Sémantique parallèle d'un langage fonctionnel indulgent

Guy Tremblay et Bruno Malenfant

Septembre 1998

Résumé

Les langages fonctionnels indulgents — non-stricts mais non-paresseux — sont particulièrement intéressants pour la programmation parallèle implicite. Dans cet article, nous allons présenter la sémantique d'un tel langage à l'aide d'une algèbre de processus (π-calcul).

Dans un premier temps, nous présenterons ce qu'est un langage fonctionnel indulgent. À l'aide d'un exemple, nous verrons pourquoi un tel langage facilite l'écriture de programmes implicitement parallèles. Cette première partie sera suivie d'une brève introduction au π-calcul, notation introduite par R. Milner pour la description de processus concurrents et mobiles.

Une sémantique formelle d'un noyau de langage fonctionnel indulgent (`kpH`*) sera ensuite présentée. Cette sémantique sera exprimée de facon dénotationnelle, en utilisant des termes de π-calcul comme dénotations d'expressions du langage; elle permettra d'exprimer comment l'indulgence permet d'exploiter le parallélisme implicite présent dans un programme fonctionnel.*

Finalement, la dernière partie présentera un interpréteur de π-calcul réalisé en `Java` *et où les* threads *de* `Java` *sont utilisés pour obtenir un interpréteur* parallèle *reflétant le parallélisme inhérent au π-calcul. Nous y décrirons aussi quelques-uns des tests effectués pour vérifier de façon empirique la validité de notre sémantique indulgente ainsi que pour examiner le parallélisme pouvant être extrait de programmes indulgents vs. stricts.*

1 Introduction

Il existe deux grandes classes de langages fonctionnels: les langages stricts et les langages non-stricts. Bien qu'on considère souvent que les langages non-stricts sont paresseux, ce n'est pas nécessairement le cas, puisqu'il existe des langages fonctionnels qui sont non-stricts sans être paresseux; de tels langages sont dits *indulgents* [21].

Un aspect intéressant des langages indulgents, que nous examinerons plus en détail à la section 2, est leur support pour la programmation parallèle implicite.

L'objectif à long terme de la recherche décrite dans cet article est de développer un compilateur pour langages indulgents ciblé vers des machines massivement parallèles, plus précisément, des architectures dites multi-contextes (*multi-threaded computers*) [10, 7]. De telles machines sont des machines *MIMD* avec mémoire distribuée (*loosely-coupled*) où chaque noeud utilise un processeur à contextes multiples; un tel processeur, à un instant donné, peut avoir plusieurs contextes actifs et un changement de contexte peut s'effectuer de façon rapide et efficace. On estime que de telles architectures sont généralement aptes à tolérer les temps de latence mémoire variables typiques aux machines à mémoire distribuée et permettent de supporter efficacement le parallélisme de granularité fine. Or, comme on le verra à la section 4, la sémantique d'un langage indulgent s'exprime assez naturellement à l'aide d'un grand nombre de processus parallèles indépendants, plus précisément, à l'aide de *threads* de granularité fine.

Le processus de compilation des langages fonctionnels, tant pour les mises en oeuvre séquentielles que parallèles, est souvent présenté en traduisant le langage source (ou un langage intermédiaire simplifié) en un langage pour une machine abstraite [19, 4, 2]. La sémantique du programme résultant est alors obtenue par l'intermédiaire d'une sémantique opérationnelle de cette machine. De telles machines sont généralement, à la base, des machines séquentielles auxquelles sont ajoutées diverses opérations pour la création et synchronisation de tâches parallèles.

Dans cet article, nous amorçons l'exploration d'une approche différente dont les grandes lignes sont présentées à la figure 1. Le langage source est le langage fonctionnel indulgent et parallèle `pH` [15]. Dans un premier temps, l'analyse lexicale et syntaxique, et les vérifications de types sont effectuées, ainsi que diverses transformations et simplifications du programme source; une représentation intermédiaire sous forme d'arbres abstraits est ensuite produite.

C'est à partir de la forme intermédiaire que le parallélisme, implicite dans la sémantique du langage source, est identifié et rendu explicite. Pour ce faire, la forme intermédiaire est traduite en un ensemble de processus concurrents exprimés à l'aide du π-calcul. Comme on le verra, cette traduction peut en fait être considérée comme une sémantique dénotationnelle exprimée non pas à l'aide du λ-calcul mais à l'aide d'une algèbre de processus. La section 3 présente plus en détail la notation utilisée pour la description des processus (π-calcul) alors que la section 4 présente la sémantique du langage intermédiaire exprimée en termes de ce π-calcul.

Dans un processus global de compilation, diverses optimisations seraient ensuite effectuées avant de procéder à la génération de code. Plus particulièrement, une étape cruciale de *partitionnement* serait effectuée, laquelle vise à obtenir, à partir des *threads* de granularité fine, des processus de taille supérieure mais qui préservent quand même la sémantique indulgente, donc les dépendances et synchronisation. Cette étape de partitionnement peut être omise sans que la

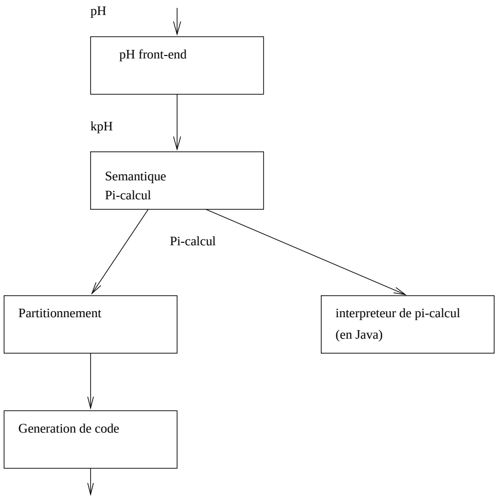

Figure 1: Grandes lignes du processus de compilation

sémantique ne soit changée; toutefois, les coûts de synchronisation et de gestion des *threads* sont nécessairement plus élevés lorsqu'aucun partitionnement n'est effectué. Finalement, la génération de code pour une machine multi-contextes serait ensuite effectuée; cette transformation, grâce au partitionnement effectué antérieurement, serait relativement facile et directe.

Faute d'espace, dans le présent article, nous mettrons surtout l'accent sur la sémantique elle-même et sur l'approche utilisée pour tester cette sémantique, à savoir, le développement d'un interpréteur parallèle de π-calcul réalisé à l'aide de classes et *threads* Java (partie droite de la figure 1); cet interpréteur sera décrit à la section 5.

2 Un langage fonctionnel indulgent et parallèle et son noyau purement fonctionnel

Le langage `pH` — `p`*arallel* `Haskell` — est une variante du langage `Haskell` [18]. Plus précisément, `pH` est une variante *parallèle* et *indulgente* de ce langage. Avant de présenter le sous-ensemble du langage que nous traiterons (section 2.2), nous rappelons tout d'abord ce qu'est un langage indulgent.

2.1 Langage fonctionnel indulgent

Un langage fonctionnel est dit *strict* lorsqu'un fonction définie par un programme ne peut pas produire un résultat lorsqu'un de ses arguments n'est pas défini. Ceci correspond, en gros, à l'utilisation de l'appel par valeur comme mécanisme de transmission des arguments, mécanisme qui assure que tous les arguments sont évalués *avant* l'appel de la fonction.

Un langage est dit *non-strict* lorsqu'il est possible qu'une fonction produise un résultat même si un (ou plusieurs) des arguments n'est pas encore évalué. Les exemples les plus connus de langages non-stricts sont les langages paresseux, langages qui transmettent les arguments des fonctions à l'aide d'une forme d'appel par nécessité (*call-by-need*): ceci assure qu'une expression utilisée comme argument effectif ne sera évaluée que si l'argument formel correspondant est effectivement utilisé. Toutefois, il est possible pour un langage d'être non-strict *sans être paresseux*; de tels langages sont dits *indulgents* (cf. [21] pour plus de détails sur la notion de non-strict vs. paresseux vs. indulgent).

Un petit exemple de programme utilisant l'évaluation non-stricte est le suivant:

```
> FacListe n =
>   let
>     l = 1 : 1 : [l!!(i-1) + l!!(i-2) | i <- [2..n]]
>   in
>     l!!n
```

Cette fonction calcule les nombres de Fibonacci, mais sans générer un nombre exponentiel d'appels récursifs comme le ferait une définition récursive naïve; les valeurs déjà calculées sont plutôt utilisées pour calculer les nouveaux résultats (programmation dynamique). L'évaluation non-stricte est requise pour ce programme car la définition de la liste `l` utilise cette même liste (dépendance cyclique).

Dans un contexte parallèle, le mécanisme d'appel de fonctions des langages indulgents est relativement simple à exprimer: le corps de la fonction *et* les expressions utilisées comme arguments sont évaluées *en parallèle*, avec des synchronisations appropriées entre l'appelant et l'appelé! C'est cette dernière caractéristique qui fait que les langages indulgents sont particulièrement intéressants pour la programmation parallèle implicite. Par exemple, soit le code `pH` suivant, dont la tâche consiste à produire, à partir d'un arbre binaire, la liste des feuilles de cet arbre:

```
> data arbre = Feuille Int | Noeud arbre arbre
>
> feuilles a = feuillesAccum a []
>
> feuillesAccum (Feuille i) feuilles
>   = i : feuilles
```

```
> feuillesAccum (Noeud gauche droit) feuilles
>   = feuillesAccum gauche (feuillesAccum droit feuilles)
```

On verra, à la section 5.2, des profils de parallélisme pour ce programme montrant le nombre d'opérations pouvant être effectuées à chaque instant: on pourra alors voir que l'évaluation stricte génère moins de parallélisme et requiert un temps plus long que l'évaluation indulgente. Intuitivement, ceci s'explique par le fait que la fonction `feuillesAccum` génère deux (2) appels récursifs visant respectivement à explorer les branches gauche et droite; toutefois, l'un de ces appels apparaît comme argument de l'autre appel. Dans le cas de l'évaluation stricte, cela signifie que le parcours du sous-arbre gauche ne peut débuter que lorsque le sous-arbre droit a été complètement exploré (évaluation des arguments *avant* l'appel); par contre, dans le cas d'une approche indulgente, les deux sous-arbres peuvent être explorés en parallèle.

2.2 Un sous-ensemble simplifié et purement fonctionnel de `pH`: `kpH`

Bien que le langage `pH` contienne un certain nombre de constructions absentes du langage `Haskell` — e.g., boucles, I/M-structures —, dans le contexte de la recherche présentée dans cet article, nous nous attarderons uniquement au sous-ensemble purement fonctionnel du langage `pH`, en omettant les constructions additionnelles. Plus précisément, nous nous restreindrons à un sous-ensemble simplifié, tel que produit par la partie frontale, et purement fonctionnel. La syntaxe abstraite de ce sous-ensemble — `kpH` pour `kernel pH` — est présentée à la figure 2.

$$
\begin{array}{lcl}
\text{Programme} & ::= & e \\
e, e_1, e_h & ::= & v \\
 & | & y \\
 & | & \oplus(e_1, \ldots, e_h) \\
 & | & \text{let } n_1 = e_1; \ldots; n_h = e_h \text{ in } e \\
 & | & \lambda x_1, \ldots, x_h . e \\
 & | & f(e_1, \ldots, e_h) \\
 & | & K_i(e_1, \ldots, e_h) \\
 & | & \text{case } y \text{ as } K_1(n_{11}, \ldots, n_{1a_1}) \text{ in } e_1; \ldots
\end{array}
$$

où

$$
\begin{array}{rcl}
v & \in & \text{Valeur de base. (Int, Float, ...)} \\
h & \in & \mathbf{N} \\
x_i, y, f, n_i & \in & \text{Identificateur} \\
K_i & \in & \text{Constructeur} \\
\oplus & \in & \text{Opérateur de base } (+, -, \ldots)
\end{array}
$$

Figure 2: Syntaxe abstraite du langage `kpH`

3 Une brève introduction au π-calcul

Afin de décrire le parallélisme présent à l'intérieur d'un programme `kpH`, nous allons utiliser un système formel permettant d'expliciter ce parallélisme. Toutefois, comme les descriptions de haut niveau traditionnelles — e.g., sémantique dénotationnelle standard, sémantique naturelle, etc. — ne sont pas vraiment appropriées pour décrire les aspects associés au parallélisme, nous allons plutôt faire appel aux algèbres de processus. Une algèbre de processus est un système formel — syntaxe, sémantique, équivalences — qui tente de modéliser l'essence d'un paradigme de programmation parallèle et concurrente [20]. Plus précisément, dans notre cas, nous allons utiliser le π-calcul.

Le π-calcul fut introduit par R. Milner [12, 13] dans le but "*de décrire et d'analyser les systèmes composés d'agents qui interagissent entre eux et dont la configuration et l'environnement sont continuellement en changement*" [12]. Chaque expression π-calcul dénote un processus dont les seules actions possibles consistent à échanger des messages sur des canaux de communication, ces messages ne pouvant eux-mêmes qu'être des références à d'autres canaux. Le π-calcul est donc une notation minimaliste qui peut être vue comme un calcul de bas niveau pour les processus concurrents, semblable à ce qu'est le λ-calcul pour les fonctions (fonction $\leftrightarrow$ processus, application $\leftrightarrow$ communication).

3.1 Syntaxe du π-calcul

Le π-calcul employé ici est *asynchrone*, c'est-à-dire que les envois de message sont exécutés immédiatement et les réceptions sont suspendues jusqu'à ce qu'un message arrive. Ainsi, les réceptions servent de *gardes* aux processus. La syntaxe utilisée est dérivée de la notation de J. Glauert [8] mais modifiée afin de mieux modéliser la notion de *thread*. Ici, par *thread*, on entend une unité exécutable qui est activée et synchronisée par des entrées d'information et qui ensuite active d'autres unités directement ou par l'intermédiaire de sortie d'information.

$$
\begin{array}{lcl}
P, Q, T & = & \text{soit } \vec{x}\text{: } R \mid R \\
R & = & * \,\vec{\imath}\; S \mid \vec{\imath}\; S \\
S & = & \{P_1 \mid \ldots \mid P_n\} \mid \\
 & & \{[x_1 = y_1]P_1 + \ldots + [x_n = y_n]P_n\} \mid \\
 & & x!\vec{y} \mid x! \oplus (\vec{y}) \mid \\
 & & \text{erreur “mssg”} \mid \text{inert} \\
\imath & = & x?\vec{y}.
\end{array}
$$

Figure 3: Syntaxe du π-calcul

La figure 3 donne la syntaxe de notre version du π-calcul. Une unité exécutable (P, Q, T) commence par la déclaration de canaux de communications, désignée par "soit $\vec{x}$", où $\vec{x}$ est un vecteur indiquant un ou plusieurs identificateurs de canaux. Ensuite (R), le symbole "$*$" est utilisé pour indiquer une réplication, s'il y a lieu; la réplication correspond à une structure itérative qui évite de devoir utiliser des définitions récursives. Une liste d'entrées suit ($\vec{\imath}$); s'il y a une réplication, la suite d'entrées doit contenir au moins un élément. Enfin, l'unité (S) se termine soit par l'activation directe de processus, soit par l'envoi d'un message. L'activation directe de processus se présente de deux façons. Une première façon est l'activation de plusieurs processus concurrents, représenté par une barre verticale ($|$). Une deuxième est l'activation d'un processus sélectionné parmi plusieurs autres processus, en fonction d'une condition ($[x = y]$); l'opérateur "+" sépare alors les différents choix possibles. Lorsque plus d'une sélection est possible, le choix est non-déterministe; une fois qu'un processus est choisi, les autres possibilités sont éliminées. L'envoi d'un message, indiqué par un "!", permet de communiquer avec un autre processus en activant une réception de message (?). Ici, une fonction primitive ($\oplus$) peut aussi être appliquée à des arguments; dans un tel cas, c'est le canal représentant le résultat[1] qui est envoyé comme message. Notons que, si nécessaire, un message peut être un vecteur de canaux ($\vec{y}$). L'utilisation du processus "erreur" termine abruptement tous les processus en cours; quant au processus "inert", il n'effectue aucune action.

Ces restrictions sur le π-calcul correspondent aux conditions suggérées par Brock et Ostheimer [6] pour obtenir un modèle asynchrone qui soit utilisable pour la mise en oeuvre d'un compilateur: les envois de message ne peuvent pas garder des processus non-vide, et une réplication doit nécessairement être gardée par une réception de message.

3.2 Sémantique du π-calcul

Nous allons maintenant examiner les règles qui définissent la sémantique du π-calcul. Ces règles se divisent en deux groupes: les règles de congruence structurelle et les règles de transition et communication.

Règles de congruence structurelle Les règles de congruence structurelle (figure 4) nous permettent de comparer deux processus afin de savoir "s'ils se ressemblent". Ces règles de congruence sont parfois triviales, e.g., commutativité, mais généralement utiles pour éviter des réductions inutiles. Par exemple, la première règle indique la commutativité de l'opérateur parallèle et peut être appliquée plusieurs fois au même endroit sans aucun apport à la réduction. Ces règles permettent donc, dans les véritables règles de réduction et communication, de faire abstraction de la structure interne des processus.

Les quatre premières règles de la figure 4 sont assez simples: commutativité, associativité, élément neutre, élément absorbant. La cinquième règle, concernant la réplication, ne devrait être effectuée qu'au besoin: puisqu'une

[1] Par exemple, le canal qui porte le nom "1" représente le nombre 1.

$$
\begin{array}{llcl}
1: & P \mid Q & \equiv & Q \mid P \\
2: & \{\{P \mid Q\} \mid T\} & \equiv & \{P \mid Q \mid T\} \\
3: & P \mid \text{inert} & \equiv & P \\
4: & \text{soit } \vec{x}\text{: inert} & \equiv & \text{inert} \\
5: & * P & \equiv & P \mid * P \\
6: & \text{soit } x\text{: } \{ P \mid Q \} & \equiv & \{ P \mid \text{soit } x\text{: } Q \} \\
 & & & x \notin \mathit{VariablesLibres}(P) \\
7: & \text{soit } x\text{: } P & \equiv & \text{soit } y\text{: } P[y/x] \\
 & & & y \notin \mathit{VariablesLibres}(P)
\end{array}
$$

Figure 4: Règles de congruence structurelle

réplication est nécessairement suivie d'une réception de message, ce n'est que lors de l'arrivée d'un message que le processus devrait être répliqué. Les deux dernières règles définissent le fonctionnement de l'environnement.

Règles de transition et communication Les règles de transition de la figure 5 décrivent le processus de réduction du π-calcul, c'est-à-dire, l'essence du comportement dynamique des processus π-calcul. La première règle, COMM, décrit l'envoi d'un message. Ici, un message $\vec{z}$ est envoyé sur le canal x; une fois le message reçu par un processus $x?\vec{y}.P$, l'environnement de P doit être modifié, par substitution, pour tenir compte des nouvelles associations introduites par la déclaration des entrées $\vec{y}$. Les règles CHOIX, PAR et RES sont assez simples et correspondent, respectivement, au choix non-déterministe, à la possibilité de réduction en parallèle et sous les restrictions. Quant à la règle STRUCT, elle permet d'effectuer une réduction sur un processus à condition que celui-ci soit similaires à un processus réductible. (Deux processus sont similaires si on peut aller d'un à l'autre à l'aide des règles de congruence structurelle.)

Transmission sur un canal négocié Pour simplifier la présentation de nos règles sémantiques, nous allons augmenter notre syntaxe d'une construction permettant une communication sur un canal négocié entre deux processus. Cet ajout à la syntaxe, qui n'est en fait qu'une forme de sucre syntaxique, est proposé par plusieurs auteurs[20, 8, 12]. Puisque nous utilisons une version asynchrone du π-calcul, nous reprenons ce que Glauert [8] propose comme forme de communication sur un canal réservé. Ce type de transmission demande aux processus communiquant de négocier un canal réservé (unique à cet échange) et ensuite de transmettre le message sur ce nouveau canal.

$$
\begin{array}{lcl}
x! < \vec{y} > & \Rightarrow & \textbf{soit } t\text{: } \{x!t \mid t!\vec{y}\} \\
x? < \vec{y} > .P & \Rightarrow & x?p.p?\vec{y}.P
\end{array}
$$

où $p \notin \mathit{VariablesLibres}(P)$.

$$\text{COMM}: \quad x?\vec{y}.P \mid x!\vec{z} \longrightarrow P[\vec{z}/\vec{y}]$$

$$\text{CHOIX}: \quad \{\ldots + [x \;=\; y]P\} \longrightarrow P \qquad \text{si } x \;=\; y$$

$$\text{PAR}: \quad \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$$

$$\text{RES}: \quad \frac{P \longrightarrow P'}{\text{soit } x\colon P \longrightarrow \text{soit } x\colon P'}$$

$$\text{STRUCT}: \quad \frac{Q \equiv P \;\wedge\; P \longrightarrow P' \;\wedge\; P' \equiv Q'}{Q \longrightarrow Q'}$$

Figure 5: Règles de transition

4 Sémantique dénotationnelle parallèle du langage kpH

Maintenant que nous avons défini notre algèbre de processus, nous allons l'utiliser pour décrire la sémantique parallèle de `kpH`. Plus précisément, nous allons définir une sémantique dénotationnelle de `kpH`. Toutefois, contrairement à la pratique usuelle où les dénotations sont des expressions λ-calcul, les dénotations associées seront plutôt des expressions π-calcul, donc des processus.

4.1 Sémantique du programme principal

Un programme `kpH` consiste en l'évaluation d'une expression (**Terme**). La fonction décrivant la sémantique d'un programme P est la suivante:

$$\Xi(P, o) \quad \triangleq \quad [\![P]\!]_o$$

Étant donné un canal o, cette fonction associe au programme P un processus π-calcul (cf. plus bas). Ce processus, lorsque réduit, permettra de produire le résultat du programme sur le canal o. Cette fonction utilise une fonction décrivant la sémantique des expressions, introduite dans la prochaine section.

4.2 Sémantique d'une expression

La fonction définissant la sémantique d'une expression prend en argument une expression, i.e., un **Terme**, et un **Canal** de sortie. Le canal de sortie est utilisé comme référence à la valeur produite par l'évaluation du terme; il suffit donc d'effectuer une lecture sur ce canal pour obtenir cette valeur. La signature de la fonction sémantique pour une expression est donc la suivante:

$$[\![\ \_\ ]\!]_{\_} : \mathbf{Terme} \rightarrow \mathbf{Canal} \rightarrow \mathbf{Processus}$$

Le type le plus simple d'expression que l'on peut retrouver est une valeur constante. Cette valeur est simplement envoyée sur le canal de sortie indiqué:

$$[\![v]\!]_o \quad \triangleq \quad o!v$$

L'évaluation d'un identificateur consiste simplement à faire une demande sur le canal associé à cet identificateur en lui transmettant le canal de sortie. L'association (voir plus bas) va retransmettre la valeur voulue sur le canal de sortie:

$$[\![y]\!]_o \quad \triangleq \quad y!o$$

Les fonctions "primitives" seront des fonctions qui utilisent tous leurs arguments et qui sont évaluées de façon stricte. Chaque argument sera évalué en parallèle et le processus contenant l'appel de la primitive ne pourra retourner une réponse tant que l'évaluation des arguments et de l'application de la primitive ne sera pas terminée:

$$\begin{array}{lll} [\![\oplus(e_1,\ldots,e_h)]\!]_o & \triangleq & \textbf{soit } o_1,\ldots,o_h\text{:} \\ & & \{[\![e_1]\!]_{o_1} \mid \ldots \mid [\![e_h]\!]_{o_h} \\ & & \quad \mid \\ & & \quad o_1?v_1.\ldots.o_h?v_h.o!\oplus(v_1,\ldots,v_h) \\ & & \} \end{array}$$

Avant d'introduire la sémantique des expressions **let** et des appels de fonction, il nous faut introduire un certain nombre d'abstractions auxiliaires. La première abstraction va nous permettre d'établir une association (*binding*) entre un identificateur et une expression, l'identificateur étant représenté par un canal du même nom. [2] Ce canal va servir pour l'émission et la réception de la valeur associée à l'identificateur. Une association identificateur/expression, après avoir évalué l'expression, deviendra donc un processus toujours prêt à être répliqué à l'aide d'une demande sur le canal représentant l'identificateur. Ce processus reçoit un canal de communication privé du demandeur, et ensuite envoie la valeur requise au demandeur sur ce canal privé:

[2]La portée des identificateurs, comme on le verra plus loin, sera limitée à l'aide de restrictions π-calcul.

$$\mathbf{associer}(n, e) \stackrel{\text{def}}{=} \mathbf{soit}\ o:$$
$$\{[\![e]\!]_o \mid o?v.\{*n?p.p!v\}\}$$

Ici, $[\![e]\!]_o$ dénote le processus permettant l'évaluation de l'expression e et qui envoie le résultat sur le canal o; une fois cette valeur reçue, on l'associe à n.

Une autre abstraction permet de définir plusieurs associations, ce qui sera utile pour la définition complète d'un environnement d'évaluation:

$$\mathbf{définir}(n_1 = e_1; \ldots; n_h = e_h) \stackrel{\text{def}}{=}$$
$$\{\mathbf{associer}(n_1, e_1) \mid \ldots \mid \mathbf{associer}(n_h, e_h)\}$$

Avec l'abstraction **définir**, il devient maintenant facile d'expliquer le fonctionnement d'une expression **let** : il s'agit simplement d'évaluer l'expression du **let** à l'intérieur d'un nouvel environnement dans lequel on définit les nouveaux identificateurs. La portée correcte de l'environnement est assurée par une restriction π-calcul. Il est à remarquer que cette évaluation est indulgente, car l'évaluation de l'expression du **let** (e) peut retourner un résultat *avant* que toutes les associations ne soient complètement établies:

$$\begin{array}{l}
[\![\mathbf{let}\ \ n_1 = e_1; \ldots; n_h = e_h\ \mathbf{in}\ \ e]\!]_o \quad \triangleq \quad \mathbf{soit}\ n_1, \ldots, n_h: \\
\qquad\qquad \{\mathbf{définir}(n_1 = e_1; \ldots; n_h = e_h) \\
\qquad\qquad\quad \mid \\
\qquad\qquad\quad [\![e]\!]_o \\
\qquad\qquad \}
\end{array}$$

La sémantique des fonctions et appels de fonction doit aussi modéliser l'évaluation indulgente. Tout d'abord, examinons la sémantique d'une λ-abstraction:

$$\begin{array}{l}
[\![\lambda x_1, \ldots, x_h.e]\!]_o \quad \triangleq \quad \mathbf{soit}\ l: \\
\qquad\qquad \{o!l \\
\qquad\qquad\quad \mid \\
\qquad\qquad\quad * \ l? < r, x_1, \ldots, x_h > . \\
\qquad\qquad\qquad \{[\![e]\!]_r\} \\
\qquad\qquad \}
\end{array}$$

Ici, l sert de *pointeur* vers la fonction; ce canal de référence est créé pour que les appelants puissent y communiquer les canaux privés qui seront utilisés pour transmettre les arguments (les canaux x_i) et recevoir le résultat (le canal r). La réception d'un message sur l va donc créer des associations locales pour les identificateurs r et x_i, lesquelles associations seront ensuite visibles dans le processus évaluant le corps de la fonction $(\{[\![e]\!]_r\})$.

C'est l'abstraction auxiliaire **activer** qui va permettre d'obtenir le pointeur d'une fonction et ensuite de l'activer en lui communiquant les arguments et le canal pour la réponse:

$$
\begin{array}{l}
\textbf{activer}(f, [p_1, \ldots, p_h]) \stackrel{\text{def}}{=} \textbf{soit } b\text{:} \\
\quad \{f!b \\
\quad\quad | \\
\quad\quad b?l.l! < p_1, \ldots, p_h > \\
\quad \}
\end{array}
$$

Avec cette dernière abstraction, il ne reste plus, pour traiter un appel de fonction, qu'à créer des associations entre de nouveaux identificateurs et les expressions utilisées comme arguments effectifs, puis à activer la fonction f avec ces nouveaux canaux:

$$
\begin{array}{lll}
[\![f(e_1, \ldots, e_h)]\!]_o & \triangleq & \textbf{soit } y_1, \ldots, y_h\text{:} \\
& & \{\textbf{activer}(f, [o, y_1, \ldots, y_h]) \\
& & \quad | \\
& & \quad \textbf{définir}(y_1 = e_1; \ldots; y_h = e_h) \\
& & \}
\end{array}
$$

Nous remarquons donc que l'évaluation de la fonction est commencée en parallèle à l'évaluation des arguments. Cette évaluation sera suspendue temporairement lorsqu'un argument dont on a besoin ne sera pas encore disponible, ce qui permettra une évaluation indulgente.

4.3 Constructeurs de données et expressions case

Un processus modélisant une structure de données doit pouvoir fournir deux types d'informations différentes: le constructeur comme tel (le *tag*), et des références aux éléments qui composent la structure. Comme on désire obtenir des structures de données non-strictes, les éléments n'ont pas besoin d'être évalués et disponibles pour que la référence vers le niveau supérieur de la structure (p) soit "prête" et puisse être retournée. Un appel à un constructeur de données retourne donc un canal de communication (p) sur lequel pourra ensuite être négocié deux canaux: un premier (c) pour l'identification du constructeur ($K_\imath$); un deuxième (v) pour référer aux éléments, plus précisément, par l'intermédiaire duquel les canaux requis pour accéder aux composants ($c_1, \ldots, c_h$) seront négociés:

$$
\begin{array}{lll}
[\![K_\imath(e_1, \ldots, e_h)]\!]_o & \triangleq & \textbf{soit } c_1, \ldots, c_h, p\text{:} \\
& & \{\textbf{définir}(c_1 = e_1; \ldots; c_h = e_h) \\
& & \quad | \\
& & \quad o!p \\
& & \quad | \\
& & \quad * \, p? < c, v > . \\
& & \quad\quad \{c!K_\imath \;\;|\;\; v!(c_1, \ldots, c_h)\} \\
& & \}
\end{array}
$$

Finalement, la dernière expression à traiter est le **case** :

$$
\begin{array}{l}
[\![\mathbf{case}\ \ y\ \mathbf{as}\ \ K_1(n_{11},\ldots,n_{1a_1})\ \mathbf{in}\ \ e_1;\ldots]\!]_o \ \ \triangleq\ \ \mathbf{soit}\, v, c{:}\\
\qquad\qquad \{\mathbf{activer}(y,[c,v])\\
\qquad\qquad\quad |\\
\qquad\qquad\quad c?K.\\
\qquad\qquad\qquad \{[K=K_1]v?(n_{11},\ldots,n_{1a_1}).\{[\![e_1]\!]_o\}\\
\qquad\qquad\qquad\quad +\ldots+\\
\qquad\qquad\qquad\quad [K=K_j]v?(n_{j1},\ldots,n_{ja_j}).\{[\![e_j]\!]_o\}\\
\qquad\qquad\qquad \}\\
\qquad\qquad \}
\end{array}
$$

Tout d'abord, on doit établir la communication avec le processus associé à la structure de données à examiner (y). Pour ce faire, nous pouvons réutiliser l'abstraction **activer**. Une fois le constructeur reçu et connu ($c?K$), l'alternative appropriée est sélectionnée; les autres ne seront pas exécutées. Notons que c'est seulement lorsque le constructeur est connu qu'on peut ensuite obtenir les canaux qui permettront de référer aux composantes de la structure; ceci se fera par l'intermédiaire du canal v reçu préalablement ($v?(n_{j1},\ldots,n_{ja_j})$). On peut alors amorcer l'évaluation de l'expression associée à la branche sélectionnée (e_j).

5 Un interpréteur parallèle de π-calcul réalisé à l'aide des *threads* de `Java`

Pour vérifier la validité de la sémantique présentée à la section précédente, une preuve formelle serait intéressante et utile. Toutefois, avant d'aborder une telle preuve, il est préférable de se convaincre informellement que cette sémantique est correcte. Dans le but d'effectuer une telle validation, nous avons développé un interpréteur parallèle de π-calcul réalisé à l'aide des *threads* du langage Java. La mise en oeuvre de cet interpréteur est décrite à la section 5.1, alors que quelques-uns des tests ayant été effectués sont décrits à la section 5.2.

5.1 Mise en oeuvre de l'interpréteur

Un interpréteur pour un langage de programmation permet d'évaluer des programmes écrits dans ce langage. Contrairement à un compilateur, un interpréteur ne génère par de code machine exécutable; un interpréteur, à partir d'une représentation du programme dans une forme intermédiaire appropriée, e.g., arbre syntaxique abstrait, analyse directement cette représentation et l'évalue.

Diverses formes d'interpréteur peuvent être réalisées pour un langage donné. Dans un interpréteur classique, la structure de données représentant le programme est *passive*: on a alors, d'un coté, la procédure d'analyse et d'interprétation, de l'autre, l'arbre abstrait qui est analysé et évalué par l'interpréteur — [1] nomme cette approche "*syntax separate from interpretation*". Dans un langage orientée-objet, une approche différente peut être utilisée, à savoir que chaque construction syntaxique peut être associée à une classe, laquelle classe possède alors une méthode d'interprétation propre à la catégorie syntaxique représentée, méthode qui réalise l'effet de la construction, qui l'interprète [1].

Dans le contexte du langage `Java`, qui permet la création et manipulation de *threads* [16], l'approche orientée-objet est aussi intéressante pour obtenir une mise en oeuvre parallèle d'un interpréteur de π-calcul. En `Java`, un *thread* est un processus indépendant; toutefois, contrairement à un processus classique (e.g., de niveau système d'exploitation), un *thread* ne possède pas un espace mémoire indépendant. Le contexte d'un *thread* est donc léger (on parle parfois de *lightweight thread*), ce qui permet d'avoir, dans un même programme, un grand nombre de *threads* et ce qui permet d'effectuer des changements de contexte à des coûts relativement faibles.

Le lien entre la sémantique formelle et l'interpréteur de π-calcul est établie en traduisant un programme `kpH` dans une forme textuelle représentant sa dénotation π-calcul. Cette forme textuelle est ensuite analysée et un arbre syntaxique abstrait approprié est construit, où chaque noeud de l'arbre abstrait correspond à une instance d'une classe. À chaque catégorie syntaxique — e.g., réplication, choix, processus concurrents — est donc associée une classe dont l'une des méthodes consiste à *interpréter* l'instance, c'est-à-dire, à l'évaluer selon les règles du π-calcul, en fonction des attributs propres à l'instance et en fonction de l'environnement statique approprié. L'interprétation parallèle est obtenue par le fait que la méthode d'interprétation, lorsque cela est utile, crée un *thread* indépendant s'exécutant en parallèle. Quant aux communications entre les divers *threads*, elles sont gérées par l'intermédiaire d'instances d'objets canaux créées pour les constructions "soit $\vec{x}$:".

5.2 Quelques tests et un profil de parallélisme

Un certain nombre de programmes tests ont été traduits à l'aide de l'approche décrite précédemment. Quelques-uns de ces tests sont présentés à la fig. 6. Les programmes `Ones, FacListe` et `Nonstrict_cond` ont été utilisés pour vérifier que la sémantique mise en oeuvre est bien indulgente: ces programmes contiennent tous une définition du style $x = f(x, \ldots)$, donc avec une dépendance cyclique, et ne peuvent donc pas être évalués de façon stricte.

Dans le but de comparer le parallélisme entre la version stricte et la version indulgente de divers programmes, nous avons défini, en utilisant une approche semblable à celle décrite dans les pages précédentes, une version *stricte* de la sémantique π-calcul de `kpH`, version stricte qui a été obtenue en modifiant la sémantique de la fonction **définir** afin qu'elle évalue tous les éléments définis avant de permettre à l'évaluation de continuer.

La figure 7 présente deux graphiques illustrant le nombre de *threads* exécutés

Nom du programme	Brève description
`FacListe`	Génére des factoriels en réutilisant les éléments de la liste.
`Feuilles`	Construit la liste des feuilles d'un arbre binaire.
`Fib`	Évalue un nombre de Fibonnacci.
`Insertionsort`	Effectue un tri par insertion sur une liste.
`Integrate_int`	Calcule une intégrale par partie.
`Nonstrict_cond`	Un élément retourné par une fonction est utilisé comme argument de la même fonction.
`Ones`	Retourne le deuxième élément d'une liste infinie de 1.
`Sqrt_int`	Applique la méthode d'approximations successives pour évaluer une racine carrée.
`Sum_par`	Effectue la sommation de 1 à n récursivement (diviser-pour-règner équilibré).

Figure 6: Les programmes tests

à chaque unité de temps, en supposant un parallélisme idéal (profil idéal de parallélisme), par les versions stricte vs. indulgente de la fonction `Feuilles`. On remarque bien que, dans la version indulgente, le parallélisme disponible est plus grand qui celui de la version stricte et que la longueur du chemin critique est plus courte; un résultat semblable avait été obtenu pour ce programme par [14].

6 Conclusion

Dans cet article, nous avons présenté une sémantique dénotationnelle d'un langage fonctionnel indulgent. Dans le but d'exprimer explicitement le parallélisme présent dans un tel langage, nous avons choisi d'utiliser comme dénotations, non pas des expressions λ-calcul classiques, mais plutôt des expressions π-calcul.

Par la suite, dans le but de valider (via des tests) cette sémantique, nous avons développé un interpréteur parallèle de π-calcul en utilisant les *threads* du langage `Java`. Nous avons alors pû réaliser que la stratégie utilisée pour produire, à partir d'un programme source `pH`, son interprétation en une sémantique π-calcul était relativement flexible. Ainsi, une fois que les divers modules permettant la production de la sémantique indulgente furent fonctionnels, la version stricte put être produite en moins d'une journée. [3]

[3] Notons qu'une sémantique formelle paresseuse a aussi été produite mais n'a pas encore été mise en oeuvre.

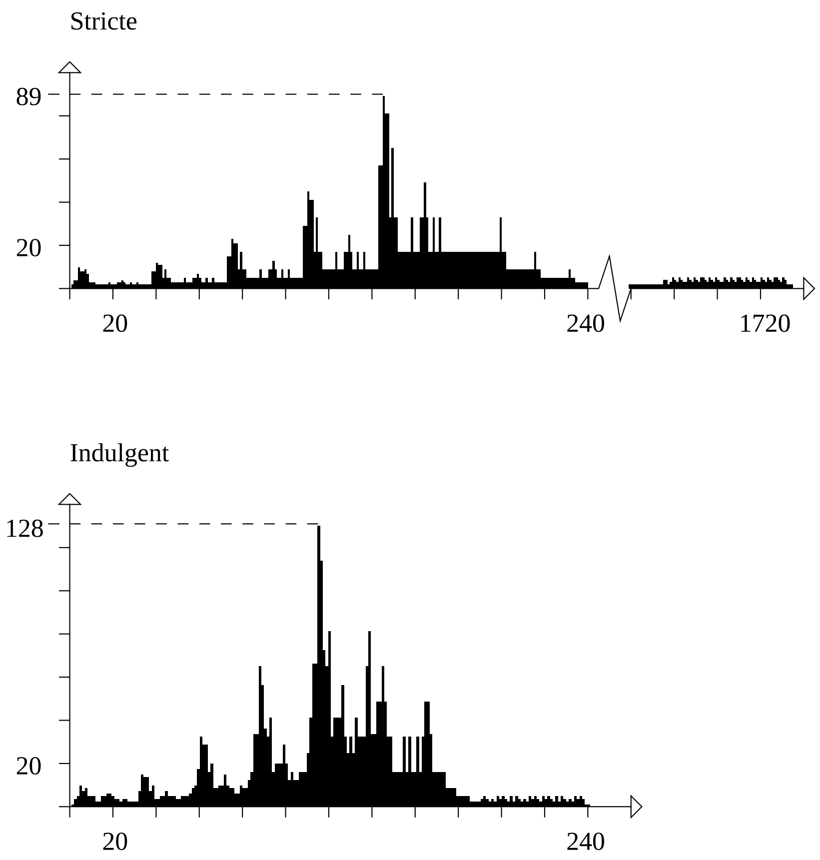

Figure 7: Comparaison de l'évaluation stricte vs. indulgente pour le programme `Feuilles`

D'autres auteurs avant nous ont utilisé le π-calcul pour produire diverses sémantiques de langages fonctionnels, tant stricts, indulgents que paresseux [11, 9, 17, 6]. Dans tous les cas, le langage source était le λ-calcul pur dont la sémantique était modélisée (encodée) en π-calcul. Notre propre sémantique est inspirée principalement de celle décrite dans [17], avec utilisation exclusive de processus asynchrones. Certains de ces auteurs ont aussi prouvé formellement l'équivalence entre la sémantique classique du λ-calcul et celle obtenue via sa modélisation en π-calcul. Des travaux dans cette direction, pour notre langage et sa sémantique, sont présentement en cours. À plus long terme, notre objectif est véritablement de mieux comprendre les avantages et limites de ces diverses sémantiques et mises en oeuvre, plus particulièrement dans un contexte parallèle.

Nous sommes aussi en train de développer et mettre en oeuvre une sémantique π-calcul pour les niveaux non-purement fonctionnels de `pH`, plus précisément, les `I-` et `M-`structures. Le premier type permet de réaliser des structures de données parallèles définies de façon incrémentales [3], alors que le deuxième permet de réaliser des structures de données mutables [5], et ce dans un contexte fonctionnel avec parallélisme implicite. Dans les deux cas, ces structures de données sont bien adaptées au parallélisme, supportant des mécanismes implicites et distribués de synchronisation.

7 Remerciements

La recherche décrite dans cet article a été supportée par une subvention de recherche du Conseil de Recherche en Sciences Naturelles et en Génie du Canada (CRNSG #OGP0183776). L'analyseur permettant de transformer la forme textuelle du π-calcul en arbres abstraits à partir desquels les classes `Java` sont générées a été développé par Robin Lafortune, étudiant au baccalauréat en informatique, dans le cadre d'un cours en lecture dirigé (Théorie et construction des compilateurs).

Bibliographie

[1] A.W. Appel. *Modern Compiler Implementation in Java — Basic Techniques.* Cambridge University Press, 1997.

[2] Arvind, A. Caro, J.-W. Maessen, and S. Aditya. A multithreaded substrate and compilation model for the implicitly parallel language pH. CSG Memo 382, MIT, May 1996.

[3] Arvind, R.S. Nikhil, and K.K. Pingali. I-Structures: Data structures for parallel computing. *TOPLAS*, 11(4):598–632, October 1989.

[4] L. Augustsson and T. Johnsson. Parallel graph reduction with the $\langle \nu, g \rangle$-machine. In *Proc. of the Fourth International Conf. on Funct. Prog. Lang. and Comp. Arch.*, pages 202–213. ACM Press, 1989.

[5] P.S. Barth, R.S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional languages with state. Technical report, Massachusetts Institute of Technology, March 1991. Computation Structures Group Memo 327.

[6] S. Brock and G. Ostheimer. Process semantics of graph reduction. Technical Report CS/95/2, Computer Science Division, University of St-Andrews, March 1995.

[7] G.T. Byrd and M.A. Holliday. Multithreaded processor architectures. *IEEE Spectrum*, 32(8):38–46, August 1995.

[8] J. Glauert. Asynchronous mobile processes and graph rewriting. In *PARLE '92*, pages 63–78. Springer-Verlag, LNCS-605, 1992.

[9] J. Glauert, L. Leth, and B. Thomsen. A new process model for functions. In M.R. Sleep, M.J. Plasmeijer, and M.C.J.D. van Eekelen, editors, *Term Graph Rewriting — Theory and Practice*, Wiley Professional Computing, pages 269–282. John Wiley & Sons, 1993.

[10] B. Lee and A.R. Hurson. Dataflow architectures and multithreading. *IEEE Computer*, 27(8):27–39, 1994.

[11] R. Milner. Functions as processes. In *Automata, Languages and Programming*, pages 167–180. Springer-Verlag, LNCS-443, 1990.

[12] R. Milner. The polyadic *pi*-calculus: A tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, 1991.

[13] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and Computation*, 100:1–40, 1992.

[14] R.S. Nikhil. The parallel programming language Id and its compilation for parallel machines. Computation Structures Group Memo 313, Laboratory for Computer Science, MIT, July 1990.

[15] R.S. Nikhil, Arvind, J. Hicks, S. Aditya, L. Augustsson, J.-W. Meassen, and Y. Zhou. pH language reference manual, version 1.0 — preliminary. Technical Report 369, Laboratory for Computer Science, MIT, Jan. 1995.

[16] S. Oaks and H. Wong. `Java` *Threads*. O'Reilly, 1997.

[17] G. Ostheimer and A.J.T. Davie. π-calculus characterizations of some practical lambda-calculus reduction strategies. Technical Report CS 93/14, University of St Andrews, Division of Computer Science, Oct. 1993.

[18] J. Peterson and K. Hammond (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.4). Technical report, Yale University, Department of Computer Science, April 1997. http://haskell.systemsz.cs.yale.edu/onlinereport/.

[19] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science, 1987.

[20] Benjamin C. Pierce. Foundational calculi for programming languages. In Allen B. Tucker, editor, *Handbook of Computer Science and Engineering*, chapter 139. CRC Press, 1996.

[21] G. Tremblay. Langages fonctionnels non-stricts: De la paresse et de l'indulgence. In *JFLA 97 – Journées Francophones des Langages Applicatifs*, pages 189–208. INRIA #16 (Collection Didactique), 1997.