```

Pour typer cette contrainte, il faut montrer que le type de `E` est sous-type de `sig type t = E.t ; val x : t end`. Si l'on part de `E : sig type t; ...`, ce n'est pas possible. Il faut renforcer au préalable le type de `E`, obtenant `sig type t = E.t; ...`, pour montrer que la contrainte de signature est bien typée.

La seconde utilité de la règle de renforcement est pour détecter la propagation des types à travers les foncteurs. Reprenons le foncteur identité vu plus haut :

```
module Id = functor(X: sig type t end) → struct type t = X.t end
```

Sans règle de renforcement, le type le plus précis que l'on puisse lui attribuer est

```
Id : functor(X: sig type t end) → sig type t end
```

et ce type ne traduit pas que la composante `t` du résultat est égale à la composante `t` de l'argument. En appliquant la règle de renforcement au type de `X`, nous obtenons le type plus précis :

```
Id : functor(X: sig type t end) → sig type t = X.t end
```

Une troisième application du renforcement est pour vérifier des contraintes de partage. Un foncteur à plusieurs arguments peut avoir à imposer que des composantes de types de ses arguments soient égales afin que le corps du foncteur soit bien typé. C'est ce qu'on appelle une contrainte de partage entre les arguments d'un foncteur, et cette contrainte de partage s'exprime facilement avec des types manifestes en position d'argument de foncteur. Par exemple, voici un foncteur qui compose des opérations entre types abstraits :

```
module Compose =
  functor(X : sig type t ; val f: t → t end) →
  functor(Y : sig type t = X.t ; val f: t → t end) →
    struct
      type t = X.t
      let f = fun x → X.f(Y.f(x))
    end
```

Grâce à la spécification `Y : sig type t = X.t ; ...`, le corps du foncteur est typé sous l'hypothèse `X.t` = `Y.t`, et cela assure que la composition des fonctions `X.f` et `Y.f` est bien typée. Si on avait déclaré `Y : sig type t ; ...`, `X.t` et `Y.t` auraient été considérés comme incompatibles, et la définition de la fonction `f` comme mal typée.

La vérification des contraintes de partage lors de l'application du foncteur s'effectue automatiquement par les règles de typage de l'application et de sous-typage. Par exemple, supposons

```
module A : sig type t = τ1 ; val f: t → t end
module B : sig type t = τ2 ; val f: t → t end
```

Le typage de l'application `Compose(A)(B)` se déroule comme suit. Tout d'abord, on type `Compose(A)`, obtenant

```
Compose(A) :
  functor (Y : sig type t = A.t end ; val f : t → t end) →
    sig type t = A.t ;  val f : t → t end
```

Maintenant, l'application de `Compose(A)` à `B` est bien typée si et seulement si `B` a le type de l'argument de `Compose(A)`, c'est-à-dire ssi τ_2 est équivalent à `A.t`, ou de manière équivalente ssi τ_1 et τ_2 sont équivalents.

Dans le cas où `B.t` est abstrait mais `A.t` est prouvablement égal à `B.t`, la règle de renforcement est nécessaire pour vérifier la contrainte de partage. Supposons

```
module A : sig type t = B.t ; val f: t → t end
module B : sig type t ; val f: t → t end
```

Pour que `Compose(A)(B)` soit bien typé, il faut que `B` possède la signature `sig type t = B.t ; val f : t → t`, et cela nécessite d'appliquer la règle de renforcement au type de `B`.

Pour aller plus loin

Sémantique statique à stamps : la définition de Standard ML utilise une approche différente du typage des modules de ML, s'appuyant sur des *stamps* uniques pour représenter l'identité des types abstraits. Cette approche est présentée dans *A type discipline for program modules*, R. Harper, R. Milner, M. Tofte, actes TAPSOFT 87, LNCS 250, et étendue au langage Standard ML tout entier dans *The definition of Standard ML*, R. Milner, M. Tofte, R. Harper et D.MacQueen, MIT Press, 1997. Une présentation plus simple ainsi qu'une preuve d'équivalence entre la sémantique statique à stamps et les règles de typage présentées dans ce cours se trouve dans *A syntactic theory of type generativity and sharing*, X. Leroy, Journal of Functional Programming 6(5), 1996, `http ://pauillac.inria.fr/~xleroy/publi/syntactic-generativity.dvi.gz`.

Implémentation du typage des modules et adaptation à d'autres langages de base : le rapport technique suivant développe une implémentation simple d'un typeur pour le langage des modules qui est paramétrée par un typeur pour le langage de base : *A modular module system*, X. Leroy, rapport de recherche 2866, INRIA, 1996, `http ://pauillac.inria.fr/~xleroy/publi/modular-modules.ps.gz`.

Preuve de sûreté du typage : la thèse suivante étudie les propriétés formelles d'un calcul de modules proche de celui présenté dans ce chapitre, en particulier la sûreté du typage : *Translucent sums : a foundation for higher-order module systems*, M. Lillibridge, PhD thesis, Carnegie-Mellon University, 1997, `http ://www.research.digital.com/SRC/personal/Mark_Lillibridge/Papers/Thesis/thesis.PS`.

Exercices

Exercice 8.1 *(*)/(**) Montrer que la relation de sous-typage* $<:$ *est réflexive (*) et transitive (**).*

Exercice 8.2 *(*) On dit que deux signatures* S_1 *et* S_2 *sont équivalentes dans l'environnement* A*, et on note* $A \vdash S_1 \approx S_2$*, si* $A \vdash S_1 <: S_2$ *et* $A \vdash S_2 <: S_1$*. Donner une axiomatisation directe de* $A \vdash S_1 \approx S_2$ *sous forme de règles d'inférence.*

Exercice de programmation 8.1 *Implémenter la relation de sous-typage* $<:$*.*

Exercice 8.3 *(***) Montrer qu'on ne perd pas d'expressivité si on restreint les règles de projection (eq-projection), (val-projection) et (mod-projection) au cas où le type de la composante extraite ne dépend pas du début* S_1^* *de la signature :*

$$\frac{E \vdash p : \texttt{sig}\ S_1^*;\ \texttt{type}\ t = \tau;\ S_2^*\ \texttt{end} \qquad \mathcal{L}(\tau) \cap \mathcal{B}(S_1^*) = \emptyset}{E \vdash p.t \approx \tau}\ \text{(eq-projection')}$$

$$\frac{E \vdash p : \texttt{sig } S_1^*;\ \texttt{val } x : \sigma;\ S_2^* \texttt{ end} \qquad \mathcal{L}(\sigma) \cap \mathcal{B}(S_1^*) = \emptyset \qquad \tau \leq \sigma}{E \vdash p.x : \tau} \text{ (val-projection')}$$

$$\frac{A \vdash p : \texttt{sig } S_1^*;\ \texttt{module } X : M;\ S_2^* \texttt{ end} \qquad \mathcal{L}(M) \cap \mathcal{B}(S_1^*) = \emptyset}{A \vdash p.X : M} \text{ (mod-projection')}$$

On a noté $\mathcal{L}(\tau)$ les identificateurs de types et de modules libres dans le type τ et $\mathcal{B}(S^*)$ les identificateurs de types et de modules liés par la signature S^*. (Indication : on peut toujours appliquer la règle de renforcement à p avant de typer la projection.)