Programmation paramétrée en ML et automates d'architecture

Ph. Narbel

LaBRI, Bordeaux I. 351, Cours de la Libération, 33405 Talence, France. narbel@labri.fr

Résumé. *Nous introduisons tout d'abord une représentation des programmes* ML *qui met en exergue les possibilités intrinsèques de* programmation paramétrée *de ce langage. Il s'agit d'*automates d'architecture *qui intègrent spécification, implémentation partielle et évolution de cette implémentation. Nous explorons ensuite les potentialités de ces automates. Tout d'abord, le système de type* ML *permet directement d'en valider leur construction, leur modification et leur fonctionnement. Ensuite, la prévision de leur évolution est résolue dans les cas simples par l'analyse de grammaires non-contextuelles classiques. Celles-ci livrent les langages des expressions de foncteurs dont le résultat satisfait à une signature donnée. Néanmoins, dans le cas où un automate d'architecture contient des cycles et des contraintes de types partagés (*«sharing types»*), nous montrons qu'il n'est alors que semi-décidable de savoir si un tel langage est vide ou non.*

1. Introduction

La «*programmation paramétrée*» (*parameterized programming*) rassemble les techniques de programmation et de conception d'architectures qui se basent sur la composition d'éléments spécifiés et paramétrés. C'est Goguen [Gog84, Gog96] qui a le premier utilisé cette appellation et qui a développé cette notion, essentiellement pour des extensions de *Ada* et pour le langage *OBJ*. De nos jours, la programmation paramétrée est surtout mise à l'honneur en *C++* par un intérêt croissant pour les «patrons» (*templates*) (cf. par ex. [VN03]) et par l'utilisation de sa bibliothèque STL (*Standard Template Library*). Les langages *Java* et *C#* préparent également des extensions afin d'intégrer ce style de programmation. D'autre part, des développements récents ont donné lieu à des généralisations, comme par ex. la «*programmation générative*» [CE00] qui réunit tous les moyens incluant une part de construction automatique de code. Il s'agit ici du vieux rêve du *génie logiciel* (cf. par ex.[Kru92]): faire en sorte que l'élaboration et la réutilisation de programmes y soient aussi efficaces et faciles qu'en électronique ou en génie civil.

L'idée sous-jacente à cet article est de montrer que le système de modules de *ML* permet *intrinsèquement* une programmation paramétrée puissante. Ce fait est dû à l'une des caractéristiques principales de ce langage: *son système de type intègre pleinement les modules et leur généricité*. La programmation paramétrée peut ainsi s'établir en *ML* dans le cadre d'un typage statique, inféré, sûr, où chaque composant est instance d'un type déclaré, et ceci avec les moyens propres du langage.

La première partie de ce texte s'attache à mettre en place une méthode de construction de programmes *ML* mettant en exergue les possibilités de programmation paramétrée de ce langage. Cet objectif passe par l'élaboration d'une représentation pour les architectures *ML*. De fait, il en existe peu dans la littérature. Paulson dans son livre [Pau97] a introduit une représentation qui permet d'indiquer la filiation des modules et des foncteurs d'un programme particulier; les signatures n'y prennent pas place. Plus récemment, Cousineau [Cou02] a utilisé une représentation plus complète qui inclut les signatures de modules, mais non pas les signatures de foncteurs. La représentation/notation que nous proposons ici en est une extension. Elle met tout d'abord en évidence le fait qu'une architecture *ML* peut être entièrement spécifiée à l'aide de signatures de modules et de foncteurs, traitées à part égale.

Cette propriété se justifie clairement dans un cadre où les foncteurs peuvent être d'ordre supérieur. Les signatures forment alors un graphe de spécification où chacune correspond à un sommet distinct auquel il est possible d'associer des implémentations qui y satisfont. L'application des foncteurs fait alors évoluer l'ensemble des implémentations. Ainsi, au lieu de rester à l'état de simple graphe, la représentation d'une architecture ML apparaît plutôt comme un *automate* (ayant d'ailleurs quelques traits de ressemblance avec les réseaux dits de *Petri*). Cette représentation met donc en relief trois phases essentielles de la constitution d'un programme *ML* : la *spécification*, l'*implémentation partielle*, et la *construction de programmes particuliers par applications de foncteurs*. Chacune de ces phases peut profiter de vérifications par le système de type de ML, ce qui permet de réaliser des outils de développement capables de valider automatiquement nombre d'actions. Un prototype qui intègre une interface graphique a été réalisé dans ce sens.

Nous nous attachons ensuite au problème de décrire plus précisément ce que recouvre la troisième phase évoquée ci-dessus, à savoir la description des programmes qu'un automate d'architecture est capable d'engendrer. Plus précisément, il s'agit de pouvoir répondre à certaines des questions de base de la «programmation générative»: est-il possible de faire évoluer un automate d'architecture de manière à obtenir une implémentation qui satisfasse à une spécification (signature) particulière ? Et si oui, comment le réaliser ?

Le premier pas que nous effectuons dans ce sens est de montrer que les automates d'architecture se prêtent facilement à une traduction en *grammaires non-contextuelles* classiques. Ce genre de traduction apparaît aussi dans d'autres modèles généraux de réutilisation de code [Kru92, CE00], en particulier dans le modèle dit «*GenVoca*» [BS92] de programmation paramétrée hiérarchisée. Les grammaires que nous utiliserons ici en sont une généralisation qui permet de strictement distinguer entre spécification (signatures) et implémentation, et également entre composants paramétrés (foncteurs) et non-paramétrés (modules). Ces distinctions rendent possibles un lien direct avec le système de type d'*OCaml*. En particulier, si `S` est un sommet d'un automate d'architecture $\mathcal{A}$, sa grammaire associée représentera le langage des *expressions d'applications de foncteurs* associé à $\mathcal{A}$ dont le résultat est de signature `S`. Dans le cas où il n'y a pas de contraintes de types sous forme de «*types partagés*» (*sharing types*) ajoutées aux paramètres des foncteurs n-aires, ces langages d'expressions de foncteurs sont tous constitués d'expressions bien typées. Il est alors aisé d'obtenir des réponses aux questions posées au paragraphe précédent. Dans le cas où des types partagés interviennent, ce n'est plus le cas: la grammaire non-contextuelle ne devient qu'un moyen d'énumération des expressions de foncteurs qui sont candidates pour instancier `S`. Nous montrons alors qu'il n'est que semi-décidable de savoir si le sous-langage des expressions bien typées est vide ou non. Cette preuve s'effectue en construisant des automates d'architecture qui émulent le «*problème de correspondance de Post*» [DW85]. Néanmoins dans la pratique, ce résultat n'a pas tant d'influence: les expressions de foncteurs sont énumérables au travers d'une grammaire et elles n'ont pas en général à devenir arbitrairement longues. Incidemment, ce résultat prouve également que l'on ne peut pas dans tous les cas se débarrasser des contraintes de types partagés par des modifications constructives d'un automate d'architecture [Har01].

Nous utiliserons ici le langage *OCaml 3.07* [LDG$^+$03]. Celui-ci inclut entre autres les foncteurs d'ordre supérieur (tout comme *Moscow ML*). Les *motifs* ou *patrons de construction* (*design patterns*) de la programmation modulaire ML n'en sont que plus riches (quelques illustrations en seront données dans ce texte). En revanche, les modules récursifs et l'interaction avec la couche objet d'*OCaml* ne seront pas traités ici.

2. Propriétés pratiques du système de module *ML*

Les propriétés pratiques principales du système de module de *ML* sont les suivantes:

- *Un moyen naturel de spécification.* Tout ensemble de modules/foncteurs peut être entièrement spécifié au moyen de types (les *signatures*).
- *Une multiplicité des implémentations.* Une signature peut être implémentée de manière multiple. Ainsi, une signature est plus qu'une *interface*: elle est un type à instancier.
- *Une potentialité.* Un ensemble de modules/foncteurs dénote un ensemble de programmes possibles et non pas un programme particulier. C'est l'ensemble des applications de foncteurs qui le détermine.
- *Des dépendances au temps.* Les composants d'un programme peuvent soit être liés entre eux par le temps, soit apparaître de manière asynchrone. Ils peuvent être dépendants des applications de foncteurs.
- *Des liens possibles entre implémentation et spécification.* Les foncteurs utilisent des paramètres typés par des signatures. Ils utilisent donc une partie de la spécification qui leur préexiste, ce qui définit leurs liens à l'ensemble tout en leur donnant une indépendance quant à leurs implémentations.

Toutes ces propriétés s'observent sur des programmes très élémentaires:

Architecture 1:

```
(* specification*)
module type S1 = sig ... end
module type S2 = sig ... end
module type SF = functor (M: S1) -> S2;;

(* implementation *)
module M1: S1 = struct ... end
module M2: S1 = struct ... end
module F: SF = functor (M: S1) -> struct ... end

(* evolution de l'implementation *)
module M3: S2 = F(M1);;
```

Ici, les signatures `S1`, `S2` permettent de spécifier une «signature» de foncteur `SF`; les implémentations `F`, `M1` et `M2` satisfont alors respectivement à ces signatures; à ce stade, le programme peut donc évoluer de deux manières: le foncteur `F` peut s'appliquer soit sur `M1`, soit sur `M2`, produisant une instance de signature `S2`.

Cet exemple à lui seul justifie d'appeler **architecture modulaire** ML, non seulement l'organisation d'un ensemble de signatures, de modules et de foncteurs, mais également la manière dont cet ensemble peut évoluer par applications de foncteurs (une idée déjà présente dans [Gog96]). Plus précisément, une architecture ML recouvre les trois phases principales de développement logiciel:

1. La spécification, i.e. la définition de signatures et les relations qu'elles entretiennent entre elles;
2. L'implémentation partielle, i.e. l'instanciation d'un sous-ensemble de ces signatures;
3. La construction par applications des foncteurs de l'implémentation de certains modules spécifiés, c'est-à-dire de certains programmes particuliers.

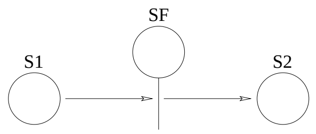
Soulignons également que dans le cadre de la programmation paramétrée, l'idée est de pouvoir inclure le plus d'informations possible dans la partie spécification, i.e. dans l'ensemble des signatures. Toutes les importations et les inclusions de code devront être rendues explicites par abstraction fonctorielle. C'est d'ailleurs une des qualités des foncteurs que d'accroître non seulement la généricité du code mais également le caractère explicite de la réutilisation de code. La programmation paramétrée induit donc une «programmation fonctorielle intensive» (*fully-functorial programming* [Pau97]).

3. Les automates d'architecture

La part de spécification d'une architecture sera représentée par un graphe orienté (V, E) appelé **graphe de spécification**. Chaque sommet V dénote une signature distincte de l'architecture. Un **sommet simple** dénote une signature de module et un **sommet transition** dénote une signature de foncteur. Graphiquement, nous représenterons les sommets simples par des disques et les sommets transition par des disques desquels partent un brin qui représente l'«applicabilité» du sommet (voir la figure qui suit). Ce brin peut être affublé d'une étiquette qui indique les contraintes de types qu'il recouvre. Les **arêtes** E sont déterminées par plusieurs relations différentes:

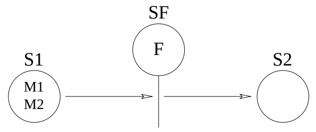
- *(Paramètres de foncteurs).* Une arête existe entre un sommet S et un sommet transition SF, si S est signature d'un des paramètres de SF (cette arête peut être multiple s'il y a plusieurs paramètres de SF qui sont de signature S). Graphiquement, cette arête part du sommet S et pointe sur le brin du sommet transition SF. Si SF a plus d'un paramètre, les arcs pointent tous sur le brin dans le même ordre. Notons que S peut être également une signature de foncteur dans le cas où SF indique un foncteur d'ordre supérieur.
- *(Résultats de foncteurs).* Une arête existe entre un sommet transition SF et un sommet S, si S est signature d'un des résultats possibles de SF (les applications partielles des fonctions n-aires sont possibles). Graphiquement, cette arête part du brin du sommet transition SF et va sur le sommet S. À nouveau S peut être une signature de foncteur.
- *(Arêtes de compatibilité).* Soit un module M de signature S. Alors M peut être explicitement typé par S', si S' est égale ou plus petite que S, si les types des éléments de S' sont égaux ou moins généraux que ceux de S, et si des types concrets de S sont abstraits dans S'. Dans ce cas, M est instance à la fois de S et de S'. On dira alors que S est **compatible** avec S', et une arête existe de S vers S'. Dans le cas des signatures de foncteurs, alors $S1 \to S2$ est **compatible** avec $S1' \to S2'$ si $S1'$ est compatible avec $S1$, et si $S2$ est compatible avec $S2'$ (attention à l'inversion des sens).

Ainsi, le graphe de spécification de l'Architecture 1 est de la forme suivante:

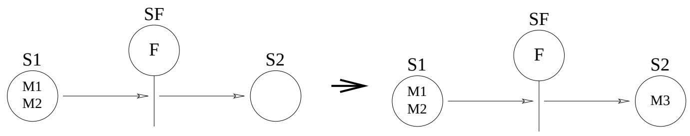


Les implémentations peuvent alors prendre place dans le graphe de spécification (V, E): un module, un foncteur ou une expression d'applications de foncteurs peut être associé à un sommet $S \in V$ si celui-ci est instance de S. Notons qu'*un sommet peut contenir plusieurs implémentations* puisqu'une signature peut être implémentée de manière multiple.

Associer un ensemble I d'éléments implémentés à un graphe de spécification le transforme en **graphe d'architecture** (V, E, I). Par rapport à l'Architecture 1, la partie implémentation complétait le graphe de spécification ainsi:



Ensuite, l'application de foncteurs peut faire évoluer l'état d'un graphe d'architecture (V, E, I). En effet, un sommet transition SF est dit **applicable** s'il contient au moins une implémentation et si tous ses arcs entrants sur son brin sont reliés à des sommets contenant au moins une implémentation[1]. Soulignons que ceci peut n'être qu'une condition nécessaire puisque des contraintes de types peuvent intervenir, en particulier dans le cas des foncteurs n-aires, $n > 1$. Un sommet transition SF peut **être appliqué** en faisant le choix d'un foncteur contenu par SF et à l'appliquer à un choix d'implémentations de ses paramètres. On ajoute alors le résultat dans le sommet relié à l'arc sortant de SF. Ce résultat est associé soit à l'expression fonctorielle qui correspond à l'application, soit à un nom d'implémentation. Le graphe d'architecture devient alors un **automate d'architecture** (V, E, I, τ) où τ correspond à la règle d'évolution. Par rapport à l'Architecture 1, la construction de `M3` faisait évoluer l'automate d'architecture ainsi:



Toute expression de foncteurs peut être également associée à un **graphe d'expression** où chaque sommet représente une implémentation particulière et où une arête entre deux sommets M_1 et M_2 dénote soit l'utilisation de M_1 comme paramètre effectif de M_2, soit le fait que M_2 soit le résultat de l'application de M_1 sur ses paramètres. Pour l'exemple précédent, cela donne trivialement:

$$\mathrm{M1} \longrightarrow \mathrm{F} \longrightarrow \mathrm{F(M1)}$$

Les graphes d'expression ne sont pas toujours structurellement isomorphes à un sous-graphe du graphe d'architecture d'où ils sont issus: ils peuvent n'être que des *revêtements*. Cela peut être le cas lorsque le graphe d'architecture possède des cycles. Les graphes d'expressions correspondent à la représentation de Paulson [Pau97], et n'incarnent donc qu'un seul programme particulier à la fois. Ils correspondent également à ce que Goguen [Gog96] appelle un *système d'architecture* (*system architecture*).

4. Quelques exemples d'automates d'architecture

Voici des exemples qui mettent en exergue quelques particularités techniques du système de type de *ML* et qui peuvent se représenter au moyen d'automates d'architecture.

Arêtes de compatibilité. Elles surgissent par exemple lorsque des *inclusions de signatures* sont utilisées:

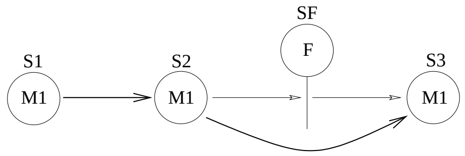
Architecture 2:

```
module type S3 = sig ... end
module type S2 = sig include S3 ... end
module type S1 = sig include S2 ... end
module type SF = functor (M: S2) -> S3;;

module M1: S1 = struct ... end
module F : SF = functor (M: S2) -> struct ... end
```

Ici, `S1` est compatible avec `S2` qui est compatible avec `S3`. La compatibilité permet donc ici à `M1` d'apparaître comme un argument valide pour `F`. Voici l'automate correspondant (l'ensemble des arêtes

1. Notons la parenté ici avec les *réseaux de Petri*, même si une différence importante existe: les implémentations ne sont pas «consommées» comme le sont les «jetons» de la version originale de ces réseaux.

de compatibilité est obtenu par transitivité):



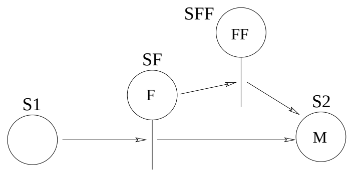
Une implémentation peut donc apparaître plusieurs fois dans un automate d'architecture: si un sommet S contient une implémentation `M`, alors `M` apparaîtra dans tous les sommets du graphe qui sont accessibles de S par des arêtes de compatibilité.

Foncteurs d'ordre supérieur. En *OCaml*, les arguments et le résultat de l'application d'un foncteur peuvent être également des foncteurs. Ce cas se produit classiquement dans les «*patrons de création*» (*creational patterns*) [GHJV95] en *OCaml*. Par exemple:

Architecture 3:

```
module type S1 = sig ... end
module type S2 = sig ... end
module type SF = functor (M: S1) -> S2;;
module type SFF = functor (F: SF) -> S2;;

module F : SF = functor (M: S1) -> struct ... end

module FF: SFF = functor (F_arg: SF) -> struct
  module M1_private: S1 = struct ... end
  include F_arg(M1_private)
end;;

module M: S2 = FF(F);;
```

Ici, le module `M1_private` est un module privé de `FF` qui assure que le foncteur `F_arg` est toujours appliqué au même argument, par ex. pour spécialiser toujours de la même façon un foncteur trop général. Voici l'automate d'architecture correspondant:



On notera que `M1_private` n'apparaît pas dans l'architecture car il ne fait partie que de l'implémentation de `FF`.

Cycles. La plupart du temps un automate d'architecture ressemble à un arbre ou à un graphe orienté acyclique. Mais les cycles sont également possibles. Le cas le plus simple est le cas d'un foncteur dont le résultat est de même signature que son paramètre:

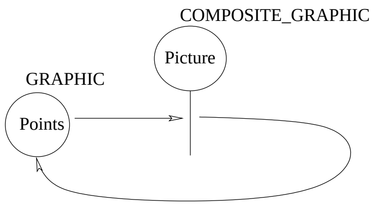
```
module type S = sig ... end
module type SF = functor (M: S) -> S;;
```

Ce cas se produit souvent pour les structures algébriques, par ex. les polynômes à coefficients dans un anneau qui redonnent un anneau; l'ensemble des automorphismes d'un groupe qui redonne un groupe. Un exemple plus appliqué provient de l'équivalent statique du «*patron composite*» [GHJV95] de la programmation objet standard. Par exemple, il est possible de hiérarchiser un ensemble d'objets graphiques grâce à un foncteur cyclique[2]:

Architecture 4:

```
module type GRAPHIC = sig
  type t  val draw: t -> unit   val remove: t -> unit
end

module type COMPOSITE_GRAPHIC = functor (G: GRAPHIC) -> GRAPHIC;;

module Picture (G: GRAPHIC) = struct
  type t = G.t list
  let draw g = List.iter G.draw g
  let remove g = List.iter G.remove g
end;;

module Points = struct
  type t = {x: int; y: int}
  let draw p = plot p.x p.y
  let remove p = set_color white; plot p.x p.y; set_color black
end;;
```

Ce qui détermine l'automate suivant:



Cet automate indique que l'on peut augmenter arbitrairement le nombre d'implémentations qui satisfont à la signature `GRAPHIC`, par ex.

```
module Points_of_Points: GRAPHIC = Picture (Points);;
module Points_of_Points_of_Points : GRAPHIC = Picture (Points_of_Points);;
```

Ici, les graphes d'expressions sont donc de simples chaînes arbitrairement longues.

Contraintes de types. Lors d'un typage explicite avec signatures, il est possible d'adjoindre des contraintes à une signature existante S en utilisant l'opérateur `with`. Cela permet par exemple de rendre *a posteriori* concret un type t qui n'était qu'abstrait dans S. Cela induit une signature S' qui est compatible avec S. Par exemple:

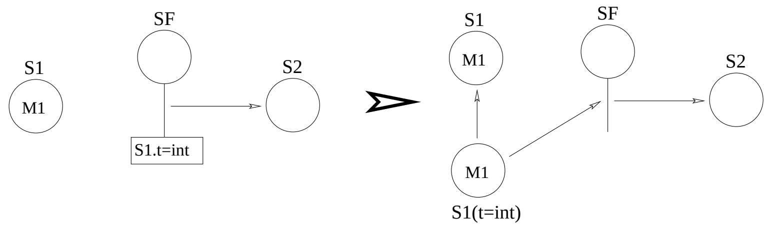
Architecture 5:

```
module type S1 = sig type t end
module type S2 = sig type t end
module type SF = functor (M: S1 with type t=int) ->  S2;;
module M1: S1 with type t=int = struct ...  end
```

On peut facilement intégrer ce cas à un automate d'architecture en appliquant un éclatement du

2. Évidemment en polymorphisme paramétrique, ce motif de programmation est beaucoup moins puissant que dans le cadre d'un polymorphisme d'inclusion: les objets graphiques qui peuvent être traités en même temps doivent tous être strictement du même type.

sommet `S1` afin que `SF` devienne un sommet applicable:

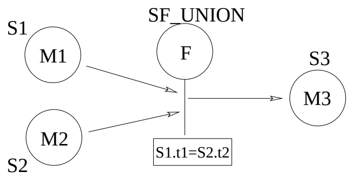


Un autre cas de contraintes de types est celui qui survient pour un foncteur n-aire, $n > 1$ où il s'agit d'indiquer un *partage de types ou de modules* (*sharing types*). Par exemple, lorsqu'on effectue l'union de deux types abstraits (un héritage multiple), on peut spécifier l'égalité des types concrets et empêcher ainsi la construction de types incohérents:

Architecture 6:

```
module type S1 = sig type t1 val f: t1 -> t1 end;;
module type S2 = sig type t2 val g: t2 -> t2 end;;
module type S3 = sig type t  val f: t -> t  val g: t -> t end;;
module type SF_UNION = functor (M1: S1) -> functor (M2: S2 with type t2=M1.t1) -> S3;;

module F: SF_UNION = functor (M1: S1) -> functor (M2: S2 with type t2=M1.t1) ->
    struct  type t = M1.t1 include M1 include M2 end;;

module M1 = struct type t1=int let f x = x+1 end;;
module M2 = struct type t2=int let g x = x+2 end;;
module M3 = F (M1)(M2);;
```

Ici, l'union est réalisée dans des conditions vérifiées par la spécification: `M3` est un type abstrait fondé sur les entiers. Ces contraintes de types ne peuvent pas être évitées par un simple éclatement de sommet comme ci-dessus: *a priori* les types inclus dans les deux paramètres peuvent être quelconques. Voici l'automate d'architecture correspondant:

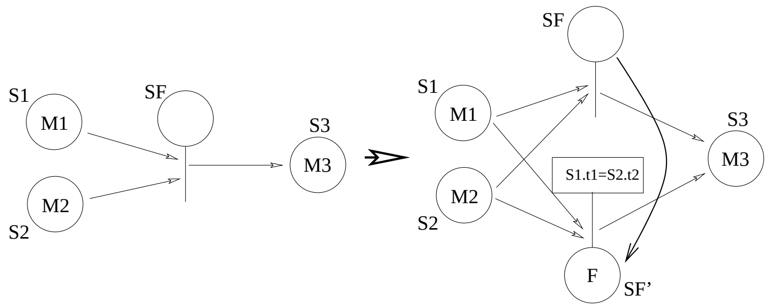


Les contraintes de partage de types sont parfois rendues obligatoires par certaines implémentations (ce n'était pas le cas ci-dessus). Considérons la variation suivante de l'exemple précédent:

Architecture 7:

```
module type S1 = sig type t1 val f: t1 -> t1 end;;
module type S2 = sig type t2 val g: t2 -> t2 end;;
module type S3 = sig type t  val h: t -> t end;;
module type SF = functor (M1: S1) -> functor (M2: S2) -> S3;;
module type SF' = functor (M1: S1) -> functor (M2: S2 with type t2=M1.t1) -> S3;;

module F: SF' = functor (M1: S1) -> functor (M2:S2 with type t2 = M1.t1) ->
    struct type t = M1.t1 let h x = M2.g (M1.f x) end;;
module M1 = struct type t1=int let f x = x+1 end;;
module M2 = struct type t2=int let g x = x+2 end;;
module M3 = F (M1) (M2);;
```

Ici, la contrainte de type dans la signature de `F` est nécessaire pour rendre sa compilation possible. Ce foncteur ne satisfait pas à `SF`, mais seulement à `SF'`. Cela correspond également à un éclatement du sommet `SF`, en créant un lien de compatibilité entre `SF` et `SF'`:

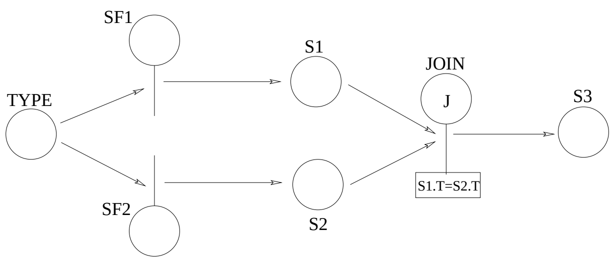


Un cas habituel d'occurrence de contraintes de types partagés se produit lorsqu'il existe deux sommets `S` et `SF` dans l'automate d'architecture qui sont reliés par deux chemins distincts et tel que `SF` soit un sommet transition. L'exemple ci-dessus se généralise en effet ainsi:

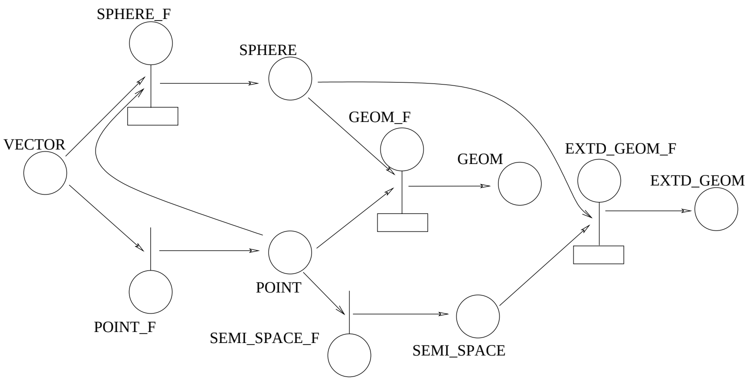
Architecture 8:

```
module type TYPE = sig type t end;;
module type S1 = sig module T: TYPE val f: T.t -> T.t end;;
module type S2 = sig module T: TYPE val g: T.t -> T.t end;;
module type S3 = sig module M1: S1 module M2: S2 val h:end;;
module type SF1 = functor (T: TYPE) -> S1
module type SF2 = functor (T: TYPE) -> S2
module type JOIN = functor (M1: S1) -> functor (M2: S2 with module T=M1.T) -> S3;;
module J: JOIN = functor (M1: S1) -> functor (M2: S2 with module T=M1.T) ->
    struct module M1=M1 module M2=M2 end;;
```

Ce qui donne le graphe d'architecture suivant:



Cet exemple est un cas similaire à celui de Paulson [Pau97] (pp. 292-293) qui combine deux signatures possédant une file de priorité (`TYPE` y est remplacé par `PRIORITY_QUEUE`, `S1` par `IN`, `S2` par `OUT`, et `JOIN` par le type du foncteur `MainFunctor`). D'autre part, les exemples de Harper dans [Har01] (Chap. 22 et 23) mêlent trois de ces motifs afin de mettre en place la base d'une architecture liée à un modèle géométrique. La représentation par automate d'architecture permet d'en saisir rapidement la globalité:

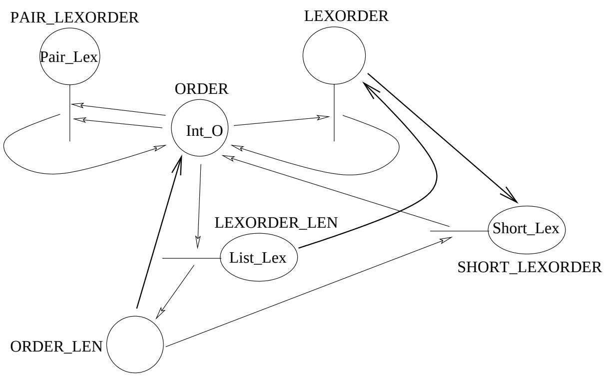


Cycles multiples. Voici un dernier exemple qui montre qu'un automate d'architecture peut impliquer des entrelacs de cycles. Il s'agit de mêler trois constructions différentes de relations d'ordre: un ordre lexicographique `LEXORDER` sur des structures linéaires (cf. par ex. [Gog84]), un ordre lexicographique `PAIR_LEXORDER` sur une paire de types ordonnés (cf. par ex. [Pau97]), et un ordre *shortlex* `SHORT_LEXORDER` sur une structure linéaire dont on peut extraire la longueur. La signature principale des ensembles ordonnés est `ORDER`, et `ORDER_LEN` est celle des ordres qui possèdent également une fonction de longueur:

Architecture 9:

```
module type ORDER =  sig type t val leq: t -> t -> int  end;;
module type ORDER_LEN = sig type t val leq: t -> t -> int val len: t -> int end;;
module type LEXORDER = functor (O: ORDER) -> ORDER;;
module type LEXORDER_LEN = functor (O: ORDER) -> ORDER_LEN;;
module type PAIR_LEXORDER = functor (O1: ORDER) -> functor (O2: ORDER) -> ORDER;;
module type SHORT_LEXORDER = functor (O: ORDER_LEN) -> ORDER;;

module List_Lex: LEXORDER_LEN = functor (O: ORDER) ->
   struct type t = O.t list let leq l1 l2 = ... (* O.leq *) let len = List.length end;;

module Pair_Lex: PAIR_LEXORDER = functor (O1: ORDER) -> functor (O2: ORDER) ->
    struct type t = O1.t * O2.t let leq (x1,y1) (x2,y2) = ... (* O_.leq *) end;;

module Short_Lex: SHORT_LEXORDER = functor (O: ORDER_LEN) -> struct
    type t = O.t let leq l1 l2 = ... (* O.len and O.leq *) end;;

module Int_O = struct type t=int let leq = compare end;;
```

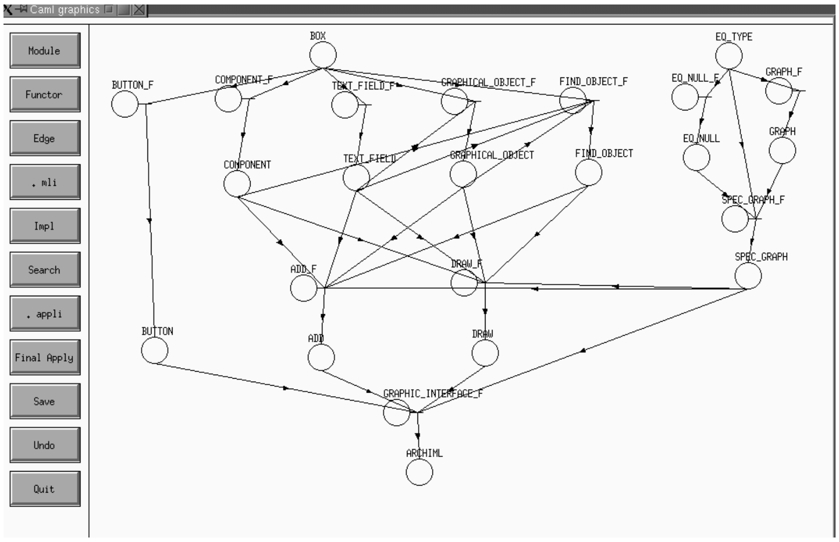
L'automate d'architecture est alors le suivant:



5. Exploitation des automates d'architecture

5.1. Contrôles et validations automatiques

Les automates d'architecture permettent donc de représenter les constructions modulaires *ML*. Mais ces automates ne se résument pas à cela puisqu'ils sont fortement liés au système de type. Ainsi, nombre de leurs propriétés peuvent être directement *vérifiées par les moyens propres du langage*. En *OCaml*, mettre en oeuvre ces vérifications est particulièrement aisé puisque ce langage possède des outils de compilation séparée et de production des types dans des fichiers textes (cf. les options `-i` et `-dtypes` du compilateur `ocamlc`). Les signatures des modules et des foncteurs peuvent donc être inférées, manipulées et mises en relation à l'extérieur de l'environnement interactif. Ainsi, on peut par exemple obtenir automatiquement: *la validation des composants* d'un automate d'architecture; *la cohérence des signatures des foncteurs* au vu de leurs arcs entrants et sortants; *la recherche des sommets* auxquels une implémentation donnée peut être associée; *la recherche des foncteurs applicables*; *l'exploitation des arcs de compatibilité*. Ces quelques points ont été validés avec un prototype incluant une interface graphique minimale. Voici par exemple l'automate d'architecture du prototype dans son état actuel, représenté par lui-même (sans les contraintes de types et sans les implémentations):



5.2. Programmation générative

La question principale liée à l'exploitation d'un automate d'architecture est alors la suivante: soit un sommet (une signature) `S` de cet automate, est-il possible de faire évoluer l'automate de manière à obtenir un composant de signature `S`? Autrement dit, y a-t-il une suite d'applications de foncteurs qui permet d'engendrer un module/foncteur qui satisfait à une signature particulière?

Tout d'abord, il faut noter qu'un automate d'architecture se prête très simplement à une traduction dans un modèle habituel de la théorie des langages. En effet, l'ensemble des expressions de foncteurs associées à un sommet S d'un automate d'architecture $\mathcal{A} = (V, E, I, \tau)$ peut être décrit par une *grammaire non-contextuelle* $G_{\mathcal{A}} = (S, \mathcal{V}, \mathcal{T}, \mathcal{P})$ où l'axiome est S, les non-terminaux $\mathcal{V}$ sont les signatures de V, les terminaux $\mathcal{T}$ sont les composants implémentés de I auxquels on ajoute les symboles de parenthèsage {"(", ")"}, et l'ensemble des règles de production $\mathcal{P}$ est le suivant:

- Pour chaque sommet S, et pour chaque arc qui part d'un sommet S' et qui arrive sur S (i.e. un arc de compatibilité), associer la règle $S \to S'$.

- Pour chaque sommet S, et pour chaque arc qui part d'un brin d'un sommet SF et qui arrive sur S (i.e. un arc résultat qui provient de SF), associer la règle $S \to SF(S_1)(S_2)...(S_n)$, où les S_i sont les signatures des paramètres de SF.
- Pour chaque sommet S, considérer l'ensemble $\{I_1, ...I_n\}$ des implémentations contenues par S (qui ne sont pas issues de recopies par arcs de compatibilité), et associer la règle $S \to I_1|I_2|...|I_n$

On remarquera que ces grammaires font la distinction entre la spécification (l'ensemble des symboles non-terminaux) et l'implémentation (l'ensemble des symboles terminaux). En revanche, elles n'incluent pas directement les informations de contraintes de types. Par exemple, pour l'Architecture 1:

$$S1 \to M1 \mid M2 \qquad SF \to F \qquad S2 \to SF(S1)$$

Pour l'Architecture 9, la grammaire associée est la suivante:

$$\begin{array}{lcl}
ORDER & \to & Int_O \mid LEXORDER(ORDER) \mid \\
 & & \mid PAIR_LEXORDER(ORDER)(ORDER) \mid \\
 & & \mid SHORT_LEXORDER(ORDER_LEN) \mid ORDER_LEN \\
ORDER_LEN & \to & LEXORDER_LEN(ORDER) \\
LEXORDER & \to & LEXORDER_LEN \\
SHORT_LEXORDER & \to & LEXORDER \mid Short_Lex \\
LEXORDER_LEN & \to & List_Lex \\
PAIR_LEXORDER & \to & Pair_Lex
\end{array}$$

On note $L_{G_\mathcal{A}}(S)$ le langage engendré par la grammaire $G_\mathcal{A} = (S, \mathcal{V}, \mathcal{T}, \mathcal{P})$ (ou plus simplement $L(S)$ lorsqu'il n'y a pas d'ambiguïté sur l'automate). Ainsi, pour le premier exemple ci-dessus, $L(S1) = \{M1, M2\}$, $L(SF) = \{F\}$, et $L(S2) = \{F(M1), F(M2)\}$.

Remarque 1 *Soit S un sommet d'un automate d'architecture $\mathcal{A}$ sans contraintes de types, alors $L(S)$ est exactement l'ensemble des expressions de foncteurs qui permettent d'engendrer une instance de la signature S par $\mathcal{A}$.*

Le cas d'un automate d'architecture sans contraintes de types voit ainsi beaucoup de questions résolues:

Remarque 2 *Soit S un sommet d'un automate d'architecture $\mathcal{A}$, alors $L(S) = \emptyset$, $|L(S)| = \infty$, et $x \in L(S)$ sont décidables.*

Néanmoins :

Remarque 3 *Une grammaire issue d'un automate d'architecture $\mathcal{A}$ peut être ambiguë.*

En effet, la même expression de foncteurs peut être engendrée de plusieurs manières différentes lorsqu'un automate d'architecture inclut des arcs de compatibilité. Par exemple:

Architecture 10:

```
module type S1  = sig val x1:int val x2:int end;;
module type S1' = sig val x1:int end;;
module type S   = sig val x:int end;;
module type SF  = functor (M1: S1) -> S;;
module type SF' = functor (M1: S1') -> S;;

module M1 : S1  = struct let x1=1 let x2=2 end;;
module F  : SF' = functor (M1: S1') -> struct let x = M1.x1 end;;
```

Ici, `S1` est compatible avec `S1'`, et `SF'` est compatible avec `SF`. La grammaire est donc la suivante:

$$S1 \to SF(S1) \mid SF'(S1') \qquad SF \to SF' \qquad S1' \to S1 \qquad S1 \to M \qquad SF' \to F$$

Ainsi, `F(M)` peut être engendrée par $S \to SF(S1) \to SF'(S1) \to F(M)$, ou bien $S \to SF'(S1') \to SF'(S1) \to F(M)$. Les algorithmes classiques d'énumération (par ex. [HJ83]) ne peuvent donc pas *a priori* être utilisés sans risques de redondance. Cependant, dans la pratique, la part du langage des expressions de foncteurs qui intéresse le programmeur est en général limitée, et donc cette redondance peut être testée.

Dans le cas où l'automate d'architecture $\mathcal{A}$ possède des contraintes de types, un langage $L(S)$ peut contenir des expressions de foncteurs qui ne sont pas applicables car mises en défaut par ces contraintes. On note alors $L'(S)$ le sous-ensemble des expressions de $L(S)$ qui satisfont à ces contraintes et qui sont donc bien typées. Seules les expressions de $L'(S)$ peuvent engendrer des composants de signatures S par $\mathcal{A}$. Comme $L(S)$ est *énumérable*, il suffit d'appliquer le système de type sur chacun de ses éléments afin d'en extraire $L'(S)$. Sans cycles, un automate d'architecture engendre des langages $L(S)$ finis, et donc la procédure précédente ne pose pas de problèmes. Le résultat suivant traite du cas infini:

Théorème 1 *Soit S un sommet d'un automate d'architecture $\mathcal{A}$ qui contient des cycles et des contraintes de types. Il n'est que semi-décidable de savoir s'il existe une expression de foncteurs bien typée dans $L(S)$.*

Preuve. On prouve que $L'(S) = \emptyset$ est indécidable en émulant le problème de *correspondance des dominos de Post* par des automates d'architecture. Ce problème possèdera alors une solution si les contraintes de types d'un foncteur possèdent elles-mêmes des solutions.

Considérons un ensemble de dominos décrits par des paires $\{(u_i/v_i)\}_{i=1,\ldots,m}$ où les u_i et v_i sont des mots sur un alphabet fini Σ. Pour chaque lettre de l'alphabet Σ, on se donne un type inductif de la forme `type x = X of unit`. On concatène alors les lettres en utilisant des foncteurs. Par exemple:

```
type a = A of unit;;
type b = B of unit;;
module type S   = sig type t end;;
module F (M:S) = struct type t = b * M.t end;;
module M1       = struct type t = a       end;;
```

Ainsi, `F(M1)` aura un type `t` égal à `b * a`. Il est donc possible d'obtenir tous les mots u_i et v_i en utilisant des expressions de foncteurs. Pour chaque domino, on se donne également un type inductif de la forme `type domino_i = D_i of unit`. On peut alors appliquer la même technique que ci-dessus pour décrire les suites de dominos.

Le noyau de la spécification de l'automate d'architecture est alors défini par les éléments suivants:

```
module type S_UP    = sig type t type d end;;
module type S_DOWN = sig type s type d end;;
module type RESULT = sig ... end
module type JOIN =
  functor (M:S_UP) -> functor (N:S_DOWN with type s = M.t
                                        and type d = M.d) -> RESULT
module type INIT = sig end
```

La signature `S_UP` correspond au mot supérieur (stocké sous le type `t`) et la signature `S_DOWN` au mot inférieur (stocké sous le type `s`). La signature de foncteur `JOIN` inclut une contrainte de type qui indique que les types représentants les mots `N.s`, `M.t` doivent être égaux, ainsi que les types représentants les suites de dominos `N.d`, `M.d`. La signature `INIT` est un sommet initialisateur.

L'idée est alors de connecter à `S_UP` (resp. à `S_DOWN`) des cycles d'applications de foncteurs qui correspondent à chacun des mots u_i (resp. v_i) et qui modifient le type `t` (resp. `s`) comme indiqué ci-dessus. Chacun de ces cycles se termine en modifiant le type `d` correspondant afin d'indiquer le domino placé.

Par exemple, pour un domino (a/ab):

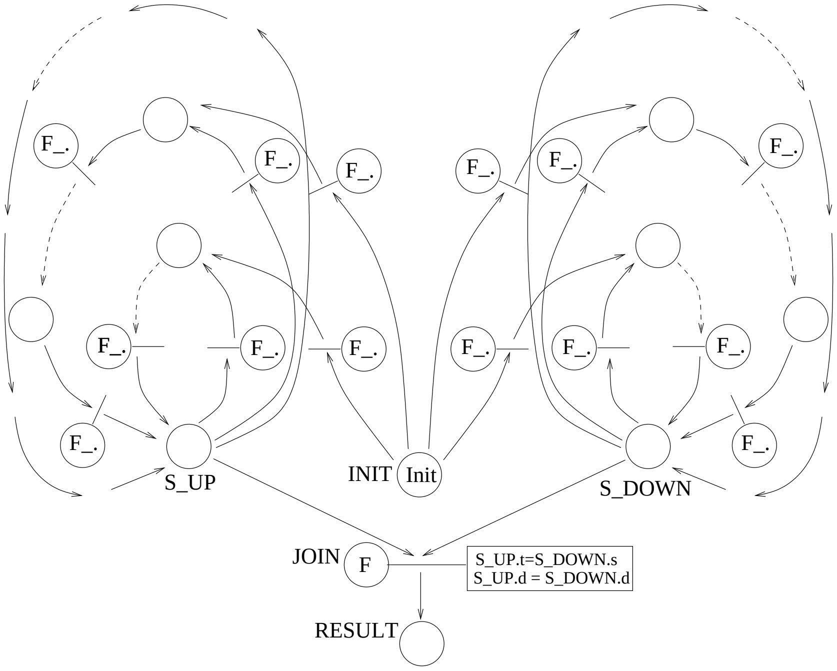


FIG. 1 – *L'aspect général de l'automate d'architecture d'une émulation d'un problème de* Post *(la notation* `F_.` *n'est qu'une simplification pour tous les foncteurs distincts insérés dans l'automate).*

```
module type S_t1a = sig type t1a type d end;;
module F_t1_a  (M:S_UP) = struct type t1a = a * M.t type d = M.d end;;
module F_t1_end(M:S_t1a)= struct type t   = M.t1a   type d = domino_1 * M.d end;;
module type S_s1a  = sig type s1a  type d end;;
module type S_s1ab = sig type s1ab type d end;;
module F_s1_a  (M:S_DOWN)= struct type s1a  = a * M.s   type d = M.d end;;
module F_s1_ab (M:S_s1a) = struct type s1ab = b * M.s1a type d = M.d end;;
module F_s1_end(M:S_s1ab)= struct type s = M.s1ab       type d = domino_1 * M.d end;;
```

On connecte alors le sommet initialisateur `INIT` à chacun de ces cycles d'expressions de foncteurs. Ces connections se concrétisent par des foncteurs qui écrivent la première lettre du premier mot correspondant (`init` est un type inductif différent de tous les autres qui initialise la suite de dominos):

```
module F_init_xx(I:INIT) = struct type txx = x  type d = init end
```

Pour le domino (a/ab) ci-dessus, cela correspond à ajouter:

```
module F_init_t1(I:INIT) = struct type t1a = a  type d = init end;;
module F_init_s1(I:INIT) = struct type s1a = a  type d = init end;;
```

Les seules implémentations qui sont alors intégrées en plus des foncteurs ci-dessus sont un module qui satisfait à `INIT` et un foncteur qui satisfait à `JOIN`:

```
module Init = struct end;;
module F (M:S_UP) (N:S_DOWN with type s = M.t
                            and type d = M.d) = struct ... end;;
```

L'aspect général de l'automate d'architecture est représenté en Fig. 1.

La question de décidabilité est alors de savoir si `L(RESULT)` est vide ou non, autrement dit de savoir si `F` est applicable. Ceci n'arrive que s'il existe dans `S_UP` et `S_DOWN` des implémentations qui satisfont aux contraintes de types `N.s=M.t` et `N.d=M.d` de `JOIN`. Mais l'automate d'architecture est construit de manière à ce que ces contraintes ne soit réalisées que si `M.t=` $u_{i_1}...u_{i_n} = v_{i_1}...v_{i_n}$ `=N.s` avec la même séquence de dominos $d_{i_1}...d_{i_n}$ `=N.d=M.d`. Autrement dit `F` n'est applicable que si le problème de *Post* de $\{(u_i/v_i)\}_{i=1,\ldots,m}$ est résolu. ◊

Remarquons que l'émulation ci-dessus demande beaucoup de foncteurs car le produit cartésien de type n'est pas associatif, par ex. `(a * b) * c` $\neq$ `a * b * c` (les types sont des arbres). Il faut donc ajouter les lettres une à une.

Le résultat précédent indique donc que, dans le cas général, la question principale que nous posions au début de cette partie est indécidable: *il n'est pas possible de prévoir si une signature d'un automate d'architecture est instanciable en faisant évoluer ce dernier.* Notons que ce résultat négatif ne l'est pas tant en pratique puisque l'énumération des candidats est possible dans de bonnes conditions par l'utilisation des grammaires. De plus, les expressions de foncteurs n'ont pas en général à devenir trop longues par la seule exploitation des cycles d'un automate. On peut d'autre part augmenter la part d'implémentation associée à un automate si un doute survient.

Une autre conséquence du résultat ci-dessus est le suivant:

Remarque 4 *Les contraintes de types partagés d'un automate d'architecture ne peuvent pas être toutes éliminées de manière constructive par des éclatements de sommets ou des modifications locales.*

Harper dans [Har01] Chap. 23.3 décrit cependant une méthode qui s'applique dans beaucoup de cas.

6. Discussion

Nous avons montré ici que *ML* (en particulier, *OCaml*) permet directement la construction de programmes dans le cadre d'une *programmation paramétrée* [Gog84, Gog96], et donc aussi d'une *programmation générative* [CE00]. Ce langage rend intrinsèquement possible la mise en oeuvre de techniques et d'outils de développement qui incluent spécification, validation et génération.

Certaines extensions du modèle peuvent être envisagées. Tout d'abord, certains points liés à l'automatisation des actions sur un automate d'architecture pourraient être étendus et affinés. Par exemple, la recherche des sommets auxquels une implémentation doit être associée pourrait utiliser le cadre décrit dans [ZW92]. Ensuite, l'intégration de la couche objet d'*OCaml* serait une généralisation naturelle; à nouveau, le système de type offre dans ce cas une facilité puisque les classes peuvent être explicitement typées et celles-ci peuvent trouver place dans des modules. D'autre part, la création d'un langage d'expressions de modules et de foncteurs (dans l'esprit de [Tra93, Gog96] et plus généralement de [MT00]) pourrait être réalisé par le biais de l'outil d'extension `camlp4`. Un automate d'architecture pourrait alors être décrit au travers de simples expressions. Par exemple, on exprimerait l'union de deux types abstraits en utilisant l'opérateur + (cf. Architecture 6). Ainsi, l'union de deux signatures `S1` et `S2` serait notée `module type S3 = S1 + S2` et l'union de deux modules, `module M3 = M1 + M2`. Finalement, l'outil de preuve *Coq* [Tea03] pourrait être intégré aux automates d'architecture afin d'obtenir des moyens d'exprimer et de vérifier des propriétés plus élaborées que celles offertes par le seul système de types de *ML*. Réciproquement, notons que *Coq* est en passe de posséder un système de modules similaire à celui de *ML* (même s'il n'intègre pas encore à ce jour les types de foncteurs et les foncteurs d'ordre supérieur). Les automates d'architecture pourraient donc s'y appliquer également.

Mes remerciements à S. Grivet, O. Ly et D. Renault pour nos discussions sur le sujet, et aux étudiants F. Coulon, S. Laporte, M. Marcal, B. Rezic, M. Serrut, C. Stentzel pour avoir participé à l'élaboration des prototypes du logiciel auquel il est fait allusion ici.

Références

[BS92] D. Batory and O'Malley S. The design and implementation of hierarchical software systems with reusable components. *ACM Transac. on Sofware Eng. and Meth.*, 1(4):355–398, 1992.

[CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming.* Addison Wesley, 2000.

[Cou02] G. Cousineau. Tilings as a programming exercise. *Theoretical Computer Science*, 281:207–217, 2002.

[DW85] M.D. Davis and E.J. Weyuker. *Computability, Complexity and Languages.* Academic press, 1985.

[GHJV95] E. Gamma, R. Helm, R. Jonhnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[Gog84] J. Goguen. Parameterized programming. *IEEE Trans. on Software Engineering*, SE-10(5):528–543, 1984.

[Gog96] J. Goguen. Parametrized programming and software architecture. In *Proceedings, Fourth Intl. Conf. on Software Reuse*, pages 2–11. IEEE Computer Society, April 1996.

[Har01] R. Harper. Programming in Standard ML, 2001. Carnegie Mellon University.

[HJ83] T. Hickey and Cohen J. Uniform random generation of strings in a context-free language. *SIAM J. Comput.*, 12(4):645–655, 1983.

[Kru92] C. Krueger. Software reuse. *ACM computing surveys*, 24:131–183, 1992.

[LDG+03] X. Leroy, Doligez D., J. Garrigue, Rémy D., and Vouillon J. *The Objective Caml system. Documentation and user's manual (3.07).* INRIA, September 2003.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.

[Pau97] L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1997.

[Tea03] The Coq Development Team. *The Coq Proof Assistant Reference Manual (Version 7.4).* INRIA, 2003.

[Tra93] W. Tracz. LILEANNA: a parameterized programming language. In IEEE Computer Society Press, editor, *Proceedings of the Second International Workshop on Software Reuse*, pages 66–78, 1993.

[VN03] D. Vandevoorde and Josuttis N.M. *C++ Templates.* Addison-Wesley, 2003.

[ZW92] A. Zaremski and J. Wing. Signature matching: A key to reuse. In *ACM SIGSOFT Conference*, pages 182–190, 1992.