LIFO
Université d'Orléans
Stage de DEA Informatique

Année universitaire
1999-2000

BSλ simplement typé : Typage et Sémantique naturelle

Armelle Merlin

Stage encadré par Gaétan HAINS et Frédéric LOULERGUE.

Table des matières

Remerciements à Gaétan Hains, Frédéric Loulergue et Quentin Miller.

Chapitre 1

Introduction

Plusieurs applications de l'informatique nécessitent des performances que seules les machines massivement parallèles peuvent offrir. Mais la conception de langages adaptés est difficile si on cherche à concilier simplicité et efficacité.

Deux types de programmations sont généralement utilisés : la programmation séquentielle, facile d'approche mais limitée par l'utilisation de tableaux et de boucles, et la programmation concurrente, combinant un langage séquentiel et une bibliothèque de communication mais d'une grande complexité due à l'indeterminisme et la possibilité de blocage.

D'où l'idée de concilier le modèle BSP [13] et la programmation fonctionnelle. Le modèle BSP introduit le parallélisme de données qui apporte le determinisme et le non-blocage. La programmation fonctionnelle évite l'utilisation de tableaux et de boucles.

Les deux principales catégories d'association de la programmation fonctionnelle et du parallélisme sont les extensions explicitement parallèles des langages fonctionnels et les implantations parallèles avec sémantique fonctionnelle. La première aboutit à des langages soit indéterministes soit fonctionnels "impurs", la deuxième des langages qui n'expriment pas directement les algorithmes parallèles et qui ne permettent pas de prévoir le temps d'exécution.

Les langages à partons permettent de rompre avec ces contraintes. Puisqu'un ensemble fixé d'opérateurs sont exécutés en parallèle.

Le formalisme BSλ [12] ajoute au λ-calcul des opérations explicitement parallèles BSP tout en conservant une sémantique purement fonctionnelle. De plus, étant basé sur le modèle BSP, il permet une précision réaliste et portable des performances.

Une implantation de ces opération BSP, sous forme d'une bibliothèque pour OCaml, a été réalisée [1] et a montré la possibilité des précision des performances.

Toutefois, aucun lien formel n'existe entre le BSλ-calcul et cette bibliothèque BSMLlib. Le but du travail effectué est de fournir un tel lien, en proposant une machine abstraite parallèle réalisant une stratégie particulière du BSλ-calcul.

Le BSλ-calcul repose sur une distinction syntaxique entre termes locaux (présents sur un processeur) et termes globaux (objets parallèles). Conserver une distinction syntaxique aurait obligé à maintenir cette distinction dans la machine abstraite, rendant plus difficile sa conception.

Nous avons donc choisi de présenter un BSλ-calcul simplement typé, dont la

sémantique naturelle est toutefois correcte par rapport au BSλ_p-calcul [10](une variante "vectorielle" de BSλ).

Après les préliminaires (Chapitre 2), nous reprenons une présentation similaire à celle de Mini-ML [3] : syntaxe (Section 3.1), sématique naturelle (Section 3.3), typage monomorphe (Section 3.4) et machine à pile (Section 3.5). Nous présentons enfin une sémantique naturelle avec coûts (Section 3.6) et l'illustrons sur des exemples de programmes. Les coûts déterminés par cette sémantique réalisent une version parallèle BSP du décompte des instructions de haut niveau, comme approximation de temps de calcul.

Chapitre 2

Préliminaires

Cette section est tirée des publications ([11],[8],[3]) sauf pour les sections 2.4.1 et 2.5.3.

2.1 Modèle BSP

Le modèle *Bulk-Synchronous Parallelism* (BSP) est un modèle de programmation parallèle introduit par Valiant [13] pour offrir un niveau d'abstraction comparable aux modèles PRAM tout en permettant des performances prévisibles et portables sur une large variété d'architectures. Un ordinateur BSP contient un ensemble de paires processeur-mémoire, un réseau de communication permettant l'échange de messages inter-processeur et une unité de synchronisation globale qui exécute des demandes collectives de barrières de synchronisation. Ses performances sont caractérisées par trois paramètres: le nombre p de paires de processeur-mémoire, le temps l nécessaire à une barrière de synchronisation et le temps g nécessaire à une 1-relation (phase de communication où chaque processeur envoie ou reçoit au plus un mot). Pour n'importe quel h le réseau peut réaliser une h-relation, c'est-à-dire une phase de communication où chaque processeur envoie ou reçoit au plus h mots, en temps gh.

Un programme BSP est exécuté comme une séquence de *super-étapes*, chacune étant au plus divisée en trois phases successives et logiquement disjointes (Figure 2.1).

Pendant la première phase, chaque processeur utilise ses données locales pour du calcul séquentiel et pour demander des transferts de données vers ou depuis d'autres noeuds. Pendant la seconde phase, le réseau effectue les transferts de données demandées. Pendant la troisème phase, une barrière de synchronisation se produit, rendant disponibles pour la super-étape suivante les données tranférées. Le temps d'exécution d'une super-étape s est ainsi la somme du maximum des temps de calculs locaux, du temps de communication des données et du temps de synchronisation globale:

$$Time(s) = \max_{i:processeur} w_i^{(s)} + \max_{i:processeur} h_i^{(s)} * g + l$$

où $w_i^{(s)}$ = temps de calcul local du processeur i durant la super-étape s et $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ où $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) est le nombre de mots transmis

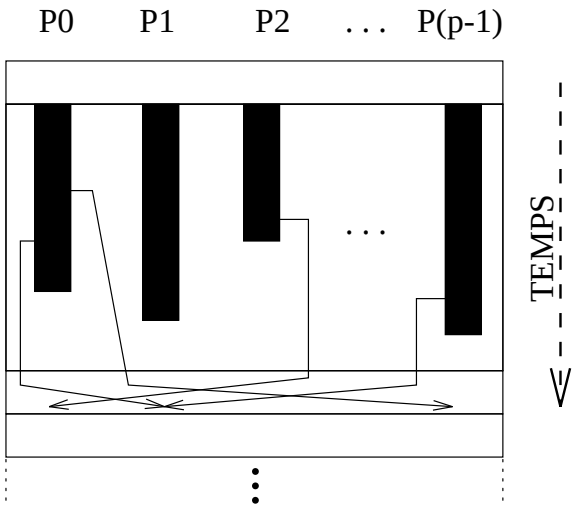

FIG. 2.1 – *Super-étapes BSP*

(resp. reçus) par le processeur i durant la super-étape s. Le temps d'exécution $\sum_s Time(s)$ d'un programme BSP composé de S super-étapes est la somme des trois termes : $W + H * g + S * l$ où $W = \sum_s \max_i wi^{(s)}$ et $H = \sum_s \max_i h_i^{(s)}$. En général W,H et S sont fonctions de p et de la taille des données n, ou de paramètres plus complexes. Pour minimiser le temps d'exécution, un algorithme BSP doit minimiser conjointement le nombre de super-étapes, le volume total H (resp. W) et les déséquilibres $h^{(s)}$ (resp. $w^{(s)}$) de communication (resp. de calcul local).

2.2 Le BSλ-calcul

2.2.1 Syntaxe

Nous considérons l'ensemble $\dot{\mathcal{V}}$ des variables *locales* et l'ensemble $\bar{\mathcal{V}}$ des variables *globales*. Nous notons $\dot{x}$,$\dot{y}$,... les variables locales et $\bar{x}$,$\bar{y}$,... les variables globales [1]. x dénote une variable soit locale soit globale.

La syntaxe de BSλ commence avec les termes *locaux* e : ce sont des λ-termes représentant des valeurs ou des programmes stockés dans la mémoire locale d'un processeur. L'ensemble $\dot{\mathcal{T}}$ des termes locaux est donné par la grammaire suivante :

$$e \quad ::= \quad \dot{x} \mid e\, e \mid \lambda \dot{x}.\, e \mid c$$

où $\dot{x}$ dénote une variable locale arbitraire. Nous notons $(e_1 \rightarrow e_2,e_3)$ la conditionnelle $e_1\ e_2\ e_3$. L'opération booléenne $\Rightarrow$ est aussi supposée être encodée sous forme de λ-terme [2].

Les termes principaux E de BSλ sont appelés termes *globaux* et représentent des *champs de données*, c'est-à-dire des fonctions d'un ensemble fixé de proces-

1. Le point $\dot{}$ symbolise un processeur précis du réseau, la barre $\bar{}$ symbolise le réseau tout entier

seurs vers des valeurs. Les noms des processeurs ne sont pas spécifiés mais l'on suppose que ce sont des λ-termes clos de telle sorte que les champs de données puissent être exprimés intentionnellement par une constante spéciale appelée π (pour objet *parallèle*). Le terme $\pi\, e$ représente un champ de données dont les valeurs sont données par la fonction e (Figure 2.2, pour illustration les noms de processeurs seront toujours $0, \ldots, p-1$).

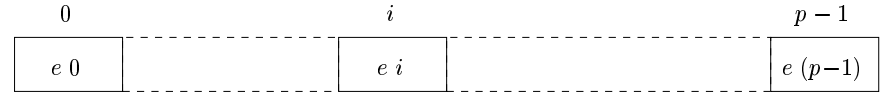

FIG. 2.2 – *Le terme* πe

L'ensemble $\bar{\mathcal{T}}$ des termes globaux est donné par la grammaire suivante :

$$\begin{array}{rcl} E & ::= & \bar{x} \mid E\,E \mid E\,e \mid \lambda\bar{x}.\ E \mid \lambda\dot{x}.\ E \\ & \mid & \pi\ e \mid E\#E \mid E?E \mid E \xrightarrow{e} E,E \end{array}$$

et a la signification suivante.

Soit p le nombre fini des noms de processeurs $\mathcal{N} = \{n_i \mid\ 0 \leq i < p\}$, chacun étant un λ-terme clos. Les termes globaux dénotent des fonctions de $\mathcal{N}$ vers des valeurs locales, des fonctions entre elles ou des fonctions des termes locaux vers de telles fonctions. En particulier, la dénotation de $\pi\, e$ a, au processeur n_i, la valeur (forme normale) de $e\, n_i$. Les formes $E_1\#E_2$ et $E_1?E_2$ sont appelées application parallèle (*apply-par*) et *get* respectivement. Apply-par représente l'application point à point d'un champ de fonctions à un champ de valeurs (phase de calcul pur d'une super-étape BSP). Get représente la phase de communication d'une super-étape BSP : un échange collectif de données avec une barrière de synchronisation. Dans $E_1?E_2$, le champ de données résultant contient les valeurs de E_1 prises aux noms de processeurs définis dans E_2. La signification exacte de apply-par et get est définie par les équations de BSλ. La dernière forme de terme global définit la conditionnelle globale synchrone. La signification de $E_1 \xrightarrow{e} E_2,E_3$ est celle de E_2 (resp. E_3) si le champ de données dénoté par E_1 a la valeur T (resp. F) au processeur de nom dénoté par e.

2.2.2 Sémantique équationnelle

L'égalité des termes locaux est simplement la β-équivalence: la fermeture réflexive, symétrique et transitive de

$$(\beta) \qquad (\lambda\dot{x}.\ e)e' = e[\dot{x} \leftarrow e']$$

appliquée à n'importe quel sous-terme.

L'égalité des termes globaux est définie par des règles basées sur la syntaxe et des règles de contexte qui déterminent l'applicabilité des premières. Nous donnons maintenant les règles basées sur la syntaxe:

Il y a d'abord les axiomes concernant la beta-équivalence globale.

$$\begin{array}{lrcl} (B) & (\lambda\bar{x}.\ E)E' & = & E[\bar{x} \leftarrow E'] \\ (B') & (\lambda\dot{x}.\ E)e' & = & E[\dot{x} \leftarrow e'] \end{array}$$

Puisque les termes $(\lambda\dot{x}.E_1)\,E_2$ sont syntaxiquement corrects, mais que la substitution $E_1[\dot{x} \leftarrow E_2]$ n'est pas syntaxiquement correcte, les deux règles (B) et (B') sont nécessaires.

Il y a également des axiomes concernant l'interaction du constructeur de champ π avec les autres opérations BSP.

Les équations $(?\pi)$ (Figure 2.3) et $(\#\pi)$ (Figure 2.4) encodent la signification dénotationelle des opérations BSP sur les champs. En particulier, get est la composition fonctionnelle à l'intérieur du π. La valeur de $\pi\, e_1 ?\pi\, e_2$ au processeur de nom n_i est $e_1\,(e_2\, n_i)$, c'est-à-dire la valeur de $\pi\, e_1$ au processeur de nom $e_2\, n_i$. On notera qu'en pratique ceci représente une opération où chaque processeur reoit une (et une seule) valeur d'un et d'un seul autre processeur. De nouveaux opérateurs de communication (multiget et put) permettent de se libérer de cette contrainte [11].

$$(?\pi) \quad \frac{\forall i \in \mathcal{N}\ .\ e_2\ i \doteq n \in \mathcal{N}}{(\pi\ e_1)?(\pi\ e_2) = \pi(\lambda\dot{x}.\ e_1(e_2\ \dot{x}))}$$

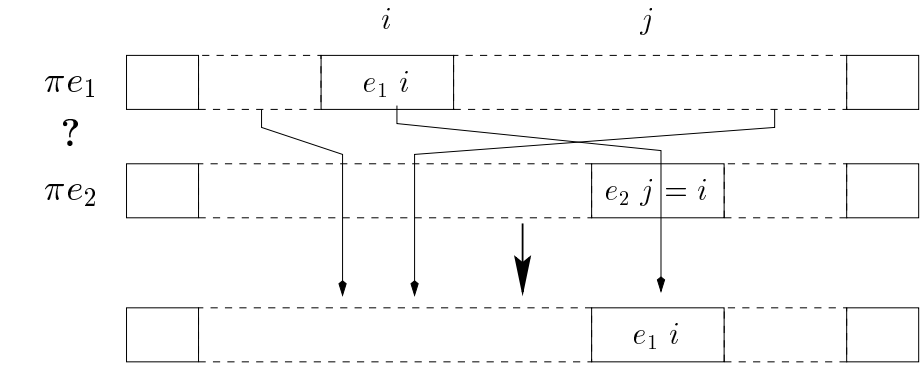

FIG. 2.3 – *La règle* $(?\pi)$

$$(\#\pi) \quad (\pi\ e_1)\#(\pi\ e_2) \quad = \quad \pi(\lambda\dot{x}.\ (e_1\ \dot{x})(e_2\ \dot{x}))$$

La conditionnelle globale synchrone est définie par deux règles dont les numérateurs font référence à des calculs locaux.

$$(\stackrel{e}{\rightarrow} T)\ \frac{e\ e' \doteq T \quad e' \in \mathcal{N}}{(\pi\ e) \stackrel{e'}{\rightarrow} E_1, E_2\ =\ E_1}$$

$$(\stackrel{e}{\rightarrow} F)\ \frac{e\ e' \doteq F \quad e' \in \mathcal{N}}{(\pi\ e) \stackrel{e'}{\rightarrow} E_1, E_2\ =\ E_2}$$

Les deux règles précédentes (Figure 2.5 pour le premier cas) génèrent le calcul BSP suivant : premièrement une phase de calcul pur où tous les processeurs évaluent le terme local e' conduisant à la valeur v. Le processeur v prend alors

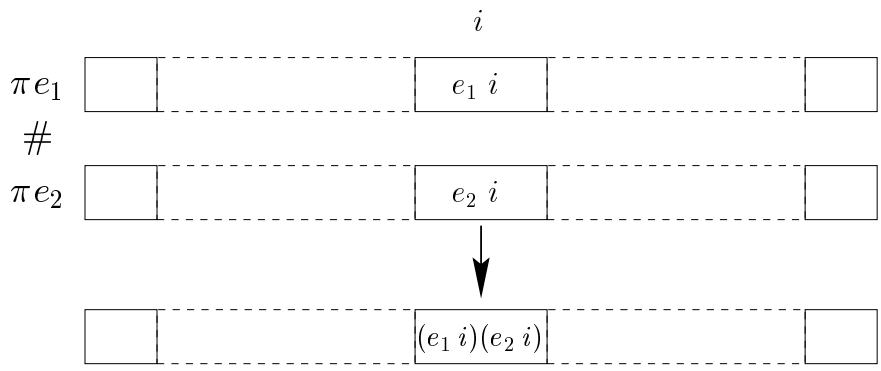

FIG. 2.4 – *La règle* ($\#\pi$)

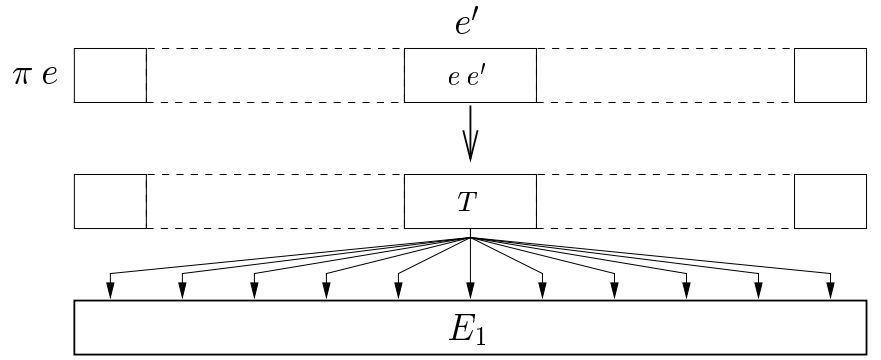

FIG. 2.5 – *La règle* ($\stackrel{e}{\rightarrow} T$)

l'initiative d'évaluer $e\,v$ donnant la valeur v'. Si $v' \equiv T$ (resp. F) alors le processeur v diffuse l'ordre d'évaluation globale de E_1 (resp. E_2); sinon le calcul échoue.

Les règles de contexte de BSλ sont nombreuses : toutes les équations précédentes peuvent être appliquées dans n'importe quel contexte.

2.2.3 BSλ_p

Le BSλ_p-calcul est une extension du BSλ-calcul dont les structures de données parallèles sont plates et correspondent physiquement aux processeurs. Les champs de données sont énumérés sur les noms de processeurs 0,...,$p-1$. Le terme BSλ_p πf devient ainsi $< f0,...,f(p-1); f >$. On ajoute f pour illustrer la possibilité de réduction parallèle.

Quelques modifications pour préciser les échanges des valeurs pour l'instruction *put* [11]
Ancien put du BSλ_p:

$$! < f_0,...,f_{p-1} >=< ...,\lambda j.(in_N\, j \rightarrow (t_{\|j\|}\, i),\mathbf{nc}),...$$

avec $in_{\mathcal{N}}$ tel que $in_{\mathcal{N}} n \rightarrow true$ pour chaque processeur n. Le terme $(c \rightarrow t1,t2)$ étant l'expression d'une conditionnelle.
Nouveau put :

$$! < f_0,...,f_{p-1} >=< ...,\underbrace{\lambda j.((j = 0 \rightarrow f_0 i,(j = 1 \rightarrow f_1 i,(...,nc))...)}_{proc\ i},... >$$

2.3 BSMLlib

BSML est un langage purement fonctionnel de données parallèles conçu pour programmer des algorithmes BSP. La BSMLlib est l'implantation du formalisme BSλ.

Le BSML est basé sur :

1. une variable `nprocs` liée à l'extérieur du programme dont la valeur est p, le nombre statique de processeurs.
2. un constructeur de types polymorphe `Par` tel que `'a Par` représente le type des vecteurs parallèles de largeur p contenant des valeurs de type `'a`, une par processeur. L'imbrication de types `Par` est interdite et un compilateur pourrait vérifier que cette restriction est respectée. Ceci améliore par exemple DPML/Caml Flight [7, 5] dans lequel la structure de contrôle parallèle globale `sync` était empêchée *dynamiquement* de s'imbriquer [6].
3. des fonctions `mkpar`, `apply`, `get`, `put` et une conditionnelle `ifat` décrites ci-après.

Les valeurs parallèles sont créées par

```
mkpar: (int ->'a) ->'a Par
```

qui est telle que `(mkpar f)` contient `(f i)` sur le processeur `i`$\in \{0,\ldots,(p-1)\}$.

Un algorithme BSP est exprimé par la combinaison de calculs locaux asynchrones et de phases de communications globales avec des synchronisations globales. Nous n'exprimons les communications point à point que collectivement entre tous les processeurs et ignorons la distinction entre la demande d'une communication et sa réalisation à la barrière de synchronisation comme c'est le cas avec BSPlib.

Les phases de calculs asynchrones sont programmées par :

```
apply: ('a ->'b)Par ->'a Par ->'b Par
```

qui est telle que `apply (mkpar f) (mkpar e)` contient `(f i) (e i)` au processeur i.

Les phases de communications et synchronisations sont exprimées par :

```
get: 'a Par ->(int list Par) ->(Hash(int,'a))Par
```

telle que `(get(mkpar x)(mkpar y))` contient `{(j, x j) | j in (y i)}` au processeur i (si nous identifions une table de hachage à un ensemble).

Il y a également une opération `put` duale de `get` :

```
put: (int * 'a) list Par ->(Hash(int,'a))Par
```

les processeurs indiquent les processeurs destinations de leurs données locales. `put` permet d'économiser des barrières de synchronisation dans certains algorithmes.

Le langage contient également une conditionnelle synchrone `ifat` telle que `ifat (v,i,v1,v2)` s'évalue à `v1` ou `v2` selon la valeur de `v` au processeur `i`. Son "type" est :

```
ifat: bool Par * int * 'a Par * 'a Par ->'a Par
```

Ocaml étant un langage strict, la conditionnelle globale ne peut être implantée comme une fonction. La librairie `BSMLlib` contient en fait une fonction

```
at: bool Par ->int ->bool
```

à utiliser avec la conditionnelle classique de Ocaml :

```
if (at v i) then v1 else v2
```

Un langage BSML pourrait réaliser `ifat` par une construction propre.

2.4 Micro-ML

L'étude du mini-ML (un petit sous-ensemble des langages de la famille ML) formalisé par Clement et al. [3] met en évidence une sémantique dynamique qui nous sert pour établir celle du micro-ML. Le micro-ML sera ensuite étendu pour créer le BSλ simplement typé.

Pour ce micro-ML, la grammaire des termes locaux est la suivante (cette grammaire sera étendue Chapitre 3 pour prendre en compte les opérateurs parallèles) :

$e ::= x \mid n \mid b \mid$ `fun` $(x : t) \to e \mid$ `let` $(x : t) = e$ `in` e
$\mid$ `letrec` $(f : t \to t')\ x = e$ `in` $e \mid$ `if` e `then` e `else` $e \mid e = e \mid e\ e$

La grammaire des valeurs est la suivante :
$val ::= c \mid \lambda x.e$

Voici la sémantique pour les termes locaux du micro-ml :

A gauche de $\triangleright$, l'expression micro-ml ; à droite, la valeur évaluée pour cette expression.

$$[\texttt{CONST}]\ \frac{}{c \triangleright c}$$

$$[\texttt{FUN}]\ \frac{}{\texttt{fun}\ (x : t) \to e \triangleright\ \texttt{fun}\ (x : t) \to e}$$

$$[\texttt{LET}]\ \frac{e_1 \triangleright v_1 \quad e_2[x \leftarrow v_1] \triangleright v}{\texttt{let}\ (x : t) = e_1\ \texttt{in}\ e_2 \triangleright v}$$

$$[\texttt{REC}]\ \frac{e(\texttt{Rec}\ e)x \triangleright v}{(\texttt{Rec}\ e)x \triangleright v}$$

`REC` est un opérateur de point fixe qui associe à une fonctionnelle F ayant le type $(t \to t)$ un point fixe noté `Rec F` de type t c'est-à-dire une valeur ayant la propriété `Rec F = F (Rec F)`. Une application de la forme `Rec e` sera sa propre valeur et ne donnera lieu à une évaluation que lorsqu'elle sera appliquée à un argument supplémentaire. De cette façon, le calcul infini sera évité.

$$[\texttt{LETREC}]\ \frac{e_2[f \leftarrow \texttt{Rec}(\texttt{fun}(f : t \to t') \to \texttt{fun}(x : t) \to e_1)] \triangleright v}{\texttt{letrec}\ (f : t \to t')\ x = e_1\ \texttt{in}\ e_2 \triangleright v}$$

$$[\texttt{IFT}]\ \frac{e_1 \triangleright\ \texttt{true} \quad e_2 \triangleright v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \triangleright v}$$

$$\texttt{[IFF]}\ \frac{e_1 \rhd\ \texttt{false} \quad e_3 \rhd v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rhd v}$$

$$\texttt{[EQ]}\ \frac{e_1 \rhd v \quad e_2 \rhd v}{e_1 = e_2 \rhd\ \texttt{true}}$$

$$\texttt{[NEQ]}\ \frac{e_1 \rhd v_1 \quad e_2 \rhd v_2 \quad v_1 \neq v_2}{e_1 = e_2 \rhd\ \texttt{false}}$$

$$\texttt{[APP]}\ \frac{e_1 \rhd\ \texttt{fun}\ \ (x:t) \to e \quad e_2 \rhd v_2 \quad e[x \leftarrow v_2] \rhd v}{e_1 e_2 \rhd v}$$

2.4.1 Exemple

Trace de l'évaluation d'un terme utilisant une fonction factorielle :

A gauche du : le terme BSλ simplement typé ; à droite son evaluation.
Terme valuer :

```
letrec (f:int->int) x= if x<=0 then 1 else x*(f (x-1)) in f 2;;


fun (x:int) -> fun (f:(int -> int)) -> if x<=0 then 1 else x*(f) (x-1) : g
   o g=fun (x:int) -> fun (f:(int -> int)) -> if x<=0 then 1 else x*(f) (x-1)
2 : 2

fun (f:(int -> int)) -> if 2<=0 then 1 else 2*(f) (2-1) :
    fun (f:(int -> int)) -> if 2<=0 then 1 else 2*(f) (2-1)
(g) (2) : fun (f:(int -> int)) -> if 2<=0 then 1 else 2*(f) (2-1)
rec (g) : rec (g)
0 : 0
2 : 2
2<=0 : false
g : g
1 : 1
2 : 2
2-1 : 1
fun (f:(int -> int)) -> if 1<=0 then 1 else 1*(f) (1-1) :
    fun (f:(int -> int)) -> if 1<=0 then 1 else 1*(f) (1-1)
(g) (2-1) : fun (f:(int -> int)) -> if 1<=0 then 1 else 1*(f) (1-1)
rec (g) : rec (g)
0 : 0
1 : 1
1<=0 : false
g : g
1 : 1
1 : 1
1-1 : 0
```

```
fun (f:(int -> int)) -> if 0<=0 then 1 else 0*(f) (0-1) :
    fun (f:(int -> int)) -> if 0<=0 then 1 else 0*(f) (0-1)
(g) (1-1) : fun (f:(int -> int)) -> if 0<=0 then 1 else 0*(f) (0-1)
rec (g) : rec (g)
0 : 0
0 : 0
0<=0 : true
1 : 1

if 0<=0 then 1 else 0*(rec (g)) (0-1) : 1
((g) (1-1)) (rec (g)) : 1
(rec (g)) (1-1) : 1
1 : 1
1*(rec (g)) (1-1) : 1
if 1<=0 then 1 else 1*(rec (g)) (1-1) : 1
((g) (2-1)) (rec (g)) : 1
(rec (g)) (2-1) : 1
2 : 2
2*(rec (g)) (2-1) : 2
if 2<=0 then 1 else 2*(rec (g)) (2-1) : 2
((g) (2)) (rec (g)) : 2
(rec (g)) (2) : 2

letrec (f:(int -> int)) x = if x<=0 then 1 else x*(f) (x-1) in (f) (2) : 2
```

2.5 SECD machine

La SECD machine a été introduite par Landin [9] et utilisée par Henderson [8] pour implanter une variante du langage Lisp de manière portable et entièrement documentée.
Son nom provient de ses 4 registres :

- S : la pile (stack), contient les résultats intermédiaires durant le calcul des valeurs des expressions.
- E : l'environnement, contient les valeurs liées aux variables pendant l'évaluation.
- C : la liste de contrôle , contient le programme en langage machine en cours d'exécution.
- D : la sauvegarde (dump), sauvegarde les valeurs des autres registres lors d'appel de fonction.

Nous allons l'utiliser pour réaliser une machine pour implanter notre langage BSλ simplement typé.

2.5.1 La machine

Une transition représente l'exécution de la commande au sommet de la pile c, avec l'environnement e, les valeurs dans la pile s et une sauvegarde dans d.

Les transitions de la SECD machine pour les termes locaux sont présentées Figure 2.6.
On s'interesse plus particulièrement aux numéros 8 et 9 qui concernent la

récursivité.
La transition 8 met la valeur $\texttt{RECLO}(f,x,C,E)$ au sommet de la pile de valeurs,
La transition 9 agit à la manière de la sématique avec l'opérateur REC
8: $(S,E,\texttt{REC}(f,x,C') :: C,D) \Rightarrow (\texttt{RECLO}(f,x,C',E) :: S,E,C,D)$
La commande $\texttt{REC}(f,x,C')$ est l'amorce de la récursion de la fonction f. Elle met la fermeture récursive de f en sommet des piles de valeurs, sans changer ni l'environnement, ni le dump.
9: $(v :: \texttt{RECLO}(f,x,C',E') :: S,E,\texttt{APP} :: C,D) \Rightarrow (v :: \texttt{CLO}(x,C',(f,\texttt{RECLO}(f,x,C',E')) :: E') :: S,E,\texttt{APP} :: C,D)$
La commande APP avec une valeur v en sommet de la pile s et la valeur $\texttt{RECLO}(f,x,C',E')$ permet l'application récursive de la fonction. Elle échange la fermeture récursive par la fermeture classique dans laquelle f est associée à la fermeture récursive.

2.5.2 Compilation

La fonction de compilation $[\![]\!] : e \rightarrow (\texttt{Com})^*$ pour les termes locaux est décrite par la Figure 2.7.
Les valeurs sont : $\texttt{Com} ::= \texttt{FUN} \mid \texttt{CLO} \mid \texttt{REC} \mid \texttt{RECLO} \mid \texttt{IF} \mid \texttt{EQ}$
CLO est la fermeture de fonction.
REC est l'opérateur de récursivité.
RECLO est la fermeture de fonction récursive.

2.5.3 Exemple

La fonction factorielle :
$\texttt{letrec}\ (f : \texttt{int} \rightarrow \texttt{int})x = (\ \texttt{if}\ \ x = 0\ \texttt{then}\ 1\ \texttt{else}\ (x*(f(x-1))))\ \texttt{in}\ f\ 3$
Compilation :
$[\![\texttt{letrec}(f : \texttt{int} \rightarrow \texttt{int})x = (\ \texttt{if}\ \ x = 0\ \texttt{then}\ 1\ \texttt{else}\ (x*(f(x-1))))\ \texttt{in}\ f\ 3]\!]$
$= \texttt{FUN}(f,[\![f\ 3]\!]) :: \texttt{REC}(f,x,[\![(\ \texttt{if}\ \ x = 0\ \texttt{then}\ 1\ \texttt{else}\ (x*(f(x-1))))]\!]) :: \texttt{APP}$
$= \texttt{FUN}(f,f :: 3 :: \texttt{APP}) :: \texttt{REC}(f,x,[\![x = 0]\!] :: \texttt{IF}([\![1]\!],[\![(x*(f(x-1)))]\!])) :: \texttt{APP}$
$= \texttt{FUN}(f,f :: 3 :: \texttt{APP}) :: \texttt{REC}(f,x,x :: 0 ::=:: \texttt{IF}(1,x :: [\![(f(x-1))]\!] :: *)) :: \texttt{APP}$

$= \texttt{FUN}(f,f :: 3 :: \texttt{APP}) :: \texttt{REC}(f,x,x :: 0 ::=:: \texttt{IF}(1,x :: f :: [\![(x-1)]\!] :: \texttt{APP} :: *)) :: \texttt{APP}$

$= \texttt{FUN}(f,f :: 3 :: \texttt{APP}) :: \texttt{REC}(f,x,x :: 0 ::=:: \texttt{IF}(1,x :: f :: x :: 1 :: - :: \texttt{APP} :: *)) :: \texttt{APP}$

Execution (Figure 2.8):
Quelques notations pour alléger le tableau d'exécution :
$[\![e]\!] = x :: 0 ::=:: \texttt{IF}(1,x :: f :: x :: 1 :: - :: \texttt{APP} :: *)$
$E_1 = (f,\texttt{RECLO}(f,x,[\![e]\!],E))$
$D_0 = (E,C)$
$D_1 = ((f,\texttt{RECLO}(f,x,[\![e]\!],E)) :: E,[])$
$D_2 = ((x,3) :: (f,\texttt{RECLO}(f,x,[\![e]\!],E)) :: E,*)$
$D_3 = ((x,2) :: (f,\texttt{RECLO}(f,x,[\![e]\!],E)) :: E,*)$
$D_4 = ((x,1) :: (f,\texttt{RECLO}(f,x,[\![e]\!],E)) :: E,*)$

	AVANT				APRES			
	S	**E**	**C**	**D**	**S**	**E**	**C**	**D**
1	S	E	$[]$	$(E',C') :: D$	S	E'	C'	D
2	S	E	$x :: C$	D	$E(x) :: S$	E	C	D
3	S	E	$n :: C$	D	$n :: S$	E	C	D
4	S	E	$b :: C$	D	$b :: S$	E	C	D
5	S	E	$\texttt{FUN}(x,C') :: C$	D	$\texttt{CLO}(x,C',E) :: S$	E	C	D
6	$v :: \texttt{CLO}(x,C',E') :: S$	E	$\texttt{APP} :: C$	D	S	$(x,v) :: E'$	C'	$(E,C) :: D$
7	$v_1 :: v_2 :: S$	E	$\oplus :: C$	D	$(v_1 \oplus v_2) :: S$	E	C	D
8	S	E	$\texttt{REC}(f,x,C') :: C$	D	$\texttt{RECLO}(f,x,C',E) :: S$	E	C	D
9	$v :: \texttt{RECLO}(f,x,C',E') :: S$	E	$\texttt{APP} :: C$	D	$v :: \texttt{CLO}(x,C',(f,\texttt{RECLO}(f,x,C',E')) :: E') :: S$	E	$\texttt{APP} :: C$	D
10	$\texttt{true} :: S$	E	$(\texttt{IF}(C_1,C_2)) :: C$	D	S	E	$C_1 @ C$	D
11	$\texttt{false} :: S$	E	$(\texttt{IF}(C_1,C_2)) :: C$	D	S	E	$C_2 @ C$	D

FIG. 2.6 – *Transitions de la secd*

BSλ simplement typé	Commandes
c	c
$e_1 \oplus e_2$	$[\![e_1]\!] :: [\![e_2]\!] :: \oplus$
$\texttt{fun}(x : t) \to e$	$\texttt{FUN}(x,[\![e]\!])$
$\texttt{let } (x : t) = e_1 \texttt{ in } e_2$	$\texttt{FUN}(x,[\![e_2]\!]) :: [\![e_1]\!] :: \texttt{APP}$
$\texttt{letrec } (f : t)\ x = e_1 \texttt{ in } e_2$	$\texttt{FUN}(f,[\![e_2]\!]) :: \texttt{REC}(f,x,[\![e_1]\!]) :: \texttt{APP}$
$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$	$[\![e_1]\!] :: \texttt{IF}([\![e_2]\!],[\![e_3]\!])$
$e_1 = e_2$	$[\![e_1]\!] :: [\![e_2]\!] :: \texttt{EQ}$
$e_1 e_2$	$[\![e_1]\!] :: [\![e_2]\!] :: \texttt{APP}$

FIG. 2.7 – *compilation*

S	E	$\mathtt{FUN}(f,f :: 3 :: \mathtt{APP}) :: \mathtt{REC}(f,x,\llbracket e\rrbracket) :: \mathtt{APP} :: C$	D
$\mathtt{CLO}(f,f :: 3 :: \mathtt{APP},E) :: S$	E	$\mathtt{REC}(f,x,\llbracket e\rrbracket) :: \mathtt{APP} :: C$	D
$\mathtt{RECLO}(f,x,\llbracket e\rrbracket,E) :: \mathtt{CLO}(f,f :: 3 :: \mathtt{APP},E) :: S$	E	$\mathtt{APP} :: C$	D
S	$(f,\mathtt{RECLO}(f,x,\llbracket e\rrbracket,E)) :: E$	$f :: 3 :: \mathtt{APP}$	$D_0 :: D$
$\mathtt{RECLO}(f,x,\llbracket e\rrbracket,E) :: S$	$E_1 :: E$	$3 :: \mathtt{APP}$	$D_0 :: D$
$3 :: \mathtt{RECLO}(f,x,\llbracket e\rrbracket,E) :: S$	$E_1 :: E$	$\mathtt{APP}$	$D_0 :: D$
$3 :: \mathtt{CLO}(x,\llbracket e\rrbracket,E_1 :: E) :: S$	$E_1 :: E$	$\mathtt{APP}$	$D_0 :: D$
S	$(x,3) :: E_1 :: E$	$\llbracket e\rrbracket$	$D_1 :: D_0 :: D$
S	$(x,3) :: E_1 :: E$	$x :: 0 ::=:: \mathtt{IF}(1,x :: f :: x :: 1 :: - :: \mathtt{APP} :: *)$	$D_1 :: D_0 :: D$
..			
$2 :: \mathtt{RECLO}(f,x,\llbracket e\rrbracket,E) :: 3 :: S$	$(x,3) :: E_1 :: E$	$\mathtt{APP} :: *$	$D_1 :: D_0 :: D$
$2 :: \mathtt{CLO}(x,\llbracket e\rrbracket,E_1 :: E) :: 3 :: S$	$(x,3) :: E_1 :: E$	$\mathtt{APP} :: *$	$D_1 :: D_0 :: D$
$3 :: S$	$(x,2) :: E_1 :: E$	$\llbracket e\rrbracket$	$D_2 :: D_1 :: D_0 :: D$
..			
$1 :: \mathtt{RECLO}(f,x,\llbracket e\rrbracket,E) :: 2 :: 3 :: S$	$(x,2) :: E_1 :: E$	$\mathtt{APP} :: *$	$D_2 :: D_1 :: D_0 :: D$
$1 :: \mathtt{CLO}(x,\llbracket e\rrbracket,E_1 :: E) :: 2 :: 3 :: S$	$(x,2) :: E_1 :: E$	$\mathtt{APP} :: *$	$D_2 :: D_1 :: D_0 :: D$
$1 :: \mathtt{CLO}(x,\llbracket e\rrbracket,E_1 :: E) :: 2 :: 3 :: S$	$(x,2) :: E_1 :: E$	$\mathtt{APP} :: *$	$D_2 :: D_1 :: D_0 :: D$
$1 :: \mathtt{CLO}(x,\llbracket e\rrbracket,E_1 :: E) :: 2 :: 3 :: S$	$(x,2) :: E_1 :: E$	$\mathtt{APP} :: *$	$D_2 :: D_1 :: D_0 :: D$
$2 :: 3 :: S$	$(x,1) :: E_1 :: E$	$\llbracket e\rrbracket$	$D_3 :: D_2 :: D_1 :: D_0 :: D$
..			
$0 :: \mathtt{RECLO}(f,x,\llbracket e\rrbracket,E) :: 1 :: 2 :: 3 :: S$	$(x,1) :: E_1 :: E$	$\mathtt{APP} :: *$	$D_3 :: D_2 :: D_1 :: D_0 :: D$
$0 :: \mathtt{CLO}(x,\llbracket e\rrbracket,E_1 :: E) :: 1 :: 2 :: 3 :: S$	$(x,1) :: E_1 :: E$	$\mathtt{APP} :: *$	$D_3 :: D_2 :: D_1 :: D_0 :: D$
$1 :: 2 :: 3 :: S$	$(x,0) :: E_1 :: E$	$\llbracket e\rrbracket$	$D_4 :: D_3 :: D_2 :: D_1 :: D_0 :: D$
$1 :: 2 :: 3 :: S$	$(x,0) :: E_1 :: E$	$x :: 0 ::=:: \mathtt{IF}(1,x :: f :: x :: 1 :: - :: \mathtt{APP} :: *)$	$D_4 :: D_3 :: D_2 :: D_1 :: D_0 :: D$
..			
$1 :: 1 :: 2 :: 3 :: S$	$(x,0) :: E_1 :: E$		$D_4 :: D_3 :: D_2 :: D_1 :: D_0 :: D$
$1 :: 1 :: 2 :: 3 :: S$	$(x,1) :: E_1 :: E$	$*$	$D_3 :: D_2 :: D_1 :: D_0 :: D$
$1 :: 2 :: 3 :: S$	$(x,1) :: E_1 :: E$		$D_3 :: D_2 :: D_1 :: D_0 :: D$
$1 :: 2 :: 3 :: S$	$(x,2) :: E_1 :: E$	$*$	$D_2 :: D_1 :: D_0 :: D$
$2 :: 3 :: S$	$(x,2) :: E_1 :: E$		$D_2 :: D_1 :: D_0 :: D$
$2 :: 3 :: S$	$(x,3) :: E_1 :: E$	$*$	$D_1 :: D_0 :: D$
$6 :: S$	$(x,3) :: E_1 :: E$		$D_1 :: D_0 :: D$
$6 :: S$	$E_1 :: E$		$D_0 :: D$
$6 :: S$	E	C	D

FIG. 2.8 – *Exemple : factorielle*

Chapitre 3

BSλ simplement typé

Le BSλ simplement typé est une extension du micro-ML présenté précédement (Section 2.4.1)à l'aide des opérations parallèles du BSλ-calcul. La différence essentielle est dans le typage des termes.

Pour traiter le parallélisme, il faut introduire de nouveaux termes.

3.1 Grammaires

– Termes
L'ensemble des termes ajoutés pour le parallélisme est donné par la grammaire suivante :

1. Termes locaux :
   $e ::=$ `is-nc` e
   `is-nc` est un symbole-fonction pour une fonction qui retourne un booléen : test d'égalité avec la constante polymorphe nc.

2. Termes globaux :
   $e ::=$ `mkpar` $e \mid$ `apply` $e\ e$
   $\mid$ `if` e `at` e `then` e `else` e
   $\mid$ `get` $e\ e \mid$ `put` e

   `mkpar` est le contructeur paralèle. Il permet d'appliquer l'expression e à tous les processeurs.
   `apply` est l'applicateur parallèle. Il permet l'application point à point.
   `if ... at ...` est la conditionnelle globale.
   `get` est le multi-get, un échange collectif de données avec synchronisation (reception de messages).
   `put` est le put. C'est un complément du get en envoi de message.

3. Les termes suivants peuvent être locaux ou globaux :
   $e ::=$ `fun` $(x : t) \rightarrow e \mid$ `let` $(x : t) = e$ `in` e
   $\mid$ `letrec` $(f : t \rightarrow t')\ x = e$ `in` e
   $\mid$ `if` e `then` e `else` $e \mid e\ e$
   Pour les termes `fun`, `let` et `letrec` on ajoute le type de la variable

pour permettre les restrictions vis à vis de l'application.

- Constantes
  $c ::=$ `nprocs` | `nc`

  nprocs est le nombre de processeurs, nc est la valeur "no communication" pour les instructions d'échanges.

- Types
  Pour respecter les sortes global et local, on introduit la sorte $\dot{t}$ pour les types "locaux" et la sorte $\bar{t}$ pour les types "globaux". Les types acceptables sont décrits par les grammaires suivantes :
  $\dot{t} ::=$ `int` | `bool` $\mid \dot{t} \to \dot{t}$
  $\bar{t} ::= \dot{t}$ `par` $\mid \dot{t} \to \bar{t} \mid \bar{t} \to \bar{t}$
  $t ::= \dot{t} \mid \bar{t}$
  Le type de forme $\bar{t} \to \dot{t}$ n'est pas acceptable.

- Valeurs
  A la grammaire de valeurs définies dans la section 2.4.1 ($val ::= c \mid \lambda x.e$)
  On ajoute les valeurs vectorielles : $< val,...,val >$
  On a ainsi comme grammaire de valeurs : $val_{\triangleright} ::= c \mid \lambda x.e \mid < val,...,val >$

3.2 Opérations parallèles

- MKPAR
  Le terme `mkpar` e représente un vecteur dont les valeurs sont données par la fonction e appliquée à chaque numéro de processeur.
- APPLY
  Le terme `apply` $e_1 e_2$ représente l'application point a point des fonctions e_1 aux valeurs e_2
- IFAT
  Le terme `if` e_1 `at` e_2 `then` e_3 `else` e_4 représente la conditionnelle globale. Si la valeur du vecteur e_1 en e_2 est `true` alors la valeur résultante est donnée par e_3 sinon par e_4.
- GET
  Le terme `get` $< v_0,...,v_{p-1} >< f_0,...,f_{p-1} >$ représente le multi-échange de données. Chaque f_i est une fonction des noms de processeurs vers les valeurs booléennes `true` ou `false`. Si $f_i j =$ `true` alors le processeur i recevra le terme e_j, si $j \in \{0,...,p-1\}$. Sinon le processeur i ne recevra aucun message du processeur j et la valeur au processeur i sera une fonction qui appliquée à j donnera la valeur `nc`.
  Il permet aux processeurs de recevoir plusieurs messages différents.
- PUT
  Le terme `put` $< f_0,...,f_{p-1} >$ permet d'envoyer plusieurs messages différents à différents processeurs en une seule super-étape. Si $f_i j =$ `nc` , il n'y a pas de communication. Sinon la valeur $f_i j$ est envoyée au processeur j par le processeur i, à condition que $i \in \{0,...,p-1\}$.

3.3 Sémantique naturelle

Cette sémantique d'évaluation sert de définition à notre langage. Ses jugements sont de la forme $e \rhd v$ où $v \in val \cup val_{\rhd}$. A gauche de $\rhd$, l'expression BSλ simplement typé; à droite, la valeur évaluée pour cette expression.

$$\texttt{[NPROCS]}\ \frac{}{\texttt{nprocs}\ \rhd p}$$

où p est une constante (un nombre naturel $\geq$ définie extérieurement (il y a ainsi une sémantique du langage par valeur positive de p)

$$\texttt{[ISNCT]}\ \frac{e \rhd nc}{\texttt{is\_nc e}\ \rhd\ \texttt{true}}$$

$$\texttt{[ISNCF]}\ \frac{e \rhd v \neq nc}{\texttt{is\_nc e}\ \rhd\ \texttt{false}}$$

$$\texttt{[FUN]}\ \frac{}{\texttt{fun}\ (x:t) \to e \rhd\ \ \texttt{fun}\ \ (x:t) \to e}$$

$$\texttt{[LET]}\ \frac{e_1 \rhd v_1 \quad e_2[x \leftarrow v_1] \rhd v}{\texttt{let}\ \ (x:t) = e_1\ \ \texttt{in}\ \ e_2 \rhd v}$$

$$\texttt{[REC]}\ \frac{e(\texttt{Rec}\ e)x \rhd v}{(\texttt{Rec}\ e)x \rhd v}$$

$$\texttt{[LETREC]}\ \frac{e_2[f \leftarrow \texttt{Rec}(\texttt{fun}(f : t \to t') \to \texttt{fun}(x:t) \to e_1)] \rhd v}{\texttt{letrec}\ \ (f : t \to t')\ x = e_1\ \ \texttt{in}\ \ e_2 \rhd v}$$

$$\texttt{[IFT]}\ \frac{e_1 \rhd\ \ \texttt{true} \quad\ e_2 \rhd v}{\texttt{if}\ \ e_1\ \ \texttt{then}\ \ e_2\ \ \texttt{else}\ \ e_3 \rhd v}$$

$$\texttt{[IFF]}\ \frac{e_1 \rhd\ \ \texttt{false} \quad\ e_3 \rhd v}{\texttt{if}\ \ e_1\ \ \texttt{then}\ \ e_2\ \ \texttt{else}\ \ e_3 \rhd v}$$

$$\texttt{[EQ]}\ \frac{e_1 \rhd v \quad e_2 \rhd v}{e_1 = e_2 \rhd\ \ \texttt{true}}$$

$$\texttt{[NEQ]}\ \frac{e_1 \rhd v_1 \quad e_2 \rhd v_2 \quad v_1 \neq v_2}{e_1 = e_2 \rhd\ \ \texttt{false}}$$

$$\texttt{[APP]}\ \frac{e_1 \rhd\ \ \texttt{fun}\ \ (x:t) \to e \quad e_2 \rhd v_2 \quad e[x \leftarrow v_2] \rhd v}{e_1 e_2 \rhd v}$$

$$\texttt{[MKPAR]}\ \frac{e_1 \rhd\ \ \texttt{fun}\ \ (x:t) \to e \qquad \forall i\ e[x \leftarrow i] \rhd v_i}{\texttt{mkpar}\ \ e_1 \rhd < v_0,\ldots,v_{p-1} >}$$

La fonction représentée par e_1 est appliquée en chaque point au numéro du processeur.

$$\texttt{[APPLY]}\ \frac{e_1 \rhd < \texttt{fun}(x_0 : t) \to u_0,\ldots,\texttt{fun}(x_{p-1} : t) \to u_{p-1} > \quad e_2 \rhd < w_0,\ldots,w_{p-1} > \quad \forall i\ u_i[x_i \leftarrow w_i] \rhd v_i}{\texttt{apply}\ e_1 e_2 \rhd < v_0,\ldots,v_{p-1} >}$$

Chaque fonction $\texttt{fun}\ (x_i : int) \to u_i$ de e_i est appliquée à la valeur w_i de e_2.

$$\texttt{[IFATT]}\ \frac{e_1 \rhd < v_0,\ldots,v_{p-1} > \quad e_2 \rhd n \quad v_n = \texttt{true} \quad e_3 \rhd v}{\texttt{if}\ \ e_1\ \ \texttt{at}\ \ e_2\ \texttt{then}\ e_3\ \texttt{else}\ e_4 \rhd v}$$

$$\texttt{[IFATF]}\ \frac{e_1 \rhd < v_0,\ldots,v_{p-1} > \quad e_2 \rhd n \quad v_n = \texttt{false} \quad e_4 \rhd v}{\texttt{if}\ \ e_1\ \ \texttt{at}\ \ e_2\ \texttt{then}\ e_3\ \texttt{else}\ e_4 \rhd v}$$

La valeur de e_2 est évaluée en chaque processeur pour déterminer quelle valeur booléenne sera prise en compte pour le branchement.

$$\texttt{[GET]}\ \frac{e_1 \rhd < v_0,\ldots,v_{p-1} > \quad e_2 \rhd < f_0,\ldots,f_{p-1} > \quad \forall i \forall j\ f_i j \rhd b_{ij} \quad req(i) = \{j | b_{ij} = \texttt{true}\} = \{n_{i1},\ldots,n_{ik_i}\}}{\texttt{get}\ \ e_1\ e_2 \rhd < \ldots,\ \underbrace{\texttt{fun}\ n_{i1} \to v_{n_{i1}} | \ldots | n_{ik_i} \to v_{n_{ik_i}} | \_ \to nc}_{\texttt{lieu i}},\ldots >}$$

Chaque f_i s'applique à tous les numéros de processeurs j en une valeur booléene qui détermine à quels processeurs (j) la valeur v_i sera transmise.

$$\texttt{[PUT]}\ \frac{e \rhd < f_0,\ldots,f_{p-1} > \quad \forall i\ dest(i) = \{j | f_i j \rhd v_{ij} \neq nc\}}{\begin{array}{l}\texttt{put}\ e \rhd < \ldots,\ \underbrace{\begin{array}{llll}\texttt{fun} & n_{j1} & \to & v_{n_{j1}j} \\ | & & \ldots & \\ | & n_{jk_j} & \to & v_{n_{jk_j}j} \\ | & \_ & \to & nc\end{array}}_{\texttt{lieu j}},\ldots >\end{array}} \forall j.\{n_{j1},\ldots,n_{jk_j}\} = \{i | j \in dest(i)\}$$

Chaque f_i s'applique à tous les numéros de processeurs j. Si la valeur n'est pas `nc`, elle est transmise au processeur j, sinon elle ne l'est pas.

La forme $\texttt{fun}\ n_0 \to v_0 | \ldots | n_k \to v_k | \_ \to v_z$ correspond à la conditionnelle imbriquée :

```
fun (x:int)-> if x=n0 then v0
    else (if x=n1 then v1
    else (...
         (if x=nk then vk else vz)...))
```

3.3.1 Nouvelle sémantique naturelle avec environnement

Pour des soucis de preuve et de programmation, il nous semblait nécessaire d'avoir une sémantique dévaluation avec un environnement. Cet environnement est essentiellement utile pour les fonctions, puisque lors des échanges de fonctions, il est utile de conserver l'environnement des variables liées. On a ainsi une

description plus proche de la machine SECD cible. On a besoin de redéfinir les valeurs pour cette sémantique.
$val_E ::= c \mid [\lambda x.e,E]$ et $val_{\rhd_E} ::= c \mid [\lambda x.e,E] \mid < val_E,...,val_E >$

$$\texttt{[0]}\ \frac{E(x) = v}{x \rhd_E v}$$

$$\texttt{[FUN]}\ \frac{}{E \vdash\ \texttt{fun}\ \ (x : t) \to e \rhd_E [\texttt{fun}(x : t) \to e,E]}$$

$$\texttt{[NPROCS]}\ \frac{}{E \vdash\ \texttt{nprocs}\ \rhd_E p}$$

$$\texttt{[IS\_NC]}\ \frac{E \vdash e \rhd_E v \quad E \vdash (v =\ \texttt{nc}\ ) \rhd_E b}{E \vdash\ \texttt{Is\_nc}\ e \rhd_E b}$$

$$\texttt{[LET]}\ \frac{E \vdash e_1 \rhd_E v_1 \quad (x,v_1) :: E \vdash e_2 \rhd_E v}{E \vdash\ \texttt{let}\ (x : t) = e_1\ \texttt{in}\ e_2 \rhd_E v}$$

$$\texttt{[REC]}\ \frac{E \vdash e\ (\ \texttt{Rec}\ \ (f,e))x \rhd_E v}{E \vdash (\texttt{Rec}(f,e)x \rhd_E v}$$

$$\texttt{[LETREC]}\ \frac{(f,\texttt{Rec}(f,\texttt{fun}(f : t \to t') \to \texttt{fun}(x : t) \to e_1)) :: E \vdash e_2 \rhd_E v}{E \vdash\ \texttt{letrec}\ (f : t \to t')\ x = e_1\ \texttt{in}\ e_2 \rhd_E v}$$

Pour traiter la récursivité, la différence essentielle avec la sémantique précedente est qu'on associe la fonction récursive à sa valeur exprimée à l'aide de l'opérateur de récrusion `Rec` dans l'environnement.

$$\texttt{[IFT]}\ \frac{E \vdash e_1 \rhd_E\ \texttt{true} \quad E \vdash e_2 \rhd_E v}{E \vdash\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rhd_E v}$$

$$\texttt{[IFF]}\ \frac{E \vdash e_1 \rhd_E\ \texttt{false} \quad E \vdash e_3 \rhd_E v}{E \vdash\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rhd_E v}$$

$$\texttt{[EQ]}\ \frac{E \vdash e_1 \rhd_E v \quad E \vdash e_2 \rhd_E v}{E \vdash e_1 = e_2 \rhd_E\ \texttt{true}}$$

$$\texttt{[NEQ]}\ \frac{E \vdash e_1 \rhd_E v_1 \quad E \vdash e_2 \rhd_E v_2 \quad v_1 \neq v_2}{E \vdash e_1 = e_2 \rhd_E\ \texttt{false}}$$

$$\texttt{[APP]}\ \frac{E \vdash e_1 \rhd_E [\texttt{fun}\ (x : t) \to e,E'] \quad E \vdash e_2 \rhd_E v_2 \quad (x,v_2) :: E' \vdash e \rhd_E v}{E \vdash e_1 e_2 \rhd_E v}$$

$$\texttt{[MKPAR]}\ \frac{E \vdash e_1 \rhd_E [\texttt{fun}(x : t) \to e,E'] \quad \forall i (x,i) :: E' \vdash e \rhd_E v_i}{E \vdash\ \texttt{mkpar}\ e_1 \rhd_E < v_0,...,v_{p-1} >}$$

$$\texttt{[APPLY]}\ \frac{\begin{array}{c} E \vdash e_1 \rhd_E < ...,[\texttt{fun}(x : t) \to u_i,E_i],... > \\ E \vdash e_2 \rhd_E < ...,w_i,... > \quad \forall i (x,w_i) :: E_i \vdash u_i \rhd_E v_i \end{array}}{\texttt{apply}\ e_1 e_2 \rhd_E < ...,v_i,... >}$$

$$\texttt{[IFATT]}\ \frac{E \vdash e_1 \rhd_E < v_0,...,v_{p-1} > \quad E \vdash e_2 \rhd_E n \quad v_n = \texttt{true} \quad E \vdash e_3 \rhd_E v}{E \vdash \texttt{if}\ e_1\ \texttt{at}\ e_2\ \texttt{then}\ e_3\ \texttt{else}\ e_4 \rhd_E v}$$

$$\texttt{[IFATF]}\ \frac{E \vdash e_1 \rhd_E < v_0,...,v_{p-1} > \quad E \vdash e_2 \rhd_E n \quad v_n = \texttt{false} \quad E \vdash e_4 \rhd_E v}{E \vdash \texttt{if}\ e_1\ \texttt{at}\ e_2\ \texttt{then}\ e_3\ \texttt{else}\ e_4 \rhd_E v}$$

Dans la règle [GET]:
si les valeurs de e_1 sont des fonctions où $v_i = [\ \texttt{fun}\ x \to c_i, E_i]$
alors $\forall x \in FV(c_i)$ renommer x en x.i : $c_i \to c'_i$, $E_i \to E'_i$ et $v_i \to v'_i$,
sinon $E_i = \emptyset$.

$$\texttt{[GET]}\ \frac{\begin{array}{c} E \vdash e_1 \rhd_E < ...,v_i,... > \quad E \vdash e_2 \rhd_E < ...,[f_i,F_i],... > \\ \forall i \forall j\ F_i \vdash f_i j \rhd_E b_{ij} \quad req(i) = \{j | b_{ij} = \texttt{true}\} = \{n_i,...,n_{i_{k_i}}\} \end{array}}{E \vdash\ \texttt{get}\ \ e_1\ e_2 \rhd_E < ..., \underbrace{[\ \texttt{fun}\ \ n_{i1} \to v'_{n_{i1}} | ... | n_{ik_i} \to v'_{n_{ik_i}} | \_ \to nc, \cup_{j \in n_{i1},...,n_{ik_i}} E'_j]}_{\texttt{lieu i}}, ... >}$$

Dans la règle du [PUT] :
si les valeurs de v_{ij} sont des fonctions où $v_{ij} = [\texttt{fun}\ x \to c_{ij}, E_{ij}]$ et $\forall ij, v_{ij} \neq nc$
alors $\forall x \in FV(c_{ij})$ renommer x en x.i : $c_{ij} \to c'_{ij}$, $E_{ij} \to E'_{ij}$ et $v_{ij} \to v'_i$,
sinon $E_{ij} = \emptyset$
Remarque : v_{ij} est la valeur envoyée à j par i

$$\texttt{[PUT]}\ \frac{\begin{array}{c} E \vdash e \rhd_E < ...,[f_i,F_i],... > \\ \forall i\ dest(i) = \{j | F_i \vdash f_i j \rhd_E v_{ij}, v_{ij} \neq nc\} \end{array}}{\texttt{put}\ e \rhd_E < \ldots, \underbrace{\begin{array}{llll} [\ \texttt{fun} & n_{j1} & \to & v'_{n_{j1}j} \\ | & & \ldots & \\ | & n_{jk_j} & \to & v'_{n_{jk_j}j} \\ | & \_ & \to & nc \end{array}}_{\texttt{lieu j}}, \cup_{i \in \{n_{j1},...,n_{jk_j}\}} E'_{ij}], \ldots >}$$

$$\texttt{tel que}\ \{n_{j1}, \ldots, n_{jk_j}\} = \{i | j \in dest(i)\}$$

Des problèmes de preuves nous ont amenés à conserver la première sémantique naturelle pour la suite.

Exemple

La fonction factorielle avec environnement.

```
letrec (f:int->int) x= if x<=0 then 1 else x*(f (x-1)) in f 2;;

f : rec (f,fun (x:int) -> fun (f:(int -> int)) -> if x<=0 then 1 else x*(f) (x-1))
2 : 2

g : [g,(f,rec (f,g));[]]
   o\‘u g=fun (x:int) -> fun (f:(int -> int)) -> if x<=0 then 1 else x*(f) (x-1)
2 : 2

h : [h,(x,2);(f,rec (f,g));[]]
  o\‘u h=fun (f:(int -> int)) -> if x<=0 then 1 else x*(f) (x-1)

(g) (2) : [h,(x,2);(f,rec (f,g));[]]
rec (f,g) : rec (f,g) \\ x : 2 \\

0 : 0 \\ x<=0 : false \\ x : 2 \\ f : rec (f,g) \\ x : 2 \\ 1 : 1 \\ x-1 : 1 \\
g : [g,(f,rec (f,g));(x,2);(f,rec (f,g));[]]
1 : 1
h : [h,(x,1);(f,rec (f,g));(x,2);(f,rec (f,g));[]]
(g) (1) : [h,(x,1);(f,rec (f,g));(x,2);(f,rec (f,g));[]]
rec (f,g) : rec (f,g)

x : 1 \\ 0 : 0 \\ x<=0 : false \\ x : 1 \\ f : rec (f,g) \\ x : 1 \\ 1 : 1 \\ x-1 : 0 \\
g : [g,(f,rec (f,g));(x,1);(f,rec (f,g));(x,2);(f,rec (f,g));[]]
0 : 0
h : [h,(x,0);(f,rec (f,g));(x,1);(f,rec (f,g));(x,2);(f,rec (f,g));[]]
(g) (0) : [h,(x,0);(f,rec (f,g));(x,1);(f,rec (f,g));(x,2);(f,rec (f,g));[]]
rec (f,g) : rec (f,g)

x : 0 \\ 0 : 0 \\ x<=0 : true \\ 1 : 1 \\
if x<=0 then 1 else x*(f) (x-1) : 1
((g) (0)) (rec (f,g)) : 1
(f) (x-1) : 1
x*(f) (x-1) : 1
if x<=0 then 1 else x*(f) (x-1) : 1
((g) (1)) (rec (f,g)) : 1
(f) (x-1) : 1
x*(f) (x-1) : 2
if x<=0 then 1 else x*(f) (x-1) : 2
((g) (2)) (rec (f,g)) : 2
(f) (2) : 2

letrec (f:(int -> int)) x = if x<=0 then 1 else x*(f) (x-1) in (f) (2) : 2
```

3.3.2 Correction de la sémantique naturelle du $BS\lambda$ simplement typé par rapport au $BS\lambda_p$

Il s'agit de montrer que les règles d'évalutation de la sémantique naturelle respectent l'équivalence BSλ_p. Pour cela il faut traduire certains termes et on suppose les égalités syntaxiques suivantes :

$BS\lambda$ simplement typé	$BS\lambda_p$
nprocs	p
c	c
fun x $\rightarrow$ e	$\lambda x.e$
let (x:t)=e_1 in e_2	$(\lambda x.e_2)e_1$
letrec (f:t) x = e_1 in e_2	(λ f e_2) (Y (λ f x.e_1))
Rec	Y
if e_1 then e_2 else e_3	$e_1 \rightarrow e_2,e_3$
mkpar	$\lambda f. < f0,f1,...,f(p-1) >$
apply	# (infixe)
if e_1 at e_2 then e_3 else e_4	$e_1 \stackrel{e_3}{\rightarrow} e_3,e_4$
get	?? (infixe)
put	!

Theorème 1 *Si $e \triangleright v$ alors $e =_{BS\lambda_p} v$ où e est une expression close.*

Preuve

On suppose par induction que l'évaluation des sous-expressions est correcte relativement à BSλ_p.

- Cas des constantes
  Soient $e = c$ une expression du BSλ simplement typé et $v = c$ une valeur dans BSλ_p
  La seule règle applicable à e est [CONT] donc $e \triangleright v$. Ces expressions sont égales donc on a bien e=v.

- Cas fonctionnel
  Soient $e = (\texttt{fun}(x : t) \rightarrow g)$ une expression du BSλ simplement typé et $v = \texttt{fun}(x : t) \rightarrow e$ une valeur dans BSλ_p
  La seule règle applicable à e est [FUN] donc $e \triangleright v$.
  Ces expressions sont égales donc on a bien e=v.

- Cas du let
  Correction de $e =$ `let` $(x : t) = e_1$ `in` $e_2 \triangleright v$.
  La seule règle applicable à e est [LET] donc $e \triangleright v$
  Comme hypothèse d'induction on a $e_1 = v_1$
  et $e_2[x \leftarrow v_1] = v$ car on a $e \triangleright v$.
  Donc $e =$ (`let` $(x : t) = e_1$ `in` $e_2) = (\lambda x.e_2)e_1$
  $= e_2[x \leftarrow e_1]$ par $\beta - Reduction$
  $= e_2[x \leftarrow v_1]$ par hypothèse
  $= v$

– Cas de la récursivité
Correction de $e =$ `Rec` $e_1\ x \rhd v$
La seule règle applicable à e est [REC] donc $e \rhd v$.
Comme hypothèse d'induction, on a $e_1(\texttt{Rec}\ e_1)x = v = e_1(Y\ e_1)x$
D'où `Rec` $e_1\ x = (Y\ e)x$
Comme Y est l'opérateur de récursivité $(Y\ e)x = e(Y\ e)x = v$
Et ainsi $e = v$
Correction de $e =$ (`letrec` $(f : t \to t')\ x = e_1$ `in` $e_2) \rhd v$
La seule règle applicable à e est [LETREC] donc $e \rhd v$ Comme hypothèse d'induction on a :
$e_2[f \leftarrow \texttt{Rec}(\texttt{fun}(f : t \to t') \to \texttt{fun}(x : t) \to e_1)] = (\lambda f.e_2)(Y(\lambda f.(\lambda x.e_1)))$
et $e_2[f \leftarrow \texttt{Rec}(\texttt{fun}(f : t \to t') \to \texttt{fun}(x : t) \to e_1)] = v$
D'où `letrec` $(f : t \to t')\ x = (\lambda f.e_2)(Y(\lambda f.(\lambda x.e_1))) = v$

– Cas de la condition
Correction de $e =$ `if` e_1 `then` e_2 `else` $e_3 \rhd v$
Les seules règles applicables à e sont soit [IFT], soit [IFF] selon l'évaluation de e_1 donc $e \rhd v$
On suppose par induction que soit $e_1 =$ `true` , soit $e_1 =$ `false`
que $e_2 = v_2$
et que $e_3 = v_3$
Si $e_1 =$ `true` alors $v = v_2$ sinon $v = v_3$
On a $e \rhd v_2$ si $e_1 =$ `true` et $e \rhd v_3$ si $e_1 =$ `false` .

D'où $e =$ `if` e_1 `then` e_2 `else` $e_3 = e_1 \to e_2$,e_3
Si $e_1 =$ `true` alors $e =$ `true` $\to e_2$,e_3
et $e = e_2 = v_2$
Sinon $e_1 =$ `false` alors $e =$ `false` $\to e_2$,e_3
et $e = e_3 = v_3$

– Cas de l'application
Correction de $e = e_1\ e_2 \rhd v$.
La seule règle applicable à e est [APP] donc $e \rhd v$. Comme hypothèse d'induction on a $e_1 = \texttt{fun}(x : t) \to e$,
$e_2 = v_2$ et $e[x \leftarrow v_2] = v$ car on a $e \rhd v$.
D'où $e_1\ e_2 = (\texttt{fun}(x : t) \to e)\ e_2$
$= (\lambda x.e)e_2$
$= e[x \leftarrow e_2]$ par les règles de BSλ_p
$= e[x \leftarrow v_2]$ par hypothèse d'induction
$= v$ par hypothèse d'induction.

– Cas du constructeur parallèle
Correction de $e =$ `mkpar` $c \rhd < ...,v_i,... >$
La seule règle applicable à e est [MKPAR] donc $e \rhd < ...,v_i,... > v$ On suppose par induction que $e = \texttt{fun}(x : t) \to f$
et $\forall i f[x \leftarrow i] = v_i$ car on a $e \rhd < ...,v_i,... >$.

$e =$ `mkpar` $c = \pi\ c = \pi\lambda x.f$ où $\pi = (\lambda g. < ..,gi,... >)$
$e =< ...,(\lambda x.f)i,... >=< ...,f[x \leftarrow i],... >$

$e =< ...,v_i,... >$

– Cas de l'application parallèle
Correction de $e =$ `apply` $e_1 e_2 \triangleright < ...,v_i,... >$
La seule règle applicable à e est [APPLY] donc $e \triangleright < ...,v_i,... >$
On suppose par induction que $e_1 =<$ `fun`$(x_0 : t) \rightarrow f_0$,...,`fun`$(x_{p-1} : t) \rightarrow f_{p-1} >$,
$e_2 =< w_0,...,w_{p-1} >$
et $\forall i f_i[x_i \leftarrow w_i] = v_i$ car on a $e \triangleright < ...,v_i,... >$.

$e =$ `apply` $e_1\ e_2 = e_1 \# e_2$
$e =<$ `fun`$(x_0 : t) \rightarrow f_0$,...,`fun`$(x_{p-1} : t) \rightarrow f_{p-1} > \# < w_0,...,w_{p-1} >$
$e =< ($`fun`$(x_0 : t) \rightarrow f_0) w_0,...,($`fun`$(x_{p-1} : t) \rightarrow f_{p-1}) w_{p-1}$
$e =< f_0[x_0 \leftarrow w_0],...,f_{p-1}[x_{p-1} \leftarrow w_{p-1}] >$
$e =< v_0,...,v_{p-1} >$

– Cas de la condition parallèle
Correction de $e =$ `if` e_1 `at` e_2 `then` e_3 `else` $e_4 \triangleright v$
Les seules règles applicables à e sont soit [IFATT], soit [IFATF] selon l'évaluation de e_1 donc $e \triangleright v$
On suppose par induction que $e_1 =< w_0,...,w_{p-1} >$, $e_2 = n$, $e_3 = v_3$
et que $e_4 = v_4$
Si $w_n =$ `true` alors $v = v_3$ sinon $v = v_4$
On a $e \triangleright v_3$ si $w_n =$ `true` et $e \triangleright v_4$ si $w_n =$ `false` .

D'où $e =$ `if` e_1 `then` e_2 `else` $e_3 = e_1 \stackrel{e_2}{\rightarrow} e_3,e_4$
Si $w_n =$ `true` alors $e =$ `true` $\rightarrow e_3,e_4$
et $e = e_3 = v_3$
Sinon $w_n =$ `false` alors $e =$ `false` $\rightarrow e_3,e_4$
et $e = e_4 = v_4$

– Cas du get
Correction de $e =$ `get` $e_1\ e_2 \triangleright <$ ..., `fun` $0 \rightarrow$ `if` b_{i0} `then` v_0 `else` nc ,... $>$
| ...
| $p-1 \rightarrow$ `if` $b_{i(p-1)}$ `then` v_{p-1} `else` nc
| - $\rightarrow nc$
La seule règle applicable à e est [GET] donc
$e \triangleright <$ ..., `fun` $0 \rightarrow$ `if` b_{i0} `then` v_0 `else` nc ,... $>$
| ...
| $p-1 \rightarrow$ `if` $b_{i(p-1)}$ `then` v_{p-1} `else` nc
| - $\rightarrow nc$
On suppose par induction que $e_1 =< v_0,...,v_{p-1} >$ et $e_2 =< f_0,\ldots,f_{p-1} >$
et $\forall i \forall j f_i j = b_{ij}$
car on a $e \triangleright <$ ..., `fun` $0 \rightarrow$ `if` b_{i0} `then` v_0 `else` nc ,... $>$.
| ...
| $p-1 \rightarrow$ `if` $b_{i(p-1)}$ `then` v_{p-1} `else` nc
| - $\rightarrow nc$

$e =$ `get` $e_1\ e_2 = e_1 ?? e_2$
$e =< ...,\lambda j.(f_i j \rightarrow v_j,nc),... >$

$e =< ...,\lambda j.(b_{ij} \rightarrow v_j,nc),... >$

$e =< ...,\lambda j.(\ \texttt{if}\ b_{ij}\ \texttt{then}\ v_j\ \texttt{else}\ nc),... >$

$$e =< ...,\ \begin{array}{lccccl} \texttt{fun} & 0 & \rightarrow & \texttt{if}\ b_{i0}\ \texttt{then}\ v_0\ \texttt{else}\ nc & & ,... > \\ | & ... & & & & \\ | & p-1 & \rightarrow & \texttt{if}\ b_{i(p-1)}\ \texttt{then}\ v_{p-1}\ \texttt{else}\ nc & & \\ | & \_ & \rightarrow & nc & & \end{array}$$

- Cas du put

  Correction de $e =\ \texttt{put}\ e_1 \triangleright < ...,(\texttt{fun0} \rightarrow v_{0i}|...|p-1 \rightarrow v_{(p-1)i}|\_ \rightarrow nc),... >$

  La seule règle applicable à e est [PUT] donc

  $e \triangleright < ...,(\texttt{fun0} \rightarrow v_{0i}|...|p-1 \rightarrow v_{(p-1)i}|\_ \rightarrow nc),... >$

  On suppose par induction que $e_1 =< f_0,...,f_{p-1} >$

  et $\forall i \forall j\ \ f_i j = v_{ij}$ car on a $E \triangleright < ...,(\ \texttt{fun}\ 0 \rightarrow v_{0i}|...|p-1 \rightarrow v_{(p-1)i}|\_ \rightarrow nc),... >$.

  $e =\ \texttt{put}\ \ e_1 =!\, e_1 =!\ < f_0,...,f_{p-1} >$

  $e =< ..., \underbrace{\lambda j.((j=0 \rightarrow f_0 i,(j=1 \rightarrow f_1 i,(...,nc))...)}_{proc\ i},... >$

  $e =< ..., \underbrace{\lambda j.((j=0 \rightarrow v_{0i},(j=1 \rightarrow v_{1i},(...,nc))...)}_{proc\ i},... >$

  $e =< ...,(\ \texttt{fun}\ 0 \rightarrow v_{0i}|...|p-1 \rightarrow v_{(p-1)i}|\_ \rightarrow nc),... >$

3.3.3 Exemples

- Mkpar

  Le constructeur parallèle : $\texttt{mkpar}\ \ \texttt{fun}\ (pid : \texttt{int}) \rightarrow (pid+1)$

  $\texttt{mkpar}\ \ \texttt{fun}\ (pid : \texttt{int}) \rightarrow (pid+1)$

  $\overset{[MKPAR]}{\triangleright}\ \ \texttt{fun}\ (pid : \texttt{int}) \rightarrow (pid+1)\quad (pid+1)[pid \leftarrow 0],...,(pid+1)[pid \leftarrow (p-1)]$

  $\overset{[FUN],opérationbinaire}{\triangleright}\quad 0+1\ ...\ (p-1)+1$

  $\texttt{mkpar}\ \ \texttt{fun}\ (pid : \texttt{int}) \rightarrow (pid+1) \triangleright < 1,...,p >$

```
mkpar (fun (pid:int)->pid+1);;

fun (pid:int) -> pid+1 : fun (pid:int) -> pid+1

1 : 1
0 : 0
0+1 : 1

1 : 1
1 : 1
1+1 : 2

1 : 1
2 : 2
```

```
2+1 : 3

1 : 1
3 : 3
3+1 : 4

mkpar fun (pid:int) -> pid+1 : < 1, 2, 3, 4>
```

3.4 Système de typage monomorphe

Ce système permet de vérifier le type des expressions du BSλ simplement typé. Ses jugements sont de la forme $E \vdash e : t$ où E est la liste d'association variable-type. A gauche de : l'expression BSλ simplement typé; à droite son type.

$$\texttt{[0]}\ \frac{E(x) = t}{E \vdash x : t}$$

$$\texttt{[CONST]}\ \frac{}{E \vdash c : \dot{t}}$$

$$\texttt{[NPROCS]}\ \frac{}{E \vdash \ \texttt{nprocs}\ : \texttt{int}}$$

$$\texttt{[NC]}\ \frac{}{E \vdash \ \texttt{nc}\ : \dot{t}}$$

Remarque: il est nécessaire de typer nc ainsi car cette constante peut se substituer à n'importe quelle valeur locale lors d'une communication.

$$\texttt{[IS\_NC]}\ \frac{E \vdash e : \dot{t}}{E \vdash \ \texttt{is\_nc e}\ : \dot{t} \rightarrow \ \texttt{bool}}$$

$$\texttt{[PAR]}\ \frac{\forall i\ v_i : \dot{t}}{< v_0,...,v_{p-1} >: \dot{t}\ \texttt{par}}$$

Les vecteurs ne font pas partie du langage source mais sont nécessaires pour la sémantique opérationnelle et seraient utilisés pour la conception d'un système interactif[1].

$$\texttt{[FUNLOC]}\ \frac{(x,\dot{t_1}) :: E \vdash e : t_2}{E \vdash (\texttt{fun}(x : \dot{t_1}) \rightarrow e) : \dot{t_1} \rightarrow t_2}$$

Il est ainsi interdit de créer des fonctions de sorte globale $\rightarrow$ locale, (comme par exemple des projections), ce qui permet une gestion à deux niveaux des valeurs et des environnements d'exécution.

1. Dans la version actuelle de BSMLlib, les valeurs vectorielles ne sont visualisées que par une commande d'impression: un effet plutôt que la valeur elle-même

$$\texttt{[FUNGLOB]}\ \frac{(x,\bar{t}_1) :: E \vdash e : \bar{t}_2}{E \vdash (\texttt{fun}(x : \bar{t}_1) \to e) : \bar{t}_1 \to \bar{t}_2}$$

$$\texttt{[LETLOC]}\ \frac{E \vdash e_1 : \dot{t}_1 \quad (x,\dot{t}_1) :: E \vdash e_2 : t_2}{E \vdash (\ \texttt{let}\ \ (x : \dot{t}_1) = e_1\ \ \texttt{in}\ \ e_2) : t_2}$$

$$\texttt{[LETGLOB]}\ \frac{E \vdash e_1 : \bar{t}_1 \quad (x,\bar{t}_1) :: E \vdash e_2 : \bar{t}_2}{E \vdash (\ \texttt{let}\ \ (x : \bar{t}_1) = e_1\ \ \texttt{in}\ \ e_2) : \bar{t}_2}$$

$$\texttt{[LETRECLOC]}\ \frac{(f,\dot{s} \to \dot{t}) :: (x,s) :: E \vdash e_1 : \dot{t} \quad (f,\dot{s} \to \dot{t}) :: E \vdash e_2 : t_2}{E \vdash (\ \texttt{letrec}\ \ (f : \dot{s} \to \dot{t})\ x = e_1\ \ \texttt{in}\ \ e_2) : t_2}$$

$$\texttt{[LETRECGLOB]}\ \frac{(f,s \to \bar{t}) :: (x,s) :: E \vdash e_1 : \bar{t} \quad (f,s \to \bar{t}) :: E \vdash e_2 : t_2}{E \vdash (\ \texttt{letrec}\ \ (f : s \to \bar{t})\ x = e_1\ \ \texttt{in}\ \ e_2) : t_2}$$

Même remarque pour [LETLOC] , [LETGLOB] et [LETRECLOC] , [LETRECGLOB] que pour [FUNLOC] [FUNGLOB] . Ces règles permettent de restreindre les expressions aux seuls types acceptables. C'est-à-dire qu'on élimine le type $\bar{t} \to \dot{t}$

$$\texttt{[IF]}\ \frac{E \vdash e_1 :\ \texttt{bool} \quad E \vdash e_2 : t \quad E \vdash e_3 : t}{E \vdash (\ \texttt{if}\ \ e_1\ \ \texttt{then}\ \ e_2\ \ \texttt{else}\ \ e_3) : t}$$

$$\texttt{[EQ]}\ \frac{E \vdash e_1 : \dot{t} \quad E \vdash e_2 : \dot{t} \quad \dot{t} \in \{\ \texttt{int,bool}\ \}}{E \vdash e_1 = e_2 :\ \texttt{bool}}$$

$$\texttt{[APP]}\ \frac{E \vdash e_1 : t_1 \to t_2 \quad E \vdash e_2 : t_1}{E \vdash e_1 e_2 : t_2}$$

$$\texttt{[MKPAR]}\ \frac{E \vdash e : \texttt{int} \to \dot{t}}{E \vdash\ \texttt{mkpar}\ \ e : \dot{t}\ \texttt{par}}$$

$$\texttt{[APPLY]}\ \frac{E \vdash e_1 : (\dot{t}_1 \to \dot{t}_2)\ \ \texttt{par} \quad E \vdash e_2 : \dot{t}_1\ \ \texttt{par}}{E \vdash\ \texttt{apply}\ e_1\ e_2 : \dot{t}_2\ \ \texttt{par}}$$

$$\texttt{[IFAT]}\ \frac{E \vdash e_1 :\ \texttt{bool}\ \ \texttt{par} \quad E \vdash e_2 : \texttt{int} \quad E \vdash e_3 : \bar{t} \quad E \vdash e_4 : \bar{t}}{E \vdash\ \texttt{if}\ e_1\ \texttt{at}\ e_2\ \texttt{then}\ e_3\ \texttt{else}\ e_4 : \bar{t}}$$

$$\texttt{[GET]}\ \frac{E \vdash e_1 : \dot{t}_1\ \ \texttt{par} \quad E \vdash e_2 : (\texttt{int} \to\ \texttt{bool}\ )\ \ \texttt{par}}{E \vdash\ \texttt{get}\ e_1\ e_2 : (\texttt{int} \to \dot{t}_1)\ \ \texttt{par}}$$

$$\texttt{[PUT]}\ \frac{E \vdash e : (\texttt{int} \to \dot{t})\ \texttt{par}}{E \vdash\ \texttt{put}\ e : (\texttt{int} \to \dot{t})\ \ \texttt{par}}$$

Le put et le get sont à l'origine de l'apparition du nc. Il est nécessaire à ce niveau de typer nc en fonction des autres valeurs du vecteur.

3.4.1 Preuve de validité du typage du BSλ_p simplement typé

Lemme 1 *Si* $\{x : t\} \vdash e : t'$ *et* $x : t$ *alors* $e[x \leftarrow v] : t'$

Theorème 1 *Soit e un terme clos, si* $e \triangleright v$ *et e:t alors v:t.*

Preuve du theorème 1
On suppose par induction que le typage des sous-expressions est valide.

- Cas des constantes
  Soient $e = c$, $v = c$ et $t = \dot{t}$.
  Les seules règles applicables à e sont $[\text{CONST}]_{\triangleright}$ et $[\text{CONST}]_{:}$, donc on a $e \triangleright v$, $e : t$
  Les termes e et v étant égaux ils ont le même type.

- Cas fonctionnel
  Soient $e = \texttt{fun}(x : \dot{s}) \rightarrow c$, $v = \texttt{fun}(x : \dot{s}) \rightarrow c$ et $t = \dot{s} \rightarrow s'$.
  Les seules règles applicables à e sont $[\text{FUN}]_{\triangleright}$ et $[\text{FUNLOC}]_{:}$, donc on a $e \triangleright v$, $e : t$
  avec $(x : \dot{s}) :: E \vdash c : s'$.
  Les termes e et v étant égaux ils ont le même type.
  Soient $e = \texttt{fun}(x : \bar{s}) \rightarrow c$, $v = \texttt{fun}(x : \bar{s}) \rightarrow c$ et $t = \bar{s} \rightarrow \bar{s'}$.
  Les seules règles applicables à e sont $[\text{FUN}]_{\triangleright}$ et $[\text{FUNGLOB}]_{:}$, donc on a $e \triangleright v$, $e : t$
  avec $(x : \bar{s}) :: E \vdash c : \bar{s'}$.
  Les termes e et v étant égaux ils ont le même type.

- Cas du let
  Soit $e =$ `let` $(x : \dot{t_1}) = e_1$ `in` e_2
  Les seules règles applicables à e sont $[\text{LET}]_{\triangleright}$ et $[\text{LETLOC}]_{:}$
  donc on a $e \triangleright v$ et $e : t_2$ avec $e_1 \triangleright v_1$, $e_1 : \dot{t_1}$,
  $e_2[x \leftarrow v_1] \triangleright v$ et $(x : \dot{t_1}) :: E \vdash e_2[x \leftarrow v_1] : t_2$.

  On suppose par induction que $v_1 : \dot{t_1}$ et $v : t_2$
  On a bien $v : t_2$.

  Soit $e =$ `let` $(x : \bar{t_1}) = e_1$ `in` e_2
  Les seules règles applicables à e sont $[\text{LET}]_{\triangleright}$ et $[\text{LETGLOB}]_{:}$
  donc on a $e \triangleright v$ et $e : \bar{t_2}$ avec $e_1 \triangleright v_1$, $e_1 : \bar{t_1}$,
  $e_2[x \leftarrow v_1] \triangleright v$ et $(x : \bar{t_1}) :: E \vdash e_2[x \leftarrow v_1] : \bar{t_2}$.

  On suppose par induction que $v_1 : \bar{t_1}$ et $v : \bar{t_2}$
  On a bien $v : \bar{t_2}$.

- Cas de la récursivité
  Soit $e =$ `letrec` $(f : \dot{s} \rightarrow \dot{t})\ x = e_1$ `in` e_2.
  Les seules règles applicables à e sont $[\text{LETREC}]_{\triangleright}$ et $[\text{LETRECLOC}]_{:}$
  donc on a $e \triangleright v$, $e : t_2$
  avec $e_2[f \leftarrow \texttt{Rec}(\texttt{fun}(f : \dot{s} \rightarrow \dot{t}) \rightarrow \texttt{fun}(x : t) \rightarrow e_1)] \triangleright v$,

$(f,\dot{s} \to \dot{t}) :: (x,s) :: E \vdash e_1 : \dot{t}$ et $(f,\dot{s} \to \dot{t}) :: E \vdash e_2 : t_2$
On peut appliquer le mini-lemme puisque $f : \dot{s} \to \dot{t}$
et $(f,\dot{s} \to \dot{t}) :: E \vdash e_2 : t_2$
et ainsi on obtient $e_2[f \leftarrow \texttt{Rec}(\texttt{fun}(f : \dot{s} \to \dot{t}) \to \texttt{fun}(x : t) \to e_1)] : t_2$

Par induction, sachant $e_2[f \leftarrow \texttt{Rec}(\texttt{fun}(f : \dot{s} \to \dot{t}) \to \texttt{fun}(x : t) \to e_1)] \triangleright v$
et $e_2[f \leftarrow \texttt{Rec}(\texttt{fun}(f : \dot{s} \to \dot{t}) \to \texttt{fun}(x : t) \to e_1)] : t_2$
on peut conclure que $v : t_2$

Soit $e =$ `letrec` $(f : s \to \bar{t})\ x = e_1$ `in` e_2.
Les seules règles applicables à e sont $[\text{LETREC}]_\triangleright$ et [LETRECLOC]:
donc on a $e \triangleright v$, $e : t_2$
avec $e_2[f \leftarrow \texttt{Rec}(\texttt{fun}(f : s \to \bar{t}) \to \texttt{fun}(x : t) \to e_1)] \triangleright v$,
$(f,s \to \bar{t}) :: (x,s) :: E \vdash e_1 : \dot{t}$ et $(f,s \to \bar{t}) :: E \vdash e_2 : t_2$

On peut appliquer le mini-lemme puisque $f : s \to \bar{t}$
et $(f,s \to \bar{t}) :: E \vdash e_2 : t_2$
et ainsi on obtient $e_2[f \leftarrow \texttt{Rec}(\texttt{fun}(f : s \to \bar{t}) \to \texttt{fun}(x : t) \to e_1)] : t_2$

Par induction, sachant $e_2[f \leftarrow \texttt{Rec}(\texttt{fun}(f : s \to \bar{t}) \to \texttt{fun}(x : t) \to e_1)] \triangleright v$
et $e_2[f \leftarrow \texttt{Rec}(\texttt{fun}(f : s \to \bar{t}) \to \texttt{fun}(x : t) \to e_1)] : t_2$
on peut conclure que $v : t_2$

– Cas de la condition
Soit $e =$ `if` e_1 `then` e_2 `else` e_3
Les seules règles applicables à e sont $[\text{IFT}]_\triangleright$ (ou $[\text{IFF}]_\triangleright$)et [IF]:
donc on a $e \triangleright v_2$ (ou v_3), $e : t$ avec $e_1 \triangleright$ `true` (ou `false`), $e_1 : \mathit{bool}$,
$e_2 \triangleright v_2$, $e_2 : t$, $e_3 \triangleright v_3$, $e_3 : t$ et $v = v_2$ (ou v_3).

On suppose par induction que `true` (ou `false`) : *bool*
que $v_2 : t$ et que $v_3 : t$
Ainsi, quelque soit l'évaluation de e_1, $v : t$.

– Cas de l'égalité
Soit $e = (e_1 = e_2)$ Les seules règles applicables à e sont $[\text{EQ}]_\triangleright$ (ou $[\text{NEQ}]_\triangleright$)et [EQ]:
donc on a $e \triangleright$ `true` (ou `false`), $e :$ `bool`
avec $e_1 \triangleright v_1$, $e_1 : \dot{t}$,
$e_2 \triangleright v_2$, $e_2 : \dot{t}$,
$v_1 = v_2$ (ou $v_1 \neq v_2$) et $\dot{t} \in \{$ `int,bool` $\}$
Or `true` : `bool` et `false` : `bool`

– Cas de l'application
Soit $e = e_1 e_2$
Les seules règles applicables à e sont $[\text{APP}]_\triangleright$ et [APP]:
donc on a $e \triangleright v$, $e : t_2$ avec $e_1 \triangleright$ `fun` $(x : t_1) \to c$, $e_1 : t_1 \to t_2$, $e_2 \triangleright v_2$,
$c[x \leftarrow v_2] \triangleright v$ et $e_2 : t_1$.
On suppose par induction que $v_1 : t_1 \to t_2$
et $v_2 : t_1$

On peut appliquer le mini-lemme avec $x : t_1 \vdash e : t_2$ et $v_2 : t_1$ et ainsi $c[x \leftarrow v_2] : t_2$
$c[x \leftarrow v_2]$ étant une sous-expression s'évaluant en v on a $v : t_2$

- Cas du contructeur parallèle
  Soit $e =$ `mkpar` c
  Les seules règles applicables à e sont $[\text{MKPAR}]_{\triangleright}$ et $[\text{MKPAR}]$:
  donc on a $e \triangleright < v_0,...,v_{p-1} >$, $e : \dot{t}$ `par` avec $c \triangleright \texttt{fun}(x : t) \rightarrow f$, $c : \texttt{int} \rightarrow \dot{t}$, $\forall i\ f[x \leftarrow i] \triangleright v_i$, $c : \texttt{int} \rightarrow \dot{t}$
  On suppose par induction que $\texttt{fun}(x : t) \rightarrow f : \texttt{int} \rightarrow \dot{t}$
  On peut appliquer p fois le mini-lemme avec $x : \texttt{int} \vdash f : \dot{t}$ et $\forall i\ i : \texttt{int}$ et ainsi $\forall i\ f[x \leftarrow i] : \dot{t}$
  $\forall i\ f[x \leftarrow i]$ étant une sous-expression s'évaluant en v_i on a $\forall i\ v_i : \dot{t}$
  On applique la règle [PAR] et ainsi $< v_0,...,v_{p-1} >: \dot{t}$ `par`

- Cas de l'application parallèle
  Soit $e =$ `apply` $e_1 e_2$
  Les seules règles applicables à e sont $[\text{APPLY}]_{\triangleright}$ et $[\text{APPLY}]$:
  donc on a $e \triangleright < v_0,...,v_{p-1} >$, $e : \dot{t_2}$ `par` avec
  $e_1 \triangleright < \texttt{fun}(x_0 : \dot{t_1}) \rightarrow u_0,...,\texttt{fun}(x_{p-1} : \dot{t_1}) \rightarrow u_{p-1} >$, $e_1 : (\dot{t_1} \rightarrow \dot{t_2})$,
  $e_2 \triangleright < w_0,...,w_{p-1} >$, $e_2 : \dot{t_1}$ `par`
  On suppose par induction que $< \texttt{fun}(x_0 : \dot{t_1}) \rightarrow u_0,...,\texttt{fun}(x_{p-1} : \dot{t_1}) \rightarrow u_{p-1} >: (\dot{t_1} \rightarrow \dot{t_2})$ `par`
  et $< w_0,...,w_{p-1} >: \dot{t_1}$ `par`
  Ce qui correspond à $\forall i\ \texttt{fun}(x_i : \dot{t_1}) \rightarrow u_i : \dot{t_1} \rightarrow \dot{t_2}$ et $\forall i\ w_i : \dot{t_1}$ On peut appliquer p fois le mini-lemme avec $x : \dot{t_1} \vdash u_i : \dot{t_2}$ et $\forall i\ w_i : \dot{t_1}$ et ainsi $\forall i\ f[x \leftarrow w_i] : \dot{t_2}$
  $\forall i\ f[x \leftarrow w_i]$ étant une sous-expression s'évaluant en v_i on a $\forall i\ v_i : \dot{t_2}$ On applique la règle [PAR] et ainsi $< v_0,...,v_{p-1} >: \dot{t_2}$ `par`

- Cas de la conditionnelle parallèle
  Soit $e =$ `if` e_1 `at` e_2 `then` e_3 `else` e_4
  Les seules règles applicables à e sont $[\text{IFATT}]_{\triangleright}$ (ou $[\text{IFATF}]_{\triangleright}$) et $[\text{IFAT}]$:
  donc on a $e \triangleright v$, $e : \bar{t}$ avec
  $e_1 \triangleright < v_0,...,v_{p-1} >$, $e_1 :$ `bool` `par`
  $e_2 \triangleright n$, $e_2 : \texttt{int}$
  $v_n =$ `true` (ou `false`),
  et $e_3 \triangleright v_3$, $e_3 : \bar{t}$
  (ou $e_4 \triangleright v_4$, $e_4 : \bar{t}$)
  et $v = v_3$ (ou v_4)
  On suppose par induction que $v_3 : \bar{t}$ (ou $v_4 : \bar{t}$)
  Par la régle [typage des vecteurs] on a $\forall i v_i :$ `bool` Donc $v : \bar{t}$.

- Cas du get
  Soit $e =$ `get` $e_1\ e_2$
  Les seules règles applicables à e sont $[\text{GET}]_{\triangleright}$ et $[\text{GET}]$:
  donc on a $e \triangleright < ..., \underbrace{\texttt{fun}\ n_{i1} \rightarrow v_{n_{i1}} | ... | n_{ik_i} \rightarrow v_{n_{ik_i}} | \_ \rightarrow nc}_{\texttt{lieu i}}, ... >$,

e : (int $\rightarrow \dot{t}_1$) par
avec $e_1 \triangleright <v_0,...,v_{p-1}>$, $e_1 : \dot{t}_1$ par ,
$e_2 \triangleright < f_0, \ldots, f_{p-1} >$, e_2 : (int → bool) par
et $\forall i \forall j\ f_i j \triangleright b_{ij}$
On suppose par induction que $< v_0,...,v_{p-1} >$: $\dot{t}_1$ par et que $< f_0, \ldots, f_{p-1} >$: (int → bool) par .
Ce qui correspond, par la régle [typage par] à $\forall i\ v_i : \dot{t_1}$ et $\forall i f_i$: int → bool .
Et ainsi $\forall i \forall j b_{ij}$: bool .
On a if b_{ij} then v_j else $nc : \dot{t_1}$
Soit $u_i =$ fun 0 → if b_{i0} then v_0 else nc

| $p-1$ → if $b_{i(p-1)}$ then v_{p-1} else nc
| _ → nc

donc $u_i : int \rightarrow \dot{t_1}$
Et donc par la règle [PAR] $< ...,u_i,... >$: $(int \rightarrow \dot{t_1})$ par

– Cas du put
Soit $e =$ put e_1
Les seules règles applicables à e sont $[\text{PUT}]_{\triangleright}$ et [PUT].
Donc on a $e \triangleright < ...,$ $\underbrace{\text{fun } n_{j1} \rightarrow v_{n_{j1}j} | ... | n_{jk_j} \rightarrow v_{n_{jk_j}j} | \_ \rightarrow nc}_{\text{lieu j}}$,... >,
e : (int → $\dot{t}$) par
avec $e_1 \triangleright < f_0,...,f_{p-1} >$, e_1 : (int → $\dot{t}$) par ,
et $\forall i$ $dest(i) = \{j | f_i j \triangleright v_{ij} \neq nc\}$
On suppose par induction que $< f_0,...,f_{p-1} >$: (int → $\dot{t}$) par
Ce qui correspond à $\forall i f_i$: int → $\dot{t}$
Ainsi $\forall i \forall j f_i j : \dot{t}\ f_i j \triangleright v_{ij}$ étant une sous-expression $v_{ij} : \dot{t}$
Soit $u_j =$ fun 0 → v_{0i}

| $p-1$ → $v_{(p-1)i}$
| _ → nc

$u_{ij} : int \rightarrow \dot{t}$
Par la règle [PAR] $< ...,u_j,... >$: (int → t) par .

3.4.2 Exemples

– Mkpar

```
mkpar (fun (pid:int)->pid+1);;
pid:int
1:int
pid+1:int
fun (pid:int) -> pid+1:(int -> int)
mkpar fun (pid:int) -> pid+1:int par
type : int par
```

– letrec

Typage de la fonction factorielle :

```
letrec (f:int->int) x=if x<=0 then 0 else x*(f (x-1)) in f 3;;
0:int
x:int
f:(int -> int)
x:int
1:int
x-1:int
(f) (x-1):int
x*(f) (x-1):int
x:int
0:int
x<=0:bool
if x<=0 then 0 else x*(f) (x-1):int
f:(int -> int)
3:int
(f) (3):int
letrec (f:(int -> int)) x = if x<=0 then 0 else x*(f) (x-1) in (f) (3):int
type : int
```

3.5 Compilation et machine à pile

Ayant défini un langage, sa sémantique naturelle et son typage, nous donnons maintenant une machine.

On suppose p copies de la SECD machine exécutant de manière asynchrone les instructions des Section 2.5.1 et Section 3.5.2, en plus des opérations collectives synchrones de la Section 3.5.2

3.5.1 Compilation

Les expressions sont d'abord compilées avant d'être exécutée par la machine SECD.

On ajoute quelques expressions au langage des commandes de la SECD pour exprimer le parallèlisme.

Com ::= PID | NPROCS | NC | IS_NC | AT | PUT | SEND | GET | FETCH

NPROCS est le nombre de processeurs. PID est le numéro du processeur localement. AT est l'expression de la conditionnelle parallèle. PUT et SEND servent à implanter le put, GET et FETCH le get.

La première partie du tableau correspond à la compilation d'expressions locales, la deuxième celle des opérations parallèles.

Exemple

- Mkpar
  Compilation du constructeur parallèle :
  $[\![$ `mkpar fun` $(pid : \texttt{int}) \rightarrow (pid + 1)]\!] =$
  $[\![$ `fun` $(pid : \texttt{int}) \rightarrow (pid + 1)]\!] ::$ PID $::$ APP $=$

BSλ simplement typé	Commandes
c	c
$e_1 \oplus e_2$	$[\![e_1]\!] :: [\![e_2]\!] :: \oplus$
$\texttt{fun}(x : t) \rightarrow e$	$\texttt{FUN}(x,[\![e]\!])$
$\texttt{let } (x : t) = e_1 \texttt{ in } e_2$	$\texttt{FUN}(x,[\![e_2]\!]) :: [\![e_1]\!] :: \texttt{APP}$
$\texttt{letrec } (f : t)\ x = e_1 \texttt{ in } e_2$	$\texttt{FUN}(f,[\![e_2]\!]) :: \texttt{REC}(f,x,[\![e_1]\!]) :: \texttt{APP}$
$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$	$[\![e_1]\!] :: \texttt{IF}([\![e_2]\!],[\![e_3]\!])$
$e_1 = e_2$	$[\![e_1]\!] :: [\![e_2]\!] :: \texttt{EQ}$
$e_1 e_2$	$[\![e_1]\!] :: [\![e_2]\!] :: \texttt{APP}$
nprocs	NPROCS
nc	NC
$\texttt{is\_nc } e$	$[\![e]\!] :: \texttt{NC} ::=$
$\texttt{mkpar } e$	$[\![e]\!] :: \texttt{PID} :: \texttt{APP}$
$\texttt{apply } e_1\ e_2$	$[\![e_1]\!] :: [\![e_2]\!] :: \texttt{APP}$
$\texttt{if } e_1 \texttt{ at } e_2 \texttt{ then } e_3 \texttt{ else } e_4$	$[\![e_1]\!] :: [\![e_2]\!] :: \texttt{AT} :: \texttt{IF}([\|e_3\|],[\|e_4\|])$
$\texttt{get } e_1\ e_2$	$[\![e_1]\!] :: [\![e_2]\!] :: \texttt{GET}$
$\texttt{put } e$	$[\![e]\!] :: \texttt{PUT}$

FIG. 3.1 – *compilation*

$\texttt{FUN}(x,[\![pid + 1]\!]) :: \texttt{PID} :: \texttt{APP} =$
$\texttt{FUN}(x,1 :: pid :: +) :: \texttt{PID} :: \texttt{APP}$
Sur chaque processeur.

3.5.2 BSPSECD machine

Commandes supplémentaires pour la machine SECD

S	E	C	D	S	E	C	D
S	E	$\texttt{NPROCS} :: C$	D	$p :: S$	E	C	D
S	E	$\texttt{PID} :: C$	D	$i :: S$	E	C	D
S	E	$\texttt{NC} :: C$	D	$nc :: S$	E	C	D

où p est le nombre de processeurs de la machine parallèle, i est le numéro du processeur sur lequel s'exécute cette copie de la machine SECD et nc la valeur ne "no communication".

Communications et synchronisations

Ces commandes sont celles pour les expressions globales qui traitent déchange et de communication.

- AT
  $< ...,(j :: e_i :: S_i,E_i,\texttt{AT} :: C_i,D_i),... > \rightarrow < ...,(e_j :: S_i,E_i,C_i,D_i),... >$
  j est le numéro de processeur qui transmet sa valeur.
  Les e_i sont les valeurs booléennes locales qui déterminent la conditionnelle.
  Une seule valeur e_i est transmise à tous les processeurs.

- GET
  Obtenu par compilation de get e_1 e_2 après exécution de la machine sur e_1 et sur e_2.

$< ...,(\texttt{CLO}(x,C',E') :: v_i :: S,E,\texttt{GET} :: C,D),... > \rightarrow$

$< ...,(v_i :: S,E@E',\texttt{FUN}(x,C') :: p-1 :: \texttt{APP} :: ... :: \texttt{FUN}(x,C') :: 0 :: \texttt{APP} ::: \texttt{FETCH},(E,C) :: D),... >$

On reconverti localement la fermeture de fonction au sommet de la pile en fonction qu'on applique localement à chaque numéro de processeur.

- FETCH
  Obtenu par exécution de la machine après un get et après l'application des fonctions à chaque numéro de processeur.
  Il y a donc au sommet de la pile de valeurs les booléens qui déterminent les échanges et la valeur à transmettre.
  C'est l'exécution de cette instruction qui réalise les échanges.
  $< ...,(b_{i0} :: ... :: b_{i(p-1)} :: v_i :: S_i,E_i,\texttt{FETCH} :: C_i,D_i),... >$
  $\rightarrow < ...,(\texttt{CLO}(x,C'_i,E') :: S_i,E_i,C_i,D_i),... >$
  où $C'_i = (x :: n_{i_1} :: \texttt{IF}(v_{n_{i_1}},(...,(x :: n_{i_{k_i}} ::=:: \texttt{IF}(v_{n_{i_{k_i}}},[\texttt{NC}])...)$
  tel que $\{n_{i_1},...,n_{i_{k_i}}\} = \{j | b_{ij} = \texttt{true}\}$

  Les $v_{n_{i_k}}$ peuvent avoir plusieurs formes:
  $\texttt{BOOLVAL}\ v$ ou $\texttt{INTVAL}\ v$ alors $E_{n_{i_k}} = \emptyset$
  sinon $v_{n_{i_k}} = \texttt{CLO}(x_{n_{i_k}},C_{n_{i_k}},E_{n_{i_k}})$ ou $\texttt{RECLO}(f_{n_{i_k}},x_{n_{i_k}},C_{n_{i_k}},E_{n_{i_k}})$
  Les formes des $v_{n_{i_k}} \forall k$ déterminent celle de E'
  Ainsi $E' = \cup_{\forall} k E_{n_{i_k}}$
  C'est-à-dire que l'environnement résultant est donné par l'union des environnements quand les valeurs à tranmettre sont des fermetures.

  Il est nécessaire de renommer les variables pour chaque $C_{n_{j_k}}$ et chaque $E_{n_{j_k}}$. On ajoute à ces variables le numéro de leur processeur d'origine.

- PUT
  Obtenu par compilation de put e après l'exécution de la machine sur e.
  $< ...,(\texttt{CLO}(x,C'_i,E'_i) :: S_i,E_i,\texttt{PUT} :: C_i,D_i),... > \rightarrow$

  $< ...,(S_i,E_i@E'_i,\texttt{FUN}(x_i,C'_i) :: p-1 :: \texttt{APP} :: ... :: \texttt{FUN}(x_i,C'_i) :: 0 :: \texttt{APP} :: \texttt{SEND} :: [],(E_i,C_i) :: D_i),... >$

  On reconverti localement la fermeture de fonction au sommet de la pile en fonction qu'on applique localement à chaque numéro de processeur.

- SEND
  Obtenu par exécution de la machine après un get et après l'application des fonctions à chaque numéro de processeur.
  Il y a donc au sommet de la pile de valeurs les valeurs à transmettre.
  C'est l'exécution de cette instruction qui réalise les échanges.
  $< ...,(v_j^0,...v_j^{p-1} :: S_j,E_j,\texttt{SEND} :: C_j,D_j),... >$
  $\rightarrow < ...,(\texttt{CLO}(x,C'_j,E') :: S_j,E_j,C_j,D_j),... >$
  avec $C'_j = (x :: n_{j_1} ::=:: \texttt{IF}(v^j_{n_{j_1}},(x :: n_{j_2} ::=:: \texttt{IF}\ (v^j_{n_{j_2}},...(x :: n_{j_{k_j}} ::=:: \texttt{IF}(v^j_{n_{j_{k_j}}},[\texttt{NC}])...)$

tel que $\{n_{j_1},...,n_{j_{k_j}}\} = \{i | j \in dest(i)\}$ avec $\forall i\ dest(i) = \{j | f_i j \rhd v_{ij} \neq nc\}$

$v_{n_{j_k} j}$ peut avoir plusieurs formes :
`BOOLVAL` v ou `INTVAL` v alors $E_{n_{j_k}} = \emptyset$
sinon $v_{n_{j_k} j} = \texttt{CLO}(x_{n_{j_k}}, C_{n_{j_k}}, E_{n_{j_k}})$ ou $\texttt{RECLO}(f_{n_{j_k}}, x_{n_{j_k}}, C_{n_{j_k}}, E_{n_{j_k}})$
Les formes des $v_{n_{j_k} j}$ déterminent celle de E'
Ainsi $E' = \cup_{\forall k} E_{n_{j_k}}$
C'est-à-dire que l'environnement résultant est donné par l'union des environnements quand les valeurs à tranmettre sont des fermetures.
Il est nécessaire de renommer les variables pour chaque $C_{n_{j_k}}$ et chaque $E_{n_{j_k}}$. On ajoute à ces variables le numéro de leur processeur d'origine.

Exemples

- Mkpar
  `mkpar` `fun` $(pid : \texttt{int}) \rightarrow (pid + 1)$

```
mkpar (fun (pid:int)->pid+1);;
```

modle :

```
<|S0|E0|C0|D0|,
|S1|E1|C1|D1|,
|S2|E2|C2|D2|,
|S3|E3|C3|D3|>

<|||FUN(pid,pid;1;+);PID;APP||,
|||FUN(pid,pid;1;+);PID;APP||,
|||FUN(pid,pid;1;+);PID;APP||,
|||FUN(pid,pid;1;+);PID;APP||>

<|CLO(pid,pid;1;+,)||PID;APP||,
|CLO(pid,pid;1;+,)||PID;APP||,
|CLO(pid,pid;1;+,)||PID;APP||,
|CLO(pid,pid;1;+,)||PID;APP||>

<|0;CLO(pid,pid;1;+,)||APP||,
|1;CLO(pid,pid;1;+,)||APP||,
|2;CLO(pid,pid;1;+,)||APP||,
|3;CLO(pid,pid;1;+,)||APP||>

<||(pid,0)|pid;1;+|(,)|,
||(pid,1)|pid;1;+|(,)|,
||(pid,2)|pid;1;+|(,)|,
||(pid,3)|pid;1;+|(,)|>

<|0|(pid,0)|1;+|(,)|,
|1|(pid,1)|1;+|(,)|,
|2|(pid,2)|1;+|(,)|,
|3|(pid,3)|1;+|(,)|>
```

```
<|1;0|(pid,0)|+|(,)|,
|1;1|(pid,1)|+|(,)|,
|1;2|(pid,2)|+|(,)|,
|1;3|(pid,3)|+|(,)|>

<|1|(pid,0)||(,)|,
|2|(pid,1)||(,)|,
|3|(pid,2)||(,)|,
|4|(pid,3)||(,)|>

<|1||||,
|2||||,
|3||||,
|4||||>
```

3.6 Les coûts

Le système suivant permet de calculer le coût de l'évalutation des expressions du BSλ simplement typé. Pour obtenir une estimation réaliste du temps de calcul selon BSP, on évalue le temps mit par la machine SECD pour l'exécution de l'instruction. Localement, le calcul est assez simple. Une transition est considérée comme l'unité de temps. Pour le calcul du coût global, c'est un peu différent. Le temps de calcul peut ne pas être le même en chaque processeur puisque le calcul peut être asynchrone. On a donc une nouvelle grammaire de valeurs :
$val ::= c \mid \lambda x.e$
$val_{\rhd_t} ::= c, t \mid \lambda x.e, t \mid \ < val,...,val > \ , < t,...,t >$
Par contre dés qu'intervient une instruction synchrone, le vecteur des coûts est uniformisé. Il y a donc un calcul plus complexe pour les expressions `ifat`, `get` et `put`. Il est basé sur le modèle BSP.

Les jugements du système sont de la forme $e \rhd v,c$ où $v,c \in val_{\rhd t}$. A gauche l'expression BSλ simplement typé ; à droite sa valeur et son coût.

$$\texttt{[CONST]}\ \frac{}{c \rhd c, 1}$$

$$\texttt{[NPROCS]}\ \frac{}{\texttt{nprocs} \ \rhd p, 1}$$

$$\texttt{[ISNCT]}\ \frac{e \rhd nc, c}{\texttt{is\_nc e} \ \rhd \ \texttt{true,} \ c+1}$$

$$\texttt{[ISNCF]}\ \frac{e \rhd v \neq nc, c}{\texttt{is\_nc e} \ \rhd \ \texttt{false,} \ c+1}$$

$$\texttt{[FUN]}\ \frac{}{\texttt{fun}\ (x : t) \rightarrow e \rhd \ \texttt{fun} \ \ (x : t) \rightarrow e, 1}$$

$$\texttt{[LETLOC1]}\ \frac{e_1 \rhd v_1, c_1 \quad e_2[x \leftarrow v_1] \rhd v, c}{\texttt{let}\ (x:t) = e_1\ \texttt{in}\ e_2 \rhd v, c + c_1 + 3}$$

3 unités de temps sont utilisées en plus du coût de e_1 et e_2 : une première pour la fermeture de la fonction, une deuxième pour la transition de l'application et une dernière pour revenir à la suite d'instructions initiales après l'application (ligne 1 dans la SECD 2.6 restauration du dump).

$$\texttt{[LETLOC2]}\ \frac{e_1 \rhd v_1, c_1 \quad e_2[x \leftarrow v_1] \rhd < w_0,...w_{p-1} > , < c'_0,...c'_{p-1} >}{\texttt{let}\ (x:t) = e_1\ \texttt{in}\ e_2 \rhd < w_0,...,w_{p-1} > , < c'_0 + c_1 + 3,...,c'_{p-1} + c_1 + 3 >}$$

En chaque processeur s'effectue: le calcul de e_1, l'application localement à e_2 et la transition de fermeture de e_2, la transition d'application et la restauration du dump.

$$\texttt{[LETGLOB]}\ \frac{e_1 \rhd < v_0,...,v_{p-1} > , < c_0,..,c_{p-1} > \quad e_2[x \leftarrow v_1] \rhd < w_0,...w_{p-1} > , < c'_0,...c'_{p-1} >}{\texttt{let}\ (x:t) = e_1\ \texttt{in}\ e_2 \rhd < w_0,...,w_{p-1} > , < c'_0 + c_0 + 3,...,c'_{p-1} + c_{p-1} + 3 >}$$

En chaque processeur s'effectue: le calcul de e_1, l'application localement à e_2 et la transition de fermeture de e_2, la transition d'application et la restauration du dump.

$$\texttt{[REC]}\ \frac{e(\texttt{Rec}\ e)x \rhd v, c}{(\texttt{Rec}\ e)x \rhd v, c}$$

Cette règle correspond à la transition du `RECLO` qui est une fermeture récursive.

$$\texttt{[LETREC]}\ \frac{e_2[f \leftarrow \texttt{Rec}(\texttt{fun}(f:t \to t') \to \texttt{fun}(x:t) \to e_1)] \rhd v, c}{\texttt{letrec}\ (f:t \to t')\ x = e_1\ \texttt{in}\ e_2 \rhd v, c + 3}$$

A la manière du let, le letrec nécessite 3 unités de temps pour la fermeture récursive, la transition d'application, et le retour aux instructions initiales.

$$\texttt{[IFT]}\ \frac{e_1 \rhd\ \texttt{true}\ , c_1 \quad e_2 \rhd v, c_2}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rhd v, c_1 + c_2 + 1}$$

$$\texttt{[IFF]}\ \frac{e_1 \rhd\ \texttt{false}\ , c_1 \quad e_3 \rhd v, c_3}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rhd v, c_1 + c_3 + 1}$$

$$\texttt{[EQ]}\ \frac{e_1 \rhd v, c_1 \quad e_2 \rhd v, c_2}{e_1 = e_2 \rhd\ \texttt{true}\ , c_1 + c_2 + 1}$$

$$\texttt{[NEQ]}\ \frac{e_1 \rhd v_1, c_1 \quad e_2 \rhd v_2, c_2 \quad v_1 \neq v_2}{e_1 = e_2 \rhd\ \texttt{false,}\ c_1 + c_2 + 1}$$

$$\texttt{[APPLOC1]}\ \frac{e_1 \rhd\ \texttt{fun}\ (x:\dot{t}) \to e, c_1 \quad e_2 \rhd v_2, c_2 \quad e[x \leftarrow v_2] \rhd v, c_3}{e_1 e_2 \rhd v, c_1 + c_2 + c_3 + 2}$$

$$\texttt{[APPLOC2]}\ \frac{e_1 \rhd\ \texttt{fun}\ \ (x:\dot{t}) \to e, c_1 \quad e_2 \rhd v_2, c_2 \quad e[x \leftarrow v_2] \rhd < w_0,...w_{p-1} > , < t_0,...t_{p-1} >}{e_1 e_2 \rhd < w_0,...w_{p-1} > , < c_1 + c_2 + t_0 + 2,...,c_1 + c_2 + t_{p-1} + 2 >}$$

$$\texttt{[APPGLOB]}\ \frac{\begin{array}{c} e_1 \rhd \ \texttt{fun}\ \ (x:\bar{t}) \to e,\, c_1 \quad e_2 \rhd < v_0,...v_{p-1} > , < t_0,...t_{p-1} > \\ e[x \leftarrow < v_0,...,v_{p-1} >] \rhd < w_0,...,w_{p-1} > , < t'_0,...t'_{p-1} > \end{array}}{e_1 e_2 \rhd < w_0,...,w_{p-1} > , < c_1 + t_0 + t'_0 + 2,...,c_1 + t_{p-1} + t'_{p-1} + 2 >}$$

2 unités de temps : une pour la transition d'application et l'autre pour rétablir le dump.

$$\texttt{[MKPAR]}\ \frac{e_1 \rhd \ \texttt{fun}\ \ (x:t) \to e,\, c \quad \forall i\ e[x \leftarrow i] \rhd v_i,\, c_i}{\texttt{mkpar}\ e_1 \rhd < v_0,...,v_{p-1} > , < c + c_0 + 3,...,c + c_{p-1} + 3 >}$$

3 unités de temps : une pour l'évaluation du pid en chaque point, une pour l'application et une pour la restauration du dump.

$$\texttt{[APPLY]}\ \frac{\begin{array}{c} e_1 \rhd < \texttt{fun}(x_0:t) \to u_0,...,\texttt{fun}(x_{p-1}:t) \to u_{p-1} > , < t_0,...t_{p-1} > \\ e_2 \rhd < w_0,...,w_{p-1} > , < c_0,...,c_{p-1} > \quad \forall i\ u_i[x_i \leftarrow w_i] \rhd v_i,\, c'_i \end{array}}{\texttt{apply}\ e_1 e_2 \rhd < v_0,...,v_{p-1} > , < t_0 + c_0 + c'_0 + 2,...,t_{p-1} + c_{p-1} + c'_{p-1} + 2 >}$$

Agit comme l'application en chaque point, d'où l'utilisation de 2 unités de temps supplémentaires.

Dans les expressions suivantes, interviennent des échanges de données. Le calcul du coût passe donc par l'introduction des paramètres BSP (g,p,l) :

$$Time(s) = \max_{i:processeur} w_i^{(s)} + \max_{i:processeur} h_i^{(s)} * g + l$$

où $w_i^{(s)}$ = temps de calcul local du processeur i durant la super-étape s et $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ où $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) est le nombre de mots transmis (resp. reçus) par le processeur i durant la super-étape s. g et l ne sont pas encore établis.

$$\texttt{[IFATT]}\ \frac{\begin{array}{c} e_1 \rhd < v_0,...,v_{p-1} > , < c_0,...,c_{p-1} > \quad e_2 \rhd n,\, t_2 \\ v_n = \ \texttt{true} \quad e_3 \rhd v,\, < c'_0,...,c'_{p-1} > \end{array}}{\texttt{if}\ \ e_1\ \ \texttt{at}\ \ e_2\ \texttt{then}\ e_3\ \texttt{else}\ e_4 \rhd v, < ...,\underbrace{\max_{\forall i \in \{0..p-1\}} (c_i) + t_2 + c_j + g(p-1) + L + 2}_{\texttt{lieu j}},... >}$$

$$\texttt{[IFATF]}\ \frac{\begin{array}{c} e_1 \rhd < v_0,...,v_{p-1} > , < c_0,...,c_{p-1} > \quad e_2 \rhd n,\, t_2 \\ v_n = \ \texttt{false} \quad e_4 \rhd v,\, < c'_0,...,c'_{p-1} > \end{array}}{\texttt{if}\ \ e_1\ \ \texttt{at}\ \ e_2\ \texttt{then}\ e_3\ \texttt{else}\ e_4 \rhd v, < ...,\underbrace{Max_{\forall i \in \{0..p-1\}}(c_i) + t_2 + c_j + gh + L + 2}_{\texttt{lieu j}},... >}$$

On attend que toutes les valeurs v_i soient calculées pour transmettre celle déterminée par e_2 d'où $\max_{\forall i \in \{0..p-1\}}(c_i)$. Deux transitions sont nécessaires une pour le `IF` et une pour le `AT`. Le coût de lévaluation de e_2 et de e_3 (resp. e_4) est à ajouter au coût final de [IFATT] (resp. [IFATF]) ainsi que les valeurs dues

aux échanges. Pour l'expression `ifat`, la communication se fait d'un processeur vers tous les autres, d'où $h = p - 1$. On a donc à ajouter au calcul g(p-1) et L.

$$\texttt{[GET]}\ \frac{\begin{array}{c} e_1 \rhd < v_0,...,v_{p-1} > , < c_0,...,c_{p-1} > \quad e_2 \rhd < f_0,\ldots,f_{p-1} > , < c'_0,...,c'_{p-1} > \\ \forall i \forall j\ f_i j \rhd b_{ij}, t_{ij} \quad req(i) = \{j | b_{ij} = \texttt{true}\} = \{n_{i1},...,n_{ik_i}\} \end{array}}{\begin{array}{l} \texttt{get}\ \ e_1\ e_2 \rhd\ \ < ...,\ \underbrace{\texttt{fun}\ n_{i1} \rightarrow v_{n_{i1}} | ... | n_{ik_i} \rightarrow v_{n_{ik_i}} | \_ \rightarrow nc}_{\texttt{lieu i}},... > , \\ \qquad < ...,\ \begin{array}{c} \max_{\forall i \in \{0..p-1\}}(c_i) \\ + \\ \max_{\forall i \in \{0..p-1\}}(c'_i) \\ + \\ \underbrace{\max_{\forall i \in \{0..p-1\}}(\sum_{\forall j}(t_{ij}))}_{\texttt{lieu i}} \end{array} + 3 + gh + L ,... > \end{array}}$$

En chaque processeur, il y a évaluation de sa composante en e_1, (p+1) évaluation de e_2 appliquée aux p processeurs qui coûte t_{ij}. Une transition pour le get, et une pour restaurer le dump. Ce qui fait en chaque processeur i , avant l'échange de données : $c_i + c'_i + \sum_{\forall j} t_{ij} + 1 + 1 = c_i + c'_i + \sum_j t_{ij} + 2$. Comme il y a barrière de synchronisation pour la transition `FETCH`, on prend le max de l'équation précédente à laquelle on ajoute 1 pour la transition, g*h pour les échanges et L pour la latence. On obtient donc :
$\max_{\forall i \in \{0..p-1\}}(c_i) + \max_{\forall i \in \{0..p-1\}}(c'_i) + \max_{\forall}\, i \in \{0..p-1\}(\sum_{\forall j} t_{ij}) + 3 + gh + L$
où h est calculé comme suit :
$h_{i+} = card\{j | b_{ji} =$ `true` $\}$ et $h_{i-} = card\{j | b_{ij} = true\} = card\ req(i)$
Ainsi $h = \max_{\forall i \in \{0..p-1\}}(\max h_{i+}, h_{i-})$

$$\texttt{[PUT]}\ \frac{\begin{array}{c} e \rhd < f_0,...,f_{p-1} > , < c_0,...,c_{p-1} > \\ \forall i\ dest(i) = \{j | f_i j \rhd v_{ij} \neq nc, t_{ij}\} \end{array}}{\begin{array}{l} \texttt{put}\ e \rhd \quad < ...,\ \underbrace{\texttt{fun}\ n_{j1} \rightarrow v_{n_{j1}j} | ... | n_{jk_j} \rightarrow v_{n_{jk_j}j} | \_ \rightarrow nc}_{\texttt{lieu j}},... > , \\ \qquad < ..., \underbrace{\max\limits_{\forall i \in \{0..p-1\}}(c_i) + \max\limits_{\forall i \in \{0..p-1\}}(\sum \forall j t_{ij}) + 3 + gh + L}_{\texttt{lieu j}},... > \end{array}}$$

avec $\forall j . \{n_{j1},...,n_{jk_j}\} = \{i | j \in dest(i)\}$

En chaque processeur, il y a évaluation de sa composante en e, évaluation de e appliquée aux p processeurs qui coûte t_{ij}. Une transition pour le put, et une pour restaurer le dump. Ce qui fait en chaque processeur i , avant l'échange de données : $c_i + c_i + \sum_{\forall j} t_{ij} + 1 + 1 = c_i + \sum_j t_{ij} + 2$. Comme il y a barrière de synchronisation pour la transition `SEND`, on prend le max de l'équation précédente à laquelle on ajoute 1 pour la transition, g*h pour les échanges et L pour la latence. On obtient donc :
$\max_{\forall i \in \{0..p-1\}}(c_i) + \max_{\forall}\, i \in \{0..p-1\}(\sum_{\forall j} t_{ij}) + 3 + gh + L$ où h est calculé comme suit :
$h_{i+} = card\{j | f_j i = v_{ji} \neq$ `nc` $\}$ et $h_{i-} = card\{j | f_i j = v_{ij} \neq$ `nc` $\} = card\ dest(i)$
Ainsi $h = \max_{\forall i \in \{0..p-1\}}(\max h_{i+}, h_{i-})$

Pour l'instant la taille des objets à transmettre n'a pas été prise en compte. Il faut trouver un moyen d'évaluer le coût de l'échange d'un objet. Surtout quand

il s'agit d'une fonction puisqu'il faut alors transmettre sa fermeture qui contient son environnement.

Chapitre 4

Conclusion

La programmation des évaluateurs pour chacune des sémantiques et pour le typage a permis de vérifier leur correction. La programmation de la machine à pile à l'aide des outils Ocaml permet également de faire des tests. L'intégration du calcul des coûts dans la machine á pile n'est pas encore complète. La programmation d'exemples types comme *fold* et *scan* [1] est en cours pour vérifier les sémantiques et le calcul des coûts.

Par la suite, nous espérons traduire le langage dans une machine à pile du type CAM/ZAM [4]. Nous pourrons alors faire une implantation parallèle et effectuer des tests où les estimations de coûts seront réalistes. Un système de typage polymorphe sera aussi développé.

Le but ultime de ce projet est une réalisation complète d'une version BSP du langage Caml. Les résultats présentés ici nous ont permis de vérifier la bonne structure du langage et les difficultés à surmonter pour une implantation parallèle correcte et efficace.

Bibliographie

[1] O. Ballereau, F. Loulergue, and G. Hains. High level BSP programming: BSML and BSλ. In P. Trinder, G. Michaelson, and H.-W. Loidl, editors, *Functional Programming Trends*, chapter 4. Intellect Books, November 2000. Preliminary version appeared in proceedings of SFP'99: First Scottish Functional Programming Workshop, TR RM/99/9, pp.43–52, Heriot-Watt University, September 1999.

[2] H.P. Barendregt. *The lambda calculus, Its syntax and Semantics*. North-Holland, 1986.

[3] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. Rapport Technique 529, INRIA, 1986.

[4] G. Cousineau, P.-L. Curien, and M. Mauny. *The categorical abstract machine*, volume 8. North-Holland, 1987.

[5] Christian Foisy and Emmanuel Chailloux. Caml flight: a portable spmd extension of ml for distributed memory multiprocessors. Conference on High Performance Functional Computing, avril 1995.

[6] Christian Foisy and Julie Vachon et Gaétan Hains. Dpml: De la sémantique à l'implantation. In Christian Queinnec Bernard Serpette, Pierre Cointe, editor, *JFLA 94, Journées francophones des langages applicatifs*, number 11. INRIA, Collection Didactique, January 31 - Febuary 1 1994.

[7] Gaétan Hains and C. Foisy. The data-parallel categorical abstract machine. *PARLE'93, Parallel Architechtures and Languages Europe*, (694), June 1993.

[8] P. Henderson. *Functional Programming. Application and Implementation.* Prentice Hall, 1980.

[9] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[10] F. Loulergue. Bsλ_p : Functional bsp programs on enumerated vectors. 2000.

[11] F. Loulergue. *Conception de langages fonctionnels pour la programmation massivement parallèle.* thése de doctorat, Université d'Orléans, LIFO, 4 rue Léonard de Vinci, BP 6759, F-45067 Orléans Cedex 2, France, January 2000. `http://www.univ-orleans.fr/SCIENCES/LIFO/Members/loulergu`.

[12] F Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3), 2000.

[13] L. G. Valiant. A bridging model for parallel computation. *Communication of the ACM*, 33(8):103–111, August 1990.