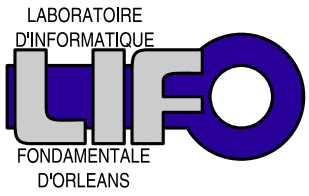
Sémantique de la bibliothèque BSPlib

Julien Tesson

Stage de Master

sous la direction de
Frédéric Loulergue
Laboratoire d'Informatique Fondamentale d'Orléans
frederic.loulergue@univ-orleans.fr

Février-Juin 2007



Remerciements

Je tiens à remercier tout particulièrement Frédéric Loulergue pour son encadrement et le temps qu'il a bien voulu me consacrer, de même que pour son soutien, espérons que ce ne soit que le début d'une collaboration fructueuse.

Je souhaite aussi remercier tous les membres du laboratoire pour leur accueil chaleureux.

Enfin je remercie les stagiaires de la "salle DEA", Matthieu, Jérémy, Simon et Christophe, pour leur bonne humeur et pour nos échanges d'idées.

Résumé

Ce rapport présente les travaux effectués lors de mon stage de Master recherche. Ces travaux consistent, dans un premier temps, en la conception de sémantiques formelles pour la bibliothèque BSPlib. BSPlib est une bibliothèque de programmation parallèle suivant le modèle BSP. Ce modèle de programmation parallèle structuré en super-étapes offre l'avantage d'être sans inter-blocage et d'avoir un modèle de coût permettant de prédire facilement la performance d'un programme sur une architecture donnée. Dans un second temps les propriétés des programmes BSP, ou de certaines classes de programmes, sont exhibées.

Table des matières

Chapitre 1

Introduction

1.1 Programmation parallèle et Modèle BSP

De nombreux problèmes, notamment la simulation de systèmes complexes, nécessitent des performances que seules les ordinateurs parallèles peuvent fournir. Cependant la programmation pour de telles architectures s'avère complexe lorsqu'il s'agit de produire des programmes portables sur différentes machines parallèle tant du point de vue de la sémantique (le programme produit le même résultat quelque soit la machine) que des prévisions de performances.

L'approche actuellement la plus utilisée pour tirer partie de ces architecture parallèles, est le data-parallèlisme, qui consiste à utiliser les processeurs pour effectuer des calculs en parallèle sur des données différentes. Lorsque le même programme est recopié sur tous les processeurs on parle de programmation SPMD pour *Simple Program Multiple Data*. De nombreuse bibliothèques suivent ce paradigme, parmi lesquelles se trouvent MPI [5] et BSP [4].

Le modèle BSP (*Bulk Synchronous parallelism*) décompose l'exécution d'un programme parallèle en une suite de super-étapes, dissociant les communications des synchronisations. Une super-étape, représentée à la figure 1.1, comporte trois phases :

- une phase de calcul asynchrone où chacun des processeurs effectue un calcul séquentiel indépendamment des autres processeurs.
- une phase de communication durant laquelle les processeurs échangent des données.
- une barrière de synchronisation impliquant tous les processeurs, ne terminant qu'après que toutes les données aient été échangées. Après la synchronisation ces données sont accessibles dans la mémoire locale des processus destinataires.

Ce modèle offre l'avantage de la simplicité de programmation et d'un modèle de coût permettant de prédire l'efficacité du programme en fonction du nombre de processeurs et des caractéristiques de la machine parallèle. En outre il assure l'absence d'inter-blocage.

1.2 Sémantique des langages impératifs

Les sémantiques formelles ont pour but de donner un sens à un langages de programmation en explicitant comment chaque élément syntaxique agit sur son environnement. Cette formalisation permet de prédire le comportement d'un programme et de vérifier

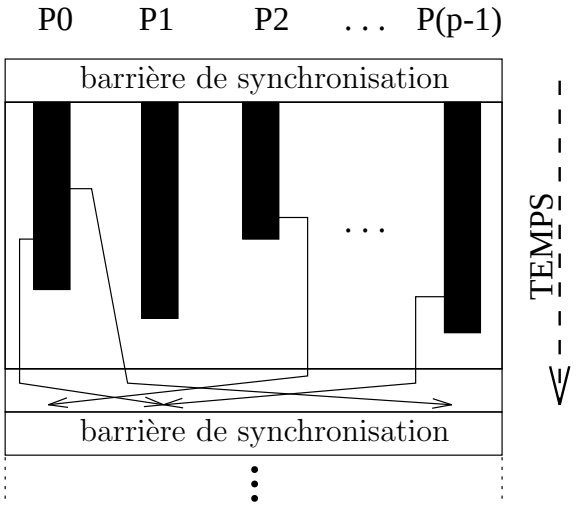


FIG. 1.1 – Super-étape BSP

certaine de ses propriétés.

Dans [8], Winskel répertorie trois catégories de sémantiques pour les langages de programmation, les sémantiques opérationnelles, les sémantiques dénotationnelle et les sémantiques axiomatiques.

Une sémantique axiomatique permet de vérifier un ensemble de propriétés du programme. À chaque instruction de celui-ci on associe un couple de propriétés, appelé pré- et post-conditions, formalisées dans un langage d'assertion, souvent une extension des expressions booléennes et arithmétiques du langages. La sémantique axiomatique décrit pour chaque éléments syntaxiques du langage une relation|transformation entre pré et post-conditions. Cette forme de sémantique, très utile pour montrer des propriétés locales d'un programme, n'est toutefois pas adéquate pour montrer les propriétés intrinsèques d'un langage de programmation.

Une sémantique dénotationnelle s'attache plus à décrire le comportement de tous les programmes exprimables dans un langage en définissant pour chaque élément syntaxique du langage une relation, par exemple entre l'état initial et l'état final des variables, ou entre l'état des variables et le résultat d'une opération arithmétique. Si le langage est déterministe ces relations sont des fonctions, un unique résultat pouvant être obtenu depuis un même état initial par une même instruction.

Une sémantique opérationnelle définit plus précisément la manière dont les instructions du langage vont être évaluées. Elle est décrite par un ensemble de règle définissant une relation de transition $\rightarrow$ menant un couple instruction - environnement vers un état final. Une telle sémantique peut être définie à différents niveaux de granularité. On parle de sémantique à petits pas lorsque chaque étape de réduction jusqu'à une valeur est donnée ou de sémantique à grands pas lorsque la sémantique donne directement une relation entre expressions et valeurs. La figure 1.2 présente la sémantique à grand pas de l'exécution conditionnelle dans un langage impératif.

$$\frac{\langle b, \sigma\rangle \ \rightarrow\ \textbf{True} \quad \langle c_1, \sigma\rangle \ \rightarrow\ \sigma'}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ end}, \sigma\rangle \ \rightarrow\ \sigma'}$$

$$\frac{\langle b, \sigma\rangle \ \rightarrow\ \textbf{False} \quad \langle c_2, \sigma\rangle \ \rightarrow\ \sigma'}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ end}, \sigma\rangle \ \rightarrow\ \sigma'}$$

FIG. 1.2 – Sémantique à grands pas du if-then-else dans un langage impératif.

1.3 Sémantique des programmes BSP

Dans [3], Lecomber donne une extension de la sémantique pour le modèle BSP définie dans [2], décrivant l'ensemble des observations (c.-à-d. l'ensemble des couples état initial - état final) possibles pour un programme. Le programme est vu comme une relation entre état initiaux et état finaux des processus. L'état d'un processus BSP est composé :

- d'une fonction v assignant une valeur aux variables locales.
- d'une séquence de valeur in des variables globales au début de chaque super-étapes.
- d'une séquence de fonction $putq$ associant aux variables globales un ensemble de valeurs possibles, correspondant aux opérations **put** affectant ces variables lors des super-étapes précédente et courante.
- d'une relation $getq$ associant aux variables locales des variables globales, modélisant les opération **get** effectuée lors de la super-étape courante.
- d'une séquence de fonction tr donnant la valeur des variables locales à la fin des super-étapes précédentes.
- du nombre c de super-étapes franchies.
- d'un booléen ok représentant la terminaison du programme.

La sémantique consiste en un ensemble d'opérateurs associés aux éléments syntaxiques du langage restreignant les couples possibles pour la relation que représente le programme. Ces opérateurs définissent les états successifs possibles pour un processus en fonction des valeurs possibles des variables globales au début de chaque super-étapes. Ces valeurs, fournies par la séquence in sont considérées comme initialement déterminées. Les différents processus sont mis en cohérence par un opérateur de composition parallèle. Cette opérateur spécifie que le comportement du programme n'est connu que jusqu'à la dernière synchronisation commune à tous les processus. Enfin un opérateur impose que les valeur des variables globales données par la séquence in soit cohérente avec les valeurs des variables locales de la séquence tr et les demandes de communication de la séquence $putq$.

L'auteur utilise la sémantique ainsi définie, proche d'une sémantique dénotationnelle pour démontrer l'équivalence de programmes BSP.

Une sémantique axiomatique est définie par Stewart et Clint dans [6, 7] pour les programmes BSP impératifs. L'état du programme est décrit dans les axiomes par deux fonctions associant une valeur aux variables. Une fonction décrit l'état de la mémoire l'autre les communications en attente. Les auteurs définissent une transformation pour les assertion, appelée *predicate substitution*, correspondant à l'application des communications. Ils présentent la sémantique de la composition parallèles d'un nombre fixe de processus sous forme d'une règle pour la composition parallèle de processus indépendants et d'une règle pour l'application des barrières de synchronisation. En effet le calcul suivant le modèle BSP est vu comme une alternance de compositions parallèles indépendantes et de synchroni-

sations. Le problème de l'indéterminisme des communications est abordé, proposant une sémantique pour les cas où plusieurs communications agissent sur la même variable. Les auteurs donnent une application de cette sémantique en démontrant la correction d'un certain nombre de programmes parallèles.

Nous proposons une sémantique opérationnelle pour un mini langage impératif BSP proche de l'implémentation impérative du modèle BSP qu'est la BSPlib en ce basant sur la description informelle de celle-ci proposée dans [1] et présentée au chapitre suivant.

Partant de cette sémantique nous discutons du déterminisme des programmes écrits dans ce langage et nous donnons une caractérisation d'un sous ensemble de programmes sans erreur de synchronisation.

Chapitre 2

Présentation de la BSPlib

La BSPlib est une bibliothèque pour la programmation parallèle impérative suivant le modèle BSP. Elle est composée de 20 opérations. Nous commencerons par décrire quelques opérations de base de la bibliothèque puis les opérations dédiées aux communications seront présentées, regroupées suivant deux types de communication : les accès directes aux mémoires distantes *(DRMA pour Direct Remote Memory Access)* et les communications par passage de messages *(BSMP pour Bulk Synchronous Messages Passing)*. Ces modes de communications seront alors décrits plus en détails.

2.1 Opérations basiques

Les opérations de base de la BSPlib permettent de déclencher et de stopper l'exécution parallèle (`bsp_begin`, `bsp_end`) et de lever une erreur arrêtant l'ensemble des processus (`bsp_abort`). Le paramètre de `bsp_begin` permet de restreindre le nombre de processeurs sur lequel est exécuté le programme. Deux fonctions permettent d'obtenir les paramètres propres au modèle BSP :

- `bsp_nprocs` renvoi le nombre de processeurs sur lesquels est distribué le programme.
- `bsp_pid` renvoi l'identifiant du processeur sur lequel elle s'exécute.

Les barrières de synchronisations sont effectués par la fonction `bsp_sync`.

2.2 Accès direct à la mémoire distante

Dans le mode de communication DRMA les processus peuvent lire et écrire dans les variables déclarées partagées par les autres processus. La déclaration du partage d'une variable se fait par la commande `bsp_push_reg(var,size)` où `var` est une variable à partager de taille `size`. Elle peut être retirée du partage par la commande `bsp_pop_reg(var)`.

Les communications sont effectuées par les commandes `bsp_put` et `bsp_get`

- `bsp_put(dest, src, tgt, offset, nbytes)` envoi les données dans une variable distante. `dest` est l'identifiant du processus distant auquel sont envoyées les données. `src` est l'emplacement du premier octet des données à transférer, `tgt` est l'emplacement du premier octet de la variable où les données seront stockées. `offset` est un décalage (en octets) depuis `tgt` indiquant l'endroit où débute la copie des données. `nbytes` est la quantité de données transférées.

– `bsp_get(dest, rloc, offset, tgt, nbytes)` demande des données depuis une variable distante. `dest` est l'identifiant du processeur possédant les données demandées, `rloc` est l'emplacement du premier octet de la variable contenant ces données. `tgt` est l'emplacement de la variable locale où seront copiées les données, `offset` est un décalage par rapport à `rloc` indiquant le début des `nbytes` octets de données à transférer.

Ces deux commandes demandent la communication de données, cependant le modèle BSP n'assure leur transmission qu'à la fin de la super-étape. Les données à transférer sont donc copiées dans un buffer pour assurer leur intégrité jusqu'à la synchronisation. Si l'utilisateur est certain que les variables ne sont pas modifiées entre la demande de communication et la barrière de synchronisation, il peut utiliser les versions `bsp_hpput` et `bsp_hpget` qui ne copient pas les données afin d'améliorer les performances.

2.3 Communication par envoi de messages

Les opérations de communication DRMA sont intéressantes tant que les quantités de données transférées à un processeur durant les super-étapes peuvent être prévues à l'avance, cependant lorsque ces quantités deviennent irrégulières la programmation par passage de messages BSMP devient plus appropriée. Il s'agit ici d'envoyer les données encapsulées dans des messages et de géré une file de messages reçus. Il n'est donc plus nécessaire d'allouer au préalable de l'espace pour une variable partagée. Les messages envoyés lors d'une super-étape ne seront disponible que lors de la super-étape suivante. La file des messages reçus est réinitialisée à la fin de chaque super-étape.

Dans la BSPlib les messages sont composés de deux parties, une partie de taille fixe contenant une étiquette *tag* destiné à aider à l'interprétation des messages reçus, et un partie de taille variable, appelée *payload*. La taille de la partie fixe peut être changée par une appel à `bsp_set_tag_size(i)` sur tous les processeur avec la même valeur i.

Les messages sont envoyés par la procédure `bsp_send(dest,tag,payload,nbytes)` où `dest` est l'identifiant du processeur qui recevra le message, `tag` est la partie fixe du message, `payload` l'emplacement du premier octet de la partie de taille variable du message. `nbytes` est la taille de cette partie.

`bsp_get_tag(status,tag)` affecte à la variable `status` la taille de la partie variable du premier message en attente de lecture , -1 si il n'y a pas de tel message, et affecte l'étiquette du premier message à la variable `tag`.

La procédure `bsp_qsize(nmess,accum_nbytes)` affecte le nombre de messages en attente de lecture à la variable `nmess` ainsi que la taille de toutes les parties *payload* cumulées des messages en attente.

La procédure `bsp_move(payload, nbytes)` affecte les `nbytes` premiers octets de la partie variable du message en attente de lecture à l'emplacement spécifié par `payload`. Le message est supprimé de la pile des messages reçus.

Chapitre 3

Modélisation de la partie DRMA de la BSPlib

Notre modélisation de la BSPlib, nommée BSP-IMP, est restreinte dans un premier temps aux communications de la partie DRMA. Elle est composée d'un ensemble d'instructions impératives classiques auxquelles sont ajoutées deux instructions de communication **put**$(dest, src, tgt)$ et **get**$(dest, rloc, tgt)$ modélisant les procédures `bsp_put` et `bsp_get`. *dest* et *src* sont des expressions arithmétiques, *tgt* et *rloc* sont des variables. Les procédure pour l'enregistrement des variables globales `bsp_push_reg` et `bsp_pop_reg` ne sont pas modélisées afin de ne pas surcharger la sémantique mais elles pourraient aisément être ajoutées. De même les instructions `bsp_begin` et `bsp_end` ne sont pas modélisées, tout programme est considéré comme s'exécutant en parallèle sur l'ensemble des processeurs, du début à la fin. Les fonctions `bsp_nprocs` et `bsp_pid` sont modélisées par les éléments arithmétiques particuliers **Nprocs** et **Pid**.

La syntaxe de BSP-IMP est décrite par la grammaire suivante. **Aexp** est l'ensemble des expressions arithmétiques tandis que **Bexp** est l'ensemble des expressions booléennes et **Com** l'ensemble des instructions ou programmes. Dans la suite X représentera toujours une variable de programme, élément de $\mathcal{V}$, ensemble dénombrable, et n une constante entière.

$$
\begin{array}{llll}
\textbf{Aexp} & a & ::= & n \mid X \mid a + a \mid a - a \mid a \times a \mid \textsf{Pid} \mid \textsf{Nprocs} \\
\textbf{Bexp} & b & ::= & \textsf{True} \mid \textsf{False} \mid a = a \mid a \leq a \mid \neg b \mid b \wedge b \mid b \vee b \\
\textbf{Com} & c & ::= & c; c \mid X := a \mid \textsf{if } b \textsf{ then } c \textsf{ end} \mid \textsf{while } b \textsf{ do } c \textsf{ end} \mid \textsf{skip} \\
 & & \mid & \textsf{put}(a, a, X) \mid \textsf{get}(a, X, X) \mid \textsf{sync}
\end{array}
$$

3.1 Sémantique

Nous définissons ici une sémantique opérationnelle pour le langage BSP-IMP. Cette sémantique ce compose de deux ensembles de règles, l'un définissant le comportement de chaque processus sur un processeur spécifique lors de la phase de calcul asynchrone, ces règles sont dite *locales*, tandis que les communications et la barrière de synchronisation, nécessitant la coopération de l'ensemble des processeurs, sont décrites par un ensemble de règles dites *globales*.

L'ensemble des règles locales défini la relation de transition $\xrightarrow[p]{i}$ entre

- Un triplet $\langle c,\ \sigma,\ r\rangle$ constitué d'un programme c, élément de l'ensemble **Com**, d'un environnement σ et d'une liste de requête de communication r ;
- Un triplet $\langle s,\ \sigma',\ r'\rangle$ constitué d'un état s du programme pouvant être Ok, Err ou Wait($c\prime$), d'un environnement et d'une liste de requêtes de communication .

Ok fait référence à l'état final normal d'une phase de calcul. Si une erreur c'est produite durant la phase de calcul l'état Err est utilisé. L'état Wait(c') est utilisé quant à lui pour modéliser un processus en attente de synchronisation globale pour lequel la commande c' doit être exécutée après synchronisation.

Un environnement décrit l'état de la mémoire par une fonction σ renvoyant une valeur pour chaque variable du programme. $\sigma[X \mapsto n]$ définit la fonction σ' telle que $\sigma'(Y) = \begin{cases} n & \text{si Y=X} \\ \sigma(Y) & \text{sinon} \end{cases}$, ce qui correspond au changement de valeur de X.

Une requête de communication peut être de la forme $\langle X_{@j} \leftarrow n\rangle$ pour une communication de type **put**(j, n, X), c.-à-d. que la valeur n sera copiée dans la variable X de la mémoire locale du processeur j, ou de la forme $\langle X_{@i} \leftarrow Y_{@j}\rangle$ pour une communication de type **get**(j, Y, X) effectuée depuis le processeur i, demandant la copie de la valeur de la variable Y au processeur j dans la variable locale X. Ces requêtes modélisent l'ensemble d'informations qui doivent être mémorisées pour effectuer les échanges de données au moment de la synchronisation.

La relation $\langle c,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s,\ \sigma',\ r'\rangle$ signifie "à partir de l'état mémoire initial σ et de la liste de requête de communication r, le programme c sera évalué au processeur i sur une machine parallèle à p processeurs dans l'état final s avec un état mémoire σ' une liste de requêtes de communication finale r'".

Les règles globales définissent la relation $\xrightarrow[p]{}$ entre :

- Un triplet $\langle\ C,\ \Sigma,\ R\ \rangle$ de vecteurs de taille p. C est le vecteur de programmes, initialement il s'agit d'un même programme c sur tous les processeur car BSP-IMP suit le paradigme SPMD. Ensuite l'exécution peut différer sur chaque processeur et C peut être écrit $[c_0, \ldots, c_{p-1}]^p$.
  Σ est un vecteur d'environnements (un par processeur) et R est un vecteur de listes de requêtes de communications (une par processeur). L'environnement (resp. la liste) au processeur i est notée $\Sigma[i]$ (resp. $R[i]$).
- Un triplet $\langle\ S,\ \Sigma',\ R'\ \rangle$ où S est l'état final d'exécution du programme. Ce peut être soit Ok soit Err. il ne s'agit pas ici d'un vecteur mais bien d'un état global du programme. Σ' et R' sont les vecteurs finaux des environnements et des listes de requêtes de communications.

3.1.1 Règles d'évaluation locale

Les règles de la sémantique opérationnelle pour l'évaluation locale des processus reposent sur une sémantique pour l'évaluation des expressions arithmétiques et booléennes. Nous présentons ci-dessous la sémantique pour les expressions arithmétiques qui comporte quelques règles spécifiques à notre langage. Les règles pour la sémantique des expressions booléennes sont les règles usuelles et sont reléguées à l'annexe A.

Les Opérateur arithmétiques sont associés a leur interprétation naturelles. Soit $\otimes$ un opérateur parmi $\{+,\ -,\ \times\}$

$$\frac{\langle a_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} n_1 \quad \langle a_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} n_2 \quad n_1 \otimes n_2 = n}{\langle a_1 \otimes a_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} n}$$

Les variables BSP sont interprétées en fonctions de la localisation du processus sur la machine parallèle et du nombre de processeurs

$$\frac{}{\langle \mathbf{Pid},\ \sigma,\ r\rangle \xrightarrow[p]{i} i} \qquad \frac{}{\langle \mathbf{Nprocs},\ \sigma,\ r\rangle \xrightarrow[p]{i} p}$$

L'évaluation des variables et constantes entières ne dépend que de l'environnement local σ

$$\frac{}{\langle X,\ \sigma,\ r\rangle \xrightarrow[p]{i} \sigma(X)} \qquad \frac{}{\langle n,\ \sigma,\ r\rangle \xrightarrow[p]{i} n}$$

Nous définissons à présent les règles d'évaluation locale des processus. l'intervalle des identifiants de processeurs, $[0, p-1]$ si le calcul est réparti sur p processeurs, est noté $\mathcal{P}$.

skip. La commande **skip** ne fait rien. Son but principal est d'indiquer qu'il n'y a rien à faire après une synchronisation.

$$\langle \mathbf{skip},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma,\ r\rangle \tag{3.1}$$

Séquence d'instructions. Pour une séquence d'instructions $c_0; c_1$ si l'évaluation de c_0 se termine sans erreur alors c_1 est évaluée dans l'environnement nouvellement obtenu, (règle 3.2), si l'évaluation de c_0 lève une erreur c_1 n'est pas évaluée et l'erreur est propagée (règle 3.3). Enfin si c_0 mène à un état d'attente de synchronisation alors c_1 est ajoutée aux instruction à effectuer après la synchronisation (règle 3.4).

$$\frac{\langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma'',\ r''\rangle \quad \langle c_1,\ \sigma'',\ r''\rangle \xrightarrow[p]{i} \langle s,\ \sigma',\ r'\rangle}{\langle c_0; c_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s,\ \sigma',\ r'\rangle} \tag{3.2}$$

$$\frac{\langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Err},\ \sigma',\ r'\rangle}{\langle c_0; c_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Err},\ \sigma',\ r'\rangle} \tag{3.3}$$

$$\frac{\langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(c_0'),\ \sigma',\ r'\rangle}{\langle c_0; c_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(c_0'; c_1),\ \sigma',\ r'\rangle} \tag{3.4}$$

Exécution conditionnelle. Lors de l'évaluation de **if** b **then** c **end** si la condition b est évaluée à vrai alors c est évaluée (règle 3.5), sinon il n'y a rien à faire (règle 3.6).

$$\frac{\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textbf{True} \quad \langle c,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s,\ \sigma',\ r'\rangle}{\langle \textbf{if } b \textbf{ then } c \textbf{ end},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s,\ \sigma',\ r'\rangle} \tag{3.5}$$

$$\frac{\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textbf{False}}{\langle \textbf{if } b \textbf{ then } c \textbf{ end},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma,\ r\rangle} \tag{3.6}$$

Boucle While. Lors de l'évaluation de **while** b **do** c **end** si la condition b est évaluée à faux alors il n'y a rien à faire (règle 3.7), sinon le corps c de la boucle est évalué. Si son évaluation mène à l'état Err alors l'évaluation de la boucle est arrêtée (règle 3.8) sinon **while** b **do** c **end** est évaluée dans l'environnement obtenu. Cette évaluation récursive peut mener à une demande de synchronisation (règle 3.10) ou non (règle 3.9).

$$\frac{\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textbf{False}}{\langle \textbf{while } b \textbf{ do } c \textbf{ end},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma,\ r\rangle} \tag{3.7}$$

$$\frac{\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textbf{True} \quad \langle c,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Err},\ \sigma',\ r'\rangle}{\langle \textbf{while } b \textbf{ do } c \textbf{ end},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Err},\ \sigma',\ r'\rangle} \tag{3.8}$$

$$\frac{\begin{array}{c}\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textbf{True} \qquad \langle c,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma'',\ r''\rangle \\ \langle \textbf{while } b \textbf{ do } c \textbf{ end},\ \sigma'',\ r''\rangle \xrightarrow[p]{i} \langle s,\ \sigma',\ r'\rangle\end{array}}{\langle \textbf{while } b \textbf{ do } c \textbf{ end},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s,\ \sigma',\ r'\rangle} \tag{3.9}$$

$$\frac{\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textbf{True} \qquad \langle c,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(c'),\ \sigma',\ r'\rangle}{\langle \textbf{while } b \textbf{ do } c \textbf{ end},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(c'; \textbf{while } b \textbf{ do } c \textbf{ end}),\ \sigma',\ r'\rangle} \tag{3.10}$$

Écriture mémoire distante. La commande **put**(a_1, a_2, X) a pour but d'écrire la valeur de l'expression a_2 dans la variable X au processeur ayant pour indice la valeur de l'expression a_1. Si l'expression a_1 a une valeur dans l'intervalle $\mathcal{P}$ la requête de communication est ajoutée à la liste locale (règle 3.12). La requête de communication $\langle X_{@j} \leftarrow n\rangle$ signifie que la valeur n doit être écrite dans la variable X du processeur j.

Si a_1 n'est pas un identifiant de processeur valide une erreur est levée (règle 3.11).

$$\frac{\langle a_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} j,\ j \in \mathcal{P} \quad \langle a_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} n}{\langle \textbf{put}(a_1, a_2, X),\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma,\ r.\langle X_{@j} \leftarrow n\rangle\rangle} \tag{3.11}$$

$$\frac{\langle a_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} j,\ j \notin \mathcal{P}}{\langle \textbf{put}(a_1, a_2, X),\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Err}_{\mathrm{PUT}},\ \sigma,\ r\rangle} \tag{3.12}$$

Lecture mémoire distante. Similaire à l'écriture distante.

$$\frac{\langle a_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} j,\ j \in \mathcal{P}}{\langle \textbf{get}(a_1, Y, X),\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma,\ r.\langle X_{@i} \leftarrow Y_{@j}\rangle\rangle} \tag{3.13}$$

$$\frac{\langle a_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} j,\ j \notin \mathcal{P}}{\langle \textbf{get}(a_1, Y, X),\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Err}_{\mathrm{GET}},\ \sigma,\ r\rangle} \tag{3.14}$$

Affectation locale. L'environnement local est modifié en changeant la valeur de $\sigma(X)$ pour la valeur de l'expression arithmétique a.

$$\frac{\langle a,\ \sigma,\ r\rangle \xrightarrow[p]{i} n}{\langle X := a,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma[X \mapsto n],\ r\rangle} \tag{3.15}$$

Attente de synchronisation. La commande **sync** demande une synchronisation globale. La barrière de synchronisation ne peut être franchie qu'au niveau globale. Au niveau local l'instruction **sync** mène donc dans un état d'attente Wait(**skip**).

$$\langle \textbf{sync},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(\textbf{skip}),\ \sigma,\ r\rangle \tag{3.16}$$

3.1.2 Règles d'évaluation globale

Les règles globales sont utilisées pour effectuer les communications, les synchronisations globales, et pour donner le résultat de l'exécution parallèle d'un programme. Dans les règles suivantes, l'état $\downarrow$ signifie la terminaison locale du calcul, c.-à-d. que le calcul local termine dans l'état Ok ou dans un état d'erreur mais pas dans un état d'attente de synchronisation. Suivant l'état des processus locaux après la phase de calcul asynchrone, quatre cas peuvent se présenter :

– tous les processus sont en attente de synchronisation. Dans ce cas les données sont échangées, ce qui est modélisé par la relation $\oplus$. Cette relation définit l'ensemble des triplets (Σ, R, Σ') où Σ est le vecteur des états mémoires initiaux, R le vecteur

des listes de communications à effectuer et Σ' un état mémoire possible après les communications. Cette relation est décrite plus en détail dans la section 3.2.1.

$$\frac{\begin{array}{c}\forall i \in \mathcal{P}, \langle c_i,\ \Sigma[i],\ R[i]\rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(c'_i),\ \Sigma'[i],\ R'[i]\rangle \\ \oplus(\Sigma', R, \Sigma'') \quad \langle\ [c'_0, \ldots, c'_{p-1}]^p,\ \Sigma'',\ \emptyset\ \rangle \underset{p}{\rightarrow} \langle\ \downarrow,\ \Sigma'',\ R''\ \rangle\end{array}}{\langle\ [c_0, \ldots, c_{p-1}]^p,\ \Sigma,\ R\ \rangle \underset{p}{\rightarrow} \langle\ \downarrow,\ \Sigma'',\ R''\ \rangle} \tag{3.17}$$

– si au moins un processus termine, dans l'état Ok ou dans l'état Err, tandis qu'au moins un autre processus demande une synchronisation globale, alors une erreur globale $\mathsf{Err}_{\text{SYNC}}$ est levée.

$$\frac{\begin{array}{c}\exists i \in \mathcal{P}, \langle c_i,\ \Sigma[i],\ R[i]\rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(c'_i),\ \Sigma'[i],\ R'[i]\rangle \\ \exists j \in \mathcal{P}, \langle c_j,\ \Sigma[j],\ R[j]\rangle \xrightarrow[p]{j} \langle\downarrow,\ \Sigma'[j],\ R'[j]\rangle\end{array}}{\langle\ [c_0, \ldots, c_{p-1}]^p,\ \Sigma,\ R\ \rangle \underset{p}{\rightarrow} \langle\ \mathsf{Err}_{\text{SYNC}},\ \Sigma',\ \emptyset\ \rangle} \tag{3.18}$$

– si tout les processus se terminent sans erreur l'état final global est Ok.

$$\frac{\forall i \in \mathcal{P}, \langle c_i,\ \Sigma[i],\ R[i]\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \Sigma'[i],\ R'[i]\rangle}{\langle\ [c_0, \ldots, c_{p-1}]^p,\ \Sigma,\ R\ \rangle \underset{p}{\rightarrow} \langle\ \mathsf{Ok},\ \Sigma',\ \emptyset\ \rangle} \tag{3.19}$$

– si au moins un processus local finit dans un état d'erreur $\mathsf{Err}_L \in \{\mathsf{Err}_{\text{GET}}; \mathsf{Err}_{\text{PUT}}\}$ et qu'aucun processus ne demande de synchronisation alors l'erreur Err_G est levée au niveau global.

$$\frac{\begin{array}{c}\exists i \in \mathcal{P}, \langle c_i,\ \Sigma[i],\ R[i]\rangle \xrightarrow[p]{i} \langle \mathsf{Err}_L,\ \Sigma'[i],\ R'[i]\rangle \\ \forall j \in \mathcal{P}, \langle c_i,\ \Sigma[j],\ R[j]\rangle \xrightarrow[p]{j} \langle\downarrow,\ \Sigma'[j],\ R'[j]\rangle\end{array}}{\langle\ [c_0, \ldots, c_{p-1}]^p,\ \Sigma,\ R\ \rangle \underset{p}{\rightarrow} \langle\ \mathsf{Err}_G,\ \Sigma',\ \emptyset\ \rangle} \tag{3.20}$$

3.1.3 Exemple d'utilisation de la sémantique

La sémantique est appliquée à un programme en construisant un arbre de dérivation. Cette arbre représente l'exécution d'un programme par l'application successive des règles de la sémantique opérationnelle.

Nous donnons ci-dessous l'arbre de dérivation d'un programme décalant le contenu de la variable X au processeur de droite. Ce programme termine en une super-étape BSP, il n'a donc qu'une phase de calcul asynchrone local.

Programme 1.
Right-shift $\equiv$
if (Pid $= 0$ **) then get**$(\mathsf{Nprocs} - 1, X, X)$ **end** ;
if (Pid > 0 **) then get**$(\mathsf{Pid} - 1, X, X)$ **end** ;
sync

Dérivation dans la sémantique locale

Au processeur 0 Dérivation locale de la première instruction :

$$d'_{l0} = \dfrac{\dfrac{}{\langle \textbf{Pid} = 0 \xrightarrow[4]{0} \textbf{True},\ \langle \textbf{get}(\textbf{Nprocs}-1, X, X),\ [X \mapsto n_0],\ \emptyset\rangle \xrightarrow[4]{0} \langle \mathsf{Ok},\ [X \mapsto n_0],\ \langle X_{@0} \leftarrow X_{@\textbf{Nprocs}-1}\rangle\rangle}\ (3.13)}{\langle \textbf{if Pid} = 0 \textbf{ then get}(\textbf{Nprocs}-1, X, X) \textbf{ end},\ [X \mapsto n_0],\ \emptyset\rangle \xrightarrow[4]{0} \langle \mathsf{Ok},\ [X \mapsto n_0],\ \langle X_{@0} \leftarrow X_{@\textbf{Nprocs}-1}\rangle\rangle}\ (3.5)$$

Dérivation locale de la deuxième instruction :

$$d''_{l0} = \dfrac{}{\langle \textbf{if Pid} > 0 \textbf{ then get}(\textbf{Pid}-1, X, X) \textbf{ end},\ [X \mapsto n_0],\ \langle X_{@0} \leftarrow X_{@\textbf{Nprocs}-1}\rangle\rangle \xrightarrow[4]{0} \langle \mathsf{Ok},\ [X \mapsto n_0],\ \langle X_{@0} \leftarrow X_{@\textbf{Nprocs}-1}\rangle\rangle}\ (3.6)$$

Dérivation locale de la demande de synchronisation :

$$d'''_{l0} = \dfrac{}{\langle \textbf{sync},\ [X \mapsto n_0],\ \langle X_{@0} \leftarrow X_{@\textbf{Nprocs}-1}\rangle\rangle \xrightarrow[4]{0} \langle \mathsf{Wait}(\textbf{skip}),\ [X \mapsto n_0],\ \langle X_{@0} \leftarrow X_{@\textbf{Nprocs}-1}\rangle\rangle}\ (3.16)$$

Dérivation locale pour l'ensemble du programme :

$$d_{l0} = \dfrac{d'_{l0} \quad \dfrac{d''_{l0} \quad d'''_{l0}}{\langle \textbf{if Pid} > 0 \textbf{ then get}(\textbf{Pid}-1, X, X) \textbf{ end}; \textbf{sync},\ [X \mapsto n_0],\ \emptyset\rangle \xrightarrow[4]{0} \langle \mathsf{Wait}(\textbf{skip}),\ [X \mapsto n_0],\ \langle X_{@0} \leftarrow X_{@\textbf{Nprocs}-1}\rangle\rangle}\ (3.2)}{\langle \texttt{Right-shift},\ [X \mapsto n_0],\ \emptyset\rangle \xrightarrow[4]{0} \langle \mathsf{Wait}(\textbf{skip}),\ [X \mapsto n_0],\ \langle X_{@0} \leftarrow X_{@\textbf{Nprocs}-1}\rangle\rangle}\ (3.2)$$

Au processeur d'identifiant $i > 0$ Dérivation locale de la première instruction :

$$d'_{li} = \dfrac{}{\langle \textbf{if Pid} = 0 \textbf{ then get}(\textbf{Nprocs}-1, X, X) \textbf{ end},\ [X \mapsto n_i],\ \emptyset\rangle \xrightarrow[4]{i} \langle \mathsf{Ok},\ [X \mapsto n_i],\ \emptyset\rangle}\ (3.6)$$

Dérivation locale de la deuxième instruction :

$$d''_{li} = \dfrac{\dfrac{}{\langle \textbf{Pid} > 0 \xrightarrow[4]{i} \textbf{True},\ \langle \textbf{get}(\textbf{Pid}-1, X, X),\ ,\ 4 \xrightarrow{i} [X \mapsto n_i]\emptyset\rangle \langle \mathsf{Ok},\ [X \mapsto n_i],\ \langle X_{@i} \leftarrow X_{@i-1}\rangle\rangle}\ (3.13)}{\langle \textbf{if Pid} > 0 \textbf{ then get}(\textbf{Pid}-1, X, X) \textbf{ end},\ [X \mapsto n_i],\ \emptyset\rangle \xrightarrow[4]{i} \langle \mathsf{Ok},\ [X \mapsto n_i],\ \langle X_{@i} \leftarrow X_{@i-1}\rangle\rangle}\ (3.5)$$

Dérivation locale de la demande de synchronisation :

$$d'''_{li} = \dfrac{}{\langle \textbf{sync},\ [X \mapsto n_i],\ \langle X_{@i} \leftarrow X_{@i-1}\rangle\rangle \xrightarrow[4]{i} \langle \mathsf{Wait}(\textbf{skip}),\ [X \mapsto n_i],\ \langle X_{@i} \leftarrow X_{@i-1}\rangle\rangle}\ (3.16)$$

Dérivation locale pour l'ensemble du programme :

$$d_{li} = \dfrac{d'_{li} \quad \dfrac{d''_{li} \quad d'''_{li}}{\langle \textbf{if Pid} > 0 \textbf{ then get}(\textbf{Pid}-1, X, X) \textbf{ end}; \textbf{sync},\ [X \mapsto n_i],\ \emptyset\rangle \xrightarrow[4]{i} \langle \mathsf{Wait}(\textbf{skip}),\ [X \mapsto n_i],\ \langle X_{@i} \leftarrow X_{@i-1}\rangle\rangle}\ (3.2)}{\langle \texttt{Right-shift},\ [X \mapsto n_i],\ \emptyset\rangle \xrightarrow[4]{i} \langle \mathsf{Wait}(\textbf{skip}),\ [X \mapsto n_i],\ \langle X_{@i} \leftarrow X_{@i-1}\rangle\rangle}\ (3.2)$$

Dérivation dans la sémantique globale

$$\dfrac{\forall i \in [0,3]d_{li} \quad \dfrac{\forall i \in [0,3] \vdash_l \langle \textsf{skip},\ \sigma',\ \emptyset\rangle \xrightarrow[4]{i} \langle \mathsf{Ok},\ \sigma',\ \emptyset\rangle}{\langle \begin{bmatrix} \textsf{skip} \\ \textsf{skip} \\ \textsf{skip} \\ \textsf{skip} \end{bmatrix}, \begin{bmatrix} [X \mapsto n_3] \\ [X \mapsto n_0] \\ [X \mapsto n_1] \\ [X \mapsto n_2] \end{bmatrix}, \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix} \rangle \xrightarrow[4]{} \langle \mathsf{Ok}, \begin{bmatrix} [X \mapsto n_3] \\ [X \mapsto n_0] \\ [X \mapsto n_1] \\ [X \mapsto n_2] \end{bmatrix}, \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix} \rangle}\ (3.19)}{\langle \begin{bmatrix} \texttt{Right shift} \\ \texttt{Right shift} \\ \texttt{Right shift} \\ \texttt{Right shift} \end{bmatrix}, \begin{bmatrix} [X \mapsto n_0] \\ [X \mapsto n_1] \\ [X \mapsto n_2] \\ [X \mapsto n_3] \end{bmatrix}, \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix} \rangle \xrightarrow[4]{} \langle \mathsf{Ok}, \begin{bmatrix} [X \mapsto n_3] \\ [X \mapsto n_0] \\ [X \mapsto n_1] \\ [X \mapsto n_2] \end{bmatrix}, \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix} \rangle}\ (3.17)$$

3.2 Déterminisme de la sémantique

Nous discutons dans cette section du déterminisme de la sémantique précédemment définie. Ce déterministe est fortement dépendant de la relation définissant les échanges de données lors de la barrière de synchronisation. Nous proposons donc différentes définitions de cette relation.

3.2.1 Évaluation des communications

L'évaluation des requêtes de communication telle que décrite dans la description informelle de la BSPlib n'est pas déterministe. Nous donnons en premiers lieu une définition de la relation $\oplus$ modélisant le comportement de la BSPlib. Nous proposons ensuite des modifications pour rendre déterministe l'évaluation des requêtes communications, soit en faisant de la relation obtenue par l'opérateur de la BSPlib une fonction en fusionnant les valeurs associées à une même variable, soit en modifiant les règles de la sémantique pour lever une erreur dans les cas où l'opérateur de la BSPlib s'avère non déterministe.

Version BSPlib

Les communications de types **put** et **get** sont évaluées séparément dans la BSPlib, les communications de type **get** étant effectuées en premier. La relation $\oplus$ a donc la définition suivante :

$$\oplus = \{(\Sigma, R, \Sigma') \mid \text{eval-get}(\Sigma, R, \Sigma'') \wedge \text{eval-put}(\Sigma'', R, \Sigma')\}$$

où

$$\text{eval-get} \quad = \left\{(\Sigma, R, \Sigma'') \mid \quad \forall i \in \mathcal{P},\ \Sigma''[i](X) = \begin{cases} \Sigma[j](Y) & \text{Si } \exists \langle X_{@i} \leftarrow Y_{@j}\rangle \in R[i] \\ \Sigma[i](X) & \text{Sinon.} \end{cases} \right\}$$

et

$$\text{eval-put} \quad = \left\{(\Sigma'', R, \Sigma')| \quad \forall i \in \mathcal{P},\ \Sigma'[I](X) = \begin{cases} v & \text{Si } \exists \langle X_{@i} \leftarrow v\rangle \in R \\ \Sigma''[i](X) & \text{Sinon.} \end{cases} \right\}$$

Version déterministe (fonction)

Une possibilité pour rendre déterministe l'évaluation des requêtes de communication est de fusionner les valeurs reçues pour une même variable via un opérateur $\odot$ (commutatif pour que l'ordre des requêtes n'affecte pas le résultat final). Cet opérateur pourrait éventuellement être défini par le programmeur ou choisi parmi un ensemble d'opérations

courantes telles que max, min, l'addition, la multiplication, etc. Ainsi la relation $\oplus$ devient une fonction associant un unique vecteur d'environnements Σ' à un couple (Σ, R). On obtient alors

$$\text{eval-get} = \{(\Sigma, R, \Sigma'') \mid \forall i \in \mathcal{P}, \Sigma'' = \left\{ \begin{array}{cl} \bigodot_{\{\langle X_{@i} \leftarrow Y_{@j} \rangle \in R\}} \Sigma[j](Y) & \text{Si } \exists \langle X_{@i} \leftarrow Y_{@j} \rangle \in R[i] \\ \Sigma[i](X) & \text{Sinon.} \end{array} \right. \}$$

et

$$\text{eval-put} = \{(\Sigma'', R, \Sigma') \mid \forall i \in \mathcal{P}, \Sigma'' = \left\{ \begin{array}{cl} \bigodot_{\{v \mid \langle X_{@i} \leftarrow v \rangle \in R\}} v & \text{Si } \exists \langle X_{@i} \leftarrow v \rangle \in R \\ \Sigma[i](X) & \text{Sinon.} \end{array} \right. \}$$

Version déterministe (erreur)

Les échanges de données peuvent aussi être déterminisés par une modification de la sémantique du langage levant une erreur lorsque plusieurs communications écrivent dans la même variable.

La règle (3.17) est alors remplacée par les deux règles suivantes :

$$\frac{\begin{array}{c} \forall i \in \mathcal{P}, \langle c_i,\ \sigma[i],\ R[i] \rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(c'_i),\ \sigma'[i],\ R'[i] \rangle \text{ avec } c'_i \in \mathbf{Com} \cup \{\epsilon\} \\ \mathit{Deterministic}(\Sigma', R) \\ \langle\ [c'_0, \ldots, c'_{p-1}]^p,\ \oplus(\Sigma', R, \Sigma''),\ \emptyset\ \rangle \xrightarrow[p]{} \langle\ \downarrow,\ \Sigma'',\ R''\ \rangle \end{array}}{\langle\ [c_0, \ldots, c_{p-1}]^p,\ \Sigma,\ R\ \rangle \xrightarrow[p]{} \langle\ \downarrow,\ \Sigma'',\ R''\ \rangle} \tag{3.21}$$

$$\frac{\begin{array}{c} \forall i \in \mathcal{P}, \langle c_i,\ \Sigma[i],\ R[i] \rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(c'_i),\ \Sigma'[i],\ R'[i] \rangle \text{ avec } c'_i \in \mathbf{Com} \cup \{\epsilon\} \\ \neg \mathit{Deterministic}(\Sigma', R') \end{array}}{\langle\ [c_0, \ldots, c_{p-1}]^p,\ \Sigma,\ R\ \rangle \xrightarrow[p]{} \langle\ \mathsf{Err}_{\mathrm{DET}},\ \Sigma',\ R'\ \rangle} \tag{3.22}$$

où $\mathit{Deterministic}(\Sigma, R)$ est vrai si et seulement si les communications affectent aux variables au plus une valeur, c.-à-d.

$$\begin{cases} \langle X_{@i} \leftarrow Y_{@j} \rangle,\ \langle X_{@i} \leftarrow Z_{@k} \rangle \in R & \Longrightarrow \Sigma[j]Y = \Sigma[k]Z \\ \langle X_{@i} \leftarrow v \rangle,\ \langle X_{@i} \leftarrow w \rangle \in R & \Longrightarrow v = w \end{cases}$$

Cette modification conserve le déterminisme de la sémantique mais ajoute un état d'erreur lorsque les communications entrent en conflit pour l'écriture dans une même variable.

Il est cependant difficile de vérifier statiquement que les communications n'entrent pas en conflit et donc que le programme n'aboutit pas à une telle erreur.

3.2.2 Démonstration du déterminisme

La preuve est décomposée en deux parties, la première montrant le déterminisme de la sémantique d'évaluation locale des processus par induction sur les arbres de dérivation, la seconde utilisant le même principe pour montrer le déterminisme de l'évaluation globale des programmes BSP-IMP en utilisant pour $\oplus$ la définition fonctionnelle.

Les démonstration sont faites par induction sur les arbres de dérivation. Pour les démonstration nous utilisons l'ordre $\prec$ sur les arbres de dérivation. Cet ordre décrit un arbre d_1 comme inférieur à d_2 ($d_1 \prec d_2$) si d_1 est une sous-dérivation propre de d_2, c'est à dire que l'arbre d_1 est un sous arbre de d_2.

La notation $d \vdash_g A \to B$ (resp.$\vdash_l$) signifie que $A \to B$ est conséquence de l'arbre de dérivation d utilisant les règles globales (resp. locales). $\vdash_g A \to B$ signifie qu'il existe un arbre concluant à $A \to B$.

Déterminisme de la sémantique locale

La sémantique des opérations arithmétiques et booléennes est clairement déterministe, aussi nous considérerons comme minimal pour $\prec$ tout arbre de dérivation dont la dernière dérivation n'a soit aucune prémisse, soit uniquement des prémisses ne faisant intervenir que les règles des sémantiques arithmétiques et booléennes.

Lemme 1 (déterminisme local)**.** *La sémantique d'évaluation locale des programmes est déterministe. c.-à-d.* $d \vdash_l \langle c,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_1,\ \sigma',\ r'\rangle$ *implique que pour toutes dérivations* $d' \vdash_l \langle c,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_2,\ \sigma'',\ r''\rangle$ *on a* $s_1 = s_2$, $\sigma' = \sigma''$ *et* $r' = r''$

Démonstration. Nous démontrons par cas suivant la structure de $c \in \mathbf{Com}$ que pour tout programme c et quelque soit l'état de la mémoire σ et la liste de requêtes de communication r, si il existe un arbre de dérivation d_1 tel que $d_1 \vdash_l \langle c,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle f_1,\ \sigma_1,\ r_1\rangle$ et un arbre d_2 tel que $d_2 \vdash_l \langle c,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle f_2,\ \sigma_2,\ r_2\rangle$ alors la propriété

$$P \equiv (s_1 = s_2 \in \mathsf{NF} = \{\mathsf{Err}; \mathsf{Wait}(c'); \mathsf{Ok}\},\ \sigma_1 = \sigma_2 \text{ et } r_1 = r_2)$$

est vérifiée :

– $c \equiv \textbf{skip}$. La seul règle s'appliquant est la règle (3.1) et

$$d_1 = d_2 = \frac{}{\langle \textbf{skip},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma,\ r\rangle}.$$

La propriété P est donc vérifiée.

– $c \equiv v := a$. Alors les dérivations d_1 et d_2 sont de la forme

$$d_1 = \frac{\dfrac{}{\langle a,\ \sigma,\ r\rangle \xrightarrow[p]{i} n_1}}{\langle X := a,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma[X \mapsto n_1],\ r\rangle}$$

et

$$d_2 = \frac{\dfrac{}{\langle a,\ \sigma,\ r\rangle \xrightarrow[p]{i} n_2}}{\langle X := a,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma[X \mapsto n_2],\ r\rangle}.$$

Comme l'évaluation des expressions arithmétiques est déterministe, $n_1 = n_2$, et donc $\sigma[X \mapsto n_1] = \sigma[X \mapsto n_2]$. La liste de requêtes de communication r reste inchangée et l'état final est le même dans les deux cas (Ok), donc la propriété P est vérifiée.

– $c \equiv$ **if** b **then** c_0 **end**. L'expression booléenne b est évaluée de manière déterministe en **True** ou **False**.
  – Si $\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i}$ **False**, la seule règle s'appliquant, 3.6, donne

$$d_1 = d_2 = \frac{\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textbf{False}}{\langle \textbf{if } b \textbf{ then } c \textbf{ end},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma,\ r\rangle}.$$

  Donc P est vérifiée
  – Si $\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i}$ **True**, la règle (3.5) donne

$$d_1 = \frac{\dfrac{}{\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textbf{True}} \quad d_1' = \dfrac{\vdots}{\langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_1,\ \sigma_1,\ r_1\rangle}}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ end},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_1,\ \sigma_1,\ r_1\rangle}$$

  et

$$d_2 = \frac{\dfrac{}{\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textbf{True}} \quad d_2' = \dfrac{\vdots}{\langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_2,\ \sigma_2,\ r_2\rangle}}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ end},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_2,\ \sigma_2,\ r_2\rangle}.$$

  Par définition, $d_1' \prec d_1$, donc par induction $d_1' \vdash_l \langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_1,\ \sigma_1,\ r_1\rangle$ et $d_2' \vdash_l \langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_2,\ \sigma_2,\ r_2\rangle$ impliquent $\sigma_1 = \sigma_2,\ r_1 = r_2$ et $s_1 = s_2 \in \mathsf{NF}$.
  P est donc vérifiée.

– $c \equiv$ **while** b **do** c_0 **end**. b est évaluée de manière déterministe en **True** ou **False**. Si $\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i}$ **False**, la règle (3.7) donne

$$d_1 = d_2 = \frac{\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textbf{False}}{\langle \textbf{while } b \textbf{ do } c_0 \textbf{ end},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma,\ r\rangle},$$

  et P est vérifiée.
  Sinon l'arbre de dérivation d_1 termine par une instance de la règle (3.9),(3.8) ou (3.10).

– Si l'arbre d_1 termine par une instance de la règle (3.9), il est de la forme

$$d_1 = \dfrac{\dfrac{}{\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textbf{True}} \quad d'_1 = \dfrac{\vdots}{\langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma'_1,\ r'_1\rangle} \qquad d''_1 = \dfrac{\vdots}{\langle \textbf{while } b \textbf{ do } c_0 \textbf{ end},\ \sigma'_1,\ r'_1\rangle \xrightarrow[p]{i} \langle s_1,\ \sigma_1,\ r_1\rangle}}{\langle \textbf{while } b \textbf{ do } c_0 \textbf{ end},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_1,\ \sigma_1,\ r_1\rangle}.$$

Comme $d'_1 \prec d_1$, par hypothèse d'induction, pour tout d'_2 tel que

$$d'_2 \vdash_l \langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s'_2,\ \sigma'_2,\ r'_2\rangle$$

nous avons $s'_2 = \mathsf{Ok}$, $\sigma'_2 = \sigma'_1$ et $r_2 = r_1$. Soit d_2 tel que

$$d_2 \vdash_l \langle \textbf{while } b \textbf{ do } c_0 \textbf{ end},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_2,\ \sigma_2,\ r_2\rangle,$$

alors d_2 est forcément de la forme

$$\dfrac{\dfrac{}{\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textbf{True}} \quad d'_2 = \dfrac{\vdots}{\langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma'_2,\ r'_2\rangle} \qquad d''_2 = \dfrac{\vdots}{\langle \textbf{while } b \textbf{ do } c_0 \textbf{ end},\ \sigma'_2,\ r'_2\rangle \xrightarrow[p]{i} \langle s_2,\ \sigma_2,\ r_2\rangle}}{\langle \textbf{while } b \textbf{ do } c_0 \textbf{ end},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_2,\ \sigma_2,\ r_2\rangle}$$

car d'_2 est incompatible avec les prémisses des règles (3.8) et (3.10). Comme $\sigma'_2 = \sigma'_1$ et $r'_2 = r'_1$, on obtient $d''_2 \vdash_l \langle \textbf{while } b \textbf{ do } c_0 \textbf{ end},\ \sigma'_1,\ r'_1\rangle \xrightarrow[p]{i} \langle s_2,\ \sigma_2,\ r_2\rangle$, et par hypothèse d'induction nous avons $s_2 = s_1 \in \mathsf{NF},\ \sigma_2 = \sigma_1,\ r_2 = r_1$ car $d''_1 \prec d_1$. Donc P est vérifiée.

– Si l'arbre d_1 termine par une instance de la règle (3.8),

$$d_1 = \dfrac{\dfrac{}{\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textbf{True}} \quad d'_1 = \dfrac{\vdots}{\langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Err},\ \sigma_1,\ r_1\rangle}}{\langle \textbf{while } b \textbf{ do } c_0 \textbf{ end},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_1,\ \sigma_1,\ r_1\rangle}.$$

$d'_1 \prec d_1$ donc par hypothèse d'induction $d'_2 \vdash_l \langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_2,\ \sigma_2,\ r_2\rangle$ implique $s_2 = \mathsf{Err}$, $\sigma_2 = \sigma_1$ and $r_2 = r_1$ pour toute dérivation d'_2. Les règles (3.9) et (3.10)

ne peuvent donc pas être appliquées et pour tout

$$d_2 = \dfrac{\dfrac{}{\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textbf{True}} \quad d_2' = \dfrac{\vdots}{\langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Err},\ \sigma_2,\ r_2\rangle}}{\langle \textbf{while } b \textbf{ do } c_0 \textbf{ end},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Err},\ \sigma_2,\ r_2\rangle},$$

l'hypothèse d'induction donne $s_1 = s_2 = \mathsf{Err} \in \mathsf{NF}$, $\sigma_1 = \sigma_2$ et $r_1 = r_2$.
P est donc vérifiée.

– Si l'arbre d_1 termine par une instance de la règle (3.10),

$$d_1 = \dfrac{\dfrac{}{\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textbf{True}} \quad d_1' = \dfrac{\vdots}{\langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(c_0'),\ \sigma_1,\ r_1\rangle}}{\langle \textbf{while } b \textbf{ do } c_0 \textbf{ end},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(c_0'; \textbf{while } b \textbf{ do } c_0 \textbf{ end}),\ \sigma_1,\ r_1\rangle}.$$

$d_1' \prec d_1$ donc par induction $d_2' \vdash_l \langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_2',\ \sigma_2,\ r_2\rangle$ implique $s_2' = \mathsf{Wait}(c_0')$, $\sigma_2 = \sigma_1$ et $r_2 = r_1$ pour toute dérivation d_2'. Les règles (3.9) et (3.8) ne sont pas applicables et donc pour tout arbre de dérivation d_2,

$$d_2 = \dfrac{\dfrac{\vdots}{\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textbf{True}} \quad d_2' = \dfrac{\vdots}{\langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(c_0'),\ \sigma_2,\ r_2\rangle}}{\langle \textbf{while } b \textbf{ do } c_0 \textbf{ end},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(c_0'; \textbf{while } b \textbf{ do } c_0 \textbf{ end}),\ \sigma_2,\ r_2\rangle}$$

et donc $s_1 = s_2 = \mathsf{Wait}(c'; \textbf{while } b \textbf{ do } c_0 \textbf{ end}) \in \mathsf{NF}$, $\sigma_1 = \sigma_2$ et $r_1 = r_2$, donc P est vérifiée.

Quelque soit la règle applicable, si $c \equiv \textbf{while } b \textbf{ do } c_0 \textbf{ end}$, P est vérifiée.

– $c \equiv c_0; c_1$. Toutes les règles applicables dans ce cas (3.2,3.3, 3.4) ont pour prémisse $\langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_1',\ \sigma_1',\ r_1'\rangle$ avec pour chaque règle une valeur différente pour s_1'. Comme par hypothèse d'induction $\vdash_l \langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_2',\ \sigma_2',\ r_2'\rangle$ implique $s_1' = s_2' \in \mathsf{NF}$, $\sigma_1' = \sigma_2'$ et $r_1' = r_2'$, les règles applicables sont mutuellement exclusives.

– Si $s_1' = \mathsf{Err}$ seule la règle (3.3) s'applique donnant

$$d_1 = d_2 = \dfrac{\langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Err},\ \sigma',\ r'\rangle}{\langle c_0; c_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Err},\ \sigma',\ r'\rangle}$$

– Si $s_1' = \mathsf{Wait}(c')$ la règle (3.4) donne

$$d_1 = d_2 = \dfrac{\langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(c'),\ \sigma',\ r'\rangle}{\langle c_0; c_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(c'; c_1),\ \sigma',\ r'\rangle}.$$

– Si $s'_1 = \mathsf{Ok}$ alors

$$d_1 = \frac{\langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma',\ r'\rangle \quad \langle c_1,\ \sigma',\ r'\rangle \xrightarrow[p]{i} \langle s_1,\ \sigma_1,\ r_1\rangle}{\langle c_0; c_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_1,\ \sigma_1,\ r_1\rangle}$$

et

$$d_2 = \frac{\langle c_0,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma',\ r'\rangle \quad \langle c_1,\ \sigma',\ r'\rangle \xrightarrow[p]{i} \langle s_2,\ \sigma_2,\ r_2\rangle}{\langle c_0; c_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s_2,\ \sigma_2,\ r_2\rangle}$$

avec $s_1 = s_2 \in \mathsf{NF}$, $\sigma_1 = \sigma_2$ et $r_1 = r_2$.

Dans les trois cas la propriété P est vérifiée.

– $c \equiv \textbf{get}(a, v_0, v_1)$. Comme l'évaluation de a est déterministe, les deux seules règles applicables dans ce cas (3.13 et 3.14) sont mutuellement exclusives. Deux cas se présentent donc selon la valeur j de l'expression a. Si j est un identifiant de processeur valide

$$d_1 = d_2 = \frac{\langle a,\ \sigma,\ r\rangle \xrightarrow[p]{i} j \qquad j \in \mathcal{P}}{\langle \textbf{get}(a, v_0, v_1),\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma,\ r.\langle X_{@i} \leftarrow Y_{@j}\rangle\rangle}.$$

sinon

$$d_1 = d_2 = \frac{\langle a,\ \sigma,\ r\rangle \xrightarrow[p]{i} j \qquad j \notin \mathcal{P}}{\langle \textbf{get}(a, v_0, v_1),\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Err}_{\mathrm{GET}},\ \sigma,\ r\rangle}$$

Dans les deux cas P est vérifiée.

– $c \equiv \textbf{put}(a_1, a_2, v)$. idem avec les règles(3.11) et (3.12).

– $c \equiv \textbf{sync}$. Alors la seul règle s'appliquant est la règle (3.16) et

$$d_1 = d_2 = \frac{}{\langle \textbf{sync},\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(),\ \sigma,\ r\rangle}.$$

La propriété P est donc vérifiée.

Les éléments minimaux pour $\prec$ (cas de base de l'induction) sont les arbres de dérivation terminant par une instance d'une des règles (3.1, 3.6, 3.7, 3.11, 3.12, 3.13, 3.14, 3.15 ou 3.16). Ces cas vérifient la propriété P. Les arbres terminant par l'une des autre règles (3.2, 3.3, 3.4, 3.5,3.9, 3.8, 3.10) vérifient la propriété P si elle est vérifiée pour toute sous-dérivation propre. Nous avons donc bien vérifié par induction le déterminisme de la sémantique locale exprimé par le lemme 1. □

Déterminisme global

Théorème 1 (Déterminisme global)**.** *La sémantique d'évaluation globale des programmes est déterministe si l'évaluation des communications est déterministe.*
c.-à-d. $d \vdash_g \langle\ [Pr, \ldots, Pr]^p,\ \Sigma,\ R\ \rangle \underset{p}{\rightarrow} \langle\ S_1,\ \Sigma',\ R'\ \rangle$ *implique que pour toutes dérivations* $d' \vdash_g \langle\ [Pr, \ldots, Pr]^p,\ \Sigma,\ R\ \rangle \underset{p}{\rightarrow} \langle\ s_2,\ \Sigma'',\ R''\ \rangle$ *on a* $s_2 = s_2$, $\Sigma' = \Sigma''$ *et* $R' = R''$.

À nouveau la preuve est faite par induction en utilisant le même ordre sur les arbres de dérivation. Cependant les arbres considérés ici sont ceux utilisant les règles de la sémantique globale.

Démonstration. Notons tout d'abord que les règles de la sémantique globale sont mutuellement exclusives. En effet, l'évaluation locale des programmes étant déterministe, si tous les prémisses d'une des règle sont satisfaits alors ceux des autres règles ne peuvent pas l'être.

– Si l'arbre de dérivation finit par la règle (3.17) au niveau local pour chaque processus i, il existe une dérivation $dl_i \vdash_l \langle c_i,\ \Sigma[i],\ R[i]\rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(c'_i),\ \Sigma[i]',\ R[i]'\rangle$ et $(\Sigma', R, \Sigma'') \in \oplus$ au niveau global il existe un arbre de dérivation

$$d'_1 = \frac{\vdots}{\langle\ [c'_0, \ldots, c'_{p-1}]^p,\ \Sigma'',\ \emptyset\ \rangle \underset{p}{\rightarrow} \langle\ \downarrow_1,\ \Sigma''_1,\ R''_1\ \rangle}$$

donc

$$d_1 = \frac{dl_0 \quad \ldots \quad dl_{\mathbf{Nprocs}-1} \quad d'_1 \quad \oplus(\Sigma', R, \Sigma'')}{\langle\ [c_0, \ldots, c_{p-1}]^p,\ \Sigma,\ R\ \rangle \rightarrow \langle\ \downarrow_1,\ \Sigma''_1,\ R''_1\ \rangle}.$$

Comme les dérivations locales sont déterministes,

$$d_2 = \frac{dl'_0 \ldots dl'_{\mathbf{Nprocs}-1} \quad \oplus(\Sigma', R', \Sigma'') \quad d'_2 = \dfrac{\vdots}{\langle\ [c'_0, \ldots, c'_{p-2}]^p,\ \Sigma'',\ \emptyset\ \rangle \rightarrow \langle\ \downarrow_2,\ \Sigma_2,\ R_2\ \rangle}}{\langle\ [c_0, \ldots, c_{p-1}]^p,\ \Sigma,\ R\ \rangle \rightarrow \langle\ \downarrow_2,\ \Sigma_2,\ R_2\ \rangle}$$

avec $dl'_i \vdash_l \langle \mathsf{Wait}(c'_i),\ \Sigma[i]',\ R_2[i]'\rangle$. De plus, la relation $\oplus(\Sigma', R', \Sigma'')$ définie une fonction associant un unique environnement Σ'' au couple (Σ', R'), et par hypothèse d'induction, comme $d'_1 \prec d_1$, la forme de d'_2 implique $\downarrow_2 = \downarrow_1 \in \{\mathsf{Err}_{\text{SYNC}}; \mathsf{Err}_G; \mathsf{Ok}\}$, $\Sigma_2 = \Sigma_1$ et $R_2 = R_1$. P_g est donc vérifiée.

– Si l'arbre de dérivation finit par la règle (3.18), (3.19) ou (3.20),

$$d_1 = \frac{\vdots}{\langle\ [c_0, \ldots, c_{p-1}]^p,\ \Sigma,\ R\ \rangle \rightarrow \langle\ S_1,\ \Sigma_1,\ \emptyset\ \rangle}$$

avec $S_1 \in \{\mathsf{Err}_{\text{SYNC}}; \mathsf{Err}_G; \mathsf{Ok}\}$. Comme les prémisses de ces règles n'impliquent que des évaluations locales (précédemment montrées déterministes), toute autre dérivation

$$d_2 \vdash_g \langle\ [c_0, \ldots, c_{p-1}]^p,\ \Sigma,\ R\ \rangle \rightarrow \langle\ S_1,\ \Sigma_1,\ \emptyset\ \rangle$$

mène forcément au même état puisque les prémisses de la dernière règle appliquée sont les mêmes. Donc $\Sigma_2 = \Sigma_1$, $R_2 = R_1$ et $\downarrow_2 = \downarrow_1 \in \{\mathsf{Err}_{\text{SYNC}}; \mathsf{Err}_G; \mathsf{Ok}\}$ et P_g est vérifiée.

□

3.3 Synchronisation sans erreur

3.3.1 Définition

Une propriété intéressante à vérifier pour les programmes BSP-IMP est l'absence d'erreur de synchronisation. Un telle erreur a lieu si un processus n'atteint pas le même nombre de barrière de synchronisation que les autres. La possible présence de **sync** dans ou après une boucle rend ce problème indécidable en général. Cependant la propriété peut être vérifiée pour un sous-ensemble de programmes BSP-IMP. Nous caractérisons ceux pour lesquels chaque instruction **if** (ou **while**) contenant **sync** a une condition booléenne s'évaluant à la même valeur sur chaque processeur. On dit d'un tel programme qu'il a la propriété de *synchronisations répliquées*.

Cependant cette propriété n'est pas suffisante. Pour s'assurer que tous les processeurs atteignent le même nombre de barrières de synchronisation, ils doivent être exempts de toute erreur locale qui arrêterait l'évaluation normale du programme. Une erreur est levée au niveau local si et seulement si il y a une erreur de communication. Une telle erreur se produit lorsque l'expression arithmétique donnant la destination de la communication s'évalue en dehors de l'intervalle $\mathcal{P}$ des identifiants de processeurs.

Un programme *communique correctement* si pour toute instruction **put** ou **get** du programme, avec a comme premier paramètre, nous avons :

$$\forall\sigma\forall r\forall i\forall p \quad \langle a,\ \sigma,\ r\rangle \xrightarrow[p]{i} n \in \mathcal{P} \quad .$$

3.3.2 Conditions suffisantes pour la synchronisation sans erreur

Lemme 2. *Si c à la propriété de synchronisations répliquées et communique correctement, alors pour tout i et j dans l'intervalle des indices de processeur $\mathcal{P}$, quelque soient les environnements locaux σ, σ', et les listes de messages en attente r et r', les dérivations $\vdash_l \langle c,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle f_i,\ \sigma'',\ r''\rangle$ et $\vdash_l \langle c,\ \sigma',\ r'\rangle \xrightarrow[p]{j} \langle f_j,\ \sigma''',\ r'''\rangle$ impliquent $f_i = f_j \in \{\mathsf{Ok}; \mathsf{Wait}(c')\}$*

Démonstration. Dans la suite $\langle c,\ \sigma,\ r\rangle \xrightarrow[p]{i} \langle s,\ \sigma',\ r'\rangle$ sera abrégé en $c \xrightarrow[p]{i} s$. En effet l'état de l'environnement d'exécution n'a aucune importance pour ce qui suit.

Soit c une séquence d'instructions ayant la propriété de synchronisations répliquées.

Nous démontrons le lemme 2 par induction sur les arbres de dérivation et par cas suivant la structure de c.

- Si $c \equiv \mathbf{put}(a_1, a_2, v)$, $c \equiv \mathbf{get}(a, v_1, v_2)$, $c \equiv v := a$ ou $c \equiv \mathbf{skip}$, comme nous avons supposé que le programme est sans erreur de communication, il est évident que pour tout $p \in \mathbb{N}$ et pour tout $i,\ j\ \in [0, p-1]$,nous avons $f_i = f_j = \mathsf{Ok}$. (Règles 3.1, 3.12, 3.13 et 3.15)
- Si $c \equiv \mathbf{sync}$, alors quelques soient i, j et p, nous avons $f_i = f_j = \mathsf{Wait}(\mathbf{skip})$. (Règle 3.16)
- Si $c \equiv c_1; c_2$ alors deux cas se présentent suivant que ce soit la règle (3.4) qui s'applique ou la règle (3.2).

– si la règle (3.4) s'applique, alors

$$d = \frac{c_0 \xrightarrow[p]{i} \mathsf{Wait}(c_0')}{c_0; c_1 \xrightarrow[p]{i} \mathsf{Wait}(c_0'; c_1)},$$

donc par induction $\vdash_l \ c_0 \ \xrightarrow[p]{j} \ \mathsf{Wait}(c_0')$, pour tout processeur j et donc $\vdash_l c_0; c_1 \xrightarrow[p]{j} \mathsf{Wait}(c_1)$.

– Sinon c'est la règle (3.2) qui s'applique, alors

$$d = \frac{c_0 \xrightarrow[p]{i} \mathsf{Ok} \quad c_1 \xrightarrow[p]{i} s}{c_0; c_1 \xrightarrow[p]{i} s}$$

et donc, par induction, $\vdash_l c_0 \xrightarrow[p]{j} \mathsf{Ok}$ et $\vdash_l c_1 \xrightarrow[p]{j} s$ pour tout $j \in \mathcal{P}$, donc $\vdash_l c_0; c_1 \xrightarrow[p]{j} s$ pour tous $j \in \mathcal{P}$.

– Si $c \equiv$ **if** b **then** c' **end** alors deux cas se présentent suivant que c' contienne **sync** ou non.
  – si c' ne contient pas **sync**, alors pour tout processeur $j \vdash_l c' \xrightarrow[p]{j} \mathsf{Ok}$. Ainsi, que b soit évalué à **True** ou **False**, $\vdash_l c \xrightarrow[p]{j} \mathsf{Ok}$ par la règle 3.6 ou 3.5.
  – si c' contient **sync**, alors $\vdash_l c' \xrightarrow[p]{i} \mathsf{Wait}(c'')$ et pour tout identifiant $j \in \mathcal{P}$ l'hypothèse d'induction donne $\vdash_l c' \xrightarrow[p]{j} \mathsf{Wait}(c'')$. Comme c' contient **sync**, b est évalué identiquement sur tous les processeurs. Ainsi b est évalué partout soit à **True** et on a $\forall j \in \mathcal{P} \ \vdash_l c \xrightarrow[p]{j} \mathsf{Wait}(c'')$ soit à **False** et on a $\forall j \in \mathcal{P} \ \vdash_l c \xrightarrow[p]{j} \mathsf{Ok}$. Dans les deux cas le lemme est vérifié.

– Si $c \equiv$ **while** b **do** c' **end**, à nouveau deux cas se présentent suivant que c' contienne **sync** ou non.
  – si c' ne contient pas **sync**, alors pour tout processeur j, $\vdash_l c' \xrightarrow[p]{j} \mathsf{Ok}$. Ainsi, que b soit évalué à **True** ou **False**, $\vdash_l c \xrightarrow[p]{j} \mathsf{Ok}$ par la règle 3.7 ou 3.9.
  – si c' contient **sync**, alors $\vdash_l c' \xrightarrow[p]{i} \mathsf{Wait}(c'')$ et pour tout identifiant $j \in \mathcal{P}$ l'hypothèse d'induction donne $\vdash_l c' \xrightarrow[p]{j} \mathsf{Wait}(c'')$. Comme c' contient **sync**, b est évalué identiquement sur tous les processeurs. Ainsi b est évalué partout soit à **True** et on a $\forall j \in \mathcal{P} \ \vdash_l c \xrightarrow[p]{j} \mathsf{Wait}(c''; \textbf{while } b \textbf{ do } c' \textbf{ end})$ soit à **False** et on a $\forall j \in \mathcal{P} \ \vdash_l c \xrightarrow[p]{j} \mathsf{Ok}$.

  Dans les deux cas le lemme est vérifié.

Quelque soit le programme c, le lemme est donc vérifié.

□

Théorème 2. *Un programme Pr qui communique correctement et vérifie la propriété de synchronisations répliquées est sans erreur de synchronisation. c.-à-d. si l'évaluation de Pr termine, elle termine dans l'état Ok.*

Démonstration. Un programme BSP-IMP évalué sur p processeur peut aboutir à un état d'erreur $\mathsf{Err} \in \{\mathsf{Err}_G; \mathsf{Err}_{\text{SYNC}}\}$ par les règles (3.17),(3.18) ou (3.20).

Soit $[Pr, \ldots, Pr]^p$ un vecteur programme ayant la propriété de synchronisations répliquées, communicant correctement tel que $d \vdash_g [Pr, \ldots, Pr]^p \underset{p}{\rightarrow} S$. Supposons que pour tout arbre de dérivation $d' \prec d$, $d' \vdash_g [c', \ldots, c']^p \underset{p}{\rightarrow} S' \implies S' = \mathsf{Ok}$, nous allons montrer $S = \mathsf{Ok}$.

Si $[Pr, \ldots, Pr]^p$ est évalué en Err_G par la règle (3.20), alors il existe c tel que $\vdash_l c \xrightarrow[p]{i} \mathsf{Err}_L$, or au niveau local un processus est évalué en Err_L si et seulement si une erreur de communication survient, ce qui est en contradiction avec l'hypothèse initiale. L'arbre de dérivation de $[Pr, \ldots, Pr]^p$ n'utilise donc pas la règle 3.20).

D'après le lemme 2,

$$\forall i, j \in [0, p-1] \vdash_l Pr \xrightarrow[p]{i} f_i \;\&\; \vdash_l Pr \xrightarrow[p]{j} f_j \implies f_i = f_j,$$

on ne peut donc pas avoir $\vdash_l c \xrightarrow[p]{j} \mathsf{Wait}(c'_j)$ et $\vdash_l c \xrightarrow[p]{i}\downarrow$. La règle (3.18) n'est donc pas applicable. Soit d un arbre de dérivation terminant par la règle (3.17) tel que

$$d = \dfrac{\forall i \in \mathcal{P}\ \dfrac{\vdots}{c \xrightarrow[p]{i} \mathsf{Wait}(c_i)} \qquad d' = \dfrac{\vdots}{[c_0, \ldots, c_{p-1}]^p \underset{p}{\rightarrow} S}}{[c, \ldots, c]^p \underset{p}{\rightarrow} S}.$$

D'après le lemme 2, pour tout i et j appartenant à $[0, p-1]$, $\mathsf{Wait}(c_i) = \mathsf{Wait}(c_j)$, donc $\forall i \in [0, p-1]\ c_i = c_0$ et par induction, comme $d' \prec d$, $S = \mathsf{Ok}$.

Ainsi quelque soit la règle terminant sa dérivation, si un programme a les propriétés requises par le théorème 2, il finit dans l'état Ok(si il termine). □

Si toutes les occurrences des variables d'une expression booléenne sont répliquées, c.-à-d. qu'elles ont la même valeur sur tous les processeurs, l'expression sera évaluée à la même valeur sur tous les processeurs.

Un sous ensemble $\mathcal{R}ep(Pr)$ des occurrences de variables répliquées du programme Pr peut être construit à partir des variables non affectées par une communication auxquelles ne sont affectés que les résultats d'expressions arithmétiques n'impliquant que des occurrences de variables répliquées et des constantes. Ces affectations ne doivent pas avoir lieu dans une instruction **if** ou **while** dont la condition ne s'évalue pas elle même identiquement sur tous les processeurs. De plus une occurrence de variable n'est répliquée que si elle est précédée d'une affectation de cette même variable dans les conditions décrites ci-dessus. En effet, de manière générale les environnements locaux initiaux ne sont pas identiques, les occurrences de variables non initialisées ne sont donc pas des occurrences répliquées.

Nous avons ici deux définitions mutuellement dépendantes des occurrences de variables répliquées et des expressions booléennes répliquées, Mais $\mathcal{R}ep(Pr)$ peut être construit comme le plus grand point fixe des occurrences des variables répondant aux exigences citées précédemment.

3.3.3 Exemple de programme sans erreur de synchronisation

Le programme 2 calculant la somme des préfixes d'un vecteur réparti sur les processeurs est sans erreur de synchronisation. Il est aisé de vérifier que toutes les communications sont correctes : Le premier **get** n'est exécuté qu'aux processeurs dont **Pid** est strictement supérieur à zéro (c.-à-d. **Pid** $\in [1, \mathbf{Nprocs} - 1]$) et le "destinataire" de la communication et **Pid**-1 qui est donc dans l'intervalle [0,**Nprocs**-2]. la seconde instruction **get** du programme est exécutée aux processeur dont l'identifiant est supérieur à 2^i et est faite à destination du processeur $\mathbf{Pid} - 2^i$ or la condition de la boucle dans laquelle se situe la requête de communication impose que 2^i soit strictement inférieur à **Nprocs** et i est inférieur ou égal à 1. Donc $2^i \leq$ **Pid** implique $\mathbf{Pid} - 2^i \geq 0$ et et **Pid** étant forcément inférieur ou égal à **Nprocs**-1 on a $\mathbf{Pid} - 2^i \in [0, \mathbf{Nprocs}]$.

De plus aucun élément **sync** du programme ne se trouve dans une boucle ou une conditionnelle ne s'évaluant pas partout à la même valeur. Le premier **sync** n'est dans aucune boucle ni aucune conditionnelle L'ensemble $\mathcal{R}ep$(Scan) contient toutes les occurrences de i succédant son initialisation, donc l'expression $2^i \leq$ **Nprocs** est répliquée et le deuxième **sync** du programme est donc sûr.

Programme 2.

Scan $\equiv$
if (Pid > 0 **) then get**$(\mathbf{Pid} - 1, X, X_{in})$; **end**
sync ;
i :=1 ;
while ($2^i \leq$ **Nprocs) do**
if (Pid $\geq 2^{i-1}$**) then** $X := X_{in} + X$ **end**;
if (Pid $\geq 2^i$**) then get**$(\mathbf{Pid} - 2^i, X, X_{in})$ **end**;
sync;
$i = i + 1$
end

FIG. 3.1 – Somme des préfixes

Chapitre 4

Modélisation de la partie BSMP de la BSPlib

Nous étendons ici notre modélisation de la BSPlib en introduisant les communications de type BSMP. Cette extension se fait d'une part en ajoutant les éléments syntaxiques nécessaires, d'autre part en modifiant la sémantique du langage afin de tenir compte de ce mode de communication.

4.1 Syntaxe

les éléments syntaxiques décrits dans le chapitre précédant se voient ajouter les éléments suivant :

- les expressions arithmétiques **Aexp** sont augmentées de la variables spéciale **Qsize** modélisant la fonction `bsp_qsize`.
- aux commandes décrite par **Com** sont ajoutés **send**(a, a) qui modélise `bsp_send` et **move**(X) qui modélise `bsp_move`.

4.2 Sémantique

Afin de prendre en compte le nouveau mode de communication, les environnements d'exécution locaux sont étendus par un multiensemble de messages à envoyer sm ainsi qu'un multiensemble de messages reçus rm :

$$\langle c,\ \sigma,\ r,\ sm,\ rm\rangle.$$

L'évaluation de la variable spéciale **Qsize** donne le nombre de message restant dans le multiensemble des messages reçus rm.

$$\frac{}{\langle \textbf{Qsize},\ \sigma,\ r,\ sm,\ rm\rangle \xrightarrow[p]{i} |rm|}$$

Cette règle est ajoutée à la sémantique des expressions arithmétiques.

Les quatre règles suivantes sont ajoutées à la sémantique locale pour former le système de règles de BSP-IMP étendu aux communications BSMP.

Au niveau local la commande **send**(i, j) ajoute un message (l'entier j à envoyer au processeur i) au multiensemble sm si $i \in [0, p-1]$, sinon une erreur est levée.

$$\frac{\langle a_1,\ \sigma,\ r,\ sm,\ rm\rangle \xrightarrow[p]{i} j \in [0,p-1] \quad \langle a_2,\ \sigma,\ r,\ sm,\ rm\rangle \xrightarrow[p]{i} n}{\langle \textbf{send}(a_1,a_2),\ \sigma,\ r,\ sm,\ rm\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma,\ r,\ \{(i,n)\} \cup sm,\ rm\rangle} \tag{4.1}$$

$$\frac{\langle a_1,\ \sigma,\ r,\ sm,\ rm\rangle \xrightarrow[p]{i} j \notin [0,p-1] \quad \langle a_2,\ \sigma,\ r,\ sm,\ rm\rangle \xrightarrow[p]{i} n}{\langle \textbf{send}(a_1,a_2),\ \sigma,\ r,\ sm,\ rm\rangle \xrightarrow[p]{i} \langle \mathsf{Err},\ \sigma,\ r,\ sm,\ rm\rangle} \tag{4.2}$$

La commande **move**(X) enlève un message du multiensemble des messages reçus rm et affecte la valeur qu'il contient à la variable X. Si il n'y a pas de message dans rm une erreur est levée.

$$\frac{|rm| > 0 \qquad (n) \in rm}{\langle \textbf{move}(X),\ \sigma,\ r,\ sm,\ rm\rangle \xrightarrow[p]{i} \langle \mathsf{Ok},\ \sigma[X \mapsto n],\ r,\ sm,\ rm \setminus (n)\rangle} \tag{4.3}$$

$$\frac{|rm| = 0}{\langle \textbf{move}(X),\ \sigma,\ r,\ sm,\ rm\rangle \xrightarrow[p]{i} \langle \mathsf{Err},\ \sigma,\ r,\ sm,\ rm\rangle} \tag{4.4}$$

Pour la sémantique globale, la règle de synchronisation (3.17) doit être modifiée afin de prendre en compte les transmissions de message.

$$\frac{\begin{array}{c}\forall i \in \mathcal{P}, \langle c_i,\ \sigma[i],\ R[i],\ Sm[i],\ Rm[i]\rangle \xrightarrow[p]{i} \langle \mathsf{Wait}(c'_i),\ \sigma'[i],\ R'[i],\ Sm'[i],\ Rm'[i]\rangle \\ \oplus(\Sigma', R, \Sigma'') \qquad Rm''[i] = \bigcup_{j\in[0,p-1]}\{n|(i,n) \in Sm[j]\} \\ \langle [c'_0,\ldots,c'_{p-1}]^p,\ \Sigma'',\ \emptyset,\ \emptyset,\ Rm''\rangle p \underset{p}{\rightarrow} \langle \downarrow,\ \Sigma''',\ R''',\ Sm''',\ Rm'''\rangle p\end{array}}{\langle [c_0,\ldots,c_{p-1}]^p,\ \Sigma,\ R,\ Sm,\ Rm\rangle p \underset{p}{\rightarrow} \langle \downarrow,\ \Sigma''',\ R''',\ Sm''',\ Rm'''\rangle p} \tag{4.5}$$

4.3 Propriétés

Les programmes utilisant le mode de communication par passage de message sont clairement indéterministes, en effet la procédure **move** retire un message quelconque du multiensemble et donc on ne peut prédire un unique résultat à partir du moment où il y a plus d'un message en attente de lecture.

Contrairement au cas des communications DRMA, où l'indéterminisme est anecdotique, indésirable et facilement évitable, le comportement indéterministe des communications BSMP est étroitement lié au modèle de communication, il ne peut donc pas être modifié simplement pour le rendre déterministe. Toutefois, les programmes utilisant ce mode de communication sont faits pour communiquer un nombre de messages fini, et pour atteindre un état unique une fois tous les messages traités, quelque soit l'ordre de traitement. Cette façon de traiter les messages peut être considérée comme une procédure déterministe. Ainsi, si un programme n'utilise les messages reçus que dans de telles procédures, il sera déterministe.

Pour ce qui est des erreurs de synchronisations l'extension de BSP-IMP aux communications par passage de messages n'apporte que la contrainte de vérifier que les opération d'envoi de message ne produisent pas d'erreur en s'adressant à un indice de processeur inexistant.

Chapitre 5

Conclusions

Nous présentons dans ce rapport une sémantique formelle pour la bibliothèque BSPlib. Dans un premier temps, la modélisation restreinte aux modes de communication DRMA nous a permis de discuter du déterminisme des programmes exprimables dans ce modèle, nous amenant à proposer deux modifications possibles de la sémantique pour obtenir un comportement prédictible.

À partir de cette sémantique, nous avons exhibé une classe de programme sans erreur de synchronisation, ceux sans erreur de communication ayant la propriété de *synchronisations répliquées* que nous avons définie. Bien que la détection des erreurs de communication soit un problème indécidable de manière générale, il peut être résolu par une démonstration ad-hoc comme dans l'exemple que nous présentons, ou à l'aide d'une sémantique axiomatique telle que celle présentée dans [7].

La propriété de *synchronisations répliquées*, reposant sur le fait que des expressions booléennes s'évaluent identiquement sur tout les processeurs, est elle aussi indécidable de manière générale. Cependant nous proposons une caractérisation pour une classe d'expressions booléennes ayant cette propriété, permettant de les détecter automatiquement en utilisant uniquement l'arbre syntaxique du programme. Cette caractérisation des expressions booléennes répliquées constitue un premier pas vers un outil de vérification automatique.

La sémantique a ensuite été étendue aux communications par passage de messages, BSMP. Cependant afin d'être applicable à un plus grand nombre de programmes la sémantique définie ici gagnerait à être enrichie par l'introduction de la notion de tableau et de différents types de données. De plus les opérations arithmétiques susceptibles de générer des erreurs telles que la division n'ont pas été prises en compte dans le modèle que nous proposons. L'ajout de telles opérations nécessiterait de compléter les règles de la sémantique pour prendre en comptes les cas d'erreur dans l'évaluation des expressions arithmétiques.

Bibliographie

[1] Johnathan M.D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling. Bsplib - the bsp programming library. Technical Report TR-29-97, Oxford University Computing Laboratory, Programming Research Group, May 1997.

[2] H. Jifeng, Q. Miller, and L. Chen. Algebraic laws for BSP programming. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, number 1123–1124 in Lecture Notes in Computer Science, Lyon, August 1996. LIP-ENSL, Springer.

[3] D. S. Lecomber. A semantics for parallel programming with bsp.

[4] David Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and answers about bsp. *Scientific Programming*, 6(3) :249–274, Fall 1997.

[5] M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.

[6] Stewart and Clint. Synchronising asynchronous communications. In *EUROPAR : Parallel Processing, 3rd International EURO-PAR Conference*. LNCS, 1997.

[7] A. Stewart, M. Clint, and J. Gabarró. Axiomatic Frameworks for Developing BSP-Style Programs. *Parallel Algorithms and Applications*, 14 :271–292, 2000.

[8] G. Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing Series. MIT Press, 1993.

Annexe A

Sémantique des expressions booléennes

Constantes True, False

$$\frac{}{\langle \textsf{True},\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{True}} \qquad \frac{}{\langle \textsf{False},\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{False}}$$

Test d'égalité

$$\frac{\langle a_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} n_1 \quad \langle a_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} n_2 \quad n_1 = n_2}{\langle a_1 = a_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{True}} \qquad \frac{\langle a_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} n_1 \quad \langle a_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} n_2 \quad n_1 \neq n_2}{\langle a_1 = a_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{False}}$$

Teste d'inégalité

$$\frac{\langle a_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} n_1 \quad \langle a_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} n_2 \quad n_1 \leq n_2}{\langle a_1 \leq a_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{True}} \qquad \frac{\langle a_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} n_1 \quad \langle a_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} n_2 \quad n_1 \not\leq n_2}{\langle a_1 \leq a_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{False}}$$

Négation

$$\frac{\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{True}}{\langle \neg b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{False}} \qquad \frac{\langle b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{False}}{\langle \neg b,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{True}}$$

Conjonction

$$\frac{\langle b_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{False}}{\langle b_1 \wedge b_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{False}} \qquad \frac{\langle b_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{True} \quad \langle b_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{False}}{\langle b_1 \wedge b_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{False}}$$

$$\frac{\langle b_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{True} \quad \langle b_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{True}}{\langle b_1 \wedge b_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{True}}$$

Disjonction

$$\frac{\langle b_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{True}}{\langle b_1 \vee b_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{True}} \qquad \frac{\langle b_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{False} \quad \langle b_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{True}}{\langle b_1 \vee b_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{True}}$$

$$\frac{\langle b_1,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{False} \quad \langle b_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{False}}{\langle b_1 \vee b_2,\ \sigma,\ r\rangle \xrightarrow[p]{i} \textsf{False}}$$