Implémentation d'un système de modules évolué en Caml-Light

François Pottier

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Implémentation d'un système de modules évolué en Caml-Light

François Pottier

N° 2449

Janvier 1995

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

ISSN 0249-6399

Implémentation d'un système de modules évolué en Caml-Light

François Pottier*

Programme 2 — Calcul symbolique, programmation et génie logiciel

Projet Cristal

Rapport de recherche n°2449 — Janvier 1995 — 69 pages

Résumé : Ce mémoire décrit la conception et l'implémentation d'un langage de modules évolué en Caml-Light. Nous décrivons d'abord ce que nous entendons par langage de modules, et l'intérêt qu'un tel langage présente pour la programmation à moyenne ou grande échelle. Nous expliquons ensuite en quoi le précurseur en la matière, SML, souffre de limitations graves vis-à-vis de la compilation séparée. Nous exposons un système similaire mais mieux adapté à la compilation séparée, et en même temps simplifié. Enfin, certains problèmes d'implémentation inattendus sont évoqués.

Mots-clé : Systèmes de modules, programmation modulaire, compilation séparée, foncteurs, typage, Caml, Standard ML

(Abstract: pto)

*École Normale Supérieure et INRIA Rocquencourt, projet Cristal. Francois.Pottier@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

An implementation of an advanced module system in Caml Light

Abstract: This report describes the design and implementation of a powerful module language on top of Caml-Light. We first describe the meaning of a module language, and show why such a language is interesting and necessary for medium- or large-scale programming. Then, we explain why the well-known SML module language does not allow separate compilation. We propose a similar system, better suited to separate compilation, and simpler. Finally, some interesting and unexpected implementation issues are discussed.

Key-words: Module systems, modular programming, separate compilation, functors, type-checking, Caml, Standard ML

1 Motivations

1.1 Modularité et compilation séparée

La décomposition en modules d'un programme consiste à le subdiviser en un ensemble d'unités logiques, appelées *modules*, aussi indépendantes que possible, et dont les relations sont clairement définies. Ce procédé permet d'écrire, de comprendre ou de mettre à jour chaque module séparément, et facilite donc grandement la mise au point de systèmes de grande taille.

La compilation séparée, quant à elle, consiste à découper le programme en *unités de compilation* (chacune correspondant habituellement à un fichier) de façon à pouvoir typer et compiler individuellement chacune de ces unités.

Ces deux procédés sont nécessaires dès que les programmes atteignent une certaine envergure. L'organisation d'un programme en modules permet de mettre sa structure en évidence, donc de le rendre plus compréhensible et plus aisé à maintenir. Son découpage en unités de compilation permet de réduire les besoins en mémoire du compilateur. De plus, il provoque un "cloisonnement" qui permet de réduire la quantité de code à recompiler après chaque modification, et permet donc un gain de temps très appréciable.

Ces deux concepts, bien que distincts, sont souvent confondus dans les langages les plus courants (Modula 2, Turbo Pascal, Caml Light 0.6, etc.). Cependant, cette identification est indésirable : elle pousse le concepteur à limiter le nombre de modules afin de ne pas avoir à manier trop de fichiers, et donc nuit à l'organisation logique du programme. De plus cette identification n'a de sens que lorsque l'on travaille uniquement sur des structures ; l'introduction des foncteurs rend impossible toute association directe entre modules et fichiers.

1.2 Le système de SML

Le langage que l'on considère actuellement comme proposant le système de modules le plus puissant est Standard ML [MQ86]. Il offre en effet, au-dessus de ML, un véritable "langage de modules". Celui-ci se présente sous la forme d'un lambda-calcul simple. Il permet d'exprimer des modules (appelés *structures*) ainsi que leurs types (*signatures*) ; il dispose également de modules paramétrés, sous la forme de fonctions des modules dans les modules (*foncteurs*). Une signature est une série de déclarations ; c'est une spécification, ou encore une interface, de module.

Un tel système est donc d'une puissance et d'une expressivité largement supérieures à celle des systèmes plus simplistes évoqués plus haut. SML, cependant, présente un grave inconvénient : il est conçu comme un système interactif, où l'utilisateur est censé implémenter les modules dans l'ordre où ils seront utilisés. Il est donc mal adapté à la compilation séparée.

On pourrait tenter de résoudre ce problème en faisant un usage intensif de foncteurs. Supposons par exemple que nous voulions écrire et compiler un module `Set` dépendant d'un module `Tree`, alors que ce dernier n'est pas encore implémenté. Nous pourrions écrire `Set` sous forme d'un foncteur dont l'argument `T` aurait la signature prévue pour `Tree`. Ceci permettrait de compiler `Set` en disposant uniquement de la signature (i.e. l'interface) de

`Tree`. Plus tard, une fois `Tree` implémenté, nous pourrions lui appliquer le foncteur `Set` et obtenir le résultat voulu.

Malheureusement cette approche ne fonctionne pas en Standard ML. En effet, les règles de typage de SML sont telles que connaître la signature Σ d'une structure `S` ne suffit pas à vérifier la correction du code qui dépend de `S`. Ceci est dû au fait que les signatures en Standard ML sont *transparentes* : elles ne cachent pas l'implémentation sous-jacente. Par exemple, supposons que Σ définisse un type `t`. Même si aucune indication n'est donnée sur `t`, un second module `S'` peut faire l'hypothèse que `S.t` est implémenté par `int`. `S'` sera correctement typé si et seulement si la structure `S` définit effectivement `t` par `int`. Il est donc impossible de typer `S'` tant que `S` n'aura pas été implémenté.

Cette transparence des signatures est bien sûr voulue : si tous les types étaient considérés comme abstraits, la compilation séparée serait aisée mais la puissance du système serait fort réduite : il serait impossible d'ajouter de nouvelles opérations à un type existant.

1.3 Types manifestes

Xavier Leroy a proposé un système inspiré de celui de SML, mais modifié de façon à permettre la compilation séparée [XL94a]. Il est basé sur la notion de déclaration de type *manifeste* : une signature peut déclarer un type `t` soit comme abstrait, soit comme manifeste. Dans le second cas la déclaration est de la forme `type t == `τ et annonce expressément que `t` est implémenté par l'expression de type τ. La transparence des signatures est abandonnée ; une signature exprime donc l'ensemble des propriétés de typage de la structure qu'elle représente, ce qui autorise la compilation séparée.

Ce système est moins expressif que celui de SML, mais outre la compilation séparée, il offre des règles de typages très nettement simplifiées. Le typage est basé sur une algèbre de termes, c'est-à-dire d'objects syntaxiques, alors que celui de SML fait appel à des objets sémantiques complexes.

1.4 Implémentation

Mon travail a consisté à étudier ce système et à l'intégrer au compilateur Caml Light existant. La tâche principale était constituée par la réécriture du système de modules. Les modifications principales concernent le typeur, qui a été étendu au langage de modules. L'analyseur syntaxique et la partie avant du compilateur ont également dû être adaptés. Enfin, certains problèmes ont été soulevés qui n'apparaissaient pas dans la description théorique du système, du fait des nombreuses caractéristiques de Caml Light laissées de côté dans l'étude théorique. Par exemple, la compilation des exceptions a due être revue pour pouvoir partager le code des foncteurs ; des modifications ont été apportées pour supporter les primitives C, les directives de compilation (`#open` et `#input`), etc.

2 Le système de types

2.1 Le système de SML et la transparence

Le langage de modules de SML peut être considéré comme un langage fonctionnel typé simple, où les structures correspondraient aux valeurs, les foncteurs aux fonctions, et les signatures aux types. Cependant, il diffère d'un langage typé classique en un point important : connaître la signature d'une structure `S` ne suffit pas à typer les expressions où apparaît `S` ; la valeur effective de `S` est parfois nécessaire.

Prenons l'exemple d'une structure `S` ayant la signature `Set`, donnée ci-dessous :

```
signature Set = sig
  type 'a t
  val empty : 'a t
  val member : 'a -> 'a t -> bool
  val add : 'a -> 'a t -> 'a t
  val elements : 'a t -> 'a list
end;
```

et considérons l'expression :

```
(S.member 3 [3])
```

Cette application est bien typée si le type `'a S.t` est implémenté par `'a list`. Au contraire, si `'a S.t` est implémenté par des arbres binaires, elle est incorrecte.

Cet exemple montre qu'une signature ne reflète pas toutes les propriétés de type de la structure correspondante ; le processus de typage a besoin d'extraire des informations de type de la structure elle-même. C'est pourquoi les déclarations de type dans une signature SML sont dites *transparentes* : même si la déclaration du type `t` ne donne aucune information sur son implémentation, celle-ci est accessible, et les utilisateurs de `S` peuvent en tirer parti.

Cela ne pose pas de problème dans un système interactif, où il faut définir la structure `S` avant de pouvoir la mentionner ; le compilateur est alors assuré d'avoir accès à `S` s'il rencontre une expression qui lui fait référence. Par contre, dans le cadre de la compilation séparée, on désire pouvoir écrire un module dépendant de `S` avant d'avoir implémenté celle-ci. La transparence des signatures interdit donc la compilation séparée.

La transparence est cependant un aspect important du système de modules de SML, et lui confère une grande partie de sa puissance. Elle permet en effet d'ajouter de nouvelles opérations à un type existant. Par exemple, la structure `S` ne dispose pas de primitive pour l'union. Il serait donc naturel de vouloir construire une structure `S'`, basée sur `S` et lui ajoutant une fonction `union`. Pour ce, nous pouvons définir une nouvelle signature

```
signature Set'= sig
  type 'a t
  val empty : 'a t
  val member : 'a -> 'a t -> bool
```

```
  val add : 'a -> 'a t -> 'a t
  val elements : 'a t -> 'a list
  val union : 'a t -> 'a t -> 'a t
end;
```

puis un foncteur

```
functor AddUnion (S : Set) : Set' =
  struct
    type 'a t = 'a S.t
    val empty = S.empty
    val member = S.member
    val add = S.add
    val elements = S.elements
    fun union s1 s2 =
      list_it S.add s2 (S.elements s1)
  end;
```

Il ne reste plus qu'à écrire explicitement l'application

```
structure S' = AddUnion(S);
```

Nous avons à présent défini une nouvelle structure comportant toutes les opérations précédentes, plus l'union, construite à l'aide de celles-ci. Or le foncteur `AddUnion` a pour signature résultat `Set'`, qui déclare un type abstrait `t`. L'examen seul des signatures ne nous permet donc pas de démontrer l'équivalence `S.t` = `S'.t`. Ainsi, si les signatures étaient opaques, `S'.t` serait considéré comme un type abstrait nouveau. Les ensembles créés par `S` et ceux créés par `S'` ne seraient donc pas compatibles. Cette restriction deviendrait rapidement insupportable. Grâce à la transparence des signatures, au contraire, le compilateur est autorisé à analyser la définition même du foncteur `AddUnion` et à en déduire l'équation voulue. La transparence est donc primordiale pour l'extension de structures existantes par de nouvelles opérations.

Prenons comme second exemple le cas d'un programme travaillant sur des types munis d'un ordre total. Il définit pour cela la signature

```
datatype order = Less | Equal | Greater;
signature Order = sig
  type t
  val cmp: t -> t -> order
end;
```

Toute structure de signature `Order` implémente un type totalement ordonné. Nous pouvons construire une telle structure autour du type de base `int` comme suit :

```
structure intOrder : Order =
```

```
  struct
    type t = int
    fun cmp i1 i2 =
      if i1 = i2 then Equal
      else if i1 < i2 then Less
      else Greater
  end
```

Grâce au fait que la spécification de `t` dans la signature `Order` est transparente, le type `intOrder.t` est compatible avec le type `int`. Ceci rend possible l'écriture d'expressions telles que

```
(intOrder.cmp 4 5)
```

Sans cela, `intOrder.t` serait un type abstrait et la structure `intOrder` serait inutilisable, puisqu'on n'aurait aucun moyen de construire des valeurs de ce type.

Enfin, la transparence joue également un rôle important dans l'écriture des foncteurs. Considérons le foncteur suivant, qui à un type ordonné associe un ordre sur les listes d'éléments de ce type :

```
functor listOrder (base : Order) : Order =
  struct
    type t = base.t list;
    fun cmp [] [] = Equal
      | cmp [] _ = Less
      | cmp _ [] = Greater
      | cmp (h1::t1) (h2::t2) =
  case base.cmp h1 h2 of
    Equal => cmp t1 t2
  | c => c
  end
```

Nous pouvons à présent définir aisément un ordre sur les listes d'entiers par

```
structure intListOrder = listOrder(intOrder);
```

Le type `intListOrder.t` est compatible avec `int list`, ce qui permet d'appliquer directement la fonction `intListOrder.cmp` à des listes d'entiers. Cependant, le type du foncteur listOrder est `functor (base : Order) : Order`. Ce type seul ne permet pas d'inférer l'équivalence entre `intListOrder.t` et tt int list. Le rôle de la transparence, encore une fois, est essentiel ; sans elle, le type `intListOrder.t` serait abstrait, et le foncteur `listOrder` serait inutilisable.

2.2 Types manifestes

Considérer les déclarations de type dans les signatures comme transparentes confère une grande puissance au langage, mais empêche la compilation séparée. Les considérer comme opaques est par trop restrictif. Il convient donc de rechercher une solution intermédiaire.

Dans le système que nous avons implémenté, deux sortes de déclarations de type sont possibles dans une signature. D'abord, la déclaration de type abstrait, de la forme `type t`, qui est considérée comme opaque : `t` est alors incompatible avec tout autre type. Ensuite, la déclaration de type *manifeste*, de la forme `type t ==` τ, qui déclare `t` et annonce que celui-ci est compatible avec l'expression de type τ. La déclaration de types concrets est également possible ; elle s'apparente en fait du point de vue théorique à la déclaration de types abstraits, car un type concret (enregistrement ou somme) peut être considéré comme un type abstrait muni d'opérations de construction et de décomposition adéquates.

Dans ce système, les signatures deviennent suffisamment expressives pour représenter toute l'information de type associée aux structures, et sont donc suffisantes pour effectuer la vérification des types. Reprenons les exemples précédents (cette fois en syntaxe Caml Light). La structure

```
module intOrder = struct
  type t == int;;
  let cmp i1 i2 =
    if i1 = i2 then Equal
    else if i1 < i2 then Less
    else Greater;;
end;;
```

se verra maintenant attribuer la signature

```
sig
  type t == int;;
  value cmp : t -> t -> order;;
end
```

Cette signature seule est suffisante pour déterminer que les types `intOrder__t` et `int` sont compatibles. Il serait donc possible, étant donnée cette signature, d'écrire et de typer des modules dépendant de `intOrder`, sans avoir accès à sa définition.

Les types manifestes sont également utiles lors de la définition de foncteurs. Nous pouvons écrire l'équivalent du foncteur `listOrder` ci-dessus comme suit :

```
module listOrder = functor (base : Order) struct
  type t == base__t list;;
  let cmp i1 i2 = ... ;;
end;;
```

Le type inféré pour `listOrder` sera

```
functor (base : Order) sig
  type t == base__t list;;
  value cmp : t -> t -> order;;
end
```

La signature résultat exprime clairement la relation entre `t` et `base_t`. Notons qu'il s'agit là d'un type dépendant : le *type* du résultat dépend de la *valeur* de l'argument.

Nous pouvons maintenant définir, comme précédemment,

```
module intListOrder = listOrder intOrder;;
```

Notons que la signature argument du foncteur `listOrder` n'est pas identique à la signature de la structure `intOrder`, puisque l'une déclare `t` abstrait tandis que l'autre le déclare manifeste. Un jeu de règles d'inclusion entre signatures permet cependant de considérer la seconde comme moins générale que la première (qui donne moins d'information sur `t`), et par conséquent autorise l'application.

Pour obtenir la signature de `intListOrder`, on substitue `intOrder` à `base` dans la signature résultat du foncteur ; il s'agit là de la règle classique pour les types dépendants. On obtient

```
sig
  type t == intOrder__t list;;
  value cmp : t -> t -> order;;
end
```

Par ailleurs, la signature de `intOrder` indique que `intOrder_t` est compatible avec `int`. On en déduit que `intListOrder_t` est compatible avec `int list`. Le résultat obtenu est donc semblable à celui donné par les règles de typage de SML, à la différence près qu'il nous a suffi d'utiliser les signatures pour le calculer, tandis que SML a également recours à la définition des structures elles-mêmes.

2.3 Exprimer les contraintes de partage

Un aspect important du système de modules de SML, que nous n'avons pas encore évoqué jusqu'ici, est constitué par les contraintes de partage (*sharing constraints*). Celles-ci se placent parmi les paramètres d'un foncteur, et spécifient des relations entre ceux-ci. Ces relations doivent être vérifiées par les arguments effectifs du foncteur pour qu'une application soit correctement typée. Par exemple, un foncteur de la forme

```
functor F
  (structure S1 : sig type t; ... end
   structure S2 : sig type t; ... end
   sharing type S1.t = S2.t) ...
```

ne peut être appliqué qu'à des structures `S1` et `S2` telles que les types `S1.t` et `S2.t` soient identiques.

Notre système ne supporte pas les contraintes de partage. Cependant, il est possible d'exprimer les contraintes de partage de types à l'aide de déclarations de type manifestes. Par exemple, l'équivalent du foncteur ci-dessus peut s'écrire :

```
module F = functor
  (S1 : sig type t;; ... end;
   S2 : sig type t == S1__t;; ... end) ...
```

Les types manifestes, de prime abord, semblent moins puissants. Les contraintes qu'ils imposent sont en effet asymétriques, puisqu'elles sont de la forme *identificateur* = *expression*, tandis que les contraintes SML s'écrivent *identificateur long* = *identificateur long* (par exemple `p.t = q.r.t`). De plus, les premières sont locales (une contrainte sur le type `t` doit accompagner la signature qui le déclare) tandis que les secondes offrent plus de souplesse. Cependant cette différence est purement syntaxique ; on constate facilement que les contraintes de type à la SML peuvent être traduites de façon systématique (si $\{t_0 \dots t_n\}$ forment une classe d'équivalence au sens de contraintes de SML, on peut les renuméroter de façon à ce que leur indice corresponde à leur ordre de déclaration dans la liste d'arguments du foncteur ; il suffit alors de déclarer t_0 comme abstrait et $t_{i+1} == t_i$).

SML autorise également des contraintes de partage portant sur les structures elles-mêmes. Ces contraintes n'ont pas d'équivalent dans notre système. Leur absence permet heureusement une simplification notable du système de types ; SML doit en effet être capable de décider statiquement de l'égalité de deux structures, ce qui impose l'usage d'un système de "cachets" (*stamps*) fort complexe.

3 Formalisation

3.1 Syntaxe

Pour présenter les règles de typage, nous allons utiliser un sous-ensemble du langage, afin de simplifier l'exposé. En particulier, dans le langage décrit ci-dessous, les foncteurs n'acceptent qu'un argument, et les types ne sont pas paramétrés. Les déclarations de types concrets sont remplacées par une déclaration `datatype` générique.

Le langage de base est le Caml-Light classique, à la différence près que les noms de variables, de types, de constructeurs et de labels peuvent maintenant être des chemins d'accès arbitrairement longs.

Chemin d'accès :

p	$::= x$	valeur présente dans l'environnement
	$\mid p.x$	accès à une structure

Expressions :

e	$::= p$	nom de valeur
	$\mid 0 \mid 1 \mid e + e \mid \dots$	expressions usuelles

Expressions de type :

T	$::= p$	nom de type
	$\mid \alpha \mid T \rightarrow T \mid T * T$	expressions usuelles

Expressions du langage de modules :

m	$::= p$	nom de structure
	$\mid$ `struct` s `end`	structure
	$\mid$ `functor` $(x : M)\, m$	foncteur
	$\mid m(m)$	application de foncteur
	$\mid (m : M)$	contrainte de type

Corps d'une structure :

s	$::= \varepsilon$	
	$\mid s_c;\ s$	liste d'éléments

Eléments de structure :

s_c	$::=$ `let` $v = e$	liaison d'une valeur
	$\mid$ `datatype` t	création d'un nouveau type
	$\mid$ `type` $t = T$	définition de type (par abréviation)
	$\mid$ `module` $x = m$	liaison d'un module
	$\mid$ `modtype` $w = M$	définition d'un type de module

Types de modules :

M	$::= p$	nom de type de module
	$\mid$ `sig` S `end`	signature
	$\mid$ `functor` $(x : M)\, M$	type de foncteur (dépendant)

Corps d'une signature :

S	$::= \varepsilon$	
	$\mid S_c;\ S$	liste d'éléments

Eléments de signature :

S_c	$::=$ `value` $v : T$	déclaration de valeur
	$\mid$ `type` t	déclaration de type abstrait
	$\mid$ `type` $t == T$	déclaration de type manifeste
	$\mid$ `module` $x : M$	déclaration de module
	$\mid$ `modtype` w	déclaration d'un type de module abstrait
	$\mid$ `modtype` $w = M$	déclaration d'un type de module manifeste

Notons que l'accès aux composants d'une structure se fait uniquement au moyen de chemins. Aucune opération générale de projection (de la forme $m.x$, où m serait une expression de modules quelconque) n'est définie. En effet, afin de gérer correctement les types abstraits, toute création de structure doit générer de nouveaux types. Par conséquent, si la structure résultat d'un foncteur f définit un type abstrait t, deux applications de f retourneront deux types abstraits incompatibles. Ceci ôte son sens à l'expression $f(m).t$.

Le langage de modules forme une couche supplémentaire au-dessus du langage de base. Les structures comme les foncteurs y sont des valeurs de première classe, c'est-à-dire qu'ils peuvent être arguments d'un foncteur. Les déclarations de type de modules sont traitées

de façon similaire aux déclarations de type usuelles: ils peuvent être déclarés abstraits ou manifestes. Le fait de pouvoir déclarer des types de modules comme faisant partie d'une signature augmente la puissance du système (il est par exemple possible d'écrire des foncteurs polymorphes), mais ses répercussions ne sont pas encore bien comprises et pourraient compromettre la décidabilité du typage [HL94]. La correction de notre algorithme n'est pas prouvée.

3.2 Typage à base d'objets syntaxiques

Ce système de typage est le plus simple : les types utilisés pour la vérification s'identifient aux expressions de type. Les règles d'inférence de type sont donc assez directes et aisées à comprendre. Elles sont données par la figure 1.

Ces règles sont basées sur l'équivalence des noms (c'est-à-dire des chemins) ; deux types ayant le même nom sont automatiquement considérés comme identiques. Ceci pose un problème de renommage (*rebinding*) ; rien n'empêche en effet l'utilisateur de redéfinir une structure comme suit :

```
module x = (struct type t = int;; let v = 3;; end
 : sig type t;; val v : t;; end);;
let u = x__v;;
module x = (struct type t = bool;; let v = true;; end
 : sig type t;; val v : t;; end);;
let w = x__v;;
```

Ici `u` et `w` ont tous deux le type `x__t`, et seront donc (à tort) considérés comme compatibles. Ce problème est résolu en renommant chaque variable, de façon interne, de sorte que chaque variable soit liée une fois et une seule. Ce renommage n'est pas décrit dans les règles de la figure 1, afin de les simplifier. Cette contrainte est fondamentalement liée à l'usage de noms pour le typage, et disparaîtra donc dans le système à base de stamps présenté plus loin.

Les règles 1 à 6 décrivent la façon dont est analysée une structure. Comme on le peut constater, l'identification entre objets syntaxiques et types rend certaines étapes immédiates. Il est cependant nécessaires de vérifier que les types entrés par l'utilisateur sont bien formés (c'est le sens de la prémisse "$E \vdash M$ type de module"). Certaines règles nécéssitent des jugements sur le langage de base ("$E \vdash e : T$" et "$E \vdash T$ expression de type") ; ceux-ci sont obtenus de façon classique. On notera que les structures sont des produits dépendants, c'est-à-dire qu'un champ peut faire référence aux précédents ; en conséquence, l'environnement doit être enrichi au fur et à mesure du typage de la structure.

La règle la plus inhabituelle est l'accès à une structure le long d'un chemin (8). Elle agit lorsque l'on rencontre un chemin $p.x$; elle exprime que p doit faire référence à une structure contenant un champ x ; le type associé à x sera alors le type de $p.x$. Cependant, celui-ci peut éventuellement faire référence à des identificateurs déclarés plus haut dans la même signature. C'est le cas dans l'exemple ci-dessous :

```
p : sig
```

Typage d'une structure :

$$E \vdash \emptyset : \emptyset \quad (1) \qquad \frac{E \vdash e : T \qquad E;\ \texttt{value}\ v : T \vdash s : S}{E \vdash (\texttt{let}\ v = e;\ s) : (\texttt{value}\ v : T;\ S)} \quad (2)$$

$$\frac{E;\ \texttt{type}\ t \vdash s : S}{E \vdash (\texttt{datatype}\ t;\ s) : (\texttt{type}\ t;\ S)} \quad (3)$$

$$\frac{E \vdash T \text{ expression de type} \qquad E;\ \texttt{type}\ t = T \vdash s : S}{E \vdash (\texttt{type}\ t = T;\ s) : (\texttt{type}\ t = T;\ S)} \quad (4)$$

$$\frac{E \vdash m : M \qquad E;\ \texttt{module}\ x : M \vdash s : S}{E \vdash (\texttt{module}\ x = m;\ s) : (\texttt{module}\ x : M;\ S)} \quad (5)$$

$$\frac{E \vdash M \text{ type de module} \qquad E;\ \texttt{modtype}\ w = M \vdash s : S}{E \vdash (\texttt{modtype}\ w = M;\ s) : (\texttt{modtype}\ w = M;\ S)} \quad (6)$$

Typage d'une expression du langage de modules :

$$E \vdash x : E(x) \quad (7) \qquad \frac{E \vdash p : (\texttt{sig}\ S_1; \texttt{module}\ x : M;\ S_2\ \texttt{end})}{E \vdash p.x : M[n \leftarrow p.n \mid n \in \mathrm{Dom}(S_1)]} \quad (8)$$

$$\frac{E \vdash m : M' \qquad E \vdash M' \subset M}{E \vdash m : M} \quad (9) \qquad \frac{E \vdash s : S}{E \vdash (\texttt{struct}\ s\ \texttt{end}) : (\texttt{sig}\ S\ \texttt{end})} \quad (10)$$

$$\frac{E \vdash M \text{ type de module} \qquad E;\ \texttt{module}\ x : M \vdash m : M'}{E \vdash \texttt{functor}\ (x : M)\, m : \texttt{functor}\ (x : M)\, M'} \quad (11)$$

$$\frac{E \vdash m_1 : \texttt{functor}\ (x : M)\, M' \qquad E \vdash m_2 : M}{E \vdash m_1(m_2) : M'[x \leftarrow m_2]} \quad (12)$$

$$\frac{E \vdash M \text{ type de module} \qquad E \vdash m : M}{E \vdash (m : M) : M} \quad (13)$$

Figure 1 : Typage à base d'objets syntaxiques

```
  type t;;
  module x : sig value v : t;; end;;
end
```

Les occurrences de tels identificateurs dans le type de x doivent recevoir le préfixe $p.$, de sorte que le type de x reste valide une fois sorti du contexte de la signature de p. Dans l'exemple ci-dessus, on obtient pour $p.x$ le type `sig value` $v : p.t$ `end`, qui est le résultat attendu.

Le typage des autres expressions du langage de modules suit des règles simples. La règle d'application d'un foncteur (12) est la règle classique pour les types de fonctions dépendants : le nom du paramètre formel est remplacé par celui du paramètre effectif dans le type résultat. Rappelons que les chemins ne peuvent pas être formés d'expressions arbitraires, mais seulement d'identificateurs ; en conséquence, cette substitution n'est possible que si l'argument effectif du foncteur est lui-même un identificateur. Dans le cas contraire, une liaison intermédiaire est nécessaire. Cette restriction sera levée dans le système que nous présentons plus loin, grâce à l'abandon des objets syntaxiques pour le typage.

3.2.1 Equivalence et inclusion entre types

On définit une relation d'équivalence entre types, notée $\approx$, reflétant les équations entre types données par les déclarations de type manifestes (cf. figure 2).

La première règle définit la relation $\approx$ pour les types apparaissant dans l'environnement courant. La seconde l'étend aux types dont le nom est un chemin de longueur arbitraire. Elle nécéssite une substitution similaire à celle utilisée plus haut par la règle 8. Une relation similaire, également notée $\approx$, est définie pour les types de modules (`modtype`).

Nous pouvons à présent définir la relation d'inclusion entre types de modules, notée $\subset$, déjà mentionnée dans les règles précédentes (cf. règle 9). On suppose donnée la relation d'inclusion entre types ($E \vdash T \subset T'$). Pour les types comme pour les types de modules, on pose que l'équivalence entraîne l'inclusion. Reste à définir l'inclusion entre signatures et entre types de foncteurs (cf. figure 3).

L'inclusion entre type de foncteurs se vérifie aisément ; notons que les foncteurs étant contravariants, la comparaison des arguments se fait en sens inverse.

L'inclusion entre deux signatures se vérifie en comparant les composants deux à deux. Cependant, la relation $S_1 \subset S_2$ autorise la signature la plus générale (S_1) à avoir moins de champs ; de plus ils peuvent apparaître dans un ordre différent. Aussi la règle 17 utilise-t-elle une injection σ de $|S_1|$ dans $|S_2|$. Celle-ci donne les paires de champs à comparer. σ est déterminée de façon unique si l'on interdit à une signature de posséder plusieurs champs de même nom. Cette condition est d'ailleurs nécessaire, car si plusieurs possibilités existaient pour σ, la sémantique même du programme serait indéterminée : il y aurait plusieurs manières d'appliquer une contrainte de type à une structure.

σ étant déterminée, il reste à comparer les éléments deux à deux. Notons que cette comparaison est faite dans un environnement contenant S_1 ; les équations de type définies par S_1 peuvent en effet être nécessaires pour vérifier l'inclusion. Les règles de comparaison

entre deux éléments sont définies par la figure 4. Elles sont assez intuitives. Notons les règles 19 et 24, qui permettent d'abstraire un type en "oubliant" l'information qui lui est associée.

3.3 Typage à base de stamps

3.3.1 Limitations du système précédent

L'utilisation d'objets syntaxiques pour le typage permet d'en exposer la théorie de façon simple et directe. Cependant, elle n'est pas adaptée à l'implémentation d'un typeur. Représenter les types par des chemins est en effet très lourd : déterminer si deux chemins arbitraires représentent le même type demande de parcourir chacun en effectuant des substitutions sur les noms. Supposons en effet que nous recherchions la définition du type `A__B__t`, si `A` a la signature suivante :

```
sig
  type u;;
  ...
  module B : sig
    ...
    type t == u;;
    ...
  end;;
  ...
end;;
```

Le résultat recherché est `A__u`. Or la signature de `B` définit `type t == u`. Il est en fait nécessaire, au fur et à mesure que nous parcourons `A` à la recherche de la définition de `B`, de construire une substitution qui accolle le préfixe `A__` aux éléments définis par `A`. Ainsi `u` se voit correctement remplacé par `A__u`. La technique est donc simple mais peu maniable ; ce petit jeu doit en effet être répété à chaque unification et ralentit notablement l'ensemble du typage.

3.3.2 Une représentation plus directe des types

Nous avons donc été amenés à utiliser une méthode plus efficace quoique plus complexe à exposer. Elle est basée sur l'attribution à chaque type, lors de sa déclaration, d'un *stamp* (cachet) unique. Deux types seront alors compatibles si et seulement si ils possèdent le même stamp. L'unification est donc immédiate. En revanche, l'analyse des déclarations de type est plus complexe.

Un mécanisme similaire est utilisé pour les types de modules, qui se voient également associer des stamps.

L'idée d'utiliser des stamps provient de la Définition de Standard ML [MTH90]. Xavier Leroy a étudié un système à base de stamps et montré son équivalence avec le système

$$E_1; \texttt{type}\ t = T; E_2 \vdash t \approx T \quad (14)$$

$$\frac{E \vdash p : \texttt{sig}\ S_1; \texttt{type}\ t = T; S_2\ \texttt{end}}{E \vdash p.t \approx T[n \leftarrow p.n \mid n \in \mathrm{Dom}(S_1)]} \quad (15)$$

Figure 2 : Equivalence entre types

$$\frac{E \vdash M_2 \subset M_1 \qquad E; \texttt{module}\ x : M_2 \vdash M_1' \subset M_2'}{E \vdash \texttt{functor}\ (x : M_1)\, M_1' \subset \texttt{functor}\ (x : M_2)\, M_2'} \quad (16)$$

$$\frac{\sigma : \{1 \ldots n\} \hookrightarrow \{1 \ldots m\} \quad \forall i \in \{1 \ldots n\} \quad E; C_1; \ldots; C_m \vdash C_{\sigma(i)} \subset D_i}{E \vdash \texttt{sig}\ C_1; \ldots; C_m\ \texttt{end} \subset \texttt{sig}\ D_1; \ldots; D_n\ \texttt{end}} \quad (17)$$

Figure 3 : Règles d'inclusion entre types de modules

$$\frac{E \vdash T \subset T'}{E \vdash \texttt{value}\ v : T \subset \texttt{value}\ v : T'} \quad (18) \qquad E \vdash \texttt{type}\ t = T \subset \texttt{type}\ t \quad (19)$$

$$E \vdash \texttt{type}\ t \subset \texttt{type}\ t \quad (20) \qquad \frac{E \vdash T \approx T'}{E \vdash \texttt{type}\ t = T \subset \texttt{type}\ t = T'} \quad (21)$$

$$\frac{E \vdash t \approx T}{E \vdash \texttt{type}\ t \subset \texttt{type}\ t = T} \quad (22) \qquad \frac{E \vdash M \subset M'}{E \vdash \texttt{module}\ x : M \subset \texttt{module}\ x : M'} \quad (23)$$

$$E \vdash \texttt{modtype}\ w = M \subset \texttt{modtype}\ w \quad (24) \qquad E \vdash \texttt{modtype}\ w \subset \texttt{modtype}\ w \quad (25)$$

$$\frac{E \vdash M \approx M'}{E \vdash \texttt{modtype}\ w = M \subset \texttt{type}\ w = M'} \quad (26)$$

$$\frac{E \vdash w \approx M}{E \vdash \texttt{modtype}\ w \subset \texttt{modtype}\ w = M} \quad (27)$$

Figure 4 : Règles d'inclusion entre composants de signature

syntaxique [XL94b]. Cependant il s'est limité au cas des foncteurs de premier ordre. Le formalisme décrit ci-dessous est similaire, mais introduit des foncteurs d'ordre arbitraire et des déclarations de types de modules en tant qu'éléments de signature.

Les stamps seront notés n et parcourent un ensemble dénombrable. Les ensembles de stamps seront notés N. En guise d'abréviation, on appellera "modtype" un schéma de type de modules. Afin de simplifier légèrement le formalisme, on notera indifféremment n les stamps associés à un type et ceux associés à un modtype. Les objets utilisés pour le typage s'écrivent alors :

Types :	$\tau ::= n \mid \alpha \mid \tau \rightarrow \tau \mid \tau * \tau$
Types de modules :	$\theta ::= n \mid \Sigma \mid \Phi$
Signatures :	$\Sigma ::= \varepsilon \mid \sigma + \Sigma$
Eléments de signature :	$\sigma ::= \{v \mapsto \tau\} \mid \{t \mapsto \tau\} \mid \{x \mapsto \theta\} \mid \{w \mapsto \Theta\}$
Signatures de foncteurs :	$\Phi ::= \forall N_1 . (\theta_1, \forall N_2 . \theta_2)$
Modtypes :	$\Theta ::= \forall N . \theta$
Environnements :	$\Gamma ::= \Sigma$

Les noms de types sont donc remplacés par des stamps ; les autres constructions (types de fonctions, types produits, etc.) subsistent. Les types de modules reflètent la distinction entre structures et foncteurs. Une signature est simplement une liste de déclarations ; elle associe des objets sémantiques aux identificateurs. Un environnement de typage se présente sous la même forme qu'une signature.

L'objet le plus complexe est le type de foncteur : en plus du type argument et du type résultat, il est paramétré par deux ensembles de stamps. Le premier est l'ensemble des stamps instanciables ; ils correspondent aux types abstraits déclarés dans l'argument du foncteur, et par conséquent fonctionnent comme des variables universellement quantifiées. Le second est l'ensemble des stamps nouveaux ; ils correspondent aux types abstraits déclarés dans le résultat du foncteur, et doivent donc être régénérés à chaque application du foncteur. Prenons un exemple : soit `F` un foncteur de type

```
functor (X : sig type t;; end) sig
  type u;;
  type w == X__t;;
end
```

Le foncteur `F` est prêt à accepter comme argument n'importe quelle structure déclarant un type `t`, quelle que soit son implémentation. C'est pourquoi le stamp associé à `t` est instanciable ; lors de l'application de `F`, il pourra être identifié au stamp de l'argument effectif. Le type `u`, quant à lui, est déclaré comme abstrait dans la structure résultat. Parce que `u` peut éventuellement dépendre de `X`, il est nécessaire que chaque application du foncteur `F` génère un nouveau type `u`. Le stamp de `u` fera donc partie de N_2, et un stamp "frais" lui sera substitué à chaque application. Enfin, le type `w` n'a pas de stamp propre ; son stamp est celui de `t`. L'objet sémantique représentant le type de `F` sera donc

$$\forall\{n_t\}.\,(\{t \mapsto n_t\},\quad \forall\{n_u\}.\,\{u \mapsto n_u;\, w \mapsto n_t\})$$

Enfin, un modtype est formé d'un type de module quantifié par un ensemble de stamps instanciables. Cet ensemble correspond aux types abstraits déclarés dans le corps du modtype.

3.3.3 Règles d'élaboration

Les règles de typage (appelées également règles d'*élaboration*, selon la terminologie SML), sont données par la figure 5. Aux expressions du langage de base sont associés des types selon les règles classiques. A chaque expression du langage de modules est associé non seulement un type, mais également un ensemble de stamps ; il s'agit de l'ensemble des stamps générés pendant l'élaboration de cette expression. Il est ensuite utilisé de façon différente selon le contexte (argument de foncteur, résultat de foncteur, déclaration de type de module, etc.).

Les règles servant à élaborer les expressions de type (signatures, types de foncteurs, etc.) sont très semblables et ont été omises.

On note $\mathrm{Inst}(N)$ l'ensemble des substitutions φ des stamps vers les types (et des stamps associés à un `modtype` vers les types de modules) telles que $\mathrm{Dom}(\varphi) \subset N$. L'application d'une telle substitution à une signature instancie donc chaque stamp de N en le remplaçant par une expression de type (respectivement, de type de modules).

On note $\mathrm{FS}(\Gamma)$ (pour *Free Stamps*) l'ensemble des stamps apparaissant libres (c'est-à-dire non liés par un quantificateur) dans l'environnement Γ.

Le typage d'une structure est relativement immédiat. Les éléments de la structure sont considérés tour à tour. La structure étant un produit dépendant, chaque élément est ajouté à l'environnement aussitôt après avoir été typé. Les règles 29 et 31 font appel aux règles de base du langage, qui ne sont pas rappelées ici. La règle 30 s'applique lors de la déclaration d'un type concret ; un nouveau stamp lui est attribué ; il est ajouté à l'ensemble des stamps définis par la structure courante.

La règle d'élaboration d'un foncteur (36) est simple ; elle se contente d'analyser son argument, puis de typer le résultat dans un environnement contenant l'argument. C'est l'analogue de la règle de typage des fonctions. Le type du foncteur est quantifié par les stamps rencontrés dans la définition ; il serviront lors du typage de l'application. Notons que c'est dans cette règle que nous utilisons les ensembles de stamps construits par les autres règles : ils deviennent les quantificateurs du type foncteur.

La règle d'application d'un foncteur (37) est de loin la plus complexe. Lorsque l'on rencontre l'application $m_1(m_2)$, on commence par déterminer les types de m_1 et m_2. Le premier doit être un type de foncteur, de la forme $\forall N_1.\,(\theta_1, \forall N_2.\,\theta_2)$. Le second est un type de modules quelconque, noté θ' ; on note N' l'ensemble des stamps générés pendant l'élaboration de m_2. Pour que l'application $m_1(m_2)$ soit valide, il faut vérifier que le type attendu par le foncteur est plus général que le type de l'argument effectif m_2.

Avant de réaliser cette comparaison, on instancie de façon convenable les stamps de N_1. En effet, ces stamps sont ceux des types abstraits apparaissant dans la signature argument

Elaboration d'un chemin : $\Gamma(p_s.x) = (\Gamma(p_s))(x)$

Elaboration d'une structure :

$$\Gamma \vdash \varepsilon : \{\}, \emptyset \quad (28) \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma + \{v \mapsto \tau\} \vdash d : \Sigma,\ N}{\Gamma \vdash (\texttt{value}\ v = e;\ d) : \{v \mapsto \tau\} + \Sigma,\ N} \quad (29)$$

$$\frac{n \notin \mathrm{FS}(\Gamma) \quad \Gamma + \{t \mapsto n\} \vdash d : \Sigma,\ N}{\Gamma \vdash (\texttt{datatype}\ t;\ d) : \{t \mapsto n\} + \Sigma,\ \{n\} \cup N} \quad (30)$$

$$\frac{\Gamma \vdash T : \tau \quad \Gamma + \{t \mapsto \tau\} \vdash d : \Sigma,\ N}{\Gamma \vdash (\texttt{type}\ t = T;\ d) : \{t \mapsto \tau\} + \Sigma,\ N} \quad (31)$$

$$\frac{\Gamma \vdash m : \theta,\ N_1 \quad \Gamma + \{x \mapsto \theta\} \vdash d : \Sigma,\ N_2}{\Gamma \vdash (\texttt{module}\ x = m;\ d) : \{x \mapsto \theta\} + \Sigma,\ N_1 \cup N_2} \quad (32)$$

$$\frac{\Gamma \vdash M : \theta,\ N \quad \Gamma + \{w \mapsto \forall N.\, \theta\} \vdash d : \Sigma,\ N_1}{\Gamma \vdash (\texttt{modtype}\ w = M;\ d) : \{w \mapsto \forall N.\, \theta\} + \Sigma,\ N_1} \quad (33)$$

Elaboration d'une expression du langage de modules :

$$\Gamma \vdash p_s : \Gamma(p_s),\ \emptyset \quad (34) \qquad \frac{\Gamma \vdash d : \Sigma,\ N}{\Gamma \vdash \texttt{struct}\ d\ \texttt{end} : \Sigma,\ N} \quad (35)$$

$$\frac{\Gamma \vdash M : \theta_1,\ N_1 \quad \Gamma + \{x \mapsto \theta_1\} \vdash m : \theta_2,\ N_2}{\Gamma \vdash \texttt{functor}\ (x : M)\, m : \forall N_1.\, (\theta_1, \forall N_2.\, \theta_2),\ \emptyset} \quad (36)$$

$$\frac{\begin{array}{c}\Gamma \vdash m_1 : \forall N_1.\, (\theta_1, \forall N_2.\, \theta_2),\ \emptyset \quad \Gamma \vdash m_2 : \theta',\ N' \\ \varphi \in \mathrm{Inst}(N_1)\ \varphi(\theta_1) \prec \theta' \quad N_2' \subset \overline{\mathrm{FS}(\Gamma)}\end{array}}{\Gamma \vdash m_1(m_2) : \varphi(\theta_2)[N_2 \leftarrow N_2'],\ N_2' \cup N'} \quad (37)$$

$$\frac{\Gamma \vdash m : \theta_1,\ N_1 \quad \Gamma \vdash M : \theta_2,\ N_2 \quad \varphi \in \mathrm{Inst}(N_2) \quad \varphi(\theta_2) \prec \theta_1}{\Gamma \vdash (m : M) : \theta_2,\ N_2} \quad (38)$$

Figure 5 : Typage à base de stamps

du foncteur ; ils sont donc universellement quantifiés, et on peut les particulariser. L'instanciation est exprimée dans la règle 37 par le choix d'une substitution $\varphi \in \mathrm{Inst}(N_1)$. La façon dont on détermine φ n'est pas explicitée par la règle ; en pratique, il s'agit d'un procédé similaire à l'instanciation des variables pendant l'unification de deux types.

Reste à comparer $\varphi(\theta_1)$ à θ'. On pourrait imposer ici l'égalité de ces deux types. Cependant, comme dans le système précédent, nous souhaitons autoriser l'omission ou l'interversion de champs dans les signatures. Aussi utilisons-nous la relation $\prec$ définie plus bas. Cette même relation est utilisée lors de l'application d'une contrainte de type à une expression du langage de modules.

3.3.4 Inclusion entre types de modules

Informellement, $\Sigma_1 \prec \Sigma_2$ signifie que la signature Σ_1 est plus générale que Σ_2, i.e. qu'elle a éventuellement moins de champs. La relation est ensuite étendue aux foncteurs de façon relativement simple : les types arguments doivent être comparés dans l'ordre inverse, et les types résultat dans le même ordre. Un type de foncteurs étant un objet quantifié par un ensemble de stamps, la comparaison de deux types de foncteurs nécessitera cependant d'instancier certains stamps.

La relation $\prec$ est définie par la figure 6. Les règles d'inclusion sont moins nombreuses que dans le système précédent, puisqu'il y en a maintenant une par objet sémantique. Cependant elles sont parfois plus complexes à cause du mécanisme d'instanciation des stamps. On notera que ces règles ne font appel à aucun environnement : l'utilisation de stamps permet en effet de savoir sans aucune recherche si deux types sont ou non compatibles. C'était le but recherché lors de l'introduction des stamps.

La règle 39 indique qu'un type abstrait de modules n'est comparable qu'avec lui-même.

Viennent ensuite les règles définissant l'inclusion entre signatures. Comme dans le système précédent, on compare les éléments deux à deux, en autorisant les permutations et les omissions (cf règle 40). Les règles 41 à 44 indiquent comment s'effectuent les comparaisons entre éléments.

Deux déclarations de valeurs vérifient $\prec$ si le types déclaré est moins général que le type effectif ($\tau_1 \prec \tau_2$).

Pour les définitions de types, en revanche, il est nécessaire d'imposer l'égalité des expressions de type (règle 42). Le type pourra en effet être utilisé par la suite tant en position covariante qu'en position contravariante ; l'inclusion est donc requise dans les deux sens.

La comparaison de deux sous-modules se fait par simple application de la relation $\prec$ (c'est-à-dire, en pratique, par un appel récursif).

Comme dans le cas des définitions de types, deux définitions de modtypes sont comparables si et seulement si les expressions de type de modules associées sont égales. Cette égalité doit cependant être vérifiée modulo un renommage des stamps qui quantifient ces deux expressions. C'est pourquoi la règle 44 utilise une bijection γ de N_1 dans N_2. Il s'agit en fait d'une simple égalité modulo α-conversion.

Reste enfin la comparaison de deux types de foncteurs $\forall N_1.\,(\theta_1, \forall N_2.\,\theta_2)$ et $\forall N_1'.\,(\theta_1', \forall N_2'.\,\theta_2')$. Nous voulons vérifier que le premier est plus général que le second. Son type argument doit

$$n \prec n \quad (39)$$

$$\frac{\forall \sigma_1 \in \Sigma_1 \quad \exists! \sigma_2 \in \Sigma_2 \quad \sigma_1 \prec \sigma_2}{\Sigma_1 \prec \Sigma_2} \quad (40)$$

$$\frac{\tau_1 \prec \tau_2}{\{v \mapsto \tau_1\} \prec \{v \mapsto \tau_2\}} \quad (41)$$

$$\frac{\tau_1 = \tau_2}{\{t \mapsto \tau_1\} \prec \{t \mapsto \tau_2\}} \quad (42)$$

$$\frac{\theta_1 \prec \theta_2}{\{x \mapsto \theta_1\} \prec \{x \mapsto \theta_2\}} \quad (43)$$

$$\frac{\gamma \in N_1 \leftrightarrow N_2 \quad \gamma(\theta_1) = \theta_2}{\{w \mapsto \forall N_1 . \theta_1\} \prec \{w \mapsto \forall N_2 . \theta_2\}} \quad (44)$$

$$\frac{\varphi \in \mathrm{Inst}(N_1') \quad \varphi(\theta_1') \prec \theta_1 \qquad \psi \in \mathrm{Inst}(N_2) \quad \psi(\theta_2) \prec \varphi(\theta_2')}{\forall N_1 . (\theta_1, \forall N_2 . \theta_2) \prec \forall N_1' . (\theta_1', \forall N_2' . \theta_2')} \quad (45)$$

Figure 6 : Relation d'inclusion entre types de modules

donc être moins général. Pour le vérifier, il faut comparer θ_1' à θ_1 en autorisant l'instanciation des stamps propres de θ_1'. Ensuite il faut comparer les types résultat, en conservant l'instanciation déjà effectuée, et en autorisant en sus l'instanciation des stamps propres de θ_2. D'où l'énoncé de la règle 45.

4 Intégration avec le compilateur Caml Light existant

Cette section évoque certains problèmes d'implémentation rencontrés lors de la mise en œuvre du système "grandeur nature". L'implémentation du compilateur Caml Light est décrite dans son ensemble par [XL90].

4.1 Compilation séparée

4.1.1 Typage

Caml Light 0.6 autorise l'utilisateur à découper son programme en un certain nombre de modules. Chaque module est implémenté dans un fichier portant le suffixe `.ml`, et publie éventuellement une interface au moyen d'un fichier `.mli`. Ce mécanisme a été conservé, et présente l'avantage de s'intégrer parfaitement au nouveau système de types : un fichier `.ml` peut en effet être considéré comme le corps d'une structure, et un fichier `.mli` comme le corps d'une signature. Vérifier que l'implémentation est conforme à l'interface n'est pas autre chose qu'appliquer une contrainte de type à un module, et ne demande donc pas de code *ad hoc*.

Notons qu'il n'était pas nécessaire de conserver le mécanisme `.ml`/`.mli`. Il serait en effet théoriquement possible de laisser l'utilisateur appliquer explicitement une contrainte de type dans le cas où il désire cacher les composants internes de son module. On aurait donc pu ne conserver que les fichiers `.ml`. Cependant, cette méthode présente le désavantage d'être plus lourde pour l'utilisateur et peu efficace (elle nécessite une indirection supplémentaire pour accéder aux champs du module).

Caml Light 0.6 offre deux moyens à l'utilisateur d'accéder aux champs d'un module externe : la directive `#open` et l'ouverture au vol.

La directive `#open` a été conservée. Son comportement a cependant été légèrement modifié. "`#open M;;`" était traité auparavant en ajoutant l'ensemble des champs de `M` à une table d'identificateurs globaux. Cette table était parcourue lorsqu'une variable était rencontrée qui ne figurait pas dans l'environnement. A présent, le typeur est formé de deux étages : l'un chargé des expressions Caml, l'autre des expressions du langage de modules, supervisant le premier. Chacun entretient un environnement de typage ; le premier peut être appelé environnement local, le second environnement global. Pour réagir à la directive, il suffit d'accoler la signature de `M` à l'environnement global. Tous les champs de `M` seront ainsi directement accessibles.

L'ouverture d'un module au vol consiste à ajouter automatiquement le module `M` à la table lorsqu'on rencontre une référence du type `M__x`. Cette méthode, bien que pratique, semble

douteuse dans le cadre du nouveau système : selon les règles de typage, cette référence est invalide si `M` n'appartient pas à l'environnement. Il semble donc plus propre d'imposer l'usage d'une directive "`#import M;;`", laquelle autorise ensuite les références directes à `M`. Celle-ci est implémentée par l'ajout dans l'environnement global d'une entrée décrivant `M`. `M` peut ainsi être utilisé normalement par la suite.

Il est important de noter que comme `#open` et `#import` agissent sur l'environnement courant, leur portée (*scope*) est à présent limitée à la structure ou la signature dans laquelle ils apparaissent. Ceci permet de les utiliser de façon plus locale, et semble bénéfique. Une autre conséquence est que l'ancienne directive `#close` devient difficile à implémenter, puisqu'il devrait retirer certains éléments placés au cœur de l'environnement. Son utilité a par ailleurs diminué, puisque `#open` est maintenant local ; aussi a-t-elle été abandonnée.

4.1.2 Compilation

La directive `#open` permet de nommer des variables par un chemin partiel, en rendant implicite le début du chemin. La partie avant du générateur de code (*front-end*) doit cependant générer correctement un accès à un module externe lorsqu'elle rencontre une telle variable. En Caml Light 0.6, le typeur modifiait l'arbre de syntaxe en place lorsqu'il rencontrait une telle variable, de sorte que le front-end travaillait sur un arbre non ambigu. Dans notre cas, la directive place directement les composants du module externe dans l'environnement, ce qui rend le typeur incapable de les différencier des variables normales. Notre front-end est donc à son tour confronté au problème. Il le résoud en réagissant lui aussi aux directives `#open` et `#import` : quand il les rencontre, il ajoute dans l'environnement de compilation des indications permettant de générer un code correct.

Enfin, notons que considérer le contenu d'un fichier `.ml` comme le corps d'une structure portant le nom du fichier nécessite une indirection supplémentaire pour accéder aux variables. En Caml Light 0.6, les variables déclarées par un module étaient globales. A présent, la seule variable globale est le module lui-même ; il est représenté par un bloc, chaque champ contenant une variable. Une certaine perte d'efficacité est donc à déplorer. Il serait possible de modifier le compilateur pour qu'il "aplatisse" le premier niveau de modules ; le code généré serait identique à celui des versions précédentes, et le formalisme serait inchangé. Nous l'avons cependant évité afin de conserver la plus grande simplicité possible.

4.2 Compilation des contraintes de type

En Caml Light usuel, imposer une contrainte de type à une valeur, par une expression de la forme `(x : T)`, ne génère aucun code ; le compilateur traite cette expression comme `x`. De même, passer un argument d'un certain type concret à une fonction plus générique (par exemple, une liste d'entiers à une fonction de type $\alpha \rightarrow \alpha$) ne génère aucun code particulier. Cela est rendu possible par la simplicité de la relation de sous-typage : le fait que le type de l'argument soit moins général que le type attendu par la fonction signifie que celle-ci ne connaît pas totalement la structure de son argument, donc, du point de vue du code généré,

que certains pointeurs seront traités comme s'il s'agissait de valeurs quelconques. Cela ne peut poser aucun problème à l'exécution.

L'ajout d'un langage de modules au-dessus de Caml Light introduit cependant une nouveauté : une relation de sous-typage entre signatures. Il est à présent possible, en appliquant une contrainte de type à une structure, de cacher certains de ses champs et de permuter les autres. On peut donc passer en argument à un foncteur une structure dont les champs ne sont pas rangés dans l'ordre attendu par celui-ci.

Deux méthodes se présentent à l'esprit pour résoudre cette difficulté : la première est d'accéder aux champs d'une structure par nom et non par index ; la seconde est de générer du code supplémentaire pour chaque contrainte de type et chaque application de foncteur.

La première méthode a été implémentée dans plusieurs systèmes possédant une relation de sous-typage entre enregistrements (*records*) [DR92]. Au lieu d'associer à chaque élément du record un numéro de champ fixe, il faut, à chaque accès, effectuer une recherche parmi les champs pour trouver la position de l'élément cherché. Diverses implémentations sont possibles ; toutes, bien entendu, impliquent une perte d'efficacité par rapport à l'accès direct.

La seconde méthode consiste à conserver l'accès direct aux éléments, chacun se voyant attribuer un numéro de champ unique. En contrepartie, il est nécessaire, lorsqu'on rencontre une contrainte de type, de construire une copie de la structure originale dans laquelle les champs sont réordonnés de façon convenable. Ce processus est lent ; c'est pourquoi les systèmes qui considèrent le sous-typage entre records comme fondamental préfèrent souvent la première approche. Cependant, dans notre cas l'accès aux structures est extrêmement fréquent (tout module externe est une structure), tandis que les restrictions de type sur un module et les applications de foncteurs sont rares : elles servent à la construction des structures, et apparaissent donc en général durant l'initialisation des programmes. En particulier, on notera qu'en tant que constructions du langage de modules, elles ne peuvent pas faire partie d'une boucle. Il nous a donc paru préférable de les pénaliser et de conserver l'efficacité de l'accès aux structures.

Nous avons donc choisi de générer du code pour les conversions de types de modules. Il y a conversion partout où apparaît la condition $\theta_1 \prec \theta_2$ dans les règles de typage. Vérifier la relation $\Sigma_1 \prec \Sigma_2$ conduit le typeur, pour chaque champ de Σ_1, à rechercher le champ correspondant de Σ_2 ; il peut donc construire au fur et à mesure une injection de $\{1 \ldots |\Sigma_1|\}$ dans $\{1 \ldots |\Sigma_2|\}$, qui sera transmise au compilateur et lui permettra de générer le code nécessaire à la conversion. Une optimisation simple consiste à ne générer aucun code si cette injection est l'identité, c'est-à-dire si les deux signatures possèdent les mêmes éléments dans le même ordre.

Il convient également de générer du code lorsqu'on rencontre une restriction de type sur un foncteur. Le foncteur modifié doit d'abord appliquer une restriction de type à ses arguments, puis exécuter le code du foncteur original, et enfin convertir le type de son résultat. Il faut faire attention à exécuter le code original dans le contexte pour lequel il a été prévu ; sans quoi les indices de De Bruijn qu'il contient deviennent faux.

La routine `restrict_module_expr`, chargée de compiler les restrictions de types de modules, est donnée en annexe (elle fait partie de `front.ml`).

4.3 Exceptions

Bien que l'ajout d'un langage de modules semble indépendant, du point de vue théorique, du système d'exceptions de Caml Light, nous avons dû revoir la façon dont les exceptions sont implémentées.

Le problème est lié au partage du code des foncteurs. Considérons le foncteur suivant :

```
module F = functor (X : S) struct
        exception E;;
        let f = function
          ... raise E ...
      end;;
```

On désire compiler les foncteurs comme des fonctions usuelles. Ainsi, si `A` et `B` sont deux modules créés par application de `F`, `A__f` et `B__f` partageront le même code.

Cependant, on souhaite que chaque application du foncteur `F` génère correctement une nouvelle exception, c'est-à-dire que `A__E` soit un constructeur distinct de `B__E`. Il est donc impossible, comme on le faisait jusqu'ici, d'attribuer un numéro (*tag*) unique à chaque exception pendant l'édition de liens. Ce tag apparaîtrait en effet dans le code de `f`, et il serait impossible de faire en sorte que `A__f` et `B__f` soulèvent des exceptions distinctes.

La solution que nous avons adoptée consiste à stocker le tag lui-même dans l'un des champs de la structure. Le foncteur ci-dessus est compilé de façon semblable à ceci :

```
module F = functor (X : S) struct
        let tag_E = new_exception_tag();;
        let f = function
          ... raise (make_exception tag_E) ...
      end;;
```

Un champ de la structure, nommé ici `tag_E`, est alloué pour contenir le tag de `E` ; la primitive `new_exception_tag` est appelée au moment où la structure est créée (c'est-à-dire à chaque application du foncteur, à l'exécution), et place dans ce champ un numéro unique. Plus tard, lorsque la fonction `f` exécute l'instruction `raise E`, elle accède au champ `tag_E` pour utiliser le bon tag. Le code généré pour le filtrage (`try ... with`) est également modifié en conséquence.

Grâce à ces modifications, les exceptions `A__E` et `B__E` sont distinctes, bien que les fonctions `A__f` et `B__f` aient le même code.

4.4 Impression des noms de types

L'utilisation de stamps uniques pour représenter les types a soulevé un problème inattendu quant à l'impression des expressions de type. Une fois un type défini, il est représenté de façon interne par un stamp. Il est facile, grâce à une série de recherches dans l'environnement, de transformer un nom de type (i.e. un chemin) en une expression à base de stamps. La conversion est plus difficile en sens inverse, puisqu'un environnement est une structure conçue

pour effectuer des recherches par nom, non par stamp. Elle est cependant indispensable pour pouvoir imprimer des messages lisibles en cas d'erreur de typage.

Le problème ne se posait pas lorsque le système de types était basé sur des objets syntaxiques (cf. section 3.2), précisément parce que les types étaient directement représentés par leur nom complet. L'usage de stamps nous a permis de nous débarrasser de la lourde mécanique des chemins ; mais nous nous trouvons forcés d'en conserver une partie à des fins d'affichage.

Il est facile d'enregistrer avec le stamp le nom du type auquel il correspond. Cependant, il s'agit là d'un simple identificateur ; le nom complet du type est un chemin d'accès de longueur arbitraire, qui dépend de l'environnement courant au moment où l'on cherche à l'obtenir. Il nous faut donc un moyen de reconstituer le préfixe correct.

On ajoute pour cela à chaque stamp la liste des noms des structures qui l'entourent. A chaque fois que l'on entre une signature Σ dans l'environnement sous un nom S, on ajoute S en tête de cette liste, et ce pour tous les stamps définis par Σ. Par exemple, si l'on définit

```
module A = struct
  type t = K of int;;
  module B = struct
    type u = One | Two;;
    type v == t;;
  end;;
end;;
```

Une fois ces définitions analysées, le stamp de `t` porte la liste [“`A`”], et celui de `u` la liste [“`A`”; “`B`”]. Pour imprimer le nom complet d'un stamp, il suffit maintenant d'imprimer son nom court, précédé du préfixe donné par la liste associée.

Il reste cependant un problème : cette méthode suppose que l'environnement au moment de l'impression est l'environnement de typage courant. Or ce n'est pas forcément le cas lorsqu'on imprime une signature. Par exemple, si nous voulons imprimer la signature du module `A` défini ci-dessus, nous souhaitons obtenir la définition `type v == t` et non pas `type v == A__t`. Au moment où nous imprimons la définition de `v`, nous sommes dans le contexte de `A`, et nous devons en tenir compte.

Pour cela, les fonctions d'impression de types de modules maintiennent une pile (représentée par une liste) représentant le contexte d'impression. A chaque fois que l'on entreprend l'impression d'une signature, on empile son nom ; on le dépile quand on la termine. Quand on doit imprimer un nom de type, on utilise la méthode précédente pour obtenir un chemin complet ; mais si ce chemin présente un préfixe commun avec la pile, on omet celui-ci.

Ainsi lorsque l'on atteint la définition de `v` dans l'exemple ci-dessus, le stamp de `t` porte la liste [“`A`”], tandis que la pile contient [“`A`”; “`B`”]. On élimine le préfixe commun [“`A`”], et le nom complet obtenu est bien `t`.

Notons que la notion de “préfixe commun” doit être basée sur l'égalité physique entre chaînes, et non sur la simple égalité des identificateurs, si l'on veut éviter les problèmes de renommage.

4.5 Primitives

Caml Light 0.6 supporte un mécanisme de déclaration de primitives externes, implémentées en C. L'intérêt de cette caractéristique est double : elle autorise l'accès à toutes les routines du système d'exploitation, et elle permet d'écrire de façon efficace les fonctions pour lesquelles la performance est importante. La librairie standard de Caml Light fait un usage intensif de primitives C ; aussi était-il indispensable de conserver ce moyen d'interface avec des routines externes.

En Caml Light 0.6, une primitive C doit être déclarée comme telle dans l'interface d'un module, et n'apparaît pas dans son implémentation. Cette façon de faire n'est pas cohérente avec le style de notre système de modules. En effet, la façon dont une fonction est implémentée (i.e. en Caml ou en C) ne change rien à son type ; il serait donc logique qu'une primitive soit déclarée comme une fonction ordinaire dans l'interface du module, et que seule l'implémentation porte mention du fait qu'elle est écrite en C.

Cependant, cette méthode entraînerait une perte d'efficacité. Lorsque le compilateur rencontre un appel à une fonction `f` déclarée comme primitive, il est capable de générer un appel "en ligne" (*inline*) au lieu d'un appel de fonction classique. Dans le premier cas, lors de l'exécution, l'interpréteur exécute un appel direct à la routine C correspondante ; dans le second, il doit effectuer un accès à la variable `f`, puis un appel de fonction classique, qui demande une β-réduction par argument.

Il est donc nécessaire, pour préserver le gain de performance dû à l'*inlining*, de déclarer les primitives comme telles également dans les signatures de modules. C'est ce mécanisme que nous avons implémenté : toute structure peut contenir une déclaration `primitive` ; dans la signature doit figurer soit une déclaration `primitive` identique, soit une déclaration `value` ordinaire, auquel cas le caractère particulier de cette fonction est oublié, et l'inlining abandonné. On peut choisir cette seconde solution, quand on se soucie peu de la performance, pour ne pas faire apparaître de détails d'implémentation dans une signature.

4.6 Persistence des stamps

Un stamp est un "cachet", un "numéro de série" attribué à un type. Les seules opérations dont nous ayons besoin sur les stamps sont la création (lors de la définition d'un type), la comparaison (lors de l'unification de deux types), et l'instanciation. Il serait donc naturel de définir un stamp comme un record contenant uniquement un lien d'instanciation, et de se baser sur l'égalité des adresses pour la comparaison :

```
type Link = Snolink
          | Slinkto of stamp

and stamp = { link : Link };;

let equal_stamp stamp1 stamp2 =
  let rec repr = function
    { link = Snolink } as stamp -> stamp
```

```
    | { link = Slinkto stamp } -> repr stamp

    in repr stamp1 == repr stamp2
  ;;
```

Cela fonctionne parfaitement tant que l'on ne travaille que sur une unité de compilation. L'introduction de la compilation séparée rend ce procédé invalide.

En effet, considérons un stamp n défini par une unité `A`. Si un module `B` importe `A`, c'est-à-dire lit dans le fichier `A.zi` la signature de `A`, tout se passe bien. En effet, un fichier `.zi` est créé, puis relu, par les fonctions `output_value` et `input_value`, qui conservent le partage physique ; autrement dit, si deux types partageaient le stamp n dans la signature originale, ce sera toujours vrai dans la signature lue depuis le fichier `A.zi`.

Les choses se compliquent si les dépendances forment un losange (*diamond import*) : supposons qu'un module `C` importe également `A`, et finalement qu'un module `D` importe `B` et `C`. Avant de traiter `D`, le compilateur va charger en mémoire les signatures stockées dans les fichiers `B.zi` et `C.zi`. Or le stamp n apparaît dans chacune des deux ; mais le compilateur n'a aucun moyen de savoir qu'il s'agit du même stamp. Des types qui devraient être identiques seront ainsi jugés incompatibles.

Il convient donc de stocker parmi les champs du stamp un identificateur unique qui permette de comparer deux stamps sans avoir recours à leur adresse. L'idée la plus simple est d'attribuer à chaque stamp, lors de sa création, un numéro unique, à l'aide d'un compteur global. Ceci n'est pas suffisant, car le compteur serait remis à zéro lors de chaque compilation, donc des stamps provenant d'unités différentes pourraient avoir le même numéro. On ajoute donc le nom de l'unité de compilation d'où provient le stamp.

Le système ainsi obtenu n'est pas encore sûr : supposons que dans l'exemple du *diamond import* ci-dessus, `A.zi` ait été modifié entre l'instant où on a compilé `B` et celui où on a compilé `C`. Il pourrait alors arriver, pendant la compilation de `D`, que l'on rencontre deux stamps provenant de `A`, portant le même numéro, et qui cependant ne correspondent pas au même type, parce que ces stamps ne proviennent pas de la même version de `A.zi`. On résoud ce problème en ajoutant à chaque stamp l'heure de sa création.

Enfin, les perfectionnistes noteront que deux stamps peuvent avoir été créés à la même heure par deux processus de compilation tournant en parallèle. On ajoute donc au stamp le numéro de process Unix du compilateur et le nom de la machine sur laquelle il tourne... Le coût des champs supplémentaire est faible, et la sécurité est ainsi totale.

5 Un exemple : la gestion d'arbres binaires

Voici à présent un exemple de la puissance apportée par le nouveau système de modules de Caml Light.

Depuis la version 0.6, la librairie standard de Caml Light contient une série de fonctions permettant d'utiliser aisément des arbres binaires équilibrés. Elles sont regroupées dans le module `baltree`.

Un arbre binaire permet de stocker un ensemble ordonné de valeurs. Le fait que l'ensemble soit ordonné permet d'effectuer les opérations de stockage et d'accès de façon efficace. L'interface du module `baltree` se présente sous la forme suivante :

```
type 'a t = Empty | Node of 'a t * 'a * 'a t * int;;

value add: ('a -> int) -> 'a -> 'a t -> 'a t
  and contains: ('a -> int) -> 'a t -> bool
  and remove: ('a -> int) -> 'a t -> 'a t
  ...
  and compare: ('a -> 'a -> int) -> 'a t -> 'a t -> int
  and join: 'a t -> 'a -> 'a t -> 'a t
  ...
```

Toutes les opérations sur les arbres prennent en argument une fonction d'ordre `f`. Celle-ci est normalement de type `'a -> 'a-> int` ; dans certains cas l'un des deux éléments à comparer est constant au cours de l'opération, et il suffit d'utiliser une fonction de type `'a -> int`.

Or il est nécessaire pour le bon fonctionnement du module qu'un arbre donné soit toujours utilisé avec la même fonction d'ordre. En effet, toutes les opérations font l'hypothèse que l'arbre sur lequel elles doivent travailler est déjà trié à l'aide de la fonction qu'on leur passe en argument. Utiliser un même arbre avec deux ordres différents conduit à des résultats incohérents.

Il serait donc souhaitable que le système de types soit capable d'imposer cette contrainte et d'empêcher l'utilisateur de passer un mauvais ordre en argument. Cependant, Caml Light 0.6 en est incapable. L'idée qui viendrait naturellement à l'esprit est de stocker la fonction d'ordre avec l'arbre lors de sa création, et de l'utiliser ensuite comme paramètre implicite. La fonction `add`, par exemple, aurait alors le type `'a -> 'a t -> 'a t`, et on serait certain que deux appels consécutifs à `add` utilisent le même ordre.

Malheureusement, cela ne résoud pas le problème dans le cas des fonctions prenant deux arguments de type `t`. Rien ne permet d'imposer aux deux ensembles d'avoir la même fonction d'ordre. Il est donc impossible de construire un système sûr en utilisant uniquement le langage de base.

En fait, le type `baltree_t` devrait être paramétré non seulement par `'a`, le type des éléments de l'ensemble, mais également par `f`, la fonction d'ordre utilisée pour construire l'arbre. Ainsi deux ensembles seraient compatibles uniquement s'ils sont fondés sur le même ordre. Paramétrer un type par une valeur est impossible en ML ; mais cela est rendu possible par l'utilisation d'un foncteur. Définissons

```
modtype Ordering = sig
  type t;;
  value f : t -> t -> int;;
end;;

module baltree : functor (BaseType : Ordering)
```

```
  sig
    type t;;

    value empty: t
      and add: BaseType__t -> t -> t
      and contains: BaseType__t -> t -> bool
      and remove: BaseType__t -> t -> t
      ...
      and compare: t -> t -> int
      and join: t -> BaseType__t -> t -> t
      ...
end;;
```

La signature `Ordering` représente un type ordonné, c'est-à-dire un type muni d'une fonction de comparaison. Le module `baltree` est maintenant défini comme un foncteur ; il prend en argument un type ordonné et renvoie une structure qui implémente les ensembles sur ce type.

Nous avons à présent obtenu le résultat recherché : si nous travaillons sur deux ordres différents, nous devrons appliquer le foncteur `baltree` à chacun, et il en résultera deux types `t` incompatibles, comme le montre l'exemple ci-dessous :

```
module Integers = struct
  type t == int;;
  let f x y = x - y;;
end;;

module IntegersMod9 = struct
  type t == int;;
  let f x y = (x - y) mod 9;;
end;;

module baltree1 = baltree Integers;;
module baltree2 = baltree IntegersMod9;;

let set1 = baltree1__add 3 baltree1__empty
and set2 = baltree2__add 3 baltree2__empty
in baltree1__compare set1 set2;;
```

La dernière phrase constitue une tentative de comparer deux ensembles fondés sur des ordres différents. Elle est correctement refusée par le typeur :

```
#  in baltree1__compare set1 set2;;
#                             ^^^^
# Expression of type baltree2__t
# cannot be used with type baltree1__t
```

Références

[HL94] R. Harper et M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. *Proc. 21st Symp. Principles of Programming Languages.* ACM Press, 1994.

[MQ86] D. MacQueen. Modules for Standard ML. In R. Harper, D. MacQueen, and R. Milner, editors, *Standard ML.* University of Edinburgh, technical report ECS LFCS 86-2, 1986.

[MTH90] R. Milner, M. Tofte, R. Harper. *The definition of Standard ML.* The MIT Press, 1990.

[XL90] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. *Rapport technique INRIA 117*, 1990.

[XL94a] Xavier Leroy. Manifest types, modules and separate compilation. In *Proc. 21st Symp. Principles of Programming Languages.* ACM Press, 1994.

[XL94b] Xavier Leroy. A syntactic theory of type generativity and sharing. In *Record of the 1994 Workshop on ML and its applications.* Rapport de recherche 2265, INRIA, 1994.

[DR92] Didier Rémy. Efficient Representation of Extensible Records. *Proceedings of the 1992 workshop on ML and its applications*, June 1992.

A Extensions du langage Caml

Cette annexe décrit brièvement la syntaxe retenue pour introduire les modules dans le langage Caml, et le lien entre le système de modules et le mécanisme de compilation séparée de Caml.

A.1 Le langage de modules

Le langage est à présent divisé en deux couches. La première est le langage Caml usuel. La seconde est le langage de modules ; il s'agit d'un lambda-calcul où les objets manipulés sont les *structures*, les *foncteurs* jouant le rôle de fonctions des modules dans les modules.

Le langage de modules est d'ordre supérieur, c'est-à-dire qu'un foncteur peut accepter un autre foncteur en argument. Nous utiliserons le terme générique de *module* pour désigner une structure ou un foncteur.

A.1.1 Structures

Une *structure* peut être considérée comme un enregistrement contenant une série de valeurs et de fonctions ; mais à la différence d'un simple enregistrement, une structure peut également contenir des définitions de types. De plus, une structure est un produit dépendant, c'est-à-dire que chaque champ peut faire référence aux précédents. Enfin, de façon naturelle, une structure peut contenir des sous-modules et des définitions de types de modules (*modtypes*).

Voici la liste exhaustive de ce que peut contenir une structure :

- Une expression seule, évaluée pour ses effets de bord.
- Un `let` liant une expression à une variable.
- Une définition de type.
- Une définition d'exception.
- Une définition de sous-module.
- Une définition de type de modules.
- Une définition de primitive C.
- Une directive de compilation.

Une structure s'écrit donc comme une suite de définitions, chacune étant terminée par le classique `;;`. L'ensemble des définitions est délimité par les mots-clef `struct` et `end`. Par exemple :

```
struct
  type t == int;;
  let (x : t) = 3;;
  module S = struct
    type v = A of t;;
  end;;
  modtype W = sig
    type v;;
  end;;
end
```

L'accès aux champs d'une structure se fait selon la notation __ usuelle. Par exemple, si la structure ci-dessus se nomme `M`, on peut accéder au champ `x` par `M__x`.

Valeurs, types et exceptions Les définitions de valeurs, de types et d'exceptions s'écrivent de la même façon qu'en Caml-Light classique. La seule différence est que les identificateurs sont maintenant remplacés par des *chemins* de longueur arbitraire. Ceci vaut pour les noms de variables, de types, de constructeurs, de labels et d'exceptions. Par exemple :

```
let (y : M__S__v) = M__S__A 5;;
```

Sous-modules Une définition de sous-module est de la forme

`module` I `=` m`;;`

où I dénote un identificateur et m une expression du langage de modules.

Définitions de motypes Une définition de type de modules est de la forme

`modtype` I `=` M`;;`

où M dénote une expression de type de modules.

Définitions de primitives C On définit une fonction implémentée par une primitive C en donnant son nom, son type, suivis de l'arité et du nom de la primitive, sous la forme

`primitive` I `:` T `=` *arity name*

Directives de compilation Les directives de compilation sont les mêmes que dans la version classique, à quelques différences près. Les directives `#infix` et `#close` ont disparu pour des raisons techniques. Le comportement de `#open` est modifié, et une nouvelle directive `#import` a été introduite (cf section A.2).

A.1.2 Expressions du langage de modules

Une expression m du langage de modules peut être :

- Un nom de module p, où p est un chemin.
- Une structure (cf A.1.1).
- Un foncteur

  $$\texttt{functor}\ (x_1 : M_1; \ldots x_n : M_n)\, m$$

  Les arguments x_i portent un type explicite M_i. Le corps m du foncteur peut y faire référence.
- Une contrainte de type sur un module $(m : M)$, où m est une expression du langage de modules et M une expression de type de modules.
- L'application d'un foncteur à un argument $m_1\, m_2$. Les applications partielles sont possibles, i.e. les foncteurs à n arguments peuvent être considérés comme du sucre syntaxique.

A.1.3 Expressions de type de modules

Une expression de type de modules peut être de l'une des formes suivantes :

- Un nom de type de modules p.
- Une signature `sig` S `end`. Une signature est la spécification d'une structure ; elle est formée d'une série de déclarations (cf A.1.4).
- Un type de foncteur, de la forme

  $$\texttt{functor}\ (x_1 : M_1; \ldots x_n : M_n)\, M$$

  Notez que les arguments du foncteur apparaissent nommés dans son type ; ceci parce que l'expression de type M_j peut faire référence à x_i pour $i < j$, et de même le type résultat M peut faire référence à tous les x_i. Le type de foncteur est donc un type dépendant.
- Un type de modules muni d'une série de contraintes

  $$M\ \texttt{with}\ t_1\ \texttt{==}\ T_1, \ldots, t_n\ \texttt{==}\ T_n$$

  Ici M doit représenter une signature déclarant des types abstraits $t_1 \ldots t_n$. L'expression "with" ci-dessus équivaut alors à cette même signature, dans laquelle les déclarations abstraites `type` t_i seraient remplacées par des déclarations manifestes `type` t_i `==` T_i. Par exemple, `sig type t;; end with t == int` est équivalent à `sig type t == int;; end`.

A.1.4 Signatures

Une *signature* est en quelque sorte le type d'une structure. Tandis qu'une structure est constituée d'une série de définitions, une signature se compose d'une série de déclarations correspondantes.

Les déclarations possibles sont :

- Une déclaration de valeur :

  `value` I `:` T

  où I est un identificateur et T une expression de type. On rencontre également la forme

  `value` I `:` T `=` *arity name*

  qui permet d'implémenter une fonction par une primitive C. Notez que la première forme est valide même pour les primitives C – une signature n'est censée fournir que des informations de type. Cependant, elle ne permet pas les appels en ligne (*inline*) ; la seconde notation est donc fournie pour des raisons d'efficacité.

- Une déclaration de type. Deux sortes de déclarations de type sont possibles: la déclaration de type abstrait

  `type` I

  et la déclaration de type manifeste

  `type` I `==` T

  La seconde publie l'implémentation du type I, tamdis que la première la cache aux yeux de l'utilisateur.

- Une déclaration d'exception.

- Une déclaration de module

  `module` m : M

- Une déclaration de type de module. Ici aussi deux formes sont possibles, la forme abstraite

  `modtype` I

  et la forme manifeste

  `modtype` I `==` M

- Une définition de primitive C.

- Une directive de compilation.

A.2 Compilation séparée

Les programmes sont toujours organisés en *unités de compilation*. Chaque unité de compilation est constituée d'une interface (contenue dans un fichier `.mli`) et d'une implémentation (contenue dans un fichier `.ml`).

Le contenu du fichier `X.ml` est considéré par le compilateur comme l'intérieur d'une structure nommée `X`. Le contenu du fichier `.mli` est considéré comme l'intérieur d'une signature ; le compilateur vérifie que la structure `X` admet bien cette signature. On retrouve ainsi le comportement du compilateur classique.

Si le fichier `X.ml` définit une variable `x`, on peut comme auparavant y faire référence par `X__x`, mais ce à condition d'avoir préalablement *importé* le fichier `X` par la directive `#import "X"`. Ceci était implicite dans les versions précédentes.

D'autre part, on peut faire référence à cette variable par `x` à condition d'avoir ouvert le fichier `X` grâce à la directive `#open "X"`. Il est possible d'importer et d'ouvrir à la fois un même fichier.

Notons que ces directives de compilation apparaissent non plus seulement à toplevel mais aussi dans une structure quelconque. Leur effet est alors local, i.e. il *prend fin lorsque l'on referme la structure dans laquelle elles apparaissent*. Par ailleurs, la directive `#close` a disparu.

B Sources

Cet annexe contient une partie des sources du compilateur Caml Light modifié décrit dans ce rapport. Afin de rester concis, seuls les modules les plus intéressants ont été inclus, c'est-à-dire le typeur du langage du modules et la partie avant du compilateur. Ce code correspond assez fidèlement aux règles de typage exposées précédemment.

Ce code n'a pas encore atteint sa forme définitive, et est encore sujet à de nombreuses modifications.

- `globals.ml` définit la représentation interne des types.
- `syntax.ml` définit la syntaxe abstraite.
- `modules.mli` définit la représentation des types de modules.
- `mtypes.mli` déclare quelques opérations de base sur les types de modules.
- `include.ml` implémente la relation d'inclusion entre types de modules.
- `mtyping.ml` contient le typeur du langage du modules.
- `front.ml` compile la syntaxe abstraite vers un lambda-calcul intermédiaire.

B.1 globals.ml

```
(* Global symbol tables *)

#open "prim";;
#open "misc";;

type 'a option = None | Some of 'a;;
type 'a pointer == ('a option) ref;;                    (* Used to add information to an existing structure *)

type path =
    Pident of string                                    (* short identifier *)
  | Pdot of path * string * int pointer                 (* long identifier, with room for the physical field number *)
;;

(* The old distinction between qualified and unqualified identifiers is gone.
   Now every identifier is a path.
*)

(* Type constructors *)

type type_stamp = {
    st_stamp: stamp__t;                 (* The actual stamp *)

    mutable st_kind: stamp_kind;        (* This one too *)
    mutable st_link: stamp_link;        (* Instantiation link *)

    st_name: string;                    (* Stamp's short name *)
```

```
    mutable st_prefix: string list      (* Enclosing modules names. Physical equality will be used *)
}

and stamp_kind =
    SKabstract
  | SKvariant of int                    (* number of constructors *)
  | SKrecord of int * (string vect)     (* number of labels and mapping from physical number to name *)

and stamp_link =
    Snolink                             (* Free/abstract stamp *)
  | Slinkto of typ list * typ           (* Parameter list and body *)

(* Type expressions *)

and typ =
  { typ_desc: typ_desc;                 (* What kind of type expression? *)
    mutable typ_level: int }            (* Binding level *)
and typ_desc =
    Tvar of mutable typ_link            (* A type variable *)
  | Tarrow of typ * typ                 (* A function type *)
  | Tproduct of typ list                (* A tuple type *)
  | Tconstr of type_stamp * typ list    (* A constructed type *)
and typ_link =
    Tnolink                             (* Free variable *)
  | Tlinkto of typ                      (* Instantiated variable *)

(* Value constructors *)

and constr_desc =
  { cs_res: typ;                       (* Result type *)
    cs_arg: typ;                       (* Argument type. Unused for constant constructors *)
    cs_mut: mutable_flag;              (* Mutable or not *)
    cs_tag: constr_tag_kind;           (* How the tag is represented *)
    cs_kind: constr_kind }             (* How the argument is represented *)

and mutable_flag =
  Mutable | Notmutable

and constr_tag_kind =
    Tag_exception                       (* Exception constructor *)
  | Tag_regular of int * int            (* Tag number, number of constructors *)

and constr_kind =
    Constr_constant                     (* Constant constructor *)
  | Constr_regular                      (* Usual constructor *)
  | Constr_superfluous of int           (* Superfluous constructor with its arity *)

(* Labels *)

and label_desc =
  { lbl_res: typ;                       (* Result type *)
    lbl_arg: typ;                       (* Argument type *)
    lbl_mut: mutable_flag;              (* Mutable or not *)
```

```
    lbl_pos: int }                      (* Position in the tuple *)
;;

let generic = (-1)                      (* Variables with this level are generic *)
and notgeneric = 0;;                    (* Variables with 0 or greater are not *)

(* Global variables *)

type value_desc =
  { val_typ: typ;                       (* Type *)
    val_prim: prim_desc }               (* Is this a primitive? *)

and prim_desc =
    ValueNotPrim                        (* Regular value *)
  | ValuePrim of int * primitive        (* Arity and implementation *)
;;
```

B.2 syntax.ml

```
(* The abstract syntax for the language *)

#open "const";;
#open "location";;
#open "globals";;

(* Type expressions, as output by the parser.
   Note that a type name is a path.
*)

type type_expression =
    Typexp of type_expression_desc * location
and type_expression_desc =
    Ztypevar of string
  | Ztypearrow of type_expression * type_expression
  | Ztypetuple of type_expression list
  | Ztypeconstr of path * type_expression list
;;

(* In a pattern, the parser is unable to know the difference between a variable and
   a constant constructor. The distinction will be made during the
   typechecking phase. Also, constructor/label names will be replaced with
   full descriptors.
*)

type pattern =
    Pat of mutable pattern_desc * location                    (* The mutable allows the typechecker to modify *)
and pattern_desc =                                            (* the syntax tree *)
    Zwildpat
  | Zaliaspat of pattern * string
  | Zconstantpat of atomic_constant
```

```
  | Ztuplepat of pattern list
  | Zorpat of pattern * pattern
  | Zconstraintpat of pattern * type_expression

  | Zconstruct0pat of path * constr_desc                    (* Created by the typechecker *)
  | Zconstruct1pat of path * constr_desc * pattern
  | Zrecordpat of (label_desc * pattern) list
  | Zvarpat of string

  | ZPrecordpat of (path * pattern) list                    (* Created by the parser *)
  | ZPconstruct0varpat of path                              (* Constant constructor or variable *)
  | ZPconstruct1pat of path * pattern
;;

(* The parser is unable to make the distinction between:
   - An identifier and a constant constructor
   - A function application and a constructor application.
   The distinction will be made only during typechecking.

   Also, the typecheker replaces paths with constructor/label descriptors.
*)

type expression =
    Expr of mutable expression_desc * location              (* The mutable allows the typechecker to add info *)
and expression_desc =
    Zconstant of struct_constant
  | Ztuple of expression list
  | Zlet of bool * (pattern * expression) list * expression
  | Zfunction of (pattern list * expression) list
  | Ztrywith of expression * (pattern * expression) list
  | Zsequence of expression * expression
  | Zcondition of expression * expression * expression
  | Zwhile of expression * expression
  | Zfor of string * expression * expression * bool * expression
  | Zsequand of expression * expression
  | Zsequor of expression * expression
  | Zconstraint of expression * type_expression
  | Zvector of expression list
  | Zassign of string * expression
  | Zstream of stream_component list
  | Zparser of (stream_pattern list * expression) list

  | Zlongident of path * prim_desc                          (* Created by the typechecker *)
  | Zconstruct0 of path * constr_desc
  | Zconstruct1 of path * constr_desc * expression
  | Zapply of expression * expression list
  | Zrecord of (label_desc * expression) list
  | Zrecord_access of expression * label_desc
  | Zrecord_update of expression * label_desc * expression

  | ZPlongident of path                                     (* Created by the parser *)
  | ZPapply of expression * expression list
  | ZPrecord of (path * expression) list
```

```
  | ZPrecord_access of expression * path
  | ZPrecord_update of expression * path * expression

and stream_component =
    Zterm of expression
  | Znonterm of expression

and stream_pattern =
    Ztermpat of pattern
  | Znontermpat of expression * pattern
  | Zstreampat of string
;;

type type_decl =
    Zabstract_type
  | Zvariant_type of constr_decl list
  | Zrecord_type of (string * type_expression * mutable_flag) list
  | Zabbrev_type of type_expression

and constr_decl =
    Zconstr0decl of string
  | Zconstr1decl of string * type_expression * mutable_flag
;;

type directiveu =
    Zdir of string * string
;;

(* ------------------------------------------------------------------------ *)
(* Syntax for the module language *)

type module_type_expression =
    MTypexp of module_type_desc * location
and module_type_desc =
    Zmtypelongident of path
  | Zmtypesignature of signature
  | Zmtypefunctor of (string * module_type_expression) list * module_type_expression
  | Zmtypewith of module_type_expression * (string * string list * type_expression) list

and signature == signature_item list

and signature_item =
    SigItem of signature_item_desc * location
and signature_item_desc =
    Zsigvalue of (string * type_expression * prim_desc) list
  | Zsigtype of (string * string list * type_decl) list
  | Zsigexception of constr_decl list
  | Zsigmodule of string * module_type_expression
  | Zsigmodtype of string * modtype_decl
  | Zsigdirective of directiveu

and modtype_decl =
    Zmodtype_abstract
```

```
  | Zmodtype_manifest of module_type_expression
;;

(* The ZMlongident constructor has a mutable argument. This allows paths which refer to
   a #opened module to be replaced with explicit paths by the typechecker.
*)

type module_expression =
    MExpr of module_expression_desc * location
and module_expression_desc =
    ZMlongident of path
  | ZMstructure of structure
  | ZMfunctor of (string * module_type_expression) list * module_expression
  | ZMapply of module_expression * (module_expression * restriction_info pointer) list
  | ZMconstraint of module_expression * module_type_expression * restriction_info pointer

and structure == structure_item list

and structure_item =
    StructItem of structure_item_desc * location
and structure_item_desc =
    Zexpr of expression
  | Zletdef of bool * (pattern * expression) list
  | Ztypedef of (string * string list * type_decl) list
  | Zexcdef of constr_decl list
  | Zmodule of string * module_expression
  | Zmodtype of string * module_type_expression
  | Zprimitive of (string * type_expression * prim_desc) list
  | Zimpldirective of directiveu

(* Restricting the type of an expression produces no code. However, restricting the type
   of a module expression does, because field numbers change.

   When typechecking a constraint over a module expression, the typechecker
   adds restriction information to the syntax tree, to be used by the compiler.

   The RMap list contains, in ith position, the position in the *old* structure
   of the ith field in the *new* structure, *and* the restriction operation which
   has to performed on this field, since modules can be embedded!
*)

and restriction_info =
    RDoNothing                                          (* no physical action required *)
  | RMap of (int * restriction_info) list               (* mapping of indexes is required *)
  | RFunctor of restriction_info list * restriction_info  (* functor: restricting at both ends is required *)
;;
```

B.3 modules.mli

```
(* Representation of module types + Handling of external modules *)

#open "misc";;
#open "const";;
#open "prim";;
#open "globals";;
#open "errors";;

type SigElement =
    SEValue of typ * prim_desc                                          (* Type + various flags *)
  | SEType of typ list * typ * (typedef_kind ref)                       (* Parameters + body + def kind *)
  | SEConstructor of constr_desc
  | SELabel of label_desc
  | SEModule of module_typ
  | SEModtype of stamp_set * module_typ * (typedef_kind ref)            (* Instantiable stamps + body + def kind *)

(* Each type definition carries a flag telling whether it's abstract (this case includes concrete types)
   or manifest. It's used by the inclusion checking routines, the printing routines, and is also useful when
   applying 'with' constraints. Because such a constraint might turn an abstract type into a manifest type,
   the field uses a ref.
*)

and typedef_kind =
    Def_manifest
  | Def_abstract

(* An environment is a list of signature elements, possibly marked with a physical
   position which will be used by the compiler to generate structure access code.
   Values, modules, and exception constructors have physical slots. Other environment
   elements do not.
*)

and environment == (string * physical_position * SigElement) list

and physical_position == int option

(* A functor type contains:
   + For each argument:
     - the parameter name, for printing purposes only
     - the set of the stamps that can be instantiated
     - the argument type
   + the set of the stamps that have to be generated every time the functor is applied.
   + the result type
*)

and module_typ =
    MTsignature of environment                                          (* A signature is an environment *)
  | MTfunctor of (string * stamp_set * module_typ) list * stamp_set * module_typ
  | MTabstract of mtype_stamp

and mtype_stamp = {
    mst_stamp: stamp__t;                                                (* The actual stamp *)
```

```
    mutable mst_link: mstamp_link;                                  (* Instantiation link *)
    mst_name: string;                                               (* Stamp's short name *)
    mutable mst_prefix: string list                                 (* List of enclosing structures *)
  }

and mstamp_link =
    MSnolink
  | MSlinkto of module_typ

and stamp_set = {
    stamps:   type_stamp list;
    mstamps: mtype_stamp list
  }
;;

value use_extended_zi : bool ref;;                                  (* Loading external interfaces *)
value read_external_module : string -> stamp_set * environment;;
value load_external_module : string -> stamp_set * environment;;

value mtype_repr : module_typ -> module_typ;;                       (* Following stamp links *)


exception Desc_not_found;;                                          (* Raised by the find* routines *)

value find_value : string -> environment -> physical_position * (typ * prim_desc);;
value find_module : string -> environment -> physical_position * module_typ;;
value find_constructor : string -> environment -> physical_position * constr_desc;;
value find_type : string -> environment -> typ list * typ * (typedef_kind ref);;
value find_modtype : string -> environment -> stamp_set * module_typ * (typedef_kind ref);;
value find_label : string -> environment -> label_desc;;
value find_value_or_constructor : string -> environment -> physical_position * SigElement;;

exception Not_a_structure of path * module_typ;;                    (* Raised by the pfind* routines *)

value pfind_value : environment -> path -> typ * prim_desc;;
value pfind_module : environment -> path -> module_typ;;
value pfind_type : environment -> path -> typ list * typ * (typedef_kind ref);;
value pfind_modtype : environment -> path -> stamp_set * module_typ * (typedef_kind ref);;
value pfind_label : environment -> path -> label_desc;;
value pfind_constructor : environment -> path -> constr_desc;;
value pfind_value_or_constructor : environment -> path -> SigElement;;


value empty_set : stamp_set;;                                       (* Stamp handling routines *)
value union_set : stamp_set -> stamp_set -> stamp_set;;
value compare_stamps : type_stamp -> type_stamp -> bool;;
value compare_mstamps : mtype_stamp -> mtype_stamp -> bool;;
value stamp_memq : type_stamp -> type_stamp list -> bool;;
value mstamp_memq : mtype_stamp -> mtype_stamp list -> bool;;
value stamp_assq : type_stamp -> (type_stamp * 'a) list -> 'a;;
value mstamp_assq : mtype_stamp -> (mtype_stamp * 'a) list -> 'a;;
value remove_stamps : stamp_set -> type_stamp list -> stamp_set;;
value prefix_stamps : string -> stamp_set -> unit;;
```

```
value diff_env : 'a list -> 'a list -> 'a list;;                    (* Environment handling utility *)
```

B.4 mtypes.mli

```
(* Basic operations on module types.

   copy_quantified_module_stamps takes a stamp set N and a module type T. It creates a set
   of new stamps N' and returns T[N <- N']. This is useful before instantiating stamps in N.

   copy_functor_type makes a copy of a functor type (i.e. replaces its quantifiers with
   new stamps).
*)

#open "location";;
#open "syntax";;
#open "globals";;
#open "modules";;

value copy_functor_type : module_typ -> module_typ;;
value copy_quantified_module_type : stamp_set * module_typ -> stamp_set * module_typ;;
value copy_quantified_sig: stamp_set * environment -> stamp_set * environment;;

value prohibit_rebindings : location -> environment -> unit;;

value merge_constraint : location -> environment -> environment ->
                         string * string list * type_expression -> type_stamp;;
```

B.5 include.ml

```
#open "misc";;
#open "globals";;
#open "syntax";;
#open "modules";;
#open "builtins";;
#open "types";;
#open "ty_error";;
#open "mtypes";;

(* ------------------------------------------------------------------------ *)
(* Inclusion between module types.
   t1 = actual type
   t2 = declared type. t2 must "have less fields"
   inst = set of stamps that can be instantiated in the declared type

   What the function does:
   - Make sure that t1 is less general than t2, raise an exception otherwise.
     t1 is left unchanged ; t2 can be instantiated.
   - Return restriction information to be used by the compiler.
     For signatures, this information consists of an injection from the fields of the declared
     signature into the fields of the actual signature. For functors, it consists
     of restriction information for each argument and for the result.
```

```
   The rules are particularly messy when it comes to comparing functor types. I think
   I've finally got it straight:
   - Comparing two functor types shoulnd't instantiate anything (I mean, it should, but only
     internally ; t1 and t2 should be left unmodified). So we work on copies.
   - We can safely ignore the 'inst' parameter, even if it's non-empty ; because stamps can
     only be instantiated when comparing typedefs, and there cannot be any typedefs for
     external stamps within a functor.
   - We first compare the argument types in contravariant order, allowing '1' stamps
     to be instantiated. Then we compare the result types in covariant order, allowing
     '2' stamps to be instantiated.
   - In the process, we collect restriction information and build the appropriate RFunctor
     structure. We can do some simple optimization by replacing RFunctor(RDoNothing .... RDoNothing)
     with RDoNothing. This is the purpose of the 'trivial' flag.

   Note that abstract module type stamps are not instantiated here. They are instantiated only
   when comparing modtype definitions.

   Note that include_signature might raise exception Inclusion with an error message. The caller
   is then responsible for calling sig_inclusion_err loc sig1 sig2 msg. This is because sig_inclusion_err
   won't work if the signatures have been partially instantiated - so the caller must have an unaltered copy
   at hand. I know, it makes things slower. Uncool.
*)

let rec include_module_type loc t1 t2 inst =
  match (mtype_repr t1, mtype_repr t2) with

    MTsignature s1, MTsignature s2 ->                                          (* Signatures *)
      let (_, backup2) = copy_quantified_sig (inst, s2) in                     (* Backup copy for printing *)
      begin try
        include_signature loc s1 s2 inst
      with Inclusion msg ->
        sig_inclusion_err loc s1 backup2 msg
      end

  | MTfunctor _, MTfunctor _ ->                                                (* Functor types *)

      begin match (copy_functor_type t1, copy_functor_type t2) with            (* Make copies *)
        MTfunctor(paraml1, _, result1), MTfunctor(paraml2, gener2, result2) ->

          let rec handle_arguments trivial restr_list = fun
            ((_, inst1, arg1)::rest1) ((_, _, arg2)::rest2) ->                 (* For each argument *)
              let restriction =
                include_module_type loc arg2 arg1 inst1 in                     (* Compare the argument types *)
              let trivial = (restriction = RDoNothing) & trivial in            (* For optimization purposes *)
              handle_arguments trivial (restriction::restr_list) rest1 rest2
          | [] [] ->
              restr_list, trivial
          | _ _ ->
              inclusion_err loc t1 t2 "These functors have different arities."

          in let restr_list, trivial = handle_arguments true [] paraml1 paraml2
          in let res_restriction =                                             (* Compare result types *)
```

```
            include_module_type loc result1 result2 gener2 in

          if trivial & (res_restriction = RDoNothing) then RDoNothing          (* If possible, optimize *)
          else RFunctor(rev restr_list, res_restriction)

      | _ ->
          fatal_error "include_module_type"                                    (* Can't happen *)
      end

  | MTabstract stamp1, MTabstract stamp2 ->                                    (* Abstract types *)
      if compare_mstamps stamp1 stamp2 then RDoNothing                         (* This is a dummy *)
      else abstract_mtyp_mismatch_err loc t1 t2

  | _ ->
      inclusion_err loc t1 t2 "These module types are incompatible."

(* ------------------------------------------------------------------------- *)
(* Inclusion between signatures

   include_signature has to build a RMap structure telling the compiler which
   fields to keep and which to discard when restricting the structure.

   Type stamps are instantiated only when comparing two definitions together, not
   when comparing values. Hence, signatures must be compared in their creation order.

   In order to allow permutations between fields, the comparison algorithm works
   in time O(mn). For each element of sig2, it searches sig1 for an element with
   the same name.
*)

and phys_length = function                                                     (* Computes the physical length of a sig *)
    [] -> 0
  | (_, _, (SEValue _ | SEModule _))::rest ->
      succ (phys_length rest)                                                  (* Values and modules take up one slot *)
  | (_, _, SEConstructor { cs_tag = Tag_exception }) :: rest ->
      succ (phys_length rest)                                                  (* So do exceptions *)
  | _ :: rest ->
      phys_length rest

and include_signature loc sig1 sig2 inst =
  let mapping = make_vect (phys_length sig2) (666, RDoNothing)                 (* 666 is crucial here *)

  and physical = function                                                      (* Small utility function *)
      Some i -> i
    | None -> fatal_error "Physical pos info missing in include_signature" in

  let do_element (id, pos2, elem2) = match elem2 with                          (* For each element in sig2 *)

    SEValue (typ2, prim2) -> begin try
      let (pos1, (typ1, prim1)) = find_value id sig1 in                        (* Get the corresponding element from sig1 *)
      include_type typ1 typ2;                                                  (* Check for inclusion *)
      mapping.(physical pos2) <- (physical pos1, RDoNothing);                  (* Update the mapping *)
```

```
    match prim2 with
      ValueNotPrim ->
        ()
    | ValuePrim _ ->                                          (* If the value is declared as a primitive *)
        if prim1 = ValueNotPrim then                          (* Make sure it is defined so *)
          raise Unify;
        if prim1 <> prim2 then                                (* With the same parameters *)
          raise (Inclusion ("Declarations for primitive " ^ id ^ " do not match."))

  with
    Desc_not_found -> raise (Inclusion ("Value " ^ id ^ " is undefined."))
  | Unify -> raise (Inclusion ("Declarations for value " ^ id ^ " do not match."))
  end

| SEType def2 -> begin try
    let def1 = find_type id sig1 in
    include_typedef inst id def1 def2                         (* Compare type definitions *)
  with
    Desc_not_found -> raise (Inclusion ("Type " ^ id ^ " is undefined."))
  | Unify -> raise (Inclusion ("Declarations for type " ^ id ^ " do not match."))
  end

| SEConstructor desc2 -> begin try                            (* Constructors *)
    let pos1, desc1 = find_constructor id sig1 in
    if desc1.cs_mut <> desc2.cs_mut or desc1.cs_tag <> desc2.cs_tag or desc1.cs_kind <> desc2.cs_kind then
      raise Unify;
    let ty1 = type_arrow(desc1.cs_arg, desc1.cs_res)
    and ty2 = type_arrow(desc2.cs_arg, desc2.cs_res)
    in equal_type ty1 ty2;
    if desc1.cs_tag = Tag_exception then                      (* If they're exceptions *)
      mapping.(physical pos2) <- (physical pos1, RDoNothing)  (* Update the mapping *)
  with
    Desc_not_found -> raise (Inclusion ("Constructor " ^ id ^ " is undefined."))
  | Unify -> raise (Inclusion ("Declarations for constructor " ^ id ^ " do not match."))
  end

| SELabel desc2 -> begin try
    let desc1 = find_label id sig1 in
    if desc1.lbl_mut <> desc2.lbl_mut or desc1.lbl_pos <> desc2.lbl_pos then
      raise Unify;
    let ty1 = type_arrow(desc1.lbl_arg, desc1.lbl_res)
    and ty2 = type_arrow(desc2.lbl_arg, desc2.lbl_res)
    in equal_type ty1 ty2
  with
    Desc_not_found -> raise (Inclusion ("Label " ^ id ^ " is undefined."))
  | Unify -> raise (Inclusion ("Declarations for label " ^ id ^ " do not match."))
  end

| SEModule mty2 -> begin try                                  (* Sub-modules *)
    let pos1, mty1 = find_module id sig1 in
    let restriction = include_module_type loc mty1 mty2 inst in   (* Compare their types *)
    mapping.(physical pos2) <- (physical pos1, restriction)       (* And store the resulting restriction_info *)
  with
```

```
      Desc_not_found -> raise (Inclusion ("Module " ^ id ^ " is undefined."))
    end

  | SEModtype def2 -> begin try                                          (* Module type definitions *)
      let def1 = find_modtype id sig1 in
      include_modtypedef loc inst id def1 def2                          (* Compare the definitions *)
    with
      Desc_not_found -> raise (Inclusion ("Module type " ^ id ^ " is undefined."))
    end

  in do_list do_element (rev sig2);

  let mapping = list_of_vect mapping in                                 (* Turn the resulting mapping into a list *)

  let trivial list = trivial_rec 0 list                                 (* Check whether it is trivial, i.e. *)
    where rec trivial_rec i = function                                  (* All fields keep the same number and *)
      [] ->                                                             (* contents *)
        (i = list_length list)
    | (index, RDoNothing)::rest ->
        (index = i) & (trivial_rec (succ i) rest)
    | _ ->
        false

  in if (phys_length sig1 = phys_length sig2) & (trivial mapping)       (* If it is trivial, then we can return *)
  then RDoNothing                                                       (* RDoNothing instead of RMap mapping *)
  else RMap mapping

(* ------------------------------------------------------------------------ *)
(* Inclusion between type definitions *)

(* Check for equality of two types. Variables might be physically different though,
   so I guess the simplest way is to use the existing 'filter' function in both ways.
   Taking instances is necessary for two reasons:
   1. We don't want the type to be destroyed
   2. filter expects that its first argument contains no generic variables.
*)

and equal_type ty1 ty2 =
  filter (type_instance ty1) ty2;
  filter (type_instance ty2) ty1

and include_type ty1 ty2 =
  filter (type_instance ty1) ty2

(* include_typedef compares two type definitions. Here are the rules :
   - If the declared type is manifest, make sure that the two type expressions are equal.
   - If the declared type is abstract, and appears in 'inst', allow it to be instantiated.
   - If the declared type is concrete, and appears in 'inst', allow it to be instantiated, but only
     with another concrete type of the same kind (variant/record and number of constructors/labels).
     Individual constructors/labels are compared later, which ensures that the type definitions as a whole match.

     When we instantiate a stamp, we do not change the corresponding "kind" field. This is because the two
     stamps actually represent the same type, so instantiating doesn't make the type manifest.
```

```
*)

and include_typedef inst id ((paraml1, body1, kind1) as def1) (paraml2, body2, kind2) =
  if list_length paraml1 <> list_length paraml2 then                    (* Check arity *)
    raise (Inclusion ("Type arity mismatch in the definitions of type " ^ id ^ "."));

  let instantiate stamp2 (paraml, body, kind) =                         (* Instantiate a stamp *)
    match body.typ_desc with
      Tconstr(stamp1, _) ->                                             (* Check for circular link first *)
        if compare_stamps stamp1 stamp2 then ()                         (* It's already the right stamp - do nothing *)
        else
          stamp2.st_link <- Slinkto (paraml, body)
  | _ ->
    stamp2.st_link <- Slinkto (paraml, body) in

  if !kind2 = Def_abstract then begin
    match (type_repr(body1)).typ_desc, (type_repr(body2)).typ_desc with (* Instantiate stamp2 if necessary *)

      _, Tconstr({ st_kind = SKabstract } as stamp2, _) ->              (* Declared type is abstract *)
        if stamp_memq stamp2 inst.stamps then                           (* If the stamp is instantiable, go *)
          instantiate stamp2 def1

    | Tconstr (stamp1, _), Tconstr(stamp2, _) ->                        (* Declared type is a concrete datatype *)
        if (stamp_memq stamp2 inst.stamps)                              (* If it's instantiable and the defs. *)
         & (stamp1.st_kind = stamp2.st_kind) then                       (* match, instantiate. Constructors/labels *)
          instantiate stamp2 def1                                       (* will be compared later *)

    | _ ->                                                              (* Avoid non-exhaustive pattern matching *)
        ()
  end;

  equal_type body1 body2                                                (* Now compare the types *)

(* ------------------------------------------------------------------------ *)
(* Comparing module type definitions.

   This is similar to comparing type definitions. The two modtypes must be equal
   for the inclusion to hold, because later the definition might be used in covariant
   or contravariant position.
*)

and include_modtypedef loc inst id (stamps1, body1, kind1) (stamps2, body2, kind2) =

  let instantiate stamp2 body1= match body1 with                        (* Instantiation function *)
    MTabstract stamp1 ->
      if compare_mstamps stamp1 stamp2 then ()
      else stamp2.mst_link <- MSlinkto body1
  | _ ->
      stamp2.mst_link <- MSlinkto body1 in

  if !kind2 = Def_abstract then begin
    match body2 with
      MTabstract stamp2 ->                                              (* If the declared type is abstract *)
```

```
        if mstamp_memq stamp2 inst.mstamps then                          (* And instantiable *)
          instantiate stamp2 body1                                       (* Then instantiate it *)
    | _ ->
        ()
  end;

  let (stamps2, body2) = copy_quantified_module_type (stamps2, body2)    (* This is an ugly hack to request equality *)
    in include_module_type loc body1 body2 stamps2;                      (* Really weird since it would give bizarre *)
  let (stamps1, body1) = copy_quantified_module_type (stamps1, body1)    (* messages, should an error occur *)
    in include_module_type loc body2 body1 stamps1;
  ()                                                                     (* Dump restriction info *)
;;
```

B.6 mtyping.ml

```
#open "misc";;
#open "globals";;
#open "syntax";;
#open "errors";;
#open "modules";;
#open "types";;
#open "ty_error";;
#open "typing";;
#open "ty_decl";;
#open "mtypes";;
#open "include";;
#open "lexer";;

(* ------------------------------------------------------------------------ *)
(* Executing directives.

   #open adds all components of the specified module to the environment.
   This allows functions from an external module to be called by their short names.
   #import adds the specified module to the current environment. This allows functions from
   an external module to be called by their full names.

   Note that these directives add things to the current environment. This implies
   that their scope is limited to the current structure, since the environment will
   be dumped when we exit the structure.

   These directives are also handled by the compiler, in order to produce references
   to global variables.

   #infix and #uninfix have been removed because the whole file is parsed in one
   fell swoop, so they never have the opportunity to act.
   #close has been removed because the new implementation of #open makes it more
   difficult to implement, and even less useful than before!
*)

let add_open_to_typ_env env modname =
  let signature = snd (load_external_module modname) in
  signature @ env
;;
```

```
(* Since the external module's signature is cached and can be imported several times, we
   can't call prefix_stamps here. What we do is that external modules' sigantures are
   prefixed once when they're created, and never afterwards.
*)

let add_import_to_typ_env env modname =
  let stamps, signature = load_external_module modname in
  prefix_stamps modname stamps;
  let item = (modname, None, SEModule (MTsignature signature)) in
  item :: env
;;

let do_directive ((position, env, cur_sig, stamps) as param) = function
    Zdir("open", name) ->
      (position, add_open_to_typ_env env name, cur_sig, stamps)
  | Zdir("import", name) ->
      (position, add_import_to_typ_env env name, cur_sig, stamps)
  | Zdir("directory", dirname) ->
      load_path := dirname :: !load_path;
      param
  | Zdir(d, name) ->
      prerr_begline " Warning: unknown directive \"";
      prerr_string d;
      prerr_endline2 "\", ignored.";
      param
;;

(* ------------------------------------------------------------------------ *)
(* Converting a module type expression to a module type *)
(* Returns the list of newly created stamps and the module type *)

let rec type_of_mtype_expression env (MTypexp(desc, loc)) =
  match desc with

    Zmtypelongident path -> begin try                                    (* Module type name *)
      let stamps, mty, _ = pfind_modtype env path in                     (* Look it up and take an instance of it *)
      copy_quantified_module_type (stamps, mty)
    with                                                                 (* This simulates inline expansion *)
      Desc_not_found -> unbound_err loc "Module type" path
    | Not_a_structure (path, mty) -> not_a_structure_err loc path mty
    end

  | Zmtypesignature signature ->                                         (* Signature *)
      let stamps, signature = sig_of_sig_expression env signature in     (* Analyze it *)
      prohibit_rebindings loc signature;                                 (* Make sure there are no rebindings *)
      stamps, MTsignature signature

  | Zmtypefunctor (argl, result) ->                                      (* Functor type *)
      let rec handle_arguments env argtyl = function
        (name, ty_expr) :: rest ->                                       (* For each argument *)
          let stamps, mty = type_of_mtype_expression env ty_expr in      (* Analyze its type *)
          let env = (name, None, SEModule mty)::env in                   (* Add it to the environment *)
```

```
          prefix_stamps name stamps;                                    (* Update stamps names *)
          let argtyl = (name, stamps, mty) :: argtyl in
          handle_arguments env argtyl rest                              (* On to the next argument *)
      | [] ->
          let stamps, mty = type_of_mtype_expression env result in      (* Analyze the result type *)
          empty_set, MTfunctor(rev argtyl, stamps, mty)                 (* Build the functor type *)

      in handle_arguments env [] argl

  | Zmtypewith (mty, constraints) ->                                    (* Type expression + 'with' constraints *)
      let stamps, mty = type_of_mtype_expression env mty in             (* Evaluate the body *)
      let mty = mtype_repr mty in
      match mty with
        MTsignature signature ->                                        (* Merge constraints into the signature *)
          let bound_stamps = map (merge_constraint loc env signature) constraints in
          let remaining_stamps = remove_stamps stamps bound_stamps in   (* Remove instantiated stamps *)
          remaining_stamps, MTsignature signature
      | _ ->
          cant_apply_with_err loc mty                                   (* If it isn't a signature, fail *)

(* type_sig_item is quite straightforward. Note that a counter is used to
   to enter the fields into the signature with the right physical position information.
   counter == Some (integer).

   Functors (and modtypes) quantifiers (== owned stamps) are stored "locally", so they
   don't appear in the list of created stamps.
   Note that if an abstract modtype is defined, its stamp is not a quantifier - it belongs
   to the signature.

   type_valuedecl handles value declarations. It is also used by type_module_expr
   to handle primitive declarations.

   sig_of_sig_expression iterates over the items in the signature expression.
   It maintains the environment, and also the signature being built. This can seem
   redundant; it is not, because #imported modules appear in the environment, but
   not in the signature being built.
*)

and type_valuedecl ((position, env, cur_sig, stamps) as param) decllist =
  let rec loop ((position, env, cur_sig, stamps) as param) = function
    (name, ty_expr, prim_desc) :: rest ->                               (* For each value *)
      push_type_level();
      let var_map = ref ([] : name2var) in
      let typ = type_of_type_expression false env var_map ty_expr in    (* Analyze the type expression *)
      pop_type_level();
      generalize_type typ;                                              (* Generalize the type *)
      let item = (name, position, SEValue(typ, prim_desc)) in           (* Update the environment *)
      loop (next_slot position, item::env, item::cur_sig, stamps) rest  (* On to the next value *)
  | _ ->
      param

  in loop param decllist
```

```
(* position     Number of the next free slot in the current signature
   env          Current environment
   cur_sig      Signature being built
   stamps       Set of stamps created while typing the current sig
*)

and type_sig_item ((position, env, cur_sig, stamps) as param) (SigItem(desc, loc)) =
  match desc with

    Zsigvalue deflist ->                                          (* Value declaration *)
      type_valuedecl param deflist

  | Zsigtype decl ->                                              (* Type declaration *)
      let new_env, new_stamps = type_typedecl env loc decl in     (* Update the environment *)
      let cur_sig = (diff_env env new_env) @ cur_sig in           (* Dirty, but more readable *)
      (position, new_env, cur_sig, union_set new_stamps stamps)   (* Add the stamps to the set of new stamps *)

  | Zsigexception decl ->                                         (* Exception declaration *)
      let new_env, position = type_excdecl env position decl in   (* Each exception uses a physical slot *)
      let cur_sig = (diff_env env new_env) @ cur_sig in
      (position, new_env, cur_sig, stamps)

  | Zsigmodule (name, expr) ->                                    (* Module declaration *)
      let new_stamps, mty = type_of_mtype_expression env expr in
      let item = (name, position, SEModule mty) in
      prefix_stamps name new_stamps;                              (* Update stamps names *)
      (next_slot position, item::env, item::cur_sig, union_set new_stamps stamps)

  | Zsigmodtype (name, Zmodtype_manifest expr) ->                 (* Manifest modtype declaration *)
      let (set, mtyp) = type_of_mtype_expression env expr in
      let item = (name, None, SEModtype (set, mtyp, ref Def_manifest)) in
      (position, item::env, item::cur_sig, stamps)                (* No new stamps added to the current sig *)

  | Zsigmodtype (name, Zmodtype_abstract) ->                      (* Abstract modtype declaration *)
      let mstamp = {
        mst_stamp = stamp__new();                                 (* Create a new abstract stamp *)
        mst_link = MSnolink;
        mst_name = name;
        mst_prefix = []
      } in
      let item = (name, None, SEModtype(empty_set, MTabstract mstamp, ref Def_abstract)) in
      let stamps = {
        stamps = stamps.stamps;
        mstamps = mstamp :: stamps.mstamps                        (* Add the stamp to the set of new stamps *)
      }
      in (position, item::env, item::cur_sig, stamps)

  | Zsigdirective dir ->
      do_directive param dir

and sig_of_sig_expression env sig_expr =
  let (_, _, signature, new_stamps) = it_list type_sig_item (Some 0, env, [], empty_set) sig_expr in
  new_stamps, signature
```

```
;;

(* ------------------------------------------------------------------------ *)
(* Typing a module expression *)
(* The function returns the list of newly created stamps and the module type *)

let rec type_module_expr env (MExpr(desc, loc)) =
  match desc with

    ZMlongident path -> begin try                                        (* Module name *)
      empty_set, pfind_module env path
    with
      Desc_not_found -> unbound_err loc "Module" path
    | Not_a_structure (path, mty) -> not_a_structure_err loc path mty
    end

  | ZMstructure struc ->                                                 (* Structure *)
      let stamps, signature = type_structure env loc struc in
      stamps, MTsignature signature

  | ZMfunctor (argl, result) ->                                          (* Functor *)
      let rec handle_arguments env paramtyl = function
        (name, ty_expr) :: rest ->
          let stamps, mty = type_of_mtype_expression env ty_expr in      (* Analyze the argument type *)
          let env = (name, None, SEModule mty)::env in                  (* Add it to the environment *)
          prefix_stamps name stamps;
          let paramtyl = (name, stamps, mty)::paramtyl in
          handle_arguments env paramtyl rest                            (* On to the next argument *)
      | [] ->
          let stamps, mty = type_module_expr env result in              (* Get the type of the functor's body *)
          empty_set, MTfunctor(rev paramtyl, stamps, mty)               (* Finally, build the functor type *)

      in handle_arguments env [] argl

  | ZMconstraint (expr, typ, restriction) ->                             (* Expression with type restriction *)
      let _, actual_type = type_module_expr env expr in                 (* Typecheck the expression *)

      let (new_stamps, declared_type) as x =                            (* Analyze the declared type *)
        type_of_mtype_expression env typ in

      let inst, declared_type2 = copy_quantified_module_type x in       (* Make a copy of it *)
      restriction :=                                                    (* Test for inclusion *)
        Some (include_module_type loc actual_type declared_type2 inst); (* And keep the restriction info *)

      new_stamps, declared_type                                         (* Return the un-instantiated type *)

(* Functor application. The tricky part is handling the stamp sets correctly. See the typing rule in
   the report.
   The returned stamp set must contain:
     - The functor result's generative stamps (types declared in the functor body)
     - Each argument's own stamps (types declared in the body of the effective argument, which
       may pass through manifest types into the result type).
   Actually, the argument's stamps might appear in the result type, and they might not. I think
```

```
   it doesn't matter if they don't appear - too many stamp quantifiers shouldn't make any difference.

   When we reach the last functor argument, 'gener' contains the functor result's stamps.
   When handling the other arguments, it should be the empty stamp set, since 'body' is a functor.
*)

  | ZMapply (e1, arg_restr_list) ->                                    (* Functor application *)
      let _, ty1 = type_module_expr env e1 in                         (* This should be a functor -> no new stamps *)
      let ty1 = copy_functor_type ty1 in                              (* Make a copy *)

      let rec loop body stamps = function
        (arg, info) :: rest ->                                        (* For each argument *)
          let arg_stamps, argty = type_module_expr env arg in         (* Typecheck the argument *)
          let body = mtype_repr body in
          begin match body with
            MTfunctor((name, inst, paramty)::_, _, _) ->              (* Make sure the left term is a functor *)
              info := Some(include_module_type loc argty paramty inst); (* Check argument type and store restr. info *)
              let gener, result_ty = make_result_type body in         (* Get the partial result type *)
              let stamps = union_set gener (union_set arg_stamps stamps)(* Collect the new stamps *)
              in loop result_ty stamps rest
          | _ ->
              not_a_functor_err loc body
          end
      | [] ->
          stamps, body                                                (* Return the result *)

      in loop ty1 empty_set arg_restr_list

and type_structure env loc struc =
    let _, _, signature, new_stamps =                                 (* Handle each field in sequence *)
      it_list type_structure_item (Some 0, env, [], empty_set) struc in
    prohibit_rebindings loc signature;                                (* Make sure no name is bound twice *)
    new_stamps, signature

(* make_result_type takes a functor and returns its type, minus the first argument *)
(* It also returns a set of generated stamps. If the result is still a functor, this set is empty;
   if it is a structure, the set is the set of generative stamps.
*)

and make_result_type = function
  MTfunctor([_], gener, result_ty) ->                                 (* Functor had only one argument *)
    gener, result_ty
| MTfunctor(_ :: paraml, gener, result_ty) ->                         (* Functor with several arguments *)
    empty_set, MTfunctor(paraml, gener, result_ty)
| _ ->
    fatal_error "Unexpected argument to make_result_type"

(* ------------------------------------------------------------------------ *)
(* Typing a structure element.
   The function takes as argument a tuple containing:
     - the item's physical position in the current structure
     - the current environment
     - the signature being built
```

```
     - the list of stamps generated while typing the current structure
  It returns a similar, updated, tuple.
*)

and type_structure_item ((position, env, cur_sig, stamps) as param) (StructItem(desc, loc)) =
  match desc with

    Zexpr expr ->                                                    (* Expression alone *)
      push_type_level();
      let ty = new_type_var() in
      let var_map = ref ([] : name2var) in
      type_expr var_map env [] expr ty;                              (* Just make sure it's correctly typed *)
      pop_type_level();
      generalize_type ty;
      param                                                          (* The environment isn't modified *)

  | Zletdef(rec_flag, pat_expr_list) ->                              (* Value definition *)
      push_type_level();
      let mapping_desc = ref([] : name2var) in                       (* Reset the name2var mapping *)

      let pat_ty_list, expr_list, ty_list = it_list                  (* Prepare fresh type variables *)
        (fun (ptlist, elist, tlist) (pat, expr) ->                   (* for the pattern matching *)
          let alpha = new_type_var() in
          (pat, alpha, Notmutable)::ptlist, expr::elist, alpha::tlist)
        ([], [], []) pat_expr_list in

      let local_env =
        type_pattern_list2 mapping_desc env pat_ty_list in           (* Type the patterns *)

      let enter_env (position, env, cur_sig) (name, (ty, _)) =       (* Compute the new environment *)
        let item = (name, position, SEValue(ty, ValueNotPrim)) in
        (next_slot position, item::env, item::cur_sig) in

      let position, new_env, cur_sig =
        it_list enter_env (position, env, cur_sig) local_env in

      do_list2                                                       (* Typecheck the bindings *)
        (type_expr mapping_desc (if rec_flag then new_env else env) [])
      expr_list ty_list;

      pop_type_level();                                              (* Generalize the resulting types *)
      do_list (fun (_, (ty, _)) -> generalize_type ty) local_env;

      (position, new_env, cur_sig, stamps)                           (* Return the updated environment *)

  | Ztypedef decl ->                                                 (* Type definition *)
      let new_env, new_stamps = type_typedecl env loc decl in        (* Update the environment *)
      let cur_sig = (diff_env env new_env) @ cur_sig in
      (position, new_env, cur_sig, union_set new_stamps stamps)      (* And return the newly created stamps *)

  | Zexcdef decl ->                                                  (* Exception definition *)
      let new_env, position = type_excdecl env position decl in      (* Exceptions take up slots too *)
      let cur_sig = (diff_env env new_env) @ cur_sig in
```

```
      (position, new_env, cur_sig, stamps)

  | Zmodule (name, expr) ->                                          (* Module expression *)
      let new_stamps, mty = type_module_expr env expr in            (* Typecheck the expression *)
      let position = next_slot position in                          (* Reserve a slot for the module *)
      let item = (name, position, SEModule mty) in                  (* Update the environment *)
      prefix_stamps name new_stamps;
      let stamps = union_set new_stamps stamps in                   (* And the stamps list *)
      (position, item::env, item::cur_sig, stamps)

  | Zmodtype (name, expr) ->                                        (* Module type definition *)
      let new_stamps, mty = type_of_mtype_expression env expr in    (* Convert the type expression to a type *)
      let item = (name, None,
                  SEModtype (new_stamps, mty, ref Def_manifest)) in      (* Enter it into the environment *)
      (position, item::env, item::cur_sig, stamps)

  | Zprimitive deflist ->                                           (* Primitive definition *)
      type_valuedecl param deflist

  | Zimpldirective dir ->                                           (* Implementation directive *)
      do_directive param dir
;;
```

B.7 front.ml

```
(* front.ml : translation abstract syntax -> extended lambda-calculus. *)

#open "misc";;
#open "const";;
#open "globals";;
#open "syntax";;
#open "location";;
#open "builtins";;
#open "modules";;
#open "lambda";;
#open "prim";;
#open "match";;
#open "tr_env";;
#open "trstream";;
#open "ty_error";;

(* ------------------------------------------------------------------------- *)
(* To optimize structured constants *)

exception Not_constant;;

let extract_constant = function
    Lconst cst -> cst
  |       _    -> raise Not_constant
;;

(* ------------------------------------------------------------------------- *)
(* To handle recursive definitions *)
```

```
let rec check_letrec_expr (Expr(e,loc)) =
  match e with
    Zlongident _ -> ()
  | Zconstant _ -> ()
  | Ztuple el -> do_list check_letrec_expr el
  | Zconstruct0(path, cstr) -> ()
  | Zconstruct1(path, cstr, expr) ->
      check_letrec_expr expr;
      begin match cstr.cs_kind with
        Constr_superfluous n ->
          begin match expr with
            Expr(Ztuple _, _) -> ()
          | _ -> illegal_letrec_expr loc
          end
      | _ -> ()
      end
  | Zfunction _ -> ()
  | Zconstraint(e,_) -> check_letrec_expr e
  | Zvector el -> do_list check_letrec_expr el
  | Zrecord lbl_expr_list ->
      do_list (fun (lbl,expr) -> check_letrec_expr expr) lbl_expr_list
  | Zparser _ -> ()
  | Zstream _ -> ()
  | _ ->
      illegal_letrec_expr loc
;;

let rec size_of_expr (Expr(e,loc)) =
  match e with
    Ztuple el ->
      do_list check_letrec_expr el; list_length el
  | Zconstruct1(path, cstr, expr) ->
      check_letrec_expr expr;
      begin match cstr.cs_kind with
        Constr_superfluous n -> n | _ -> 1
      end
  | Zfunction _ ->
      2
  | Zconstraint(e,_) ->
      size_of_expr e
  | Zvector el ->
      do_list check_letrec_expr el; list_length el
  | Zrecord lbl_expr_list ->
      do_list (fun (lbl,expr) -> check_letrec_expr expr) lbl_expr_list;
      list_length lbl_expr_list
  | Zlet(flag, pat_expr_list, body) ->
      do_list (fun (pat,expr) -> check_letrec_expr expr) pat_expr_list;
      size_of_expr body
  | Zstream _ ->
      2
  | Zparser _ ->
      2
```

```
  | _ ->
      illegal_letrec_expr loc
;;

(* ------------------------------------------------------------------------ *)
(* These two handlers are used when a matching completes without finding
   a match. The first one is for regular matchings, it raises a Match_failure;
   the second one is for try..with matchings, it just propagates the exception.
*)

let partial_fun (Loc(start,stop) as loc) tsb =
  let handler =
    Lprim(Praise,
          [Lprim(Pexceptionblock,
            [Lprim(Pget_global "Match_failure", []);
             Lconst(SCatom(ACstring !input_name));
             Lconst(SCatom(ACint start));
             Lconst(SCatom(ACint stop))])]) in
  match tsb with
    True ->
      prerr_location loc;
      prerr_begline " Warning: pattern matching is not exhaustive";
      prerr_endline2 "";
      handler
  | _ ->
      handler
;;

let partial_try (tsb : tristate_logic) =
  Lprim(Praise, [Lvar 0])
;;

(* ------------------------------------------------------------------------ *)
(* The main expression translation routine *)

let rec fun_of_prim arity prim =                          (* Makes a function out of a primitive *)
  let rec make_fct args n =
    if n >= arity
    then Lprim(prim, args)
    else Lfunction(make_fct (Lvar n :: args) (succ n))
  in
    make_fct [] 0
;;

let rec translate_expr env =
  let rec transl (Expr(desc, loc)) =
  match desc with
    Zlongident(path, ValueNotPrim) ->                     (* Regular variable, global or local *)
      translate_access env path
  | Zlongident(path, ValuePrim(0, _)) ->                  (* Primitive, with null arity *)
      translate_access env path                           (* Handled in the same way as a regular variable *)
  | Zlongident(path, ValuePrim(arity, p)) ->              (* Primitive. Put a Lfunction wrapper around it *)
      fun_of_prim arity p                                 (* It will be beta-reduced later if possible *)
```

```
  | Zconstant cst ->
      Lconst cst
  | Ztuple(args) ->                                        (* Tuple *)
      let tr_args =                                        (* If all arguments are constant, optimize by *)
        map transl args in                                 (* creating directly a structured constant *)
      begin try
        Lconst(SCblock(0, map extract_constant tr_args))
      with Not_constant ->
        Lprim(Pmakeblock 0, tr_args)
      end
  | Zconstruct0(path, desc) ->
      begin match desc.cs_tag with
        Tag_regular(tag, span) ->                          (* For regular constructors, the tag is known *)
          Lconst(SCblock(tag, []))                         (* at compile-time *)
      | Tag_exception ->                                   (* For exceptions, we must go fetch it from a *)
          Lprim(Pexceptionblock, [translate_access env path])   (* variable *)
      end
  | Zconstruct1(path, desc, arg) ->                        (* Constructor with argument *)
      begin match desc.cs_tag with
        Tag_regular(tag, span) ->                          (* Regular constructor *)
          begin match desc.cs_kind with
            Constr_regular ->                              (* Regular representation *)
              let tr_arg = transl arg in                   (* Create a one-slot block with a pointer to the *)
              begin match desc.cs_mut with                 (* argument in it *)
                Mutable ->
                  Lprim(Pmakeblock tag, [tr_arg])
              | Notmutable ->                              (* If the constructor is non-mutable, we can try *)
                  begin try                                (* to optimize constants *)
                    Lconst(SCblock(tag, [extract_constant tr_arg]))
                  with Not_constant ->
                    Lprim(Pmakeblock tag, [tr_arg])
                  end
              end
          | Constr_superfluous n ->     begin               (* "Superfluous" (optimized out) constructor *)
              match arg with                               (* Create a n-slot block and store the tuple's *)
                Expr(Ztuple argl, _) ->                    (* elements directly in it *)
                  let tr_argl = map transl argl in
                  begin try                                (* superfluous ==> not mutable *)
                    Lconst(SCblock(tag, map extract_constant tr_argl))
                  with Not_constant ->
                    Lprim(Pmakeblock tag, tr_argl)
                  end
              | _ ->                                       (* If the argument isn't a tuple, we must generate *)
                  let rec extract_fields i =               (* code to extract its fields. Stupid, eh? *)
                    if i >= n then [] else
                      Lprim(Pfield i, [Lvar 0]) :: extract_fields (succ i) in
                  Llet([transl arg],
                       Lprim(Pmakeblock tag, extract_fields 0))
            end
          | Constr_constant ->
              fatal_error "Constant constructor with arguments!"(* Can't happen *)
          end
      | Tag_exception ->                                   (* Exception *)
```

```
        let tag = translate_access env path in                (* Same as above, except that the tag is unknown, *)
        begin match desc.cs_kind with                         (* and we can't create structured constants *)
          Constr_regular ->
            Lprim(Pexceptionblock, [tag; transl arg])
        | Constr_superfluous n -> begin
            match arg with
              Expr(Ztuple argl, _) ->
                Lprim(Pexceptionblock, tag :: map transl argl)
            | _ ->
                let rec extract_fields i =
                  if i >= n then [] else
                    Lprim(Pfield i, [Lvar 0]) :: extract_fields (succ i) in
                Llet([transl arg],
                     Lprim(Pexceptionblock, tag :: extract_fields 0))
          end
        | Constr_constant ->
            fatal_error "Constant exception with arguments!"  (* Can't happen *)
        end
    end
| Zapply(Expr(Zfunction ((patl,_)::_ as case_list), _) as funct, args) ->
    if list_length patl == list_length args then              (* Optimize out trivial redexes *)
      Llet(translate_let env args,
           translate_match loc env (partial_fun loc) case_list)
    else
    Lapply(transl funct, map transl args)

| Zapply((Expr(Zlongident(_, ValuePrim(arity, p)), _) as fct), args) ->
    if arity == list_length args                              (* Inline primitive function applications *)
    then Lprim(p, map transl args)
    else Lapply(transl fct, map transl args)

| Zapply(funct, args) ->                                      (* General case for function applications *)
    Lapply(transl funct, map transl args)

| Zlet(false, pat_expr_list, body) ->
    let cas = map (fun (pat, _) -> pat) pat_expr_list in
      Llet(translate_bind env pat_expr_list,
           translate_match loc env (partial_fun loc) [cas, body])
| Zlet(true, pat_expr_list, body) ->
    let new_env =
      add_let_rec_to_env env pat_expr_list in
    let translate_rec_bind = function
        (Pat(Zvarpat v,_), expr) ->
          translate_expr new_env expr, size_of_expr expr
      | _ ->
          fatal_error "translate_rec_bind" in
    Lletrec(map translate_rec_bind pat_expr_list,
            translate_expr new_env body)
| Zfunction [] ->
    fatal_error "translate_expr: empty fun"
| Zfunction((patl1,act1)::_ as case_list) ->
    let rec transl_fun = function
         []  -> translate_match loc env (partial_fun loc) case_list
```

```
      | a::L -> Lfunction(transl_fun L) in
    transl_fun patl1
| Ztrywith(body, pat_expr_list) ->
    Lhandle(transl body,
            translate_simple_match loc env partial_try pat_expr_list)
| Zsequence(E1, E2) ->
    Lsequence(transl E1, transl E2)
| Zcondition(Eif, Ethen, Eelse) ->
    Lifthenelse(transl Eif, transl Ethen, transl Eelse)
| Zwhile(Econd, Ebody) ->
    Lwhile(transl Econd, transl Ebody)
| Zfor(id, Estart, Estop, up_flag, Ebody) ->
    Lfor(transl Estart,
         translate_expr (Treserved env) Estop,
         up_flag,
         translate_expr (add_for_parameter_to_env env id) Ebody)
| Zsequand(E1, E2) ->
    Lsequand(transl E1, transl E2)
| Zsequor(E1, E2) ->
    Lsequor(transl E1, transl E2)
| Zconstraint(E, _) ->
    transl E
| Zvector [] ->
    Lconst(SCblock(0, []))
| Zvector args ->
    Lprim(Pmakeblock(0), map transl args)
| Zassign(id, E) ->
    translate_update id env (transl E)
| Zrecord lbl_expr_list ->                                  (* Record construction *)
    let v = make_vect (list_length lbl_expr_list) Lstaticfail in
      do_list
        (fun (lbl, e) -> v.(lbl.lbl_pos) <- transl e)       (* Compile every field *)
        lbl_expr_list;
      begin try                                             (* If no field is mutable, we can try to optimize *)
        if for_all                                          (* constant records by replacing them with a *)
             (fun (lbl, e) -> lbl.lbl_mut == Notmutable)    (* structured constant *)
             lbl_expr_list
        then Lconst(SCblock(0, map_vect_list extract_constant v))
        else raise Not_constant
      with Not_constant ->
        Lprim(Pmakeblock(0), list_of_vect v)
      end
| Zrecord_access (e, lbl) ->
    Lprim(Pfield lbl.lbl_pos, [transl e])
| Zrecord_update (e1, lbl, e2) ->
    Lprim(Psetfield lbl.lbl_pos, [transl e1; transl e2])
| Zstream stream_comp_list ->
    translate_stream translate_expr env stream_comp_list
| Zparser case_list ->
    translate_parser translate_expr loc env case_list
| _ ->
    fatal_error "A ZP construct made it into transl_expr!"
in transl
```

```
and translate_match loc env failure_code casel =
  let transl_action (patlist, expr) =
    let (new_env, add_lets) = add_pat_list_to_env env patlist in
      (patlist, add_lets (translate_expr new_env expr)) in
  translate_matching failure_code loc env (map transl_action casel)

and translate_simple_match loc env failure_code pat_expr_list =
  let transl_action (pat, expr) =
    let (new_env, add_lets) = add_pat_to_env env pat in
      ([pat], add_lets (translate_expr new_env expr)) in
  translate_matching failure_code loc env (map transl_action pat_expr_list)

and translate_let env = function
     [] ->  []
  | a::L -> translate_expr env a :: translate_let (Treserved env) L

and translate_bind env = function
    [] -> []
  | (pat, expr) :: rest ->
      translate_expr env expr :: translate_bind (Treserved env) rest
;;

(* ------------------------------------------------------------------------ *)
(* Translation of type restriction on module language expressions.

   restrict_module_expr takes the compiled code for a module expression and
   a restriction_info parameter, and produces code for the restricted module
   expression.
   Restricting the type of a module requires some code, because field numbers
   change. This problem wouldn't occur if record access was based on names instead
   of indexes, but then record access would be slower. Since restricting the type
   of a module is seldom used, it's more efficient this way.

   Restricting the types of functors is somewhat complex. Once we have the arguments,
   we must restrict them; then we must execute the original functor code.
   And finally, we must restrict the result. When executing the code, we must be
   careful to run it inside the environment it expects (i.e. adding Llets around
   it would be invalid because it would make the De Bruijn indices in it invalid).
*)

let rec restrict_module_expr code = function
    RDoNothing ->                                          (* Simplest case: nothing to do *)
      code
  | RMap indexlist ->                                      (* Structure remapping *)
      Llet([code],                                         (* Push the original struct onto the environment *)
           Lprim(Pmakeblock 0,                             (* And create a new structure *)
           map (fun (index, info) ->                       (* Into which we stuff the appropriate fields *)
                  Lprim(Pfield index, [restrict_module_expr (Lvar 0) info]))
                indexlist))
  | RFunctor (arg_info_list, res_info) ->                  (* Functor *)
      let num_args = list_length arg_info_list in
```

```
      let rec restrict_args i = function                        (* Generate code to restrict each argument *)
        [] ->
          []
      | arg_info :: rest ->
          (restrict_module_expr (Lvar i) arg_info) :: (restrict_args (i-1) rest)

      in let restricted_args =
        restrict_args (num_args-1) arg_info_list in
      let restricted_body =                                     (* Generate code to restrict the result *)
        restrict_module_expr (Lapply(Lvar num_args, restricted_args)) res_info in

      let rec wrap body = function                              (* Wrap the body with the right number *)
        0 ->                                                    (* of Lfunction *)
          body
      | i ->
          Lfunction(wrap body (i-1))

      in Llet([code], wrap restricted_body num_args)
;;

(* ------------------------------------------------------------------------ *)
(* Translation of structures.

   translate_structure works by creating an Llet/Lletrec clause for each structure
   element, and then grouping all elements into a block.
   It builds a list of all variable names which is used at the end to look up the
   names in the environment and build the structure. This works because rebindings
   are forbidden (two variables can't have the same name in the same structure).

   Primitives are usually inlined, so they shouldn't need a slot in the structure,
   but:
     - primitives of arity 0 need a slot
     - it is possible to "forget" that a function is a primitive, so the calling
       module will issue a regular function call, and a slot is needed.

   In order to share as much code as possible with the toplevel, a hack has been
   devised. In the toplevel, structure items should be compiled the same way, only
   they should be stored into globals instead of being stored into a structure.
   This means that the transl function can be used for both, only it must behave
   differently when it reaches the end of the structure. This behavior is
   parameterized by the end_of_struct function.
*)

let name_of_constr_decl = function                              (* Small utilities *)
    Zconstr0decl name ->         name
  | Zconstr1decl (name, _, _) -> name
;;

let add name env =
  Tenv([name, Path_root], env)
;;

let rec translate_struct end_of_struct env =
```

```
let rec transl names env = function
  StructItem(desc, loc) :: rest -> begin
    match desc with

      Zexpr(expr) ->                                        (* Lone expression. Code it, but discard its result *)
        Lsequence(translate_expr env expr, transl names env rest)

    | Zletdef (false, pat_expr_list) -> begin               (* Regular let definition *)
        match pat_expr_list with

          [Pat(Zvarpat x, _), expr] ->                      (* Handle the most common case (let x = ...) first *)
            Llet([translate_expr env expr],
                 transl (names @ [x]) (add x env) rest)

        | _ ->                                              (* General case *)
            let pat_list = map fst pat_expr_list in
            let vars_of_pat pat = rev (vars_of_pat pat) in  (* Get variables in the same order as the typchecker! *)
            let vars = flat_map vars_of_pat pat_list in     (* Get the list of newly defined variables *)
            let code = translate_bind env pat_expr_list in  (* Compile the right-hand expressions *)
            let env = it_list                               (* Create an env which explains how to extract *)
                (fun env pat -> Tenv(paths_of_pat Path_root pat, env))
              env pat_list in                               (* their values from the right-hand expressions *)
            let names = names @ vars in                     (* Enter these vars into the field list *)
            Llet(code,                                      (* This means: 1. evaluate the expressions *)
                 translate_matching (partial_fun loc) loc env (* 2. match them against the given patterns *)
                   [pat_list, transl names env rest])       (* 3. put them into the environment and go on *)
      end

    | Zletdef (true, pat_expr_list) ->                      (* Recursive let definition *)
                                                            (* Simpler because there is no pattern matching *)
        let env = add_let_rec_to_env env pat_expr_list in   (* Immediately add the variables to the environment *)
        let rec translate_rec_bind = function               (* Compile the matchings and build the list of names *)
          (Pat(Zvarpat v, _), expr) :: rest ->
            let codel, namesl = translate_rec_bind rest in
            (translate_expr env expr, size_of_expr expr) :: codel, v :: namesl
        | _ ->
            [], [] in
        let codel, namesl = translate_rec_bind pat_expr_list in
        Lletrec(codel, transl (names @ namesl) env rest)

    | (Ztypedef _ | Zmodtype _) ->                          (* Type declarations - no code *)
        transl names env rest

    | Zexcdef exclist ->                                    (* Exception definition *)
        let rec make_tags names env = function
          one :: rest ->
            let name = name_of_constr_decl one in           (* Generate code to create a tag at runtime *)
            let new_tag = Lprim(Pccall("new_exception_tag", 1),
                                [Lconst(SCatom(ACstring name))]) in
            Llet([new_tag], make_tags (names @ [name]) (add name env) rest)
        | [] ->
            transl names env rest
```

```
          in make_tags names env exclist

      | Zmodule (name, expr) ->                                  (* Sub-module *)
          Llet([translate_module_expr env expr],
               transl (names @ [name]) (add name env) rest)

      | Zprimitive deflist ->                                   (* Primitives *)
          let rec loop names env = function
            (name, _, ValuePrim (arity, prim)) :: rest ->
              Llet([fun_of_prim arity prim],
                   loop (names @ [name]) (add name env) rest)
          | _ ->
            transl names env rest

          in loop names env deflist

      | Zimpldirective dir ->                                   (* Implementation directives *)
          transl names (translate_directive env dir) rest
    end

  | [] ->
      end_of_struct env names

  in transl [] env

and translate_directive env = function
  Zdir("open", modname) ->
    add_open_to_comp_env env modname
| Zdir("import", modname) ->
    add_import_to_comp_env env modname
| _ ->
    env

and create_structure env names =                                (* Once we reach the end of the structure *)
  Lprim(Pmakeblock(0),                                          (* Build a block *)
        map (translate_local_access env) names)                 (* And fill its fields from the environment *)

and translate_module_expr env (MExpr(desc, loc)) = match desc with
    ZMlongident path ->
      translate_access env path
  | ZMstructure structure ->
      translate_struct create_structure env structure
  | ZMfunctor (paraml, body) ->                                 (* Functor *)
      let rec build env = function                              (* Compiled exactly the same way as a function *)
        (name, _) :: rest ->
          Lfunction(build (add name env) rest)
      | [] ->
          translate_module_expr env body
      in build env paraml
  | ZMapply (fctor, arg_restr_list) ->                          (* Functor application *)
      let transl_arg = function                                 (* Each argument might have to go through a type *)
        (arg, ref (Some restr_info)) ->                         (* restriction, as recorded by type_module_expr *)
          restrict_module_expr (translate_module_expr env arg) restr_info
```

```
      | _ ->
          fatal_error "Missing restriction info in translate_module_expr ZMapply!" in

      Lapply(translate_module_expr env fctor,
             map transl_arg arg_restr_list)
  | ZMconstraint (body, _, ref (Some restr_info)) ->            (* Type restriction *)
      restrict_module_expr (translate_module_expr env body) restr_info
  | ZMconstraint _ ->
      fatal_error "Missing restriction info in translate_module_expr ZMconstraint!"
;;

let translate_structure =
  translate_struct create_structure
;;
```

Unité de recherche Inria Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 Villers Lès Nancy
Unité de recherche Inria Rennes, Irisa, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Unité de recherche Inria Rhône-Alpes, 46 avenue Félix Viallet, 38031 Grenoble Cedex 1
Unité de recherche Inria Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 Le Chesnay Cedex
Unité de recherche Inria Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex

Éditeur
Inria, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
ISSN 0249-6399