Sémantiques d'un langage de squelettes

Nicolas Lupinski[1,2] Joel Falcou[1] & Christine Paulin-Mohring[1,2]

1: Laboratoire de Recherche en Informatique (Université Paris-Sud, CNRS)
91405 Orsay Cedex, France
2: INRIA Saclay - Île-de-France, projet Toccata
nicolas.lupinski@gmail.com
Joel.Falcou@lri.fr
Christine.Paulin@lri.fr

1 Introduction

La programmation des machines parallèles modernes est devenue une tâche relativement complexe de part la multiplicité des architectures et des modèles de programmation disponibles. Contrairement à la programmation séquentielle qui a su développer en son sein un écosystème de méthodes et de bonnes pratiques, la programmation parallèle reste, le plus souvent, non structurée et réservée à des experts. Pour pallier à ce manque de méthode, plusieurs outils pour la programmation parallèle structurée ont fait l'objet d'études. Parmi eux, on peut citer les outils basés sur des modèles de coût comme LogP[5] ou BSP[19] et ses variantes; les modèles à patrons comme les Design Pattern parallèles ou les Squelettes Algorithmiques[4]. Si le design et l'implantation de langages et de bibliothèques basés sur ces modèles[7, 13, 11] est un sujet classique[10], peu d'entre eux font l'objet d'une vérification formelle. Hormis BSP et les travaux de Gava[9], peu de vérification formelle et encore moins de preuve ont été proposées pour les modèles à base de squelettes bien que des sémantiques opérationnelles[1, 2] et/ou fonctionnelles[6] aient été définies. Alors que l'utilisation du parallélisme se démocratise dans de nombreux domaines applicatifs, l'absence de vérification pour ces modèles de programmation est potentiellement un frein pour leur acceptation dans certains scénarios industriels critiques.

Cet article introduit un mini-langage de squelettes (Skel) suffisant pour exprimer les schémas de programmation parallèle les plus usités. Le langage est présenté dans la section 1.1 et des exemples d'utilisation sont détaillés dans la section 1.2. Nous définissons dans la section 2 une sémantique relationnelle pour le langage Skel dans laquelle les entrées et sorties sont vues comme des listes de données, cette sémantique a été formalisée en Coq et permet de raisonner sur la correction de la parallélisation. Nous introduisons ensuite dans la section 3 une sémantique opérationnelle en montrant comment le langage peut être traduit dans le join-calculus. Ceci permet de préciser les schémas de communication retenus, de gérer la terminaison du processus. Nous justifions la correspondance entre les deux sémantiques. De plus, par traduction vers JoCaml nous obtenons une implantation des processus.

1.1 Langage de squelettes

Les squelettes de parallélisation ont été proposés par Cole[4] comme une réponse au problème de l'expressivité des outils de développement parallèle comme MPI ou OpenMP qui forcent le développeur à revoir, pour chaque application et chaque architecture, l'implantation de ses algorithmes. Un squelette est ainsi défini comme un schéma parallèle récurrent encapsulé sous la forme d'une fonction d'ordre supérieur. Des exemples classiques de squelettes sont le pipeline, la ferme et le schéma *divide-and-conquer*. Avec cette approche, le travail du programmeur se limite à choisir et à 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. La qualité – ou du moins l'expressivité – d'un jeu de squelettes est alors mesurée par la facilité avec laquelle un développeur est capable, à partir d'un jeu limité mais adéquat de squelettes, de reconstruire des applications arbitraires.

Dans [7], Falcou et al. avaient proposé un tel sous-ensemble de squelettes inspiré des squelettes canoniques de Cole et augmenté par un système d'équilibrage de charge du type "ferme" proposé par Kuchen[16]. Ce langage – nommé Skel – présente les constructions suivantes:

- **Seq** convertit une fonction utilisateur de type $\alpha \to \beta$ en processus combinable avec d'autres squelettes;

- **Pipe** effectue la composition parallèle de deux squelettes via un canal de communication asynchrone;
- **Par** orchestre l'exécution indépendante de deux squelettes. Le type de retour de **Par** est le produit des sorties de ses composants;
- **Farm** est un système d'équilibrage de charge qui distribue de manière asynchrone des éléments d'un flux de données à une réserve de squelettes esclaves. On distinguera les "fermes ordonnées" (**OrderedFarm**) qui, malgré l'aspect asynchrone de leurs calculs, assurent l'ordre de leurs sorties et les "fermes non-ordonnées" (**UnorderedFarm**) qui émettent leur résultat au fur et à mesure de leur disponibilité;
- **Map** qui effectue de manière synchrone le découpage, traitement et fusion d'un élément du flux.

Pipe, **Farm** et **Par** implémentent ainsi des primitives classiques de parallélisme de tâches, délégant le parallélisme de données à **Map**. A ces squelettes classiques, nous ajoutons un jeu de squelettes auxiliaires qui permettent d'exprimer des modifications sur le flux de données lui même :

- **ToStream** (resp. **FromStream**) transforme une entrée en un flux de sorties (resp. récupère un flux d'entrée pour produire une sortie);
- **Rearrange** réorganise les sorties d'un squelette afin de les permuter, dupliquer ou ignorer avant de les injecter en entrée du squelette suivant;
- **Mux** (resp. **Demux**) est un squelette comprenant une entrée de contrôle i, n entrées de valeur (resp. une entrée de valeur) et une sortie (resp. n sorties). Sa sortie correspond à la $i^{\text{ème}}$ entrée (resp. la donnée d'entrée est envoyée à la $i^{\text{ème}}$ sortie);
- **Dispatch** (resp. **Merge**), de manière asynchrone et non-déterministe, envoie son entrée sur l'une de ses n sorties (resp. envoie une de ses n entrées sur sa sortie) et renvoie également l'indice sélectionné sur un canal spécifique;
- **Join** (resp. **Split**), à partir de n valeurs d'entrée, génère une sortie formée du produit de ces entrées (resp. à partir d'une entrée qui est le produit de n valeurs, génère sur n sorties chaque composante du produit d'entrée).

Chacun de ces squelettes est exprimable sous la forme d'un constructeur Caml comme proposé dans le listing 1.

Listing 1: Constructions du langage de squelettes

```
type skel =
| Seq of ('a -> 'b)
| Pipe of skel * skel
| Par of skel * skel
| Farm of int * skel
| Map of int * ('a -> 'b list) * ('b -> 'c) * ('c list -> 'd)
| ToStream of ('a -> 'b list)
| FromStream of ('b list -> a option)
| Rearrange of int * int list
| Mux of int | Demux of int | Join of int | Split of int
| Dispatch of int | Merge of int
```

Par souci de concision, on utilisera la notion infixe | pour **Pipe** et & pour **Par**. De même, la construction $S\&S\&\ldots\&S$ qui consiste à construire un schéma de type **Par** en répliquant n fois un squelette S sera noté **Par n S**.

1.2 Exemples

1.2.1 Squelettes composites

Codage des fermes. Les fermes ordonnées ou non peuvent se décomposer en opérations plus simples.

Pour une ferme ordonnée, on se ramène au cas d'un squelette à une seule entrée et une seule sortie en utilisant les opérateurs **Split** et **Join**. Ceci est possible si le squelette est "synchrone" à savoir qu'il attend des entrées de même taille sur tous les canaux et émet également des données sur l'ensemble des canaux de sortie.

On utilise l'opérateur **Par** pour créer n instances du processus correspondant au squelette S, c'est-à-dire la ferme de calcul. L'opérateur **Dispatch** envoie chaque entrée sur un des processus de la ferme (la politique

de choix n'est pas spécifiée), on transmet également l'indice du processus sélectionné. En sortie de la ferme, le processus **Mux** reçoit l'indice du processus qui a calculé la prochaine valeur et transmet cette valeur en sortie.

```
OrderedFarm n S = Join ni | Dispatch
                | (Seq id & (Par n (Split ni | S | Join no))
                | Mux n | Split no
```

Listing 2: Codage d'une ferme ordonnée

Dans le cas de la ferme non ordonnée, il n'est pas nécessaire de garder trace de l'indice du processus auquel on envoie les calculs. En sortie on se contente de l'opérateur **Merge** qui fait un choix non déterministe d'une entrée à transmettre. On utile **Rearrange** pour éliminer les sorties d'indice des processus **Dispatch** et **Merge**.

```
UnorderedFarm n S = Join ni | Dispatch | Rearrange (n+1) [2;...;n+1]
                  | Par n (Split ni | S | Join no)
                  | Merge n | Rearrange (n+1) [2;...;n+1] | Split no
```

Listing 3: Codage d'une ferme non-ordonnée

Codage de Map. De même la construction **Map**, pour un entier n donné, est codable à partir des opérations **Split**, **Join** et **Par**. On utilise des fonctions adhoc `l2pn` qui transforment une liste en produit n-aire et la fonction inverse `p2ln`.

```
Map_n s c m = Seq (fun x -> l2pn (s x))
            | Split n | Par n (Seq c) | Join n
            | Seq (fun x -> m (p2ln x))
```

Listing 4: Codage de **Map**

1.2.2 Applications

Détection de point d'intérêt de Harris et Stephen
La détection de points d'intérêt est, au même titre que la détection de contours, une étape préliminaire à de nombreux processus de vision par ordinateur. Les points d'intérêt, dans une image, correspondent à des discontinuités dans les deux directions de l'intensité des pixels. Ce sont par exemple les coins ou les points de fortes variations de texture. L'algorithme de Harris et Stephen[12] effectue la détection de tels points via le calcul d'une mesure dite de coarsité qui approxime la décomposition en valeurs singulière des gradients de l'image par des opérations de filtrages et de produits[17]. La figure 1 présente le schéma bloc de l'algorithme.

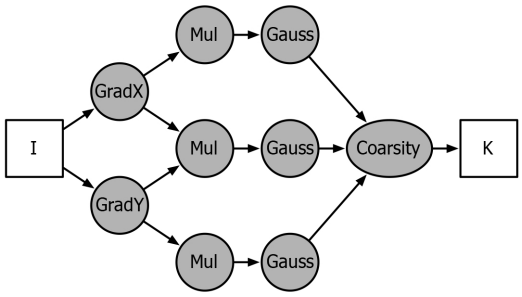

Figure 1: Graphe de dépendance de `harris`

Une implantation de cet algorithme en SKEL consiste à utiliser **Rearrange** et **Pipe** pour reproduire le graphe de dépendance initial

```
harris nb_image = ToStream(read_image nb_image) | Rearrange 1 [1;1]
                | (Seq gradX & Seq gradY)       | Rearrange 2 [1;1;1;2;2;2]
                | Par 3 multiply | Par 3 Gauss
                | Join 3         | Seq coarsity
                | FromStream(save_image nb_image)
```

Listing 5: L'algorithme `harris`

L'avantage de **Rearrange** dans cette situation et de permettre d'exprimer de manière indépendante des PIDs des processus des schémas de communications "point à point".

Décompte d'objets dans un flux vidéo. Application classique en vidéo-surveillance, le décompte d'objets en avant-plan fait appel à des algorithmes de type étiquetage en composantes connexes pour effectuer une séparation entre un arrière plan statique et des éléments mobiles sur l'avant plan après une détection des éléments mobiles de l'image[14]. Il suffit d'accumuler le nombre d'éléments détectés pour obtenir le nombre moyen d'éléments dans la séquence. Néanmoins, le temps d'exécution de l'algorithme d'étiquetage dépendant des données, il est intéressant d'exécuter cette étape au sein d'un système d'équilibrage de charge comme illustré dans le listing 6.

```
count_entity nb_image nproc = ToStream(read_image nb_image)
                            | Seq detect_motion
                            | UnorderedFarm nproc (Seq labelize)
                            | FromStream(accumulate 0 nb_image)
```

Listing 6: L'algorithme `entity_count`

Cet exemple montre un cas d'utilisation réaliste de **UnorderedFarm** basé sur les propriétés des squelettes qui la suivent. En effet, l'opération d'accumulation étant insensible à l'ordre d'arrivée de ses entrées (compter trois entités puis deux ou l'inverse donnera toujours un décompte final de cinq entités), l'étape d'équilibrage peut être effectuée sans se soucier de l'ordre de sortie. Une implantation possible de cet algorithme de décompte peut alors se faire via une ferme non ordonnée.

Divide & Conquer. La stratégie *Divide and Conquer* est un idiome classique permettant d'implanter des algorithmes récursifs. Une approximation de cette technique consiste à répéter récursivement un motif constitué d'une **OrderedFarm** encapsulée dans une paire **To/FromStream** transformant les entrées et sorties en sous-flux. Le motif de base est donné dans le listing 7 :

```
divide_conquer d c S nproc  = ToStream(d) | OrderedFarm nproc S | FromStream(c)
```

Listing 7: L'algorithme `divide_conquer`

Pour construire une structure *Divide and Conquer* multi-niveaux, il suffit de s'appuyer sur l'aspect "fonction d'ordre supérieur" des squelettes parallèles et d'appeler ce motif récursivement. Par exemple, le listing 8 présente un schéma *Divide and Conquer* à trois niveaux.

```
dc3 d c S nproc = divide_conquer d c
                   (divide_conquer  d c
                     (divide_conquer d c S nproc)
                     nproc
                   )
                   nproc
```

Listing 8: L'algorithme `divide_conquer`

L'intérêt de cette représentation est de pouvoir inférer la correction d'un *Divide and Conquer* à N niveaux à partir de la correction du motif de base.

2 Sémantique relationnelle

Nous proposons une sémantique pour le langage de squelette Skel et sa formalisation en Coq. Cette sémantique peut être utilisée pour montrer de manière compositionnelle des propriétés de programmes.

2.1 Principe de la sémantique

Chaque programme a une "arité" qui est connue statiquement. Cette arité précise le nombre d'entrées et le nombre de sorties du programme et pour chaque entrée et sortie, son type. Nous donnons dans la figure 2 la correspondance entre les constructions et le type des entrées et des sorties. On notera une arité $(\alpha_1, \ldots, \alpha_n) \to (\beta_1, \ldots, \beta_p)$ avec α_i et β_j des types ou encore $\sigma \to \tau$ avec σ et τ représentant des tuples de types. Si on a deux tuples $\sigma = (\alpha_1, \ldots, \alpha_n)$ et $\tau = (\beta_1, \ldots, \beta_p)$, on notera σ, τ le tuple $(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_p)$ de taille $n + p$. Si α est un type, on note α^n le tuple $(\alpha, \ldots, \alpha)$ de taille n dont toutes les composantes sont de type α.

On remarquera que certaines constructions comme **Pipe** demandent une condition supplémentaire pour être correctement typées, on suppose dans la suite que tous nos programmes vérifient ces conditions. Les

	arité	conditions
Seq(f)	$(\alpha) \to (\beta)$	$f : \alpha \to \beta$
Pipe(s,t)	$\sigma \to \tau$	$s : \sigma \to \rho, t : \rho \to \tau$
Par(s,t)	$\sigma, \rho \to \tau, \xi$	$s : \sigma \to \tau, t : \rho \to \xi$
Farm(n,s)	$\sigma \to \rho$	$s : \sigma \to \rho$
Map(n,s,c,m)	$(\alpha) \to (\delta)$	$s : \alpha \to \beta$ `list`$, c : \beta \to \gamma, m : \gamma$ `list` $\to \delta$
ToStream(f)	$(\alpha) \to (\beta)$	$f : \alpha \to \beta$ `list`
FromStream(f)	$(\beta) \to (\alpha)$	$f : \beta$ `list` $\to \alpha$ **option**
Rearrange$(n, [i_1; \ldots; i_p])$	$(\alpha_1, \ldots, \alpha_n) \to (\alpha_{i_1}, \ldots, \alpha_{i_p})$	$1 \leq i_k \leq n$
Mux(n)	$(\textbf{int}), \alpha^n \to (\alpha)$	
Demux(n)	$(\textbf{int}, \alpha) \to \alpha^n$	
Merge(n)	$\alpha^n \to (\textbf{int}, \alpha)$	
Dispatch(n)	$(\alpha) \to (\textbf{int}), \alpha^n$	
Join(n)	$(\alpha_1, \ldots, \alpha_n) \to (\alpha_1 \times \ldots \times \alpha_n)$	
Split(n)	$(\alpha_1 \times \ldots \times \alpha_n) \to (\alpha_1, \ldots, \alpha_n)$	

Figure 2: Arité des constructions

constructions **Rearrange**, **Mux**, **Demux**, **Join** et **Split** se contentent de faire transiter des données et sont donc polymorphes dans le type des données manipulées.

Le programme est interprété comme une relation entre les flux d'entrées et les flux de sorties. Chaque flux est lui-même représenté par une liste de données du bon type.

2.2 Représentation en Coq

Le code Coq correspondant à cette section peut être consulté à l'adresse `http://www.lri.fr/~paulin/Skel`.

Nous représentons les arités par des listes de types. Étant donnée une arité $\sigma = [A_1; \ldots; A_n]$, on définit une nouvelle arité (notée $\texttt{FL}\,\sigma$) correspondant à remplacer chaque entrée de type A par une entrée de type $\texttt{list}\,A$, on a donc $\texttt{FL}\,\sigma = [\texttt{list}\,A_1; \ldots; \texttt{list}\,A_n]$. On définit également le type $\texttt{fam}\,\sigma$ des entrées d'arité σ comme étant le produit $A_1 \times \ldots \times A_n$. Ce type se définit de manière récursive sur l'arité, dans les cas de base on pose $\texttt{fam}\,[] = \texttt{unit}$ et $\texttt{fam}\,[A] = A$.

Si on se donne deux arités σ et τ, alors on peut introduire le type $\texttt{process}\,\sigma\,\tau$ des process d'entrées σ et de sorties τ comme étant le type des relations $\texttt{fam}\,(\texttt{FL}\,\sigma) \to \texttt{fam}\,(\texttt{FL}\,\tau) \to \textbf{Prop}$.

On peut ensuite définir la sémantique de chaque construction du langage Skel. Nous les donnons sous forme de règles d'inférence dans la figure 3. On utilise la notation $(l_1, \ldots, l_n)\texttt{S}(m_1, \ldots, m_p)$ pour représenter la relation associée au squelette `S` qui a n entrées et p sorties. Les sémantiques de **Demux**, **Dispatch**, **Mux** et **Merge** se définissent toutes à l'aide d'une relation ternaire $\mathcal{D}_n$ qui prend en argument une liste de données de type α, une liste d'entiers (entre 1 et n), et une famille de n listes de type α. Nous avons utilisé cette formalisation pour faire quelques preuves élémentaires sur les squelettes. Par exemple que la composition parallèle de deux processus séquentiels correspond à la séquence de la composition des fonctions ou que composer une opération **Split** avec une opération **Join** revient au processus identité.

3 Sémantique opérationnelle

Nous proposons une sémantique opérationnelle qui s'appuie sur le join-calculus [8] et en donnons une implantation en JoCaml [15].

3.1 Join-calculus

Le Join-Calculus repose sur le modèle de la CHemical Abstract Machine (CHAM) [3]. On y conçoit le calcul comme un multi-ensemble de "molécules" qui peuvent se combiner et réagir, réaction qui pourra produire de nouvelles molécules. Les molécules peuvent s'échanger des messages.

Dans Fournet et Gonthier [8], les valeurs du join-calculus sont seulement des noms (de molécules). Un processus P est soit l'émission asynchrone d'un n-uplet de noms (une molécule qui en transporte d'autres), soit une définition D de nouveaux noms (de nouvelles molécules et réactions), soit la composition parallèle de

$$\frac{}{([])\mathbf{Seq}(f)([])} \qquad \frac{(l)\mathbf{Seq}(f)(m)}{(a :: l)\mathbf{Seq}(f)((f\,a) :: m)}$$

$$\frac{(l_1, \ldots, l_n)F(m_1, \ldots, m_k) \quad (m_1, \ldots, m_k)G(n_1, \ldots, n_p)}{(l_1, \ldots, l_n)(F|G)(n_1, \ldots, n_p)}$$

$$\frac{(l_1, \ldots, l_n)F(m_1, \ldots, m_k) \quad (l_{n+1}, \ldots, l_p)G(m_{k+1}, \ldots, m_q)}{(l_1, \ldots, l_p)(F\&G)(m_1, \ldots, m_q)}$$

$$\frac{}{\mathcal{D}_n([], [], ([], \ldots, []))} \qquad \frac{\mathcal{D}_n(lx, li, (ly_1, \ldots, ly_n))}{\mathcal{D}_n(x :: lx, i :: li, (ly_1, \ldots, x :: ly_i, \ldots, ly_n))}$$

$$\frac{\mathcal{D}_n(lx, li, (ly_1, \ldots, ly_n))}{(li, lx)\mathbf{Demux}_n(ly_1, \ldots, ly_n)} \qquad \frac{\mathcal{D}_n(lx, li, (ly_1, \ldots, ly_n))}{(lx)\mathbf{Dispatch}_n(li, ly_1, \ldots, ly_n)}$$

$$\frac{\mathcal{D}_n(lx, li, (ly_1, \ldots, ly_n))}{(ly_1, \ldots, ly_n)\mathbf{Merge}_n(li, lx)} \qquad \frac{\mathcal{D}_n(lx, li, (ly_1, \ldots, ly_n))}{(li, ly_1, \ldots, ly_n)\mathbf{Mux}_n(lx)}$$

$$\frac{}{([])\mathbf{Split}_n([], \ldots, [])} \qquad \frac{(lx)\mathbf{Split}_n(ly_1, , \ldots, ly_n)}{((x_1, \ldots, x_n) :: lx)\mathbf{Split}_n(x_1 :: ly_1, , \ldots, x_n :: ly_n)} \qquad \frac{(lx)\mathbf{Split}_n(ly_1, \ldots, ly_n)}{(ly_1, \ldots, ly_n)\mathbf{Join}_n(lx)}$$

Figure 3: Sémantique relationnelle de Skel

plusieurs processus. Une définition consiste en une ou plusieurs définitions élémentaires, chacune associant la réception d'un ou plusieurs noms à l'exécution d'un processus P.

Par exemple :

$$\texttt{def}\ x\langle u\rangle\ =\ y\langle u\rangle\ \texttt{in}\ P$$

est une définition qui introduit le nouveau nom x, le nom y doit lui exister au préalable, la variable u est liée dans la définition. En présence d'une molécule x qui transporte une valeur t, une réaction se produit qui émet une instance de y qui transportera la même valeur t.

Un autre exemple est (on utilise la syntaxe de JoCaml pour la mise en parallèle)

$$\texttt{def}\ x\langle v\rangle\ \&\ y\langle k\rangle\ =\ k\langle v\rangle\ \texttt{in}\ P$$

Dans cette nouvelle définition, x qui transporte v et y qui transporte k réagissent quand ils sont présents simultanément, et produisent une instance de k qui transporte v. Cette définition introduit x et y comme de nouveaux noms, comme précédemment la variable k est liée et donc l'émission se fera via un canal dont le nom est passé en argument à y.

L'exemple suivant illustre le non-déterminisme :

$$\texttt{def}\ s\langle\rangle\ =\ P\ \textbf{or}\ s\langle\rangle\ =\ Q\ \texttt{in}\ s\langle\rangle$$

Ici, la définition introduit deux règles qui peuvent transformer s soit en P soit en Q, d'une manière non déterministe.

$$\texttt{def}\ loop\langle\rangle\ =\ P\ |\ loop\langle\rangle\ \texttt{in}\ loop\langle\rangle$$

Enfin, quand ici *loop* réagit, cela génère en parallèle le processus P et une nouvelle copie de *loop*, ce qui permet de produire une nouvelle copie de P à répétition.

JoCaml est une extension du langage de programmation Ocaml avec les primitives du join-calculus. Les expressions Ocaml peuvent intégrer l'exécution d'un processus via le mot clef `spawn`. Les définitions de processus de JoCaml peuvent exploiter les expressions Ocaml, les séquences, les if... then ... else, les let... in et le filtrage (pattern matching).

3.2 Principe de la traduction

Le choix du join-calculus (et de son implantation JoCaml) permet de traduire aisément les différents modes de communication qui apparaissent dans Skel (déterministes ou non, synchrones ou non, avec ou sans accumulateur).

Chaque construction de squelette peur se traduire en une fonction JoCaml. On utilise une "molécule" (en JoCaml, on parle de canal) différente pour représenter chaque lien d'entrée et de sortie. Ce canal transporte une valeur du type de l'entrée ou de la sortie concernée.

Du fait de la richesse du langage, il peut être nécessaire de "bufferiser" les communications. En effet imaginons que l'on mette en série un squelette **Demux** qui a deux sorties avec un squelette **Join** qui a deux entrée. Si l'entrée de contrôle du **Demux** est de la forme $[1;1;1;2;2;2]$ la composition des deux processus est valide d'un point de vue relationnel, mais le processus **Join** reçoit 3 données sur la première entrée avant de pouvoir émettre en sortie. La taille du buffer nécessaire pour faire le calcul n'est pas bornée.

On choisit de traduire chaque processus en introduisant des buffers qui accumulent les entrées dans une file d'attente (fifo) avant de les traiter.

On utilisera les notations suivantes pour les files d'attentes (qui peuvent s'implanter en JoCaml avec un couple de listes). La file vide est notée $[]$, soit l une file et a un objet, on note $a{+}{+}l$ la file formée de la file l suivie de l'élément a (dernier élément de la file) et on note $l{+}{+}a$ la file dont le premier élément est a suivi des éléments de la file l.

Les données en entrée du processus sont en général finies, pour terminer proprement le processus, on introduit pour chaque lien un nouveau canal qui signale la fin du calcul ce qui permet de "fermer" proprement les communications.

On appellera une *communication* de type α (noté α `com`) un couple formé d'un nom de canal pour transmettre les données de type α et un nom de canal (sans valeur) pour clore la communication.

Soit un squelette dont la signature est $(\alpha_1, \ldots, \alpha_n) \to (\beta_1, \ldots, \beta_p)$. Sa traduction en JoCaml est une fonction qui prend en argument les *communications* correspondant aux types des sorties (β_1 `com` $\times \ldots \times \beta_p$ `com`) et introduit des définitions JoCaml qui définissent les communications pour les entrées α_1 `com` $\times \ldots \times \alpha_n$ `com`) qui sont renvoyées par la fonction. Ces définitions introduisent des molécules internes, et mettent en place les réactions qui implantent le comportement du squelette, elles initialisent les molécules internes.

Les canaux internes peuvent servir à implanter un état interne du calcul, ils régulent aussi les communications. Pour chaque communication d'entrée, le programme JoCaml introduit un canal interne (que l'on notera en général `buf`i) dont la valeur est une file de type α et qui sert à bufferiser les entrées. A l'initialisation des processus, une instance de chaque molécule `buf` est présente avec une file vide.

3.2.1 Exemples de traduction

Certaines constructions ont un type qui dépend de l'entier passé en argument. Du fait du système de typage de Ocaml on ne peut pas toujours construire une fonction unique pour chaque constructeur de Skel. Par contre on a un schéma de traduction qui fonctionne de manière uniforme quelque soit l'arité du squelette.

Seq : cette construction prend en argument une fonction f de type $\alpha \to \beta$ ainsi que la communication de sortie qui est de type β et qui se présente sous la forme d'une couple (`co`, `ko`) avec `co` le canal pour les données et `ko` le canal pour finir le processus.

Le programme va créer une communication de type α qui se compose aussi de deux canaux (`inp`, `ki`) avec `inp` le canal d'entrée et `ki` le canal de fin. Il introduit également un canal interne qui sert de buffer à la communication d'entrée. Une implantation peut être donnée par le programme suivant :

```
let (seq : ('a -> 'b) -> 'b com -> 'a com) = fun f (co,ko) ->                1
   def inp(a) & buf(l) =  buf(a++l)                                          2
    or buf(l++a) =  co(f a) & buf(l)                                         3
    or buf([]) & ki() = ko()                                                 4
   in spawn (buf([])); (inp,ki)                                              5
```

L'entrée `inp` (ligne 2) qui arrive est mise dans le buffer. La première entrée a du buffer est transformée par la fonction f et transmise en sortie dans le canal `co` (ligne 3). Le signal de fin `ki` reçu n'est retransmis que lorsque l'ensemble des données d'entrée a été traité (ligne 4). L'initialisation (ligne 5) met en place un buffer vide et la communication d'entrée formée du canal pour les données `inp` et du canal de fin `ki` qui sont définis dans ce processus sont renvoyés en résultat.

Pipe et Par : ces constructions sont de simples compositions de processus qui n'introduisent pas de nouvelles définitions JoCaml. La construction **Pipe** correspond à la composition de fonctions. La construction **Par** dépend des arités des processus passés en argument. On donne ici le cas de la réplication d'un processus ayant seulement une entrée et une sortie :

```
let pipe (s1,s2) = fun lc -> s1 (s2 lc)

let (par_3 : ('b com->'a com) -> ('b com*'b com*'b com -> 'a com*'a com*'a com))
  = fun s (co1,co2,co3) ->
    let ci1 = s co1 and ci2 = s co2 and ci3 = s co3 in (ci1,ci2,ci3)
```

ToStream, FromStream : la construction **ToStream** ressemble à **Seq** mais utilise en plus un état interne `state` qui lui sert à stocker le morceau de liste produit par la fonction passée en argument qui n'a pas encore été émise :

```
let (toStream : ('a -> 'b list) -> 'b com -> 'a com) = fun f (co,ko) ->
   def buf(l) & inp(a) =  buf(a++l)
    or buf(l++a) & state ([]) =  state(f a) & buf(l)
    or state(b::m) = co(b) & state(m)
    or buf([]) & ki() & state([]) = ko()
   in spawn (buf([]) & state([])); (inp,ki)
```

Pour la construction **FromStream**, c'est l'entrée qui est stockée jusqu'à ce que la fonction de traitement produise un résultat de la forme `Some` b, auquel cas la valeur b est émise.

```
let (fromStream : ('a list -> 'b option) -> 'b com -> 'a com) = fun f (co,ko) ->
    def buf(l) & inp(a) =  buf(a++l)
     or buf(l++a) & state(li) = buf(l) &
              match f (a::li) with None ->  state(a::li)
                                  | Some b -> co(b) & state([])
     or buf([]) & ki() = ko()
    in spawn (buf([]) & state([])); (inp,ki)
```

Rearrange : L'exemple illustré correspond à **Rearrange** $3\,[2;1;1;2]$, c'est-à-dire qu'il y a 3 entrées et 4 sorties. La première entrée va sur les sorties 2 et 3, la deuxième entrée va sur les sorties 1 et 4 et la troisième entrée est simplement ignorée. La fonction JoCaml correspondante prend en argument quatre communications : les première et quatrième (resp. seconde et troisième) ont le même type β (resp α). La fonction renvoie trois communications de type α, β et γ. Il y a trois canaux internes qui stockent les communications d'entrée.

```
let (rearrange_3_2112 : ('b com*'a com*'a com*'b com) -> ('a com*'b com*'c com))
fun ((o1,ko1),(o2,ko2),(o3,ko3),(o4,ko4)) ->
  def buf1(f1) & inp1(a1) = buf1(a1++f1)
   or buf2(f2) & inp2(a2) = buf2(a2++f2)
   or buf3(f3) & inp3(a3) = buf3(a3++f3)
   or buf1(l++a) = buf1(l) & o2(a) & o3(a)
   or buf2(l++a) = buf2(l) & o1(a) & o4(a)
   or buf3(l++a) = buf3(l)
   or buf1([]) & buf2([]) & buf3([]) & ki1() & ki2() & ki3()
      = ko1() & ko2() & ko3() & ko4()
  in spawn (buf1([]) & buf2([]) & buf3([]));
  (inp1,ki1),(inp2,ki2),(inp3,ki3)
```

Les entrées sont bufferisées à leur arrivée, ce qui est reçu sur la première entrée (resp. 2ème entrée) est renvoyé sur les sorties 2 et 3 (resp. 1 et 4). Ce qui est reçu sur la troisième entrée est juste ignoré (on pourrait éviter l'étape de bufferisation sur cette entrée).

Autres constructions. on donne ici les implantations **Mux**, **Demux**, **Merge**, **Dispatch**, **Split** et **Join** dans le cas de 3 communications à traiter en entrée ou sortie.

```
let (mux_3 : 'a com -> 'int com * 'a com * 'a com * 'a com) = fun (co,ko) ->
  def buf1(f1) & inp1(a1) = buf1(a1++f1)
   or buf2(f2) & inp2(a2) = buf2(a2++f2)
   or buf3(f3) & inp3(a3) = buf3(a3++f3)
   or bufi(fi) & inpi(ai) = bufi(ai++fi)
   or bufi(li++1) & buf1(l++a) = buf1(l) & bufi(li) & co(a)
   or bufi(li++2) & buf2(l++a) = buf2(l) & bufi(li) & co(a)
   or bufi(li++3) & buf3(l++a) = buf3(l) & bufi(li) & co(a)
   or bufi([]) & buf1([]) & buf2([]) & buf3([])
      & ki() & ki1() & ki2() & ki3() = ko()
  in spawn (buf1([]) & buf2([]) & buf3([]) & bufi([]));
     ((inpi,ki),(inp1,ki1),(inp2,ki2),(inp3,ki3))
```

```
let (demux_3 : ('a com * 'a com * 'a com ) -> 'int com * 'a com) =
  fun ((o1,ko1),(o2,ko2),(o3,ko3)) ->
  def bufi(fi) & inpi(ai) = bufi(ai++fi)
   or buf(f) & inp(a) = buf(a++f)
   or bufi(li++1) & buf(l++a) = buf(l) & bufi(li) & o1(a)
   or bufi(li++2) & buf(l++a) = buf(l) & bufi(li) & o2(a)
   or bufi(li++3) & buf(l++a) = buf(l) & bufi(li) & o2(3)
   or bufi([]) & buf([]) & kic() & ki() = ko1() & ko2() & ko3()
  in spawn (buf([]) & bufi([]));((inpi,kic),(inp,ki))

let (merge_3 : 'int com * 'a com -> 'a com * 'a com * 'a com) =
  fun ((oc,koc),(co,ko)) ->
  def buf1(f1) & inp1(a1) = buf1(a1++f1)
   or buf2(f2) & inp2(a2) = buf2(a2++f2)
   or buf3(f3) & inp3(a3) = buf3(a3++f3)
   or buf1(l++a) = buf1(l) & oc(1) & co(a)
   or buf2(l++a) = buf2(l) & oc(2) & co(a)
   or buf3(l++a) = buf3(l) & oc(3) & co(a)
   or buf1([]) & buf2([]) & buf3([]) & ki1() & ki2() & ki3() = ko() & koc()
  in spawn (buf1([]) & buf2([]) & buf3([]));((inp1,ki1),(inp2,ki2),(inp3,ki3))

let (dispatch_3 : ('int com * 'a com * 'a com * 'a com ) -> 'a com) =
   fun ((oc,koc),(o1,ko1),(o2,ko2),(o3,ko3)) ->
   def buf(f) & inp(a) = buf(a++f)
    or buf(l++a) = buf(l) & o1(a) & oc(1)
    or buf(l++a) = buf(l) & o2(a) & oc(2)
    or buf(l++a) = buf(l) & o3(a) & oc(3)
    or buf([]) & ki() = koc() & ko1() & ko2() & ko3()
   in spawn (buf([]));(inp,ki)

let (split_3 : ('a com * 'b com * 'c com) -> ('a * 'b * 'c) com)  =
   fun ((o1,ko1),(o2,ko2),(o3,ko3)) ->
   def buf(f) & inp(a) = buf(a++f)
    or buf(l++(a1,a2,a3)) = buf(l) & o1(a1) & o2(a2) & o3(a3)
    or buf([]) & ki() = ko1() & ko2() & ko3()
   in spawn (buf([]));(inp,ki)

let (join_3 : ('a * 'b * 'c) com -> 'a com * 'b com * 'c com) = fun (co,ko) ->
   def buf1(f1) & inp1(a1) = buf1(a1++f1)
    or buf2(f2) & inp2(a2) = buf2(a2++f2)
    or buf3(f3) & inp3(a3) = buf3(a3++f3)
    or buf1(l1++a1) & buf2(l2++a2) & buf3(l3++a3) =
       buf1(l1) & buf2(l2) & buf3(l3) & co(a1,a2,a3)
    or buf1([]) & buf2([]) & buf3([]) & ki1() & ki2() & ki3() = ko()
   in spawn (buf1([]) & buf2([]) & buf3([]));
      ((inp1,ki1),(inp2,ki2),(inp3,ki3))
```

On a proposé ici un schéma de compilation du langage de squelettes vers JoCaml qui s'applique à l'ensemble du langage et permet de préciser finement les modes de communication.

3.3 Correction

Pour établir la correction de notre schéma de compilation, il est nécessaire de faire le lien entre la sémantique relationnelle d'un squelette S et celle du programme JoCaml correspondant.

Un programme JoCaml comporte une partie de définitions qui est statique et un *état* formé d'un multi-ensemble de "molécules". La sémantique opérationnelle de JoCaml se définit par réécriture de l'état en transformant un sous-ensemble de molécules par l'application d'une des règles de la définition. Nous notons $S \longrightarrow_{\text{Jo}} T$ le fait que $S = T$ soit une instance d'une définition du programme.

Nous nous intéressons aux relations entre des entrées et des sorties du programme, nous allons donc introduire une sémantique qui met en évidence les communications suivant le modèle que nous avons choisi. Soit un squelette P dont l'arité est $(\alpha_1, \ldots, \alpha_n) \to (\beta_1, \ldots, \beta_p)$. La traduction de ce squelette est une fonction qui étant donnés p communications de sorties $(\texttt{co}_j, \texttt{ko}_j)_j$ définit n communications d'entrées $(\texttt{ci}_i, \texttt{ki}_i)_i$.

On introduit alors la relation de transition suivante entre les états du programme formés d'un multi-ensemble de molécules. On suppose que le programme JoCaml évolue suivant une instance $R = T$ d'une règle du programme. Le motif R peut contenir des molécules d'entrées $\{\texttt{ci}_i(a_i)\}_{i \in I}$ et le processus T des molécules de sortie $\{\texttt{co}_j(b_j)\}_{j \in J}$.

On a donc en général :

$$R \equiv R_0 \& \mathtt{ci}_{i_1}(a_{i_1}) \& \ldots \& \mathtt{ci}_{i_k}(a_{i_k})$$
$$T \equiv T_0 \& \mathtt{co}_{j_1}(b_{j_1}) \& \ldots \& \mathtt{co}_{j_l}(b_{j_l})$$

La relation de transition garde trace des entrées et des sorties.

On s'intéresse à l'ensemble I qui contient les molécules d'entrées $\mathtt{ci}_i(a_i)$ dans le motif R et l'ensemble J qui contient les molécules de sortie $\mathtt{co}_j(b_j)$ dans le processus T :

$$\frac{R_0 \& \{\mathtt{ci}_i(a_i)\}_{i\in I} \longrightarrow_{\mathrm{Jo}} T_0 \& \{\mathtt{co}_j(b_j)\}_{j\in J}}{S \& R_0 \xrightarrow{\{i\mapsto a_i\}_{i\in I}, \{j\mapsto b_j\}_{j\in J}} S \& T_0}$$

Un exécution du programme est une suite E de transitions $S_i \xrightarrow{\iota,\omega} S_{i+1}$ avec i les entrées et o les sorties. La i-ème entrée (resp. j-ème sortie) d'une exécution E est la liste des données qui sont reçues sur l'entrée i (resp. émises sur la sortie j). La définition se fait par récurrence sur l'exécution :

$$\begin{array}{llll}
\mathrm{inp}(i,\emptyset) & = [] & \mathrm{out}(j,\emptyset) & = [] \\
\mathrm{inp}(i, E+S \xrightarrow{\iota,\omega} T) & = a :: \mathrm{inp}(i,E) \quad (i \mapsto a \in \iota) & \mathrm{out}(j, E+S \xrightarrow{\iota,\omega} T) & = b :: \mathrm{out}(j,E) \quad (j \mapsto b \in \omega) \\
 & = \mathrm{inp}(i,E) \quad (i \mapsto _ \notin \iota) & & = \mathrm{out}(j,E) \quad (j \mapsto _ \notin \omega)
\end{array}$$

On peut maintenant lier la sémantique relationnelle et la sémantique opérationnelle d'un squelette.

Proposition 1 (Correction de la traduction) *Soit un squelette K d'arité $(\alpha_1,\ldots,\alpha_n) \to (\beta_1,\ldots,\beta_p)$, on note $\mathtt{Rel}(K)$ le relation associée. On note $(\overrightarrow{\mathtt{cki}} \leftarrow K(\overrightarrow{\mathtt{cko}}))$ le programme* JoCaml *associé qui définit les communications d'entrée* cki *et utilise les communications de sorties* cko *et on désigne par (K_0) l'état initial de ce programme.*

La relation $(l_1,\ldots,l_n)\mathtt{Rel}(K)(m_1\ldots,m_p)$ est vérifiée si et seulement si il existe une exécution E du programme $(\overrightarrow{\mathtt{cki}} \leftarrow K(\overrightarrow{\mathtt{cko}}))$ telle que l'état initial et l'état final sont égaux à K_0 et pour tout $i = 1\ldots n$ on a $\mathrm{inp}(i,E) = l_i$ et pour tout $j = 1\ldots p$ on a $\mathrm{out}(j,E) = m_j$.

Une autre propriété de notre traduction est la bonne gestion de la terminaison : Si on a $\mathtt{cki} = (\mathtt{ci},\mathtt{ki})$ et $\mathtt{cko} = (\mathtt{co},\mathtt{ko})$ alors l'état $K_0 \& \mathtt{ki}_1() \& \ldots \mathtt{ki}_n()$ évolue vers l'état $\mathtt{ko}_1() \& \ldots \mathtt{ko}_p()$.

La preuve se fait en regardant chaque construction.

- **Seq**(f). On se donne une communication de sortie $(\mathtt{co},\mathtt{ko})$ et le squelette définit une communication d'entrée $(\mathtt{ci},\mathtt{ki})$, l'état initial est $K_0 = \mathtt{buf}([])$. On suppose l et m sont reliés par **Seq**. Soit $l = m = []$ et alors on a une exécution de longueur 0 qui convient. Soit $l = a :: l'$ et $m = b :: m'$ avec $b = f\,a$ et l' et m' sont en relation. On a une exécution E qui réalise la relation entre l' et m' et ramène à l'état initial. On ajoute le morceau d'exécution suivant :

$$\begin{array}{rcl}
\mathtt{buf}([]) \& \mathtt{ci}(a) & \longrightarrow_{\mathrm{Jo}} & \mathtt{buf}([a]) \\
\mathtt{buf}([a]) & \longrightarrow_{\mathrm{Jo}} & \mathtt{co}(f\,a) \& \mathtt{buf}([])
\end{array}$$

  et donc $\mathtt{buf}([]) \xrightarrow{1\mapsto a,\emptyset} \mathtt{buf}([a]) \xrightarrow{\emptyset,1\mapsto f\,a} \mathtt{buf}([])$.

Dans l'autre sens on montre un résultat plus général qui prend en compte l'état du buffer. Si on a une exécution qui part de l'état $\mathtt{buf}([a_k;\ldots;a_1])$ et arrive dans l'état $\mathtt{buf}([a'_n;\ldots;a'_l])$ et dont l'entrée est $[a_n;\ldots;a_{k+1}]$ et la sortie est $[b_p;\ldots;b_1]$, ce que l'on note :

$$\mathtt{buf}([a_k;\ldots;a_1]) \xrightarrow[*]{[a_n;\ldots;a_{k+1}],[b_p;\ldots;b_1]} \mathtt{buf}([a'_n;\ldots;a'_l])$$

alors $a_i = a'_i$ pour $i = n,\ldots,l$ et $[f(a_n);\ldots;f(a_1)] = [f(a'_n);\ldots;f(a'_l);b_p;\ldots;b_1]$. On raisonne par récurrence sur la taille de l'exécution et par cas suivant la règle qui s'applique.

Un autre cas intéressant est celui de la construction **Pipe** de deux squelettes K et K'. On se donne les communications de sortie $\overrightarrow{cko}$ et on introduit $\overrightarrow{cks} \leftarrow K'(\overrightarrow{cko})$ et $\overrightarrow{cki} \leftarrow K(\overrightarrow{cks})$. Si on regarde l'exécution de K on a :

$$\frac{R_0 \subseteq K_0 \qquad R_0 \& \{\mathtt{ci}_i(a_i)\}_{i\in I} \longrightarrow_{\mathrm{Jo}} T_0 \& \{\mathtt{cs}_j(b_j)\}_{j\in J}}{K_0 \xrightarrow{\{i\mapsto a_i\}_{i\in I}, \{j\mapsto b_j\}_{j\in J}} (K_0 \setminus R_0) \& T_0}$$

L'exécution de K' donne :

$$\frac{U_0 \subseteq K'_0 \qquad U_0 \& \{\mathtt{cs}_k(b_k)\}_{k\in K} \longrightarrow_{\mathrm{Jo}} V_0 \& \{\mathtt{co}_l(c_l)\}_{l\in L}}{K'_0 \xrightarrow{\{k\mapsto b_k\}_{k\in K},\{l\mapsto b_l\}_{l\in L}} (K'_0 \setminus U_0)\& V_0}$$

Le cas simple est le cas pour lequel les ensembles J et K coïncident, on peut alors simuler l'exécution du système combiné de la manière suivante :

$$\begin{array}{rcl} K'_0 \& K_0 & \xrightarrow{\{i\mapsto a_i\}_{i\in I},\emptyset} & K'_0 \& (K_0 \setminus R_0) \& T_0 \& \{\mathtt{cs}_j(b_j)\}_{j\in J} \\ & = & (K'_0 \setminus U_0) \& U_0 \& \{\mathtt{cs}_k(b_k)\}_{k\in K} \& (K_0 \setminus R_0) \& T_0 \\ & \xrightarrow{\emptyset,\{l\mapsto b_l\}_{l\in L}} & (K'_0 \setminus U_0) \& V_0 \& (K_0 \setminus R_0) \& T_0 \\ & = & (K'_0 \setminus U_0) \& V_0 \& (K_0 \setminus R_0) \& T_0 \end{array}$$

Et on peut donc continuer à entremêler les exécutions de K et K'. Néanmoins la condition que l'ensemble des sorties de l'un en une étape est égal aux entrées de l'autre n'est pas garantie. Cependant, grâce à notre système de bufferisation, les règles d'entrées pour les processus sont toujours applicables. De plus les squelettes introduisent des règles qui concernent des molécules indépendantes sauf celles d'entrées et de sorties et donc cela permet de réordonner les exécutions pour faire fonctionner la combinaison.

4 Conclusion

Nous avons montré dans cet article qu'il était possible, à partir d'une extension du langage proposé dans [18], de définir pour un jeu de squelettes non triviaux une sémantique relationnelle et une sémantique opérationnelle qui correspondent. Par rapport aux travaux précédents, la correction de ces sémantiques est cette fois démontrable et les programmes issus du langage sont traduisibles vers JoCaml. Ces sémantiques posent donc une base pour un début de système de raisonnement sur des programmes à base de squelettes algorithmiques, permettant ainsi de pouvoir raisonner sur des programmes plus complexes et moins contraints dans leur structures que d'autres systèmes existants.

À terme, ces travaux vont permettre d'explorer plus avant de nouvelles techniques de génération de code. En effet, si la traduction JoCaml est intéressante pour le déroulement du raisonnement, on peut penser qu'une réimplantation, cette fois sur une base vérifiable, de Skel vers un langage natif de type C++ (comme le fût Quaff[6]) est envisageable en prenant en compte les éléments techniques issus de JoCaml. Une autre piste consiste à utiliser ces sémantiques afin de proposer une analyse de la complexité en temps de programmes squelettiques d'une manière similaire à celle des programmes BSP.

References

[1] Marco Aldinucci and Marco Danelutto. An operational semantics for skeletons. In *PARCO*, pages 63–70, 2003.

[2] Marco Aldinucci and Marco Danelutto. Skeleton-based parallel programming: Functional and parallel semantics in a single shot. *Computer Languages, Systems and Structures*, pages 179–192, 2007.

[3] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theor. Comput. Sci.*, pages 217–248, 1992.

[4] Murray Cole. *Algorithmic skeletons: structured management of parallel computation.* MIT Press, Cambridge, MA, USA, 1991.

[5] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *PPOPP '93 Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12. ACM, 1993.

[6] Joel Falcou and Jocelyn Sérot. Formal semantics applied to the implementation of a skeleton-based parallel programming library. 2008.

[7] Joel Falcou and Jocelyn Sérot. Une bibliothèque métaprogrammée pour la programmation parallèle. *Technique et Science Informatiques*, pages 645–675, 2009.

[8] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *POPL, Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1995.

[9] Fr'edéric Gava. *Approches fonctionnelles de la programmation parallèle et des méta-ordinateurs. Sémantiques, implantations et certification.* PhD thesis, University Paris XII-Val de Marne, LACL, 2005.

[10] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Softw., Pract. Exper.*, pages 1135–1160, 2010.

[11] Khaled Hamidouche. *Programmation des Machines Hiérarchiques et hétérogènes.* PhD thesis, University Paris XI - ile-de-France, September 2008.

[12] C. Harris and M. Stephens. A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*, pages 147–151, 1988.

[13] Noman Javed and Frédéric Loulergue. A formal programming model of orléans skeleton library. In *PaCT*, pages 40–52, 2011.

[14] Lionel Lacassagne and Bertrand Zavidovique. Light speed labeling: Efficient connected component labeling on risc architectures. *Journal of Real Time Image Processing*, 6(2):117–135, 2011.

[15] Louis Mandel and Luc Maranget. *The JoCaml language Release 4.00 Documentation and user's manual.* INRIA. `http://jocaml.inria.fr/doc/index.html`.

[16] Michael Poldner and Herbert Kuchen. On implementing the farm skeleton. *Parallel Processing Letters*, 18(1):117–131, 2008.

[17] Tarik Saidani, Lionel Lacassagne, Joel Falcou, Claude Tadonki, and Samir Bouaziz. Transactions on high-performance embedded architectures and compilers iii. chapter Parallelization schemes for memory optimization on the cell processor: a case study on the Harris corner detector, pages 177–200. Springer-Verlag, Berlin, Heidelberg, 2011.

[18] Jocelyn Sérot, Joel Falcou, and Joel Falcou. Functional meta-programming for parallel skeletons. In *ICCS (1)*, pages 154–163, 2008.

[19] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, pages 103–111, 1990.