RenPar'10
Strasbourg, 9 juin - 12 juin 1998

Des étages en Concurrent ML

Marc Gengler et Matthieu Martel

Laboratoire d'Informatique de Marseille (LIM)
Parc Scientifique et Technologique de Luminy
163, avenue de Luminy - Case 901 F
13288 Marseille Cedex 9, France
E-mail: [Marc.Gengler,Matthieu.Martel]@esil.univ-mrs.fr

Résumé

Dans cet article, nous nous intéressons à la spécialisation de programmes communicants, relativement à une partie de leurs arguments. Pour cela, nous présentons la sémantique dynamique d'une version multi-étagée de Concurrent ML. La notion d'étage permet de stratifier l'interprétation d'un programme en fonction de ses paramètres. La connaissance d'une partie des arguments permet une évaluation partielle du programme lors de l'exécution du premier étage du programme. Un programme résiduel est construit qui sera à son tour interprété lors de l'exécution de l'étage suivant, etc... Nous montrons comment la notion d'étage permet de gérer de manière fine l'aspect statique ou dynamique des communications. La sémantique proposée décrit la façon dont celles-ci sont réduites en fonction des annotations. Nous abordons les problèmes liés à la suppression d'une partie des interactions entre processus.

Mots-clés : Langages fonctionnels parallèles, Communications, Etages, Evaluation Partielle.

---

1. Introduction

La notion d'*étages* (aussi appelés *niveaux*) est une généralisation des systèmes d'annotations utiles à l'évaluation partielle de programmes [3]. Elle consiste à considérer un morceau de code *dynamique*, c.à.d. une expression ne pouvant être réduite à la compilation car elle dépend de paramètres inconnus, comme une expression du premier étage. Les expressions *statiques* (dont les variables sont connues à la compilation) sont des expressions du niveau zéro [8]. Un *évaluateur partiel* est un programme qui calcule statiquement les expressions de niveau zéro d'un code source et qui génère un programme résiduel spécialisé, composé des expressions du niveau supérieur.

Des études récentes concernant les techniques d'évaluation partielle de programmes parallèles ont donné de premiers résultats encourageants, tant pour les systèmes à mémoire partagée [2] que pour ceux à mémoire distribuée [4, 5, 6]. Dans ce dernier cas, l'évaluation partielle permet en outre de réduire le nombre de communications et de synchronisations réalisées par un programme, tout en garantissant la correction des programmes spécialisés.

Dans un langage *multi-étagé* (*multi-stage language*), les spécialisations faites par un évaluateur partiel peuvent être répétées plusieurs fois. Un programme est décomposé en niveaux qui sont interprétés successivement, donnant à chaque étape un programme plus spécialisé dans lequel des paramètres ont été fixés. Une expression de niveau $n+1$ représente au niveau n un calcul gelé (opérateur `quote` du LISP). META ML, une version multi-étagée du langage ML, a récemment été décrite [11] et implémentée [7].

Dans ce papier nous nous intéressons à l'intégration des étages dans CONCURRENT ML [1, 9], une extension de ML offrant des primitives pour les communications. Nous montrons que l'utilisation des étages permet d'annoter de manière précise des processus séquentiels communicants dans le but de leur optimisation par évaluation partielle.

Jusqu'à présent, les travaux relatifs à l'évaluation partielle de programmes communicants se sont principalement concentrés soit sur des langages pour lesquels les processus et les canaux de communication sont crées statiquement [6] soit sur des noyaux de langages plus dynamiques [4, 5]. Le langage présenté ici est un langage complet permettant la création dynamique de nouveaux processus et canaux.

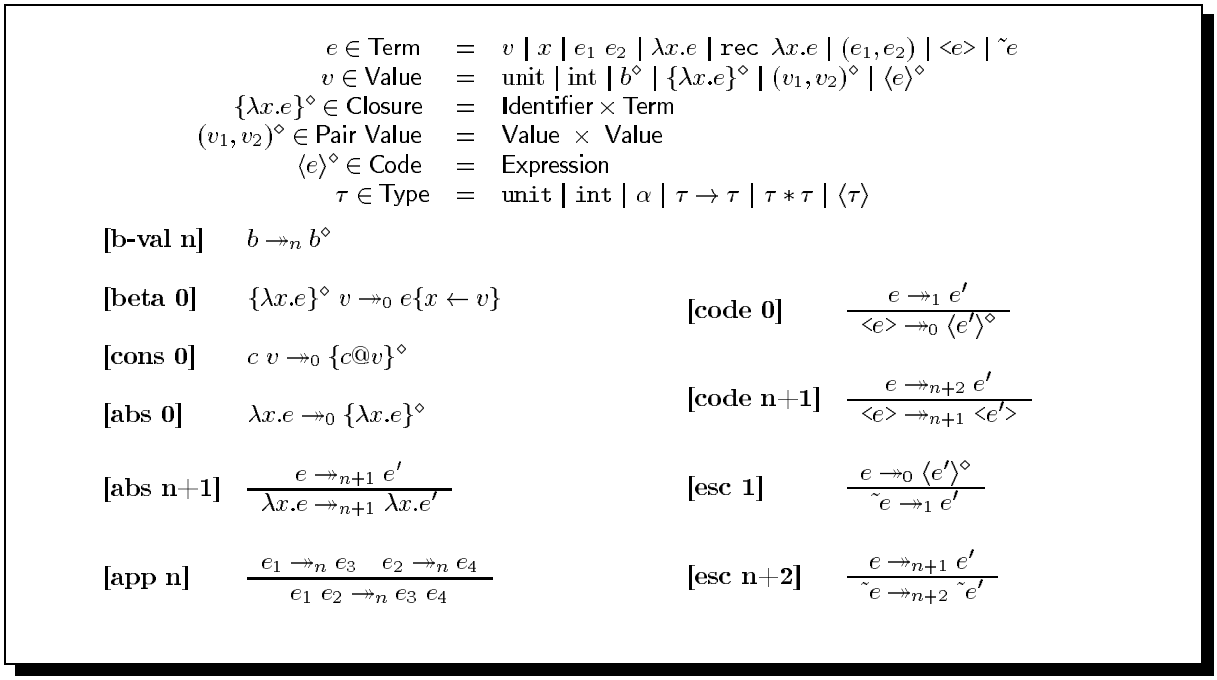

FIG. 1: *Syntaxe et sémantique dynamique du langage séquentiel.*

2. Notion d'étage

Un programme du second ordre manipule les fonctions comme des objets de base du langage (au même titre que les entiers par exemple). Similairement, un programme à deux étages manipule les *représentations* des expressions comme des objets de base. Un programme à trois étages manipule les représentations des représentations des expressions comme des objets de base, etc... Nous utilisons la syntaxe <*e*> pour décrire la représentation d'une expression e. Par exemple, <17 + 1> représente le code de l'expression arithmétique $17+1$. Au niveau zéro, <17 + 1> est une *valeur*, ce qui implique qu'aucune réduction ne sera effectuée lors de la première étape d'exécution du programme.

Les règles de typage d'un tel langage sont présentées dans [10]. Un nouveau constructeur de type $\langle . \rangle$ est introduit pour la gestion des expressions gelées. Ainsi, le type de <17 + 1> est $\langle \mathtt{int} \rangle$, indiquant que la valeur est la représentation d'une expression de type entier. $\langle . \rangle$ étant un constructeur, au même titre par exemple que $\to$ pour les fonctions, il est possible de le composer avec tout autre constructeur du système. Ainsi, <$\lambda x.x + 1$> a pour type $\langle \mathtt{int} \to \mathtt{int} \rangle$.

L'opérateur ~ (équivalent à l'opérateur `backquote` de LISP) permet de composer des représentations en autorisant le *dégel* d'une sous-expression à l'intérieur d'un morceau de code. Considérons par exemple la fonction `p` qui prend pour argument une représentation <*e*> de type $\langle \alpha \rangle$ et qui produit la représentation de la paire (e, e). Le programme `p` $\equiv \lambda x.$<(x, x)> est incorrect (et sera rejeté par le vérificateur de types) car la variable x n'étant pas dégelée, est considérée comme une expression de niveau un. Au lieu de cela, la fonction `p'` $\equiv \lambda x.$<$(\tilde{}x, \tilde{}x)$>, de type $\forall \alpha, \langle \alpha \rangle \to \langle \alpha * \alpha \rangle$ doit être utilisée. Lorsque `p'` est appliquée, le corps de la fonction est parcouru. $\tilde{}x$ indique que x est une sous-expression de niveau inférieur. La variable est donc interprétée, c.à.d. est remplacée par le morceau de code correspondant à sa valeur.

La figure 1 décrit la syntaxe de la partie séquentielle du langage. Il s'agit du noyau d'un langage fonctionnel contenant des constantes et des variables, un opérateur de points fixes, un constructeur de paires et les opérateurs permettant de manipuler les représentations d'expressions. Les expressions sont supposées sans variables libres. La fonction de réduction dépend de l'étage n qui est interprété. Par exemple, la β-réduction s'effectue uniquement au niveau zéro et dans les étages supérieurs le corps d'une abstraction est parcouru afin de réduire d'éventuels sous-termes dégelés. L'occurrence de chevrons incrémente le niveau de réduction. Le corps de l'expression est réduit au niveau supérieur afin de gérer les dégels. [10] décrit comment la substitution s'effectue dans des termes à plusieurs niveaux.

$$
\begin{array}{rcl}
b' \in \mathsf{Basic\ PValue} & = & e \mid k \mid c \mid \{c@v\}^\diamond \mid \texttt{fork} \mid \texttt{sync} \\
c \in \mathsf{Constructor} & = & \texttt{send} \mid \texttt{receive} \\
\{c@v\}^\diamond \in \mathsf{Constructed\ Value} & = & \mathsf{Constructor} \times \mathsf{PValue} \\
\tau' \in \mathsf{PType} & = & \tau \mid \tau\ \texttt{channel} \mid \tau\ \texttt{com}
\end{array}
$$

[seq n] $$\frac{e \twoheadrightarrow_n e'}{K, P[p\ :\ e] \rightrightarrows_n^{\{p\}} K, P[p\ :\ e']}$$

[chan 0] $$\frac{k \notin K}{K, P[p\ :\ \texttt{channel()}] \rightrightarrows_0^{\{p\}} K \cup \{k\}, P[p\ :\ k]}$$

[fork 0] $$\frac{q \notin dom(P) \cup \{p\}}{K, P[p\ :\ \texttt{fork}\ \lambda x.e] \rightrightarrows_0^{\{p,q\}} K, P[p\ :\ \text{unit}][q\ :\ e]}$$

[fork n+1] $$\frac{K, P[p\ :\ e] \rightrightarrows_{n+1}^{S} K', P'[p\ :\ e']}{K, P[p\ :\ \texttt{fork}\ \lambda x.e] \rightrightarrows_{n+1}^{S} K', P'[p\ :\ \texttt{fork}\ \lambda x.e']}$$

[sync 0] $$\frac{com_1, com_2 \hookrightarrow e_1, e_2}{K, P[p\ :\ \texttt{sync}(com_1)][q\ :\ \texttt{sync}(com_2)] \rightrightarrows_0^{\{p,q\}} K, P[p\ :\ e_1][q\ :\ e_2]}$$

[com] $$\langle \texttt{send}, (k, v)\rangle^\diamond, \langle \texttt{receive}, k\rangle^\diamond \hookrightarrow v, v$$

[swap] $$\frac{com_2, com_1 \hookrightarrow e_2, e_1}{com_1, com_2 \hookrightarrow e_1, e_2}$$

FIG. 2: *Syntaxe et sémantique dynamique de la partie parallèle du langage.*

3. Etages et parallélisme

Outre la réduction des parties séquentielles, la spécialisation d'un programme concurrent consiste à effectuer lors de l'exécution des étages inférieurs les communications connues et les créations prévisibles de processus et canaux. Ces opérations étant en général coûteuses, des gains de performances sensibles peuvent être obtenus, comme cela a déjà été mis en évidence pour des langages pour machines à mémoire partagée [3]. Dans cette partie, nous définissons les règles de spécialisation de programmes écrits en CONCURRENT ML au travers de la sémantique dynamique d'une version multi-étagée de ce langage. Afin de réaliser au mieux ces optimisations, il est nécessaire de disposer d'un système précis d'annotations. Nous montrons comment les étages sont utilisés pour décrire les aspects statiques et dynamiques des opérations gérant le parallélisme.

La figure 2 décrit la syntaxe et la sémantique opérationnelle des nouvelles constructions du langage. k représente un nom de canal. `fork` permet la création dynamique de processus. Afin d'exécuter une communication (`send` ou `receive`), celle-ci doit être activée par la commande `sync`. Par exemple, dans l'expression `let` $c =$ `send`(k, v) `in sync` c, la communication ne sera pas réalisée avant l'exécution de la commande `sync`. En CONCURRENT ML, un processus est une paire dans le domaine ProcessId $\times$ Term. Une *configuration* P est une liste de processus notée $P[p_1 : e_1] \ldots [p_n : e_n]$. Les règles de réduction pour les configurations sont données dans la figure 2. Les environnements K permettent de maintenir la cohérence des noms de canaux [9]. Tout comme pour la partie séquentielle du langage, différentes règles sont appliquées selon le niveau. La réduction d'une expression séquentielle de niveau n s'effectue selon les règles de la figure 1 pour le même niveau. Lorsqu'un `fork` de niveau zéro est rencontré, le processus correspondant est créé. Lorsque celui-ci est gelé (de niveau supérieur), le corps du processus est parcouru afin d'être simplifié en fonction des dégels qu'il contient. Seules les communications dégelées et activées par la commande `sync` sont réalisées. Leur gestion est décrite par les règles de réduction **[com]** et **[swap]**.

Les étages permettent de différencier plusieurs types de communications en fonction de la disponibilité de leurs arguments. Naturellement, l'émission d'une valeur e disponible (ou connue) au niveau courant sur un canal α également disponible s'écrit `send`(k, e) et est exécutée lors de l'étape courante de réduction.

Symétriquement, l'émission d'une valeur inconnue <e> sur un canal inconnu <α> ne peut être exécutée. Dans ce cas, la communication est entièrement gelée : <`send`(α, e)>. Pour l'émission d'une valeur disponible e sur un canal indisponible <α>, l'expression à émettre est réduite au niveau courant et la communication résiduelle est gelée : <`send`($\alpha, \tilde{}e$)>. Enfin lorsque seul le canal α est connu, il est possible de *migrer* l'expression gelée <e> afin de ne pas déferrer la communication : `send`(α, <e>).

4. Limites et perspectives

La spécialisation consiste à transformer un programme non étagé, ou plus généralement un programme non annoté, en un programme étagé qui en fonction d'une partie peu variable de ses arguments d'entrée produit un nouveau programme plus performant pour les arguments restants. Cependant, dans le cadre des programmes communicants, ce processus peut parfois produire des programmes résiduels non équivalents aux programmes initiaux, comme cela est discuté dans [5].

Tout d'abord, la spécialisation peut réduire le non-déterminisme. Cela se produit lorsqu'une communication d'un programme p sur un canal α partagé par d'autres processus est réalisée au moment de la spécialisation. Dans ce cas, l'exécution de la version réduite p' de p dans un environnement $\mathcal{E}$ contenant d'autres processus pouvant communiquer sur α ne sera pas équivalente à celle de p dans le même environnement. En effet, p aurait pu interagir sur α avec un processus de $\mathcal{E}$ alors que p' a réalisé cette communication de manière interne. Le non-déterminisme a été restreint par la spécialisation et p' ne pourra adopter qu'une partie des comportements de p. Il est possible de remédier à ce problème en ne spécialisant que les communications utilisant des canaux privés.

Par ailleurs, la réduction de certaines communications peut supprimer des synchronisations et ainsi produire des programmes dont le comportement est différent de celui des programmes initiaux. Prenons par exemple le cas d'un processus communicant successivement sur des canaux α et β. En anticipant l'exécution de la première interaction, on obtient un processus spécialisé qui communique immédiatement sur β. Cette communication peut entrer en conflit avec d'autres messages utilisant le même canal alors que ceci ne pouvait pas se produire tant que l'ordre causal imposé par α était présent dans p. L'ajout de synchronisations résiduelles dans les programmes spécialisés est nécessaire pour remédier à ce problème.

Nous travaillons actuellement sur une méthode d'analyse permettant d'étager automatiquement des programmes écrits en CONCURRENT ML (analyse des temps de liaison). Cette analyse devrait permettre de détecter les différentes sortes de communications énumérées dans la section 3 tout en tenant compte des contraintes décrites ci-dessus.

Bibliographie

1. Berry (Dave), Milner (Robin) et Turner (David N.). – A semantics for ML concurrency primitives. *Principles of Programming Languages POPL'92*, 1992. – ACM.
2. Concel (Charles) et Danvy (Olivier). – Partial evaluation in parallel. *Lisp and symbolic computation*, no3, 1993, pp. 327–342.
3. Consel (Charles) et Danvy (Olivier). – Partial evaluation: Principles and perspectives. *Principles of Programming Languages POPL'93*, 1993. – ACM.
4. Gengler (Marc) et Martel (Matthieu). – Self-applicable partial evaluation for the pi-calculus. *Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, 1997, pp. 36–46. – ACM.
5. Gengler (Marc) et Martel (Matthieu). – Compiler generation for communication-based languages. *Journal of Theoretical Computer Science (à paraître)*, 1998.
6. Marinescu (Mihnea) et Goldberg (Benjamin). – Partial evaluation techniques for concurrent programs. *Partial Evaluation and semantic based program manipulations PEPM'97*, 1997, pp. 47–62. – ACM.
7. Martel (Matthieu) et Sheard (Tim). – *Introduction to multi-stage programming using* MetaML. – Rapport technique, Oregon Graduate Institute of Science and Technology, 1997.
8. Nielson (Flemming) et Nielson (Hanne Riis). – Cambridge tracts in theoretical computer science (34). *Two-level functional languages*, 1993.
9. Reppy (John H.). – *Higher-order Concurrency*. – Thèse de PhD, Department of Computer Science, Cornell University, Ithaca, NY 14853, 1992.
10. Taha (Walid) et Benaissa, Zine-El-Abidine Sheard (Tim). – Multi-stage programming: Axiomatization and type safety. *Soumis à ICALP'98*, 1998.
11. Taha (Walid) et Sheard (Tim). – Multi-stage programming with explicit annotations. *Partial Evaluation and semantic based program manipulations PEPM'97*, 1997, pp. 203–217. – ACM.