Métaprogrammation fonctionnelle appliquée à la génération d'un DSL dédié à la programmation parallèle

Jocelyn Serot, Joel Falcou

Métaprogrammation fonctionnelle appliquée à la génération d'un DSL dédié à la programmation parallèle

J. Sérot[1] & J. Falcou[2]

1: LASMEA - UMR 6602 CNRS/UBP,
Campus des Cézeaux, 63177 Aubière CEDEX
Jocelyn.Serot@lasmea.univ-bpclermont.fr
2: IEF - U. Paris-Sud Orsay
joel.falcou@ief.u-psud.fr

Résumé

On décrit l'implémentation en MetaOcaml d'un petit langage dédié (DSL) à la programmation parallèle. Le langage repose sur la notion de *squelettes* qui autorisent la spécification de programmes parallèles par simple composition de constructeurs de haut niveau encapsulant des schémas communs et récurrents de parallélisme. On montre comment les facilités de *métaprogrammation* offertes par MetaOcaml permettent d'éliminer presque totalement le surcoût à l'exécution du code généré par rapport à une implantation du même programme écrite avec des primitives de bas niveau comme celles de mpi. Pour cela, la spécification haut niveau du programme est d'abord transformée en une représentation équivalente sous la forme d'un réseau de processus séquentiels communiquants puis cette représentation est utilisée pour générer *dynamiquement* le code exécuté par chaque processeur de la machine.

1. Introduction

De nos jours, le principal style de programmation utilisé pour exploiter les *clusters* de calcul est celui dit par *passage de messages*, supporté par exemple par des bibliothèques comme mpi. Le programmeur est amené à décomposer explicitement son application en processus communiquants et à ordonnancer "à la main" les communications entre ces processus. Cette approche, si elle permet à un programmeur expérimenté de tirer le meilleur parti d'une architecture donnée, conduit toutefois à des programmes longs à écrire et à mettre au point compte-tenu du très faible niveau d'abstraction utilisé.

Les *squelettes de parallélisation* ont été proposés en réponse à ce problème. Un *squelette* encapsule un *schéma de parallélisation* récurrent pour lequel une ou plusieurs implémentations efficaces sont connues. De tels squelettes se présentent classiquement comme des constructeurs génériques qui doivent être instanciés avec les fonctions spécifiques de l'application. Des exemples classiques de squelettes sont pipeline, farm et divide-and-conquer. Avec cette approche, le travail du programmeur se limite à choisir et combiner un ensemble de squelettes pris dans une base prédéfinie, sans qu'il ait à se préoccuper des détails de mise en œuvre du parallélisme sur l'architecture.

Les squelettes constituent un exemple de langage spécifique d'un domaine d'application (DSL, *Domain Specific Language*) répondant à un besoin d'abstraction. La nécessité de disposer d'un formalisme permettant de spécifier, puis de raisonner sur les programmes est une des raisons expliquant l'intérêt pour ces approches. De nombreux travaux ont été consacrés à la définition à l'implantation

de systèmes de programmation parallèles fondés sur les squelettes [1, 7, 2, 10, 9, 19]. Ces réalisations iffèrent essentiellement sur la manière dont le DSL correspondant est implanté. En pratique cela suppose de définir la syntaxe et la sémantique de ce DSL d'une part et les règles de transformation permettant de passer de cette spécification haut niveau à une implémentation concrète bas niveau – faisant appel à des primitives MPI par exemple – d'autre part.

Pour la plupart des réalisations citées, le DSL est implanté sous la forme d'une bibliothèque dédiée au sein d'un langage de programmation hôte (C++ ou Caml par exemple). Cette approche offre en effet deux avantages : elle évite d'avoir à implanter un analyseur lexical et syntaxique dédié et elle facilite l'interfaçage aux fonctions séquentielles (soit parce que ces fonctions sont écrites dans le langage hôte, soit parce que cet interfacage peut se faire via les facilités offertes par ce langage hôte). Mais, en contrepartie, cette manière de procéder conduit à des implémentations relativement peu efficaces, comparées au code "bas-niveau" qu'un programmeur aurait écrit pour le même problème. En effet, les squelettes étant par essence des objets d'ordre supérieur, leur support au sein d'un langage de programmation implique toujours un surcoût à l'exécution par rapport à un code bas-niveau faisant appel directement à des primitives MPI.

Nous expliquons dans ce papier comment les techniques d'*évaluation partielle* et de *métaprogrammation* – utilisés par ailleurs avec succès dans d'autres domaines d'applications – peuvent être exploitées pour résoudre le problème sus-cité. Plus précisément, nous décrivons l'implantation, en MetaOcaml, d'un petit DSL permettant

- la spécification de programmes parallèles sous la forme de combinaisons de squelettes
- la génération automatique d'un programme bas-niveau équivalent, sous la forme d'un ensemble de processus séquentiels communiquants
- l'exécution de ce programme sur une architecture de type *cluster* via la bibliothèque MPI.

La plan suivi est le suivant. La section 2 rappelle brièvement le principe de la métaprogrammation et comment celle-ci est supportée au sein du langage MetaOcaml. Dans la section 3 on explique en quoi cette technique présente un intérêt dans le contexte de la programmation parallèle par passage de messages. La section 4 est consacrée à la présentation d'un DSL dédié à programmation parallèle par squelettes. La sémantique de ce langage est explicitée en termes de réseaux de processus séquentiels communiquants puis on en dérive une implémentation en MetaOcaml. La section 5 présente les résultats expérimentaux obtenus avec cette implémentation. Un bref état de l'art est fait dans la section 6. La conclusion permet de faire le bilan des apports de ce premier prototype et d'indiquer les pistes de travail à court et moyen termes.

2. Métaprogrammation en MetaOcaml

De manière très générale, la métaprogrammation est la technique par laquelle un programme particulier peut analyser, transformer et générer d'autres programmes. La programmation *multi-niveaux* (*multi-staged programming*), supportée par le langage MetaOcaml, est une une instance particulière de cette technique.

MetaOcaml [20] est une extension du langage Ocaml [17] permettant

- de distinguer, grâce à un système du "quotation", au moins deux niveaux dans un programme : celui du code générant (le métaprogramme) et celui du code généré (les programmes objets).
- de générer et d'exécuter le code correspondant à ces programmes objets à l'exécution
- de garantir statiquement, via un système de typage adapté, que ceci se fait sans risque d'erreur à l'exécution

En pratique, cela signifie qu'un programme MetaOcaml peut, lors de son exécution, construire d'autres programmes et les exécuter. Ceci se fait à l'aide de trois constructeurs, nommés *Brackets*, *Escape* et *Run* :

- la construction `.< >.` (**brackets**) permet de délimiter les fragments des programmes objets et

donc à retarder l'exécution du code correspondant :

```
# let a = .< 1+2 >.;;
val a : int code = .<(1+2)>.
```

L'expression `1+2` n'est pas évaluée mais sa représentation est mémorisée dans la valeur `a`. Le type de `a` reflète ceci[1] : `int code` est le type des programmes qui, exécutés, produisent des valeurs de type `int`. Le fragment de programme correspondant peut être inséré dans d'autre programmes ou compilé et exécuté.

– l'opérateur `.~` (**escape**) permet justement d'insérer un fragment de programme objet dans un autre :

```
#let b = .< 3 * .~a >.;;
val b : int code = .<(3*(1+2))>.
```

– l'opérateur `.!` (**run**) compile et exécute un programme objet :

```
#let c = .!b;;
val c : int = 9
```

3. Métaprogrammation appliquée à la génération de code parallèle

Afin d'illustrer en quoi la métaprogrammation « à la MetaOcaml » présente un intérêt dans le cadre de la programmation parallèle considérons un programme (hypothétique) dans lequel un ensemble de $2N$ processus s'échangent des données selon le schéma suivant (cf figure 1) :

– un processus de rang pair envoie une donnée, générée par une fonction `f`, au processus de rang impair immédiatement supérieur puis attend une réponse de ce même processus
– un processus de rang impair reçoit une donnée du processus de rang pair immédiatement inférieur, applique une fonction `g` à cette donnée puis renvoie le résultat à ce même processus.

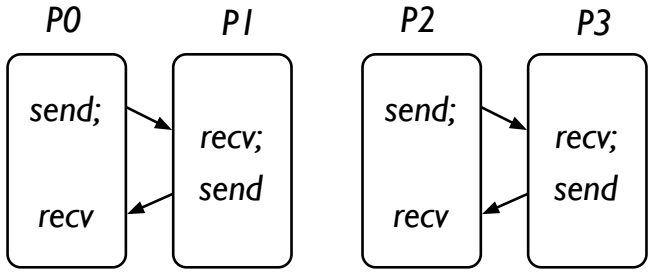

Fig. 1 – Exemple 1

Sur la quasi-totalité des architectures de type *cluster*, le modèle d'exécution est de type SPMD (*Single Program Multiple Data*) ce qui signifie que le programmeur écrit un seul programme qui s'exécute sur l'ensemble des processeurs. Ecrit avec des primitives de communication point à point de bas niveau de type `send` et `recv`[2] le code du programme précédent ressemble alors à ceci :

[1] Le type de `a` retourné par le compilateur MetaOcaml est en fait `('a, int) code` ; le paramètre `'a` est utilisé par le système de type pour garantir la correction du code dans le cas où des variables sont utilisées à plusieurs niveaux. Par souci de lisibilité, on l'omettra ici.

[2] On supposera par exemple que la signature de ces primitives est `val send : 'a -> pid -> unit` et `val recv : pid -> 'a`. De telles primitives sont par exemple offertes par la bibliothèque OcamlMpi.

```
let myrank = get_proc_rank() in
if myrank % 2 = 0 then
  send (f()) (myrank+1);
  recv (myrank+1)
else
  let y = recv (myrank-1)
  send (g y) (myrank-1);
```

Ici la primitive `get_proc_rank()` permet d'obtenir, à l'exécution, le rang[3] effectif du processeur qui exécute le programme et la conditionnelle décide, en fonction de ce rang, de la séquence d'instructions à exécuter.

On peut noter que dans ce programme, le schéma de communication est fixe et donc que, pour chaque processeur, le code à exécuter est connu dès que le rang du dit processeur est connu. On peut tirer parti de cette propriété pour réécrire le programme précédent de la manière suivante en MetaOcaml :

```
let pgm rank =
  if rank % 2 = 0 then
    .< send (f()) (rank+1); recv (rank+1) >.
  else
    .< let y = recv (rank-1) in send (g y) (rank-1) >.

let myrank = get_proc_rank() in
.!(pgm myrank)
```

Le code est exécuter est désormais *généré dynamiquement* au niveau de chaque processeur, puis exécuté. Cette technique est extrêmement puissante puisqu'elle permet de spécialiser – et donc d'optimiser – le code exécuté par chaque processeur en fonction de paramètres qui ne sont connus qu'à l'exécution (à la différence de systèmes de métaprogrammation opérant à la compilation comme les templates de C++ ou Template Haskell [12]). On peut objecter que ceci se fait au détriment du temps d'exécution mais dans la mesure où la génération du code n'est faite qu'une fois, au début de l'exécution du programme, le surcoût introduit peut être totalement recouvert par le gain en performance induit par le code optimisé.

Dans la suite, on va exploiter cette idée pour donner une implémentation efficace d'un DSL adapté à la programmation parallèle par squelettes.

4. Un DSL pour la programmation parallèle par squelettes

Le langage visé – appelons-le Skl– permet la spécification de programmes parallèles au moyen d'un nombre limité de combinateurs (les squelettes). Ces squelettes encapsulent complètement les activités de communication et de coordination associées au parallélisme. Les parties séquentielles de l'application sont spécifiées sous la forme de fonctions passées en paramètres. La structure de l'application est statique (pas de création / suppression dynamique de processus).

On décrit ici un jeu limité de squelettes, qui nous a servi à valider l'approche[4] :

[3] On utilise ici la terminologie mpi. Le rang permet de distinguer à l'exécution les copies d'un même programme dans un modèle SPMD.

[4] L'extension de ce jeu ne pose pas de problème *a priori* dès lors que ces squelettes peuvent être décrits dans les termes de l'algèbre de processus détaillée par la suite.

- le squelette PIPE encapsule un parallélisme de type flux : n étapes de calcul sont enchaînées sur les données d'entrée, chaque étape pouvant s'exécuter en parallèle sur des données distinctes ;
- le squelette PARDO encapsule un parallélisme de type tâche : n opérations sont appliquées en parallèle sur chaque donnée d'entrée ;
- le squelette FARM encapsule un parallélisme de type données : une même opération est appliquée à l'ensemble des données d'entrées ; le modèle d'exécution sous-jacent est celui dit en "ferme de processus", où un processus "maître" distribue dynamiquement les données à traiter à un ensemble de processus esclaves et collecte les résultats, afin de réaliser un équilibre de charge.

Dans la description précédente, les "étapes" (resp. "opérations") sont elles-mêmes des squelettes ; en d'autres termes, le langage supporte naturellement l'*imbrication* des squelettes. L'insertion des fonctions séquentielles se fait donc via un "squelette" particulier, nommé Seq.

La syntaxe abstraite des programmes SKL est donc la suivante :

$$\begin{array}{lcl} \Sigma & ::= & \mathsf{Seq}\ f \\ & | & \mathsf{Pipe}\ \Sigma_1 \ldots \Sigma_n \\ & | & \mathsf{Pardo}\ \Sigma_1\ \ldots \Sigma_n \\ & | & \mathsf{Farm}\ n\ \Sigma \\ f & ::= & \textit{fonction de calcul séquentielle} \\ n & ::= & \textit{entier} \geq 1 \end{array}$$

Imaginons par exemple un programme pouvant se décomposer en trois étapes : une étape de production de données (réalisée par une fonction f), une étape de traitement de ces données en parallèle à l'aide d'une ferme à trois esclaves (chacun exécutant la même fonction g) et une étape de traitement des résultats (réalisée par une fonction h) (cf figure 2). Ce programme s'écrit de la manière suivante en SKL :

$$\sigma = \mathsf{Pipe}(\mathsf{Seq}\ f, \mathsf{Farm}(3, \mathsf{Seq}\ g), \mathsf{Seq}\ h)$$

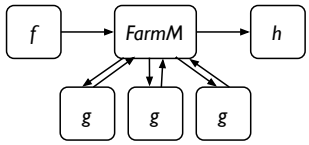

FIG. 2 – Exemple 2

4.1. Sémantique

On donne ici une sémantique des programmes SKL en termes de réseaux de processus séquentiels communiquants. Cette sémantique servira de base à la fonction d'interprétation qui, codée en METAOCAML, va permettre de générer le code à exécuter par chaque processeur de la machine parallèle cible.

On commence par définir ce que l'on entend exactement par réseau de processus séquentiels communiquant (RPSC).

Formellement, un RPSC est un triplet $\pi = \langle P, I, O \rangle$ où

- P est un ensemble de processus étiquetés, c-à-d. de paires (pid, σ) où pid est un identificateur (unique) de processus et σ un triplet contenant : une liste de *prédécesseurs* (pid des processus

p pour lesquels il existe un canal de communication de p au processus en question), une liste de *successeurs* (*pid* des processus p pour lesquels il existe un canal de communication du processus en question à p) et un descripteur Δ. On note $\mathcal{L}(\pi)$ l'ensemble des *pid*s d'un réseau de processus π. Pour un processus p, ses prédécesseurs, successeurs and descripteur seront notés $\mathcal{I}(p)$, $\mathcal{O}(p)$ et $\delta(p)$ respectivement.

- $I(\pi) \subseteq \mathcal{L}(\pi)$ désigne l'ensemble des processus p pour lesquels $\mathcal{I}(p) = \emptyset$
- $O(\pi) \subseteq \mathcal{L}(\pi)$ désigne l'ensemble des processus p pour lesquels $\mathcal{O}(p) = \emptyset$

Le descripteur de processus Δ est une paire $(instrs, kind)$ où $instrs$ est une séquence de *macro-instructions* et $kind$ un drapeau (la signification de ce drapeau sera donnée plus loin).

$$\begin{array}{ll} \Delta & ::= \langle instrs, kind \rangle \\ instrs & ::= instr_1, \ldots, instr_n \\ kind & ::= \mathsf{Regular} \mid \mathsf{FarmM} \end{array}$$

La séquence de macro-instructions décrivant le comportement du processus est implicitement itérée (en première approximation on considère que les processus ne se terminent pas ; gérer de manière "propre" la terminaison des processus complique la description du modèle et nous avons choisi de ne pas détailler ce point ici).

Les macros-instructions utilisent toutes un mode d'adressage *implicite* se référant à quatre variables nommées iv, ov, q et iws. Le jeu de macro-instructions est décrit ci-dessous. Dans les explications afférentes, p désigne le processus exécutant l'instruction en question.

$$\begin{array}{lll} instr & ::= & \mathsf{SendTo} \mid \mathsf{RecvFrom} \mid \mathsf{Comp}\ \text{fid} \mid \mathsf{RecvFromAny} \mid \mathsf{SendToQ} \mid \\ & & \mathsf{Ifq}\ instrs_1\ instrs_2 \mid \mathsf{GetIdleW} \mid \mathsf{UpdateWs} \end{array}$$

L'instruction SendTo envoie le contenu de la variable v au processus dont le *pid* apparaît en dernier dans $\mathcal{O}(p)$. L'instruction RecvFrom reçoit une donnée en provenance du processus dont le *pid* apparaît en premier dans $\mathcal{I}(p)$ et écrit cette donnée dans la variable iv. L'instruction Comp effectue un calcul en appelant un fonction séquentielle. L'argument de cette fonction est lu dans iv et son résultat écrit dans ov. L'instruction RecvFromAny attend (de manière non-déterministe) une donnée de l'ensemble des processus dont les *pid*s apparaissent dans $\mathcal{I}(p)$. La donnée reçue est placée dans la variable iv et le *pid* du processus émetteur est placé dans la variable q. L'instruction SendToQ envoie le contenu de la variable ov au processus dont le *pid* est donné dans la variable q. L'instruction Ifq compare la valeur contenue dans la variable q au *pid* listé en premier dans $\mathcal{I}(p)$. En cas d'égalité, la séquence d'instructions $instrs_1$ est exécutée ; sinon $instrs_2$ est exécutée. L'instruction UpdateWs lit la variable q et met à jour la variable iws en conséquence. La variable iws maintient la liste des processus esclaves libres pour un processus maître dans le cas d'un squelette de type FARM. L'instruction GetIdleW extrait un identificateur de processus de la liste iws et le place dans la variable q. Conjointement, ces deux dernières instructions encapsulent la stratégie utilisée au sein du squelette FARM pour assurer l'équilibre de charge.

On définit ensuite une algèbre simple permettant de construire, de manière compositionnelle des réseaux de processus. On utilisera pour cela les notations suivantes. Si $\mathcal{E}$ est un ensemble, on note $\mathcal{E}[e \leftarrow e']$ l'ensemble obtenu en remplaçant e par e' (en posant $\mathcal{E}[e \leftarrow e'] = \mathcal{E}$ si $e \notin \mathcal{E}$). Cette notation est supposée associative à gauche : $\mathcal{E}[e \leftarrow e'][f \leftarrow f']$ signifie $(\mathcal{E}[e \leftarrow e'])[f \leftarrow f']$. Si $e_1, \ldots, e_m$ est un sous-ensemble indexé de $\mathcal{E}$ et $\phi : \mathcal{E} \rightarrow \mathcal{E}$ une fonction, on notera $\mathcal{E}[e_i \leftarrow \phi(e_i)]_{i=1..m}$ l'ensemble $(\ldots((\mathcal{E}[e_1 \leftarrow \phi(e_1)])[e_2 \leftarrow \phi(e_2)])\ldots)[e_m \leftarrow \phi(e_m)]$. Sauf mention explicite, on notera $I(\pi_k) = \{i_k^1, \ldots, i_k^m\}$ et $O(\pi_k) = \{o_k^1, \ldots, o_k^n\}$. Par souci de concision, les listes $\mathcal{I}(o_k^j)$ et $\mathcal{O}(i_k^j)$ seront notées s_k^j et d_k^j respectivement. Pour les listes, on suppose définie l'opération de concaténation $++$ usuelle : si $l_1 = [e_1^1, \ldots, e_1^m]$ et $l_2 = [e_2^1, \ldots, e_2^n]$ alors $l_1{+}{+}l_2 = [e_1^1, \ldots, e_1^m, e_2^1, \ldots; e_2^n]$. La liste vide est notée $[]$. La longueur d'une liste l ou le cardinal d'un ensemble l sont tous deux notés $|l|$.

L'opérateur $\lceil . \rceil$ crée un réseau de processus contenant un seul processus à partir de son descripteur. La fonction NEW() fournit un nouveau *pid* à chaque appel.

$$\frac{\delta \in \Delta \qquad l = \textsc{New}()}{\lceil \delta \rceil = \langle \{(l, \langle [], [], \delta \rangle)\}, \{l\}, \{l\} \rangle} \quad (\textsc{Singl})$$

L'opérateur $\bullet$ "sérialise" deux réseaux de processus, en connectant les sorties du second aux entrées du premier :

$$\frac{\pi_i = \langle P_i, I_i, O_i \rangle \ (i = 1, 2) \qquad |O_1| = |I_2| = m}{\pi_1 \bullet \pi_2 = \langle (P_1 \cup P_2)[(o_1^j, \sigma) \leftarrow \phi_d((o_1^j, \sigma), i_2^j)]_{j=1\ldots m}[(i_2^j, \sigma) \leftarrow \phi_s((i_2^j, \sigma), o_1^j)]_{j=1\ldots m}, I_1, O_2 \rangle} \quad (\textsc{Serial})$$

La règle précédente utilise deux fonctions auxiliaires ϕ_s et ϕ_d définies comme suit :

$\phi_s((p, \langle s, d, \langle \delta, \mathsf{Regular} \rangle \rangle), p') = (p, \langle [p']{+}{+}s, d, \langle [\mathsf{RecvFrom}]{+}{+}\delta, \mathsf{Regular} \rangle \rangle)$
$\phi_d((p, \langle s, d, \langle \delta, \mathsf{Regular} \rangle \rangle), p') = (p, \langle s, d{+}{+}[p'], \langle \delta{+}{+}[\mathsf{SendTo}], \mathsf{Regular} \rangle \rangle)$
$\phi_s((p, \langle s, d, \langle \delta, \mathsf{FarmM} \rangle \rangle), p') = (p, \langle [p']{+}{+}s, d, \langle \delta, \mathsf{FarmM} \rangle \rangle)$
$\phi_d((p, \langle s, d, \langle \delta, \mathsf{FarmM} \rangle \rangle), p') = (p, \langle s, d{+}{+}[p'], \langle \delta, \mathsf{FarmM} \rangle \rangle)$

La fonction ϕ_s (resp. ϕ_d) ajoute un processus p' aux prédécesseurs (resp. successeurs) d'un processus p et met à jour sa liste de macro-instructions. Concrètement, cela consiste à ajouter au début (resp. à la fin) de cette liste une instruction RecvFrom (resp. SendTo), sauf pour les processus maîtres au sein d'un squelette de type FARM, pour lesquels la liste d'instructions n'est pas modifiée (ces processus sont identifiés à l'aide du drapeau *kind*, positionné à la valeur FarmM).

L'opérateur $\|$ met deux réseaux de processus en parallèle en fusionnant leurs entrées et sorties respectivement.

$$\frac{\pi_i = \langle P_i, I_i, O_i \rangle \ (i = 1, 2)}{\pi_1 \parallel \pi_2 = \langle P_1 \cup P_2, I_1 \cup I_2, O_1 \cup O_2 \rangle} \quad (\textsc{Par})$$

L'opérateur $\bowtie$ relie deux réseaux de processus en connectant chaque sortie et sortie du second à la sortie du premier. Cet opérateur est utilisé pour décrire la structure cyclique qui correspond à une ferme de processus :

$$\frac{\pi_i = \langle P_i, I_i, O_i \rangle \ (i = 1, 2) \qquad |O_1| = 1 \qquad |I_2| = m \qquad |O_2| = n}{\pi_1 \bowtie \pi_2 = \langle (P_1 \cup P_2)[(o_1, \sigma) \leftarrow \Phi((o_1, \sigma), I(\pi_2), O(\pi_2))][(i_2^j, \sigma) \leftarrow \phi_s((i_2^j, \sigma), o_1)]_{j=1\ldots m} [(o_2^j, \sigma) \leftarrow \phi_d((i_2^j, \sigma), o_1)]_{j=1\ldots n}, I_1, O_1 \rangle} \quad (\textsc{Join})$$

où $\Phi(p, ps_s, ps_d) = \Phi_s(\Phi_d(p, ps_d), ps_s)$ et Φ_s (resp. Φ_d) est la généralisation de la fonction ϕ_s (resp. ϕ_d) à une liste de processus :

$\Phi_s(p, [p_1, \ldots, p_n]) = \phi_s(\ldots, \phi_s(\phi_s(p, p_1), p_2), \ldots, p_n)$
$\Phi_d(p, [p_1, \ldots, p_n]) = \phi_d(\ldots, \phi_d(\phi_d(p, p_1), p_2), \ldots, p_n)$

On peut dès lors expliciter la fonction $\mathcal{C}$ transformant un programme SKL en un réseau de processus :

$$
\begin{aligned}
\mathcal{C}[[\mathsf{Seq}\ f]] &= \lceil f \rceil \\
\mathcal{C}[[\mathsf{Pipe}\ \Sigma_1 \ldots \Sigma_n]] &= \mathcal{C}[[\Sigma_1]] \bullet \ldots \bullet \mathcal{C}[[\Sigma_n]] \\
\mathcal{C}[[\mathsf{Farm}\ n\ \Sigma]] &= \lceil \mathsf{FarmM} \rceil \bowtie (\mathcal{C}[[\Sigma]]_1 \parallel \ldots \parallel \mathcal{C}[[\Sigma]]_n) \\
\mathcal{C}[[\mathsf{Pardo}\ \Sigma_1\ \ldots \Sigma_n]] &= \mathcal{C}[[\Sigma_1]] \parallel \ldots \parallel \mathcal{C}[[\Sigma_n]]
\end{aligned}
$$

où FarmM est le descripteur encapsulant le comportement d'un processus maître au sein d'un squelette FARM :

$$\Delta(\mathsf{FarmM}) = \langle [\mathsf{RecvFromAny}; \mathsf{Ifq}\ [\mathsf{GetIdleW}; \mathsf{SendToQ}]\ [\mathsf{UpdateWs}; \mathsf{SendTo}]], \mathsf{FarmM} \rangle$$

4.2. Implémentation en MetaOcaml

L'intégration ("*embedding*") du langage SKL au langage hôte (METAOCAML donc ici) se fait via une fonction dite d'interprétation (`run`) qui se charge de convertir une spécification SKL en un code exécutable, appelé *code résiduel*, sur chaque processeur.

Concrètement, le programme exemple donné au paragraphe précédent s'écrira alors de la manière suivante :

```
let f () = (* code de la fonction séquentielle générant les données à traiter *)
let g x = (* code de la fonction séquentielle de traitement des données *)
let h x = (* code de la fonction séquentielle de traitement des résultats *)
let pgm =  Pipe [seq f; Farm(3, seq g); seq h]
let _ = run pgm
```

Le programme SKL est ici décrit via un type algébrique récursif :

```
type skl_tree =
    Seq of seq_comp
  | Pipe of skl_tree list
  | Pardo of skl_tree list
  | Farm of int * skl_tree
```

La définition exacte du type `seq_comp` sera donnée plus loin. Il suffit de dire pour l'instant que l'on dispose d'une fonction d' "encapsulation" `seq` permettant de transformer toute fonction séquentielle `f : 'a->'b` en une valeur de type `seq_comp`.

La génération du code résiduel par la fonction `run` se fait en deux temps :

1. la représentation arborescente du programme parallèle est d'abord transformée en une représentation de ce même programme sous la forme d'un réseau de processus séquentiels communiquants (RPSC)
2. puis, le code résiduel est généré en confrontant cette représentation intermédiaire – au sein de laquelle chaque processus est repéré par un *pid* unique – avec le rang effectif du processeur sur lequel la fonction d'interprétation `run` s'exécute ; en d'autres termes ne sera généré sur le processeur i que le code correspondant au processus ayant le *pid* i au sein du RPSC.

```
type process_network = {
    procs: (pid * process_desc) list;
    inputs: pid list;
    outputs: pid list;
  }

and process_desc = {
    kind: process_kind;
    instrs: instr list;
    recvfrom: pid list;
    sendto: pid list;
  }

and process_kind = Regular | FarmM
```

```
and instr =
  | Comp of seq_comp
  | SendTo
  | RecvFrom
  | RecvFromAny
  | SendToQ
  | Ifq of instr list * instr list
  | GetIdleW
  | UpdateWs

and seq_comp = process_state -> unit

and process_state = {
    mutable iv: seq_val;
    mutable ov: seq_val;
    mutable q: pid;
    mutable iws: pid list
  }
```

FIG. 3 – Listing 1

4.2.1. Génération du réseau de processus

Le listing 1 donne la traduction des principaux types de données utilisés pour cette étape.

Les types de `process_network`, `process_desc`, `process_kind` et `instr` découlent immédiatement des définitions données à la section 4.1 et n'appellent pas de remarque particulière. Les points intéressants concernent les types `process_state` et `seq_comp`.

Le type `process_state` regroupe les quatre variables manipulées par les macro-instructions et le type `seq_comp` traduit désormais que l'effet des fonctions de calcul séquentielles est de mettre à jour l'état d'un processus. En pratique, une telle fonction sera appelée avec comme argument le contenu de la variable `iv` et son résultat sera écrit dans la variable `ov`[5].

Typage. Une difficulté se pose toutefois lorsque l'on cherche à préciser le type `seq_val` ; en effet, le type des composantes `iv` et `ov` dépend *in fine* du type de la fonction séquentielle de calcul exécutée par le processus. En supposant que le type de cette fonction soit `'a->'b`, on pourrait imaginer paramétrer le type `process_state` :

```
type ('a,'b) process_state = {
    mutable iv: 'a;
    mutable ov: 'b;
    ... }
```

Mais dans ce cas, le type `process_network` devient `('a,'b) process_network` et il est impossible de définir des réseaux dont les processus opèrent sur des données de types différents[6].

Dans le contexte des DSLs métaprogrammés, ce problème est bien connu et apparaît chaque fois que les langages objet et hôte sont statiquement typés (comme ici). La solution consiste en général

[5] Une fonction séquentielle de calcul doit donc avoir un type de la forme `'a -> 'b`. On peut toujours de se ramener, par décurryfication si besoin, à cette forme de signature.

[6] Ce problème est analogue à celui auquel on se heurte classiquement lorsque l'on cherche à définir une fonctionnelle `compose` généralisée : si on écrit `let rec gen_compose = function [f] -> f | f : :fs -> compose f (gen_compose fs)`, avec `let compose f g = function x -> g (f x)` alors le type inféré est `('a -> 'a) list -> ('a -> 'a)` ce qui oblige toutes les fonctions a avoir la même signature.

à introduire un type « universel » encapsulant de manière uniforme tous les types susceptibles d'être manipulés par les fonctions séquentielles de calcul du programme. Dans notre cas, on définit alors le type `seq_val` comme :

```
type seq_val = Int of int | Float of float | IntArray of int array | ...
```

et un *wrapper* assure l'étiquetage (resp. « dés-étiquetage ») des données en provenance des (resp. passées aux) fonctions séquentielles de calcul. Par exemple, si `f` est une fonction de type `int -> float`, on définit :

```
let f' = function (Int x) ->  Float (f x)
```

et c'est cette fonction qui est passée au constructeur `Seq` afin de construire les programmes Skl :

```
let seq f = Seq (fun s -> s.ov <- f s.iv)
```

Remarque. Cette solution oblige évidemment à créer un *wrapper* spécifique pour chaque fonction séquentielle et à recompiler l'interpréteur à chaque fois que des types nouveaux sont requis. Elle induit par ailleurs un surcoût à l'exécution. On peut éviter ceci en contournant les règles du typage statique d'Ocaml au moyen des primitives `Obj.repr` et `Obj.obj`. On on a alors simplement :

```
type seq_val = Obj.repr

let (seq : ('a -> 'b) -> seq_comp) =
  function f -> Seq (fun s -> s.ov <- Obj.repr (f (Obj.obj s.iv)))
```

Mais ceci se fait évidemment au détriment de la sûreté puisque le contrôle de type est alors *de facto* inhibé. Par exemple, un programme comme `Pipe [sql f; sql g]`, où `f:int->float` et `g:int->int`, est accepté et déclenche évidemment une erreur à l'exécution.

En pratique, nous disposons de deux versions de l'interpréteur : la première, utilisant la définition « universelle » de `seq_val` permet de vérifier la cohérence des types dans un programme Skl. La seconde, utilisant `Obj.repr`, permet de supprimer l'*overhead* lié au *tagging* dynamique des valeurs.

Les opérateurs $\lceil . \rceil$, $\bullet$, $\|$ et $\bowtie$ sont implantés par traduction directe des règles de productions de la sémantique sur la base des types définis ci-dessus. On ne reproduira donc ici que le code associé aux deux premiers :

```
let singl d =
  let p = new_pid () in
  { procs = [p, d]; inputs = [p]; outputs = [p] }

let serial pn1 pn2 =
  if List.length pn1.outputs = List.length pn2.inputs then
    { procs = List.map
        (function (pid, pd) ->
          if List.mem pid pn1.outputs then
            let dst = List.assoc pid (List.combine pn1.outputs pn2.inputs) in
            (pid, add_dst pd dst)
          else if List.mem pid pn2.inputs then
            let src = List.assoc pid (List.combine pn2.inputs pn1.outputs) in
            (pid, add_src pd src)
```

```
          else (pid, pd))
        (pn1.procs @ pn2.procs);
      inputs = pn1.inputs;
      outputs = pn2.outputs }
  else failwith "cannot serialize process networks: size mismatch"
```

Dans ce qui précède, les fonctions `add_src` et `add_dst` implémentent les fonctions ϕ_s et ϕ_d. Par exemple :

```
let add_dst pd1 pid2 =
  match pd1.kind with
    Regular -> {pd1 with instrs = pd1.instrs @ [SendTo]; sendto = pd1.sendto @ [pid2]}
  | FarmM -> {pd1 with sendto = pd1.sendto @ [pid2]}
```

Enfin, la fonction `expand_tree` implémente la fonction de conversion $\mathcal{C}$ introduite à la section 4.1 par simple filtrage sur le type récursif représentant les programmes Skl :

```
let (expand_tree : skl_tree -> process_network) = function t ->
  let rec expand = function
      Seq f -> singl {kind=Regular; instrs=[Comp f]; recvfrom=[]; sendto=[]; workers=[]}
    | Pipe [] -> failwith "empty pipe"
    | Pipe [t] -> expand t
    | Pipe (t::ts) ->
        serial (expand t) (expand (Pipe ts))
    | Pardo [] -> failwith "empty pardo"
    | Pardo [t] -> expand t
    | Pardo (t::ts) ->
        par (expand t) (expand (Pardo ts))
    | Farm (n, t) ->
        if n > 0 then
          let workers =
            let rec create n =
              if n=1 then expand t
              else
                par (expand t) (create (n-1)) in
            create n in
          join (farmer workers.inputs) workers
        else
        failwith "cannot expand farm with n<=0 workers"
  in
  reset_pid ();
  expand t

let farmer ws = singl {
  kind=FarmM;
  instrs=[RecvFromAny; Ifq ([GetIdleW; SendToQ],[UpdateWs; SendTo])];
  recvfrom=[]; sendto=[]; workers=ws }
```

4.2.2. Génération du code résiduel

La fonction `code` génère, à partir de la représentation du réseau de processus calculée à l'étape précédente, le code résiduel sur un processeur de rang donné :

```
let code pn rank =
  let mydesc = List.assoc rank pn.procs in
 .< while true do .~(code_instrs mydesc mydesc.instrs) done >.
```

Le code est "assemblé" par simple parcours de la liste de macro-instructions associée au descripteur du processus. C'est à ce niveau que la variable d'état du processus est créée[7]. Le contenu initial des composantes *iv*, *ov* et *q* est indifférent. Les processus de type *FarmM* voient leur composante *iws* initialisée avec la liste des processus esclaves auxquels ils sont reliés :

```
let code_instrs pd is =
  let s = { iv=Obj.repr 0; ov=Obj.repr 0; q=0; iws=pd.workers } in
  List.fold_left
    (fun c i -> .< begin .~c; .~(code_instr pd s i) end >.)
    .< () >.
    is
```

Enfin la fonction `code_instr` génère le code correspondant à chaque macro-instruction :

```
let rec code_instr pd s = function
    Comp f -> .< f s >.
  | SendTo ->
      let dst = List.hd (List.rev pd.sendto) in
     .< Mpi.send s.ov dst 0 Mpi.comm_world  >.
  | RecvFrom ->
      let src = List.hd (pd.recvfrom) in
     .< s.iv <- Obj.repr (Mpi.receive src Mpi.any_tag Mpi.comm_world) >.
  | RecvFromAny ->
     .< let r,q,_ = Mpi.receive_status Mpi.any_source Mpi.any_tag Mpi.comm_world in
        s.ov <- Obj.repr r; s.q <- q >.
  | SendToQ ->
     .< Mpi.send s.ov s.q 0 Mpi.comm_world >.
  | Ifq (is1,is2)->
     .< if s.q = List.hd pd.recvfrom
        then .~(code_instrs pd is1) else .~(code_instrs pd is2) >.
  | GetIdleW ->
      .< begin s.q <- List.hd s.iws; s.iws <- List.tl s.iws end >.
  | UpdateWs ->
      .< s.iws <- s.iws @ [s.q] >.
```

Une instruction Comp f produit un appel à la fonction séquentielle f. Les instructions SendTo, SendToQ, RecvFrom et RecvFromAny sont traduites directement en appel aux primitives MPI (on utilise ici la bibliothèque OcamlMpi [18]). Pour l'instruction Ifq l'opérateur d'échappement de MetaOcaml permet d'insérer directement dans le code produit les fragments correspondant aux deux branches de la conditionnelle. Concernant les instructions GetIdleW et UpdateWs, on s'est contenté ici d'implanter une simple gestion du *pool* d'esclaves en tourniquet en manipulant la liste *iws* (*idle workers*) en FIFO (retrait en tête par l'instruction GetIdleW, insertion en queue par l'instruction UpdateWs). D'autres stratégies de gestion sont évidemment envisageables.

La fonction d'interprétation principale s'écrit alors aisément :

[7] On tire parti ici de la possibilité offerte par MetaOcaml d'utiliser une variable définie au niveau i aux niveaux $i+n$ ("cross-stage persistence").

```
let size = Mpi.comm_size Mpi.comm_world
let myrank = Mpi.comm_rank Mpi.comm_world

let run pgm =
   let pn = expand_tree pgm  in
   let mycode = code pn myrank in
   .! (mycode)
```

A titre d'exemple, voici le code résiduel produit sur chaque processeur pour le programme `Pipe[seq f; seq g; seq h]` :

PID	Code
2	`.< while true do f s; Mpi.send s.ov 1 0 Mpi.comm_world done >.`
1	`.< while true do s.iv <- Obj.repr (Mpi.receive 2 0 Mpi.comm_world); g s; Mpi.send s.ov 0 0 Mpi.comm_world done >.`
0	`.< while true do s.iv <- Obj.repr (Mpi.receive 1 0 Mpi.comm_world); h s done >.`

Ce code est identique à celui qu'aurait écrit un programmeur MPI expérimenté.

5. Résultats

Nous avons évalué l'impact de cette technique d'implémentation en mesurant le surcoût (*overhead*) ρ en temps d'exécution qu'elle introduit par rapport à un code parallèle écrit directement à l'aide de primitives MPI (Ocaml + bibliothèque OcamlMPI v 1.0.1)[8].

Ce surcoût a été mesuré pour deux types de programmes : des programmes ne mettant en jeu qu'un seul squelette et des programmes pour lesquels plusieurs squelettes sont imbriqués à une profondeur arbitraire. Pour le premier type de test, on observe l'effet de deux paramètres : la durée d'exécution τ de la fonction de calcul séquentielle et la "taille" N du squelette (cette taille correspond au nombre d'étages pour un squelette PIPE, au nombre de tâches parallèles pour un squelette PARDO et au nombre d'esclaves pour un squelette FARM). Pour le second type de test, nous nous sommes limités au cas du squelette FARM. On évalue alors les performances d'un programme formé de P squelettes FARM imbriqués, chacun mettant en jeu ω esclaves.

Les résultats ont été obtenus sur un *cluster* à 30 noeuds de type PowerPC G5 pour N variant entre 2 et 30 et $\tau = 1ms, 10ms, 100ms, 1s$.

Pour le squelette PIPE, ρ n'excède jamais 2 %. Pour FARM et PARDO, ρ reste inférieur à 3 % et devient négligeable pour $N > 8$ ou $\tau > 10ms$. En cas d'imbrication, le pire cas est obtenu pour $P = 4$ et $\omega = 2$; ρ varie alors entre 7 % et 3 % quand τ passe de $1ms$ à $1s$.

Ces résultats sont bien meilleurs que ceux obtenus avec des systèmes de programmation par squelettes n'exploitant pas la métaprogrammation. Dans l'implémentation décrite dans [13], par exemple, où les squelettes sont implantées comme de simples fonctions d'ordre supérieur ordonnançant dynamiquement (c-à-d. *à l'exécution*) des primitives MPI, le surcoût était de l'ordre de 10 à 20 %. Pour la dernière version du système SKIPPER [14], dans laquelle les squelettes étaient traduits en graphes flots de données exécutés par un interpréteur dédié, ce surcoût atteignait parfois 100 %. D'autres implantations, comme celle de Kuchen [7], dans laquelle les squelettes sont traduits par des appels de méthodes virtuelles en C++, font aussi état de surcoût à l'exécution – par rapport à du code C+MPI dans ce cas - de l'ordre de 20 à 120 %.

[8] En utilisant la version « optimisée » de l'interpréteur, *i.e.* celle exploitant les primitives `Obj.repr` et `Obj.obj`.

6. Travaux connexes

L'idée d'exploiter la métaprogrammation pour implanter de manière efficace un DSL n'est pas neuve. Le principe sous-jacent consiste en transformer par *évaluation partielle* un couple (interpréteur de ce langage, programme dans ce langage) en un programme compilé . Les techniques de programmation dite *générative* [6] qui en découlent visent en général à résoudre la tension entre abstraction et performances, comme dans notre cas. Les bibliothèques FFTW [4], EVE [3] et SPIRAL [15] peuvent être vues comme des illustrations de cette approche.

Les langages fonctionnels offrent un substrat favorable pour la métaprogrammation grâce en particulier aux fonctions d'ordre supérieur et à la possibilité de coder très facilement la syntaxe abstraite du DSL sous la forme d'un type algébrique du méta-langage. Dans [11], Sheard fait une revue des techniques de métaprogrammation dans le contexte de la programmation fonctionnelle. Il y introduit notamment le langage MetaML qui ajoute au langage ML des annotations permettant de transformer des expressions ML en fragment de code, d'insérer des fragments de code dans d'autres fragments de code et d'exécuter des fragments de code. Le langage MetaOcaml [20] développé par Taha *et al.* est une extension similaire du langage Ocaml. Le langage *Template Haskell* [12] ajoute, de son coté, un large panel de facilités pour la métaprogrammation au langage *Haskell*, avec notamment des possibilités pour le méta-langage d'accéder à sa propre syntaxe abstraite (instrospection). En revanche, et contrairement à MetaML et MetaOcaml, la génération du code résiduel a lieu à la compilation et ne peut donc pas prendre en compte des données dynamiques.

Les problématiques générales liées à la définition d'un DSL pour le parallélisme sont étudiées par Lengauer dans [8]. Mais, dans ce domaine, seul Herrmann, dans [5] semble avoir exploré les possibilités offertes par la génération dynamique de code avec MetaOcaml pour implanter un tel DSL. Le système qu'il décrit est similaire à celui présenté ici mais en diffère toutefois en plusieurs points. Premièrement, le modèle de parallélisme sur lequel il repose est restreint aux schémas fixes et déterministes (structures série-parallèle) ; il est donc impossible d'y exprimer des squelettes pour lesquels l'ordonnancement des communications n'est pas connu à la compilation (comme farm). Deuxièmement, dans l'approche décrite, les squelettes ne font pas à proprement partie du DSL mais sont définis à partir du vocabulaire de ce dernier. Enfin, la sémantique du langage n'est pas définie formellement mais directement encodée dans l'interpréteur en MetaOcaml (il n'y pas de représentation intermédiaire explicite sous la forme de réseau de processus communiquants en particulier).

7. Conclusion

Nous avons montré dans cet article comment les facilités de métaprogrammation offertes par le langage MetaOcaml permettent d'implanter efficacement un DSL dédié à la programmation parallèle en conciliant haut niveau d'abstraction et performances. Le fait que le code résiduel soit généré à l'exécution est ici un avantage dans la mesure où cette génération peut prendre en compte la configuration effective de la machine (nombre de processeurs, ...). Bien sur le temps requis pour générer ce code s'ajoute au temps d'exécution de l'application elle-même mais, pour des problèmes de taille suffisante, il reste négligeable.

Le prototype décrit ici ne génère que du *bytecode*, le compilateur natif de MetaOcaml n'étant pas disponible sur toutes les plateformes. Dans la pratique ce point n'est pas critique dans la mesure où il est parfaitement possible d'appeler des fonctions de calcul séquentielles écrites en C, C++ ou Fortran en s'appuyant sur la *Foreign Function Interface* d'Ocaml. Une autre approche consisterait à utiliser les facilités d'*offshoring* offertes par MetaOcaml, c-à-d. la possibilité de générer automatiquement du code C à partir de code Ocaml.

Le principal problème à régler dans notre implémentation concerne toutefois le typage. La solution retenue, avec deux versions de l'interpréteur, n'est pas théoriquement satisfaisante. Il reste donc à voir

dans quelle mesure certaines techniques d'élimination automatique des étiquettes de typage, décrites par exemple dans [16], pourraient s'appliquer.

A plus long terme nous envisageons d'appliquer la techniques présentée ici à la réimplémentation d'autres systèmes de programmation parallèle fondés sur les squelettes, comme Skipper[14] ou OcamlP3L[19].

Références

[1] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L : A Structured High Level Programming Language And Its Structured Support. *Concurrency : Practice and Experience*, pages 225–255, 1995.

[2] M. Cole. Bringing Skeletons out of the Closet : A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 3 :389–406, 2004.

[3] J. Falcou and J. Sérot. EVE : an object-oriented SIMD library. *Scalable Computing : Practice and Experience*, 6(4) :31–42, 2005.

[4] Kang Su Gatlin and Larry Carter. Faster FFTs via architecture-cognizance. In *IEEE PACT*, pages 249–260, 2000.

[5] Christoph A. Herrmann. Generating message-passing programs from abstract specifications by partial evaluation. *Parallel Processing Letters*, 15(3) :305–320, 2005.

[6] Neil D. Jones and Arne J. Glenstrup. Program generation, termination, and binding-time analysis. In *ICFP '02 : Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 283–283, New York, NY, USA, 2002. ACM Press.

[7] H. Kuchen. A skeleton library. *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 620–629, 2002.

[8] Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors. *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*. Springer, 2004.

[9] G. Michaelson N. Scaife and S. Horiguchi. Parallel Standard ML with Skeletons. *Scaleable Computing Practise and Experience*, 6(4), 2006.

[10] J. Sérot and D. Ginhac. Skeletons for parallel image processing : an overview of the SKiPPER project. *Parallel Computing*, 28(12) :1785–1808, Dec 2002.

[11] Tim Sheard. Accomplishments and research challenges in meta-programming. In *SAIG 2001 : Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44, London, UK, 2001. Springer-Verlag.

[12] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

[13] J. Sérot Embodying parallel functional skeletons : an experimental implementation on top of MPI. *3rd Intl Euro-Par Conference on Parallel Processing*, Aout 1997, Passau. Volume 1300 of LNCS, pp 629–633, Springer.

[14] J. Sérot. Tagged-token data-flow for skeletons. *Parallel Processing Letters*, 11(4) :377-392, 2002.

[15] José M. F. Moura, Jeremy Johnson, Robert W. Johnson, David Padua, Viktor K. Prasanna, Markus Püschel, Bryan Singer, Manuela Veloso, and Jianxin Xiong. Generating platform-adapted dsp libraries using SPIRAL. In *High Performance Embedded Computing (HPEC)*, 2001.

[16] Emir Pašalic, Walid Taha, Tim Sheard. Tagless staged interpreters for typed languages. *SIGPLAN Not.*, 37(9) : :218–229, 2002.

[17] `http ://caml.inria.fr`

[18] `http ://caml.inria.fr/cgi-bin/hump.en.cgi ?contrib=401`

[19] `http ://ocamlp3l.inria.fr`

[20] `http ://www.metaocaml.com`