Développement systématique et sûreté d'exécution en programmation parallèle structurée

Louis Gesbert

Université Paris XII – Val de Marne **PRES Paris-est**

École doctorale SIMME **UFR de sciences**

Développement systématique et sûreté d'exécution en programmation parallèle structurée

THÈSE

présentée et soutenue publiquement le jeudi 5 mars 2009

Pour l'obtention du titre de

Docteur de l'université Paris Est

par

Louis Gesbert

Composition du jury

Rapporteurs :	Emmanuel Chailloux	
	Jocelyn Sérot	
Examinateurs :	Zhenjiang Hu	
	Olivier Michel	
	Frédéric Gava	
	Frédéric Loulergue	(*directeur*)

Laboratoire d'Algorithmique, Complexité et Logique

Remerciements

Merci à Emmanuel Chailloux et à Jocelyn Sérot d'avoir accepté d'être mes rapporteurs ; merci également à Zhenjiang Hu et Olivier Michel de faire partie de mon jury.

Je tiens à remercier mon directeur de thèse Frédéric Loulergue qui a cru en moi et, malgré la distance, a su être là quand il fallait. Un grand merci à Frédéric Gava qui l'a secondé, pour nos longs débats techniques qui finalement se sont révélés productifs, et pour le coup de pouce décisif sur la fin.

Pour le temps agréable qu'on a passé ensemble au LACL, je remercie Marie, Danièle, Joëlle, Alexis, Catalin et tous les autres. Sans oublier, bien sûr, l'irremplaçable Flore.

Je remercie du fond du cœur ma famille pour avoir toujours été derrière moi quand les temps étaient un peu moins faciles, et pour m'avoir maintenu en vie pendant ces derniers mois. Merci à Nils pour avoir accepté de me relire et de commenter.

Pour les bons moments, qui ont aussi contribué à faire de cette thèse l'expérience qu'elle a été, je remercie mes amis fidèles ; en particulier Damien qui m'a supporté un bon moment.

Je tiens à remercier les membres de l'IPL à Tokyo avec qui j'ai eu la chance de travailler pour m'avoir permis de découvrir leur pays, pour leur accueil et leur aide.

Enfin, Nathalie mérite ma plus profonde reconnaissance pour m'avoir soutenu à travers des moments particulièrement difficiles, et je ne l'oublierai pas.

Table des matières

Chapitre 1

Introduction

1.1 Généralités

La programmation multi-processeur est un sujet ancien, mais qui a dernièrement suscité un intérêt considérable. En effet, traditionnellement réservé au calcul scientifique, le parallélisme s'empare désormais des ordinateurs personnels et soulève à cette occasion des problématiques différentes, telles que la facilité d'utilisation ou la sécurité. Cet élargissement des perspectives du domaine d'une part, et la perspective d'un changement radical dans l'architecture des ordinateurs qui entraînera inévitablement une nouvelle façon d'envisager la programmation, d'autre part, font que la recherche dans ce domaine est très active. Aucun paradigme de programmation général n'a encore été massivement accepté dans ce contexte.

Ce travail participe à la recherche d'un tel paradigme, associant la programmation de haut niveau au parallélisme afin d'obtenir une structure claire qui rende le parallélisme accessible tout en évitant qu'il soit source de bogues difficiles à identifier. Il consiste, pour la première partie, en une évolution du langage BSML, dont les premières idées ont été établies dans [Lou98]. De nombreux travaux se sont attachés à établir un formalisme fort pour ce langage, avec sémantiques, modèle de coûts, machine abstraite, *etc.* Soutenu par ces travaux, le présent document étend BSML de plusieurs façons, afin d'en faire un langage général :

- Inclusion de nouvelles fonctionnalités, en particulier impératives, telles que les exceptions et les références, qui sont généralement considérées comme souhaitables dans un langage moderne.
- Amélioration de la sûreté du langage, avec la définition d'un système de types complet permettant d'étendre la sûreté d'exécution fournie par le typage statique de langages fonctionnels tels que ML au parallélisme.
- Définition d'extensions syntaxiques qui rendent les programmes parallèles plus lisibles, et plus simples à écrire.

Ces évolutions font de BSML un langage complet et utilisable, qui peut s'établir parmi les paradigmes émergents pour le parallélisme. Un autre domaine, qui est une conséquence de la généralisation du parallélisme mais reste relativement nouveau, est la certification de programmes parallèles. La deuxième partie de ce travail développe, par l'exemple, une méthodologie de développement certifié basée sur BSML, l'assistant de preuve Coq et les squelettes algorithmiques.

1.1.1 Parallélisme et calcul scientifique

Le parallélisme est un moyen naturel pour multiplier la puissance de calcul par rapport à la vitesse nominale des processeurs. Les processeurs traditionnels, séquentiels, évoluant à une vitesse considérable, il a donc pendant longtemps été utilisé pour anticiper sur leur évolution dans les applications requérant une grande quantité de calculs, comme les simulations et le calcul scientifique. Avec l'avènement des micro-ordinateurs sont également apparues les grappes («beowulf clusters», [SSB+95]), qui, mettant en commun la puissance d'un grand nombre d'ordinateurs bon marché, ont un coût bien moindre que les super-calculateurs.

Ces machines, dont la puissance ne fait qu'anticiper de quelques années sur celle des ordinateurs grand public, demandent une programmation délicate qui reste en accord avec leur utilisation. Ainsi, les outils de programmation parallèle sont longtemps restés peu accessibles, et sont encore aujourd'hui souvent basés sur les langages Fortran ou C. Deux approches du parallélisme de bas niveau se différencient par leur notion de mémoire : celle-ci peut être un espace unique auxquels accèdent tous les processeurs (mémoire partagée), ou bien des espaces privés réservés à chacun des processeurs (mémoire distribuée). Les outils les plus connus suivant chacune de ces approches sont OpenMP [CJvdP07], pour la mémoire partagée, et PVM [GBD+94] ou MPI [SG98] pour la mémoire distribuée.

PVM comme MPI laissent un contrôle total de la gestion du parallélisme au programmeur : il s'agit, en quelque sorte, de contrôler individuellement chacun des processeurs ainsi que leurs échanges de messages et leurs synchronisations, approche que nous appellerons concurrence. Cela permet un ajustement optimal des performances, pour qui est prêt à y consacrer le travail nécessaire ; cependant, la tâche est extrêmement ardue, les tests délicats puisque les instants d'arrivée des messages sont imprévisibles, et le déboguage difficile en raison de la complexité des interactions qui peuvent aboutir à un problème [Gor04]. C'est pourquoi, en pratique, même en MPI le parallélisme suit des règles implicites qui permettent de comprendre son déroulement.

OpenMP permet un parallélisme semi-automatique par annotations du code source, ce qui simplifie la transition de programmes séquentiels vers des programmes parallèles. Cependant, celui-ci ne se révèle guère efficace en dehors du traitement de tableaux et de boucles itératives, la parallélisation automatique étant un domaine qui a donné peu de résultats probants. Les algorithmes parallèles efficaces ont souvent une structure très différente de celle des algorithmes séquentiels équivalents : OpenMP permet leur implantation en offrant également un contrôle plus fin du parallélisme, mais perd du coup les notions de structure et de sûreté qu'il aurait pu offrir par rapport à MPI.

Notons que, contrairement à ce qu'on pourrait croire, mémoire partagée ou distribuée correspondent ici plus à des modèles qu'à des architectures matérielles concrètes : MPI fonctionne aisément (et est optimisé) sur une machine à mémoire partagée, bien que les échanges se fassent par messages explicites. De même, il est possible de simuler une mémoire partagée à partir de mémoires locales, en effectuant des communications quand nécessaire. Dans ce dernier cas cependant, l'efficacité du programme peut s'en ressentir. Notons que la gestion d'une mémoire partagée, même sans simulation, passe moins bien à l'échelle à cause des limitations de bande passante et des risques d'accès concurrents.

Il est ainsi généralement accepté que l'on obtient de meilleurs résultats – moyennant plus d'efforts – en supposant la mémoire distribuée qu'en la supposant partagée.

1.1.2 Évolution du matériel

Les fabricants de micro-processeurs ont atteint dernièrement un palier dans les fréquences d'horloge, qui avaient jusqu'à présent été le moteur de l'évolution des performances. La miniaturisation, qui permet de réduire le courant nécessaire, semble en effet atteindre ses limites, et la puissance dissipée par le processeur étant fonction de sa fréquence de fonctionnement, les approches actuelles de la fabrication ne peuvent plus pousser dans cette direction sans envisager une consommation électrique et des systèmes de refroidissement rédhibitoires.

Les améliorations techniques dans l'architecture des processeurs permettent encore des gains de performances, mais qui ne sont plus de la même échelle. Néanmoins, les fabricants promettent de poursuivre une progression exponentielle[1] dans la puissance de calcul : la vitesse du traitement séquentiel d'opérations touchant à une limite, cela est fait par multiplication du nombre de cœurs de calcul par processeur. Le parallélisme gagne ainsi les ordinateurs personnels.

Ce parallélisme soulève des questions, mais ne met pas encore en évidence les difficultés que soulèverait un parallélisme de masse : les utilisateurs bénéficient encore raisonnablement de la puissance de leurs deux, ou quatre cœurs, l'ordinateur ayant le plus souvent un petit nombre de tâches à accomplir simultanément. Les jeux vidéo, par exemple, qui sont les applications d'usage courant les plus exigeantes en puissance de calcul, gèrent simultanément l'affichage, le son, la simulation du monde, l'«intelligence artificielle» des personnages, *etc.* Si l'évolution continue en ce sens, cependant – et on nous promet déjà des processeurs à 64 cœurs – ce bénéfice ne durera plus, et les programmes actuels se révèleraient incapables d'utiliser la puissance disponible.

1.1.3 Parallélisme moderne

C'est une des raisons qui nous poussent à développer de nouveaux modèles de programmation, afin, à terme, de permettre la généralisation de la programmation parallèle. Il est ainsi naturel que les différentes approches de la programmation déjà connue cherchent leur place dans ce nouveau contexte. La programmation fonctionnelle, par exemple, offre un certain niveau de structure et évite la majorité des problèmes à l'exécution : ces problèmes étant multipliés par le nombre de processeurs, quelles structures supplémentaires permettent de conserver ces garanties ?

Parmi les approches de haut niveau se détachant de la concurrence brute et du parallélisme automatisé, on trouve le parallélisme de données et les squelettes algorithmiques. Le parallélisme de données [HS86] relâche en partie le contrôle du programmeur sur les processus, et est basé sur l'idée d'un traitement unifié sur de grandes quantités de données, bien qu'il ne se limite pas à cela (en première approximation, on peut comparer son domaine d'utilisation à celui d'OpenMP). Il offre l'avantage d'être synchrone, et aisé à concevoir ; il évite les interblocages. Typiquement, on peut spécifier un ensemble de données et effectuer des opérations sur cet ensemble et sur ses éléments, l'ensemble sera divisé entre les processeurs et les opérations parallélisées (ce qui est rendu possible par une syntaxe qui impose certaines propriétés lors de leur définition). BSP, dont nous reparlerons, est une variante de ce modèle qui laisse le contrôle sur les processeurs.

Les squelettes [Col89, Col04b, Col04a] sont des fonctions de haut niveau, parallèles et hautement optimisées. Le programmeur écrit son programme de façon déclarative, en implantant les opérations coûteuses en calcul par l'intermédiaire des squelettes. Les squelettes sont souvent

[1] comme la décrit la célèbre «loi de Moore»

associés à des traitements mathématiques qui, moyennant certaines propriétés sur le programme initial, en extraient des informations permettant la parallélisation. Ainsi, par exemple, pour la composition de squelettes : deux opérations de communication successives peuvent être groupées si elles n'ont pas d'interdépendance. En utilisant les squelettes, l'utilisateur n'a pas à s'inquiéter des problèmes de bas niveau relatifs au parallélisme, mais il doit néanmoins avoir une bonne connaissance des schémas de fonctionnement de son application afin de les utiliser à bon escient. Les squelettes se combinent avantageusement à notre approche, où il est possible de les définir en tant que simples fonctions : nous nous étendrons sur le sujet en 6.2.

Un certain nombre de tentatives existent pour donner à des langages fonctionnels accès au parallélisme. Les premières permettent un traitement de la concurrence compatible avec les systèmes de typage, comme Concurrent ML [Rep99] ou JoCaml [MM08] pour ML, ou Eden [LOMP05] pour Haskell. Ces systèmes préservent les atouts de la programmation fonctionnelle mais ne les étendent pas au parallélisme, et restent ainsi sujets aux interblocages et à l'indéterminisme.

D'autres langages ajoutent un niveau de structure étendant effectivement la sûreté associée au typage fort au parallélisme, et évitent ces problèmes : citons NESL [BHS+94], langage basé sur ML, Glasgow Parallel Haskell (Gph) [Tri99] et Oz [Smo95]. Ces langages ont pour point commun d'autoriser le parallélisme multi-niveaux et de fonctionner par génération dynamique de processus légers (*threads*) : il en résulte une expression du parallélisme plus simple, particulièrement adaptée au parallélisme de données. À l'opposé des approches précédentes, dans ces langages, le contrôle des ressources de calcul est implicite, et dissimulé à l'utilisateur. Celui-ci se charge de séparer sa tâche en un nombre quelconque de threads, qui sont automatiquement distribués. Cette approche centre le parallélisme sur l'algorithme et non sur la machine, ce qui peut s'avérer une limitation lorsqu'on désire implanter des algorithmes complexes de façon optimale, ou tirer le plein parti du matériel disponible. En particulier, le sous-découpage du problème doit être significativement plus fin qu'il ne serait nécessaire pour obtenir un bon équilibrage. Il est, d'autre part, plus difficile de se représenter l'exécution réelle du programme et d'analyser ses coûts.

BSML, le langage auquel est consacré le présent travail, est basé sur un modèle simple qui lui permet d'assurer la sûreté d'exécution tout en laissant le strict contrôle des processeurs au programmeur, ce qui lui permet de fournir un modèle de coûts simple et fiable. La possibilité de définir des fonctions de plus haut niveau, de plus, permet d'explorer d'autres mécanismes de parallélisme à l'intérieur du langage.

1.2 Certification et parallélisme

La généralisation du parallélisme s'accompagne de nouveaux besoins issus de la programmation traditionnelle. Les méthodes formelles en sont un exemple, et elles n'en sont qu'à leurs premiers pas dans ce domaine. Elles permettent la validation de programmes à partir de spécifications mathématiques précises : cette validation est d'autant plus utile (et difficile) que le programme est complexe et éloigné de la notion intuitive qu'on pourrait en avoir – donc particulièrement utile pour les programmes parallèles.

Certains travaux établissent des preuves sur des mécanismes bas niveau pour le parallélisme [GK07, TW04], mais il s'agit plus de prouver les protocoles de communication et l'absence d'interblocages à l'aide de *model-checkers* que de prouver des propriétés générales sur des programmes. Prouver des programmes réalistes n'est envisageable que dans le contexte d'un parallélisme forte-

ment structuré. C'est le cas de BSP, pour lequel existent des travaux de dérivation de programmes impératifs à partir de sémantique à la Hoare par exemple [CS03, JMC96, SC01], ou à partir de transitions d'état globales [SCG00], ou encore de sémantique par raffinements [Ski98].

Ces approches ne proposent pas d'outils spécifiques pour les preuves, et ne se basent pas sur des assistants de preuves : on reste au niveau d'une formalisation purement manuelle. Des travaux récents basés sur l'assistant de preuves Coq comblent cette lacune et permettent la certification de programmes BSPlib [TL07, GF08, GF09]. Enfin, dans un contexte de programmation fonctionnelle, on établit des preuves en Coq de programmes BSML dans [Gav03, Gav05] : c'est une approche similaire que nous suivrons ici.

Les squelettes, par ailleurs, constituent des briques de construction pour le parallélisme et fournissent en tant que tels une bonne base pour la certification. [GF08] fait un pas en ce sens en proposant une implantation certifiée du problème des N-corps, considéré comme représentant une des grandes classes de programmes parallèles (les *dwarfs*, [ABC+06]). Par ailleurs, des méthodes formelles de dérivation de programmes utilisant les squelettes, basées sur l'algorithmique constructive, existent [HIT97].

Nous étudierons donc une méthode de dérivation de programmes parallèles certifiés se basant sur une combinaison de ces techniques.

1.3 Exceptions et parallélisme

Les langages de programmation modernes offrent tous des mécanismes de gestion des évènements anormaux ou exceptionnels : face à la complication des systèmes informatiques, de tels évènements sont voués à se produire, et bien qu'il soit souhaitable d'éviter qu'ils ne provoquent un arrêt brutal du programme, il n'est pas envisageable de les traiter tous, cas par cas. Un langage réaliste se voulant sûr se doit donc d'intégrer un tel système : même dans le cadre d'un système certifié, les opérations d'entrées-sorties, d'allocation mémoire ou en règle générale liées au matériel sont sujettes à des cas d'erreur.

Les travaux sur la gestion des exceptions remontent aux années 1970 [Goo75]. Aujourd'hui, ayant dépassé le rôle d'un simple mécanisme de gestion d'erreur dans beaucoup de langages, elle en est un élément structurel important – et, par conséquent, est fortement liée à la structure du langage et au modèle d'exécution. On rassemble, sous le terme «gestion d'exceptions», trois rôles distincts :

- La capacité à continuer l'exécution du programme en l'absence d'un résultat, celui-ci ayant échoué à se calculer – éventuellement, simplement pour afficher la nature de l'erreur avant de terminer. Ce rôle pourrait être rempli par une valeur spéciale de retour, pour chaque opération, et une disjonction de cas suivant cette valeur. C'est souvent ce qui est fait avec l'utilisation de pointeurs `null` en C ou des types `option` en OCaml ([LDG+07]) : ces cas ne correspondent pas à de la gestion d'exceptions, car ils obligent à une gestion explicite, à tous les niveaux d'appels, des cas exceptionnels. Un système de gestion d'exceptions introduit un traitement implicite des cas exceptionnels par défaut, permettant justement de ne pas alourdir le code.
- Une exception permet donc, quand elle est déclenchée, d'interrompre le flux normal d'exécution du programme. Concrètement, l'exécution est déroutée sur un code de traitement de l'exception : ainsi, une exception doit déclencher un traitement spécifique sans devoir

remonter étape par étape dans la pile d'appels comme le ferait un retour normal de fonction.
- Un mécanisme de rattrapage ou de reprise, suivant les cas, permet de relancer l'exécution du programme dans un état sain à partir d'un point donné, ou de réessayer l'opération ayant échoué avec des paramètres différents.

Ouvrons une parenthèse pour noter que les systèmes de gestion d'exceptions dits monadiques, qui s'intègrent élégamment à la programmation fonctionnelle, ne respectent pas directement nos deux premiers critères (qui sont certes restrictifs) : ils ne permettent pas la propagation des exceptions de façon entièrement transparente. L'utilisation des monades pour la gestion d'exceptions se justifiant dans les langages non stricts, mais étant plus lourde pour l'utilisateur, nous ne détaillerons pas le sujet. Le parallélisme dans les langages non stricts est en effet un sujet assez éloigné du nôtre, et faisant l'objet d'autres études, comme par exemple [Mil02] pour le langage Haskell (dans ce langage, l'approche utilisée pour Eden est d'abandonner les traits paresseux lorsque des opérations sont parallélisées, et celle de Gph utilise des threads dynamiques). Pour des raisons de performances et de contrôle explicite sur le parallélisme, on ne s'étendra pas sur le sujet, le lecteur pouvant se reporter à [Hai94] pour une discussion plus étendue sur le sujet.

Les mécanismes de gestion d'exceptions varient en expressivité et en simplicité d'usage d'un langage à un autre. Common Lisp propose un système de *conditions* par exemple, extrêmement souple mais délicat d'usage et d'implantation. Celui-ci permet en effet de relancer l'exécution à l'endroit où elle aurait échoué, tout en en modifiant les paramètres. Cependant, aucun langage à typage statique ne propose un tel système à l'heure actuel, nous n'envisagerons donc pas de tel système pour un langage parallèle.

La gestion des exceptions en ML est impérative : elle est de fait similaire à celle de la plupart des langages impératifs modernes, et relativement simple, avec l'usage d'une commande pour lever une exception, et la construction de blocs permettant de rattraper les exceptions que le code qu'ils contiennent auraient levées. Un tel bloc est donc protégé contre les évènements inhabituels (ou contre certains d'entre eux, à la volonté du programmeur), et est muni de code à exécuter pour traiter ces évènements et renvoyer un résultat, sans que cela influe sur le programme à l'extérieur de ce bloc. De tels blocs peuvent être imbriqués, et s'intègrent très bien dans le système de types de ML. Ce mécanisme a de plus le mérite de s'implanter efficacement sur les processeurs actuels. Il est utilisé entre autres par Ada, C++, Java, C#, Delphi, PHP, Python et Ruby.

Pour se représenter la complexité de la gestion des exceptions dans le cas d'une exécution parallèle, il est utile de se représenter la propagation d'exceptions levées comme un saut dans la pile d'appels du programme, ou plusieurs sauts successifs jusqu'à trouver un traitement correspondant à l'exception. Lors de l'exécution parallèle, il existe une pile d'appels par processeur, et celles-ci peuvent être entièrement indépendantes si on ne suppose pas de structure sous-jacente. La notion même d'état de l'exécution à un instant donné n'est pas aisée à définir, si une barrière de synchronisation n'est pas imposée. Il existe ainsi une grande quantité de travaux [RDKT01] sur les exceptions dans un contexte concurrent, ou plus généralement sur les calculs concurrents tolérants aux pannes, adaptés aux différents modèles et langages [BLKC05]. Ainsi, comme le soulignent les auteurs dans [RK01], le traitement des exceptions dans un environnement d'exécution concurrent ou parallèle doit être profondément lié au modèle employé : il n'existe pas de bonne solution dans l'absolu. La plupart des systèmes existants, surtout ceux qui présentent des traits de concurrence, ne traitent les exceptions que de façon locale ou séquentielle, ou bien ne garantissent pas la cohérence du système après reprise d'une exception.

Dans les modèles concurrents, la gestion des exceptions ne pouvant se faire globalement, celle-ci est souvent transformée en un protocole de communication auxiliaire qui permet de propager

les informations sur l'état d'un processus aux autres, et de prendre ainsi des décisions cohérentes à différents niveaux de localité, suivant la portée de l'exception ou de l'erreur. Certains systèmes, comme celui d'Argus [Bal92] permettent de sélectionner un sous-ensemble des processeurs d'après certains critères (tout ceux ayant défini une exception particulière, par exemple), et effectuent une synchronisation sur ce sous-ensemble en cas d'exception. Cela se révèle complexe, du point de vue de la gestion aussi bien que du point de vue du programmeur, et impose de plus un surcoût en communications.

Un contexte différent mais néanmoins lié à la programmation concurrente, et ayant donné lieu à un certains nombre de travaux sur la gestion des exceptions, est celui des *Enterprise Java Beans* ; le système choisi qui a été retenu pour cette application a l'avantage de ne pas induire de communications supplémentaires, mais n'est plus, à notre sens, un réel système de gestion des exceptions dans la mesure où il s'apparenterait plus à une valeur de retour spécifique pour signaler un cas inhabituel. De plus, ce système n'assure pas que tous les processus concernés par un calcul soient avertis de l'erreur. Cela nous conforte dans l'idée que, dans un contexte concurrent, la notion d'exception telle qu'elle est envisagée dans les langages séquentiels perd la majeure partie de son sens.

Nous avons l'avantage, dans ce contexte, d'utiliser un modèle de parallélisme structuré. Cette structure nous fournit une base solide pour établir des règles sur la propagation et le rattrapage qui nous assureront la sûreté et la cohérence d'exécution. Cependant, à notre connaissance, les différentes implantations de BSP ne proposent pas à l'heure actuelle de système de gestion d'exceptions spécifique – la plupart d'entre elles étant de simples bibliothèques. Par ailleurs, nous n'avons trouvé aucune étude sur les exceptions dans les langages data-parallèles ou fonctionnels parallèles excepté les travaux préalables de notre équipe [Dab03], portant sur un calcul restreint (BSλ) et sans implantation, et de travaux concernant les exceptions asynchrones dans Concurrent Haskell [MJM01]. Ces derniers permettent à un processus d'envoyer une exception à un autre de façon asynchrone, tout en préservant la nature fonctionnelle du langage (mais en abandonnant le déterminisme de l'exécution). Ces travaux se situent dans un contexte purement concurrent, et n'offrent pas d'éclairage quant à la gestion globale des exceptions dans un contexte parallèle.

Plus récemment, le langage en développement Manticore [FRRS08] devrait combiner les fonctionnalités de CML et NESL et offre un mécanisme d'exceptions qui semble suivre les mêmes principes que celui que nous allons décrire ici.

1.4 Réalisations

Deux axes principaux ont été explorés lors de ce travail : l'amélioration et l'extension du langage BSML, et l'étude de méthodes de développement parallèle certifié. Dans le chapitre 2, nous allons donc décrire l'approche du parallélisme que nous avons choisie et la base du langage BSML, ainsi que les traits qui y ont été ajoutés afin d'en faciliter l'utilisation. Le chapitre 3 sera l'occasion d'introduire une description formelle détaillée d'un sous-ensemble de BSML, à partir de laquelle l'ajout d'un mécanisme de traitement des exceptions et de références impératives pourront être étudiés formellement.

Garantir l'absence d'échec des programmes à l'exécution, au sens de Milner [Mil78], se fait généralement par l'ajout d'un système de types : dans notre cas, cette notion doit être étendue à des cas d'échec spécifiquement liés au parallélisme. En conséquence, il nous est nécessaire de

définir un système de typage étendu pour assurer la sûreté d'exécution des programmes BSML, ce que nous faisons au chapitre 4.

Le chapitre 5 discute pour sa part de l'implantation du langage et donne des exemples de programmation en BSML, ainsi que quelques résultats expérimentaux.

Le chapitre 6 donne un aperçu plus large de ce qu'il est possible de réaliser en BSML, en posant les bases permettant de construire une méthode de développement certifié : en combinant les différents outils décrits, il est possible, depuis une spécification d'algorithme, d'obtenir une implantation parallèle certifiée exacte d'un programme exécutant cet algorithme. Le chapitre 7 donne un cas d'application de cette méthode qui utilise les squelettes de listes, et fournit de nombreux exemples.

Chapitre 2

Principe de BSML

L'objectif de ce chapitre est de donner une vision claire de l'approche du parallélisme que nous avons choisi d'employer, ainsi qu'une compréhension générale de BSML : sa lecture devrait être suffisante pour permettre à un programmeur d'utiliser le langage. Les principes généraux retenus, tels que le modèle BSP, puis les concepts fondateurs de BSML sont présentés. Le rôle des quatre primitives de base permettant la manipulation du parallélisme est détaillé, et leur utilisation développée dans des exemples. Enfin, on donne la justification et le fonctionnement des extensions facilitant l'utilisation du langage.

2.1 Une approche du parallélisme

2.1.1 Le modèle BSP

La concurrence directe, comme nous l'avons vu, entraîne une très grande complexité dans l'analyse du flot d'exécution, et introduit de l'indéterminisme et la possibilité du blocage de l'exécution de manière imprévisible. Il est donc judicieux de s'en abstraire par l'utilisation d'un modèle de plus haut niveau, qui moyennant certaines restrictions rend l'analyse de programmes parallèles complexes possible.

Le modèle BSP (pour *Bulk Synchronous Parallel*, soit parallélisme quasi-synchrone), introduit dans [Val90], puis développé dans de nombreux travaux [Bis04, McC94, McC95, McC96a, McC96b], décrit la machine parallèle comme un ensemble de paires processeur-mémoire homogènes en nombre fixe p. C'est donc un modèle à mémoire distribuée. Les possibilités d'échanges entre ces processeurs sont modélisées par un réseau de communications, et une unité de synchronisation globale.

La particularité du modèle est que les communications ne peuvent être faites de façon quelconque. De fait, un processeur ne pourra bénéficier de données qu'il a reçues d'un autre processeur qu'une fois une barrière de synchronisation globale effectuée : de la sorte, on a l'assurance que toutes les communications se sont terminées, et on échappe aux conditions de concurrence qui pourraient se produire si l'on accepte la réception de données à un instant quelconque.

Cela conduit à un découpage de l'exécution en *super-étapes*, comme le montre la figure 2.1, où le temps s'écoule du haut vers le bas : l'exécution d'un programme BSP est une suite séquentielle

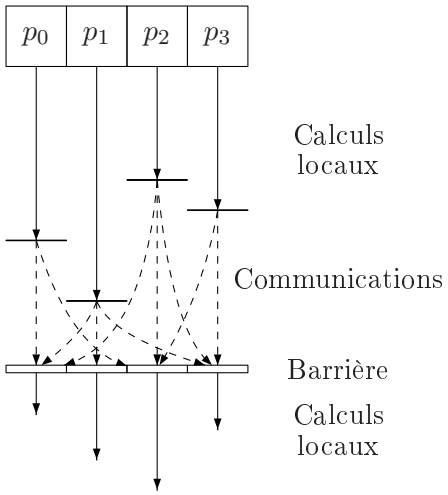


FIG. 2.1 – Exécution BSP sur une machine à 4 processeurs

de super-étapes, d'où sa qualification de *quasi-synchrone*. Chacune de ces super-étapes se déroule de la façon suivante :

1. Une phase de calculs locaux, indépendants, sur chaque processeur.
2. Une phase de communications, qui procède à tous les échanges qui ont pu être demandés pendant la première phase
3. Une barrière de synchronisation (le plus souvent logicielle). Notons qu'à certaines conditions, celle-ci peut être relâchée afin de limiter son impact sur les performances ([SCG04]).

Cela assure que l'exécution est déterministe, d'une part, et ne peut présenter d'interblocages d'autre part : deux caractéristiques extrêmement intéressantes dans notre cadre de programmation parallèle sûre.

Le modèle BSP a de plus l'intérêt de présenter un modèle de coûts relativement simple ; nous en reparlerons dans la section 2.1.3. Enfin, ce modèle est reconnu par la communauté scientifique et il existe une grande quantité de travaux sur les algorithmes parallèles BSP, ce qui donne du matériau pour le langage BSML, que nous allons maintenant présenter.

2.1.2 L'approche BSML

Plutôt que de redéfinir un langage fonctionnel à part entière, nous avons choisi d'utiliser comme base pour BSML un langage fonctionnel non-parallèle existant : comme nous allons le voir, l'approche que nous utilisons est largement orthogonale au langage fonctionnel sous-jacent (le BSλ-calcul [LHF00], base théorique de BSML, est en ce sens une extension du λ-calcul). Le choix d'un langage strict ou non est néanmoins d'une importance cruciale pour la suite. Notre choix se dirige naturellement vers un langage strict pour des raisons de prévisibilité de l'exécution et de la performance : notre souhait que le parallélisme soit explicite nécessite un ordre d'exécution explicite et prévisible que ne fournissent pas les langages non-stricts [Hai94]. Notons cependant que ce choix ne va pas de soi dans le cas général, et qu'en particulier une grande quantité de travaux sont consacrés au parallélisme dans le langage non-strict Haskell :

dans *Glasgow Parallel Haskell* (GPH) [Tri99], le parallélisme est semi-explicite et utilise une gestion de *threads* dynamique.

Parmi les langages fonctionnels à évaluation stricte, ML [MTHM97] est le standard majeur, a donné lieu à plusieurs implantations et est largement utilisé. BSML se base sur Objective Caml (OCaml dans la suite) [LDG+07] ; ce choix s'impose parmi les différentes variantes de ML pour des raisons de performance de l'implantation, puisque nos applications se destinent à la programmation haute performance [HFA+96]. D'autres raisons, comme le grand nombre de bibliothèques et les outils disponibles pour ce langage nous ont conforté dans ce choix. Camlp4 [DdR07], le pré-processeur générique d'OCaml, est par exemple un outil que nous utilisons dans le développement des extensions de BSML.

La syntaxe de BSML, ou du moins son cœur, est donc directement celle d'OCaml – moyennant quelques restrictions qui seront développées dans la suite. Il est en effet possible de lire un programme BSML comme un programma OCaml pour la majeure partie. L'ordre d'exécution, en particulier – parfois obscur lorsqu'on offre la possibilité de définir des opérations simultanées – ne devrait pas dérouter un habitué de la programmation fonctionnelle et d'OCaml. En outre, les programmes OCaml usuels sont directement des programmes BSML valides, n'utilisant pas ses traits parallèles et s'exécutant donc de façon séquentielle. La parallélisation d'applications OCaml à l'aide de BSML peut donc être faite de façon incrémentale.

Objective Caml est également l'une des cibles possibles pour l'extraction de programmes dans l'assistant de preuves Coq [Let04]. Il nous est ainsi possible d'étendre cette propriété à BSML et d'extraire des programmes parallèles prouvés depuis des définitions Coq, comme nous le faisons au chapitre 6.

Il n'est pas dans notre propos ici de décrire en détail la syntaxe et l'usage d'OCaml. En cas de doute, le lecteur pourra se référer à [LDG+07]. Les points du langage important dans notre contexte seront néanmoins expliqués ; les sémantiques que nous définissons, en outre, sont autonomes et ne dépendent pas d'autres travaux bien qu'elles demandent une certaine familiarité du lecteur avec ce type de formalisme.

2.1.3 Prédiction de performance

L'analyse de complexité est une préoccupation majeure en algorithmique : l'étude de nouveaux algorithmes, quel que soit le domaine, est tributaire des moyens qu'il existe pour les comparer entre eux. On s'abstrait généralement dans ce cas des spécificités du matériel et, lorsqu'on étudie des algorithmes séquentiels, on se contente de leur comportement asymptotique. Ces résultats mathématiques sont ainsi beaucoup plus pertinents que de simples tests de performance sur des cas particuliers.

Cette analyse est d'autant plus difficile quand on étudie des algorithmes parallèles. En effet, au coût de calcul pur s'ajoute le coût des communications et synchronisations : et on dépend ici de la façon dont sont spécifiés les échanges, et donc du modèle utilisé. En premier lieu, on ne peut pas étudier de la même façon le coût d'un algorithme spécifié dans le cadre d'un système à mémoire partagée ou dans celui d'un système par échange de messages. C'est pourquoi il est important que, dès la conception du modèle, la facilité à évaluer le temps de calcul soit prise en compte.

Ceci, à trois niveaux : le premier, et le moins formel mais non le moindre, est celui du

programmeur. En particulier dans le cadre du calcul haute performance, il est indispensable que le programmeur ait une bonne intuition du coût du code qu'il écrit, afin qu'il soit en mesure de minimiser celui-ci. Pour cela, un modèle de coût simple et explicite nous paraît être un atout important de BSP, par rapport aux systèmes à parallélisme ou communications implicites. Il est trop souvent admis dans ce contexte que la prévisibilité des coûts est inversement proportionnelle au niveau du langage : un langage de bas niveau offrirait une bonne maîtrise des coûts, et un langage de haut niveau demanderait un abandon de cette maîtrise. Cela est vrai dans une certaine mesure : l'existence d'un *garbage collector* dans un langage, par exemple, rend les opérations d'allocation de mémoire implicites, et fait perdre, dans une faible mesure, la maîtrise des coûts. L'argument perd de sa valeur lorsqu'on fait du calcul haute performance parallèle : même dans le contexte d'une programmation très bas niveau, trop de facteurs entrent en jeu (latence et vitesse du réseau, pagination...) pour qu'on puisse prétendre maîtriser la performance d'un programme. Même quand la performance est bien estimée ou obtenue par des tests répétitifs et coûteux en temps, elle se révèle très variable d'une architecture à l'autre. On constate souvent, à l'opposé, que l'utilisation d'un langage parallèle de plus haut niveau atténue l'effet aléatoire de ces facteurs et rend la performance plus proche des estimations [Gor04].

Le deuxième niveau d'estimation de la performance se rapproche de l'analyse de complexité. Lorsqu'on propose un algorithme parallèle suivant un modèle donné, il doit être possible le plus simplement possible d'obtenir une formule de coûts permettant de le comparer à d'autres. Le modèle BSP offre un modèle de coûts simple, qui permet une estimation en fonction de trois paramètres :

- p, le nombre de processeurs dans la machine parallèle.
- g, qui décrit la vitesse du réseau de communication. h correspond au temps, en nombre d'opérations locales, qui est nécessaire à un envoi/réception d'au plus un mot mémoire par processeur.
- L, qui est le temps que coûte la barrière de synchronisation en nombre d'opérations locales.

Le coût d'exécution d'une super étape est ainsi le maximum des temps de calculs locaux, plus le temps de communication (qui est égal à $n \times g$, où n est le nombre de mots mémoire maximal qu'un processeur a reçu ou envoyé), et le temps de synchronisation. Cette formule est ainsi adaptée à différentes architectures matérielles, en utilisant les paramètres obtenus par des tests simples, pour obtenir une estimation fiable du temps d'exécution de l'algorithme ([Gav08, Kru08]).

Le troisième niveau concerne l'estimation dynamique de performance. En effet, l'utilisation optimale du parallélisme repose généralement sur un compromis entre équilibrage de la charge et communications. Les paramètres BSP de la machine étant connus au cours de l'exécution, des formules de coût établies au préalable permettent de choisir dynamiquement le compromis optimal en fonction de l'architecture [Bam00, BH99].

Les coûts en BSML sont intuitifs : le parallélisme, les communications et les barrières sont explicites, et suivent le modèle BSP. Un modèle de coût sémantique détaillé est développé dans [Gav05] ; il est de plus possible, en étendant la sémantique formelle que nous proposons en Coq, de prouver le coût d'algorithmes en même temps que leur implantation. Cet axe de recherche n'a pas encore été exploré à l'heure actuelle.

2.1.4 Historique

Avant de détailler le fonctionnement de BSML, voici un aperçu des évolutions qui l'ont amené à son état actuel. Les premièrs travaux remontent à 1998, où un prototype de simulation (d'exécution purement séquentielle) permettant une analyse de coûts a été réalisé [Lou98], ainsi que la première version du BSλ-calcul [LHF98] permettant son étude. La première version du langage, 0.1, est sortie en 2000 après une amélioration des primitives de communication d'origine [BLH99].

La suite a donné lieu à de nombreuses études théoriques sur le langage : sémantique distribuée [Lou01], machines abstraites [MHL01, GLD03], et certification de programmes [GL03, Gav03].

Une nouvelle implantation modulaire de BSML en 2005 [LGB05] a donné lieu à une nouvelle primitive parallèle, proj, qui améliore grandement la souplesse et l'expressivité du langage. Dans [Gav05], l'auteur fait le point sur les développements de BSML et donne une étude sémantique du langage très détaillée, fournissant les bases et des exemples pour des développements parallèles prouvés.

Le présent travail s'établit sur cette base pour étendre le langage et le rendre plus général, en particulier avec des fonctionnalités non purement fonctionnelles. Des études précédentes sur le filtrage d'expressions parallèles par motifs [DLG03] comblaient l'absence de la primitive proj et se rapprochent assez peu du filtrage que nous proposons ici, qui s'ajoute à cette primitive et augmente l'expressivité du langage. Le traitement des références en BSML a été pour la première fois étudié dans [GL04] : le présent travail se base sur ce formalisme et l'étend en en assurant la sûreté par typage, et en définissant la communication de références. La gestion des exceptions, quant à elle, reprend certaines idées de [Dab03] et traite les problèmes qui y sont laissés ouverts en fournissant un mécanisme complet et une implantation.

Enfin, nos développements certifiés suivent le principe de [Gav05] et fournissent une nouvelle version des définitions du langage.

2.2 Vecteurs parallèles

Nous n'avons pas besoin, au niveau de la syntaxe, d'un grand nombre de points d'entrée dans le parallélisme : c'est ce qui justifie de baser notre langage sur un langage fonctionnel existant. BSML se base sur un type que nous appellerons *vecteur parallèle* et qui, seul, permet l'accès au parallélisme. Un vecteur parallèle a pour type α par et contient p valeurs *locales* de type α (ou 'a dans la syntaxe du langage). Chacune de ces valeurs est hébergée dans la mémoire de l'un des processeurs de notre machine parallèle. Le nombre de processeurs p est défini comme une constante bsp_p tout au long de l'exécution du programme.

Ces vecteurs parallèles sont différents des *tableaux parallèles* tels qu'utilisés pour le parallélisme de données dans des langages tels que NESL [BHS+94], Nepal cou Manticore [FRR+07] : dans ces tableaux, les données et les opérations sur celles-ci sont implicitement réparties sur les processeurs. Ils consistent donc en une approche de plus haut niveau pour le parallélisme de données, alors que les vecteurs parallèles correspondent à un contrôle direct du parallélisme, plus général.

Nous identifierons les p processeurs de la machine BSP par des entiers de 0 à $p-1$ tout au long de cette thèse ; cet identifiant sera appellé *pid* (pour «Processor IDentifier») du processeur.

L'ensemble des pids sera noté $\mathcal{P}$; on note un vecteur parallèle de la façon suivante :

$$\langle x_0, x_1, \ldots, x_{p-1} \rangle : \alpha \,\mathsf{par}$$

ou bien en notation abrégée, $\langle x_i \rangle_i$. Ce vecteur contient la valeur x_i au processeur i, avec tous les x_i de type α. On note dans le cas général $\langle\ \rangle_i$ pour signifier que le pid du processeur est représenté par l'index i dans le vecteur, $\langle i \rangle_i$ étant donc le vecteur des pids.

Cette structure se distingue d'un vecteur de taille p habituel par le fait que les valeurs qu'il contient sont isolées les unes des autres. On ne peut en effet accéder à x_i que dans deux cas :
- lors de calculs locaux sur le processeur i
- après des communications

On ne peut pas, par exemple, comparer deux valeurs locales sur des processeurs différents sans l'usage de primitives de communication.

Ces règles constituent l'essence même du parallélisme à mémoire distribuée ; l'usage du type opaque $\alpha\,\mathsf{par}$ nous permet de les garantir. Ce type rend également le parallélisme explicite, et les programmes plus lisibles. Il reste bien sûr possible, comme pour tout autre type, d'imbriquer les vecteurs parallèles dans d'autres types, ou de leur appliquer des fonctions polymorphes : le même traitement sera appliqué sur chaque composante, indépendamment de son contenu. On peut de la sorte utiliser `List.map` de la bibliothèque standard sur une liste de vecteurs parallèles, par exemple.

les vecteurs parallèles peuvent être manipulés par l'intermédiaire de quatre primitives, qui forment le cœur de BSML : deux d'entre elles sont asynchrones, et correspondent à un accès local aux valeurs de chaque composante des vecteurs, les deux autres effectuent des communications et sont synchrones.

2.3 Niveaux d'exécution dans un programme BSML

BSML gère à la fois une machine parallèle dans son ensemble et des processeurs individuels. Une distinction entre les différents niveaux d'exécution à l'intérieur de la machine parallèle va ainsi nous être utile.

- L'exécution **répliquée** est celle qui a lieu par défaut, dans du code séquentiel n'utilisant pas les primitives BSML (et donc pas les vecteurs parallèles). Ce code est exécuté comme si la machine disposait d'un processeur unique. Dans la pratique, son exécution est faite simultanément sur tous les processeurs afin de ne nécessiter aucune communication, d'où le nom ; cela impose que ce code soit strictement déterministe, afin qu'on ait la garantie d'avoir le même résultat partout. Du point de vue de l'utilisateur, l'exécution pourrait avoir lieu sur un processeur séquentiel unique.
- L'exécution **locale** est celle qui se produit dans les vecteurs parallèles, sur chacune de leurs composantes : les processeurs utilisent leurs données locales pour faire des calculs qui peuvent différer de l'un à l'autre. La fonction passée en argument de `mkpar`, par exemple, est exécutée localement avec des paramètres différents. Les exécutions locales et répliquées sont disjointes, et typiquement les programmes alternent entre les deux.
- L'exécution **globale** concerne les processeurs dans leur ensemble en tant que tout, mais met en œuvre le parallélisme contrairement à l'exécution répliquée ; typiquement, on considère

les communications comme une opération globale. Lorsqu'on considère une opération sur un vecteur parallèle, l'exécution globale est un ensemble d'exécutions locales.

La base du modèle de BSML est la stricte distinction entre exécutions répliquée et locale, qui est assurée par la structure de vecteur parallèle, opaque. En effet, de la sorte le contenu des vecteurs ne peut être lu en mode répliqué, ce qui assure la cohérence même du mode répliqué.

Des précautions supplémentaires sont nécessaires si l'on traite les références, les entrées-sorties ou les opération non déterministes ; nous les traitons dans la suite.

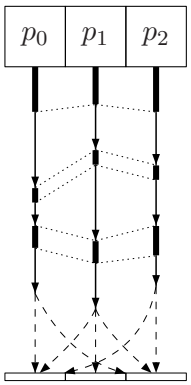


FIG. 2.2 – Exécution BSML sur une machine à trois processeurs

La figure 2.2 représente le déroulement d'un programme BSML sur une machine à trois processeurs, l'exécution répliquée étant figurée en trait épais, et l'exécution locale en trait fin. On remarque que l'exécution répliquée, bien que cela ne prête pas à conséquence, n'est pas nécessairement exécutée simultanément sur tous les processeurs. BSML offre de plus la possibilité d'exécuter le code séquentiellement, à des fins de tests, tout en ayant la garantie d'obtenir le même résultat que pour une exécution parallèle. De la sorte, il fournit un mode interactif utile pour le typage et l'étude rapide de fragments de programmes.

primitive	**type**	**description**
mkpar	$(\text{pid} \to \alpha) \to \alpha\ \mathsf{par}$	$f \mapsto \langle f\ 0, \ldots, f\ (p-1)\rangle$
apply	$(\alpha \to \beta)\ \mathsf{par} \to \alpha\ \mathsf{par} \to \beta\ \mathsf{par}$	$\langle f_0, \ldots, f_{p-1}\rangle \mapsto \langle x_0, \ldots, x_{p-1}\rangle \mapsto$ $\langle f_0\ x_0, \ldots, f_{p-1}\ x_{p-1}\rangle$
proj	$\alpha\ \mathsf{par} \to \text{pid} \to \alpha$	$\langle x_0, \ldots, x_{p-1}\rangle \mapsto \mathsf{fun}\, i \to x_i$
put	$(\text{pid} \to \alpha)\ \mathsf{par} \to (\text{pid} \to \alpha)\ \mathsf{par}$	$\langle f_0, \ldots, f_{p-1}\rangle \mapsto$ $\langle \mathsf{fun}\, i \to f_i\ 0, \ldots, \mathsf{fun}\, i \to f_i\ (p-1)\rangle$

FIG. 2.3 – Récapitulatif des primitives BSML

2.4 Primitives asynchrones

2.4.1 mkpar

mkpar: (pid → 'a) → 'a par est le point d'entrée dans le parallélisme BSML : cette primitive construit un vecteur parallèle dont les composantes sont les résultats de l'application locale de la fonction passée en argument au pid de chaque processeur (on utilise, pour le confort de lecture, le type pid pour désigner les identifiants de processeurs, celui-ci étant défini en tant que int).

$$\text{mkpar } f = \langle f\ 0, \ldots, f\ (p-1) \rangle$$

mkpar est la façon normale de créer un vecteur parallèle – les autres primitives permettant la transformation à partir de vecteurs déjà existants.

Exemples usuels

- Répliquer une valeur :

  val replicate : 'a → 'a par
  ```
  let replicate x = mkpar (fun _ → x)
  ```
  replicate x = $\langle x, x, \ldots, x \rangle$
- Vecteur des pids (en tant que int) :

  val pids : int par
  ```
  let pids = mkpar (fun i → i)
  ```
  pids = $\langle 0, 1, \ldots, p-1 \rangle$
- **mkpar** est souvent utilisé pour diviser un ensemble de données entre les processeurs. La fonction suivante donne un moyen simple de distribuer une liste entre les processeurs.

  val select_list : 'a list → 'a list par
  ```
  let select_list l =
    let len = List.length l in
      mkpar (fun i → cut_list l (i * len / bsp_p) ((i + 1) * len / bsp_p))
  ```
  La fonction cut_list l a b utilisée renvoie le segment constitué des éléments de l d'index compris entre a (inclus) et b (exclus). Un exemple plus avancé de distribution équilibrée sur un arbre est exposé dans la section 5.6

2.4.2 apply

apply: ('a → 'b) par → 'a par → 'b par est la primitive des calculs locaux. Elle applique une fonction locale à un paramètre local sur chaque processeur.

$$\text{apply } \langle f_0, \ldots, f_{p-1} \rangle\ \langle x_0, \ldots, x_{p-1} \rangle = \langle f_0\ x_0, \ldots, f_{p-1}\ x_{p-1} \rangle$$

Exemples usuels

- Le plus souvent, on désire appliquer une même fonction sur tous les membres d'un vecteur parallèle, à la façon du parallélisme de données. C'est aussi simple que :

```
val parfun : ('a → 'b) → 'a par → 'b par
let parfun f v = apply (replicate f) v
```

– Si on désire appliquer une fonction sur tous les éléments d'une liste distribuée (obtenue, par exemple, grâce à la fonction select_list), à la façon de List.map, on peut écrire :

```
val parmap : ('a → 'b) → 'a list par → 'b list par
let parmap f parlist = parfun (List.map f) parlist
```

– apply est utilisé pour toutes les manipulations de vecteurs n'ayant pas trait à leur structure. Ne conserver qu'une seule valeur dans un vecteur (cela peut être utile avant communication) se fait par :

```
val filter_pid : 'a par → pid → 'a option par
let filter_pid v i =
  let filter = mkpar (fun j → if i = j then fun x → Some x else fun x → None) in
    apply filter v
```

filter est ici un vecteur de fonctions qui, sur le processeur i, contient la fonction qui emboîte dans un type option (Some x), et sur les autres processeurs renvoie toujours None. filter_pid pids 2 sur une machine à quatre processeurs serait ainsi le vecteur ⟨None, None, Some 2, None⟩.

2.5 Primitives synchrones

2.5.1 proj

proj: 'a par → pid → 'a est la primitive duale[2] de mkpar, et le seul moyen d'extraire une valeur non parallèle à partir d'un vecteur parallèle. Étant donné un vecteur, proj renvoie une fonction non parallèle qui, appliquée à un pid, renvoie la valeur de la composante du vecteur correspondant à ce pid.

$$\mathsf{proj}\langle x_0, \ldots, x_{p-1}\rangle = \mathsf{fun}\, i \to x_i$$

proj effectue des communications, afin de rendre des résultats locaux accessibles globablements par l'intermédiaire de la fonction renvoyée. Par conséquent, il établit un point de rendez-vous pour tous les processeurs, et – en termes BSP – met fin à la super-étape courante. proj est souvent utilisé à la fin d'un calcul parallèle, afin de rassembler les résultats obtenus.

Exemples usuels

– proj est souvent utile pour récupérer les résultats d'un processeur unique avec proj v i. Cette formulation peut provoquer des communications inutiles, on préfère donc la remplacer par[3] :

```
val at : 'a par → pid → 'a
let at v i = match proj (filter_pid v i) i with Some x → x
```

– Il est parfois utile de convertir un vecteur parallèle en liste plutôt qu'en fonction :

[2] cette dualité se limite aux pids valides ; elle est formalisée en 6.3.4.0

[3] La valeur particulière None ne provoque aucune communication ; nous nous étendrons sur le sujet plus tard

```
val proj_list : 'a par → 'a list
let proj_list v = List.map (proj v) procs_list
```

où procs_list est la liste des pids : [0; 1; ... bsp_p−1].

– Il est simple de rassembler une liste distribuée par la fonction select_list, afin de reconstituer une liste globale :

```
val gather_list : 'a list par → 'a list
let gather_list parlist = List.concat (proj_list parlist)
```

La définition de fonctions similaires de distribution et rassemblement pour d'autres types de données permet de faire de la programmation de type parallélisme de données en BSML. Notons que nous utiliserons plutôt une version récursive terminale de List.concat, pour traiter les grands ensembles de données (plus d'une dizaine de milliers d'éléments par processeur) :

```
val concat : 'a list → 'a list list → 'a list
let rec concat acc ll = match ll with
  | l::lr → concat (List.rev_append l acc) lr
  | [] → List.rev acc

val gather_list : 'a list par → 'a list
let rec gather_list parlist = concat [] (proj_list parlist)
```

– La fonction précédente peut être généralisée par une réduction simple en une étape[4] :

```
val simple_reduce : ('a → 'a → 'a) → 'a par → 'a
let simple_reduce op v =
  let vl = proj_list v in
    List.fold_left op (List.hd vl) (List.tl vl)
```

– En tant qu'exemple de ce qui précède, montrons comment on peut tester qu'une condition locale est vérifiée partout afin de prendre une décision globale :

```
val test_local : ('a → bool) → 'a par → bool
let test_local condition v = simple_reduce (&& ) (parfun condition v)
```

La fonction que **proj** renvoie lève une exception si elle est appliquée à un pid invalide (négatif, ou supérieur à $p-1$). Le choix a été fait de définir nos primitives de façon purement fonctionnelle, mais nous aurions aussi bien pu choisir une interface reposant sur des tableaux de taille p ou des listes. Comme nous le voyons dans les exemples, les conversions entre ces approches sont simples et ce n'est qu'un choix d'interface.

2.5.2 put

put: (pid → 'a) par → (pid → 'a) par est la primitive générique de communications : elle permet de décrire tout échange de valeurs locales entre processeurs. Comme **proj**, elle met implicitement fin à la super-étape en cours.

$$\mathsf{put}\langle f_0, \ldots, f_{p-1}\rangle = \langle \mathsf{fun}\, i \to f_i\ 0, \ldots, \mathsf{fun}\, i \to f_i\ (p-1)\rangle$$

[4] Nous donnons ici la fonction dont la lecture est la plus facile, la version optimisée ne reposant pas sur des listes

put prend pour paramètre un vecteur de fonctions f_i : pid $\rightarrow \alpha$ tel que $f_i\, x$ donne la valeur que le processeur i doit envoyer au processeur x. L'usage canonique de **put** est le suivant :

```
put (mkpar (fun sender sendto → e))
```

où l'expression e calcule (ou plutôt sélectionne, en général), en fonction des pids sender et sendto, la valeur qui devra suivre le chemin de l'un à l'autre. La valeur renvoyée par **put** est homogène avec son paramètre : c'est un vecteur de fonctions f'_i duales telles que $f'_i\, x$ indique la valeur que le processeur i à reçue du processeur x.

Exemples usuels

– Il est possible d'obtenir un résultat similaire à celui de **put** à partir des primitives **proj** et **mkpar** :

```
val xput : (pid → 'a) par → (pid → 'a) par
let xput f =
  let pr = proj f in
    mkpar (fun i j → pr j i)
```

On intercale ici une transposition globale entre les appels aux deux fonctions duales **proj** et **mkpar**. Le résultat est bien sûr beaucoup moins efficace en termes de communications qu'un appel à **put**, qui envoie directement la valeur à sa destination. Inversement, les communications de **proj**, plus simples, peuvent facilement être décrites en termes de **put** si l'on se munit d'une opération (non sûre) lift_vector : 'a par → 'a qui renvoie une valeur répliquée à partir d'un vecteur parallèle dont toutes les composantes sont égales.

```
val xproj : 'a par → pid → 'a
let xproj v = lift_vector (put (apply (mkpar (fun _ v' _ → v')) v))
```

Dans les faits, ceci est proche de la façon dont **proj** est implanté.

– On peut considérer le paramètre de **proj** comme une matrice $p \times p$, contenant les p valeurs à envoyer depuis chacun des p processeurs. **proj** est alors une opération de transposition.

$$\text{put}\left\langle \begin{array}{cccc} f_0\ 0 & f_1\ 0 & \cdots & f_{p-1}\ 0 \\ f_0\ 1 & f_1\ 1 & & f_{p-1}\ 1 \\ \vdots & & \ddots & \\ f_0\ (p-1) & f_1\ (p-1) & & f_{p-1}\ (p-1) \end{array} \right\rangle = \left\langle \begin{array}{cccc} f_0\ 0 & f_0\ 1 & \cdots & f_0\ (p-1) \\ f_1\ 0 & f_1\ 1 & & f_1\ (p-1) \\ \vdots & & \ddots & \\ f_{p-1}\ 0 & f_{p-1}\ 1 & & f_{p-1}\ (p-1) \end{array} \right\rangle$$

La chose est plus évidente si on se base sur des tableaux que sur des fonctions, à l'aide du code suivant :

```
val put_array : 'a array par → (pid → 'a) par
let put_array comm_array = put (parfun Array.get comm_array)
```

parfun applique localement Array.get afin d'obtenir une fonction de int, comme le requiert **put**. On peut obtenir un résultat homogène sous forme de tableau par :

```
val put_array2 : 'a array par → 'a array par
let put_array2 comm_array = parfun (Array.init bsp_p) (put_array comm_array)
```

Array.init p f crée un tableau de taille p à partir de la fonction f : c'est le traitement dont nous avons besoin.

– Réduction en plusieurs étapes : le précédent exemple de réduction ne fait pas usage du parallélisme. Il convient parfaitement dans le cas où l'opérateur de combinaison utilisé a un coût très faible, mais il est souvent plus intéressant de faire la réduction de manière diviser-pour-régner avec des combinaisons faites localement.
L'algorithme suivant, à la super-étape n pour n allant de 0 à $\lceil \log_2 p \rceil$, combine les valeurs des processeurs i et $i + 2^n$. Le résultat final est calculé sur le processeur 0.

```
val reduce : int → ('a → 'a → 'a) → 'a → 'a par → 'a par
let rec reduce step op unit v =
  if step >= bsp_p then v else
    (* les valeurs sont envoyées à tout processeur
       rassembleur "dest" depuis "dest+step" *)
    let comm = put (apply
                      (mkpar (fun src v dest →
                                  if (dest mod (2*step) = 0) && (src = dest + step)
                                  then v else unit))
                      v)
    (* combinaison de la valeur locale courante avec la valeur reçue,
       aux processeurs rassembleurs *)
    in let v' = apply (apply (mkpar (fun i v comm →
                                       if i mod (2*step) = 0 then
                                         if i+step < bsp_p then
                                           op v (comm (i + step))
                                         else v
                                       else unit))
                             v)
                      comm
    in reduce (step*2) op unit v'
```

Le programme (reduce 1)[5], moyennant un opérateur associatif op et une «unité» de communications unit (typiquement, None ou [], nous y reviendrons), rassemble les données sur chaque processeur de pid pair, puis sur les multiples de 4, 8, *etc.* La première partie du code effectue les communications : le paramètre de **put** renvoie l'unité de communication sauf dans le cas de communications de $(2*\text{step}+1)*i$ vers $2*\text{step}*i$, pour tout i. Le rassemblement des données de ces deux processeurs est alors fait sur le processeur «rassembleur» $2*\text{step}*i$ à l'aide de op. À la dernière étape, les deux derniers ensembles sont réduits sur le processeur 0.

put est la primitive la plus délicate à utiliser du fait de sa généralité : elle permet tout schéma de communication inter-processeurs, et est ainsi utilisée aussi bien pour des opérations complexes, telles que le rééquilibrage des données entre les processeurs.

[5] On peut trouver la lecture de ce programme délicate. En particulier, la définition du paramètre de **put** et les appels imbriqués à **apply** sont quelque peu ardus. La syntaxe alternative que nous décrivons en 2.7 améliore significativement la lisibilité ; le lecteur impatient pourra trouver la version correspondante de reduce p.28. D'autre part, les usages de **put** sont souvent faits par l'intermédiaire de fonctions intermédiaires, plus faciles d'usage – comme celle-ci.

2.5.3 Remarque par rapport aux définitions antérieures des primitives

Dans les versions antérieures de BSML, le paramètre des primitives de communication était de type option afin de permettre la définition de schémas de communications partiels. Rappelons que le type option, en OCaml, est défini par :

```
type 'a option = None | Some of 'a
```

La valeur None indiquait ainsi qu'aucune communication n'était souhaitée dans le paramètre, ou qu'aucune communication n'avait été reçue dans la valeur de retour. Bien que satisfaisante au niveau de la définition du langage, cette approche se révélait vite encombrante au niveau de la définition de programmes, contraignant à constamment ajouter des constructeurs Some et les filtrer pour communiquer et utiliser les résultats.

L'approche présentée ici, utilisant des types quelconques, peut paraître moins générale car elle ne semble pas permettre la définition de schémas de communications partiels : dans le cas de **put**, par exemple, une valeur à échanger doit être fournie pour toutes les paires de processeurs. Grâce à une astuce de programmation, certes pragmatique mais allégeant considérablement le code, ce n'est pas le cas : on définit une valeur particulière que nous appellerons «unité» à l'intérieur de chaque type, et cette valeur sera désignée comme correspondant à une absence de communications et restituée de la sorte à l'arrivée. None est ainsi l'unité du type option, ce qui rend le système entièrement compatible avec l'ancienne approche et donc au moins aussi général. La liste vide, le tableau vide, l'unité (), l'entier 0, le premier constructeur des types somme définis par l'utilisateur s'il est sans argument) sont tous les unités de leurs types respectifs, ce qui rend les primitives de communications plus souples à l'usage.

Typiquement, il est confortable et satisfaisant de pouvoir utiliser la liste vide lorsqu'on applique **put** sur des listes et qu'on souhaite représenter l'absence de communications. Ce mécanisme est plus compliqué à définir dans le langage, et peut-être moins satisfaisant d'un point de vue théorique (à cause de la notion d'unité qui transcende les types, et reste certes un peu floue), mais il reste sûr et d'un grand intérêt pratique. La subtilité, en outre, concerne les communications qui sont faites, et a donc un impact sur le coût du programme mais pas sur son résultat final.

2.6 Exemple : tri parallèle par échantillonnage régulier (PSRS)

Les algorithmes de tri sont souvent les plus étudiés, car ils présentent un intérêt théorique et sont très utilisés en pratique [DFRC96, Las99, CDT05]. Ils ne font pas exception à la règle qui veut que les algorithmes parallèles les plus efficaces ressemblent peu à leurs homologues séquentiels. Le tri parallèle par échantillonnage régulier [Tis98] est parmi les plus simples, et reste très efficace même sur des données non homogènes. Il procède en quatre étapes, la donnée d'origine étant une liste répartie sur les p processeurs :

1. Le tri local des p sous-listes, sur chaque processeur. Le problème est alors ramené à la fusion de ces listes.
2. Le choix global de $p-1$ pivots, afin de partitionner l'ensemble des données en p sous-ensembles.
3. La communication de ces sous-ensembles : le processeur i reçoit de chaque autre la sous-liste d'éléments compris entre le $(i-1)^e$ et le i^e pivots.

4. La fusion locale de toutes les sous-listes ordonnées reçues.

Le résultat est une liste ordonnée répartie sur les p processeurs. L'efficacité de l'algorithme repose sur la bonne répartition des données entre eux, et donc sur le choix judicieux des pivots lors de l'étape 2. C'est là qu'entre en jeu l'échantillonnage régulier : il se déroule en trois étapes

a. Chaque processeur extrait $p-1$ pivots potentiels de sa liste locale triée, répartis régulièrement.
b. Ces listes de pivots potentiels sont fusionnées globalement, formant une liste ordonnée de taille $p(p-1)$.
c. $p-1$ pivots sont choisis régulièrement espacés dans cette liste.

Ormis l'extraction des pivots, qui demande un parcours des listes locales (taille de la liste divisée par p), ces opérations sont en temps constant par rapport à la taille de la liste. Notons que le bon choix des pivots permet également d'obtenir en résultat une liste distribuée de façon homogène, ce qui pourra être utile dans la suite des calculs.

PSRS est simple à implanter en BSML. Le programme prend en paramètres le vecteur de listes correspondant à la liste répartie lv, et le vecteur des longueurs locales de ces listes lvlengths (donc lvlengths = parfun List.length lv)

```
(* tri parallèle par échantillonnage régulier *)
val psrs : int par → 'a list par → 'a list par
let psrs lvlengths lv =
    (* étape 1: tri local *)
  let locsort = parfun (List.sort compare) lv in
    (* étape 2a: extraction locale de (p−1) échantillons *)
  let regsampl = apply (parfun (fun l len → extract_n bsp_p len l) locsort) lvlengths in
    (* étape 2b: échange des listes d'échantillons *)
  let glosampl = List.sort compare (gather_list regsampl) in
    (* étape 2c: sélection globale des pivots parmi les échantillons *)
  let pivots = extract_n bsp_p (bsp_p*(bsp_p−1)) glosampl in
    (* étape 3: partitionnement des listes locales et communication *)
  let comm = parfun (fun l → slice_p l pivots) locsort in
  let recv = put (parfun List.nth comm) in
    (* étape 4: fusion locale des données reçues *)
    parfun (fun ll → p_merge bsp_p (List.map ll procs_list)) recv
```

La fonction extract_n n, définie plus bas, extrait de la liste $n-1$ éléments équitablement répartis ; slice_p transforme une liste ordonnée en liste de liste en fonction des pivots, et p_merge p fusionne (de façon séquentielle) p listes triées. Les étapes 1 et 2a sont asynchrones, n'utilisant que parfun et apply sur les données locales. L'étape 2b effectue un échange global des échantillons à l'aide de gather_list défini plus haut. L'étape 2c est entièrement globale, travaillant sur les échantillons ; l'étape 3 est synchrone et effectue, en une seule étape, l'échange des données à l'aide de put. L'étape 4 est asynchrone, terminant le tri de façon locale.

Tout le parallélisme est contrôlé par la fonction implantant l'algorithme : les fonctions auxiliaires extract_n, slice_p et p_merge sont des fonctions OCaml standard (on voit d'ailleurs dans cet exemple que extract_n est utilisé localement et globalement). En voici une implantation :

```
(* renvoie la liste des éléments de l dont l'index i est tel que f i *)
val filter_nth : (int → bool) → 'a list → 'a list
let filter_nth f l =
  let rec aux i = function
    | x::r → if f i then x::(aux (i+1) r) else aux (i+1) r
    | [] → []
  in aux 0 l

(* extrait n−1 éléments régulierement répartis dans l, de longeur len *)
val extract_n : int → int → 'a list → 'a list
let extract_n n len l =
  filter_nth (fun i → (n * i − 1) mod len >= len − n) l

(* Sépare une liste ordonnée l en deux par rapport à p *)
val split_lt : 'a list → 'a list → 'a → 'a list * 'a list
let rec split_lt acc l p = match l with
  | x::r → if x < p then split_lt (x::acc) r p
      else (List.rev acc),l
  | [] → (List.rev acc),[]

(* Partitionne une liste ordonnée l par rapport à une liste de pivots *)
val slice_p : 'a list → 'a list → 'a list list
let rec slice_p l pivots = match pivots with
  | p::r → let l1,l2 = split_lt [] l p in
      l1::(slice_p l2 r)
  | [] → [l]

(* fusionne deux listes ordonnées *)
val merge : 'a list → 'a list → 'a list → 'a list
let rec merge acc l1 l2 = match l1,l2 with
  | x1::r1,x2::r2 → if x1 < x2 then merge (x1::acc) r1 l2
      else merge (x2::acc) l1 r2
  | [],l | l,[] → List.rev_append acc l

(* renvoie le couple des n premiers éléments d'une liste et du reste *)
val splitn : int → 'a list → 'a list * 'a list
let rec splitn n (x::r) =
  if n=0 then [],x::r
  else let l1,l2 = splitn (n−1) r in (x::l1,l2)

(* fusionne p listes ordonnées *)
val p_merge : int → 'a list list → 'a list
let rec p_merge p = function
  | [] → []
  | l::[] → l
  | ll → let ll1,ll2 = splitn (p/2) ll in
      merge [] (p_merge (p/2) ll1) (p_merge (p−p/2) ll2)
```

On remarquera que merge et split_lt sont récursives terminales, ce qui est nécessaire car elles sont appliquées sur des listes dont la taille dépend de celle de la liste à trier. Les autres fonctions ne seront appliquées que sur des listes dont la taille dépend de p, cet effort n'est donc pas utile.

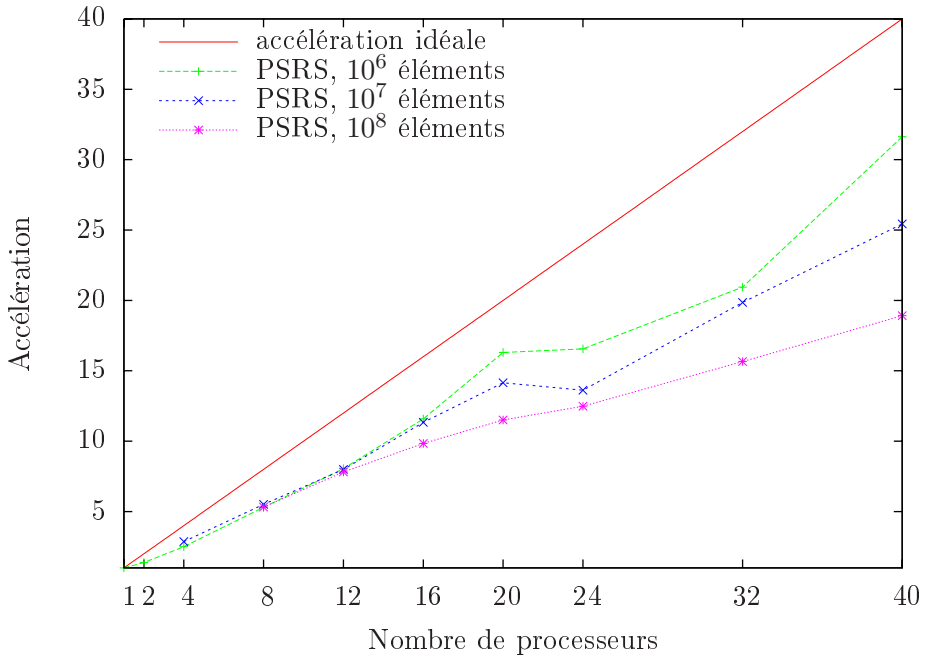


FIG. 2.4 – Résultats expérimentaux du tri par PSRS.

La figure 2.4 montre les résultats en accélération pour l'exécution de cet algorithme sur des listes de nombres à virgule flottante aléatoires. L'accélération correspond au rapport du temps d'exécution séquentiel (la fonction List.sort d'OCaml, basée sur un tri par fusion) au temps d'exécution parallèle, le cas idéal étant cleui où le temps d'exécution est divisé par le nombre de processeurs utilisés. Le programme de test a été compilé en code natif, en utilisant la version de BSML basée sur MPI. Ces résultats ont été obtenus sur la grappe du LACL, composée de 20 processeurs dual-core Intel E2180 à 2GHz munis de 2Go de mémoire, reliés par un réseau Gigabit Ethernet. Cela explique le décrochement au-dessus de 20 processeurs : en effet, au-delà, certaines machines devront héberger deux processus de calcul. La puissance de calcul pour chaque processus n'est pas modifiée, mais la mémoire, et le bus mémoire en particulier, sont partagés, or l'opération est particulièrement intensive en accès mémoire – il en résulte un décrochement visible.

Pour les nombres d'éléments les plus grands, nous n'avons pas pu obtenir de résultats sur un petit nombre de processeurs, la mémoire s'étant révélée insuffisante. L'accélération pour les listes de longueur 10^8 est ainsi calculée à partir d'une valeur extrapolée du temps de calcul séquentiel.

Afin de comparer, et d'évaluer l'utilisation de BSML – qui traite la machine comme homogène – sur une grappe de multi-cœurs, nous avons fait un test supplémentaire pour comparer l'exécution précédente avec une exécution utilisant le moins de machines possible simultanément (et donc le plus de cœurs par machine possible). Les résultats sont présentés sur la figure 2.5. On remarque que tant que chaque machine n'héberge qu'un processus de calcul, les performances sont significativement meilleures ; à partir du moment où l'une d'entre elles en héberge deux, les performances deviennent similaires, puisque le temps de calcul d'une super étape en BSML est le

maximum des temps locaux.

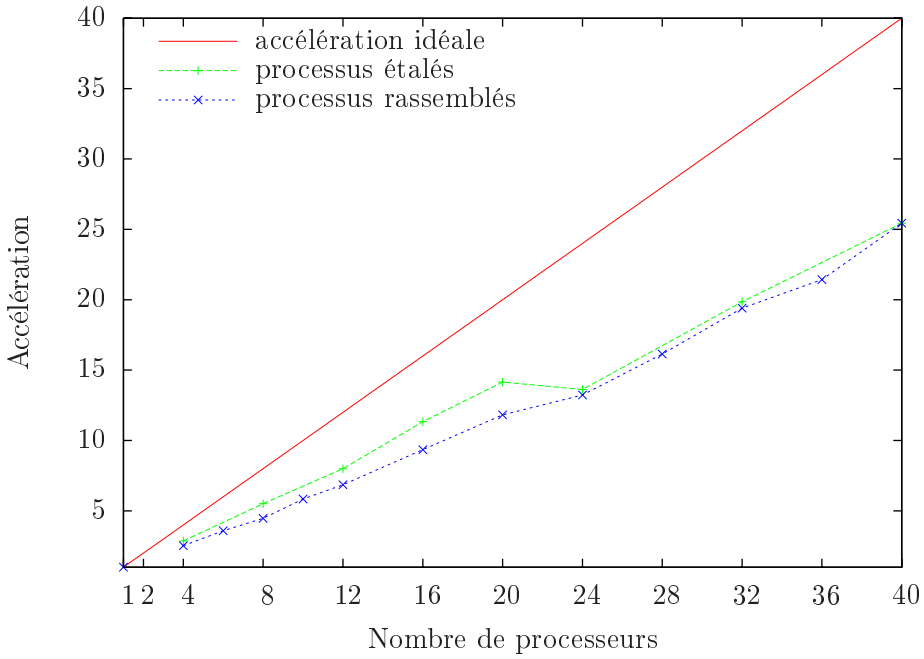


FIG. 2.5 – PSRS, 10^7 éléments. Comparaison des schémas de distribution des processus sur la grappe.

2.7 Syntaxe BSML alternative

2.7.1 Pourquoi ?

Définir le parallélisme à l'aide d'un cœur minimal d'opérations est une grande force pour la formalisation du langage. De la sorte, les définitions sont claires, et les preuves plus simples. Pouvoir intégrer ces opérations dans des fonctions de plus haut niveau permet d'obtenir des traitements parallèles complexes de façon transparente. Malgré tout, les programmes, même de haut niveau, ont toujours à gérer les valeurs répliquées et les vecteurs parallèles, et parfois l'usage des primitives ne se révèle pas des plus pratiques. En effet, toute opération à l'intérieur d'un vecteur parallèle a besoin non seulement d'un appel à une primitive, mais aussi d'une fonction ad hoc, souvent anonyme. Lorsqu'on travaille avec plusieurs vecteurs parallèles, les appels à **apply** s'imbriquent et font obstacle à la lisibilité.

Prenons un exemple simple : pour transformer une paire de vecteurs en un vecteur de paires, on écrit

```
val combine_vectors : 'a par * 'b par → ('a * 'b) par
let combine_vectors (v,w) = apply (parfun (fun v w → v,w) v) w
```

On peut alléger quelque peu ce code avec une définition intermédiaire (que l'on pourrait étendre à un nombre fixe quelconque de paramètres)

```
val parfun2 : ('a → 'b → 'c) → 'a par → 'b par → 'c par
let parfun2 f x y = apply (parfun f x) y
```

le code devient alors :

```
val combine_vectors : 'a par * 'b par → ('a * 'b) par
let combine_vectors (v,w) = parfun2 (fun v w → v,w) v w
```

le code est plus lisible, mais toujours pas entièrement satisfaisant car il nécessite la définition d'une fonction spécifique : cela implique de créer des paramètres nommés même si la fonction n'a qu'un cas d'application. Ces noms de paramètres sont alors redondants, et peuvent même provoquer des erreurs, comme

```
val combine_vectors : 'a par * 'b par → ('a * 'b) par
let combine_vectors (v,w) = parfun2 (fun w v → w,v) v w
```

qui est équivalent à la fonction ci-dessus mais plus obscur, la signification des identifiants v et w dépendant du contexte. Les noms des paramètres sont choisis par le programmeur, qui doit décider entre conserver ceux des paramètres originaux comme nous l'avons fait (mais alors, un même nom correspond à plusieurs valeurs non homogènes au sein de l'expression), ou en créer de nouveau ce qui ne simplifie pas vraiment les choses.

2.7.2 Solution

La définition de BSML est basée sur les primitives ; notre syntaxe alternative envisage les choses sous un angle complètement différent, en s'articulant autour de la notion de niveau d'exécution. On va donc donner à l'utilisateur la possibilité de déclarer si son code doit être exécuté de façon globale ou de façon locale : dans le premier cas, il s'agit de code OCaml standard, dans le second cela résultera en un vecteur parallèle.

Un des intérets de cette approche se manifeste par le fait qu'en BSML, le code local (corps de la fonction passée en argument à **mkpar**, par exemple) a accès à toutes les variables répliquées, mais pas aux autres valeurs locales situées sur le même processeur, qui sont protégées à l'intérieur de vecteurs parallèles. Y accéder nécessite de préalablement «ouvrir» ces vecteurs depuis le code global à l'aide d'un appel à **apply**. L'approche par niveaux d'exécution permet d'accéder aux composantes locales d'autres vecteurs depuis l'intérieur du code local, ce qui est beaucoup plus direct. Illustrons cette remarque :

```
val f : (int → int) → int → int → int par
let f g x y = mkpar (fun i → g (i*x/y))
```

g, x et y sont tous ici des valeurs répliquées, qui sont utilisées localement de façon directe. Si nous avions voulu faire le même calcul en utilisant un vecteur d'entiers pour x, le code serait devenu considérablement plus complexe (un paramètre supplémentaire à la fonction anonyme, et un appel à **apply**), alors que somme toute accéder localement à une valeur locale semble une chose naturelle. Si la même chose est vraie pour y, cela se complique encore : l'ouverture des vecteurs parallèles est trop encombrante du fait qu'elle a besoin d'être faite globalement.

```
val f : (int → int) → int par → int par → int par
let f g x y = apply (apply (mkpar (fun i x y → g (i*x/y))) x) y
```

syntaxe	type	description
« e »	t par (si $e : t$)	$\langle e, \ldots, e \rangle$
$this$ (dans une section locale)	int	i sur le processeur i
v (dans une section locale)	t (si $v : t$ par)	v_i sur i (si $v = \langle v_0, \ldots, v_{p-1} \rangle$)

proj et **put** sont inchangées, **mkpar** et **apply** ne sont plus nécessaires.

FIG. 2.6 – Récapitulatif de la syntaxe BSML alternative

On utilise la syntaxe ≪ ≫ pour représenter une section locale, le code entre chevrons étant exécuté localement. Cette construction retourne par conséquent un vecteur parallèle constitué des résultats des p exécutions locales. On a à l'intérieur de cette section accès normalement aux valeurs répliquées, et ≪ x ≫ est le vecteur contenant x partout. Mais on fournit également une syntaxe v disponible à l'intérieur des sections locales, et qui donne accès à la composante locale du vecteur v, sans que cela ait besoin d'être prévu depuis le code global comme dans le cas de **apply**.

On peut de la sorte écrire, par rapport à l'exemple de 2.7.1 et à l'exemple ci-dessus :

```
val combine_vectors : 'a par * 'b par -> ('a * 'b) par
let combine_vectors (v,w) = ≪ $v$, $w$ ≫

val f : (int -> int) -> int par -> int par -> int par
let f g x y = ≪ g ( i * $x$ / $y$ ) ≫
```

ce qui s'avère plus court, plus clair et moins sujet à l'erreur. On peut, enfin, accéder au pid local à l'aide de la syntaxe $this$, ce qui permet de remplacer l'usage de **mkpar**.

Les primitives synchrones ne demandent pas de modification de syntaxe, car on n'a pas besoin d'en imbriquer les appels comme c'est le cas pour **apply** : on modifie donc la façon de définir les vecteurs, mais en conservant la même syntaxe pour les communications. Cela suffit à simplifier les appels à **put** par exemple :

```
let p = put (apply (mkpar (fun sendfrom x sendto -> e(sendfrom,sendto,x))) x)
```

le code calculant les valeurs à communiquer en fonction de la source, de la destination et d'un vecteur x peut maintenant s'écrire :

```
let p = put ≪ fun sendto -> e($this$, sendto, $x$) ≫
```

cela a l'avantage supplémentaire de rendre évident le type de l'argument de **put**, qui est un vecteur de fonctions, ce qui n'était pas lisible dans l'écriture précédente.

2.7.3 Exemples

La fonction filter de la section 2.4.2 est délicate à lire parce que le lecteur a besoin de reconstituer à quoi les indices i et j font références :

```
val filter_pid : 'a par -> int -> 'a option par
```

```
let filter_pid v i =
  let filter = mkpar (fun j → if i = j then fun x → Some x else fun x → None) in
    apply filter v
```

La version en syntaxe alternative est beaucoup plus évidente :

```
val filter_pid : 'a par → int → 'a option par
let filter_pid v i = ≪ if $this$ = i then Some $v$ else None ≫
```

Un exemple plus complet est le programme de réduction parallèle de la page 20. En le réécrivant avec la nouvelle syntaxe, on obtient :

```
val reduce : int → ('a → 'a → 'a) → 'a → 'a par → 'a par
let rec reduce step op unit v =
  if step >= bsp_p then v else
    let comm = put ≪ fun j → if (j mod (2*step) = 0) && ($this$ = j + step)
                             then $v$ else unit ≫
    in let v' = ≪ if $this$ mod (2*step) = 0
                  then if $this$ + step < bsp_p
                    then op $v$ ($comm$ ($this$ + step))
                    else $v$
                  else unit ≫
    in reduce (step*2) op unit v'
```

Le programme est plus court et plus lisible, il contient beaucoup moins de fonction anonymes et de paramètres à usage unique qui encombrent le code. Les conditions sur le pid courant (if $this$...) sont évidentes, et les endroits où ont lieu les calculs sont plus visibles.

Enfin, l'exemple ci-dessous, qui est l'équivalent en syntaxe alternative du tri par échantillonnage de la page 22, donne une idée de l'aspect typique d'un algorithme parallèle implanté en BSML :

```
(* tri parallèle par échantillonnage régulier *)
val psrs : int par → 'a list par → 'a list par
let psrs lvlengths lv =
    (* étape 1: tri local *)
  let locsort = ≪ List.sort compare $lv$ ≫ in
    (* étape 2a: extraction locale de (p−1) échantillons *)
  let regsampl = ≪ extract_n bsp_p $lvlengths$ $locsort$ ≫ in
    (* étape 2b: échange des listes d'échantillons *)
  let glosampl = List.sort compare (gather_list regsampl) in
    (* étape 2c: sélection globale des pivots parmi les échantillons *)
  let pivots = extract_n bsp_p (bsp_p*(bsp_p−1)) glosampl in
    (* étape 3: partitionnement des listes locales et communication *)
  let comm = ≪ slice_p $locsort$ pivots ≫ in
  let recv = put ≪ List.nth $comm$ ≫ in
    (* étape 4: fusion locale des données reçues *)
    ≪ p_merge bsp_p (List.map $recv$ procs_list) ≫
```

2.7.4 Relation entre la syntaxe alternative et les primitives

Un des intérêts majeurs des primitives BSML est la sûreté qu'elles assurent. Heureusement, la syntaxe alternative est équivalente à l'usage des primitives, ces propriétés sont donc conservées. On peut en effet établir une transformation entre l'usage de ≪ ≫ et $this$ d'une part, et **mkpar** d'autre part, et entre l'usage de $$ et **apply** d'autre part. L'implantation de la syntaxe alternative est basée sur cette transformation, et le code compilé n'utilise que les primitives.

Cette transformation est relativement simple, et la réécriture de code utilisant les sections locales se fait de la façon suivante :

1. Le contenu d'une section locale (≪ e ≫) devient le corps d'une fonction (**fun** ... → e)
2. cette fonction est construite automatiquement de façon récursive, à partir des $$ de la section locale qui deviennent ses paramètres
3. **mkpar** est appliqué à cette fonction, dont le premier paramètre correspond à $this$
4. des appels à **apply** sont déroulés autour de l'appel à **mkpar**, correspondant à chacun des paramètres de la fonction construite (et donc à chacun des vecteurs parallèles utilisés).

Une transformation dans l'autre sens ne nous est pas utile, sauf pour montrer que les sections locales ont la même puissance que **mkpar** et **apply** combinés. Cette transformation nécessite de réduire les paramètres de **mkpar** et **apply** en valeurs de façon globale à l'aide de constructions **let in**, puis d'appliquer le paramètre de **mkpar** ainsi réduit à $this$ à l'intérieur d'une section locale, ou dans le cas de **apply**, d'appliquer ses paramètres réduits point à point à l'aide de ≪ $vf$$vx$≫ .

La transformation de sections locales vers primitives est implantée dans le langage à l'aide du préprocesseur générique d'OCaml, Camlp4 ; elle est non exclusive, et il est tout à fait possible de mêler les deux styles suivant les goûts ou les besoins. Dans notre expérience, la réduction de la taille du code au passage à la syntaxe alternative est significative.

2.8 Filtrage par motifs

Un des traits les plus utiles d'OCaml est le filtrage par motifs. Celui-ci est en particulier utilisé dans les blocs de rattrapage d'exceptions afin de sélectionner les exceptions à rattraper. Dans ce contexte, et entre autres pour compléter notre mécanisme de gestion des exceptions parallèles – la manipulation d'un ensemble d'exceptions rattrapées pouvant s'avérer fastidieuse – il est raisonnable d'étudier les possibilités de filtrage par motifs parallèle.

L'idée est de permettre d'effectuer un filtrage sur un vecteur parallèle. Ce filtrage est effectué globalement, et concerne une décision globale : en effet, rien n'empêche en BSML d'utiliser le filtrage standard d'Ocaml localement, pour prendre une décision locale, ou globalement sur une valeur répliquée. Ainsi, le mécanisme de filtrage proposé ici, bien que non trivial, peut être considéré comme du sucre syntaxique au sens qu'il n'ajoute aucune fonctionnalité liée au parallélisme : le bloc de filtrage **trypar withpar** appliqué à un vecteur parallèle v commence par effectuer des communications à l'aide de **proj** pour ramener ce vecteur à une liste répliquée, et effectue la suite de ses opérations de façon répliquée. En ce qui concerne les exceptions, l'ensemble d'exceptions levées localement a déjà été ramené au niveau global au moment du filtrage.

```
parallel_pattern  ::=  ≪ ppatt_list ≫
      ppatt_list  ::=  ppatt_elem // ppatt_list | ppatt_elem
     ppatt_elem   ::=  _ | pattern [at x] | (pattern [at x])*
```

FIG. 2.7 – Syntaxe des motifs parallèles

Le filtrage permet de rechercher des valeurs correspondant à un motif dans un vecteur (ou un ensemble d'exceptions) sans que l'ordre des éléments du vecteur (ou de l'ensemble) ait une importance. On l'introduit à l'aide d'une construction similaire au **match with** d'OCaml :

```
matchpar e withpar
 | parallel_pattern1 → e1
 | parallel_pattern2 → e2
 ...
```

Un motif parallèle parallel_pattern comporte, entre ≪ ≫ , une liste de motifs séparés par // qui seront recherchés successivement dans le vecteur ou l'ensemble. Ces motifs peuvent être soit un motif standard OCaml pattern, soit un motif standard affublé d'une étoile (pattern)* pour un nombre quelconque, strictement positif, d'occurrences de ce motif dans le vecteur ou l'ensemble, soit le motif par défaut _ pour tout contenu de vecteur. La correspondance est établie si aucun des motifs de la liste n'échoue et que tous les éléments du vecteur parallèle ou de l'ensemble sont acceptés par l'un des motifs.

Ainsi, le motif ≪ (None)* ≫ correspond à un vecteur dont tous les éléments sont None, et le motif ≪ None // _ ≫ à un vecteur comprenant au moins une fois l'élément None. Comme dans les motifs OCaml, un motif peut lier des variables : ≪ Some x // _ ≫ lie x selon la première instance du constructeur Some trouvée dans vecteur. Le résultat est un peu plus complexe pour les motifs répétés : ≪ (Some x)* // _ ≫ devrait lier la variable x à un nombre de valeurs différentes inconnu a priori. Le comportement du filtrage parallèle est d'allouer une liste pour toutes variables apparaissant sous une étoile : x sera ici la liste des variables correspondant.

Le filtrage fonctionne par élimination : en cas de motifs se recouvrant, seul le premier apparaissant dans la liste et acceptant le terme filtré est considéré. Ainsi, le motif ≪ (0)* // (x)* ≫ attribuera à x la liste des valeurs non nulles du vecteur, puisque tout élément nul sera éliminé par le premier motif.

Enfin, il est parfois utile de savoir à quel processeur attribuer quel résultat. La syntaxe pattern at p, où p est un nom de variable, lie à p le pid du processeur où le motif a correspondu. En cas de motif répété, comme pour les autres variables, une liste des correspondances est liée à p. Il est ainsi aisé, en utilisant les fonctions de la bibliothèque standard List.combine et List.split, de travailler sur une liste d'associations pid-valeur.

Exemples

Notre syntaxe de filtrage parallèle permet d'écrire une fonction équivalente à proj_list (section 2.5.1) de la façon suivante :

```
val proj_list2 : 'a par → 'a list
let proj_list2 vl = matchpar vl withpar ≪ (x)* ≫ → x
```

De façon similaire, le filtrage peut souvent remplacer l'utilisation de simple_reduce, pour le test d'une condition par exemple, en étant plus souple et plus lisible :

```
matchpar ≪ is_empty $v$ ≫ withpar
  | ≪ (true)* ≫ → ...
  | ≪ (false at p)* // _ ≫ → ...
```

plutôt que

```
if simple_reduce (&& ) ≪ is_empty $v$ ≫
  then ...
  else ...
```

La méthode par filtrage a ici l'avantage de conserver des informations sur les processeurs concernés. Elle permet de plus de rajouter des cas plus spécifiques comme «des données sur un seul processeur» (≪ false at p // (true)* ≫), ou bien «tous les processeurs ont encore des données» (≪ (false)* ≫), cas qui pourraient être utiles pour rééquilibrer la distribution des données par exemple. Ce genre de résultats avec simple_reduce auraient nécessité la définition d'un opérateur ad-hoc complexe.

L'implantation de fonctions standard telles que List.find, List.filter ou List.assoc sur des listes parallèles (obtenues, par exemple, à l'aide des fonctions décrites en 2.4.1) est également simplifiée. Ce résultat se généralise aussi bien à des fonctions sur les ensembles ou les tables d'association parallèles.

```
val assoc_par : 'a → ('a * 'b) list par → 'b
let assoc_par x vl =
  let vassoc = ≪ try Some (List.assoc x $vl$) with Not_found → None ≫
  in matchpar vassoc withpar
      | ≪ Some x // _ ≫ → x
      | ≪ (None)* ≫ → raise Not_found
```

Remarquez que l'on transforme la valeur de retour de List.assoc en type option, car un échec local (Not_found) ici est normal et ne doit pas se traduire en erreur globale. On aurait également pu écrire, en supposant préalablement définie une exception Found, une solution utilisant uniquement le rattrapage d'exceptions locales :

```
val assoc_par2 : 'a → ('a * 'b) list par → 'b
let assoc_par2 x vl =
  trypar ≪ raise (Found (List.assoc x $vl$)) ≫ withpar
    | ≪ Found y // _ ≫ → y
    | ≪ (Not_found)* ≫ → raise Not_found
```

L'expression locale ici lève soit l'exception Not_found, soit l'exception Found si un résultat a été trouvé. Les motifs parallèles permettent de faire rapidement le tri entre les deux cas, «un résultat a été trouvé» ou «aucun processeur n'a trouvé de résultat».

2.9 Superposition parallèle

Une des limitations apparentes de BSML comparé à des langages à génération dynamique de *threads* est la facilité d'implantation d'algorithmes de type diviser pour régner. En effet, là où l'on voudrait pouvoir subdiviser récursivement l'exécution en sous-parties, notre modèle de parallélisme, fidèle à la machine physique, offre seulement deux niveaux de parallélisme, global et local. Il n'y a donc aucune correspondance entre parallélisme et division du travail de calcul, et l'interaction entre les deux devient un obstacle à gérer par le programmeur, qui pourra par exemple diviser manuellement le travail en p parties, résoudre par un algorithme séquentiel localement puis effectuer une réduction parallèle. Cette solution est très lourde si l'on considère que ce type d'algorithmes est parallèle par nature ; de plus, suivant le problème, elle peut poser des problèmes d'équilibrage de charge et demander des étapes de rééquilibrage.

L'idée des auteurs dans [Lou03, Gav08], plutôt que de tenter d'introduire une récursivité spatiale dans la machine parallèle, est d'établir un nouveau découpage qui lui est orthogonal, un découpage temporel. Contrairement à un ordonnancement (*scheduling*) habituel, celui-ci se place au-dessus du parallélisme, ce qui permettra des optimisations. Cette approche permet donc de sous-diviser la machine parallèle autant que nécessaire, tout en conservant l'usage du parallélisme BSML dans chacune de ces sous-divisions, mais sans induire le coût prohibitif qu'aurait causé un nombre exponentiel de super-étapes.

Pour ce faire, une primitive **super**: $(\mathsf{unit} \to \mathsf{'a}) \to (\mathsf{unit} \to \mathsf{'b}) \to \mathsf{'a} * \mathsf{'b}$ (pour *superposition*) est ajoutée au langage. Si l'on se place dans un contexte purement fonctionnel[6], on a :

$$\mathsf{super}\ f\ g = (f(), g())$$

Où l'exécution n'est possible que dans un contexte global. Bien sûr, super ne se limite pas à une réduction de couple : tout son intérêt est de jouer sur le parallélisme BSP pour réduire le coût de la réduction combinée de f et g. Ainsi, super va entremêler le calcul de f et de g afin de *fusionner* – ou *superposer* – leurs super-étapes. Ainsi, les deux *super-threads* f et g sont exécutés l'un après l'autre localement, tant que dure la phase BSP de calculs locaux. Les phases de communication et de barrière, par contre, sont reportées jusqu'à l'arrêt de tous les super-threads, et fusionnées entre eux. Ainsi, l'exécution de n programmes BSML superposés de la sorte et effectuant respectivement S_i super-étapes donnera lieu à $\max(S_i)$ super-étapes au total au lieu de $\sum_i S_i$ s'ils avaient été combinés simplement.

[6] la définition de la superposition parallèle en présence de traits impératifs nécessite une définition plus fine de l'ordonnancement afin de préserver le déterminisme de l'exécution. Voir 5.3.3.

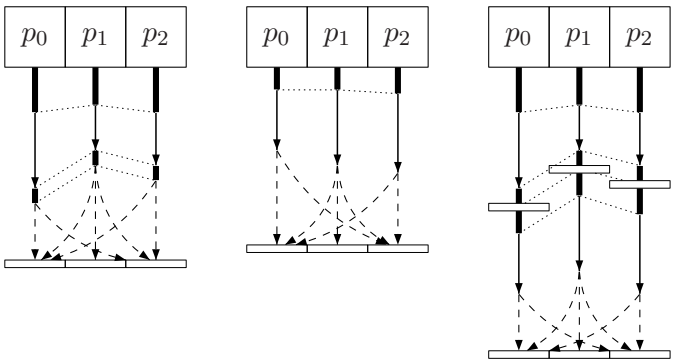


FIG. 2.8 – Deux super-étapes et leur superposition.

Chapitre 3

Sémantique de μBSML

Après avoir donné un aperçu du langage et de son utilisation dans le chapitre précédent, on s'attachera ici à lui fournir un formalisme et à prouver certaines propriétés (classiques, et liées au modèle de parallélisme) importantes le concernant. Le cœur du langage est détaillé dans un premier temps, et les extensions permettant la gestion des références et des extensions sont étudiées ensuite.

3.1 Généralités

Nous avons ici besoin d'un langage suffisamment simple pour pouvoir être étudié de façon formelle en détail, mais qui englobe toutes les fonctionnalités importantes de BSML. Comme on le fait généralement lors de la définition d'une sémantique, nous définissons ainsi un sous-ensembles de BSML que nous utiliserons comme base pour les développements suivants.

La définition de μBSML ressemble de près à celle des langages de type mini-ML que l'on peut trouver dans la littérature [DDK86] : c'est un cœur de langage de type ML purement fonctionnel que nous étendons avec les vecteurs parallèles et nos primitives, lesquelles sont définies en tant qu'opérateurs. μBSML décrit le comportement de BSML en ce qui concerne l'évaluation parallèle, mais n'intègre pas certaines fonctionnalités très utiles de ML qui alourdiraient trop la théorie. Ainsi, les enregistrements, types sommes, le filtrage par motifs, les modules sont exclus de μBSML.

Nous reparlerons de ces fonctionnalités par rapport à BSML, qui les inclut par l'intermédiaire du langage OCaml, mais nous n'établirons pas d'étude sémantique : ces traits ont déjà été étudiées en détail pour des langages non parallèles, et notre objet n'est pas de réécrire ces études. Ils rentrent dans le paradigme fonctionnel, et il nous est donc permis de les considérer comme orthogonaux à notre approche fonctionnelle du parallélisme ; certaines extensions qui nous ont paru utiles, telles que le filtrage par motifs parallèle (sec. 5.4), seront discutées plus avant dans le chapitre 5, mais elles relèvent plus du sucre syntaxique que de l'étude sémantique.

Pour les traits impératifs, la question ne se pose pas dans les mêmes termes : ils rentrent directement en conflit avec le parallélisme. Ils sont pourtant eux aussi d'une grande utilité au programmeur, et nous étudions donc en détail les références dans une version étendue de μBSML que nous noterons $\mu\text{BSML}^{\mathsf{ref}}$, ainsi que les exceptions dans $\mu\text{BSML}^{\mathsf{exn}}$.

3.2 Langage de base μBSML

3.2.1 Expressions, valeurs et opérateurs

Le langage μBSML est défini à partir des expressions et opérateurs suivants :

$$
\begin{array}{rcll}
e & ::= & x & \textit{variable} \\
 & | & c & \textit{constante} \\
 & | & op & \textit{opérateur} \\
 & | & \mathsf{fun}\, x \to e & \textit{fonction} \\
 & | & e\ e & \textit{application} \\
 & | & \mathsf{let}\, x = e\, \mathsf{in}\, e & \textit{définition} \\
 & | & \mathsf{let\ rec}\, x\ y = e\, \mathsf{in}\, e & \textit{définition récursive} \\
 & | & (e, e) & \textit{couple} \\
 & | & \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e & \textit{conditionnelle} \\
 & | & [e_i]_i^n & \textit{n-tableau} \\
 & \multicolumn{3}{l}{\textit{la suite n'est pas accessible à l'utilisateur}} \\
 & | & \langle e_i \rangle_i & \textit{vecteur parallèle}
\end{array}
$$

$$
\begin{array}{rcll}
op & ::= & \mathsf{mkpar} \mid \mathsf{apply} \mid \mathsf{proj} \mid \mathsf{put} & \textit{primitives parallèles} \\
 & | & \mathsf{fst} \mid \mathsf{snd} & \textit{opérateurs de couples} \\
 & | & \mathsf{length} \mid \mathsf{nth} & \textit{opérateurs de tableaux} \\
 & | & \mathit{fix} \mid \mathit{send} & \textit{opérateurs internes}
\end{array}
$$

On distingue ici les constantes littérales c et les opérateurs op qui ont leurs règles de réduction propres, bien qu'ils soient techniquement eux aussi des constantes. On supposera définis dans la suite, parmi les constantes au moins la valeur unité (), les booléens true et false et les entiers.

Les valeurs, qui seront les expressions acceptées comme résultat d'une réduction, sont définies de la façon suivante :

$$
\begin{array}{rcl}
v & ::= & c \\
 & | & op \\
 & | & \mathsf{fun}\, x \to e \\
 & | & (v, v) \\
 & | & [v_i]_i^n \\
 & | & \langle v_i \rangle_i
\end{array}
$$

Comme pour les vecteurs parallèles, nous notons en abrégé $[\ ]_i^n$ pour le tableau de taille n où l'indice i est lié dans les crochets à l'index (commençant à 0) de l'élément : $[i]_i^n$ est le tableau $[0, 1, \ldots, (n-1)]$.

3.2.2 Sémantique à petits pas

La définition d'une sémantique à petits pas passe par la définition d'une relation de réduction *de tête* $\rightarrowtail^{\epsilon}$ s'appliquant à des expressions d'un format spécifique, et celle d'un contexte d'éva-

luation E[·] désignant une expression avec un «trou» unique [·]. Les règles de réduction de tête ne permettant de réduire que des expressions dont les sous-termes sont des valeurs, les contextes permettent, dans le cas d'expressions imbriquées, de définir dans quelle sous-expression il est permis de réduire. E[e] désigne le contexte E[·] où [·] a été remplacé par l'expression e : E[e] est donc une expression.

$$\mathrm{E} ::= [\cdot] \mid e\ \mathrm{E} \mid \mathrm{E}\ v \mid \mathsf{let}\, x = \mathrm{E}\, \mathsf{in}\, e \mid (\mathrm{E}, e) \mid (v, \mathrm{E}) \mid \mathsf{if}\, \mathrm{E}\, \mathsf{then}\, e\, \mathsf{else}\, e \mid [v, \ldots, v, \mathrm{E}, e, \ldots, e]_n$$

Notre sémantique représente le parallélisme sous la forme de deux règles, contre une seule dans les sémantiques à petits pas habituelles : la première de ces règles réduit normalement à l'intérieur du trou d'un contexte, et la seconde procède à la réduction dans un contexte étendu par la présence d'un vecteur parallèle (et un seul, le parallélisme imbriqué n'étant pas possible dans notre langage). Cette dernière correspond ainsi à la réduction locale, et la première à la réduction globale.

$$\frac{e_1 \rightarrowtail^{\epsilon} e_2}{\mathrm{E}[e_1] \rightarrowtail \mathrm{E}[e_2]}\ \textsc{context}$$

$$\forall i \in \mathcal{P}, \frac{e_1 \rightarrowtail^{\epsilon} e_2}{\mathrm{E}_g\left[\langle e'_0 \ldots e'_{i-1}, \mathrm{E}_\ell[e_1], e'_{i+1} \ldots e'_{p-1}\rangle\right] \rightarrowtail \mathrm{E}_g\left[\langle e'_0 \ldots e'_{i-1}, \mathrm{E}_\ell[e_2], e'_{i+1} \ldots e'_{p-1}\rangle\right]}\ \textsc{local}_i$$

On a une instance de la règle LOCAL$_i$ par processeur, chacune permettant la réduction locale sur un processeur donné. E$_g$ et E$_\ell$ correspondent à des contextes quelconques, LOCAL$_i$ pouvant être ainsi vue comme une règle de contexte capable de «traverser» une (seule) structure de vecteur parallèle : E$_g$ permet de situer le vecteur parallèle à réduire dans tout le terme et E$_\ell$ réduit par contexte à l'intérieur de la i^e composante locale. Chaque étape de réduction de tête est ainsi faite sur une composante du vecteur parallèle choisie de façon non déterministe, jusqu'à ce que celui-ci ne contienne plus que des valeurs, et soit ainsi lui-même considéré comme une valeur.

Cela ne décrit pas exactement le parallélisme de façon réaliste : la réduction ne se fait que sur un processeur à la fois. Cependant, le fait qu'on puisse effectuer les étapes de réduction dans un ordre quelconque est suffisant pour rendre apparent tout problème d'indéterminisme.

La figure 3.2.2 représente l'ensemble des règles de réduction de tête du langage, celles-ci incluant les règles sémantiques correspondant aux constructions syntaxiques d'une part – sans subtilités particulières liées au parallélisme du langage –, et les δ-règles permettant la réduction des opérateurs, en particulier des primitives parallèles, d'autre part.

Notons que certains opérateurs (apply, nth) sont sous forme curryfiée : leur application partielle est considéré comme un nouvel opérateur (et donc comme une valeur) par la sémantique. Dans le cas de nth, on aura la possibilité dans la suite de lever une exception pour les paramètres k incorrects : $\mathsf{nth}\, a\, k \rightarrowtail^{\epsilon} \mathsf{raise}\, \epsilon$, avec ϵ une exception constante.

Règles de réduction syntaxique

APP
$(\mathsf{fun}\, x \to e)\ v \rightarrowtail^{\epsilon} \{v/x\}e$

LET-IN
$\mathsf{let}\, x = v\, \mathsf{in}\, e \rightarrowtail^{\epsilon} \{v/x\}e$

LET-REC
$\mathsf{let\ rec}\, x\ y = e_1\, \mathsf{in}\, e_2 \rightarrowtail^{\epsilon} \mathsf{let}\, x = \mathit{fix}(\mathsf{fun}\, x \to \mathsf{fun}\, y \to e_1)\, \mathsf{in}\, e_2$

IFTHEN
$\mathsf{if}\ \mathrm{True}\ \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \rightarrowtail^{\epsilon} e_1$

IFELSE
$\mathsf{if}\ \mathrm{False}\ \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \rightarrowtail^{\epsilon} e_2$

δ-règles

primitives parallèles

$$\begin{array}{rcl}
\mathsf{mkpar}\, v & \rightarrowtail^{\epsilon} & \langle v\ i\rangle_i \\
\mathsf{apply}\, \langle v_i^1\rangle_i\ \langle v_i^2\rangle_i & \rightarrowtail^{\epsilon} & \langle v_i^1\ v_i^2\rangle_i \\
\mathsf{proj}\, \langle v_i\rangle_i & \rightarrowtail^{\epsilon} & \mathsf{nth}\, [v_i]_i^p \\
\mathsf{put}\, \langle v_i\rangle_i & \rightarrowtail^{\epsilon} & \mathsf{apply}\, \langle \mathsf{nth}\rangle_i\ (\mathit{send}\, \langle [v_i\ 0, \ldots, v_i\ (p-1)]_p\rangle_i) \\
\mathit{send} \left\langle \left[v_j^i\right]_j^p \right\rangle_i & \rightarrowtail^{\epsilon} & \left\langle \left[v_j^i\right]_i^p \right\rangle_j
\end{array}$$

sur les paires

$$\begin{array}{rcl}
\mathsf{fst}(v_1, v_2) & \rightarrowtail^{\epsilon} & v_1 \\
\mathsf{snd}(v_1, v_2) & \rightarrowtail^{\epsilon} & v_2
\end{array}$$

sur les tableaux

$$\begin{array}{rcl}
\mathsf{length}\, [v_i]_i^n & \rightarrowtail^{\epsilon} & n \\
\mathsf{nth}\, [v_i]_i^n\ k & \rightarrowtail^{\epsilon} & v_k\ \mathit{si}\ 0 \leq k < n
\end{array}$$

sur les fonctions

$$\mathit{fix}(\mathsf{fun}\, x \to e) \rightarrowtail^{\epsilon} \{\mathit{fix}(\mathsf{fun}\, x \to e)/x\}e$$

FIG. 3.1 – Règles de réduction de tête pour μBSML

3.2.3 Exemples

Prenons pour exemple l'expression suivante, qui effectue la multi-diffusion depuis le processeur 0 en utilisant put sur un vecteur de tableaux $\langle [i,\ldots,i]^i \rangle_i$:

$$\begin{array}{lll} \mathsf{apply} & (\mathsf{mkpar}(\mathsf{fun}\, i \to \mathsf{fun}\, f \to (f\ 0))) \\ & (\mathsf{put}(\mathsf{apply} & (\mathsf{mkpar}(\quad \mathsf{fun}\, i \to \mathsf{fun}\, v \to \\ & & \qquad\qquad \mathsf{if}\ i = 0\, \mathsf{then}(\mathsf{fun}\, j \to v)\, \mathsf{else}(\mathsf{fun}\, j \to []^0))) \\ & & (\mathsf{mkpar}(\mathsf{fun}\, i \to [i,\ldots,i]^{i+1})))) \end{array}$$

Cette expression est réduite par notre sémantique de la façon suivante :

$$\begin{array}{ll} & \mathsf{apply}(\cdots)(\mathsf{put}(\mathsf{apply}(\mathsf{mkpar}(\mathsf{fun}\, i \to \mathsf{fun}\, v \to \mathsf{if}\ i = 0\, \mathsf{then}(\mathsf{fun}\, j \to v)\, \mathsf{else}(\mathsf{fun}\, j \to []^0))) \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{mkpar}(\mathsf{fun}\, i \to [i,\ldots,i]^{i+1})))) \\ \rightarrowtail & \mathsf{apply}(\cdots)(\mathsf{put}(\mathsf{apply}(\cdots)\left\langle (\mathsf{fun}\, i \to [i,\ldots,i]^{i+1})\ 0, (\mathsf{fun}\, i \to [i,\ldots,i]^{i+1})\ 1 \right\rangle)) \\ \rightarrowtail & \mathsf{apply}(\cdots)(\mathsf{put}(\mathsf{apply}(\cdots)\left\langle (\mathsf{fun}\, i \to [i,\ldots,i]^{i+1})\ 0, [1,1]^2 \right\rangle)) \\ \rightarrowtail & \mathsf{apply}(\cdots)(\mathsf{put}(\mathsf{apply}(\cdots)\left\langle [0]^1, [1,1]^2 \right\rangle)) \\ \rightarrowtail & \mathsf{apply}(\cdots)(\mathsf{put}(\mathsf{apply}\left\langle (\mathsf{fun}\, i \to \cdots)\ 0, (\mathsf{fun}\, i \to \cdots)\ 1 \right\rangle \left\langle [0]^1, [1,1]^2 \right\rangle)) \\ \rightarrowtail & \mathsf{apply}(\cdots)(\mathsf{put}(\mathsf{apply}\left\langle \mathsf{fun}\, v \to \mathsf{if}\ 0 = 0\, \mathsf{then} \cdots, \mathsf{fun}\, v \to \mathsf{if}\ 1 = 0\, \mathsf{then} \cdots \right\rangle \left\langle [0]^1, [1,1]^2 \right\rangle)) \\ \rightarrowtail & \mathsf{apply}(\cdots)(\mathsf{put}\left\langle (\mathsf{fun}\, v \to \mathsf{if}\ 0 = 0\, \mathsf{then} \cdots)\ [0]^1, (\mathsf{fun}\, v \to \mathsf{if}\ 1 = 0\, \mathsf{then} \cdots)\ [1,1]^2 \right\rangle) \\ \rightarrowtail & \mathsf{apply}(\cdots)(\mathsf{put}\left\langle \mathsf{if}\ 0 = 0\, \mathsf{then}(\mathsf{fun}\, j \to [0]^1), \mathsf{if}\ 1 = 0\, \mathsf{then} \cdots \mathsf{else}(\mathsf{fun}\, j \to []^0) \right\rangle) \\ \rightarrowtail & \mathsf{apply}(\cdots)(\mathsf{put}\left\langle (\mathsf{fun}\, j \to [0]^1), (\mathsf{fun}\, j \to []^0) \right\rangle) \\ \rightarrowtail & \mathsf{apply}(\cdots)(\mathsf{apply}\left\langle \mathsf{nth}, \mathsf{nth} \right\rangle (\mathit{send}\left\langle [(\mathsf{fun}\, j \to [0]^1)\ 0, \right. \\ & \qquad\qquad\qquad\qquad\qquad\qquad \left. (\mathsf{fun}\, j \to [0]^1)\ 1]^2, [(\mathsf{fun}\, j \to []^0)\ 0, (\mathsf{fun}\, j \to []^0)\ 1]^2 \right\rangle)) \\ \rightarrowtail & \mathsf{apply}(\cdots)(\mathsf{apply}\left\langle \mathsf{nth}, \mathsf{nth} \right\rangle (\mathit{send}\left\langle [(\mathsf{fun}\, j \to [0]^1)\ 0, \right. \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \left. (\mathsf{fun}\, j \to [0]^1)\ 1]^2, [[]^0, (\mathsf{fun}\, j \to []^0)\ 1]^2 \right\rangle)) \\ \rightarrowtail & \mathsf{apply}(\cdots)(\mathsf{apply}\left\langle \mathsf{nth}, \mathsf{nth} \right\rangle (\mathit{send}\left\langle [[0]^1, (\mathsf{fun}\, j \to [0]^1)\ 1]^2, [[]^0, (\mathsf{fun}\, j \to []^0)\ 1]^2 \right\rangle)) \\ \rightarrowtail & \mathsf{apply}(\cdots)(\mathsf{apply}\left\langle \mathsf{nth}, \mathsf{nth} \right\rangle (\mathit{send}\left\langle [[0]^1, [0]^1]]^2, [[]^0, (\mathsf{fun}\, j \to []^0)\ 1]^2 \right\rangle)) \\ \rightarrowtail & \mathsf{apply}(\cdots)(\mathsf{apply}\left\langle \mathsf{nth}, \mathsf{nth} \right\rangle (\mathit{send}\left\langle [[0]^1, [0]^1]]^2, [[]^0, []^0]^2 \right\rangle)) \\ \rightarrowtail & \mathit{Communications} \\ & \mathsf{apply}(\cdots)(\mathsf{apply}\left\langle \mathsf{nth}, \mathsf{nth} \right\rangle \left\langle [[0]^1, []^0]^2, [[]^0, [0]^1]]^2 \right\rangle) \\ \rightarrowtail & \mathsf{apply}(\cdots)\left\langle \mathsf{nth}[[0]^1, []^0]^2, \mathsf{nth}[[]^0, [0]^1]]^2 \right\rangle \\ \rightarrowtail & \mathsf{apply}\left\langle (\mathsf{fun}\, i \to \mathsf{fun}\, f \to (f\ 0))\ 0, (\mathsf{fun}\, i \to \mathsf{fun}\, f \to (f\ 0))\ 1 \right\rangle \left\langle \mathsf{nth}[[0]^1, []^0]^2, \mathsf{nth}[[]^0, [0]^1]]^2 \right\rangle \\ \rightarrowtail & \mathsf{apply}\left\langle (\mathsf{fun}\, f \to (f\ 0)), (\mathsf{fun}\, i \to \mathsf{fun}\, f \to (f\ 0))\ 1 \right\rangle \left\langle \mathsf{nth}[[0]^1, []^0]^2, \mathsf{nth}[[]^0, [0]^1]]^2 \right\rangle \\ \rightarrowtail & \mathsf{apply}\left\langle (\mathsf{fun}\, f \to (f\ 0)), (\mathsf{fun}\, f \to (f\ 0)) \right\rangle \left\langle \mathsf{nth}[[0]^1, []^0]^2, \mathsf{nth}[[]^0, [0]^1]]^2 \right\rangle \\ \rightarrowtail & \left\langle (\mathsf{nth}[[0]^1, []^0]^2\ 0), (\mathsf{nth}[[]^0, [0]^1]]^2\ 0) \right\rangle \\ \rightarrowtail & \left\langle [0]^1, (\mathsf{nth}[[]^0, [0]^1]]^2\ 0) \right\rangle \\ \rightarrowtail & \left\langle [0]^1, [0]^1 \right\rangle \end{array}$$

La multi-diffusion peut être faite de façon différente en utilisant proj :

$$\begin{array}{lll} \mathsf{proj} & (\mathsf{apply} & (\mathsf{mkpar}(\mathsf{fun}\, i \to \mathsf{if}\ i = 0\, \mathsf{then}(\mathsf{fun}\, v \to v)\, \mathsf{else}(\mathsf{fun}\, v \to []^0))) \\ & & (\mathsf{mkpar}(\mathsf{fun}\, i \to [i,\ldots,i]^{i+1}))) \\ & 0 & \end{array}$$

Ce qui se réduit par :

$$
\begin{array}{ll}
 & \mathsf{proj}(\mathsf{apply}(\mathsf{mkpar}(\mathsf{fun}\, i \to \mathsf{if}\, i = 0\, \mathsf{then}(\mathsf{fun}\, v \to v)\, \mathsf{else}(\mathsf{fun}\, v \to []^0)))\\
 & \qquad\qquad (\mathsf{mkpar}(\mathsf{fun}\, i \to [i, \ldots, i]^{i+1})))0\\
\rightarrowtail & (\mathsf{proj}(\mathsf{apply}(\cdots)(\mathsf{mkpar}(\mathsf{fun}\, i \to [i, \ldots, i]^{i+1})))\ 0\\
\rightarrowtail & (\mathsf{proj}(\mathsf{apply}(\cdots)(\langle(\mathsf{fun}\, i \to [i, \ldots, i]^{i+1})\ 0, (\mathsf{fun}\, i \to [i, \ldots, i]^{i+1})\ 1\rangle)))\ 0\\
\rightarrowtail & (\mathsf{proj}(\mathsf{apply}(\cdots)(\langle [0]^1, (\mathsf{fun}\, i \to [i, \ldots, i]^{i+1})\ 1\rangle)))\ 0\\
\rightarrowtail & (\mathsf{proj}(\mathsf{apply}(\cdots)(\langle [0]^1, [1,1]^2\rangle)))\ 0\\
\rightarrowtail & (\mathsf{proj}(\mathsf{apply}(\cdots)(\langle [0]^1, [1,1]^2\rangle)))\ 0\\
\rightarrowtail & (\mathsf{proj}(\mathsf{apply}\, \langle(\mathsf{fun}\, i \to \cdots)\ 0, (\mathsf{fun}\, i \to \cdots)\ 1)\rangle\, \langle [0]^1, [1,1]^2\rangle))\ 0\\
\rightarrowtail & (\mathsf{proj}(\mathsf{apply}\, \langle \mathsf{if}\, 0 = 0\, \mathsf{then}(\mathsf{fun}\, v \to v)\, \mathsf{else}(\mathsf{fun}\, v \to []^0, (\mathsf{fun}\, i \to \cdots)\ 1)\rangle\, \langle [0]^1, [1,1]^2\rangle)))\ 0\\
\rightarrowtail & (\mathsf{proj}(\mathsf{apply}\, \langle \mathsf{fun}\, v \to v, (\mathsf{fun}\, i \to \cdots)\ 1)\rangle\, \langle [0]^1, [1,1]^2\rangle)))\ 0\\
\rightarrowtail & (\mathsf{proj}(\mathsf{apply}\, \langle \mathsf{fun}\, v \to v, \mathsf{if}\, 1 = 0\, \mathsf{then}(\mathsf{fun}\, v \to v)\, \mathsf{else}(\mathsf{fun}\, v \to []^0\rangle\, \langle [0]^1, [1,1]^2\rangle)))\ 0\\
\rightarrowtail & (\mathsf{proj}(\mathsf{apply}\, \langle \mathsf{fun}\, v \to v, \mathsf{fun}\, v \to []^0\rangle\, \langle [0]^1, [1,1]^2\rangle)))\ 0\\
\rightarrowtail & (\mathsf{proj}\, \langle (\mathsf{fun}\, v \to v)[0]^1\ , (\mathsf{fun}\, v \to []^0)[1,1]^2\rangle)\ 0\\
\rightarrowtail & (\mathsf{proj}\, \langle [0]^1, (\mathsf{fun}\, v \to []^0)[1,1]^2\rangle)\ 0\\
\rightarrowtail & (\mathsf{proj}\, \langle [0]^1, []^0\rangle)\ 0\\
\rightarrowtail & \mathsf{nth}[[0]^1, []^0]^2\ 0\\
\rightarrowtail & [0]^1
\end{array}
$$

3.2.4 Confluence

La confluence est une propriété permettant d'exprimer qu'une sémantique est bien formée : elle indique informellement que la réduction finale d'une expression ne dépend pas du chemin choisi. Une sémantique déterministe, où jamais plus d'une règle de réduction peut s'appliquer à une expression donnée, est confluente de manière évidente ; ce n'est pas le cas de notre sémantique pour la réduction dans les vecteurs parallèles (le choix des règles LOCAL$_i$ est non déterministe). Prouver la confluence montre ainsi que μBSML est compatible avec le fait que certaines réductions puissent être faites «en parallèle».

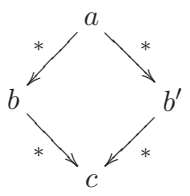
Soit $\mathcal{R}$ une relation. On note $\mathcal{R}^*$ la fermeture réflexive et transitive de $\mathcal{R}$.

Définition 3.2.1 (*Confluence*)

$\mathcal{R}$ est confluente si, et seulement si pour tous a, b, b' tels que $a\,\mathcal{R}^*\,b$ et $a\,\mathcal{R}^*\,b'$, il existe c tel que

$$b\,\mathcal{R}^*\,c \text{ et } b'\,\mathcal{R}^*\,c$$

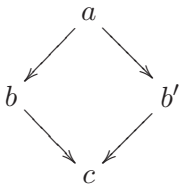
Cela se représente schématiquement sous la forme de la «propriété du parallélogramme» :



Définition 3.2.2 (*Confluence forte*)

$\mathcal{R}$ est fortement confluente si, et seulement si pour tous a, b, b' tels que $a \mathcal{R} b$ et $a \mathcal{R} b'$, il existe c tel que

$$(b = c \text{ ou } b \mathcal{R} c) \text{ et } (b' \mathcal{R} c \text{ ou } b' = c)$$



Lemme 3.2.3

Toute relation fortement confluente est confluente.

Propriété 3.2.4 (*Confluence forte de $\rightarrowtail$ dans μBSML*)

$\rightarrowtail$ est fortement confluente.

Démonstration (confluence de $\rightarrowtail$ pour μBSML) : On montrera que si $e \rightarrowtail e^1$ et $e \rightarrowtail e^2$, alors soit l'étape de réduction est déterministe et $e^1 = e^2$, soit $e^1 \rightarrowtail e^2$ (ou inversement), soit enfin il existe e^3 tel que $e^1 \rightarrowtail e^3$ et $e^2 \rightarrowtail e^3$

Par induction structurelle sur l'arbre de e :

- e ne peut être une valeur, sans quoi aucune règle de réduction ne s'appliquerait.
- Toutes les valeurs apparaissant dans les membres de gauche de nos règles de réduction de tête correspondent, dans la définition du contexte, à un sous-contexte. Ainsi, soit la réduction directe est possible, si on a des valeurs, soit on peut définir un sous-contexte. Il n'est pas possible, par ailleurs, de définir un sous-contexte là où une règle de réduction de tête est applicable (pour la règle let rec par exemple), par structure des définitions. Le choix d'une réduction de tête, par l'intermédiaire du contexte vide $[\cdot]$, ou d'une réduction à l'intérieur d'un autre contexte, est donc déterministe.
- La réduction de tête est déterministe : nos règles et δ-règles s'appliquent à des termes de structure disjointes.
- La réduction par contexte se fait soit hors de tout vecteur parallèle, directement par la règle CONTEXT – le contexte est dans ce cas choisi de façon déterministe – soit par l'intermédiaire d'une des règles LOCAL_i si le contexte se heurte à un vecteur parallèle.
- Dans ce dernier cas, le choix de la composante dans laquelle est faite la réduction est non déterministe. Posons $e = \text{E}_g \langle \text{E}_i[e_i] \rangle_i$, et i, j tels que $e^1 = \text{LOCAL}_i(e)$ et $e^2 = \text{LOCAL}_j(e)$. Si $i = j$, le résultat résulte du même raisonnement que ci-dessus appliqué localement ; dans le cas contraire, la réduction sur les processeurs i et j étant indépendante, on peut poser $e^3 = \text{LOCAL}_i(e^2) = \text{LOCAL}_j(e^1)$, ce qui nous donne le résultat. ■

3.3 μBSML avec références : μBSML$^{\text{ref}}$

3.3.1 Problème et description informelle

Les traits impératifs les plus utilisés dans OCaml sont les types physiquement modifiables, tels que les références. Ils offrent une souplesse très appréciée des programmeurs, de meilleures performances dans certains cas (par exemple, pour traitements sur les tableaux) et sont largement utilisés. Cependant, ils peuvent poser des problèmes quant à la cohérence de notre modèle en brisant la barrière entre valeurs locales et globales.

Les tableaux, les chaînes de caractères, les enregistrements à champs modifiables («mutables») et les références sont dans ce cas : ils correspondent tous à une donnée – ou un ensemble de données – que l'on peut modifier directement en mémoire à l'aide d'un opérateur :

```
(* tableaux *)
arr.(n) ← x;
(* chaînes *)
str.[n] ← x;
(* champs mutables *)
record.field ← x;
(* références *)
r := x
```

Du point de vue de notre problématique, tous ces cas sont équivalents : les champs modifiables peuvent être remplacés par des champs contenant des références (à l'inverse, les références sont d'ailleurs considérées comme des enregistrements en OCaml). Les tableaux modifiables pourraient être remplacés par des tableaux fixes de références : bien que l'implantation et les performances diffèrent grandement, on obtiendrait une sémantique similaire. Quant aux chaînes, elles peuvent être vues comme des tableaux de caractères.

Nous allons donc ici nous concentrer sur l'étude des références, les autres cas en découlant simplement. Cette section décrit en détail la sémantique des références, qui est similaire à celle d'OCaml dans les cas local et répliqué. Cette sémantique nous servira de point de départ pour étudier les interactions entre μBSML et les références, et détailler les cas problématiques ; nous définissons également le comportement des références vis-à-vis des communications entre processeurs.

Pour assurer la confluence de μBSML avec références, un système de types est nécessaire. Cette partie est expliquée en 3.3.3 et développée en 4.3

3.3.2 Sémantique

Extensions du langage

Afin de traiter les références, on étend le langage de base avec les constructions suivantes :

$$
\begin{array}{rcll}
e & ::= & \cdots & \\
 & | & l & \textit{adresse mémoire}
\end{array}
\qquad
\begin{array}{rcll}
op & ::= & \cdots & \\
 & | & \mathsf{ref} & \textit{référence} \\
 & | & \mathsf{get} & \textit{accès} \\
 & | & \mathsf{set} & \textit{mise à jour}
\end{array}
$$

$$
\begin{array}{rcll}
v & ::= & \cdots & \\
 & | & l & \textit{adresse mémoire}
\end{array}
$$

l est absent du langage programmeur et ne peut donc être manipulé que par l'intermédiaire des opérateurs ref, qui crée une nouvelle référence vers une valeur donnée (ce qui assure qu'une référence pointe toujours sur une valeur), get qui permet d'accéder à cette valeur et set qui permet de la modifier.

Il est nécessaire de définir un *état mémoire* dans la sémantique, afin de mémoriser les valeurs auxquelles correspondent les adresses mémoire : un état mémoire s est une table associative entre adresses mémoire et valeurs. Suivant les besoins, nous considérons dans la suite s comme une fonction ou comme un ensemble de paires (l, v).

On écrira $(s)e$ pour l'expression e dans l'état mémoire s. Cette écriture est toujours supposée cohérente, c'est-à-dire que s est défini pour toute adresse mémoire apparaissant dans e. Nous utiliserons dans la suite les opérations suivantes sur les tables associatives, et en particulier sur s :

- $s \cdot \{l \mapsto v\}$ est la fonction équivalente à s pour tout adresse différente de l et égale à v sur l. En termes d'ensemble de couples, tout éventuel couple de la forme $(l, *)$ est supprimé de s et un couple (l, v) y est ajouté.
- $s(l)$ est l'écriture fonctionnelle usuelle, renvoyant la valeur associée à l dans s.
- fresh(s) renvoie une adresse mémoire l avec la garantie que celle-ci n'apparaît pas dans s. On supposera qu'une infinité d'adresses mémoire est disponible et que fresh ne peut échouer.

Règles de réduction

Les règles de réduction de tête de μBSML sont toutes implicitement étendues pour réduire les expressions avec état mémoire, celui-ci étant inchangé : $e_1 \rightarrowtail^{\epsilon} e_2$ devient $(s)e_1 \rightarrowtail^{\epsilon} (s)e_2$. Les modifications de s ne sont ainsi possibles qu'en utilisant les nouveaux opérateurs. Les règles de contexte et de réduction locale deviennent :

$$\frac{(s_1)e_1 \rightarrowtail^{\epsilon} (s_2)e_2}{(s_1)\mathrm{E}[e_1] \rightarrowtail (s_2)\mathrm{E}[e_2]}\ \text{CONTEXT}$$

$$\forall i \in \mathcal{P},\ \frac{(s_1)e_1 \rightarrowtail^{\epsilon} (s_2)e_2}{(s_1)\mathrm{E}_g\left[\langle e'_0 \dots e'_{i-1}, \mathrm{E}_\ell[e_1], e'_{i+1} \dots e'_{p-1}\rangle\right] \rightarrowtail (s_2)\mathrm{E}_g\left[\langle e'_0 \dots e'_{i-1}, \mathrm{E}_\ell[e_2], e'_{i+1} \dots e'_{p-1}\rangle\right]}\ \text{LOCAL}_i$$

C'est une simplification extrême par rapport à la machine parallèle BSP, qui a p paires processeur-mémoire et devrait donc avoir p états mémoire, alors qu'on en suppose ici un unique. Néanmoins, on assure que le contexte reste valide au cours de l'exécution, et les arguments suivants appuient la validité de cette représentation :

- lorsqu'on considère les expressions répliquées, on utilise des références répliquées qui ont la même valeur partout : nous avons besoin d'un «état mémoire global» pour représenter cela, à moins d'un moyen ardu de représenter la cohérence des valeurs pointées par p adresses mémoire locales. Cet argument certes n'exlut pas des états mémoire locaux.
- les processeurs peuvent individuellement accéder à la mémoire globale, mais en lecture seulement, l'écriture n'étant possible que par du code répliqué : notre modèle ne permet pas à un processeur d'écrire dans la mémoire d'un autre. La distinction entre états mémoires locaux et globaux par rapport aux opérations sur les références serait complexe et délicate.
- une référence créée localement est une adresse mémoire fraîche ; il nous est facile de supposer qu'elle l'est vis-à-vis de la machine parallèle dans son ensemble (par découpage en p parties de notre espace d'adressage). La structure de μBSML nous garantit qu'elle est inaccessible aux autres processeurs, comme toute autre valeur locale : ainsi, bien que la référence locale soit stockée dans l'état mémoire global, on a la garantie qu'aucun autre processeur ne dispose de la clef pour y accéder[7] .

On a ainsi l'assurance que la séparation des états mémoires dans notre sémantique n'est pas nécessaire à la cohérence du modèle : on suppose que l'état mémoire global contient des adresses pointant soit sur un *espace d'adressage global* (virtuel, physiquement ces adresses correspondent à un ensemble de p adresses locales), soit sur un *espace d'adressage local* sur un processeur donné. Le système de types (voir section 4.3) rendra explicite cette distinction.

Les opérateurs dédiés à la manipulation de l'état mémoire obéissent aux δ-règles suivantes :

$$(s)\,\mathsf{ref}\,v \rightarrowtail^{\epsilon} (s \cdot \{l \mapsto v\})\,l \quad \text{with } l = \text{fresh}(s)$$

$$(s)\,\mathsf{get}\,l \rightarrowtail^{\epsilon} (s)s(l)$$

$$(s)\,\mathsf{set}\,l\,v \rightarrowtail^{\epsilon} (s \cdot \{l \mapsto v\})()$$

où () est une constante du langage. Cette définition garantit qu'il ne peut y avoir qu'une seule valeur pour une adresse mémoire à un instant donné, et que toute adresse mémoire l utilisée dans un programme est dans s puisqu'elle a nécessairement été préalablement obtenue à l'aide de ref.

Conflits entre parallélisme et références

Aucune difficulté particulière n'est causée par le fait d'utiliser les références localement ou de manière répliquée ; cependant, la communication des références à l'aide de proj ou put pose de réels problèmes liés à la réalité physique de la la machine parallèle. Ainsi, une adresse mémoire dans l'espace d'adressage de l'un des processeurs n'a aucun sens pour les autres. Il reste possible de communiquer la valeur associée plutôt que l'adresse mémoire elle-même, mais quel doit être alors le comportement de mises à jour ultérieures sur cette référence ?

[7] on suppose ici qu'il n'y a pas de communications de références ; le cas sera traité spécifiquement dans le suite

Comme nous l'avons déjà mentionné, ce problème peut être résolu si l'on considère un modèle de parallélisme à mémoire partagée, éventuellement implanté à l'aide de communications implicites [HM05]. On se retrouve alors dans un contexte trop éloigné de celui de BSML, où les problèmes de concurrence et d'indéterminisme refont surface, sans parler de la prédiction de performance.

Il nous est néanmoins vital de tolérer la communication de références, la programmation courante faisant un usage fréquent d'enregistrements à champs *mutables* ou de tableaux, dont les cellules ont un comportement similaire à celui des références. Il serait peu réaliste d'exiger du programmeur une conversion manuelle de ces structures en une version purement fonctionnelle avant de procéder à toute communication. Ainsi, le langage effectue une copie automatique des références lors des communications. Cela ajoute un coût supplémentaire aux primitives de communication, mais cette solution est la plus satisfaisante par rapport à la facilité d'utilisation du langage et sa sûreté théorique. Le programmeur, malgré tout, doit être conscient que la valeur retournée par proj ou put contient de nouvelles références qui pointent vers une copie des valeurs : l'élément retourné est identique à l'élément d'origine, mais l'égalité physique n'est pas préservée. On a néanmoins la préservation d'une égalité structurelle, définie comme l'égalité des valeurs dans leurs états mémoires respectifs (et de celles dont elle dépendent par l'intermédiaire de ces états mémoire) modulo le choix des adresses mémoire.

Dans l'implantation du langage, ce comportement correspond à l'utilisation des fonctions de sérialisation du module Marshal d'Ocaml avec l'option Closures, que nous utilisons pour les communications. La communication de fonctions par ce module souffre de certaines limitations, qui corroborent les problèmes que nous allons découvrir dans notre théorie. En effet, elle suppose d'une part que tous les processeurs utilisent exactement la même version de l'exécutable (et donc la même architecture) – les fonctions étant transmises en tant qu'adresses dans le code, d'autre part son comportement peut dépendre de la cible de compilation (le programme Marshal.to_string (**fun** _ → (**ref** 0):=1) [Marshal.Closures] renvoie une erreur dans le mode interactif, par exemple, mais fonctionne en code natif bien qu'avec un comportement différent de celui du code-octet).

Intuitivement, si r_j est une référence, proj $\langle r_j \rangle_j$ i renvoie l'équivalent de ref(get r_i) au lieu de r_i. On s'assure néanmoins que plusieurs instances d'une même référence dans la valeur communiquée ne conduisent pas à des copies multiples dans le résultat.

Voici le mécanisme employé : la communication d'une expression est précédée d'une opération locale de *déréférenciation* (*deref*), et suivie d'une opération de *reréférenciation* (*reref*). Cette dernière a lieu dans un contexte local dans le cas de put et global dans le cas de proj. La valeur retournée par *deref* ne contient aucune référence et il est donc sûr de la communiquer dans le cadre de la sémantique que nous avons définie. On fait cependant en sorte qu'elle contienne toutes les valeurs de l'état mémoire nécessaires pour reconstituer la valeur d'origine. *reref*, à l'aide de ces valeurs, crée de nouvelles références qui prendront la place de celles que contenait l'exception d'origine. Ainsi, la composition *reref* ∘ *deref* recopie les références, mais préserve type et égalité structurelle.

La *déréférenciation* suit l'algorithme suivant :

pour toute valeur v dans l'état s, on commence par extraire l'ensemble $E(v)$ des adresses

mémoire contenues dans v :

$$E(l) ::= l \cup E(s(l))$$

$$E(x) ::= \emptyset \qquad E(c) ::= \emptyset \qquad E(op) ::= \emptyset$$

Le calcul de E se propage récursivement à tous les sous-termes de l'expression dans tous les autres cas. *deref* est ensuite défini récursivement sur l'ensemble des adresses mémoire par :

$$deref_E(\emptyset, s, v) ::= v$$

$$deref_E(\{l\} \cup E, s, v) ::= deref_E(E, \{x/l\}s, (\mathsf{fun}\, x \to \{x/l\}v, s(l)))$$

avec x aussi fraîche que nécessaire, et la substitution $\{x/l\}$ définie comme le remplacement complet de l par x partout où elle apparaît dans v, ou dans le membre de droite des couples contenus dans s. On abstrait ainsi les adresses mémoire apparaissant dans l'expression en construisant des fonctions, et on joint la valeur de l'état mémoire correspondante en construisant un couple.

On peut ensuite définir $deref(s, v)$ comme $deref_E(E(v), s, v)$; ce résultat devra être communiqué accompagné du cardinal n de $E(v)$.

La *reréférenciation* crée de nouvelles références par application des fonctions générées par *deref* :

$$reref_0(f, a) ::= f\ (\mathsf{ref}\, a)$$

$reref_0$ devant être appliqué une fois par adresse mémoire transformée, on définit $reref = reref_0^n$

Propriété 3.3.1 (*Déplacement mémoire*)

Pour tous v, v' tels que $v' = \mathrm{reref} \circ \mathrm{deref}\, v$, il existe une bijection φ_l sur les adresses mémoire qui, étendue aux valeurs de la façon usuelle (sans capture de variables), est telle que $v' = \varphi_l(v)$.

La preuve est directe par récursivité sur le nombre d'adresses mémoire n apparaissant dans v.

3.3.3 Conditions supplémentaires pour la confluence et preuve

Nous supposerons dans la suite un terme $(s)e$ défini modulo renommage bijectif des adresses mémoire dans s et e, ce qui nous permet de considérer l'opération fresh comme déterministe.

La sémantique proposée pour $\mu\mathrm{BSML}^{\mathsf{ref}}$ n'est pas confluente dans le cas général : l'état mémoire global est partagé par tous les processeurs, qui peuvent y écrire de manière individuelle. Prenons un exemple :

$$\mathsf{let}\, r = \mathsf{ref}\, 0\, \mathsf{in}\, \mathsf{mkpar}(\mathsf{fun}\, i \to \mathsf{set}\, r\, i)$$

Après quelques étapes de réduction, cela devient :

$$(\{l \mapsto 0\})\,\langle \mathsf{set}\, l\, i\rangle_i$$

Là, le choix non déterministe de l'ordre de réduction des composantes de $\langle \mathsf{set}\, l\, i\rangle_i$ conduit à des valeurs différentes pour l dans l'état mémoire à la fin de la réduction, et le résultat de l'opération $\mathsf{get}\, r$ pourrait être une valeur quelconque de 0 à $p-1$.

On remarque que dans ce cas, la sémantique ne décrit plus un comportement réaliste pour la machine parallèle : celle-ci n'a pas réellement de mémoire partagée, et les accès concurrents à la référence répliquée nous conduiraient dans un état incohérent (ce qui est pire). Le programme BSML équivalent au programme μBSML ci-dessus :

```
let r = ref 0 in ≪ r:=$this$ ≫
```

conduit à une référence r qui n'est plus répliquée puisqu'elle contient des valeurs différentes sur les différents processeurs. Bien que l'évaluation sémantique diffère de l'évaluation BSML, on a dans les deux cas un problème qu'on devra donc éviter. Nous supposerons donc la propriété suivante, qui décrit un accès correct à l'état mémoire :

Propriété 3.3.2 (*Séparation mémoire*)

Si l'opérateur set *est appliqué à l'adresse mémoire l dans une composante d'un vecteur parallèle, alors l n'apparaît que localement, sur le même processeur, dans tout le programme.*

Cette propriété nous garantit que le problème ci-dessus ne se produit pas, et nous limite aux cas où la sémantique est fidèle à l'évaluation en mémoire distribuée. Elle nous sera garantie par le système de types, qui est défini en 4.3. On est en mesure de prouver la confluence de μBSML$^{\mathsf{ref}}$pour les programmes vérifiant cette propriété, ce qui est le cas de tous les programmes bien typés.

Propriété 3.3.3 (*confluence de μBSML$^{\mathsf{ref}}$*)

La sémantique de μBSML$^{\mathsf{ref}}$est confluente pour les programmes respectant la propriété de séparation mémoire.

Démonstration (confluence de μBSML$^{\mathsf{ref}}$) : On suppose $(s)e \rightarrowtail (s^1)e^1$ et $(s)e \rightarrowtail (s^2)e^2$, et on reprend la preuve de confluence de μBSML (3.2.4), en ne s'attachant qu'aux cas qui diffèrent :

- ref utilise une variable fraîche choisie de façon non déterministe, mais notre critère d'équivalence par renommage nous permet de le considérer comme bijectif.
- Réduction dans un vecteur parallèle : on suppose $e = \mathrm{E}_g\,\langle e_i\rangle_i$; une des règles LOCAL_i s'applique, et on pose $(s^1)e^1 = \mathrm{LOCAL}_i((s)e)$ et $(s^2)e^2 = \mathrm{LOCAL}_j((s)e)$. Si $i = j$, on a $(s^1)e^1 = (s^2)e^2$.
  Si ce n'est pas le cas, on définit

$$(s^2\prime)e^2\prime = \mathrm{LOCAL}_i((s^2)e^2) \qquad (s^1\prime)e^1\prime = \mathrm{LOCAL}_j((s^1)e^1)$$

  Il est possible que s^2 soit différent de s pour deux raisons, soit parce qu'une paire y a été ajoutée ppar usage de ref, soit parce qu'une mise à jour y a été faite par l'intermédiaire de set. Dans le premier cas, cette adresse est fraîche, dans le second,

la propriété de séparation mémoire nous assure que l'adresse l dont la liaison a été changée dans s est absente de la i^e composante du vecteur (set a été utilisé sur la j^e composante, et on a $i \neq j$). Dans les deux cas, la réduction dans la i^e composante n'a pas accès aux adresses concernées par ces modifications, et par conséquent ces dernières n'ont pas de conséquence sur cette réduction. Par le même raisonnement, on sait que l'évaluation sur la i^e composante n'a aucun impact sur l'évaluation dans la j^e composante, et on en conclut que $e^2\prime = e^1\prime$. Les modifications dans s concernant des adresses différentes, celles-ci sont orthogonales également et $s^2\prime = s^1\prime$.
On a donc $(s^2\prime)e^2\prime = (s^1\prime)e^1\prime$, et la confluence forte de la sémantique. ■

3.4 μBSML avec exceptions : μBSML$^{\mathsf{exn}}$

3.4.1 Problème et description informelle

Exceptions en OCaml

OCaml comporte un système de gestion des exceptions performant et efficace en pratique. D'autres langages se fondent sur les monades [BHM00, Wad92] pour obtenir le même type de fonctionnalités, mais ceux-ci sont plus complexes à utiliser, moins efficaces, et ne s'avèrent pas nécessaires dans un langage fonctionnel impur à évaluation stricte tel que OCaml. Les exceptions sont utilisées aussi bien pour traiter les erreurs d'exécution (échec d'une opération telle qu'ouverture de fichier ou division, dépassement de pile ou de mémoire) que les traitements exceptionnels souhaités : elles sont en effet un moyen de sortir directement d'un niveau quelconque d'imbrication de boucles ou d'appels pour récupérer un résultat, ce qui entraîne souvent un code plus lourd par d'autres moyens.

La levée d'une exception interrompt les calculs en cours, et remonte dans la pile d'exécution ; si cette exception n'est pas rattrapée, l'exception remonte jusqu'à la racine du programme qui se termine par une erreur. Il est possible de rattraper l'exception pour déclencher un traitement de l'erreur rencontrée, ou gérer le cas exceptionnel si l'exception était attendue. On déclare une exception en OCaml par

```
exception Exc
```

où Exc est le constructeur de l'exception déclarée. La récupération de cette exception s'échappant d'un bloc de code [*code*] est effectuée de la façon suivante :

```
try [code] with Exc → [traitement Exc]
```

Le **with** ici ouvre un cas de filtrage par motifs. Le mécanisme permet également de définir des exceptions paramétrées, ressemblant aux types somme :

```
exception Excb of int

try [code] with Excb n → [traitement Excb en fonction de n]
```

de cette façon, il est possible d'extraire soit des informations sur l'erreur, soit le résultat correspondant à un cas exceptionnel pour lequel l'exception aurait été levée volontairement par

l'opérateur raise (Excb x). Comme pour le filtrage par motifs standard d'OCaml, il est possible d'énumérer différents motifs à récupérer sur les erreurs. Notons cependant que si la syntaxe d'Objective Caml offre la souplesse du filtrage pour récupérer les exceptions, la gestion des exceptions en elle-même en reste indépendante : il est toujours possible d'écrire, si on ne veut pas utiliser cette facilité,

```
try [code] with ex → match ex with
  | Excb n → [traitement Excb en fonction de n]
  | e → raise e
```

ce qui est sémantiquement équivalent au code ci-dessus, ou même, de façon plus générique et sans utiliser le filtrage :

```
try [code] with
  ex → if [conditions sur ex] then [traitement ex] else raise ex
```

il est donc possible de re-lever l'exception si elle ne correspond pas aux critères, en liant simplement celle-ci à un identifiant au lieu d'utiliser le filtrage. Ce dernier point permet de justifier l'absence de filtrage dans notre sémantique sans remettre en cause les mécanismes de traitement des exceptions proposés.

On ne s'intéressera pas ici à l'implantation à proprement parler des exceptions dans OCaml, celle-ci utilisant du code de très bas niveau : notre mécanisme est entièrement implanté comme sur-couche du mécanisme existant et en reste donc indépendant.

Exceptions en BSML

Il convient d'examiner tous les cas d'interaction pouvant se présenter entre les exceptions et notre modèle de parallélisme : des exceptions peuvent être levées globalement où à l'intérieur d'une (ou plusieurs) composantes d'un vecteur parallèle. De même, les blocs de rattrapage d'exception **try with** peuvent être présents localement ou globalement. Intuitivement, nos langages local et global étant sensiblement équivalents à OCaml, la levée et le rattrapage d'exceptions à l'intérieur de chacun d'eux ne posera pas de problème, mais ceux-ci peuvent se poser à l'interface.

En effet, examinons les différents cas :

- Exception levée globalement (exception répliquée) : raise Exc utilisé dans du code répliqué. Dans ce cas, tous les processeurs suivent la même exécution, que l'exception Exc soit rattrapée ou conduise à une erreur du programme. Une exception globale ne pose pas de problème vis-à-vis de notre modèle.
- Exception levée localement (exception locale) : il faut considérer la section locale où l'exception est levée, c'est-à-dire l'expression réduite dans la composante du vecteur parallèle. Si cette expression lève une exception et la rattrape, cela n'a pas de conséquence visible à l'extérieur de la composante locale concernée. Par exemple, le programme

  ```
  val f : int par → int par
  let f v = ≪ try $v$ / $this$ with Division_by_zero → 0 ≫
  ```

  ne nous pose pas de problème, et la fonction f renvoie un vecteur parallèle sans que l'utilisation d'exceptions soit visible.

– Exception levée localement, non rattrapée : un problème se pose en revanche si une exception s'échappe d'un vecteur parallèle. Il est en effet plus délicat de donner un sens à une exception locale au niveau global, de façon cohérente pour tous les proceseurs.

```
val f : int par → int par
let f v = ≪ $v$ / $this$ ≫
```

lève une exception sur le processeur 0, qui pourrait compromettre la cohérence de notre modèle.

Il y a donc un cas problématique qu'il nous faut traiter. Examinons ce qui se passe dans ce dernier exemple, sans dispositions particulières de traitement des exceptions. Tous les processeurs de pid non nul vont continuer de fonctionner comme une machine parallèle, n'ayant pas d'information sur l'exception levée par le processeur 0. Le processeur 0, pour sa part, se comporte comme si l'exception avait été levée globalement, ne faisant pas cette distinction. Il remonte donc dans sa pile d'exécution jusqu'à trouver un éventuel bloc **try with** (nécessairement global, le parallélisme emboîté étant interdit) qui rattrapera son exception. À ce stade, il aura selon toute probabilité divergé du chemin d'exécution répliquée des autres processeurs, s'il ne s'est pas terminé prématurément.

Cela casse la cohérence répliquée, désynchronise les super-étapes et, très probablement, conduit à un inter-blocage ou pire (notre mécanisme de communication repose sur la connaissance des types reçus : ici, une donnée du mauvais type pourrait être envoyée et entraîner une *erreur de segmentation*, le marshalling d'Ocaml étant non typé[8]). Il faut donc un mécanisme capable de discerner les exceptions locales des exceptions globales, et limiter la portée de la propagation des exceptions locales.

Traitement des exceptions parallèles

Remarquons qu'intuitivement, on souhaite voir la machine parallèle s'interrompre pour laisser place à l'exception locale. Cela imposerait un mécanisme supplémentaire dans la machine BSP, une sorte d'alerte interrompant les autres processeurs à distance pour rendre immédiatement l'exception locale, globale. Outre sa faisabilité technique, cette solution pose plusieurs problèmes : en pliant ainsi les limites du modèle BSP, on en perd les propriétés capitales comme la prédiction de performance, et surtout le déterminisme : si plusieurs processeurs doivent lever une exception locale, laquelle sera propagée devient imprévisible. L'exemple suivant montre un programme qui serait indéterministe avec un tel mécanisme :

```
let non_deterministic = try ≪ raise (Excb $this$) ≫ with
  | Excb n → n
```

Il convient donc de se conformer au modèle : dans ce but, l'exception devra attendre la fin de la super-étape pour pouvoir être communiquée aux autres processeurs. En pratique, à moins d'utiliser des exceptions locales à une fréquence très élevée – ce qui n'est généralement pas leur usage, cela a peu d'impact sur les performances globales.

Pour ce faire, l'exception locale doit pouvoir être distinguée des exceptions globales au niveau de chaque processeur. De la sorte, elle permet aux calculs répliqués de continuer à se dérouler

[8] Des travaux existent pour typer la sérialisation en OCaml [HMC07], mais ils ne nous sont pas nécessaires ici, nos communications étant structurées et typées

normalement, sans mettre en cause la cohérence répliquée. Les calculs locaux, pour leur part, doivent être omis sur un processeur qui est en attente de communications après avoir levé une exception locale : ces calculs pourraient dépendre de valeurs locales précédentes dont le calcul a échoué, ou même lever de nouvelles exceptions locales.

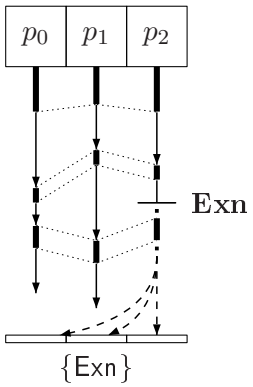


FIG. 3.2 – Blocage de calculs locaux par exception, et globalisation.

La figure 3.2 montre le comportement du système face à une exception levée sur le processeur p_2 : celui-ci ignore ses calculs locaux, poursuit les calculs répliqués et attend la phase de communication pour signaler l'exception.

On mémorise donc un état local pour chaque processeur :
- soit normal
- soit, ayant levé une exception locale, en attente de la phase de communications et n'effectuant plus que les opérations répliquées.

En conséquence, toutes les primitives parallèles se voient ajouter deux traitements particuliers :
- le premier rattrape toute exception levée localement afin qu'elle ne se propage pas dans le niveau global, et mémorise l'état du processeur.
- le second prévient toute exécution de code local si l'état du processeur ne s'y prête pas.

Remarquons que de la sorte, peuvent apparaître des vecteurs parallèles à «trous», certaines de leur composantes locales étant inexistantes (dans la sémantique, nous noterons ces composantes $\perp$). Ceux-ci ne posent pas de problème pour le moment : il est garanti qu'on ne peut y accéder. L'introduction de références dans le langage, cependant, nous fait perdre cette garantie : le problème est discuté en 5.3.1.

Propagation et rattrapage

Nous avons mis en place les mécanismes évitant une mise en déroute de notre système en présence d'exceptions locales non rattrapées. Il reste à définir un moyen de rattraper ces exceptions une fois la phase de communications atteinte.

Il devient, à ce moment là, possible de communiquer les exceptions à tous les processeurs sans sortir du modèle BSP. Cette communication remplace la communication des valeurs qui aurait normalement eu lieu, et permet de revenir dans un mode d'exécution répliqué cohérent.

On aura ainsi à traiter trois types d'exceptions différentes :

- les exceptions répliquées, levées en mode répliqué
- les exceptions locales, levées localement, et qui peuvent être rattrapées localement ou communiquées à la fin de la super-étape
- les «ensembles d'exceptions», ou exceptions globales, qui sont le résultat de la communication d'exceptions locales qui n'ont pas été rattrapées localement.

À ce stade, il reste à gérer la propagation et le rattrapage d'un ensemble d'exceptions pouvant avoir de 1 à p éléments.

- La propagation a lieu de la même façon que pour une exception répliquée habituelle.
- En ce qui concerne le rattrapage, nous choisissons d'introduire une nouvelle construction syntaxique **trypar** [*code*] **withpar** [*motif*] → [*traitement*] | ....

Il aurait été possible de maintenir une unité de traitement entre exceptions répliquées et ensembles d'exceptions, et d'utiliser un bloc de rattrapage standard pour rattraper ceux-ci en définissant une exception Exception_set **of** exn list par exemple. On les aurait alors rattrapé globalement avec :

```
try [ code levant des exceptions locales ] with
  | Exception_set exnlist → [ traitement exnlist ]
```

Le principal avantage de cette approche se situe au niveau de l'implantation (pas de construction supplémentaire nécessaire) ; de plus elle évite tout conflit entre ensembles d'exceptions et exceptions répliquées en ce qui concerne le rattrapage. Nous avons fait le choix d'une nouvelle construction pour deux raisons :

- Le rattrapage d'un ensemble d'exceptions est soumis à l'existence d'une barrière de synchronisation. Le **trypar withpar** nous permet de préciser explicitement cette barrière : avec l'approche utilisant **try** [*code*] **with**, une exception locale qui serait levée après la dernière barrière dans [*code*] ne serait pas rattrapée, ce qui serait déroutant pour le programmeur. Celui-ci devrait placer manuellement une barrière à la fin de ce bloc pour éviter d'étranges surprises dans la suite, comme l'ensemble d'exception réapparaissant à la prochaine barrière. Notre syntaxe permet de manière générale d'assurer que l'ensemble d'exceptions est rattrapé là où les exceptions locales ont été levées, et non pas là où la barrière a lieu.
- Dans notre implantation, le **withpar** nous permet d'utiliser l'extension syntaxique pour le filtrage par motifs de valeurs parallèles (voir section 2.8), qui simplifie grandement la gestion d'un ensemble d'exceptions, évitant des filtrages et tests répétés sur le contenu de cet ensemble.

Un point reste délicat : sans traitement particulier, la propagation d'une exception locale aurait lieu à partir du moment où celle-ci est globalisée, or il est plus souhaitable d'avoir une propagation de l'ensemble d'exceptions correspondant au moment où l'exception a été levée. Par exemple, dans le code suivant :

```
let x = ≪ 1 / $this$ ≫ in trypar () withpar e → [traitement]
```

l'exception Division_by_zero levée par le premier terme sur le processeur 0 est mémorisée dans l'état des processeurs. Elle est généralisée à la première barrière, à l'intérieur du bloc **trypar withpar** : en résulte un ensemble d'exceptions que l'on ne souhaite pas voir rattrapé par ce bloc, l'exception d'origine ayant été levée à l'extérieur. En plus de la barrière du **withpar**, qui empêche l'exception locale de s'échapper vers l'extérieur, il faut donc ajouter au niveau du

trypar un mécanisme qui sépare les exceptions locales déjà levées des autres (une barrière avant chaque **trypar** fonctionnerait, mais des solutions moins coûteuses existent).

Cette séparation est valable dans les deux sens, comme le montre l'exemple suivant :

```
let x = ≪ 1 / $this$ ≫ in
  trypar ≪ 1 / ($this$−1) ≫ withpar e → [traitement]
```

Deux exceptions Division_by_zero sont ici levées localement sur les processeurs 0 et 1 avant la barrière qui intervient au **withpar**. Une fois ces exceptions communiquées et levées sous forme d'ensemble, on fait face à un paradoxe : si cet ensemble est rattrapé par le bloc **trypar withpar**, il rattrape l'exception du processeur 0 qui a été levée à l'extérieur, mais s'il ne l'est pas, l'exception du processeur 1 s'échappe du bloc à l'intérieur duquel elle a été levée. Comme il existe, dans ce cas, une notion de chronologie entre les deux exceptions, on a la possibilité d'ignorer l'exception du processeur 1 : **trypar** instaure une hiérarchie entre les exceptions. Dans notre approche, le programme ci-dessus se terminera simplement par l'exception non rattrapée du processeur 0.

3.4.2 Sémantique

Expressions et opérateurs

Les constructions suivantes sont ajoutées à μBSML pour permettre le traitement des exceptions. Dans le contexte de cette sémantique, on peut considérer que les exceptions sont des valeurs quelconques.

$$
\begin{array}{rcll}
e & ::= & \cdots & \\
 & | & \mathsf{try}\, e\, \mathsf{with}\, x \to e & \textit{rattrapage} \\
 & | & \mathsf{trypar}\, e\, \mathsf{withpar}\, x \to e & \textit{rattrapage parallèle} \\
\multicolumn{4}{l}{\textit{Absents du langage programmeur}} \\
 & | & \mathit{fail}\, e & \textit{échec} \\
 & | & \mathit{failpar}\, \Omega & \textit{échec parallèle} \\
 & | & \mathit{tryparsub}\, e\, \mathsf{withpar}\, x \to e & \textit{rattrapage parallèle intermédiaire}
\end{array}
$$

$$
\begin{array}{rcll}
op & ::= & \cdots & \\
 & | & \mathsf{raise} & \textit{levée d'exception}
\end{array}
$$

Une exception levée est dénotée par l'expression $\mathit{fail}\, v$, l'opérateur $\mathsf{raise}\, e$ permettant de déclencher cette exception après réduction de e. Afin de représenter la propagation des ensembles d'exceptions, en cas d'exceptions locales non rattrapées localement, on utilise $\mathit{failpar}\, v$. Ces expressions sont non réductibles, mais ne sont pas des valeurs puisqu'elles ne peuvent pas être utilisées dans la réduction d'expressions (excepté les blocs de rattrapage bien sûr). Des règles spécifiques permettront de réduire la totalité de l'expression à l'exception levée.

Le contexte est étendu pour réduire à l'intérieur des blocs de rattrapage (le **trypar** subit une transformation préliminaire afin d'etre transformé en *tryparsub*) :

$$\begin{array}{rcl} \mathrm{E} & ::= & \cdots \\ & | & \mathsf{try}\,\mathrm{E}\,\mathsf{with}\,x \to e \\ & | & \mathit{tryparsub}\,\mathrm{E}\,\mathsf{withpar}\,x \to e \end{array}$$

Il est nécessaire de pouvoir mémoriser l'état de chaque processeur, comme indiqué précédemment. On définit dans ce but l' «état du système» Ω : il mémorise l'éventuelle exception levée et non rattrapée localement pour chaque processeur, ce qui permettra à la fin de la super-étape de transformer l'expression en $\mathit{failpar}(\Omega)$. Ce dernier pourra ensuite se propager de façon répliquée.

Afin de conserver la hiérarchie d'imbrication de blocs trypar withpar, on définit Ω comme une suite finie d'ensembles de couples associant un processeur à une exception : l'union de tous ces ensembles forme une fonction des processeurs vers les exceptions. On notera $\Omega = \omega_1, \ldots, \omega_n$, où ω_n correspond au niveau d'imbrication de trypar withpar le plus interne ; on notera $\cup\omega = \bigcup_i \omega_i$ l'état général dy système, que l'on considérera comme une fonction de l'ensemble des processeurs $\mathcal{P}$ dans l'ensemble des expressions.

Les notations utilisées sont similaires à celles que nous avons définies pour les états mémoire dans le cadre de $\mu\text{BSML}^{\mathsf{ref}}$:

- on écrira $(\Omega)e$ pour l'expression e dans l'état système Ω.
- $\cup\Omega(i)$ désigne la valeur associée à i dans $\omega_1 \cup \ldots \cup \omega_n$.
- $\Omega \cdot \{i \mapsto v\}$ désigne Ω auquel a été ajouté la paire (i, v) dans le niveau le plus interne : $\Omega \cdot \{i \mapsto v\} = (\omega_1, \ldots, \omega_n \cup \{(i, v)\})$. On suppose dans ce cas que $i \notin \mathrm{Dom}(\cup\Omega)$ (ce que garantit notre sémantique), afin que $\cup\Omega$ soit bien une fonction partielle.

Un état sans exceptions correspond donc à $\cup\Omega = \emptyset$. Les tests, localement, ne porteront que sur l'éventuelle appartenance du pid i du processeur à $\mathrm{Dom}(\cup\Omega)$ (une réponse positive indiquant que le processeur a levé une exception localement). Ceci, en accord avec l'absence de communications, l'information contenue dans Ω étant en réalité distribuée sur la machine parallèle. Les primitives synchrones, en revanche, effectuant des communications, ont la possibilité de faire des tests sur Ω tout entier : $\cup\Omega \neq \emptyset$ indique qu'on est en présence d'exceptions locales non rattrapées, qui doivent donc être levées sous forme d'ensemble d'exceptions. Ω est alors communiqué et placé dans l'expression par l'intermédiaire de *failpar*, ce qui permet de le traiter de manière répliquée.

Enfin, dans un langage avec exceptions, le retour d'un programme bien formé ne se réduit plus seulement à une valeur, mais à un *résultat* r :

$$r ::= (\emptyset)v \mid (\emptyset)\,\mathit{fail}\,v \mid (\omega)\,\mathit{failpar}(\omega)$$

Sémantique

Une exception se lève par raise, qui obéit à la δ-règle suivante, indépendante de la localité car *fail* peut correspondre aussi bien à une exception locale qu'à une exception répliquée. *failpar* pour sa part correspond aux ensembles d'exceptions et ne peut pas être levé directement :

$$(\Omega)\,\mathsf{raise}\,v \rightarrowtail^{\epsilon} (\Omega)\,\mathit{fail}\,v$$

Comme pour μBSML$^{\mathsf{ref}}$, on suppose qu'implicitement toutes les règles de réduction de tête déjà définies conservent l'état Ω sans modification. Les règles CONTEXT et LOCAL$_i$ transmettent les modifications d'état des processeurs dues à la réduction du sous-terme à tout le terme, toujours comme dans le cas de μBSML$^{\mathsf{ref}}$:

$$\frac{(\Omega_1)e_1 \rightarrowtail^{\epsilon} (\Omega_2)e_2}{(\Omega_1)\mathrm{E}[e_1] \rightarrowtail (\Omega_2)\mathrm{E}[e_2]}\ \text{CONTEXT}$$

$$\forall i \in \mathcal{P}, \frac{(\Omega_1)e_1 \rightarrowtail^{\epsilon} (\Omega_2)e_2}{(\Omega_1)\mathrm{E}_g\left[\langle e'_0 \dots e'_{i-1}, \mathrm{E}_\ell[e_1], e'_{i+1} \dots e'_{p-1}\rangle\right] \rightarrowtail (\Omega_2)\mathrm{E}_g\left[\langle e'_0 \dots e'_{i-1}, \mathrm{E}_\ell[e_2], e'_{i+1} \dots e'_{p-1}\rangle\right]}\ \text{LOCAL}_i$$

Il serait possible de procéder en définissant des règles de réduction de tête similaires à chacune de celles de μBSML pour traiter le cas où on a un échec au lieu d'une valeur, toutes ces règles réduisant à l'échec. Ainsi, toutes l'expression serait progressivement réduite à l'échec seul. Une approche plus simple est de définir un nouveau contexte Δ et de réduire en une seule étape avec une règle semblable à CONTEXT. Cela nous permet également de traiter séparément les cas local et global.

Le *contexte de propagation d'exceptions* est défini comme :

$$\Delta ::= [\cdot] \mid e\ \Delta \mid \Delta\ v \mid \mathsf{let}\, x = \Delta\, \mathsf{in}\, e \mid (\Delta, e) \mid (v, \Delta) \mid \mathsf{if}\, \Delta\, \mathsf{then}\, e\, \mathsf{else}\, e \mid [v, \dots, v, \Delta, e, \dots, e]_n$$

ce qui correspond à E, excepté les blocs de rattrapage, au travers desquels les exceptions ne doivent pas être propagées. Ces derniers sont réduit par les deux règles suivantes, la première traitant le cas du rattrapage d'une exception, la seconde la réduction d'un bloc sans exception.

$$\mathsf{try}\, \Delta[\mathit{fail}\, v]\, \mathsf{with}\, x \rightarrow e \rightarrowtail^{\epsilon} \{v/x\}e \qquad\qquad \mathsf{try}\, v\, \mathsf{with}\, x \rightarrow e \rightarrowtail^{\epsilon} v$$

Il reste le cas où une composante d'un vecteur parallèle se réduit en échec ($\mathit{fail}\, v$), ce qui est précisément le cas problématique que nous avons mentionné dans notre analyse du rapport entre exceptions et BSML. Dans ce cas, l'exception est mémorisée dans l'état processeur et l'exécution répliquée continue ; on obtient ce comportement à l'aide de la règle de réduction suivante, au niveau global :

$$(\Omega)\mathrm{E}\left[\langle e_0, \dots, e_{i-1}, \Delta[\mathit{fail}\, v], \dots, e_{p-1}\rangle\right] \rightarrowtail (\Omega \cdot \{i \mapsto v\})\mathrm{E}\left[\langle e_0, \dots, e_{i-1}, \bot, \dots e_{p-1}\rangle\right]$$

Lors de l'évaluation, un vecteur parallèle contenant partout soit des valeurs, soit $\bot$ est considéré comme une valeur : c'est ce qui permet à l'exécution répliquée de se poursuivre. Toute exécution locale ultérieure, en revanche, est suspendue sur le processeur concerné, et les primitives parallèles asynchrones (mkpar et apply) renvoient $\bot$ sur tout processeur i appartenant à $\mathrm{Dom}(\cup\Omega)$.

$$(\Omega)\,\mathsf{mkpar}\,v \rightarrowtail^{\epsilon} (\Omega) \left\langle \begin{array}{ll} \bullet\ \bot & \text{si } i \in \mathrm{Dom}(\cup\Omega) \\ \bullet\ v\ i & \text{sinon} \end{array} \right\rangle_i$$

$$(\Omega)\,\mathsf{apply}\,\langle v_i^1 \rangle_i\ \langle v_i^2 \rangle_i \rightarrowtail^{\epsilon} (\Omega) \left\langle \begin{array}{ll} \bullet\ \bot & \text{si } i \in \mathrm{Dom}(\cup\Omega) \\ \bullet\ v_i^1\ v_i^2 & \text{sinon} \end{array} \right\rangle_i$$

$$(\Omega)\,\mathsf{proj}\,\langle v_i \rangle_i \rightarrowtail^{\epsilon} \begin{array}{ll} \bullet\ (\Omega)\ \mathit{failpar}(\Omega) & \text{si } \cup\Omega \neq \emptyset \\ \bullet\ (\Omega)\ \mathsf{nth}\,[v_i]_i^p & \text{sinon} \end{array}$$

$$(\Omega)\,\mathsf{put}\,\langle v_i \rangle_i \rightarrowtail^{\epsilon} (\Omega)\,\mathsf{apply}\,\langle \mathsf{nth} \rangle_i\ \ (\mathit{send} \left\langle \begin{array}{ll} \bullet\ \bot & \text{si } i \in \mathrm{Dom}(\cup\Omega) \\ \bullet\ [v_i\ j]_j^p & \text{sinon} \end{array} \right\rangle_i)$$

$$(\Omega)\ \mathit{send} \left\langle \left[v_j^i\right]_j^p \right\rangle_i \rightarrowtail^{\epsilon} \begin{array}{ll} \bullet\ (\Omega)\ \mathit{failpar}(\Omega) & \text{si } \cup\Omega \neq \emptyset \\ \bullet\ (\Omega) \left\langle \left[v_j^i\right]_i^p \right\rangle_j & \text{sinon} \end{array}$$

Dans le cas où il n'y a aucune exception ($\cup\Omega = \emptyset$), ces règles correspondent exactement aux règles de μBSML.

Avant de permettre la réduction par le contexte à l'intérieur des blocs trypar withpar en les transformant en *tryparsub* withpar, on ajoute un niveau d'imbrication au contexte : c'est le seul rôle de la construction intermédiaire *tryparsub*.

$$(\Omega)\,\mathsf{trypar}\,e\,\mathsf{withpar}\,x \rightarrow e' \rightarrowtail^{\epsilon} (\Omega, \emptyset)\ \mathit{tryparsub}\,e\,\mathsf{withpar}\,x \rightarrow e'$$

À l'inverse, les règles de réduction de *tryparsub* withpar éliminent le dernier élément de Ω :

$$(\Omega, \omega)\ \mathit{tryparsub}\,\Delta[\mathit{failpar}(\Omega, \omega)]\,\mathsf{withpar}\,x \rightarrow e \rightarrowtail^{\epsilon} \begin{array}{ll} \bullet\ (\Omega)\{\mathcal{T}(\omega)/x\}e & \text{si } \cup\Omega = \emptyset \\ \bullet\ (\Omega)\ \mathit{failpar}\,\Omega & \text{sinon} \end{array}$$

$$(\Omega, \omega)\ \mathit{tryparsub}\,v\,\mathsf{withpar}\,x \rightarrow e \rightarrowtail^{\epsilon} \begin{array}{ll} \bullet\ (\Omega)v & \text{si } \cup\Omega \cup \omega = \emptyset \\ \bullet\ (\Omega)\{\mathcal{T}(\omega)/x\}e & \text{si } \cup\Omega = \emptyset \text{ et } \omega \neq \emptyset \\ \bullet\ (\Omega)\ \mathit{failpar}(\Omega) & \text{si } \Omega \neq \emptyset \text{ et } \omega = \emptyset \end{array}$$

Où la notation $\mathcal{T}(\omega)$ indique la transformation en expression du langage de ω : par exemple, un tableau des valeurs levées localement. Dans ces règles, on décompose l'état global en (Ω, ω), où ω est l'élément de l'environnement correspondant au niveau de trypar withpar courant.

- Les tests portant sur Ω permettent de s'assurer qu'il n'y a pas d'exceptions en dehors du bloc courant – auquel cas il convient d'ignorer les exceptions courantes pour relever celles-ci ;
- Les tests portant sur ω indiquent les exceptions à rattraper dans le bloc trypar withpar courant ;
- Les tests portant sur (Ω, ω) permettent de conclure qu'il n'y a aucune exception locale, et de continuer normalement.

La deuxième règle rend apparente la nature synchrone de withpar, en effectuant des tests sur l'ensemble Ω afin d'éviter que des exceptions locales contenues dans ω ne puissent s'échapper.

Deux remarques nous seront utiles dans la suite : d'une part, lorsqu'on propage un *failpar*(Ω), on préserve dans toutes les règles la cohérence entre l'état des processeurs et Ω : par conséquent, des règles (Ω_1) *failpar*(Ω_2) ne sont pas nécessaires. D'autre part, le rattrapage d'un ensemble d'exceptions est fait sous la condition que $\cup\Omega = \emptyset$, qui garantit que le rattrapage ramène l'exécution dans un état sain.

Un dernier élément reste à éclaircir, concernant les interactions entre les exceptions locales et globales : un bloc try with laisse se propager un ensemble d'exceptions, et inversement, une exception répliquée n'est pas rattrapée par un bloc trypar withpar.

$$(\Omega)\,\mathsf{try}\ \mathit{failpar}\,v\ \mathsf{with}\,x \to e \rightarrowtail^{\epsilon} (\Omega)\ \mathit{failpar}\,v$$

$$(\Omega,\omega)\ \mathit{tryparsub}\ \mathit{fail}\,v\ \mathsf{withpar}\,x \to e \rightarrowtail^{\epsilon} \begin{cases} \bullet\ (\Omega)\ \mathit{fail}\,v & \text{si } \cup\Omega\cup\omega = \emptyset \\ \bullet\ (\Omega)\{\mathcal{T}(\omega)/x\}e & \text{si } \cup\Omega = \emptyset \text{ et } \omega \neq \emptyset \\ \bullet\ (\Omega)\ \mathit{failpar}(\Omega) & \text{si } \Omega \neq \emptyset \text{ et } \omega = \emptyset \end{cases}$$

Ce dernier cas est particulier et nécessite, comme nous le faisons ici, de choisir une politique de traitement. En effet, en n'établissant pas de barrière, on pourrait laisser une exception locale en attente (celle-ci serait globalisée lors de la prochaine barrière) ; mais l'établissement d'une barrière ici signifie d'une part que l'on interrompt temporairement la propagation de l'exception répliquée, d'autre part que l'on peut se retrouver en présence d'un ensemble d'exceptions *et* d'une exception répliquée à la fois. L'un ou l'autre doit être abandonné, et notre politique dans ces cas de conflits est de toujours préserver l'exception la plus ancienne[9]. Ici, il s'agit nécessairement de l'exception locale, la levée de l'exception répliquée n'ayant pu être suivie de calculs locaux : l'exception répliquée est donc abandonnée au profit du traitement de l'ensemble d'exceptions locales.

Cette approche a un défaut : si une exception répliquée est levée à l'intérieur de n niveaux d'imbrication de trypar withpar (par exemple, au sein d'une fonction récursive), l'exécution suivra n barrières de synchronisation successives, une à chaque withpar, dont $(n-1)$ inutiles avant de pouvoir récupérer l'exception. Il n'est, heureusement, pas très difficile d'optimiser l'exécution pour éviter ces barrières inutiles dans l'implantation, au prix d'une légère perte dans la simplicité de la prédiction de performances. Notre sémantique ne tient pas compte de cette optimisation.

Enfin, ces dernières règles réduisent les programmes comportant des exceptions non rattrapées, afin d'obtenir un résultat dans r

$$(\omega)\Delta[\mathit{fail}\,v] \rightarrowtail \begin{cases} \bullet\ (\omega)\ \mathit{fail}\,v & \text{si } \omega = \emptyset \\ \bullet\ (\omega)\ \mathit{failpar}(\omega) & \text{si } \omega \neq \emptyset \end{cases} \quad (\text{si } \Delta \neq [\cdot])$$

$$(\omega)\Delta[\mathit{failpar}(\omega)] \rightarrowtail (\omega)\ \mathit{failpar}(\omega)\ (\text{si } \Delta \neq [\cdot]) \qquad (\omega)v \rightarrowtail (\omega)\ \mathit{failpar}(\omega)\ (\text{si } \omega \neq \emptyset)$$

[9] cette notion d'ancienneté a ici un sens précis, bien qu'il ne soit pas purement temporel : en effet, on ne peut établir qu'une exception répliquée est levée à un instant précis, cet instant pouvant varier suivant les processeurs. On peut en revanche établir qu'une exception locale sur le processeur i est plus ou moins ancienne que cette exception répliquée en choisissant l'instant où l'exception répliquée est levée sur le processeur i.

On peut ici supposer que l'état global est de la forme (ω) puisqu'il n'y a aucune imbrication de trypar withpar. Les première et troisième règles imposent la présence d'une barrière de synchronisation à la fin du programme, assurant qu'un message d'erreur correct indiquant toutes les exceptions levées peut être affiché.

Exemple

La réduction de la récupération globale d'exceptions locales se déroule comme suit : on prend comme exemple l'expression

$$\mathsf{trypar\ mkpar}(\mathsf{fun}\ i \to \mathsf{raise}\ \epsilon)\ \mathsf{withpar}\ x \to 1$$

$$\begin{array}{ll}
 & (\emptyset)\ \mathsf{trypar\ mkpar}(\mathsf{fun}\ i \to \mathsf{raise}\ \epsilon)\ \mathsf{withpar}\ x \to 1 \\
\rightarrowtail & (\emptyset,\emptyset)\ \mathit{tryparsub}\ \mathsf{mkpar}(\mathsf{fun}\ i \to \mathsf{raise}\ \epsilon)\ \mathsf{withpar}\ x \to 1 \\
\rightarrowtail & (\emptyset,\emptyset)\ \mathit{tryparsub}\ \langle(\mathsf{fun}\ i \to \mathsf{raise}\ \epsilon)\ 0, (\mathsf{fun}\ i \to \mathsf{raise}\ \epsilon)\ 1\rangle\ \mathsf{withpar}\ x \to 1 \\
\rightarrowtail & (\emptyset,\emptyset)\ \mathit{tryparsub}\ \langle\mathsf{raise}\ \epsilon, (\mathsf{fun}\ i \to \mathsf{raise}\ \epsilon)\ 1\rangle\ \mathsf{withpar}\ x \to 1 \\
\rightarrowtail & (\emptyset,\emptyset)\ \mathit{tryparsub}\ \langle \mathit{fail}\ \epsilon, (\mathsf{fun}\ i \to \mathsf{raise}\ \epsilon)\ 1\rangle\ \mathsf{withpar}\ x \to 1 \\
\rightarrowtail & (\emptyset,\emptyset)\ \mathit{tryparsub}\ \langle \mathit{fail}\ \epsilon, \mathsf{raise}\ \epsilon\rangle\ \mathsf{withpar}\ x \to 1 \\
\rightarrowtail & (\emptyset,\{(0,\epsilon)\})\ \mathit{tryparsub}\ \langle\bot, \mathsf{raise}\ \epsilon\rangle\ \mathsf{withpar}\ x \to 1 \\
\rightarrowtail & (\emptyset,\{(0,\epsilon)\})\ \mathit{tryparsub}\ \langle\bot, \mathit{fail}\ \epsilon\rangle\ \mathsf{withpar}\ x \to 1 \\
\rightarrowtail & (\emptyset,\{(0,\epsilon),(1,\epsilon)\})\ \mathit{tryparsub}\ \langle\bot,\bot\rangle\ \mathsf{withpar}\ x \to 1 \\
\rightarrowtail & (\emptyset)\{\mathcal{T}(\{(0,\epsilon),(1,\epsilon)\})/x\}1 \\
\rightarrowtail & (\emptyset)\{[(0,\epsilon),(1,\epsilon)])/x\}1 \\
\rightarrowtail & (\emptyset)1
\end{array}$$

L'intérêt du *tryparsub* et de l'imbrication de contexte se serait manifesté ici si on avait eu une exception locale tout au début du programme :

$$\mathsf{let}\ y = \mathsf{mkpar}(\mathsf{fun}\ i \to \mathsf{if}\ i = 1\ \mathsf{then\ raise}\ \epsilon'\ \mathsf{else}\ i)\ \mathsf{in\ trypar\ mkpar}(\mathsf{fun}\ i \to \mathsf{raise}\ \epsilon)\ \mathsf{withpar}\ x \to 1$$

$$\begin{array}{ll}
 & (\emptyset)\ \mathsf{let}\ y = \mathsf{mkpar}(\mathsf{fun}\ i \to \mathsf{if}\ i = 1\ \mathsf{then\ raise}\ \epsilon'\ \mathsf{else}\ i)\ \mathsf{in}\cdots \\
\rightarrowtail & (\emptyset)\ \mathsf{let}\ y = \langle(\mathsf{fun}\ i \to \mathsf{if}\ i = 1\ \mathsf{then\ raise}\ \epsilon'\ \mathsf{else}\ i)\ j\rangle_j\ \mathsf{in}\cdots \\
\rightarrowtail & (\emptyset)\ \mathsf{let}\ y = \langle\mathsf{if}\ 0 = 1\ \mathsf{then\ raise}\ \epsilon'\ \mathsf{else}\ j, \mathsf{if}\ 1 = 1\ \mathsf{then\ raise}\ \epsilon'\ \mathsf{else}\ j\rangle\ \mathsf{in}\cdots \\
\rightarrowtail & (\emptyset)\ \mathsf{let}\ y = \langle\mathsf{if}\ 0 = 1\ \mathsf{then\ raise}\ \epsilon'\ \mathsf{else}\ j, \mathsf{if\ true\ then\ raise}\ \epsilon'\ \mathsf{else}\ j\rangle\ \mathsf{in}\cdots \\
\rightarrowtail & (\emptyset)\ \mathsf{let}\ y = \langle\mathsf{if\ false\ then\ raise}\ \epsilon'\ \mathsf{else}\ j, \mathsf{if\ true\ then\ raise}\ \epsilon'\ \mathsf{else}\ j\rangle\ \mathsf{in}\cdots \\
\rightarrowtail & (\emptyset)\ \mathsf{let}\ y = \langle\mathsf{if\ false\ then\ raise}\ \epsilon'\ \mathsf{else}\ j, \mathsf{raise}\ \epsilon'\rangle\ \mathsf{in}\cdots \\
\rightarrowtail & (\emptyset)\ \mathsf{let}\ y = \langle\mathsf{if\ false\ then\ raise}\ \epsilon'\ \mathsf{else}\ j, \mathit{fail}\ \epsilon'\rangle\ \mathsf{in}\cdots \\
\rightarrowtail & (\{1,\epsilon'\})\ \mathsf{let}\ y = \langle\mathsf{if\ false\ then\ raise}\ \epsilon'\ \mathsf{else}\ j, \bot\rangle\ \mathsf{in}\cdots \\
\rightarrowtail & (\{1,\epsilon'\})\ \mathsf{let}\ y = \langle j,\bot\rangle\ \mathsf{in\ trypar\ mkpar}(\mathsf{fun}\ i \to \mathsf{raise}\ \epsilon)\ \mathsf{withpar}\ x \to 1 \\
\rightarrowtail & (\{1,\epsilon'\})\{\langle j,\bot\rangle\ /y\}\ \mathsf{trypar\ mkpar}(\mathsf{fun}\ i \to \mathsf{raise}\ \epsilon)\ \mathsf{withpar}\ x \to 1 \\
\rightarrowtail & (\{1,\epsilon'\})\ \mathsf{trypar\ mkpar}(\mathsf{fun}\ i \to \mathsf{raise}\ \epsilon)\ \mathsf{withpar}\ x \to 1 \\
\rightarrowtail & (\{1,\epsilon'\},\emptyset)\ \mathit{tryparsub}\ \mathsf{mkpar}(\mathsf{fun}\ i \to \mathsf{raise}\ \epsilon)\ \mathsf{withpar}\ x \to 1 \\
\rightarrowtail & \cdots \\
\rightarrowtail & (\{1,\epsilon'\},\{(0,\epsilon),(1,\epsilon)\})\ \mathit{tryparsub}\ \langle\bot,\bot\rangle\ \mathsf{withpar}\ x \to 1 \\
\rightarrowtail & (\{1,\epsilon'\})\ \mathit{failpar}\{1,\epsilon'\}
\end{array}$$

Les exceptions locales levées à l'intérieur du bloc trypar withpar ont été abandonnées au profit de l'exception qui les précédait.

3.4.3 Confluence

Les règles ajoutées à la sémantique de μBSML permettent de réduire uniquement les expressions contenant les nouveaux éléments syntaxiques (exceptions et blocs de rattrapage). La sémantique de μBSML n'est donc pas changée tant qu'on n'utilise pas ces nouveaux éléments.

Propriété 3.4.1 (*confluence de μBSML$^{\mathsf{exn}}$*)

La sémantique de μBSML$^{\mathsf{exn}}$ est confluente.

Démonstration : On étend la preuve par induction structurelle sur e de 3.2.4 en tenant compte des nouveaux cas possibles :

- Dans tous les cas supposant des valeurs qui apparaissent dans la preuve 3.2.4, on peut trouver $\mathit{fail}\, v$ ou $\mathit{failpar}\, v$ à la place de ces valeurs : toute réduction par la sémantique de μBSML est alors bloquée.
  La réduction se fait alors par les règles de rattrapage d'exceptions : l'expression peut se décomposer de façon unique sous l'une des formes (l'unicité découle de l'absence des blocs de rattrapage dans les contextes δ) :
  - $\mathrm{E}[\mathsf{try}\, \Delta[\mathit{fail}\, v]\, \mathsf{with}\, x \to e]$
  - $\mathrm{E}[\mathsf{try}\, \Delta[\mathit{failpar}\, v]\, \mathsf{with}\, x \to e]$
  - $\mathrm{E}[\mathsf{trypar}\, \Delta[\mathit{fail}\, v]\, \mathsf{withpar}\, x \to e]$
  - $\mathrm{E}[\mathsf{trypar}\, \Delta[\mathit{failpar}\, v]\, \mathsf{withpar}\, x \to e]$

  Ce qui autorise une réduction détermniniste par les règles CONTEXT et LOCAL$_i$, en utilisant les règles de réduction de tête correspondantes. Pour les cas où l'exception n'est pas rattrapée, que ce soit dans le programme complet où dans une composante d'un vecteur parallèle, il s'écrit, toujours de façon unique, sous l'une des formes :
  - $\Delta[\mathit{fail}\, v]$
  - $\Delta[\mathit{failpar}\, v]$
  - $\mathrm{E}[\langle \ldots, \Delta[\mathit{fail}\, v], \ldots \rangle]$
  - $\mathrm{E}[\langle \ldots, \Delta[\mathit{failpar}\, v], \ldots \rangle]$ (expression bloquée)

  La réduction par les règles CONTEXT et LOCAL$_i$ n'est plus possible, celle-ci se fait donc, toujours de manière déterministe, par les règles d'exceptions non rattrapées.
- De nouveaux cas se présentent en dehors de l'apparition de $\mathit{fail}\, v$ et $\mathit{failpar}\, v$ pour la réduction de tête : $\mathsf{try}\, e_1\, \mathsf{with}\, x \to e_2$ et $\mathsf{trypar}\, e_1\, \mathsf{withpar}\, x \to e_2$ peuvent aussi être réduits dans le cas où e_1 est une valeur, ou par contexte dans le cas contraire. Ces cas sont similaires aux cas de μBSML simple.
- Les nouvelles règles de réduction des primitives parallèles effectuent un choix déterministe en fonction de l'état global des processeurs, ce qui ne remet pas en cause la confluence forte. ■

3.5 Conclusion

Dans ce chapitre, nous avons décrit en détail des sémantiques à petits pas et fourni des preuves de cohérence pour μBSML et ses extensions qui permettent, respectivement, les références et les exceptions. Les problèmes soulevés dans les différents cas ont été examinés en détail ainsi que leurs solutions.

La combinaison des extensions proposées ici a été, également, étudiée en détail ; mais la complexité en devient trop grande pour qu'une description formelle reste aussi pertinente. Nous décrirons donc cette étude plus en détail dans le chapitre 5, qui traite de l'implantation de BSML.

Chapitre 4

Système de typage

Afin de garder le maximum de clarté et de concision, notre définition du système de types est séparée en plusieurs parties correspondant aux extensions impératives de μBSML : notre première définition est d'un système de type strictement consacré à μBSML, puis nous l'étendrons à $\mu\text{BSML}^{\mathsf{ref}}$ et à $\mu\text{BSML}^{\mathsf{exn}}$.

4.1 Sûreté des programmes parallèles

De même qu'en ML, un système de types nous permet de nous assurer que les programmes sont bien formés. Il est naturel de baser notre système de types sur celui d'OCaml, mais nous sommes ici confrontés à une complexité supplémentaire du fait du parallélisme (comme l'impossibilité de réduire les primitives parallèles localement). Nous examinons, dans cette section, les difficultés qui sont soulevées et définissons la notion de programmes BSML *bien formés* par extension de cette notion en ML. Trois problèmes principaux sont examinés dans cette section : la préservation de la cohérence répliquée, l'exécution locale de primitives et l'emboîtement de vecteurs parallèles.

4.1.1 Cohérence répliquée

BSML repose à la fois sur l'exécution répliquée et sur l'exécution locale. La sémantique que nous avons présentée considère les variables répliquées comme uniques, mais l'implantation traitera physiquement une instance de ces variables par processeur. Il faut par conséquent s'assurer que ces instances restent égales en permanence, sans quoi l'exécution devient imprévisible et on perd la garantie que les opérations collectives sont exécutées par tous les processeurs. La cohérence répliquée se résume à prouver que toute valeur répliquée découle de façon localement déterministe d'une autre valeur répliquée.

Ce n'est évidemment pas le cas si on admet l'exécution répliquée d'une opérateur aléatoire, par exemple. Soit rndbool un opérateur qui renvoie un booléen aléatoire ; considérons le programme BSML rndbool (). D'après la sémantique, une unique δ-règle est appliquée, ce qui conduit à une valeur répliquée qui est soit **true** soit **false**. Cela ne reflète pas l'exécution[10] qui est faite

[10] à moins qu'il ne s'agisse d'un opérateur pseudo-aléatoire initialisé avec la même graine sur tous les processeurs :

simultanément sur des processeurs indépendants et peut conduire à des valeurs locales incohérentes, sortant BSML de ses rails. Par exemple, le programme **if** rndbool() **then** **proj** v 0 **else** 0 a toutes les chances de conduire à l'exécution de **proj** sur un sous-ensemble des processeurs, et cette opération étant collective, la désynchronisation qui en résultera conduira soit à un blocage, soit à une incompréhension dans les communications et un résultat imprévisible.

4.1.2 Exécution locale de primitives parallèles

Dans notre contexte, les primitives parallèles n'ont pas de sens localement : un programme tel que **mkpar** (**fun** i → **mkpar** f) est mal formé. Au niveau de la sémantique, il nous serait aisé de réduire un tel programme, mais on perdrait la cohérence avec une machine parallèle BSP concrète n'étant pas récursivement parallèle. Différentes représentations existent : le langage NESL, par exemple, abstrait la réalité des processeurs et utilise des threads, ce qui rendrait ce type de programme acceptable, mais retire le contrôle des processeurs au programmeur et a un coût élevé en difficulté d'implantation, portabilité et prédictabilité des performances.

Des débats existent sur la nature multi-niveaux du parallélisme et l'utilité pour l'utilisateur de contrôler ces niveaux (de multiples cœurs à l'intérieur de multiples processeurs, eux-même dans de multiples machines formant des grappes, que l'on peut réunir en grilles...). Une expérience basée sur BSML a été faite à ce sujet, résultant en un langage DMML destiné à la programmation de grille [GL05, Gav04]. Notre avis, basé sur cette expérience, est que la programmation parallèle est déjà d'une considérable complexité par rapport à la programmation usuelle si on se limite à deux niveaux ; de plus, dans le cas d'une grappe de multi-cœurs, le bénéfice obtenu en utilisant trois niveaux est minime. Ces approches semblent ainsi se limiter aux concours de vitesse d'exécution, et il est peu probable qu'elles se généralisent un jour, le rapport entre gain de performance et complexité de programmation ajoutée étant insuffisant.

4.1.3 Vecteurs parallèles emboîtés

Il reste possible, même en interdisant l'exécution locale de primitives, d'obtenir des vecteurs parallèles emboîtés comme dans cet exemple :

let v = ≪ $**this**$ ≫ **in** ≪ v ≫

L'absence de $$ autour du v du membre de droite indique qu'on inclut le vecteur entier, et non pas sa composante locale, dans le nouveau vecteur. La valeur ainsi obtenue est de type int par par, un vecteur de vecteurs. Encore une fois, on s'écarte de la machine parallèle réaliste : la signification pourrait être de diviser chaque processeur en p sous-processeurs, et ce de façon récursive ; ou bien qu'un processeur a la possibilité de contrôler une structure globale. Aucune de ces deux approches n'est compatible avec notre modèle : la première n'est pas cohérente avec le code ci-dessus (elle impliquerait des communications) et produirait des vecteurs non parallèles non dissociables des vecteurs parallèles ; étudions la seconde plus en détail.

La possibilité d'accéder aux valeurs répliquées depuis le code local s'étend aux vecteurs parallèles. Cependant, le critère précédent nous empêche d'ouvrir ceux-ci, les primitives parallèles ne pouvant être utilisées : tant que l'on reste dans le contexte local, aucune incohérence n'est

dans ce cas, le problème revient à conserver une référence répliquée cohérente

avérée car le vecteur emboîté reste opaque. Il y a cependant un problème si **proj** est utilisé pour quitter ce contexte. D'après la sémantique, **proj** ≪ v ≫ (ou, en μBSML, proj($mkpar$(fun $i \rightarrow v$))) se réduit en **fun** i → v.

Le vecteur v complet n'est pas connu sur tous les processeurs, cependant, et (≪ v ≫ : int par par) et (≪ v≫ = v: int par) correspondent à une même information disponible sur chaque processeur. L'opérateur **proj** ne prend pas en considération les niveaux d'emboîtement de ≪ v ≫ pour ses communications, et aplatirait ce vecteur. À moins de traiter le cas à part, chaque processeur enverrait sa composante du vecteur en guise de tout, et **proj** ≪ v ≫ 0 renverrait le vecteur $\langle v_0, \ldots, v_0 \rangle$ au lieu de $v = \langle v_0, \ldots, v_{p-1} \rangle$.

Traiter ce cas à part serait possible, afin d'obtenir une implantation qui, dans cette circonstance, suive notre sémantique. Mais ce serait difficile : il faudrait que les opérateurs de communication détectent tout vecteur parallèle emboîté, et retrouvent un identifiant global de ce vecteur afin de le communiquer aux autres plutôt que sa composante locale. Outre la perte de performances impliquée, tout ceci est plus propice à induire en erreur qu'à rendre service. En effet, la seule opération possible sur un vecteur parallèle emboîté serait de le projeter, tel quel : il aurait probablement été plus simple de ne pas commencer par l'emboîter, dans ce cas.

Il est de plus plus clair pour le programmeur de n'avoir que des valeurs locales lors des calculs locaux : simplement interdire un tel emboîtement est plus cohérent avec notre système.

Le cas d'application qui pourrait rendre l'usage de vecteurs emboîtés tentant est l'approche diviser-pour-régner : l'utilisation de cette approche est ainsi particulièrement élégante dans les langages utilisant des threads dynamiques. Nous avons présenté une solution pour cette approche, orthogonale au parallélisme, avec la superposition parallèle exposée en 2.9

4.1.4 Autres considérations

Les considérations exposées ci-dessus n'ont pas pour but de prouver la cohérence de notre sémantique : cela fera l'objet de preuves formelles. Elles concernent la sûreté de l'exécution en BSML : ce dont il s'agit est de s'assurer qu'il est possible d'implanter cette sémantique de façon fidèle sur une machine BSP concrète. Des preuves ont été établies en ce sens pour BSML dans [Gav05], où l'auteur établit des sémantiques de haut et bas niveau et prouve leur équivalence, allant jusqu'à une machine virtuelle. On s'appuie donc sur ces travaux, et on se contente ici de s'arrêter à un stade de spécification dans le réalisme duquel on peut avoir confiance. Une sémantique permettant des accès locaux arbitraires aux valeurs de tous les processeurs, par exemple, n'est pas cohérente dans notre modèle puisque cela impliquerait des communications cachées.

Des critères simples nous servent donc ici de garde-fou, afin d'assurer le réalisme de notre spécification vis-à-vis d'une machine parallèle BSP :

1. La séparation des données : la machine étant composée de paires processeur/mémoire, chaque processeur n'a accès qu'aux données de sa mémoire propre. Les opérations faisant des communications sont précisées, et elles le font dans un cadre limité : un processeur ne peut envoyer que des données dont il dispose. (ces opérations se limitent dans notre système à proj, *send* et, plus tard, withpar)
2. La séparation de l'exécution : de la même façon, le comportement d'un processeur ne peut dépendre de celui d'un autre, sauf par l'intermédiaire de communications.

Les éléments de sûreté que nous avons énoncés ci-dessus découlent en partie de ces critères : la séparation de l'exécution peut être assurée dans notre système bien que nous ayons une sémantique comprenant un niveau d'exécution globale grâce à notre critère de cohérence répliquée. De même, si on acceptait l'exécution locale de primitives et les vecteurs parallèles emboîtés, des mécanismes supplémentaires seraient nécessaires pour imposer la séparation des données et de l'exécution : c'est ce qui entraîne un parallélisme implicite dans les langages tels que NESL.

Ce qui nous amène à une autre considération fondamentale : les restrictions que nous avons choisies ici relèvent principalement de notre modèle. Elles permettent de conserver des mécanismes de parallélisme et de communications relativement simples, et de préserver la validité de notre système de coûts : comme nous l'avons mentionné, d'autres systèmes n'ont pas ces contraintes, au prix d'autres inconvénients. Bien que nous n'ayons pas établi de méta-formalisme nous permettant de prouver les critères de séparation dans notre sémantique, ces préoccupations sont sous-jacentes à toutes nos définitions.

4.2 Système de types pour μBSML

Description informelle :

- Afin de prévenir l'usage local des primitives parallèles et l'usage global d'opérateurs non-déterministes, on utilise des *effets* [TJ92, Tal93] : la réduction d'une primitive parallèle produit l'effet g (pour *global*). Les constructions syntaxiques usuelles, par la suite, demandent l'unification des effets de leurs prémisses. On refuse de la sorte une expression telle que $(\mathsf{mkpar}(\cdots), \mathsf{rndbool}\ ())$, où l'application de mkpar, ne pouvant être que globale, a produit l'effet g, et celle de rndbool, ne pouvant être que locale, a produit l'effet ℓ, pour *local*. Les fonctions déclenchent un effet lors de leur application : on peut de la sorte définir une fonction utilisant rndbool dans n'importe quel contexte, mais son application ne pourra être que locale. Dans ce but, les types flèches intègrent des effets *latents*.
- L'effet est lié à l'évaluation, le type est lié à la valeur : cette différence se manifeste lorsqu'on utilise des constructions comme $\mathsf{let}\, x = e^1\, \mathsf{in}\, e^2$, où la valeur résultant de e^1 se propage dans e^2 ainsi que son type (par l'intermédiaire de l'environnement), alors que l'effet se propage des branches de l'arbre de syntaxe e^1 et e^2 vers sa racine. Dans cet exemple, il est possible que e^2 utilise la valeur x résultant de e^1 localement, même si la réduction de e^1 a produit l'effet g.
- L'emboîtement de vecteurs est rendu impossible par des contraintes que nous ajoutons sur les types des variables accessibles, suivant le contexte d'exécution.

Depuis le premier système de types pour ML défini par Damas et Milner [Mil78, DM69], de nombreuses extensions et améliorations lui ont été apportées. Celles-ci vont de simples évolutions, gardant une approche similaire, à une reformulation plus complète comme celle de [PR05], basée sur les contraintes. Toutes conservent les deux propriétés capitales du système de types original dit Hindley/Milner, à savoir l'existence d'un type principal et la conservation du typage par réduction.

Ces extensions ont permis d'enrichir le langage avec des enregistrements [Rém89], de la surcharge et du sous-typage [Kae92, OWW95], *etc.* Ces approches varient, mais se basent en général sur des notions de contraintes ou d'effets [Wri92, WF94, LP00] ajoutées au système d'origine. Dans [OSW99], les auteurs développent un nouveau système qui généralise la notion de contraintes en introduisant le système HM(X), pour Hindley-Milner dépendant d'un système de

contraintes X. Plus récemment, une description détaillée du système de types actuellement utilisé par OCaml [PR05] utilise un système purement basé sur les contraintes, PCB(X) (équivalent à HM(X)).

Notre langage étant basé sur OCaml, le système de types utilisé est naturellement une extension de celui d'OCaml. Notre présentation reste fidèle aux conventions généralement adoptées dans la définition de systèmes de types de la famille de Damas et Milner : pour cette raison, notre approche s'apparente à HM(X) plutôt qu'à PCB(X). Le système de contraintes X devra donc assurer les propriétés de sûreté du parallélisme énoncées plus haut. L'inférence de types, quant à elle s'appuie largement sur celle de HM(X).

4.2.1 Définition des types

Nos types τ sont construits de la façon suivante, et utilisent les effets Λ.

$$
\begin{array}{rcll}
\tau & ::= & \alpha & \textit{variable de type} \\
 & | & \mathsf{Base} & \textit{type de base} \\
 & | & \tau \xrightarrow{\Lambda} \tau & \textit{flèche} \\
 & | & \tau \times \tau & \textit{couple} \\
 & | & \tau\ \mathsf{array} & \textit{tableau} \\
 & | & \tau\ \mathsf{par} & \textit{vecteur parallèle}
\end{array}
\qquad
\begin{array}{rcll}
\Lambda & ::= & \delta & \textit{variable de localité} \\
 & | & \ell & \textit{local} \\
 & | & g & \textit{global}
\end{array}
$$

$$
\begin{array}{rcll}
\mathrm{C} & ::= & \mathsf{true} & \textit{constrainte unitaire} \\
 & | & \mathsf{false} & \textit{constrainte fausse} \\
 & | & \mathrm{C} \wedge \mathrm{C} & \textit{conjonction} \\
 & | & \tau = \tau & \textit{égalité de types} \\
 & | & \Lambda = \Lambda & \textit{égalité de localités} \\
 & | & \tau \triangleleft \Lambda & \textit{type acceptable localement} \\
 & | & \Lambda \triangleleft \Lambda & \textit{contrainte de localité} \\
 & | & \exists \alpha.\mathrm{C} & \textit{projection (type)} \\
 & | & \exists \delta.\mathrm{C} & \textit{projection (localité)}
\end{array}
$$

Les effets, dans notre système, sont des localités Λ qui peuvent correspondre à ℓ, g ou bien à une variable. Λ peut rester latent dans une fonction, auquel cas on écrit $\tau_1 \xrightarrow{\Lambda} \tau_2$, ce qui indique que l'effet Λ sera produit lors de l'application de la fonction. Les contraintes, pour leur part, permettent de restreindre les valeurs que sont susceptibles de prendre les variables. La principale contrainte que nous utiliserons est $\tau \triangleleft \Lambda$, qui indique que τ doit être *acceptable* dans la localité Λ. Cette propriété est définie par :

Définition 4.2.1 (*Acceptabilité d'un type dans une localité*)

- *tout type est acceptable globalement*
- *les types acceptables localement sont définis par :*

$$\dot{\tau} ::= \mathsf{Base} \mid \tau \xrightarrow{\ell} \dot{\tau} \mid \dot{\tau} \times \dot{\tau} \mid \dot{\tau}\ \mathsf{array}$$

Ainsi, $\tau \triangleleft \ell$ est vérifié si et seulement si $\tau \in \dot{\tau}$, et pour sa part $\tau \triangleleft g$ est toujours vérifié. Cette contrainte nous permet de nous assurer du non-emboîtement de vecteurs : on a la garantie qu'une expression de type τ tel que $\tau \triangleleft \ell$ ne contient pas de vecteurs, donc en assurant cette propriété sur toute composante locale d'un vecteur parallèle (règle d'inférence VECTOR ci-dessous), on ne peut avoir de vecteurs emboîtés.

$\Lambda_1 \triangleleft \Lambda_2$ est simplement défini par $\Lambda_2 = \ell \Rightarrow \Lambda_1 = \ell$.

En vérifiant que $\tau \triangleleft \ell$ lorsqu'on tente d'accéder localement à une variable de type τ dans le contexte d'une part, et en interdisant d'utiliser les primitives parallèles localement et donc de créer de nouveaux vecteurs d'autre part, on garantit qu'il est impossible d'imbriquer des vecteurs parallèles.

Définition 4.2.2 (*Schéma de type*)

Un schéma de type σ est la généralisation d'un type. On écrit $\sigma = \forall \alpha_1 \cdots \alpha_k \delta_1 \cdots \delta_l[\mathrm{C}].\tau$ pour dénoter l'ensemble des types obtenus par instanciation des variables $\alpha_1 \cdots \alpha_k \delta_1 \cdots \delta_l$ par des types et localités quelconques respectant la contrainte C. Par abus de notation, on assimile le type τ au schéma $\forall\emptyset[\mathsf{true}].\tau$, et on omet parfois la contrainte quand elle se résume à true.

Dans la suite, on considère les schémas de types égaux modulo α-conversion.

Définition 4.2.3 (*Contexte de typage*)

Un contexte Γ est une fonction associant des schémas de types aux variables du langage. On note $\Gamma(x)$ pour le schéma de types lié à x par Γ, et $\Gamma; x : \sigma$ pour le contexte similaire à Γ mais liant σ à x.

On écrit le jugement $\mathrm{C}, \Gamma \vdash e : \tau/\Lambda$ pour signifier que l'expression e accepte le type τ et produit l'effet Λ dans le contexte Γ et sous la contrainte C[11]. Γ indique le contexte dans lequel on se place, C renferme les propriétés vérifiées par les variables de Γ. Dans ce contexte, l'expression e correspond à une donnée de type τ, et sa réduction produit l'effet Λ : il s'ensuit que τ est préservé par réduction de e, à la différence de Λ.

Par exemple,

$$\tau \triangleleft \ell, \{f : \mathrm{int} \xrightarrow{\ell} \tau\} \vdash \mathsf{mkpar}\ f : \tau\ \mathsf{par}\ /g$$

Il est également important de noter la nuance entre types «simples» τ et types avec localité τ/Λ. Nos schémas de types ne comportent que des informations de types simples sur les variables : les effets Λ correspondent à un effet de bord d'une réduction, or seules des valeurs sont stockées dans l'environnement. On dénote ainsi le fait que les valeurs n'ont pas de localité dans notre système, et que seule l'application d'opérations peut en avoir. La localité des valeurs est en effet conséquence de la structure dans laquelle elles apparaissent et de la présence dans celle-ci de vecteurs parallèles. De la sorte, la même variable a peut apparaître localement et globalement, sans qu'il soit nécessaire de généraliser son type.

Les expressions, en revanche, produisent un effet et se typent par τ/Λ. La règle de typage VAR, sur laquelle nous anticipons légèrement, permet d'attribuer une localité arbitraire à une variable dont le schéma de type est présent dans le contexte.

[11] On emploie ici la présentation alternative pour HM(X) proposée dans [PR05], où les jugements portent sur des types et non des schémas.

Dans la suite, nous adopterons les notations et conventions suivantes :

- $\mathcal{L}(e)$ désigne les variables libres de e, définies de la façon usuelle. On définit de même $\mathcal{L}(\tau)$, $\mathcal{L}(\Lambda)$, $\mathcal{L}(\Gamma)$, $\mathcal{L}(\mathrm{C})$; notons que ces ensembles peuvent contenir des variables de type comme de localité.
- $\overline{\alpha}$ pour un ensemble de variables de types, $\overline{\delta}$ pour un ensemble de variables de localité. On écrira de la sorte un schéma de types quelconque sous la forme $\forall\overline{\alpha\delta}[\mathrm{C}].\tau$.
- La relation $\#$ dénote des ensembles disjoints : on écrira par exemple $\overline{\alpha\delta}\,\#\,E$ pour $\overline{\alpha\delta}\cap E=\emptyset$.
- Un jugement est dit valide s'il peut être dérivé par les règles de typage et si sa contrainte est satisfiable.

Interprétation des contraintes

Nous donnons ici un sens précis aux contraintes définies dans notre système. En dehors des contraintes de localité ($\tau \triangleleft \Lambda$), l'interprétation est fidèle à celles de [Smo01, SP02, PR05] pour les modèles syntaxiques, basés sur les univers de Herbrand. Le lecteur impatient peut passer cette section, et se baser uniquement sur les règles d'équivalence de la figure 4.1.

Cette interprétation est possible car les variables de localité sont *plus simples* que les variables de type usuelles (au sens où elles ne présentent aucun trait spécifique de sous-typage, abstraction, *etc.*), et peuvent donc être traitées de la même façon que celles-ci.

Définition 4.2.4 (*monotype*)

Un monotype, dans notre modèle, est un type sans variables libres (de type ou de localité).

Par souci de simplification, on notera γ dans la suite pour toute variable, de type ou de localité, et x pour un type ou une localité quelconque (soit τ ou Λ).

L'interprétation des contraintes est définie en fonction d'une assignation ρ de monotypes aux variables de types, et de monolocalités (ℓ ou g) aux variables de localité. On étend ρ aux types et aux localités de façon naturelle : de la sorte, pour tout τ, $\rho(\tau)$ est un monotype. On note $\rho \vDash \mathrm{C}$ si ρ *satisfait la contrainte* C :

$$\rho \vDash \mathsf{true} \qquad \frac{\rho \vDash \mathrm{C}_1 \qquad \rho \vDash \mathrm{C}_2}{\rho \vDash \mathrm{C}_1 \wedge \mathrm{C}_2} \qquad \frac{\rho(x_1)=\rho(x_2)}{\rho \vDash x_1 = x_2} \qquad \frac{\rho;\gamma : x \vDash \mathrm{C}}{\rho \vDash \exists\gamma.\mathrm{C}} \qquad \frac{\tau \in \dot{\tau}}{\rho \vDash \tau \triangleleft \ell} \qquad \rho \vDash \tau \triangleleft g$$

Cette interprétation nous permet de définir la relation $\Vdash$ entre contraintes.

Définition 4.2.5 (*Induction entre contraintes* $\Vdash$)

On dit que C_1 *induit* C_2*, et on écrit* $\mathrm{C}_1 \Vdash \mathrm{C}_2$ *si, et seulement si pour tout* ρ*,* $\rho \vDash \mathrm{C}_1$ *implique* $\rho \vDash \mathrm{C}_2$*.*

$\mathrm{C}_1 \Vdash \mathrm{C}_2$ indique donc que C_1 est *plus restrictive* que C_2 par rapport aux valeurs acceptées pour les variables de type et de localité.

On étend cette définition aux schémas de types en écrivant $\mathrm{C} \Vdash \forall\overline{\alpha\delta}[\mathrm{C}'].\tau$ (lire : «$\forall\overline{\alpha\delta}[\mathrm{C}'].\tau$ est cohérent par rapport à C») si $\mathrm{C} \Vdash \exists\overline{\alpha\delta}.\mathrm{C}'$.

$$\mathrm{C}_1 \wedge \mathrm{C}_2 \equiv \mathrm{C}_2 \wedge \mathrm{C}_1 \tag{4.1}$$
$$\mathrm{C}_1 \wedge (\mathrm{C}_2 \wedge \mathrm{C}_3) \equiv (\mathrm{C}_1 \wedge \mathrm{C}_2) \wedge \mathrm{C}_3 \tag{4.2}$$
$$\mathrm{C}_1 \wedge \mathrm{C}_2 \equiv \mathrm{C}_1 \quad \text{si } \mathrm{C}_1 \Vdash \mathrm{C}_2 \tag{4.3}$$
$$\gamma = x \wedge \mathrm{C} \equiv \gamma = x \wedge \{x/\gamma\}\mathrm{C} \tag{4.4}$$
$$\exists\gamma.\mathrm{C} \equiv \mathrm{C} \quad \text{si } \gamma \notin \mathcal{L}(\mathrm{C}) \tag{4.5}$$
$$\exists\gamma.(\mathrm{C}_1 \wedge \mathrm{C}_2) \equiv (\exists\gamma.\mathrm{C}_1) \wedge \mathrm{C}_2 \quad \text{si } \gamma \notin \mathcal{L}(\mathrm{C}_2) \tag{4.6}$$
$$\exists\gamma.\gamma = x \equiv \mathsf{true} \quad \text{si } \gamma \notin \mathcal{L}(x) \tag{4.7}$$
$$\exists\gamma.(\gamma = x \wedge \mathrm{C}) \equiv \{x/\gamma\}\mathrm{C} \quad \text{si } \gamma \notin \mathcal{L}(x) \tag{4.8}$$
$$\tau \lhd g \equiv \mathsf{true} \tag{4.9}$$
$$\Lambda \lhd g \equiv \mathsf{true} \tag{4.10}$$
$$\Lambda \lhd \ell \equiv \Lambda = \ell \tag{4.11}$$
$$\mathsf{Base} \lhd \Lambda \equiv \mathsf{true} \tag{4.12}$$
$$\tau_1 \xrightarrow{\Lambda'} \tau_2 \lhd \Lambda \equiv \tau_2 \lhd \Lambda' \wedge \Lambda' \lhd \Lambda \tag{4.13}$$
$$\tau_1 \times \tau_2 \lhd \Lambda \equiv \tau_1 \lhd \Lambda \wedge \tau_2 \lhd \Lambda \tag{4.14}$$
$$\tau\ \mathsf{array} \lhd \Lambda \equiv \tau \lhd \Lambda \tag{4.15}$$
$$\tau\ \mathsf{par} \lhd \Lambda \equiv \Lambda = g \tag{4.16}$$

FIG. 4.1 – Règles d'équivalence entre contraintes

Définition 4.2.6 (*équivalence de contraintes*)

$\mathrm{C}_1 \equiv \mathrm{C}_2$ *si, et seulement si* $\mathrm{C}_1 \Vdash \mathrm{C}_2$ *et* $\mathrm{C}_2 \Vdash \mathrm{C}_1$.

De façon évidente, $\Vdash$ est réflexive et transitive, et $\equiv$ est une relation d'équivalence. Le tableau d'équivalences (fig. 4.1) découle de notre interprétation des contraintes, et permettra leur réduction lors de l'inférence de types.

Les règles 4.1 à 4.4 découlent directement de la définition de $\wedge$. Les règles 4.5 à 4.8 concernent $\exists$ et découlent également de sa définition, et de la structure des contraintes. Les règles suivantes permettent de simplifier les contraintes de localité, et se déduisent directement de la structure de $\dot{\tau}$.

Des règles d'équivalence de la figure 4.1 découle informellement une simplification possible des contraintes. En particulier, toutes les règles satisfiables comportant $\lhd$ (c'est-à-dire toutes les règles ne découlant pas de HM(Herbrand) standard) peuvent se réduire sous forme de règles $\gamma \lhd \delta$ ou $\gamma \lhd \ell$ et d'égalités. Les règles 4.9 à 4.12 permettent en effet une distributivité de $\lhd\Lambda$ sur les types, et les règles 4.13 à 4.16 éliminent les cas où $\Lambda = g$ et où les terminaux sont autres que des variables de types.

Les jugements, dans la suite, sont considérés modulo équivalence des contraintes.

4.2.2 Règles de typage

La localité d'une expression dénote sa présence sur un seul processeur ou bien sur leur ensemble. Une fonction, en tant que valeur, n'a pas de localité spécifique (les fonctions sont, le plus souvent, définies globalement), mais peut être restreinte à une localité spécifique pour son exécution.

Les programmes s'exécutent dans un contexte initial Γ_0 où les types des constantes et opérateurs sont définis. Pour les opérateurs :

$$\begin{aligned}
\Gamma_0(\mathsf{mkpar}) &= \forall\alpha[\alpha \lhd \ell].(\mathrm{int} \xrightarrow{\ell} \alpha) \xrightarrow{g} \alpha\ \mathsf{par}\\
\Gamma_0(\mathsf{apply}) &= \forall\alpha\beta\delta[\beta \lhd \ell].(\alpha \xrightarrow{\ell} \beta)\ \mathsf{par} \xrightarrow{\delta} \alpha\ \mathsf{par} \xrightarrow{g} \beta\ \mathsf{par}\\
\Gamma_0(\mathsf{proj}) &= \forall\alpha\delta.\alpha\ \mathsf{par} \xrightarrow{g} \mathrm{int} \xrightarrow{\delta} \alpha\\
\Gamma_0(\mathsf{put}) &= \forall\alpha[\alpha \lhd \ell].(\mathrm{int} \xrightarrow{\ell} \alpha)\ \mathsf{par} \xrightarrow{g} (\mathrm{int} \xrightarrow{\ell} \alpha)\ \mathsf{par}\\
\Gamma_0(\mathit{send}) &= \forall\alpha[\alpha \lhd \ell].(\alpha\ \mathsf{array})\ \mathsf{par} \xrightarrow{g} (\alpha\ \mathsf{array})\ \mathsf{par}\\
\Gamma_0(\mathsf{fst}) &= \forall\alpha\beta\delta[\alpha \lhd \delta].\alpha \times \beta \xrightarrow{\delta} \alpha\\
\Gamma_0(\mathsf{snd}) &= \forall\alpha\beta\delta[\beta \lhd \delta].\alpha \times \beta \xrightarrow{\delta} \beta\\
\Gamma_0(\mathsf{nth}) &= \forall\alpha\delta\delta'[\alpha \lhd \delta'].\alpha\ \mathsf{array} \xrightarrow{\delta} \mathrm{int} \xrightarrow{\delta'} \alpha\\
\Gamma_0(\mathit{fix}) &= \forall\alpha\delta[\alpha \lhd \delta].(\alpha \xrightarrow{\delta} \alpha) \xrightarrow{\delta} \alpha
\end{aligned}$$

Les types des constantes c sont pour leur part supposés dans Base.

La localité des primitives parallèles apparaît clairement dans cette définition : elles sont toutes limitées à une exécution globale, tandis que les fonctions qu'elles prennent en argument doivent être compatibles avec une exécution locale – et donc ne doivent pas elles-mêmes exécuter de primitives. *fix* s'évalue dans la même localité que son paramètre, d'où la variable de localité δ.

Les règles du système de types sont présentées dans la figure 4.2. On utilise la définition alternative de HM(X) proposée dans [PR05], plus proche des définitions habituelles du système de Damas et Milner (DM) en ce qu'elle force à ce que la généralisation et l'instanciation ne se produisent que dans VAR et LET-IN (HM(X) permet celles-ci en tout point de l'arbre de dérivation, ce qui ne change pas le résultat de typage final). Les règles sont donc présentées à la façon de DM, avec l'ajout du traitement des contraintes d'une part, des localités d'autre part :

La règle VAR permet de typer une valeur qu'on prend dans le contexte. Les schémas de types étant définis modulo α-conversion, la règle n'a pas besoin d'inclure de renommage explicite. On notera, par ailleurs, qu'elle permet d'ajouter une contrainte C_1 quelconque au jugement et que l'instanciation peut se passer dans une localité Λ quelconque, tant que le type concerné est acceptable dans cette localité (ainsi, on ne peut par cette règle typer un vecteur parallèle dans un contexte d'exécution local). Cette règle permet, typiquement, de réutiliser des résultats obtenus par proj dans un contexte local.

LET-IN effectue une généralisation explicite des variables libres de e_1 dans e_2 en utilisant le schéma de type $\forall\overline{\alpha\delta}[\mathrm{C}_1].\tau_1$, avec $\overline{\alpha\delta} \# \mathcal{L}(\Gamma)$. Par ailleurs, elle permet de séparer la contrainte en deux parties C_1 et C_2 et de n'inclure dans le schéma de type de e_1 que les contraintes concernant

$$\textsc{var}\ \frac{\Gamma(x) = \forall\overline{\alpha\delta}[\mathrm{C}_2].\tau \qquad \mathrm{C}_1 \wedge \mathrm{C}_2 \Vdash \tau \lhd \Lambda}{\mathrm{C}_1 \wedge \mathrm{C}_2, \Gamma \vdash x : \tau/\Lambda}$$

$$\textsc{let-in}\ \frac{\mathrm{C}_2 \wedge \mathrm{C}_1, \Gamma \vdash e_1 : \tau_1/\Lambda \qquad \mathrm{C}_2 \wedge \exists\overline{\alpha\delta}.\mathrm{C}_1, (\Gamma; x : \forall\overline{\alpha\delta}[\mathrm{C}_1].\tau_1) \vdash e_2 : \tau_2/\Lambda \qquad \overline{\alpha\delta} \# \mathcal{L}(\mathrm{C}_2, \Gamma)}{\mathrm{C}_2 \wedge \exists\overline{\alpha\delta}.\mathrm{C}_1, \Gamma \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : \tau_2/\Lambda}$$

$$\textsc{deffun}\ \frac{\mathrm{C}, \Gamma; x : \tau_1 \vdash e : \tau_2/\Lambda \qquad \mathrm{C} \Vdash \Lambda \lhd \Lambda_1}{\mathrm{C}, \Gamma \vdash \mathsf{fun}\, x \to e : \tau_1 \xrightarrow{\Lambda} \tau_2/\Lambda_1}$$

$$\textsc{app}\ \frac{\mathrm{C}, \Gamma \vdash f : \tau_1 \xrightarrow{\Lambda} \tau_2/\Lambda \qquad \mathrm{C}, \Gamma \vdash e : \tau_1/\Lambda}{\mathrm{C}, \Gamma \vdash f\ e : \tau_2/\Lambda}$$

$$\textsc{pair}\ \frac{\mathrm{C}, \Gamma \vdash e_1 : \tau_1/\Lambda \qquad \mathrm{C}, \Gamma \vdash e_2 : \tau_2/\Lambda}{\mathrm{C}, \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2/\Lambda}$$

$$\textsc{ifthenelse}\ \frac{\mathrm{C}, \Gamma \vdash e : \mathrm{bool}/\Lambda \qquad \mathrm{C}, \Gamma \vdash e_1 : \tau/\Lambda \qquad \mathrm{C}, \Gamma \vdash e_2 : \tau/\Lambda}{\mathrm{C}, \Gamma \vdash \mathsf{if}\, e \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2 : \tau/\Lambda}$$

$$\textsc{array}\ \frac{\mathrm{C}, \Gamma \vdash n : \mathrm{int}/\Lambda \qquad \forall i, 0 \leq i < n, (\mathrm{C}, \Gamma \vdash e_i : \tau/\Lambda)}{\mathrm{C}, \Gamma \vdash [e_i]_i^n : \tau\ \mathsf{array}\ /\Lambda}$$

$$\textsc{vector}\ \frac{\forall i \in \mathcal{P}, (\mathrm{C}, \Gamma \vdash e_i : \tau/\ell) \qquad \mathrm{C} \Vdash \tau \lhd \ell}{\mathrm{C}, \Gamma \vdash \langle e_i \rangle_i : \tau\ \mathsf{par}\ /\Lambda}$$

$$\textsc{eqt-cstr}\ \frac{\mathrm{C}, \Gamma \vdash e : \tau/\Lambda \qquad \mathrm{C} \Vdash \tau = \tau'}{\mathrm{C}, \Gamma \vdash e : \tau'/\Lambda}$$

$$\textsc{eql-cstr}\ \frac{\mathrm{C}, \Gamma \vdash e : \tau/\Lambda \qquad \mathrm{C} \Vdash \Lambda = \Lambda'}{\mathrm{C}, \Gamma \vdash e : \tau/\Lambda'}$$

$$\textsc{ex-cstr}\ \frac{\mathrm{C}, \Gamma \vdash e : \tau/\Lambda \qquad \overline{\alpha\delta} \# \mathcal{L}(\Gamma, \tau, \Lambda)}{\exists\overline{\alpha\delta}.\mathrm{C}, \Gamma \vdash e : \tau/\Lambda}$$

FIG. 4.2 – Règles de typage

les variables qui y sont généralisées ($\overline{\alpha\delta} \# \mathcal{L}(\mathrm{C}_2)$ nous assure que celles-ci sont toutes dans C_1), ce qui simplifiera grandement l'inférence de types. Les contraintes apparaissant dans C_1 concernent, typiquement, uniquement e_1. Néanmoins, celles-ci ne seront imposées que lors d'une éventuelle instanciation future de x par VAR : la contrainte $\exists\overline{\alpha\delta}.\mathrm{C}_1$ apparaissant dans le jugement final est garante du fait qu'on ne peut définir de valeur dont le type n'aurait aucune instance.

DEFFUN mémorise la localité induite par une fonction dans le type flèche ; cette règle assure également qu'une fonction globale n'est pas créée localement (les fonctions globales pourraient être source de problèmes vis-à-vis des communications, à cause des fermetures). On ne fait aucune hypothèse quant à la localité dans laquelle sont définies les fonctions locales.

APP, par la suite, assure que les fonctions appliquées peuvent l'être dans la localité courante.

Les règles PAIR, IFTHENELSE et ARRAY ne présentent aucune particularité, elles conservent simplement les contraintes et unifient les localités de leurs sous-termes. La localité n'est en effet affectée que par les primitives, et les fonctions qui les utilisent.

La règle VECTOR n'est pas nécessaire pour typer les programmes utilisateurs, puisque ceux-ci n'ont accès aux vecteurs que par l'intermédiaire des primitives. Elle est en revanche nécessaire pour prouver la cohérence de notre typage, et sa prémisse $\mathrm{C} \Vdash \tau \lhd \ell$, en particulier, nous interdira de typer des vecteurs parallèles imbriqués : la propriété de cohérence du typage nous assurera donc de l'absence de ces derniers dans les programmes bien typés.

Le fait que la règle VECTOR n'impose pas une localité g peut paraître étrange, mais simplifie le système et ne fait perdre aucune sûreté. Les valeurs ont en effet une localité quelconque, et cela inclut des vecteurs parallèles : l'absence de vecteurs imbriqués est assurée par les mécanismes de localité et de contraintes ci-dessus.

Enfin, les trois dernières règles (EQT-CSTR, EQL-CSTR et EX-CSTR) concernent les contraintes et sont standard pour HM(X) : elles permettent, pour les deux premières, d'utiliser les contraintes d'égalité entre types et localité, respectivement, et pour la troisième de quantifier existentiellement dans les contraintes les variables libres, ce qui permet d'obtenir des jugements fermés.

On a, volontairement, omis la règle concernant la construction let rec. Cette construction étant en effet une simple syntaxe se ramenant à un appel à l'opérateur *fix*, on se contentera de la typer de la même façon que sa réécriture directe. Cette construction nous sera néanmoins nécessaire quand on voudra prouver le bon typage de l'opérateur *fix*, car elle limite l'occurence de *fix* à un contexte particulier.

Remarque : Les valeurs peuvent être typées à partir des règles VAR, DEFFUN, PAIR, ARRAY et VECTOR. Par induction triviale sur la structure de la valeur, on montre aisément que ces règles acceptent une localité arbitraire Λ : les valeurs ne produisent pas d'effet, et ne sont donc limitées en localité que par les contraintes.

Une dernière remarque par rapport au typage de la syntaxe alternative, même si celle-ci n'est pas présente dans μBSML. La réécriture de cette syntaxe vers les primitives nous donne le typage simple suivant :

$$\text{ALTSYN-VEC}\ \frac{\mathrm{C}, \Gamma \vdash e : \tau/\ell \qquad \mathrm{C} \Vdash \tau \lhd \ell}{\mathrm{C}, \Gamma \vdash \ll e \gg : \tau\ \mathsf{par}\ /g} \qquad \text{ALTSYN-OPEN}\ \frac{\mathrm{C}, \Gamma \vdash e : \tau\ \mathsf{par}\ /g}{\mathrm{C}, \Gamma \vdash \$e\$: \tau/\ell}$$

On peut considérer **this** comme le vecteur parallèle $\langle i \rangle_i$, soit $\Gamma_0(\text{this}) = \text{int}\,\mathsf{par}$.

4.2.3 Exemple

Un jugement de typage possible pour l'expression $\mathsf{mkpar}(\mathsf{fun}\, i \to i)$ dans Γ (incluant Γ_0) donné :

$$\dfrac{\dfrac{\dfrac{\Gamma(\mathsf{mkpar}) = \forall\alpha[\alpha \lhd \ell].(\text{int} \xrightarrow{\ell} \alpha) \xrightarrow{g} \alpha\,\mathsf{par} \quad \text{C} \Vdash \alpha \lhd \ell}{\text{C}, \Gamma \vdash \mathsf{mkpar} : (\text{int} \xrightarrow{\ell} \alpha) \xrightarrow{g} \alpha\,\mathsf{par}\,/g}\ \text{VAR} \quad \text{C} \Vdash \alpha = \text{int}}{\text{C}, \Gamma \vdash \mathsf{mkpar} : (\text{int} \xrightarrow{\ell} \text{int}) \xrightarrow{g} \tau'\,\mathsf{par}\,/g}\ \text{EQT-CSTR} \qquad \dfrac{\dfrac{\dfrac{(\Gamma; i : \beta)(i) = \beta \qquad \beta \lhd \ell}{\text{C}, (\Gamma; i : \beta) \vdash i : \beta/\ell}\ \text{VAR} \quad \text{C} \Vdash \beta = \text{int}}{\text{C}, (\Gamma; i : \beta) \vdash i : \text{int}/\ell}\ \text{EQT-CSTR}}{\text{C}, \Gamma \vdash (\mathsf{fun}\, i \to i) : \text{int} \xrightarrow{\ell} \text{int}/g}\ \text{DEFFUN}}{\text{C}, \Gamma \vdash \mathsf{mkpar}(\mathsf{fun}\, i \to i) : \text{int}\,\mathsf{par}\,/g}\ \text{APP}$$

Une contrainte C vérifiant ces règles est $\alpha = \text{int} \wedge \alpha \lhd \ell \wedge \beta = \text{int} \wedge \beta \lhd \ell$. On peut la simplifier en $\alpha = \text{int} \wedge \beta = \text{int}$ par équivalence, voire éliminer complètement les variables α et β par application de EX-CSTR puis équivalence et obtenir $\text{C} = \mathsf{true}$.

Cet exemple montre que l'information sur le type peut être répartie entre la contrainte et le terme τ lui-même. Cette propriété sera utilisée pour l'inférence de type, qui se base sur la détermination d'une contrainte minimale pour un jugement donné.

4.2.4 Correction

Comme défini par Milner en 1978 [Mil78], la propriété de sûreté du typage se résume à «les programmes bien typés n'échouent pas» (*«do not go wrong»*). On caractérise la notion d'échec par le fait que le programme n'est plus réductible, alors qu'il n'est pas une valeur (ou, de manière plus générale, un résultat pouvant être une valeur ou une exception, lorsqu'on considère celles-ci).

Cette caractéristique se déduit directement des deux propriétés suivantes [WF94] :

1. La préservation («subject reduction») : la réduction d'une expression préserve la propriété de bon typage.
2. Le progrès («progress») : tout programme bien typé qui n'est pas une valeur est réductible.

Nous commencerons par prouver quelques propriétés relatives au parallélisme, qui nous permettront ensuite de prouver les propriétés de preservations et de progrès, ainsi que quelques résultats annexes.

Lemme 4.2.7 (*Validité des expressions locales*)

Si $\text{C}, \Gamma \vdash e : \tau/\Lambda$, *alors* $\text{C} \Vdash \tau \lhd \Lambda$.

Ce lemme, combiné à notre règle VECTOR, nous assure la propriété de non imbrication des vecteurs parallèles.

Démonstration : On raisonne par induction sur la dérivation de e

- VAR : le résultat est explicite dans la règle de dérivation.
- LET-IN : par application de l'hypothèse d'induction sur e_2.
- DEFFUN : e a la forme $\mathsf{fun}\, x \to e'$, et est typé par

$$\frac{\mathrm{C}, (\Gamma; x : \tau_1) \vdash e' : \tau_2/\Lambda' \qquad \mathrm{C} \Vdash \Lambda' \lhd \Lambda}{\mathrm{C}, \Gamma \vdash \mathsf{fun}\, x \to e' : \tau_1 \xrightarrow{\Lambda'} \tau_2/\Lambda} \text{DEFFUN}$$

  Par hypothèse d'induction sur e', on a $\mathrm{C} \Vdash \tau_2 \lhd \Lambda'$. On en déduit $\mathrm{C} \Vdash \Lambda' \lhd \Lambda \wedge \tau_2 \lhd \Lambda$, et donc $\mathrm{C} \Vdash \tau_1 \xrightarrow{\Lambda'} \tau_2 \lhd \Lambda$.
- APP : on pose $e = f\, e'$, et on a $\mathrm{C}, \Gamma \vdash f : \tau' \xrightarrow{\Lambda} \tau/\Lambda$. Par hypothèse d'induction, $\mathrm{C} \Vdash \tau' \xrightarrow{\Lambda} \tau \lhd \Lambda$, et donc $\mathrm{C} \Vdash \tau \lhd \Lambda$ par définition.
- Les autres cas se résolvent par induction directe. ■

Préservation

Quelques définitions annexes vont nous être nécessaires :

Définition 4.2.8 (*Compatibilité du typage* $\sqsubseteq$)

La relation $\sqsubseteq$ est définie pour toutes expressions e_1, e_2 comme $e_1 \sqsubseteq e_2$ si, et seulement si pour tous $\mathrm{C}, \Gamma, \tau, \Lambda$, on a

$$\mathrm{C}, \Gamma \vdash e_1 : \tau/\Lambda \;\Rightarrow\; \mathrm{C}, \Gamma \vdash e_2 : \tau/\Lambda$$

$e_1 \sqsubseteq e_2$ signifie donc que tous types et localités acceptés par e_1 le sont également par e_2. Cette relation nous permet de définir la propriété de préservation :

Définition 4.2.9 (*Préservation*)

Notre système de types a la propriété de préservation par rapport à la réduction $\rightarrowtail$ si $e_1 \rightarrowtail e_2$ implique $e_1 \sqsubseteq e_2$.

Il est possible, d'après la définition de $\sqsubseteq$, que la réduction donne un type / localité plus général que le type d'origine. C'est particulièrement visible vis-à-vis du parallélisme : l'expression $e = \mathsf{proj}\,(\mathsf{mkpar}\, id)\, 0$, par exemple, se type $e : \mathrm{int}/g$, mais en la réduisant on perd le parallélisme : $0 : \mathrm{int}/\Lambda$, avec Λ quelconque. L'important est que la relation «est bien typé» soit préservée.

Trois lemmes nous sont nécessaires pour prouver cette propriété :
- la congruence de $\sqsubseteq$ par rapport aux contextes d'évaluation,
- l'affaiblissement des jugements, qui est standard en HM(X),
- et le lemme de substitution qui sera au cœur des cas de APPLY et LET-IN pour la preuve de préservation.

Lemme 4.2.10 (*Congruence de* $\sqsubseteq$)

Si $e_1 \sqsubseteq e_2$, alors $E[e_1] \sqsubseteq E[e_2]$

Démonstration : Par induction sur la structure de E, et application des règles de typage sur chacun des cas.

■

Lemme 4.2.11 (*Affaiblissement*)

Si $\mathrm{C}' \Vdash \mathrm{C}$, *alors* $\mathrm{C}, \Gamma \vdash e : \tau/\Lambda$ *implique* $\mathrm{C}', \Gamma \vdash e : \tau/\Lambda$.

En d'autres mots, un jugement reste valide si on renforce sa contrainte (ce qui affaiblit le jugement lui-même).

Démonstration : Par induction sur la dérivation du jugement $\mathrm{C}, \Gamma \vdash e : \tau/\Lambda$.

Dans le cas de la règle VAR, on pose $\mathrm{C} = \mathrm{C}_1 \wedge \mathrm{C}_2$. Comme le choix de C_1 dans la règle est arbitraire, on peut la remplacer par C'. On obtient $\mathrm{C}' \wedge \mathrm{C}_2, \Gamma \vdash e : \tau/\Lambda$, or $\mathrm{C}' \Vdash \mathrm{C}_1 \wedge \mathrm{C}_2$, implique $\mathrm{C}' \Vdash \mathrm{C}_2$ et $\mathrm{C}' \wedge \mathrm{C}_2 \equiv \mathrm{C}'$.

Dans le cas de LET-IN, on a $\mathrm{C} = \mathrm{C}_2 \wedge \exists\overline{\alpha\delta}.\mathrm{C}_1$. Comme $\mathrm{C}' \wedge \mathrm{C}_2 \wedge \mathrm{C}_1 \Vdash \mathrm{C}_2 \wedge \mathrm{C}_1$ et qu'on peut supposer $\overline{\alpha\delta} \# \mathcal{L}(\mathrm{C}')$, en renommant au besoin, on peut conclure de même par hypothèse d'induction en remplaçant C_2 par $\mathrm{C}' \wedge \mathrm{C}_2$ dans la dérivation.

Les cas restants sont résolus de façon immédiate par hypothèse d'induction. ■

Lemme 4.2.12 (*Substitution*)

Sous réserve des hypothèses suivantes :
- $(\mathrm{C}_2 \wedge \exists\overline{\alpha\delta}.\mathrm{C}_1), (\Gamma; x : \forall\overline{\alpha\delta}[\mathrm{C}_1].\tau_1) \vdash e : \tau_2/\Lambda$
- $\mathrm{C}_2 \wedge \mathrm{C}_1, \Gamma \vdash v : \tau_1/\Lambda$
- $\overline{\alpha\delta} \# \mathcal{L}(\mathrm{C}_2, \Gamma, \Lambda)$

on a $(\mathrm{C}_2 \wedge \exists\overline{\alpha\delta}.\mathrm{C}_1), \Gamma \vdash \{v/x\}e : \tau_2/\Lambda$.

Démonstration : On pose $\Gamma_x = \Gamma; x : \forall\overline{\alpha\delta}[\mathrm{C}_1].\tau_1$ et $\mathrm{C} = \mathrm{C}_2 \wedge \exists\overline{\alpha\delta}.\mathrm{C}_1$. Notre première hypothèse s'écrit de la sorte $\mathrm{C}, \Gamma_x \vdash e : \tau_2/\Lambda$. On raisonne par induction sur la dérivation de ce jugement.

Cas de VAR : on pose $e = x'$, et la dérivation est de la forme suivante, avec $\mathrm{C} = \mathrm{C}'_0 \wedge \mathrm{C}_0$:

$$\frac{\Gamma_x(x') = \forall\overline{\alpha'\delta'}[\mathrm{C}_0].\tau_2 \qquad \mathrm{C} \Vdash \tau_2 \triangleleft \Lambda}{\mathrm{C}, \Gamma_x \vdash x' : \tau_2/\Lambda}$$

Si $x' \neq x$, alors $\{v/x\}e = e$ et $\Gamma_x(x') = \Gamma(x')$, on obtient donc le résultat en remplaçant Γ_x par Γ dans la dérivation ci-dessus.

Si au contraire $x = x'$, $\{v/x\}e = v$, et donc par hypothèse $\mathrm{C}_2 \wedge \mathrm{C}_1, \Gamma \vdash \{v/x\}e : \tau_1/\Lambda$.

On a de plus $\Gamma_x(x) = \forall\overline{\alpha\delta}[\mathrm{C}_1].\tau_1 = \forall\overline{\alpha'\delta'}[\mathrm{C}_0].\tau_2$. Par conséquent, τ_1 et τ_2 sont égaux modulo renommage de $\overline{\alpha\delta}$; on peut effectuer ce renommage de $\overline{\alpha\delta}$ en $\overline{\alpha'\delta'}$ dans toutes nos hypothèses, seule $\mathrm{C}_2 \wedge \mathrm{C}_1, \Gamma \vdash \{v/x\}e : \tau_1/\Lambda$ est changée en $\mathrm{C}_2 \wedge \mathrm{C}_0, \Gamma \vdash \{v/x\}e : \tau_2/\Lambda$ ($\overline{\alpha\delta}$ sont quantifiées dans les termes où elles apparaissent dans les autres hypothèses et notre conclusion).

On peut en outre librement supposer que $\overline{\alpha\delta}$ et $\overline{\alpha'\delta'}$ sont disjoints. On en déduit, par EX-CSTR, que $\exists\overline{\alpha\delta}.(\mathrm{C}_2 \wedge \mathrm{C}_0), \Gamma \vdash \{v/x\}e : \tau_2/\Lambda$. Comme $\overline{\alpha\delta} \# \mathrm{C}_2$, on a en outre l'équivalence $\exists\overline{\alpha\delta}.(\mathrm{C}_2 \wedge \mathrm{C}_1) \equiv \mathrm{C}_2 \wedge \exists\overline{\alpha\delta}.\mathrm{C}_1$, ce qui nous donne le résultat.

Cas de LET-IN : on pose $e = (\mathsf{let}\, x' = e_1 \,\mathsf{in}\, e_2)$, et la dérivation est de la forme suivante, avec $\mathrm{C} = \mathrm{C}'_2 \wedge \exists\overline{\alpha'\delta'}.\mathrm{C}'_1$:

$$\frac{\mathrm{C}'_2 \wedge \mathrm{C}'_1, \Gamma_x \vdash e_1 : \tau'/\Lambda \qquad \mathrm{C}, (\Gamma_x; x' : \forall\overline{\alpha'\delta'}[\mathrm{C}'_1].\tau') \vdash e_2 : \tau_2/\Lambda \qquad \overline{\alpha'\delta'} \# \mathcal{L}(\mathrm{C}'_2, \Gamma_x)}{\mathrm{C}, \Gamma_x \vdash \mathsf{let}\, x' = e_1 \,\mathsf{in}\, e_2 : \tau_2/\Lambda}$$

On a clairement $\mathrm{C}'_2 \wedge \mathrm{C}'_1 \Vdash \mathrm{C}$, donc, en utilisant le lemme d'affaiblissement sur les hypothèses et l'hypothèse d'induction sur la première prémisse, on a $\mathrm{C}'_2 \wedge \mathrm{C}'_1, \Gamma \vdash \{v/x\}e_1 : \tau'/\Lambda$.

Si $x = x'$, on a $(\Gamma_x; x' : \forall\overline{\alpha'\delta'}[\mathrm{C}'_1].\tau') = (\Gamma; x' : \forall\overline{\alpha'\delta'}[\mathrm{C}'_1].\tau')$, et par conséquent $\mathrm{C}, (\Gamma; x' : \forall\overline{\alpha'\delta'}[\mathrm{C}'_1].\tau') \vdash e_2 : \tau_2/\Lambda$. Par application de la règle LET-IN, on obtient ainsi $\mathrm{C}, \Gamma \vdash \mathsf{let}\, x' = \{v/x\}e_1 \,\mathsf{in}\, e_2 : \tau_2/\Lambda$ et par définition de la substitution, $\mathrm{C}, \Gamma \vdash \{v/x\}(\mathsf{let}\, x' = e_1 \,\mathsf{in}\, e_2) : \tau_2/\Lambda$.

Supposons maintenant $x \neq x'$. On a alors $(\Gamma_x; x' : \forall\overline{\alpha'\delta'}[\mathrm{C}'_1].\tau') = (\Gamma; x' : \forall\overline{\alpha'\delta'}[\mathrm{C}'_1].\tau'; x : \forall\overline{\alpha\delta}[\mathrm{C}_1].\tau_1)$, et par hypothèse d'induction, $\mathrm{C}, (\Gamma; x' : \forall\overline{\alpha'\delta'}[\mathrm{C}'_1].\tau') \vdash \{v/x\}e_2 : \tau_2/\Lambda$. De nouveau, par application de la règle de typage LET-IN, on obtient le jugement $\mathrm{C}, \Gamma \vdash \mathsf{let}\, x' = \{v/x\}e_1 \,\mathsf{in}\{v/x\}e_2 : \tau_2/\Lambda$, et par conséquent $\mathrm{C}, \Gamma \vdash \{v/x\}(\mathsf{let}\, x' = e_1 \,\mathsf{in}\, e_2) : \tau_2/\Lambda$.

Cas de DEFFUN : on pose $e = \mathsf{fun}\, x' \to e'$, et on obtient, avec $\tau_2 = \tau'_1 \xrightarrow{\Lambda'} \tau'_2$:

$$\frac{\mathrm{C}, (\Gamma_x; x' : \tau'_1) \vdash e' : \tau'_2/\Lambda' \qquad \mathrm{C} \Vdash \Lambda' \triangleleft \Lambda}{\mathrm{C}, \Gamma_x \vdash \mathsf{fun}\, x' \to e' : \tau'_1 \xrightarrow{\Lambda'} \tau'_2/\Lambda}$$

On suppose que $x' \neq x$, et utilise la même manipulation que pour LET-IN : on a $(\Gamma_x; x' : \tau'_1) = (\Gamma; x' : \tau'_1; x : \forall\overline{\alpha\delta}[\mathrm{C}_1].\tau_1)$, et on peut donc appliquer l'hypothèse d'induction sur la prémisse et obtenir $\mathrm{C}, (\Gamma; x' : \tau'_1) \vdash \{v/x\}e' : \tau'_2$. Le résultat suit par application de DEFFUN.

Si $x = x'$, $(\Gamma_x; x' : \tau'_1) = (\Gamma; x' : \tau'_1)$ et $\{v/x\}(\mathsf{fun}\, x' \to e') = \mathsf{fun}\, x' \to e'$, on conclut donc par application directe de DEFFUN.

Autres cas : les règles restantes, ne jouant pas sur le contexte Γ, se résolvent par induction directe. ∎

Propriété 4.2.13 (*Préservation dans notre système de types*)

*Notre système de types pour μ*BSML *a la propriété de préservation :*

$$e_1 \rightarrowtail e_2 \Rightarrow e_1 \sqsubseteq e_2$$

Démonstration : On suppose $e^1 \rightarrowtail e^2$, et $\mathrm{C}, \Gamma \vdash e^1 : \tau/\Lambda$. Il s'agit donc de prouver que le jugement $\mathrm{C}, \Gamma \vdash e^2 : \tau/\Lambda$ est valide.

Pour cela, on raisonne par cas sur la règle utilisée pour la réduction $e^1 \rightarrowtail e^2$. On commencera par étudier les règles de réduction de tête générales, puis les δ-règles afin de prouver la cohérence de nos opérateurs (et en particulier des primitives parallèles), et enfin les règles de réduction par contexte.

Règles de réduction de tête

Cas de APP : $e^1 = (\mathsf{fun}\, x \to e)\ v$, $e^2 = \{v/x\}e$. e^1 est typé par dérivation suivante :

$$\frac{\dfrac{\mathrm{C}, (\Gamma; x : \tau') \vdash e : \tau/\Lambda}{\mathrm{C}, \Gamma \vdash \mathsf{fun}\, x \to e : \tau' \xrightarrow{\Lambda} \tau/\Lambda}\,\text{DEFFUN} \qquad \mathrm{C}, \Gamma \vdash v : \tau'/\Lambda}{\mathrm{C}, \Gamma \vdash (\mathsf{fun}\, x \to e)\ v : \tau/\Lambda}\,\text{APP}$$

On conclut par application directe du lemme de substitution, avec $\overline{\alpha\delta} = \emptyset$ et $\mathrm{C}_1 = \mathsf{true}$.

Cas de LET-IN : $e^1 = \mathsf{let}\, x = v\, \mathsf{in}\, e$, $e^2 = \{v/x\}e$. e^1 is typed by the following derivation :

$$\frac{(\mathrm{C}_2 \wedge \mathrm{C}_1), \Gamma \vdash v : \tau/\Lambda \qquad (\mathrm{C}_2 \wedge \exists\overline{\alpha\delta}\mathrm{C}_1), (\Gamma; x : \forall\overline{\alpha\delta}[\mathrm{C}_1].\tau') \vdash e : \tau/\Lambda \qquad \overline{\alpha\delta} \# \mathcal{L}(\mathrm{C}_2, \Gamma)}{(\mathrm{C}_2 \wedge \exists\overline{\alpha\delta}\mathrm{C}_1), \Gamma \vdash \mathsf{let}\, x = v\, \mathsf{in}\, e : \tau/\Lambda}\ \text{LET-IN}$$

Les prémisses de la règle nous permettent d'appliquer directement le lemme de substitution et de conclure.

Cas de IFTHEN **et** IFELSE

$e^1 = \mathsf{if}\, v\, \mathsf{then}\, e\, \mathsf{else}\, e'$

$$\frac{\Gamma \vdash v : \mathrm{bool}[\mathrm{C}]/\Lambda \qquad \Gamma \vdash e : \tau[\mathrm{C}]/\Lambda \qquad \Gamma \vdash e' : \tau[\mathrm{C}]/\Lambda}{\Gamma \vdash \mathsf{if}\, v\, \mathsf{then}\, e\, \mathsf{else}\, e' : \tau[\mathrm{C}]/\Lambda}\ \text{IFTHENELSE}$$

On peut directement conclure.

δ-règles :

Cas de mkpar : le type de $\mathsf{mkpar}\, v$ est déduit par l'arbre suivant :

$$\dfrac{\dfrac{\dfrac{\Gamma(\mathsf{mkpar}) = \forall\alpha[\alpha \triangleleft \ell].(\mathrm{int} \xrightarrow{\ell} \alpha) \xrightarrow{g} \alpha\, \mathsf{par} \qquad \mathrm{C} \Vdash \alpha \triangleleft \ell}{\mathrm{C}, \Gamma \vdash \mathsf{mkpar} : (\mathrm{int} \xrightarrow{\ell} \alpha) \xrightarrow{g} \alpha\, \mathsf{par}\, /g}\ \text{VAR} \qquad \mathrm{C} \Vdash \alpha = \tau'}{\mathrm{C}, \Gamma \vdash \mathsf{mkpar} : (\mathrm{int} \xrightarrow{\ell} \tau') \xrightarrow{g} \tau'\, \mathsf{par}\, /g}\ \text{EQT-CSTR} \qquad \mathrm{C}, \Gamma \vdash v : \mathrm{int} \xrightarrow{\ell} \tau'/g}{\mathrm{C}, \Gamma \vdash \mathsf{mkpar}\, v : \tau'\, \mathsf{par}\, /g}\ \text{APP}$$

ainsi, τ a ici la forme $\tau'\, \mathsf{par}$ et $\Lambda = g$.

$\mathsf{mkpar}\, v$ se réduit en $\langle v\, i \rangle_i$. v étant une valeur, on a également $\mathrm{C}, \Gamma \vdash v : \mathrm{int} \xrightarrow{\ell} \tau'/\ell$. De plus, à partir de $\mathrm{C} \Vdash \alpha = \tau'$ et $\mathrm{C} \Vdash \alpha \triangleleft \ell$, on déduit $\mathrm{C} \Vdash \tau' \triangleleft \ell$, ce qui nous permet d'utiliser la règle VECTOR :

$$\frac{\forall i \in \mathcal{P}, \quad \dfrac{\mathrm{C}, \Gamma \vdash v : \mathrm{int} \xrightarrow{\ell} \tau'/\ell \qquad \mathrm{C}, \Gamma \vdash i : \mathrm{int}/\ell}{\mathrm{C}, \Gamma \vdash v\ i : \tau'/\ell}\ \text{APP} \qquad \mathrm{C} \Vdash \tau' \triangleleft \ell}{\mathrm{C}, \Gamma \vdash \langle v\ i \rangle_i : \tau'\, \mathsf{par}\, /\Lambda}\ \text{VECTOR}$$

La réduction de tête de mkpar préserve donc le type.

Cas de apply : on suit une approche similaire pour les applications partielles et totales, et on pose $\mathrm{C}, \Gamma \vdash \mathsf{apply}\, v^1 : \tau/\Lambda$

$$\dfrac{\dfrac{\Gamma(\mathsf{apply}) = \forall\alpha\beta[\beta \triangleleft \ell].(\alpha \xrightarrow{\ell} \beta)\, \mathsf{par} \xrightarrow{\delta} \alpha\, \mathsf{par} \xrightarrow{g} \beta\, \mathsf{par} \qquad \mathrm{C} \Vdash \tau_2 \triangleleft \ell}{\mathrm{C}, \Gamma \vdash \mathsf{apply} : (\tau_1 \xrightarrow{\ell} \tau_2)\, \mathsf{par} \xrightarrow{\Lambda} \tau_1\, \mathsf{par} \xrightarrow{g} \tau_2\, \mathsf{par}\, /\Lambda}\ \text{VAR'} \qquad \mathrm{C}, \Gamma \vdash v^1 : (\tau_1 \xrightarrow{\ell} \tau_2)\, \mathsf{par}\, /\Lambda}{\mathrm{C}, \Gamma \vdash \mathsf{apply}\, v^1 : \tau_1\, \mathsf{par} \xrightarrow{g} \tau_2\, \mathsf{par}\, /\Lambda}\ \text{APP}$$

Par souci de clarté, on omet dans la suite les contraintes d'égalité pour effectuer la substitution correspondante directement en notant la règle VAR', ce qui correspond à une combinaison de VAR et EQT-CSTR.

D'après la dérivation ci-dessus, τ a la forme $\tau_1\,\mathsf{par} \xrightarrow{g} \tau_2\,\mathsf{par}$. On peut ainsi typer l'application $\mathsf{apply}\,v^1\,v^2$:

$$\frac{\mathrm{C}, \Gamma \vdash \mathsf{apply}\,v^1 : \tau_1\,\mathsf{par} \xrightarrow{g} \tau_2\,\mathsf{par}\,/g \qquad \mathrm{C}, \Gamma \vdash v^2 : \tau_1\,\mathsf{par}\,/g}{\mathrm{C}, \Gamma \vdash \mathsf{apply}\,v^1\ v^2 : \tau_2\,\mathsf{par}\,/g}\ \text{APP}$$

D'où l'on déduit que v^1 et v^2 sont des vecteurs. On pose par conséquent $v^1 = \langle v_i^1 \rangle_i$ and $v^2 = \langle v_i^2 \rangle_i$. On ne peut typer ces vecteurs que par la règle VECTOR, ce qui nous donne les types des composantes de v^1 et v^2 :

$$\frac{\forall i \in \mathcal{P}, (\mathrm{C}, \Gamma \vdash v_i^1 : \tau_1 \xrightarrow{\ell} \tau_2/\ell) \qquad \mathrm{C} \Vdash \tau_1 \xrightarrow{\ell} \tau_2 \triangleleft \ell}{\mathrm{C}, \Gamma \vdash v^1 : (\tau_1 \xrightarrow{\ell} \tau_2)\,\mathsf{par}\,/\Lambda}\ \text{VECTOR}$$

$$\frac{\forall i \in \mathcal{P}, \mathrm{C}, \Gamma \vdash v_i^2 : \tau_1/\ell \qquad \mathrm{C} \Vdash \tau_1 \triangleleft \ell}{\mathrm{C}, \Gamma \vdash v^2 : \tau_1\,\mathsf{par}\,/g}\ \text{VECTOR}$$

On peut alors typer la réduction de tête de $\mathsf{apply}\,v^1\,v^2$ dans Γ :

$$\frac{\forall i \in \mathcal{P}, \quad \dfrac{\mathrm{C}, \Gamma \vdash v_i^1 : \tau_1 \xrightarrow{\ell} \tau_2/\ell \qquad \mathrm{C}, \Gamma \vdash v_i^2 : \tau_1/\ell}{\mathrm{C}, \Gamma \vdash v_i^1\ v_i^2 : \tau_2/\ell}\ \text{APP} \qquad \mathrm{C} \Vdash \tau_2 \triangleleft \ell}{\mathrm{C}, \Gamma \vdash \langle v_i^1\ v_i^2 \rangle_i : \tau_2\,\mathsf{par}\,/g}\ \text{VECTOR}$$

Ainsi, la δ-règle pour apply préserve le typage.

Cas de proj : on pose $\mathrm{C}, \Gamma \vdash \mathsf{proj}\,v : \tau/\Lambda$. Comme précédemment, v est nécessairement un vecteur et on peut donc poser $v = \langle v_i \rangle_i$

$$\frac{\dfrac{\Gamma(\mathsf{proj}) = \forall\alpha\delta.\alpha\,\mathsf{par} \xrightarrow{g} \mathrm{int} \xrightarrow{\delta} \alpha}{\mathrm{C}, \Gamma \vdash \mathsf{proj} : \tau'\,\mathsf{par} \xrightarrow{g} \mathrm{int} \xrightarrow{\Lambda} \tau'/g}\ \text{VAR'} \qquad \dfrac{\forall i \in \mathcal{P}, (\mathrm{C}, \Gamma \vdash v_i : \tau'/\ell) \qquad \mathrm{C} \Vdash \tau' \triangleleft \ell}{\mathrm{C}, \Gamma \vdash v : \tau'\,\mathsf{par}\,/g}\ \text{VECTOR}}{\mathrm{C}, \Gamma \vdash \mathsf{proj}\,v : \mathrm{int} \xrightarrow{\Lambda} \tau'/g}\ \text{APP}$$

Ce qui nous donne le type des v_i, à partir duquel on peut typer la réduction de tête de proj (en utilisant le fait que $\tau' \triangleleft \ell \Vdash \tau' \triangleleft \Lambda$) :

$$\frac{\dfrac{\Gamma(\mathsf{nth}) = \forall\alpha\delta\delta'[\alpha \triangleleft \delta'].\alpha\,\mathsf{array} \xrightarrow{\delta} \mathrm{int} \xrightarrow{\delta'} \alpha \quad \mathrm{C} \Vdash \tau' \triangleleft \Lambda}{\mathrm{C}, \Gamma \vdash \mathsf{nth} : \tau'\,\mathsf{array} \xrightarrow{g} \mathrm{int} \xrightarrow{\Lambda} \tau'/g}\ \text{VAR'} \qquad \dfrac{\mathrm{C}, \Gamma \vdash p : \mathrm{int}/g \quad \forall i \in \mathcal{P}, (\mathrm{C}, \Gamma \vdash v_i : \tau'/g)}{\mathrm{C}, \Gamma \vdash [v_i]_i^p : \tau'\,\mathsf{array}\,/g}\ \text{ARRAY}}{\mathrm{C}, \Gamma \vdash \mathsf{nth}\,[v_i]_i^p : \mathrm{int} \xrightarrow{\Lambda} \tau'/g}\ \text{APP}$$

Cette règle nous permet de prendre des valeurs dans l'espace local pour les replacer dans l'espace global, en changeant la localité de v_i.

Cas de *send*. On pose $\mathrm{C}, \Gamma \vdash \mathit{send}\, v : \tau/\Lambda$, et $v = \langle v_i \rangle_i$ comme précédemment. Comme les v_i sont des tableaux, on écrit $v_i = \left[v_i^j\right]_j^p$, en faisant l'hypothèse qu'ils ont tous la taille p : cela est vérifié par le seul usage de *send*, qui est fait dans la définition de put – *send* ne faisant pas partie du langage programmeur.

$$\dfrac{\dfrac{\Gamma(\mathit{send}) = \forall\alpha[\alpha \triangleleft \ell].\alpha\ \mathsf{array\ par} \xrightarrow{g} \alpha\ \mathsf{array\ par} \quad \mathrm{C} \Vdash \tau' \triangleleft \ell}{\mathrm{C}, \Gamma \vdash \mathit{send} : \tau'\ \mathsf{array\ par} \xrightarrow{g} \tau'\ \mathsf{array\ par}\ /g}\ \text{VAR'} \qquad \dfrac{\dfrac{\forall j < p, (\mathrm{C}, \Gamma \vdash v_i^j : \tau'/\ell)}{\forall i \in \mathcal{P}, (\mathrm{C}, \Gamma \vdash v_i : \tau'\ \mathsf{array}\ /\ell)}\ \text{ARRAY} \quad \mathrm{C} \Vdash \tau'\ \mathsf{array} \triangleleft \ell}{\mathrm{C}, \Gamma \vdash v : \tau'\ \mathsf{array\ par}\ /g}\ \text{VECTOR}}{\mathrm{C}, \Gamma \vdash \mathit{send}\, v : \tau'\ \mathsf{array\ par}\ /g}\ \text{APP}$$

On peut typer la réduction de $\mathit{send}\, v$:

$$\dfrac{\forall j \in \mathcal{P} \qquad \dfrac{\forall i < p, (\mathrm{C}, \Gamma \vdash v_i^j : \tau'/\ell)}{\mathrm{C}, \Gamma \vdash \left[v_i^j\right]_i^p : \tau'\ \mathsf{array}\ /\ell}\ \text{ARRAY} \qquad \mathrm{C} \Vdash \tau'\ \mathsf{array} \triangleleft \ell}{\mathrm{C}, \Gamma \vdash \left\langle \left[v_i^j\right]_i^p \right\rangle_j : \tau'\ \mathsf{array\ par}\ /g}\ \text{VECTOR}$$

Cas de put. On pose $\mathrm{C}, \Gamma \vdash \mathsf{put}\, v : \tau/\Lambda$, $v = \langle v_i \rangle_i$

$$\dfrac{\dfrac{\Gamma(\mathsf{put}) = \forall\alpha[\alpha \triangleleft \ell].(\mathrm{int} \xrightarrow{\ell} \alpha)\ \mathsf{par} \xrightarrow{g} (\mathrm{int} \xrightarrow{\ell} \alpha)\ \mathsf{par} \quad \mathrm{C} \Vdash \tau' \triangleleft \ell}{\mathrm{C}, \Gamma \vdash \mathsf{put} : (\mathrm{int} \xrightarrow{\ell} \tau')\ \mathsf{par} \xrightarrow{g} (\mathrm{int} \xrightarrow{\ell} \tau')\ \mathsf{par}\ /g}\ \text{VAR'} \qquad \dfrac{\dfrac{\forall i \in \mathcal{P}, (\mathrm{C}, \Gamma \vdash v_i : \mathrm{int} \xrightarrow{\ell} \tau'/\ell)}{\mathrm{C} \Vdash \mathrm{int} \xrightarrow{\ell} \tau' \triangleleft \ell}}{\mathrm{C}, \Gamma \vdash v : (\mathrm{int} \xrightarrow{\ell} \tau')\ \mathsf{par}\ /g}\ \text{VECTOR}}{\mathrm{C}, \Gamma \vdash \mathsf{put}\, v : (\mathrm{int} \xrightarrow{\ell} \tau')\ \mathsf{par}\ /g}\ \text{APP}$$

On déduit maintenant le type de la réduction de $\mathsf{put}\, v$ depuis le type des v_i, Γ et C_0.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{C}, \Gamma \vdash v_i : \mathrm{int} \xrightarrow{\ell} \tau'/\ell \quad \dfrac{\mathrm{C}, \Gamma \vdash j : \mathrm{int}/\ell}{\mathrm{C}, \Gamma \vdash v_i\ j : \tau'/\ell}\ \text{APP}}{\forall j < p, \quad \mathrm{C}, \Gamma \vdash [v_i\ j]_j^p : \tau'\ \mathsf{array}\ /\ell}\ \text{ARRAY} \qquad \mathrm{C} \Vdash \tau'\ \mathsf{array} \triangleleft \ell}{\forall i \in \mathcal{P}, \quad \mathrm{C}, \Gamma \vdash \left\langle [v_i\ j]_j^p \right\rangle_i : \tau'\ \mathsf{array\ par}\ /g}\ \text{VECTOR}}{\mathrm{C}, \Gamma \vdash \mathit{send} : \tau'\ \mathsf{array\ par} \xrightarrow{g} \tau'\ \mathsf{array\ par}\ /g}\ \text{APP}}{\mathrm{C}, \Gamma \vdash \mathit{send} \left\langle [v_i\ j]_j^p \right\rangle_i : \tau'\ \mathsf{array\ par}\ /g}\ \text{APP}}{\mathrm{C}, \Gamma \vdash \mathsf{apply}\, \langle \mathsf{nth} \rangle_i : \tau'\ \mathsf{array\ par} \xrightarrow{g} (\mathrm{int} \xrightarrow{\Lambda} \tau')\ \mathsf{par}\ /g}\ \text{APP}}{\mathrm{C}, \Gamma \vdash \mathsf{apply}\, \langle \mathsf{nth} \rangle_i\, (\mathit{send} \left\langle [v_i\ j]_j^p \right\rangle_i) : (\mathrm{int} \xrightarrow{\Lambda} \tau')\ \mathsf{par}\ /g}\ \text{APP}$$

Opérateurs non parallèles. Les cas des opérateurs $\mathsf{fst}, \mathsf{snd}, \mathsf{length}, \mathsf{nth}$ sont résolus trivialement. Le cas de *fix* est plus intéressant : si $\Gamma \vdash \mathit{fix}\, v : \tau/\Lambda$,

$$\dfrac{\dfrac{\Gamma(\mathit{fix}) = \forall\alpha\delta[\alpha \lhd \delta].(\alpha \xrightarrow{\delta} \alpha) \xrightarrow{\delta} \alpha \qquad \mathrm{C} \Vdash \tau \lhd \Lambda}{\mathrm{C}, \Gamma \vdash \mathit{fix} : (\tau \xrightarrow{\Lambda} \tau) \xrightarrow{\Lambda} \tau/\Lambda}\ \textsc{var'} \qquad \mathrm{C}, \Gamma \vdash v : \tau \xrightarrow{\Lambda} \tau/\Lambda}{\mathrm{C}, \Gamma \vdash \mathit{fix}\, v : \tau/\Lambda}\ \textsc{app}$$

Comme *fix* ne peut être introduit dans une expression que par l'usage de let rec, on a la garantie que v a la forme $\mathsf{fun}\, x \to e$ et qu'il ne s'agit pas d'un opérateur. On peut donc le typer :

$$\dfrac{\mathrm{C}, (\Gamma; x : \tau) \vdash e : \tau_2/\Lambda}{\mathrm{C}, \Gamma \vdash \mathsf{fun}\, x \to e : \tau \xrightarrow{\Lambda} \tau/\Lambda}\ \textsc{deffun}$$

La réduction de tête de $\mathit{fix}(\mathsf{fun}\, x \to e)$ est $\{\mathit{fix}(\mathsf{fun}\, x \to e)/x\}e$. On applique le lemme de substitution avec $\overline{\alpha\delta} = \emptyset$ pour conclure.

Règles de réduction générales

Il nous reste à montrer la propriété pour la réduction par CONTEXT et $\textsc{local}_i$. Pour CONTEXT, elle découle directement de la congruence de $\sqsubseteq$. Pour $\textsc{local}_i$:

$$\forall i \in \mathcal{P}, \dfrac{e_1 \rightarrowtail^{\epsilon} e_2}{\mathrm{E}_g\left[\left\langle e'_0 \dots e'_{i-1}, \mathrm{E}_\ell[e_1], e'_{i+1} \dots e'_{p-1}\right\rangle\right] \rightarrowtail \mathrm{E}_g\left[\left\langle e'_0 \dots e'_{i-1}, \mathrm{E}_\ell[e_2], e'_{i+1} \dots e'_{p-1}\right\rangle\right]}\ \textsc{local}_i$$

Soit $e = \left\langle e'_0 \dots e'_{i-1}, \mathrm{E}_\ell[e_1], e'_{i+1} \dots e'_{p-1}\right\rangle$. On suppose $\mathrm{C}, \Gamma \vdash e : \tau/\Lambda$, et e étant typée par VECTOR, on déduit $\mathrm{C}, \Gamma \vdash \mathrm{E}_\ell(e_1) : \tau'/\ell$ et $\mathrm{C} \Vdash \tau' \lhd \ell$. On en déduit, par congruence, $\mathrm{C}, \Gamma \vdash \mathrm{E}_\ell[e_2] : \tau'/\ell$. Le résultat suit d'une nouvelle application de VECTOR et de la congruence.

Notons que l'on ne peut typer des vecteurs parallèles emboîtés par VECTOR : c'est en accord avec le fait qu'aucune règle de réduction ne permet de réduire de tels vecteurs, qui seraient de fait des expressions bloquées. ■

Progrès

Propriété 4.2.14 (*Progrès*)

Tout programme bien typé est soit une valeur, soit réductible.

Démonstration : On suppose la localité d'exécution initiale globale. La réduction de tout sous-terme de l'expression e se fait soit dans un sous-terme global par l'intermédiaire de CONTEXT, soit dans une composante d'un vecteur parallèle par l'intermédiaire de $\textsc{local}_i$.

Remarquons tout d'abord que, par induction structurelle triviale sur E, on a préservation de localité lors du passage au contexte (ce qui justifie l'existence de deux règles pour la réduction par contexte global, et à l'intérieur des vecteurs). Ainsi, si on a le jugement $\mathrm{C}, \Gamma \vdash \mathrm{E}[e] : \tau/\Lambda$, le jugement de typage de e qui apparaîtra dans son arbre de dérivation sera nécessairement de la forme $\mathrm{C}', \Gamma' \vdash e : \tau'/\Lambda$.

Dans le cas d'une réduction par LOCAL$_i$, le typage du vecteur parallèle est fait par la règle VECTOR, qui impose à l'élément local d'être réductible localement (en produisant l'effet ℓ) : d'après le lemme de validité des expressions locales, on ne peut avoir de vecteurs emboîtés, et donc ces deux règles couvrent tous les contextes de réduction possibles.

On raisonnera par induction sur le type du programme, en envisageant les types ayant été engendrés par chacune des règles de typage. Toutes les règles qui suivent préservent la localité, le raisonnement est donc valide aussi bien dans les composantes locales que globalement.

Les règles VAR et DEFFUN attribuent un type à des termes qui sont directement des valeurs, le langage étant strict.

Les règles de typage LET-IN et APP supposent que leurs sous-termes sont bien typés : la règle de réduction par contexte permettant de réduire, pour chacune de ces règles, au moins l'un des sous-termes en question, l'hypothèse d'induction nous permet de conclure que soit une réduction est possible, soit que le sous-terme est une valeur. Dans ce dernier cas, l'une des règles de réduction de tête LET-IN, LET-REC ou APP s'applique, ou bien encore une δ-règle dans le cas de l'application d'un opérateur.

Les δ-règles peuvent demander un peu de travail supplémentaire : la plupart d'entre elles sont applicables quelle que soit la forme de leur argument, mais nth par exemple peut rechercher un élément en dehors d'un tableau. Ce cas, similaire à une division par zéro, sera géré dynamiquement par le mécanisme d'exceptions, on peut donc le distinguer d'une réelle expression bloquée. Il serait néanmoins possible dans notre cas de supprimer les tableaux vides de notre sémantique, et d'appliquer l'argument de nth *modulo* la taille du tableau afin d'obtenir un typage assurant totalement la propriété de progrès. Nous choisissons de ne pas le faire parce que ce problème est résolu par le mécanisme d'exceptions que nous allons introduire d'une part, et que cette astuce ne s'étend pas à un langage réaliste comportant par exemple la division, d'autre part.

La δ-règle concernant l'opérateur *fix* est particulière, car elle exige une forme précise de son argument (le type $\alpha \xrightarrow{\delta} \alpha$ pourrait correspondre à un opérateur, et *fix* nécessite une fonction). Cependant, *fix* est interdit dans les programmes utilisateur, et son apparition dans l'expression par l'intermédiaire de let rec respecte la forme demandée, ce qui nous permet d'éluder le problème.

Les règles PAIR et ARRAY supposent également leurs sous-termes bien typés, or les réductions par les règles CONTEXT et LOCAL$_i$ permettent de réduire dans chacun de ces sous-termes. Par le même raisonnement d'induction, soit une réduction de sous-terme est possible, soit tous les sous-termes sont des valeurs et le terme final en est une également.

La règle IFTHENELSE suppose son premier sous-terme de type bool. Par conséquent, soit celui-ci est réductible par contexte, soit c'est une valeur parmi {true, false}. Quelle que soit cette valeur, il existe une règle de réduction de tête pour réduire l'expression. ■

Autres propriétés

D'autres propriétés, plus spécifiques au parallélisme, sont intéressantes à prouver. Elles correspondent aux règles de sûreté du parallélisme BSML énoncées en début de section : pas d'exécution locale de primitives, et pas d'emboîtement de vecteurs. Le non-emboîtement de vecteurs, comme nous l'avons vu, découle directement du lemme de validité des expressions locales. Détaillons un

peu plus l'exécution locale de primitives :

Démonstration (Pas d'exécution locale de primitives) : Ce qu'on appelle exécution d'une primitive correspond à l'application de l'une des règles de δ-réduction de mkpar, apply, proj, put et *send*. On prouve donc que, pour un programme bien typé, la réduction par la règle LOCAL_i :

$$\text{LOCAL}_i\ \frac{e_1 \rightarrowtail^{\epsilon} e_2}{\mathrm{E}_g\left[\left\langle e'_0 \dots e'_{i-1}, \mathrm{E}_\ell[e_1], e'_{i+1} \dots e'_{p-1}\right\rangle\right] \rightarrowtail \mathrm{E}_g\left[\left\langle e'_0 \dots e'_{i-1}, \mathrm{E}_\ell[e_2], e'_{i+1} \dots e'_{p-1}\right\rangle\right]}$$

peut voir sa prémisse restreinte par une version plus limitée de la réduction de tête, d'où les δ-règles ci-dessus sont absentes. Il s'agit donc d'une extension de la preuve de la propriété de *progrès* à une sémantique plus restrictive.

Par congruence, on peut se limiter au cas où $\mathrm{E}_g = [\cdot]$. Il convient donc de prouver que pour tout terme bien typé de la forme $\left\langle e'_0 \dots e'_{i-1}, \mathrm{E}[e_1], e'_{i+1} \dots e'_{p-1}\right\rangle$, le terme e_1 est réductible sans l'aide des δ-règles sus-mentionnées ; ou bien, par contraposée, que si e_1 se réduit par une δ-règle d'opérateur parallèle, le terme est mal typé.

Supposons donc que e_1 soit réduit par l'une des δ-règles parallèles : e_1 est donc une application d'opérateur de la forme $op\, e$, typée par APP :

$$\text{APP}\ \frac{\mathrm{C}, \Gamma \vdash op : \tau_1 \xrightarrow{\Lambda} \tau_2/\Lambda \qquad \mathrm{C}, \Gamma \vdash e : \tau_1/\Lambda}{\mathrm{C}, \Gamma \vdash op\, e : \tau_2/\Lambda}$$

Il se trouve que tous les opérateurs parallèles ont une localité latente de g, forçant ici $\Lambda = g$. Or le vecteur en lui-même est typé par la règle :

$$\text{VECTOR}\ \frac{\forall i \in \mathcal{P} \setminus \{i\}, (\mathrm{C}, \Gamma \vdash e'_i : \tau/\ell) \qquad \mathrm{C}, \Gamma \vdash \mathrm{E}[e_1] : \tau/\ell \qquad \mathrm{C} \Vdash \tau \lhd \ell}{\mathrm{C}, \Gamma \vdash \left\langle e'_0 \dots e'_{i-1}, \mathrm{E}[e_1], e'_{i+1} \dots e'_{p-1}\right\rangle : \tau\ \mathsf{par}\ /\Lambda}$$

d'où l'on déduit qu'il existe τ, C tels que $\mathrm{C}, \Gamma \vdash \mathrm{E}[e_1] : \tau/\ell$. Or on vérifie aisément par structure de E que le contexte préserve la localité. Par préservation de la localité par le contexte, on en déduit que pour que le vecteur soit typable, il faut que e_1 ait une localité ℓ. On vient de voir que l'application d'opérateur parallèle a une localité g, le terme est donc non typable.

Le système de types assure donc bien qu'il est impossible d'exécuter localement des primitives parallèles. ■

4.2.5 Assouplissement du système de types

Un assouplissement des contraintes dans ce système de types est possible ; nous avons choisi cette définition car elle est directe et plus claire, mais il n'est pas nécessaire d'imposer la contrainte

$\tau \triangleleft \Lambda$ pour tout accès à une variable de type τ dans la localité Λ. Examinons par exemple le programme :

$$\mathsf{let}\, z = (0, \mathsf{mkpar}(\mathsf{fun}\, i \to i))\, \mathsf{in}\, \mathsf{mkpar}(\mathsf{fun}\, i \to \mathsf{fst}\, z)$$

Ce programme est valable : on n'accède qu'à la première composante du couple z, et elle ne contient pas de vecteur parallèle. Il n'est par contre pas typable dans notre système, puisque l'accès local à z dans le paramètre de mkpar engendre la contrainte $\tau \triangleleft \ell$, où τ est le type de z, soit int $\times$(int par), cette contrainte étant fausse.

La variante du système de types permet d'accepter ce type de programmes (ce qui se généralise à l'accès local à un champ d'un enregistrement, même si cet enregistrement contient des champs non locaux). Pour ce faire, la contrainte de localité est déplacée de la règle VAR à la règle DEFFUN, et contraint le type de retour de toute fonction locale. Nos primitives parallèles étant construites de telle sorte que toute exécution locale se fait par application de fonction, en garantissant que ces fonctions ne renvoient que des valeurs acceptables localement, on s'assure de l'impossibilité d'obtenir des vecteurs parallèles tout en autorisant l'accès local à toutes les valeurs, quel que soit leur type.

Les nouvelles règles sont donc :

$$\text{VAR}\ \frac{\Gamma(x) = \forall\overline{\alpha\delta}[\mathrm{C}_2].\tau}{\mathrm{C}_1 \wedge \mathrm{C}_2, \Gamma \vdash x : \tau/\Lambda} \qquad \text{DEFFUN}\ \frac{\mathrm{C}, \Gamma; x : \tau_1 \vdash e : \tau_2/\Lambda \qquad \mathrm{C} \Vdash \Lambda \triangleleft \Lambda_1 \wedge \tau_2 \triangleleft \Lambda}{\mathrm{C}, \Gamma \vdash \mathsf{fun}\, x \to e : \tau_1 \xrightarrow{\Lambda} \tau_2/\Lambda_1}$$

La fonction identité $\mathsf{fun}\, i \to i$ impose ainsi la même contrainte que précédemment, donnant le schéma de type (si on la suppose définie globalement) $\forall\alpha\delta[\alpha \triangleleft \delta].\alpha \xrightarrow{\delta} \alpha$. La fonction $\mathsf{fun}\, i \to ()$ est en revanche typée $\forall\alpha\delta.\alpha \xrightarrow{\delta}$ unit, sans contrainte, alors que le système non assoupli imposerait la contrainte $\alpha \triangleleft \delta$.

Nous nous contenterons de notre système de types plus direct, en remarquant que la limitation est aisée à contourner (il suffit de projeter globalement le terme comportant des éléments parallèles, en supprimant ceux-ci). Dans l'exemple précédent, il suffira d'écrire

$$\mathsf{let}\, a = \mathsf{fst}\, z\, \mathsf{in}\, \mathsf{mkpar}(\mathsf{fun}\, i \to a)$$

4.3 Système de types pour μBSML$^{\mathsf{ref}}$

4.3.1 Problématique

Les références sont non sûres vis à vis de BSML : elles ne respectent pas la séparation entre valeurs locales et globales, comme aperçu lors de la preuve de confluence en 3.3.3. Ainsi, le programme suivant :

```
let r = ref false in
  ≪ r := ($this$ <> 0) ≫
```

permet à une valeur locale (ici $this$<> 0) de s'«échapper» de son vecteur parallèle pour devenir une valeur globale. On perd la cohérence répliquée : la valeur de !r à l'issue du programme ci-dessus est différente d'un processeur à l'autre.

Le principal problème résolu dans cette section est donc celui que nous avons décrit en 3.3.3 comme *séparation de la mémoire*. En catégorisant les références comme locales ou globales, on peut déterminer si un accès à une référence est valide ou non, et éviter le problème de fuite ci-dessus. Plus précisément, l'écriture dans une variable globale est interdite depuis du code exécuté localement. On montre, en 4.3.5 que cela suffit à assurer la séparation mémoire.

L'autre problème rencontré consiste à typer fidèlement les communications de références. En effet, les opérations *deref* et *reref* définies en 3.3.2.0 reconstruisent les références à l'intérieur d'une valeur, et peuvent donc changer leur type et celui de la valeur.

4.3.2 Définitions

μBSML$^{\mathsf{ref}}$ ajoute les adresses mémoire aux expressions : celles-ci sont typées $\tau\,\mathsf{ref}$, où τ est le type de la valeur vers laquelle pointe l'adresse mémoire. Comme les références locales et répliquées correspondent à des réalités différentes, il nous faut distinguer leurs types en indexant le constructeur de types ref par la localité de la référence :

$$\tau ::= ...|\tau\ \mathsf{ref}_\Lambda \,|\, \mathsf{glob}(\tau)|\, \mathsf{loc}(\tau)$$

Les adresses mémoire l sont ajoutées aux termes : elles sont gérées par le système de typage de la même façon que les variables, et on étend en particulier l'environnement Γ afin qu'il puisse associer des types aux adresses mémoire. Les variables de type apparaissant dans le type des adresses ne peuvent être généralisées, sans quoi on autoriserait des types incohérents dans une même référence par l'opération d'écriture – Γ n'associe donc que des types aux adresses mémoire, par opposition aux schéma de types correspondant aux variables. De plus, il est nécessaire de s'assurer qu'on ne généralise pas le type d'expressions susceptibles de créer de nouvelles références. Ce problème a été largement étudié, et n'est pas spécifique à BSML – la solution généralement adoptée consiste à limiter la généralisation aux expressions non-expansives, même si des approches légèrement moins restrictives existent ([Wri92, Gar04]). La règle LET-IN est ainsi étendue pour traiter les deux cas :

LET-IN
$$\frac{\mathrm{C}_2 \wedge \mathrm{C}_1, \Gamma \vdash e_1 : \tau_1/\Lambda \qquad \mathrm{C}_2 \wedge \exists\overline{\alpha\delta}.\mathrm{C}_1, (\Gamma; x : \forall\overline{\alpha\delta}[\mathrm{C}_1].\tau_1) \vdash e_2 : \tau_2/\Lambda \qquad \overline{\alpha\delta} \# \mathcal{L}(\mathrm{C}_2, \Gamma) \qquad e_1 \text{ non expansive}}{\mathrm{C}_2 \wedge \exists\overline{\alpha\delta}.\mathrm{C}_1, \Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau_2/\Lambda}$$

LET-IN-REF
$$\frac{\mathrm{C}, \Gamma \vdash e_1 : \tau_1/\Lambda \qquad \mathrm{C}, (\Gamma; x : \tau_1) \vdash e_2 : \tau_2/\Lambda \qquad e_1 \text{ expansive}}{\mathrm{C}, \Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau_2/\Lambda}$$

Les expressions expansives étant les applications de fonctions ou de l'opérateur ref, et toute expression contenant ce type d'opération autrement que sous une λ-abstraction.

Les transformations effectuées sur les types par *reref* ∘ *deref* produisent des modifications de types : en effet, les références sont re-créées par l'intermédiaire de ref après communications, ce qui leur donne un type ref_ℓ ou ref_g suivant la localité dans laquelle on se trouve (et non pas suivant leur localité d'origine). glob et loc sont utilisés pour signifier ces transformations.

Les opérateurs manipulant les références sont typés par :

$$\begin{aligned}\Gamma_0(\mathsf{ref}) &= \forall\alpha\delta[\alpha \lhd \delta].\alpha \xrightarrow{\delta} \alpha\,\mathsf{ref}_\delta\\ \Gamma_0(\mathsf{get}) &= \forall\alpha\delta_1\delta_2[\alpha \lhd \delta_2].\alpha\,\mathsf{ref}_{\delta_1} \xrightarrow{\delta_2} \alpha\\ \Gamma_0(\mathsf{set}) &= \forall\alpha\delta_1\delta_2.\alpha\,\mathsf{ref}_{\delta_1} \xrightarrow{\delta_2} \alpha \xrightarrow{\delta_1} \mathrm{unit}\end{aligned}$$

ref crée une référence dans la localité courante ; get permet l'accès en lecture quel que soit la localité (sauf pour les références vers des types non-locaux) ; le type de set assure qu'on n'écrit que dans les références indexées par la localité courante. Cela permet d'éviter les écritures locales dans les références globales, ce qui causerait une perte de la cohérence répliquée.

Les types de nos primitves synchrones doivent eux aussi être changés pour indiquer l'opération de recréation des références dans la nouvelle localité :

$$\begin{aligned}\Gamma_0(\mathsf{proj}) &= \forall\alpha\delta.\alpha\,\mathsf{par} \xrightarrow{g} \mathrm{int} \xrightarrow{\delta} \mathsf{glob}(\alpha)\\ \Gamma_0(\mathsf{put}) &= \forall\alpha[\mathsf{loc}(\alpha) \lhd \ell].(\mathrm{int} \xrightarrow{\ell} \alpha)\,\mathsf{par} \xrightarrow{g} (\mathrm{int} \xrightarrow{\ell} \mathsf{loc}(\alpha))\,\mathsf{par}\end{aligned}$$

On définit $\alpha\,\mathsf{ref}_\delta \lhd \Lambda \equiv \alpha \lhd \Lambda$: de la sorte il reste possible localement d'accéder aux références globales.

4.3.3 Communication de références

Les transformations de types glob et loc correspondent à la transformation *reref* ∘ *deref*, en fonction de sa localité d'arrivée ; cela reste relativement simple tant qu'on ne considère pas les types flèches (*reref* ∘ *deref* ne nous garantit en effet pas de renvoyer une expression bien typée si des types flèches arbitraires sont autorisés. Voir la sous-section suivante). *reref* ∘ *deref* recrée toutes les adresses mémoire de l'expression, en les faisant pointer vers des données similaires. En termes de types, les nouvelles références ont donc le même type que les anciennes, mais leur localité est changée.

glob correspond à une ré-instanciation des références dans la localité g, c'est-à-dire après proj :

$$\begin{aligned}\mathsf{glob}(\mathsf{Base}) &= \mathsf{Base}\\ \mathsf{glob}(\tau_1 \times \tau_2) &= \mathsf{glob}(\tau_1) \times \mathsf{glob}(\tau_2)\\ \mathsf{glob}(\tau\,\mathsf{array}) &= \mathsf{glob}(\tau)\,\mathsf{array}\\ \mathsf{glob}(\tau\,\mathsf{ref}_\Lambda) &= \mathsf{glob}(\tau)\,\mathsf{ref}_g\end{aligned}$$

$$\begin{aligned}\mathsf{glob}(\mathsf{glob}\,\tau) &= \mathsf{glob}(\tau)\\ \mathsf{glob}(\mathsf{loc}\,\tau) &= \mathsf{glob}(\tau)\end{aligned}$$

glob n'est défini que sur les types τ tels que $\tau \triangleleft \ell$, du fait que par construction et validité des expressions locales, il ne peut être appliqué que sur de tels types. Ces égalités nous permettent d'obtenir des classes d'équivalence entre types, que nous caractériserons par leur représentant où glob n'est appliqué qu'à des variables.

Cette dernière remarque est valable également pour loc, qui est utilisé pour la ré-instanciation des références en ℓ, après usage de put :

$$\begin{aligned}\mathsf{loc}(\mathsf{Base}) &= \mathsf{Base}\\ \mathsf{loc}(\tau_1 \times \tau_2) &= \mathsf{loc}(\tau_1) \times \mathsf{loc}(\tau_2)\\ \mathsf{loc}(\tau\,\mathsf{array}) &= \mathsf{loc}(\tau)\,\mathsf{array}\\ \mathsf{loc}(\tau\,\mathsf{ref}_\Lambda) &= \mathsf{loc}(\tau)\,\mathsf{ref}_\ell\\ \mathsf{loc}(\mathsf{glob}\,\tau) &= \mathsf{loc}(\tau)\\ \mathsf{loc}(\mathsf{loc}\,\tau) &= \mathsf{loc}(\tau)\end{aligned}$$

4.3.4 Communications, références et types flèches

Notre sémantique nous décrit en détail le comportement des références par rapport aux communications. Intuitivement, le problème vient de ce que la communication conjointe d'opérateurs sur ces références, et donc d'effets latents sur des valeurs susceptibles de changer de type (de ref_ℓ en ref_g par exemple) est beaucoup plus délicate à analyser. Des problèmes subtils peuvent en effet se poser :

```
≪ let r = ref false in
    fun x → set r (rndbool ()) ≫
```

Ce programme définit localement une fonction qui écrit dans une référence locale ; cela n'est pas visible dans le type de cette fonction, qui serait $\alpha \xrightarrow{\ell}$ unit. Le programme se réduit en $(l \mapsto \mathsf{false})\,\langle \mathsf{fun}\, x \to \mathsf{set}\, l\, (\mathsf{rndbool}\,())\rangle_i$. Si on projette cette fonction, la transformation sur les valeurs locales $\mathit{reref} \circ \mathit{deref}$ supprime l et la remplace par une nouvelle référence :

$$\begin{aligned}&\mathit{reref} \circ \mathit{deref}\,(\{l \mapsto \mathsf{false}\}, \mathsf{fun}\, x \to \mathsf{set}\, l\, (\mathsf{rndbool}()))\\ &\quad = \mathit{reref}\,(\mathsf{fun}\, r \to \mathsf{fun}\, x \to \mathsf{set}\, r\, (\mathsf{rndbool}()), \mathsf{false})\\ &\quad = (\mathsf{fun}\, r \to \mathsf{fun}\, x \to \mathsf{set}\, r\, (\mathsf{rndbool}()))\,(\mathsf{ref}\,\mathsf{false})\end{aligned}$$

Si le programme original est communiqué par proj, la nouvelle référence ref false est instanciée globalement. La fonction qui en résulte écrit dans cette référence globale, ce qui lui donnerait la localité latente g ; mais cette fonction applique également l'opérateur rndbool, qui n'est valable que dans la localité ℓ : cette fonction n'est pas correctement typée, et en effet elle casserait la cohérence répliquée qu'elle soit exécutée localement (en changeant une référence globale sur

l'un des processeurs) ou qu'elle soit exécutée globalement (en écrivant des valeurs aléatoires indépendantes dans les composantes du vecteur).

Il nous est donc nécessaire d'introduire une restriction supplémentaire sur la communication des fonctions ; malheureusement, les types ne contiennent pas suffisamment d'informations pour permettre de retracer si la localité d'une fonction est due à des utilisations de références ou à d'autres paramètres. Une analyse statique qui le permettrait dépasserait largement en complexité ce que nous avons défini jusqu'ici puisqu'elle interdirait toute simplification intermédiaire dans le typage. Outre cette difficulté technique, il deviendrait délicat de présenter les types de façon compréhensible au programmeur, et l'utilité en serait somme toute limitée.

Une restriction raisonnable consisterait à n'autoriser que la projection de fonctions *sans effet*, c'est-à-dire n'imposant comme localité ni ℓ ni g. Cela n'est malheureusement pas réaliste au niveau du système de types, car cette propriété se décrit par «les types flèches inclus dans le type ont pour localité des variables libres dans le contexte». Cette propriété ne pouvant s'exprimer localement dans notre système, elle sort du cadre de nos définitions et de nos mécanismes d'inférence.

Des restrictions plus limitantes restent possibles, mais nous avons opté pour la simplicité, en nous appuyant pragmatiquement sur le fait que les cas problématiques restent très limités, et qu'il est peu fréquent d'utiliser la projection sur les fonctions : les fonctions en question sont nécessairement locales, et il aurait été plus judicieux d'utiliser put. On choisit donc d'interdire les types fonctionnels en paramère de la primitive proj, à l'aide de la contrainte $(\not\rightarrow) : \tau(\not\rightarrow)$ est vérifiée si, et seulement si τ ne contient aucun type flèche. Le type de proj est ajusté en conséquence :

$$\Gamma_0(\mathsf{proj}) = \forall\alpha\delta[\alpha(\not\rightarrow)].\alpha\ \mathsf{par} \xrightarrow{g} \mathrm{int} \xrightarrow{\delta} \mathsf{glob}(\alpha)$$

Notons que la problématique deviendrait différente si l'on voulait ajouter les objets d'OCaml au langage, cette restriction empêchant toute communication d'objet comportant des méthodes : dans ce cas, une analyse statique plus fine deviendrait nécessaire.

La primitive put, pour sa part, ne pose pas ces problèmes du fait qu'elle ne peut communiquer que des fonctions locales, et que leur localité n'est donc pas changée. Il nous suffit d'étendre loc :

$$\mathsf{loc}(\tau_1 \xrightarrow{\ell} \tau_2) = \tau_1 \xrightarrow{\ell} \mathsf{loc}(\tau_2)$$

4.3.5 Correction

La confluence de la sémantique de $\mu\text{BSML}^{\mathsf{ref}}$ a été établie à la condition qu'une propriété de séparation mémoire soit respectée. Rappelons que cette propriété requiert que l'opérateur set ne soit appliqué localement que sur des adresses mémoire dont la portée est limitée à la même composante locale du vecteur. Commençons donc par prouver que notre système de types assure cette propriété.

Propriété 4.3.1 (*Séparation mémoire pour les programmes μBSML$^{\mathsf{ref}}$ bien typés*)

Dans tout programme μBSML$^{\mathsf{ref}}$ bien typé, si l'opérateur set *est appliqué à l'adresse mémoire*

l dans une composante d'un vecteur parallèle, alors l n'apparaît que localement, sur le même processeur, dans tout le programme.

Démonstration (Séparation mémoire) : Une application de set se type de la façon suivante :

$$\frac{\mathrm{C}, \Gamma \vdash \mathsf{set} : \tau\,\mathsf{ref}_\Lambda \xrightarrow{\Lambda'} \tau \xrightarrow{\Lambda} \mathrm{unit}/\Lambda' \qquad \mathrm{C}, \Gamma \vdash l : \tau\,\mathsf{ref}_\Lambda\,/\Lambda'}{\mathrm{C}, \Gamma \vdash \mathsf{set}\ l : \tau \xrightarrow{\Lambda} \mathrm{unit}/\Lambda'}\ \text{APP}$$

L'application locale de $\mathsf{set}\ l$ implique donc que l soit typée $\tau\,\mathsf{ref}_\ell$. Un tel résultat peut être obtenu par un appel local à l'opérateur ref, ou bien par communication de références. La construction des vecteurs parallèles nous garantit que l n'est accessible que localement par l'intermédiaire de apply, donc à l'intérieur de la même composante. Dans le cas de communications par proj ou put, la transformation *reref* $\circ$ *deref* nous garantit que toutes les adresses mémoire apparaissant après communication sont toutes fraîches. On a donc bien séparation mémoire. ■

Nous donnons ici les grandes lignes des preuves de préservation et de progrès. Notre typage des références ne présente pas, hors communications, de différences notables avec les systèmes habituels. Nous nous référons donc à [PR05] pour la preuve de préservation vis-à-vis des références, en se reposant sur le critère de non généralisation des expressions expansives. Le non-emboîtement des vecteurs et l'absence d'exécution locale de primitives parallèles sont préservés : les preuves ne présentent pas de subtilités particulières dues aux références.

Les communications, étant munies de la transformation *reref* $\circ$ *deref*, recréent à l'arrivée toutes références utiles à l'aide de ref. glob et loc correspondent bien à l'application de $\mathsf{ref} \circ \mathsf{get}$ sur les références incluses dans le terme, ce qui se montre aisément par induction sur le type du terme.

4.4 Système de types pour μBSML$^{\mathsf{exn}}$

La gestion des exceptions en parallèle n'introduit aucune difficulté particulière vis-à vis du typage. Nous supposons, dans notre système, défini un type quelconque τ_{exn}, toutes les exceptions adoptant ce type. On peut alors typer les constructions spécifiques :

$$\text{TRY}\ \frac{\mathrm{C}, \Gamma \vdash e : \tau/\Lambda \qquad \mathrm{C}, (\Gamma; x : \tau_{\mathsf{exn}}) \vdash e' : \tau/\Lambda}{\mathrm{C}, \Gamma \vdash \mathsf{try}\ e\ \mathsf{with}\ x \to e' : \tau/\Lambda} \qquad \text{TRYPAR}\ \frac{\mathrm{C}, \Gamma \vdash e : \tau/g \qquad \mathrm{C}, (\Gamma; x : \tau_{\mathsf{exn}}\,\mathsf{array}) \vdash e' : \tau/g}{\mathrm{C}, \Gamma \vdash \mathsf{trypar}\ e\ \mathsf{withpar}\ x \to e' : \tau/g}$$

$$\text{FAIL}\ \frac{\mathrm{C}, \Gamma \vdash e : \tau_{\mathsf{exn}}/\Lambda}{\mathrm{C}, \Gamma \vdash \mathit{fail}\, e : \tau/\Lambda} \qquad \text{FAILPAR}\ \frac{\mathrm{C}, \Gamma \vdash e : \tau_{\mathsf{exn}}\,\mathsf{array}\,/g}{\mathrm{C}, \Gamma \vdash \mathit{failpar}\, e : \tau/g}$$

$$\Gamma_0(\mathsf{raise}) = \forall \alpha\delta[\alpha \lhd \delta].\tau_{\mathsf{exn}} \xrightarrow{\delta} \alpha$$

La règle concernant *tryparsub* est identique à celle de trypar. On suppose, afin d'assurer qu'on puisse lever des exceptions localement, que $\tau_{\mathsf{exn}} \triangleleft \ell$.

Les preuves de cohérence de ce typage s'étendent trivialement à partir des preuves du typage de μBSML.

4.5 Algorithme d'inférence

Dans HM(X), l'inférence se fait purement par résolution de contraintes : le problème d'inférence de types se ramène à déterminer s'il existe une contrainte C telle que $\mathrm{C}, \Gamma \vdash e : \tau/\Lambda$, pour Γ, e, τ, Λ donnés. L'inconnue est ici la contrainte et non pas, comme on pourrait l'attendre, le type et la localité : en effet, en choisissant pour τ et Λ des variables fraîches dans Γ, l'existence de C nous donnera l'existence d'une instance pour ces variables, et la contrainte contiendra toutes les informations nécessaires sur elles.

On doit donc, dans un premier temps, construire un mécanisme de génération d'une contrainte C qui soit nécessaire et suffisante pour obtenir $\mathrm{C}, \Gamma \vdash e : \tau/\Lambda$. Cette contrainte est aussi la contrainte la moins spécifique possible garantissant le résultat : toute contrainte C' le garantissant aussi sera telle que $\mathrm{C}' \Vdash \mathrm{C}$.

4.5.1 Génération des contraintes

Le mécanisme de génération de contraintes complet et cohérent que nous décrivons ici nous fournit une propriété équivalente à la propriété d'existence de schémas de types principaux dans DM. La cohérence du mécanisme nous assure que la contrainte C générée est suffisante pour assurer $\mathrm{C}, \Gamma \vdash e : \tau/\Lambda$, et la complétude, comparable à la principalité des schémas dans DM, nous assure que la contrainte est la moins spécifique possible.

Certaines contraintes ne nous ont pas été nécessaires pour la définition du système de types, mais doivent être ajoutées au système afin que nous puissions assurer cette propriété :

$$
\begin{array}{rcll}
\mathrm{C} & ::= & \cdots & \\
 & & \mathsf{let}\, x : \sigma \,\mathsf{in}\, \mathrm{C} & \text{Introduction de schéma de type} \\
 & & x \preceq \tau & \text{Instanciation de schéma de type}
\end{array}
$$

Où $x \preceq \tau$ indique que τ est une instance du schéma de type associé à x. On utilisera également la notation $\sigma \preceq \tau$ pour signifier, si $\sigma = \forall \overline{\alpha\delta}[\mathrm{C}].\tau'$ avec $\overline{\alpha\delta} \# \mathcal{L}(\tau)$, la contrainte $\exists \overline{\alpha\delta}.(\mathrm{C} \wedge \tau = \tau')$, c'est-à-dire que τ est une instance de σ.

La contrainte $\mathsf{let}\, x : \sigma \,\mathsf{in}\, \mathrm{C}$ lie l'identifiant x au schéma de types σ à l'intérieur de C. Elle suppose également l'existence d'une instance pour σ. Cela donne donc, à l'intérieur de C, le sens $\sigma \preceq \tau$ à toute sous-contrainte $x \preceq \tau$ où x apparaît libre.

Ces deux contraintes permettent, par rapport à DM, de remplacer l'extension de l'environnement de typage, et la recherche d'une correspondance dans cet environnement.

La contrainte générée pour Γ, e, τ, Λ sera notée $\mathbb{C}(\Gamma \vdash e : \tau/\Lambda)$. La génération se fait par induction sur la structure de e : les nœuds let donnent lieu au déplacement d'une partie de la

contrainte courante à l'intérieur d'un nouveau schéma de types dans l'environnement, et cette sous-contrainte est ensuite reprise dans la contrainte courante lors de l'apparition de la variable concernée – ce qui correspond aux règles de typage LET-IN et VAR. On simplifie ainsi le problème en $\mathbb{C}(e : \tau/\Lambda)$ (sans environnement Γ), la solution du problème original découlant de celui-ci par $\mathsf{let}\,\Gamma\,\mathsf{in}\,\mathbb{C}(\Gamma \vdash e : \tau/\Lambda)$. La génération des contraintes est décrite en figure 4.3.

$$
\begin{aligned}
\mathbb{C}(x : \tau/\Lambda) &= x \preceq \tau \wedge \tau \lhd \Lambda \\
\mathbb{C}(\mathsf{fun}\, x \to e : \tau/\Lambda) &= \exists \alpha\beta\delta.((\mathsf{let}\, x : \alpha\, \mathsf{in}\, \mathbb{C}(e : \beta/\delta)) \wedge \delta \lhd \Lambda \wedge \tau = \alpha \xrightarrow{\delta} \beta) \\
\mathbb{C}(e_1\, e_2 : \tau/\Lambda) &= \exists \alpha.(\mathbb{C}(e_1 : \alpha \xrightarrow{\Lambda} \tau/\Lambda) \wedge \mathbb{C}(e_2 : \alpha/\Lambda)) \\
\mathbb{C}(\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau/\Lambda) &= \mathsf{let}\, x : \forall \alpha[\mathbb{C}(e_1 : \alpha/\Lambda)].\alpha\, \mathsf{in}\, \mathbb{C}(e_2 : \tau/\Lambda) \\
\mathbb{C}((e_1, e_2) : \tau/\Lambda) &= \exists \alpha\beta.(\mathbb{C}(e_1 : \alpha/\Lambda) \wedge \mathbb{C}(e_2 : \beta/\Lambda) \wedge \tau = \alpha \times \beta) \\
\mathbb{C}(\mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : \tau/\Lambda) &= \mathbb{C}(e : \mathrm{bool}/\Lambda) \wedge \mathbb{C}(e_1 : \tau/\Lambda) \wedge \mathbb{C}(e_2 : \tau/\Lambda) \\
\mathbb{C}([e_i]_i^n : \tau/\Lambda) &= \mathbb{C}(n : \mathrm{int}/\Lambda) \wedge \exists \alpha.(\bigwedge_i \mathbb{C}(e_i : \alpha/\Lambda) \wedge \tau = \alpha\, \mathsf{par}) \\
\mathbb{C}(\langle e_i \rangle_i : \tau/\Lambda) &= \exists \alpha.(\bigwedge_i \mathbb{C}(e_i : \alpha/\ell) \wedge \alpha \lhd \ell \wedge \tau = \alpha\, \mathsf{par})
\end{aligned}
$$

FIG. 4.3 – Génération de contraintes

On suppose, dans ces règles, que les variables apparaissant dans le terme droit sont toujours choisies distinctes et fraîches par rapport au terme gauche ; les contraintes obtenues sont de la sorte égales modulo α-conversion. Ce mécanisme de génération de contrainte est très proche de celui proposé dans [PR05], les contraintes de localité s'y ajoutant naturellement.

La première équation correspond à l'instanciation d'une variable (règle de typage VAR) : x a le type τ dans la localité Λ si, et seulement si τ est acceptable dans cette localité, et τ est une instance du schéma de type associé à x. Cette dernière contrainte prend son sens lorsqu'on a une utilisation préalable de let liant x à un schéma de types.

La deuxième équation correspond à la règle de typage DEFFUN ; elle donne une contrainte existentielle, c'est-à-dire qu'il sera à la charge du système de résolution des contraintes de déterminer les termes correspondant. Elle établit que le terme $\mathsf{fun}\, x \to e$ aura le type τ dans la localité Λ si, et seulement si on peut trouver de types α, β et une localité δ tels que τ ait la forme $\alpha \xrightarrow{\delta} \beta$, que la localité δ soit plus locale que Λ, et qu'on puisse assurer que e ait le type β dans la localité δ si x a pour type α. Cette dernière condition est assurée par la contrainte $\mathsf{let}\, x : \alpha\, \mathsf{in}\, \mathbb{C}(e : \beta/\delta)$: il est vraisemblable que e contient des instances de x, qui introduiront dans la contrainte générée la sous-contrainte $x \preceq \tau'$, forçant à ce moment la cohérence de α et τ'.

La quatrième règle correspond à la règle de typage LET-IN et revêt une importance particulière, puisqu'elle régit la généralisation : elle stipule que le terme $\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$ a le type τ dans la localité Λ si, et seulement si, en supposant que x ait un type quelconque tel que $\mathbb{C}(e_1 : \alpha/\Lambda)$, on a la garantie que e_2 a le type τ dans la localité Λ. En supposant inductivement que $\mathbb{C}(e_1 : \alpha/\Lambda)$ est une contrainte suffisante et minimale, le schéma de type $\forall \alpha[\mathbb{C}(e_1 : \alpha/\Lambda)].\alpha$ est de fait un schéma de type principal pour e_1. De la sorte, α est la seule variable de type apparaissant libre dans $\mathbb{C}(e_1 : \alpha/\Lambda)$, et donc la seule ayant besoin d'être généralisée.

Les autres règles reprennent nos règles de typage en assurant, d'une part, que leurs prémisses sont bien respectées en les réécrivant sous forme de contraintes, d'autre part que le format du type final est correct.

Théorème 4.5.1 (*Correction de la génération de contraintes*)

$(\mathbb{C}(\Gamma \vdash e : \tau/\Lambda)), \Gamma \vdash e : \tau/\Lambda.$

Théorème 4.5.2 (*Complétude de la génération de contraintes*)

Si $\mathrm{C}, \Gamma \vdash e : \tau/\Lambda$, *avec* C *et* Γ *sans variables programmeur libres, et* Γ *cohérent dans* C, *alors* $\mathrm{C} \Vdash \mathbb{C}(\Gamma \vdash e : \tau/\Lambda)$.

Les preuves de ces théorèmes sont assez directes depuis la définition du mécanisme de génération de contraintes, chaque étape de celui-ci établissant une contrainte nécessaire et suffisante. Le lecteur souhaitant une démonstration plus détaillée pourra se reporter à [PR05], les contraintes que nous ajoutons n'interférant pas avec les systèmes qui y sont décrits.

Extensions

$\mu\text{BSML}^{\mathsf{exn}}$, présentant des règles de typage supplémentaires, nécessite l'adjonction des règles de génération de contraintes suivantes :

$$\begin{aligned}
\mathbb{C}(\mathsf{try}\, e\, \mathsf{with}\, x \to e' : \tau/\Lambda) &= \mathbb{C}(e : \tau/\Lambda) \wedge \mathsf{let}\, x : \tau_{\mathsf{exn}}\, \mathsf{in}\, \mathbb{C}(e' : \tau/\Lambda) \\
\mathbb{C}(\mathsf{trypar}\, e\, \mathsf{withpar}\, x \to e' : \tau/\Lambda) &= \mathbb{C}(e : \tau/g) \wedge \mathsf{let}\, x : \tau_{\mathsf{exn}}\, \mathsf{array}\, \mathsf{in}\, \mathbb{C}(e' : \tau/g) \wedge \Lambda = g \\
\mathbb{C}(\mathit{fail}\, e : \tau/\Lambda) &= \mathbb{C}(e : \tau_{\mathsf{exn}}/\Lambda) \\
\mathbb{C}(\mathit{failpar}\, e : \tau/\Lambda) &= \mathbb{C}(e : \tau_{\mathsf{exn}}\, \mathsf{array}\, /\Lambda)
\end{aligned}$$

$\mu\text{BSML}^{\mathsf{ref}}$ ne nécessite pour sa part aucun ajout d'équation, mais ajoute la contrainte $(\nrightarrow)$ au système (en plus des constructions glob et loc).

4.5.2 Résolution des contraintes

Nous ne décrirons pas ici de solveur de contraintes en détail ; au lieu de ça, nous nous baserons sur les nombreux travaux existants sur le sujet et en particulier sur la version correspondant à HM(=) proposée dans [PR05].

En effet, une fois la génération établie, les seules différences entre nos contraintes et celles pour lesquelles nous disposons d'un mécanisme de résolution sont :

- L'existence de variables de localité.
- L'existence de contraintes de la forme $\tau \triangleleft \Lambda$.
- Pour $\mu\text{BSML}^{\mathsf{ref}}$, l'existence de contraintes de la forme $\tau(\nrightarrow)$.

Les variables de localité, à ce niveau, ont un comportement exactement similaire à celui des variables de type : de fait, elles peuvent être considérées comme des variables de types ne prenant que g ou ℓ comme valeur. Leur présence n'ajoute ainsi aucune complexité au solveur de contrainte.

Les contraintes $\tau \lhd \Lambda$ et $\tau(\twoheadrightarrow)$ ont une structure particulière et portent uniquement sur la structure de τ. De fait, nous avons montré lors de leur définition que ces contraintes se propagent directement dans les sous-termes d'un type, et il nous est ainsi possible de définir une forme normale où elles ne s'appliquent qu'à des variables de type. Elles n'ont donc de rôle que lors de l'unification des variables en question, celle-ci pouvant être rendue invalide (si on tente d'unifier α par et β sous la contrainte $\beta \lhd \ell$ par exemple) ou donner lieu à de nouvelles contraintes.

Ainsi, il nous suffit de modifier l'algorithme d'unification des contraintes, dont le rôle est de résoudre des équations de typage dénuées d'introduction et d'instanciation de schémas de type ; la partie majeure du solveur est inchangée. De fait, le solveur proposé dans [PR05] propose une détection des types cycliques, ce qui n'est pas très différent de notre approche. Lorsque l'algorithme d'unification est en mesure de rapprocher une variable et un type ou une localité au sein de la même classe d'équivalence, toute sous-contrainte $\lhd$ ou $(\twoheadrightarrow)$ qui apparaîtrait dans la contrainte courante et mettrait en cause cette variable subit ainsi la substitution correspondante, et est immédiatement normalisée :

$$\alpha = \tau \wedge \alpha \lhd \Lambda \Rightarrow \alpha = \tau \wedge \tau \lhd \Lambda$$
$$\delta = \Lambda \wedge \alpha \lhd \delta \Rightarrow \delta = \Lambda \wedge \alpha \lhd \Lambda$$
$$\alpha = \tau \wedge \alpha(\twoheadrightarrow) \Rightarrow \alpha = \tau \wedge \tau(\twoheadrightarrow)$$

Ces règles peuvent être appliquées aussitôt qu'une nouvelle égalité est formée lors du processus de résolution, et sont suivies de la réduction du terme de droite, et d'éventuelles nouvelles applications des règles en chaîne.

Chapitre 5

Implantation et expériences

5.1 Implantation de BSML

BSML est implanté au-dessus du langage OCaml : le langage BSML complet correspond aux fonctionnalités parallèles de μBSML ajoutées à OCaml plutôt qu'à un mini-ML. Ainsi, les opérateurs parallèles de BSML ont le même comportement que décrit dans notre sémantique, mais le langage de base, disponible aussi bien localement qu'en mode répliqué, est beaucoup plus riche. Le but de notre sémantique n'est pas ici d'étudier le détail d'OCaml, dont les fonctionnalités ont déjà fait le sujet de nombreux travaux : il nous suffira de remarquer que tous les traits fonctionnels tels que types somme, types énumérés, filtrage par motifs, variants polymorphes, modules, foncteurs, *etc.* n'interfèrent pas avec le parallélisme et que BSML se contente de gérer de façon sûre l'interaction entre les deux niveaux d'exécution, local et global.

Les traits non fonctionnels qui n'ont pas été traités dans la sémantique, tels que les objets, sont pour le moment exclus de notre analyse du langage.

Outre l'étendue du langage, sa disponibilité sur une variété de plates-formes est un atout qui garantit que le code sera réutilisable sur différents types de super-calculateurs. De plus, OCaml offre un compilateur vers du code natif dont les performances sont comparables à celles de langages de plus bas niveau tels que C ou C++. Bien que le code produit reste généralement moins efficace que du code C, OCaml est le meilleur choix de langage fonctionnel possible en matière de performances [HFA+96] : il est donc naturel de l'utiliser pour notre approche.

La distribution de BSML offre deux versions des primitives :

- la première, séquentielle, est en réalité une simulation du parallélisme : elle permet d'intégrer simplement BSML au mode interactif d'OCaml (*toplevel*), de se familiariser avec son fonctionnement, et de tester de manière directe du code BSML sur de petits échantillons. L'objectif de ce mode, comme en OCaml, est de faciliter la programmation en permettant la visualisation rapide des résultats (y compris les vecteurs parallèles) ;
- la deuxième version est réellement parallèle, et permet le déploiement des programmes sur les architectures de destination. Cette implantation est modulaire : elle repose sur un module de communication de bas niveau externe, que l'on peut choisir en fonction du contexte et des bibliothèques disponibles.

Les deux versions partagent bien sûr la même bibliothèque standard, celle-ci ne recourrant pas au parallélisme de bas niveau. Les outils de compilation fournis avec BSML integrent également le

pré-traitement nécessaire à la syntaxe alternative, au traitement des exceptions, au filtrage par motifs et – bien que ce deriner ne soit pas encore entièrement implanté – le système de types.

5.1.1 Mode interactif

Il est souvent utile de pouvoir effectuer quelques tests sur des fonctions auxiliaires aussitôt celles-ci programmées, plutôt que de retrouver par la suite l'origine des bogues dans une application complète et complexe, ou éventuellement lancer des séries de tests unitaires sur l'ensemble du code. L'utilisation du mode interactif permet d'une part la vérification de typage d'OCaml de façon instantanée et permet à l'utilisateur de vérifier que le type inféré est bien celui qui l'attendait, d'autre part d'effectuer interactivement des tests pour vérifier le comportement de son code.

La difficulté de la programmation parallèle ne faisant que renforcer ce besoin, l'existence d'un mode interactif pour BSML, bien que celui-ci ne comporte aucun parallélisme réel, nous paraît une fonctionnalité très appréciable, qui est peu répandue (sans être unique, ce mode étant par exemple présent dans OcamlP3L [CLPW08]). La visualisation des vecteurs parallèles se fait de la façon suivante :

```
# let v = mkpar (fun i → float_of_int i);;
val v : float Bsml.par = ≪ 0., 1., 2., 3. ≫
```

Où le mode interactif a été parametré pour simuler quatre processeurs (ce paramètre est une option de configuration du mode interactif). En utilisant la syntaxe alternative, il est même possible d'obtenir :

```
# let v' = ≪ sqrt $v$ ≫ ;;
val v' : float Bsml.par =
  ≪ 0., 1., 1.41421356237309515, 1.73205080756887719 ≫
```

Imaginons qu'on veuille vérifier rapidement le comportement de notre fonction de rééquilibrage non ordonné de listes parallèles (voir 109) :

```
# vl;;
val vl : int list Bsml.par =
  ≪ [76; 60; 32; 74; 92; 20; 75], [12; 88; 14], [24; 78], [62; 0; 90; 4; 21; 60] ≫
# rebalance;;
- : 'a list Bsml.par → 'a list Bsml.par = <fun>
# rebalance vl;;
- : int list Bsml.par =
  ≪ [74; 92; 20; 75], [32; 60; 12; 88; 14], [76; 24; 78; 62], [0; 90; 4; 21; 60] ≫
```

5.1.2 Mode parallèle

Afin de pouvoir exécuter les problèmes BSML de façon efficace sur des architectures parallèles, notre langage se base sur différentes bibliothèques de communication. Il est organisé comme suit :

- un foncteur générique prenant en paramètres les modules de configuration, de communication et d'entrées-sorties, qui fournit les vecteurs parallèles, les primitives BSML et la gestion des exceptions ;

- des modules sous-jacents au module générique, offrant la gestion des exceptions et la superposition parallèle ;
- des modules de communication basés sur MPI[SG98], PUB[BJvOR03] ou directement sur l'interface TCP/IP d'OCaml. Ceux-ci permettent la compilation de différentes versions de BSML. Ces modules fournissent la primitive de bas niveau send qui sera utilisée pour l'implantation des primitives.

Une instance du programme devra être lancée sur chaque nœud de la machine parallèle, et des canaux de communication ouverts : ce rôle est délégué au module de communication, puisqu'il est spécifique à chaque bibliothèque et à sa procédure d'initialisation propre. Des outils annexes permettent la compilation et l'exécution des programmes suivant le choix qui a été fait : bsmlopt.tcp et bsmlrun.tcp, respectivement, se chargent de la compilation en code natif et de l'exécution (incluant le déploiement sur la machine parallèle) en utilisant la bibliothèque TCP/IP OCaml.

Dans le cas de la bibliothèque MPI, par exemple, le module de communication se charge d'appeller les fonctions en C de la bibliothèque, en particulier MPI_Init() ; au cours de chaque super-étape, les p instances du programme se chargent des calculs répliqués et locaux (ces derniers correspondant aux primitives asynchrones, traitées dans le module generic sans appel au module de communication), et l'appel à une primitive synchrone se traduit par une communication implantée en termes de MPI_Alltoallv(). Le traitement correct des exceptions permet également de s'assurer que MPI_Finalize() est bien appellé en fin de programme, et que des processus bloqués ne restent pas en attente sur les nœuds, même en cas d'erreur.

Notons que – bien que rien ne l'exclue dans notre modèle – notre implantation ne permet pas, à l'heure actuelle, une exécution réellement efficace sur une machine multi-cœur (et donc à mémoire partagée). En effet, il reste possible de lancer une instance du programme sur chacun des cœurs, mais les communications de données nécessiteront une sérialisation et une communication par l'intermédiaire du système d'exploitation. Il serait possible de limiter le problème en ajoutant un module de communication utilisant de la mémoire partagée entre processus : les communications se verraient grandement accélérées.

Pour éviter la sérialisation et les recopies mémoire qu'elle entraîne, cependant, il faudrait pouvoir utiliser des processus légers (*threads*) au lieu de processus sur chacun des cœurs : le problème qui se pose alors au niveau du langage est celui la récupération mémoire concurrente, problème qui n'a pas encore été résolu pour notre langage hôte OCaml (on envisage néanmoins une implantation spécifique à BSML, la structure de la mémoire en répliqué/local rendant ce problème beaucoup plus simpl).

5.2 Spécificités de l'implantation de la gestion des exceptions parallèles

5.2.1 Protection des opérations locales

Les opérations locales doivent être protégées de deux façons : d'une part, pour éviter que les exceptions locales ne puissent bloquer l'exécution du processus, d'autre part, pour bloquer l'exécution locale si une exception a précédemment été levée sur le processeur (les valeurs locales pouvant être, dans ce cas, incohérentes). Cela donne, en prenant l'exemple de mkpar :

```
let mkpar f = match Exceptions.get_proc_state() with
  | Exceptions.Fine →
      (try Vec (f (Communication.pid())) with
         | e → Exceptions.recover_exn e; Invalid e)
  | Exceptions.Stopped (i,e) → Invalid e
```

Ce code vérifie l'état du processeur par la fonction Exceptions.get_proc_state ; dans le cas Fine, le processus est en état de continuer les calculs et renvoie le résultat Vec (f (Communication.pid())), tout en rattrapant une éventuelle exception levée lors de l'application de f. Dans ce dernier cas, la fonction recover_exn se charge de mettre à jour l'état du processeur et d'éviter d'autres calculs locaux jusqu'à la fin de la super-étape.

En cas d'échec des calculs, la valeur renvoyée est Invalid e où e est l'exception levée, au lieu d'être un vecteur parallèle Vec v. L'état de chaque processeur est stocké dans une référence (pour chaque super-thread, dans le cas où la superposition est utilisée).

La sémantique est respectée : une exception locale est rattrapée et mémorisée, permet la suite des opérations répliquées mais interdit, sur le processeur concerné, de nouvelles opérations locales. L'implantation de **apply** est tout à fait similaire à celle de **mkpar**.

5.2.2 Communication des exceptions

La communication des exceptions doit permettre une décision globale à partir des informations distribuées. Ce rôle est dévoué à la primitive send du module de communication. Cette primitive est donc typée comme suit :

```
type ('ctrl_type, 'a) ctrl_or_data =
  | Ctrl of 'ctrl_type
  | Data of 'a array

val send : ('ctrl_type,'a) ctrl_or_data → ('ctrl_type option array,'a) ctrl_or_data
```

Le paramètre de send a un type somme, Data pour une communication normale de données ou Ctrl, dans les cas exceptionnels, pour signifier que la communication des données doit être abandonnée dès que possible et laisser place à l'échange global des exceptions. Si tous les processeurs appellent send avec une valeur Data, le résultat est donc le Data après communications, send correspondant dans ce cas à la primitive *send* de la sémantique sur nos tableaux. Si, en revanche, au moins un des processeurs appelle send avec une valeur Ctrl, la valeur de retour contiendra un tableau d'options, indiquant l'éventuelle exception présente à chacun des processeurs.

Dans l'exemple de l'implantation sur une bibliothèque MPI, la communication est faite en deux étapes : une première étape indiquant la taille des données qui seront échangées et s'il s'agit d'exceptions ou non, et une seconde qui ne communique donc les données que si aucune exception n'est présente.

Un problème purement technique, mais très délicat, s'est présenté lors de l'implantation de ce mécanisme : en effet, Objective Caml, dans son implantation actuelle, ne permet pas le «Marshalling» sur les exceptions, c'est-à-dire la sérialisation des données afin de les rendre transmissibles d'un processeur à un autre. Une parenthèse sur l'implantation des exceptions dans OCaml s'impose : les exceptions sont un type somme, mais qui est dynamiquement étendu lors de l'exécution

afin qu'il soit possible à l'utilisateur de définir autant de nouveaux constructeurs d'exceptions qu'il le souhaite. Ainsi la déclaration d'une exception **exception** E entraîne la création d'un nouvel identifiant pour le constructeur E, et ce de façon indépendante à chaque exécution (il s'agit d'une adresse mémoire allouée) : on ne peut donc pas comparer des éléments de type exception entre deux processus différents, leurs constructeurs étant liés à une implantation mémoire donnée.

La solution que nous proposons n'est possible que grâce à la structure bien définie de l'exécution en BSML. Elle repose sur le fait que la définition des exceptions sera faite en mode répliqué, et donc de la même façon sur tous les processeurs[12]. En utilisant, une nouvelle fois, camlp4, nous ajoutons donc une exécution de code auxiliaire lors de toute déclaration d'exception. Ce code va nous permettre, à l'intérieur des mécanismes de BSML, de créer une table de toutes les exceptions déclarées, locale à chaque processeur. Cette table nous permet d'obtenir un identifiant global pour chaque exception, qu'il nous sera possible de communiquer pour ré-obtenir, sur chaque processeur, l'exception voulue. Les exceptions standard d'OCaml telles que Match_failure ou Stack_overflow sont ajoutées dans cette table à l'initialisation de BSML.

5.2.3 Propagation et rattrapage

Comme indiqué dans la description informelle du système d'exceptions, l'implantation est entièrement construite par-dessus le langage OCaml existant. Les ensembles d'exceptions locales sont donc traités à l'aide des mécanismes d'exceptions usuels : comme leur propagation est tout à fait similaire à celle des exceptions standard, cela est relativement simple.

Dans notre implantation, nous utilisons des listes de couples (pid, exception) pour les ensembles d'exceptions ; une précédente version se basait sur les Set d'OCaml, mais la complexité de traitement était accrue, et le bénéfice en temps inexistant par rapport à la taille usuelle de cet ensemble (limité par p, qui ne dépasse les 1000 que sur les plus grosses machines parallèles à l'heure actuelle). Le format de cette liste nous permet de plus d'utiliser directement le mécanisme de filtrage par motifs parallèle. Par souci de simplicité, nous supposerons dans cette section que l'utilisateur traite lui-même cette liste, l'implantation du filtrage parallèle faisant l'objet de sa propre section.

Nous utilisons donc, de manière invisible à l'utilisateur, la définition suivante comme véhicule pour les ensembles d'exceptions :

```
type locexn = { proc: int; exc: exn }
exception Global_exn of locexn list
```

La structure **trypar withpar** est ensuite transformée par le pré-processeur en un **try with** normal, qui ne rattrape que les exceptions de type Global_exn. Comme indiqué dans la sémantique, une barrière est nécessaire avant le **withpar**, afin de s'assurer que celui-ci rattrape toute exception locale levée à l'intérieur du bloc. Cette barrière, de plus, doit être imposée même si une exception globale a été levée :

```
(* code initial *)
let x = trypar [code] withpar | → [traitement]
```

[12] Le seul cas où il est possible de définir une exception «localement» est la syntaxe, un peu particulière, permettant en OCaml de définir des modules locaux **let module** = ... Leur usage étant marginal, nous les interdisons en BSML.

```
(* code transformé *)
let x =
  try
    let ans = [code] in barrier(); ans
  with
    | Global_exn l → [traitement]
    | other_exn → try barrier(); raise other_exn
        with Global_exn l → [traitement]
```

Où nous supposons que barrier() met fin à la super-étape courante. Il est important de constater que ce code établit au maximum une barrière de synchronisation : si une exception répliquée a été levée, le traitement sur other_exn est nécessairement déclenché, et effectue une barrière, relevant l'exception une fois assuré qu'aucune exception locale n'est en attente. Dans le cas contraire, soit [*code*] comportait une barrière et a déjà globalisé des exceptions locales, soit la barrière est effectuée, appelant dans les deux cas le code [*traitement*] en cas d'exception locale.

Les exceptions locales appartenant aux différents niveaux d'imbrication de **trypar withpar**, correspondant à la suite d'états $\omega_0, \ldots, \omega_n$ dans la sémantique, sont implantées à l'aide d'indices de Bruijn : un indice 0 est attribué à toute exception placée dans l'environnement lors d'un calcul local. Ensuite, toute entrée dans un bloc **trypar withpar** incrémente cet indice par effet de bord, indiquant la profondeur relative de l'exception en attente. Une fois l'ensemble d'exceptions globalisé, seules les exceptions d'indice maximal (donc les plus anciennes) sont propagées, et elles remontent autant de niveaux d'imbrication de **trypar withpar**. De la sorte, la sémantique est respectée et il est garanti que l'exception est rattrapée dans le bloc où elle a été levée.

Ces opérations sont effectuées par des fonctions cachées de la bibliothèque Bsml, __trypar et __withpar, que le pré-processeur insère automatiquement à l'entrée et à la sortie des blocs **trypar withpar** :

```
let __trypar () = match SthrData.get proc_state with
  | Stopped (depth,e) → SthrData.set proc_state (Stopped (depth+1,e))
  | Fine → ()

let __withpar exnset =
  let depths = (SthrData.get depth_of_exception) in
  let deeper_exns = List.filter
    (fun e →
       depths.(e.proc) ← depths.(e.proc) − 1;
       depths.(e.proc) >= 0)
    exnset
  in
    SthrData.set last_exnset_raised deeper_exns;
    if deeper_exns = []
    then List.rev_map (fun { proc=p; exc=e } → p,e) exnset
    else raise (Global_exn deeper_exns)
```

(le module SthrData, qui ne nous est pas utile pour le moment, permet de manipuler des données locales au super-thread). La fonction __withpar filtre l'ensemble d'exceptions pour en retenir les éléments qui ne correspondent pas au niveau d'imbrication courant : s'il y en a,

l'exception est relevée, dans le cas contraire, on renvoie la liste de couples (processeur, exception), ce qui correspond au rattrapage d'un ensemble d'exceptions locales.

Un défaut de notre approche utilisant une exception standard Global_exn est que cette exception est visible de l'utilisateur : celui-ci peut la rattraper avec un simple **try with**, ce qui est en désaccord avec notre sémantique. Il est donc nécessaire, encore une fois, d'utiliser camlp4 pour la dissimuler aux yeux de l'utilisateur. On obtient ce résultat en réécrivant les blocs de rattrapage **try with** afin qu'ils transmettent automatiquement toute exception Global_exn avant tout autre cas de filtrage. Ainsi,

```
let x = try [code] with x → [traitement]
```

devient

```
let x = try [code] with
  | Global_exn e → raise (Global_exn e)
  | x → [traitement]
```

Un dernier mécanisme, assez complexe et fortement dépendant de l'implantation d'OCaml, est destiné à afficher les messages d'erreurs clairs lors de la terminaison d'un programme par exceptions non rattrapées. Nous ne le détaillerons pas ici, mais il est nécessaire, à la terminaison du programme, d'une part d'effectuer une barrière afin de récupérer d'éventuelles exceptions locales en attente, d'autre part de détecter une exception Global_exn qui serait la cause de cette terminaison. L'implantation affichera toujours, lorsque le programme se termine, l'ensemble des exceptions levées et les processeurs concernés.

Une solution plus simple est de considérer tout programme comme inclus dans un bloc **trypar withpar** chargé de rattraper et d'afficher toutes les exceptions : notre solution est équivalente à celle-ci du point de vue de l'utilisateur, y compris au niveau de la barrière de synchronisation finale. Cependant, BSML reposant sur le compilateur d'OCaml et sur camlp4, et notamment à cause de la programmation modulaire, il n'est pas possible d'implanter le mécanisme de cette façon. Il en aurait été différemment si nous utilisions notre propre compilateur.

5.3 Combinaison d'extensions

5.3.1 μBSML$^{\mathsf{ref/exn}}$

μBSML$^{\mathsf{ref}}$ et μBSML$^{\mathsf{exn}}$ ont été étudiés en détail dans leurs chapitres respectifs. Cependant, il convient également d'étudier si leur combinaison ne pose aucun problème avant de pouvoir tirer des conclusions sur la sûreté du langage dans son ensemble. Cette étude atteint cependant rapidement une complexité très élevée, c'est pourquoi nous nous contenterons dans cette section d'étudier de façon informelle les problèmes posés, et d'expliquer la façon dont ils sont résolus dans notre implantation.

De même que les références peuvent créer des «fuites» dans les vecteurs parallèles, elles peuvent permettre à des valeurs de s'échapper d'un bloc **trypar withpar** partiellement calculé. Sans remettre en cause la gestion des exceptions, cela réfute l'argument de sûreté qui permet d'affirmer, dans μBSML$^{\mathsf{ref}}$, que le mécanisme d'exceptions empêche tout accès à un vecteur partiellement calculé. Par exemple, dans le programme suivant :

```
val r : int par ref
let r = ref ≪ $this$ ≫

val v : int par
let v =
  trypar
    r := ≪ 1 / $!r$ ≫ ; !r
  withpar
    | ≪ Division_by_zero // _ ≫ → !r
```

Une exception est ici levée localement lors de l'affectation de r. Mais le vecteur résultant, étant enregistré dans la référence globale r, reste accessible après l'échec du calcul une fois l'exception globalisée et rattrapée. L'état du processeur 0, après ce rattrapage, est à nouveau sain, et celui-ci est donc en mesure d'effectuer des opérations locales. Or, si l'on cherche à accéder au vecteur v (ou !r) depuis ce processeur, la valeur calculée est absente.

Il pourrait être tentant de considérer que cette valeur est bien présente et reste la valeur précédente de !r locale à ce processeur. Mais cela introduirait une notion d'«affectation partielle», et l'affectaion := est bien écrite en tant qu'opération répliquée : la sémantique locale/répliqué deviendrait quelque peu obscure. Ce serait, de plus, faire la confusion entre vecteur de références et référence vers un vecteur : le vecteur considéré dans l'exemple ci-dessus doit se limiter à deux cas, soit le calcul échoue, soit il aboutit.

Afin de supprimer toute notion de vecteur partiel, l'approche choisie est la suivante : en cas de calcul local ne pouvant pas retourner de résultat parce qu'une exception a été levée, le vecteur parallèle renvoyé mémorise cette exception. Toute tentative d'accès local à ce vecteur par la suite, détectant l'erreur, relève la même exception. Ainsi, le code ci-dessus est valide, mais le vecteur v résultant est inutilisable : toute tentative d'accès local (par l'intermédiaire d'une primitive ou d'une section locale) lève à nouveau l'exception Division_by_zero sur le processeur 0, celle-ci pouvant être globalisée, propagée et rattrapée par un **trypar withpar** comme une nouvelle exception (elle ne peut plus, cependant, être rattrapée localement).

Au niveau de la sémantique, cela revient à ajouter l'exception levée à l'élément $\perp$ des vecteurs parallèles, et à ajouter à la réduction des primitives le cas correspondant. Un accès à $\perp(e)$ est alors transformé en *faile*. Ce cas excluant toute réduction locale, la confluence n'est pas remise en cause.

Exemple 5.3.1

Notons que la solution ci-dessus, bien que n'étant utile qu'en présence de références, reste indépendante de celles-ci. Le résultat qu'on aurait pu souhaiter dans l'exemple ci-dessus peut être obtenu à l'aide d'un vecteur de références au lieu d'une référence vers un vecteur. C'est un moyen d'utiliser les exceptions locales en ne perdant pas les calculs effectués par les processeurs n'ayant pas levé d'exception (un autre moyen, plus lourd, est de remplacer e *par* try Some e with _ → None *dans toute section locale)*

```
val r : int ref par
let r = ≪ ref $this$ ≫

val v : int par
let v =
```

```
trypar
  ≪ $r$ := 1 / !$r$ ≫ ; ≪ !$r$ ≫
withpar
  | ≪ Division_by_zero // _ ≫ → ≪ !$r$ ≫
```

L'affectation étant faite localement, et le programme n'utilisant que des références purement locales, la cohabitation exceptions/références ne pose plus de problème ici, et le vecteur renvoyé sera $\langle 0, 1, 0, 0, \ldots, 0\rangle$*, la valeur n'ayant pas été mise à jour au processeur 0.*

La différence de comportement entre les deux programmes est donc justifiée par la localité de l'affectation : une affectation locale peut être différente sur les différents processeurs, une affectation globale se doit d'être répliquée.

Une dernière précaution concerne la structure **trypar withpar** : en effet, en cas d'exception, celle-ci fait d'une valeur locale une valeur globale. Dans le cas où l'exception locale levée contiendrait des références, celles-ci sont transformées de la même façon que pour **proj**. La règle de typage de **trypar** devient donc la suivante :

$$\text{TRYPAR}\quad \frac{\mathrm{C}, \Gamma \vdash e : \tau/g \qquad \mathrm{C}, (\Gamma; x : \mathsf{glob}(\tau_{\mathsf{exn}}\,\mathsf{array})) \vdash e' : \tau/g}{\mathrm{C}, \Gamma \vdash \mathsf{trypar}\, e\, \mathsf{withpar}\, x \rightarrow e' : \tau/g}$$

Il nous faut également assurer que la contrainte $\tau_{\mathsf{exn}}(\nrightarrow)$ est respectée, ce qui est une restriction raisonnable.

5.3.2 Implantation de la superposition parallèle

La superposition parallèle pose un problème technique, car elle ne suit pas un fil d'exécution linéaire. En effet, les super-threads doivent pouvoir être interrompus au moment de leur barrière de synchronisation, et rétablis dans le même état de leur pile d'exécution au moment de leur reprise. Le modèle fonctionnel ne permet pas directement de jouer ainsi avec la pile d'exécution.

Deux solutions existent. La solution actuelle, utilisée dans l'implantation de BSML et que nous détaillerons ici, utilise les *threads* du système. Une autre solution, qui est en cours d'étude pour une nouvelle version de BSML, utilise une transformation partielle vers un programme de style passage de continuations (CPS) [GG08], à l'aide d'une analyse de flot monovariante.

Lors de l'utilisation de la primitive **super**, deux nouveaux threads système sont créés. Néanmoins, afin de préserver le déterminisme du système, on souhaite que ces deux threads ne s'exécutent pas de façon concurrente, comme cela se ferait naturellement dans le système, mais strictement l'un après l'autre, jusqu'à leur première barrière. Un ordonnanceur interne à BSML assure ainsi qu'un seul thread peut être actif à la fois.

Un arbre binaire des threads présents, partagé, est gardé en mémoire. L'utilisation de **super** crée deux fils au nœud courant de l'arbre ; lorsqu'un superthread atteint sa barrière, ses informations de communication sont mémorisées dans le nœud correspondant, et il se met en sommeil, laissant la main au suivant.

Cela est fait par la fonction next, qui effectue un parcours des feuilles de l'arbre uniquement, de la gauche vers la droite. Lorsque toutes les feuilles ont été parcourues, la super-étape généralisée

peut se terminer et les communications mises en attente ont lieu, puis l'exécution reprend à la première feuille à gauche. Le super-thread parent est réveillé dès que ses deux fils se sont terminés, et on lui renvoie le couple des valeurs de retour de ses fils, qui ont été mémorisées.

5.3.3 Combinaison avec les traits impératifs

Les super-threads peuvent interférer avec les références, mais grâce à l'ordonnancement statique leurs accès à celles-ci restent déterministes – c'est d'ailleurs la principale raison qui a poussé à ce choix [Gav05].

La combinaison des super-threads avec la gestion des exceptions, quant à elle, pose de nombreux problèmes, d'ordre conceptuel aussi bien que dans l'implantation. En effet, deux super-threads peuvent lever des exceptions indépendemment, localement ou globalement, se continuer ou se terminer ; dans chacun de ces cas, il faut déterminer ce qui prendra le pas – mais les exceptions locales ne peuvent être rassemblées qu'à la super-étape suivante, celles-ci n'étant détectées qu'après les communications.

Une multitude de cas délicats peuvent ainsi se produire. À partir du moment où l'un de ses fils s'est terminé par une exception, la primitive **super** doit se terminer en transmettant cette exception ; dans le cas où on ne peut avoir que des exceptions répliquées, cela est relativement simple, et l'ordre d'exécution des threads étant déterministe, on peut rappeller le père en lui indiquant l'exception levée, et il se charge de mettre fin à son autre fils avant de relever l'exception.

Dans le cas où on a également des exceptions locales, cependant, celles-ci ne peuvent être détectées immédiatement. Par exemple, dans le cas où le fils gauche a levé une exception locale, puis effectué une barrière et ainsi passé la main au fils droit, si ce dernier lève une exception répliquée, celle-ci étant postérieure à l'exception locale devrait être abandonnée au profit de la première. Cela signifie qu'on devrait établir une barrière avant que l'appel à **super** ne puisse se terminer, malgré l'exception répliquée levée par un des fils.

Cela se complique encore si l'on considère le fait qu'un super-thread peut lever des exceptions locales puis une exception répliquée durant la même-super étape, et qu'un super-thread peut aussi se terminer sans barrière de communication, auquel cas il devrait directement rendre la main à son père, mais en lui transmettant d'éventuelles exceptions locales. On pourrait alors se retrouver avec des ensembles d'exceptions locales comportant plusieurs éléments pour le même processeur ; *etc.*

Voici l'approche employée : lorsque les deux nouveaux threads ont été créés suite à **super**, le père appelle une fonction wait et donne la main à son premier fils. Tant que l'exécution continue normalement et qu'aucun des fils ne se termine, le père ne sera pas exécuté par l'ordonnanceur puisque celui-ci ne donne la main qu'aux feuilles de l'arbre. Il ne reprendra donc la main que dans l'un des cas suivants :

- Ses deux fils se sont terminés, normalement ou par une exception répliquée.
- Un de ses fils vient de récupérer un ensemble d'exceptions locales (qui ont donc été levées à la super-étape précédente).
- Un de ses fils a levé une exception répliquée, et l'autre fils s'est exécuté une fois depuis (ce qui lui a laissé la chance de récupérer une exception locale qui aurait été levée auparavant, chaque super-thread ne s'exécutant qu'une fois par super-étape).

Notons que, lorsqu'un fils se termine, on le laisse dans l'ordonnanceur pendant une super-étape supplémentaire si son frère n'est pas terminé, afin que des exceptions locales qu'il aurait laissées en suspens puissent se propager jusqu'à leur père. Ce mécanisme a le défaut de pouvoir causer une super-étape supplémentaire en cas d'exception répliquée, mais il nous garantit que l'exception qui a été levée la première est celle qui est levée, qu'elle soit locale ou répliquée, ce qui est cohérent avec le reste de notre sémantique sur les exceptions.

Un dernier cas problématique se présente : dans le cas où la main revient au père sans que la cause en soit une exception locale, l'état des processeurs doit être préservé puisque cela n'implique pas de barrière, et qu'un des processeurs pourrait ne pas disposer de certains résultats locaux. Par exemple, le programme suivant :

```
super (fun () → ≪ 1/$this$ ≫ ) (fun () → raise Exit)
```

n'effectue aucune communication et laisse le processeur 0 en état d'exception locale. Il est donc nécessaire que le père, dans ce cas, hérite de l'état des processeurs de ses fils. Mais on se retrouve face à un choix dans le cas où le père hérite de ses deux fils :

```
super (fun () → ≪ 1/$this$ ≫ ) (fun () → ≪ if $this$ < 2 then raise Exit ≫ )
```

Que faire dans ce dernier cas ? Généraliser les ensembles d'exceptions pour qu'ils acceptent potentiellement plusieurs exceptions par processeur augmenterait la complexité du mécanisme de gestion des exceptions pour une utilité somme toute mineure : on choisit donc dans ce cas de donner priorité à la première exception ayant été levée sur chaque processeur. L'exemple ci-dessus nous donnera donc l'ensemble $\{(0 \mapsto \text{Division\_by\_zero}), (1 \mapsto \text{Exit})\}$

5.4 Traits additionels : implantation de la syntaxe alternative et du filtrage parallèle

5.4.1 Syntaxe alternative

L'implantation de la syntaxe alternative est une simple réécriture des sections locales ≪ ≫ en code utilisant les primitives. Nous avons choisi d'utiliser les «quotations» de camlp4 pour intégrer ces sections locales : d'autres approches étaient possibles, mais offraient une syntaxe moins agréable compte tenu du parseur OCaml. En effet, sans modifier celui-ci, les possibilités pour définir de nouveaux caractères délimitant un bloc (tels que les parenthèses) sont limitées.

La transformation procède comme suit : à l'entrée d'une section locale, une référence est créée pour contenir la liste de ses paramètres. Lors de l'analyse de son corps, une «antiquotation», c'est-à-dire la syntaxe x, va donner lieu à un traitement particulier : on crée une variable fraîche fresh_id[13] qui va remplacer l'élément x, et on ajoute à la liste des paramètres le couple (fresh_id,x). Dans le cas où x est un simple identifiant, on vérifie qu'il n'est pas déjà dans les paramètres, et réutilise la variable lui correspondant le cas échéant afin d'éviter d'imbriquer un apply par instance de la même variable.

Une fois le corps e de la section locale analysé de la sorte, il suffit de le transformer en mkpar (fun fresh_id_this → e), où fresh_id_this correspond à la variable fraîche qui a été attri-

[13] simulée par le préfixe __Li_, camlp4 n'offrant pas de réel mécanisme pour les variables fraîches

buée à **this**, puis de l'englober dans des appels à **apply** par récursivité sur la liste des paramètres. Cette dernière étape est réalisée par la fonction :

```
let rec build_apply _loc e explist = match explist with
  | [] → e
  | ex::exr → <:expr< Bsml.apply $build_apply _loc e exr$ $ex$ > >
```

où la syntaxe <:expr< ... > > permet de construire un nœud de type expression dans l'arbre de syntaxe. La totalité de cette transformation fait moins d'une centaine de lignes de code : camlp4 est un outil puissant bien qu'ardu.

5.4.2 Implantation du filtrage parallèle par motifs

Cette section décrit en détail l'implantation du filtrage parallèle par motifs, qui a été décrit à la section 2.8.

Comme nos autres extensions syntaxiques, celle-ci est implantée à l'aide de Camlp4. L'expression à filtrer est d'abord projetée en une liste, puis la correspondance de cette liste à chacun des motifs parallèles est testée de la façon suivante : on élimine successivement de la liste les éléments correspondant à chacun des motifs standard apparaissant dans le motif parallèle, concluant à un échec si aucun élément n'a été trouvé ; le motif _ absorbe quant à lui la totalité de la liste. Une fois tous les motifs testés de la sorte, si la liste restante est vide, le motif parallèle correspond à la valeur. Dans le cas contraire, le motif parallèle suivant est examiné.

La difficulté de l'implantation de ce mécanisme sous forme de réécriture automatique vient du fait que les motifs parallèles réutilisent, pour la correspondance des éléments du vecteur parallèle, les motifs OCaml standard : cela est à la fois plus satisfaisant et plus puissant que des motifs simplifiés que nous aurions pu implanter par ailleurs. Mais notre transformation automatisée va devoir construire une fonction qui se charge de faire ce filtrage par motifs standard, et qui sera appliquée sur la liste à l'aide des opérateurs adéquats. Le problème est que la liaison des variables du motif se fera à l'intérieur de cette fonction : la transformation commence par extraire la liste de ces variables, puis crée une construction **let** (var1,...,varn) = [filtrage parallele] **in** [expr] afin que les variables soient liées comme souhaité (le filtrage parallèle renvoyant des listes dans le cas de motifs avec étoile).

Présenter ici le code de la transformation, fort complexe et peu lisible, aurait peu d'intérêt ici : on se concentrera sur un exemple simple de transformation. Soit le programme suivant :

```
val f : 'a option par → 'a option
let f x = matchpar x withpar
  | ≪ Some v at p // (None)* ≫ → Some v
  | ≪ _ ≫ → None
```

Celui-ci, après transformation et ajout de commentaires, devient :

```
let f x =
  (* l'argument initial est d'abord projeté sous forme de liste de paires (pid,valeur) *)
  let __l = Bsml.proj_list_pids x
  in
```

```
(* bloc try..with correspondant au motif parallèle ≪ Some v at p // (None)* ≫ *)
try
  let (v, p, __l) =
    let rec __bsml_patt_aux acc = (* filtrage pour (Some v at p) *)
      function
      | x :: r →
          (match x with
           | (p, Some v) → (v, p, (List.rev_append acc r))
           | x → __bsml_patt_aux (x :: acc) r)
      | [] → raise Bsml.Parmatch_nextcase
    in __bsml_patt_aux [] __l in
  let (__doesmatch, __l) = (* filtrage pour (None)* *)
    List.fold_right
      (fun x (__doesmatch, __l) →
         match x with
         | (_, None) → (true, __l)
         | _ → (__doesmatch, (x :: __l)))
      __l (false, [])
  in
    (* on vérifie que toute la liste a été filtrée entièrement,
       et applique le resultat (Some v) le cas échéant *)
    if not __doesmatch
    then raise Bsml.Parmatch_nextcase
    else if __l <> [] then raise Bsml.Parmatch_nextcase else Some v
with
| Bsml.Parmatch_nextcase →
  (* motif suivant: motif parallèle ≪ _ ≫ *)
    (try
       let __l = []
       in if __l <> [] then raise Bsml.Parmatch_nextcase else None
  (* cas d'échec du filtrage *)
     with | Bsml.Parmatch_nextcase → raise Bsml.Parmatch_failure)
```

On remarque l'emploi de noms de variables préfixés par __, destiné à combler une limitation de Camlp4, qui ne permet pas d'obtenir automatiquement des variables garanties fraîches : faute de mieux, on interdit ce préfixe dans les noms de variables utilisateur afin d'éviter toute possibilité de conflits.

Le code transformé a la structure suivante :

1. On utilise la fonction proj_list_pids pour projeter le vecteur parallèle à filtrer et obtenir une liste (pid,valeur).
2. Les différents motifs parallèles sont chaînés entre eux à l'aide d'exceptions : chaque bloc de filtrage est placé dans une structure **try with**, et lève l'exception Bsml.Parmatch_nextcase en cas de non correspondance : l'exception rattrapée amène au bloc de filtrage correspondant au motif parallèle suivant, et ainsi de suite jusqu'à correspondance ou échec. Pour le cas d'échec, un dernier rattrapage d'exception est ajouté et lève l'exception Bsml.Parmatch_failure, similaire à l'exception Match_failure d'OCaml.

3. À l'intérieur de chaque motif parallèle, on construit une fonction pour chaque motif : un motif sans étoile donne lieu à une fonction de recherche appliquée simplement à la liste initiale, un motif avec étoile donne lieu à une fonction de filtrage qui sera appliquée à l'aide de List.fold_left. Dans un cas comme dans l'autre, la fonction renvoie un n-uplet comprenant les variables liées par le motif, et la liste des éléments non filtrés restants.
4. Une fois la fin du motif parallèle atteinte, on teste si la totalité de la liste a été filtrée : si c'est le cas, le motif est validé et l'expression correspondante évaluée dans un contexte comportant les variables liées par le motif, sinon l'exception Bsml.Parmatch_nextcase renvoie au cas suivant.

5.5 Exemple : recherche en profondeur

5.5.1 Algorithme séquentiel

La recherche en profondeur, ou backtracking, est souvent utilisée pour résoudre des problèmes de satisfaction de contraintes : c'est une méthode générique qui consiste, en sous-divisant récursivement l'espace des réponses au problème, à explorer toutes les possibilités qui sont susceptibles de conduire à une solution. On n'y a généralement recours que pour résoudre des problèmes NP-difficiles, quand il n'existe pas de solution *ad hoc* connue.

On représente généralement le backtracking comme une exploration en profondeur de l'espace des réponses organisé sous forme d'arbre, en coupant toute branche qui ne conduira à coup sûr à aucune réponse. Par exemple, la satisfaisabilité d'une formule logique de n variables booléenes (problème SAT) peut s'étudier en décomposant l'ensemble des possibilités en arbre binaire de profondeur n ; un nœud pour lequel les variables fixées permettent de réduire la formule en *faux* permet de couper les branches qui en descendent. Si l'on atteint une feuille, la formule est satisfaite et le problème résolu.

De part la nature exponentielle des problèmes à résoudre, le backtracking est rapidement très coûteux en temps. À titre d'exemple, nous proposons donc ici une implantation générique du backtracking en BSML. La difficulté consiste à répartir de façon équitable les nœuds à explorer entre les différents processeurs et à rééquilibrer quand nécessaire, le facteur de branchement n'étant pas prévisible dans le cas général.

Pour cet exemple, on suppose le problème fourni sous forme de la racine de l'arbre et d'une fonction childs qui teste si un nœud est solution (et lève, si c'est le cas, une exception) et renvoie ses fils sous forme de liste dans le cas contraire. L'utilisation d'une exception permet de rattraper la solution trouvée très simplement malgré les niveaux de récursion et nous permettra de montrer une utilisation de la gestion des exceptions parallèles. C'est un des cas où l'utilisation d'exceptions est particulièrement pratique pour le programmeur.

Une solution séquentielle du problème pourrait s'écrire, très simplement :

```
val descend : Problem.t → unit
let rec descend n =
  ignore (List.rev_map descend (Problem.childs n))

val backtrack : unit → Problem.t option
let backtrack () =
```

```
try
  descend Problem.root; None
with
  | Problem.Solution_found s → Some s
```

La fonction descend se charge, par appels récursifs, de l'exploration de l'arbre. Elle ne renvoie pas de résultat, c'est l'exception levée par la fonction Problem.childs qui s'en chargera le cas échéant. Nous utilisons la fonction de la bibliothèque standard List.rev_map, équivalente à la composée de List.rev et List.map, mais qui est récursive terminale et plus performante : pour notre application, l'ordre d'exploration importe peu.

5.5.2 Version parallèle naïve

La version parallèle suit le même procédé, à la différence que les nœuds seront distribués dans les composantes d'un vecteur parallèle. Une première solution, extrêmement naïve, pourrait être :

```
val backtrack : unit → Problem.t option
let backtrack () =
  trypar
    let init = select_list (Problem.childs Problem.root)
    in ≪ descend $init$ ≫ ; None
  withpar
    | ≪ Problem.Solution_found s // _ ≫ → Some s
```

Ici, on commence par calculer un ensemble de nœuds initial (en utilisant Problem.childs Problem.root) que l'on distribue sur les processeurs. La descente est ensuite effectuée de façon séquentielle sur chaque processeur : quand l'un d'entre eux trouvera une solution, il lèvera localement l'exception Problem.Solution_found qui sera rattrapée globalement par la structure trypar withpar.

Cette approche simpliste a deux principaux défauts : elle n'effectue aucun équilibrage, et suppose par conséquent que l'ensemble de nœuds initial pourra être réparti de façon équitable sur tous les processeurs, et que les sous-ensembles locaux entraîneront une quantité comparable de calculs. Ce n'est généralement pas le cas, certaines branches étant coupées rapidement ce qui entraînerait ici le regroupement des calculs sur un seul processeur.

Deuxième défaut, le programme est entièrement asynchrone, et fonctionne en une seule super-étape : suivant notre modèle d'exceptions, une solution ne pourra être propagée qu'à la fin de cette super-étape et donc une fois que tous les processeurs auront fini leurs calculs, La durée sera donc la même si un processeur trouve une solution rapidement, et la quasi-totalité de l'arbre aura du être explorée dans tous les cas. Notons que ce problème sera résolu en se chargeant du précédent, le rééquilibrage nécessitant des barrières. Cette méthode a le mérite d'une grande simplicité, mais une méthode plus sérieuse requiert donc d'implanter le parallélisme dans la fonction descend et non pas uniquement dans backtrack. L'intérêt de la récupération globale d'exceptions locales se révèle alors.

5.5.3 Version parallèle efficace

Une solution plus avancée consiste donc à découper le calcul en super-étapes, celles-ci effectuant une partie d'exploration suivie d'un rééquilibrage éventuel, qui sera l'occasion de détecter une solution éventuelle. Ce découpage peut s'effectuer de plusieurs façons : une des plus simples consisterait, en conservant un ensemble réparti de nœuds dont les fils n'ont pas encore été explorés, à autoriser un nombre d'opérations locales maximal pour chaque super-étape.

Notre approche est légèrement plus stricte : elle garde les calculs locaux au même niveau de profondeur dans l'arbre. Le taux de branchement, bien qu'aléatoire en général, dépend en effet le plus souvent de la profondeur. En explorant ces niveaux l'un après l'autre, on diminue les chances d'un écart rapide entre la charge des différents processeurs, et on limite donc les besoins de rééquilibrage. À chaque super-étape, la machine parallèle va donc explorer une forêt correspondant à des sous-arbres dont les racines forment un sous-ensemble des nœuds de niveau n dans l'arbre à explorer. Cette super-étape se terminera lorsque tous les descendants de niveau n' de ces nœuds ont été calculés, ceux-ci étant rééquilibrés et partitionnés, et donnant lieu à une nouvelle super-étape. Le paramètre nchilds détermine la largeur de la sous-forêt explorée à chaque super-étape par chaque processeur.

```
val rev_split : 'a list → int → 'a list → 'a list * 'a list
let rec rev_split acc n = function
  | [] → acc,[]
  | x::r → if n = 0 then acc,x::r else rev_split (x::acc) (n−1) r

val descend : Problem.t list par → unit
let rec descend nodes = (* récursif sur la profondeur *)
  matchpar ≪ List.length $nodes$ ≫ withpar
    | ≪ (0)* ≫ → () (* backtracking *)
    | ≪ (nl)* ≫ →
        let nodes, ntot, max_sz = rebalance_if_needed nodes nl
        in explore nodes max_sz

val explore : Problem.t list par → int → unit
and explore nodes max_sz = (* récursif sur la largeur *)
  let split_nodes = ≪ rev_split [] nchld $nodes$ ≫ in
  let current_nodes, next_nodes = ≪ fst $split_nodes$ ≫ , ≪ snd $split_nodes$ ≫ in
    descend ≪ List.flatten (List.map Problem.childs $current_nodes$) ≫ ;
    if max_sz − nchld <= 0 then ()
    else explore next_nodes (max_sz − nchld)

val backtrack : unit → Problem.t option
let backtrack () =
  trypar
    let init = ≪ if $this$=0 then Problem.childs Problem.root else [] ≫
    in descend init; None
  withpar
    | ≪ Problem.Solution_found s // _ ≫ → Some s
```

L'algorithme comporte deux niveaux de récursivité : la fonction descend en elle-même est récursive par rapport à la profondeur de l'arbre, et commence par procéder au rééquilibrage (en calculant les tailles locales des listes, et par un appel à la fonction rebalance_if_needed qui prend la décision de rééquilibrer en fonction de ce résultat, renvoyant dans tout les cas une liste parallèle distribuée équitablement, ainsi que le nombre total d'éléments et la taille locale maximale). La fonction explore se charge de l'exploration en largeur : elle sélectionne récursivement une partie des nœuds disponibles sur chaque processeur à l'aide de split_nodes et procède sur cette partie à une exploration en profondeur par descend.

Chacune des deux fonctions retourne () lorsqu'elle n'a plus de nœuds à explorer, laissant l'appellant reprendre l'exploration, ce qui correspond au «backtracking». Le lecteur pourra être surpris par le fait qu'on impose une barrière à chaque niveau de descente dans l'arbre : lorsqu'on applique l'algorithme à un problème donné, on compose plusieurs fois la fonction childs donnée, et ce afin d'obtenir un taux de branchement suffisant pour que l'équilibrage puisse se faire. La fréquence des barrières peut donc être ajustée par ce niveau de composition et par le paramètre nchld.

5.5.4 Rééquilibrage

Il nous reste à définir une fonction capable de rééquilibrer les données. De nombreuses approches existent ; dans notre cas, la liste est non ordonnée, on se contentera donc de calculer le nombre d'eléments désirés sur chaque processeur, et de déplacer les éléments excedentaires d'un processeur i vers le premier processeur «cycliquement à gauche» de i (c'est-à-dire le premier de la liste $[(i-1) \mod p, \ldots, (i-p+1) \mod p]$) ayant un déficit en éléments.

La fonction de rééquilibrage prend en paramètres la liste distribuée v, la liste sizes de longueur p comportant les tailles locales des listes sur chaque processeur, et la taille totale de la liste distribuée ntot (la somme de sizes). Le processeur i se verra attribuer, à la fin du rééquilibrage, $(i+1)*\text{ntot}/p - i*\text{ntot}/p$ éléments. La liste d'associations deltas calcule, pour chaque processeur, son écart en nombre d'éléments par rapport à la situation finale. La fonction find_dest se charge, pour un processeur pid en excédent d'eléments, de déterminer comment ces éléments devront être répartis sur les autres processeurs : elle est récursive sur la liste des processeurs cycliquement à gauche de pid, et renvoie une liste d'associations (processeur, nombre d'eléments). Par souci de simplicité, les éléments non déplacés sont considérés comme envoyés sur leur processeur d'origine, ce qui ne provoque pas de communications.

```
val rebalance : 'a list par → int list → int → 'a list par
let rebalance v sizes ntot =
  let deltas = List.map
    (fun (i,n) → i, n − ((i + 1) ∗ ntot / bsp_p − i ∗ ntot / bsp_p))
    (List.combine procs_list sizes) in
  let rec find_dest pid size data nlost = function
    | (i,x)::r →
        let delta = nlost + x in
          if delta >= 0 then find_dest pid size data delta r
          else if size > − delta then
            let d1,d2 = rev_split [] (−delta) data in
              (i,d1) :: find_dest pid (size + delta) d2 0 r
          else
```

```
            let d1,d2 = rev_split [] size data in
              [ (i,d1); (pid,d2) ]
  in
  let sendto_list =
    ≪ let left,(i,d)::right = rev_split [] $this$ deltas
      in if d>0 then find_dest $this$ d $v$ 0 (left @ List.rev right)
                else [ $this$, $v$ ] ≫
  in let exch = put ≪ fun sendto → try List.assoc sendto $sendto_list$
                                   with Not_found → [] ≫
  in ≪ unordered_flatten (List.map $exch$ procs_list) ≫
```

Les communications ainsi calculées sont ensuite réalisées par un appel à **put**, et réassemblées sous forme de liste distribuée par un appel local à unordered_flatten (version optimisée, non ordonnée et récursive terminale de List.flatten). Il nous reste à déterminer un critère de rééquilibrage, afin d'éviter les communications inutiles. Notre critère est ici nmax − nmin > ntot / **bsp_p** + 1, qui produit un rééquilibrage si les écarts entre les quantités de données locales relatifs à la charge totale sont trop grands, mais en tolérant au minimum un écart de 1 pour les ensembles de petite taille.

```
val rebalance_if_needed : 'a list par → int list → 'a list par * int * int
let rebalance_if_needed v sizes =
  let nmin,nmax,ntot =
    List.fold_left (fun (nmin,nmax,ntot) i → min i nmin, max i nmax, i+ntot)
      (max_int,0,0) sizes
  in
    if nmax − nmin <= ntot / bsp_p + 1 then v,ntot,nmax
    else
      rebalance v sizes ntot,
      ntot,
      (ntot/bsp_p + if ntot mod bsp_p = 0 then 0 else 1)
```

5.5.5 Résultats

Un exemple amusant, mais néanmoins intéressant d'utilisation de cet algorithme est la résolution d'un problème de Sudoku. L'engouement international pour ce jeu n'a pas épargné la communauté scientifique ; on trouve en particulier des travaux le ramenant à un problème SAT [LO06], et une variété d'algorithmes de simplification et de résolution. Nous ne nous interesserons pas ici à la structure interne et aux propriétés mathématiques du problème, nous appliquant à en trouver une solution par simple force brute, afin de tester notre algorithme de backtracking. De ce point de vue, le problème a la particularité d'avoir un taux de branchement très variable, rendant indispensable le rééquilibrage régulier des données.

Un problème de sudoku de dimension n se compose d'une grille de taille n^2 sur n^2 – décomposée en $n \times n$ sous-grilles de taille n sur n, partiellement remplie de nombres de 1 à n^2. L'objectif est de remplir la totalité de la grille de telle sorte que chaque ligne, chaque colonne et chaque sous-grille comporte une fois et une seule chaque nombre de 1 à n^2. Un problème est bien formé si les nombres initiaux assurent qu'il existe une seule solution ; il est minimal si on ne peut en enlever de donnée sans qu'il perde son caractère bien formé.

Notre résolution par force brute consiste à essayer, pour chaque case libre, toutes les possibilités n'aboutissant pas une contradiction directe des contraintes. La fonction childs renvoie donc, pour une grille donnée, les différentes possibilités de remplissage de la première case libre de cette grille. Le taux de branchement sera ainsi élevé en début de ligne, et faible en fin de ligne, de colonne ou de sous-grille, rendant l'équilibrage du calcul difficile.

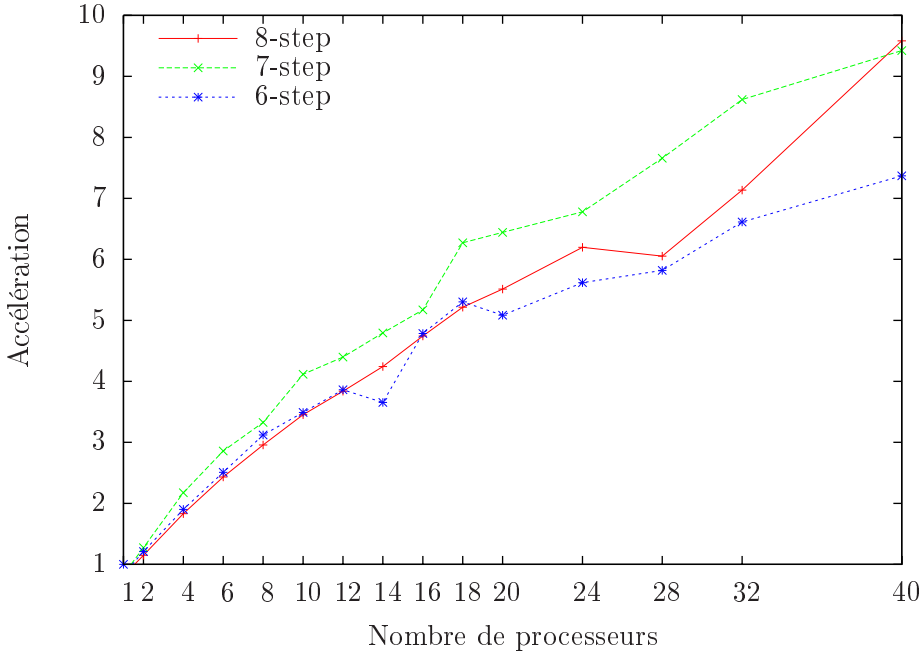


FIG. 5.1 – Vitesse d'exécution pour la résolution d'un sudoku 16×16 par backtracking.

La figure 5.1 montre les résultats d'exécution sur la grappe du LACL, composée de 20 processeurs dual-core Intel E2180 à 2GHz munis de 2Go de mémoire. Le problème consistant en une recherche de solution, son temps d'exécution est très aléatoire : en effet, cette solution peut être trouvée dès le début du calcul sur une machine lente alors qu'une machine rapide, lors d'une exécution donnée, aura à explorer une grande partie de l'espace des réponses et mettra donc plus de temps. L'approche usuelle serait d'effectuer le test sur un très grand nombre d'exemples aléatoires afin de calculer une moyenne, mais dans notre cas, une approche plus économique est possible : on sait que le problème ne comporte qu'une solution, et donc que l'espérance statistique du temps de calcul correspond à l'exploration de la moitié de l'espace des réponses.

Le graphique est donc basé sur le temps mis par le programme pour explorer chaque nœud de l'arbre, ce qui est proportionnel à l'espérance du temps d'exécution. La valeur en ordonnée correspond au nombre de nœuds explorés par seconde, mise en rapport avec cette valeur pour une exécution séquentielle. On constate que, pour un choix judicieux du paramètre n−step, correspondant au nombre de cases que chaque processeur tente de remplir avant de terminer la super-étape, l'accélération approche de 10 pour 40 processeurs. Ce résultat, sans être extraordinaire (l'implantation naïve du problème de sudoku génère une grande quantité de communications), nous montre que l'accélération reste plutôt stable lorsqu'on augmente le nombre de processeurs et donc les risques de déséquilibre.

5.6 Exemple : distribution d'arbre par m-ponts et application à PPP

Ce deuxième exemple est basé sur [MH06] et nous permettra d'anticiper légèrement sur la section suivante à propos des squelettes. Il se décompose en deux parties, la première concernant une méthode de découpage équilibré d'un arbre entre p processeurs, que nous implantons en BSML. La seconde utilise ce découpage pour résoudre un problème de marquage maximal particulier dans un arbre, le problème de l'organisation de soirée (*Party Planning Problem*, que nous abrégerons ppp).

5.6.1 Découpage d'arbre par m-ponts

Dans le cas de la contraction comme dans de nombreux autres, deux critères sont à prendre en compte lors de la division d'une structure de données entre les processeurs :

1. La localité : les connexions entre les sous-divisions de la structure doivent être minimales, afin de réduire au maximum la quantité de communications nécessaires.
2. L'équilibrage, afin d'utiliser au maximum le parallélisme.

Nous décrivons ici le découpage d'un arbre par m-ponts tel qu'il est utilisé dans [MH06], tiré de [Rei93]. Quelques définitions de théorie des graphes sont nécessaires. On suppose m, un entier strictement supérieur à 1, et inférieur à la taille de l'arbre considéré ; on note $size_b(v)$ la taille du sous-arbre prenant racine au nœud v.

Définition 5.6.1 (*Nœud m-critique*)

Un nœud interne v est un nœud m-critique si, et seulement si, pour tout nœud v' fils de v, on a l'inégalité suivante : $\lceil size_b(v)/m \rceil > \lceil size_b(v')/m \rceil$.

Définition 5.6.2 (*m-pont*)

Un m-pont est un ensemble maximal de nœuds qui ne comporte des nœuds m-critiques qu'à ses extrémités (racine ou feuilles).

Les m-ponts ont un certain nombre de propriétés intéressantes. parmi celles-ci, celles qui nous intéressent le plus sont les suivantes (voir [Rei93] pour plus de détails) :

1. Si on ne considère pas sa racine, chaque m-pont comporte au plus un nœud m-critique, que nous appellerons son nœud terminal. Les m-ponts forment donc une structure d'arbre.
2. Un m-pont comporte au maximum $m + 1$ nœuds.
3. Un arbre de taille N comporte au plus $2N/m - 1$ nœuds m-critiques, et, dans le cas d'un arbre binaire, au moins $(N/m - 1)/2$.

Sans s'attarder sur les détails, il s'ensuit qu'avec un bon choix de m par rapport au nombre de processeurs, on a un nombre suffisant de m-ponts pour qu'ils puissent être répartis de façon équitable. Ceux-ci gardent une forte localité, ce qui permet d'appliquer des fonctions de réduction et d'accumulation (vers le haut ou vers le bas) efficacement sur l'arbre ainsi débité. Ces fonctions forment des squelettes, et un traitement préliminaire basé sur la programmation dynamique permet de dériver d'une accumulation sur un arbre un algorithme sur les m-ponts [MH06] ; nous reviendrons sur cette approche dans la partie suivante.

Notre implantation est basée sur des arbres binaires, définis comme suit :

```
type ('a,'b) btree =
  | BLeaf of 'a
  | BNode of ('a,'b) btree * 'b * ('a,'b) btree
  | BTerm of 'b

type ('a, 'b) m_bridge = (('a, 'b) btree, ('a, 'b) btree) btree
```

On distingue dans la définition d'un m-pont les feuilles (BLeaf) des nœuds critiques (BTerm). La reconstitution de l'arbre d'origine à partir de l'arbre de ses m-ponts consiste ainsi, pour chaque m-pont, à recoller les racines de ses deux fils au niveau de son nœud critique :

```
val un_m_bridges : ('a, 'b) m_bridge → ('a, 'b) btree
let rec un_m_bridges = function
  | BLeaf l → l
  | BNode (l,BLeaf lf,r) → BLeaf lf
  | BNode (l,BNode (l',v,r'),r) →
      BNode (un_m_bridges (BNode(l,l',r)), v, un_m_bridges (BNode(l,r',r)))
  | BNode (l,BTerm t,r) → BNode (un_m_bridges l, t, un_m_bridges r)
```

Le découpage se fait pour sa part à l'aide de

```
val m_bridges : int → ('a,'b) btree → ('a,'b) m_bridge
let m_bridges m t =
  let t' = sizes t in
  let ceildiv m n = if m mod n=0 then m / n else 1+m/n in
  let rec div = function
    | BLeaf (v,_) → BLeaf (BLeaf v)
    | BNode (l,(v,s),r) →
        let tl = div l and tr = div r in
          if s > m * ceildiv (value l) m && s > m * ceildiv (value r) m
          then BNode (tl,(BTerm v),tr)
          else match tl,tr with
            | BLeaf(l1), BLeaf(r1) → BLeaf(BNode(l1,v,r1))
            | BNode(l1, v1, r1), BLeaf(lf) → BNode(l1, BNode(v1, v, lf), r1)
            | BLeaf(lf), BNode(l1, v1, r1) → BNode(l1, BNode(lf, v, v1), r1)
  in div t'
```

La répartition des m-ponts peut ensuite se faire sur la machine parallèle : la méthode la plus simple consiste à convertir l'arbre en liste par parcours gauche, puis à distribuer la liste comme nous l'avons déjà vu.

Parmi les fonctions utiles sur les arbres, citons la réduction, qui calcule en chaque nœud une valeur dépendant des valeurs obtenues sur ses descendants :

```
val reduce : ('a → 'b → 'a → 'a) → ('a, 'b) btree → 'a
let rec reduce k = function
  | BLeaf a → a
  | BNode(r,b,l) → k (reduce k r) b (reduce k l)
```

L'accumulation vers le haut étant l'arbre constitué des résultats intermédiaires de reduce. L'accumulation vers le bas quant à elle calcule en chaque nœud un résultat dépendant de ses ancêtres :

```
val dAcc :
  ('a → 'b → 'a) →
  ('a → 'b → 'a) → 'a → ('c, 'b) btree → ('a, 'a) btree

let rec dAcc gl gr c = function
  | BLeaf a → BLeaf c
  | BNode(l,b,r) →
      let l' = dAcc gl gr (gl c b) l
      and r' = dAcc gl gr (gr c b) r
      in BNode(l',c,r')
```

L'implantation parallèle de ces squelettes, s'étendant aux arbres découpés par m-ponts, ne sera pas détaillée ici. Elle est plus complexe et dépend de fonctions auxiliaires phi et psi ; à titre de simple exemple :

```
val dAcc_par :
  ('a → 'b → 'a) →
  ('a → 'b → 'a) →
  ('a → 'c → 'a) →
  ('b → 'c) →
  ('b → 'c) →
  ('c → 'c → 'c) →
  'a → ('d, 'b) parallel_btree → ('a, 'a) parallel_btree

let dAcc_par gl gr psid phil phir psiu c (ParTree t) =
  let rec floc = function
    | BLeaf l → None
    | BTerm b → Some (phil b, phir b)
    | BNode (l,b,r) → match (floc l, floc r) with
        | Some (toL,toR), None → Some (psiu (phil b) toL, psiu (phil b) toR)
        | None, Some (toL,toR) → Some (psiu (phir b) toL, psiu (phir b) toR)
        | None, None → None
  in
    (* étape 1 *)
  let local_res = map_scattered_tree floc t in
    (* étape 2 *)
  let toLR = gather_tree local_res in
    (* étape 3 *)
  let glo_dacc = dAcc (fun c (Some (l,r)) → psid c l) (fun c (Some (l,r)) → psid c r) c toLR in
    (* étape 4 *)
  let scatt_acc = scatter_tree glo_dacc in
    (* étape 5 *)
  let rec loc_upd c = function
    | BLeaf l → BLeaf c
    | BTerm b → BTerm c
```

```
  | BNode (l,b,r) → BNode (loc_upd (gl c b) l, c, loc_upd (gr c b) r)
in
  ParTree (zip_scattered_tree loc_upd loc_upd scatt_acc t)
```

5.6.2 Application à PPP

PPP, pour *Party Planning Problem* (problème de l'organisation de soirée) est un problème de marquage maximum bien connu [Bir01]. Il s'énonce comme suit :

Dans une entreprise hiérarchisée sous forme d'arbre, on souhaite organiser un gala afin de mettre en avant le potentiel amical des employés. Chaque employé a donc été noté en fonction de sa faculté à paraître amical aux yeux des clients ; mais afin que l'ambiance ne soit pas trop tendue, on s'interdit d'inviter à la fois un employé et son supérieur direct. Le but est donc de maximiser le potentiel amical global des invités.

Ce problème se résout par une accumulation vers le haut suivie d'une accumulation vers le bas ; la figure 5.2 donne les résultats en performance (secondes inverses, s^{-1}) sur un arbre de 16777215 nœuds, avec un paramètre m de 20000. L'axe des ordonnées n'a pas été gradué en accélération (qui est le rapport entre la performance parallèle et la performance séquentielle) car, en raison de la très grande taille du problème, on atteint les limites de la mémoire d'une machine seule, et la performance séquentielle est donc peu représentative. Les tests ont été effectués sur la grappe de l'Université de Tokyo, composée de processeurs Xeon à 2.4GHz.

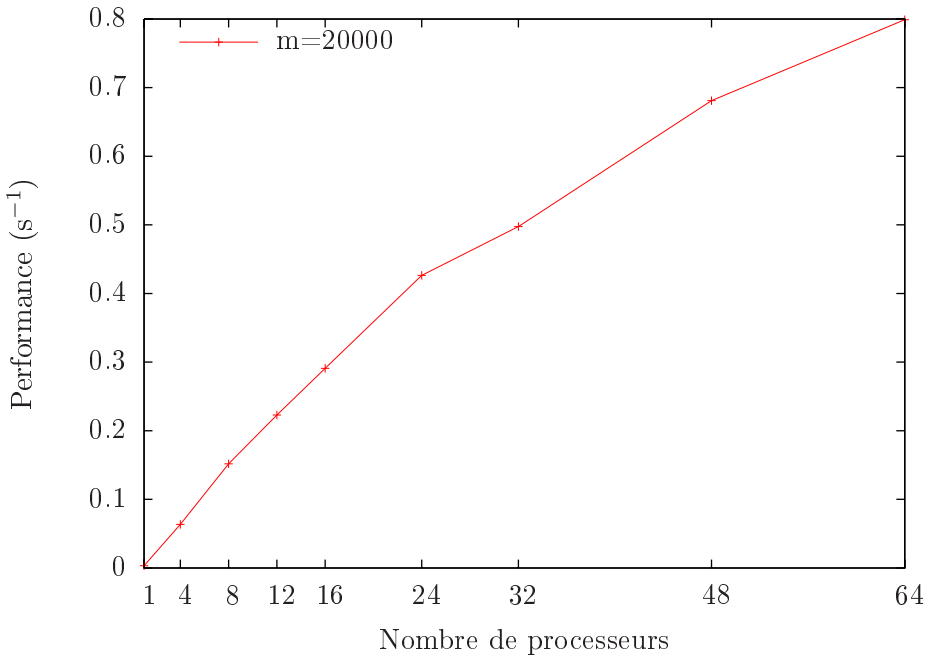


FIG. 5.2 – Résolution de PPP

Sur cet exemple, l'accélération garde une bonne progression avec un nombre de processeurs élevé. Comparativement à une implantation très fortement optimisée, en C++ et basée sur des structures de données bas niveau [MH06], la performance est bien sûr beaucoup plus faible, mais le rapport aurait été le même en comparant ces deux approches sur des applications séquentielles.

Le résultat important ici est ainsi que l'accélération suit la même progression avec notre approche qu'avec l'approche de plus bas niveau.

Chapitre 6

Méthodologie pour le développement d'algorithmes parallèles certifiés

6.1 Généralités

Un des intérêts de BSML est d'offrir une sémantique formelle, l'un des échelons nécessaires à l'obtention de programmes certifiés. Dans cette partie, nous détaillons une méthode combinant différents éléments (squelettes algorithmiques, l'assistant de preuves Coq, ...) et ayant pour but l'obtention d'algorithmes parallèles certifiés. Nous entendons ici le mot «certifié» dans le sens mathématique du terme, c'est-à-dire formellement démontré.

La certification correspond à la preuve qu'un programme machine correspond à un cahier des charges, le programme et le cahier des charges correspondant aux deux niveaux extrêmes de la chaîne certifiée. Dans le contexte le plus simple, cette chaîne peut se limiter à la compilation prouvée du programme utilisateur vers du code machine ; dans les contextes les plus critiques, comme le pilotage automatique de transports de personnes, le cahier des charges est une spécification mathématique de haut niveau et le code machine prouvé est exécuté sur un processeur dont la fiabilité est analysée statistiquement par des méthodes expérimentales.

Notre but ici n'est pas aussi extrême. Néanmoins, la programmation parallèle est plus délicate, et donc plus sujette à l'erreur humaine, que la programmation séquentielle. De plus, le coût du déploiement et du test d'applications massivement parallèles sur des grappes de calcul peut s'avérer prohibitif ; enfin, et bien que les architectures utilisées dans des contextes critiques aient généralement quelques générations de retard sur les architectures modernes, on se dirige aujourd'hui vers le tout-parallèle.

Pourtant, on fait face à un paradoxe, qui veut que performance et certification soient antagonistes : l'accumulation d'optimisations ad hoc étant trop complexe, on se limite au plus simple quand on veut s'assurer de la correction d'un programme. Le parallélisme, qui lui aussi vise la performance, ajoute lui-même un niveau de complexité certain à la programmation et augmente grandement les possibilités d'erreurs de la part du programmeur, ce qui nous convainc de l'utilité d'approches certifiées dans ce domaine. L'expérience menée ici vise à montrer que l'opposition sûreté et performances n'est pas une fatalité, et qu'il est possible avec une approche de haut niveau bien choisie et des outils adaptés d'obtenir de bonnes performances, un algorithme prouvé et le tout en relativement peu d'efforts une fois la méthode établie.

Bien sûr, cela ne va pas sans certaines restrictions raisonnables : la chaîne de preuves ici n'ira pas jusqu'au processeur. Les niveaux les plus bas de certification pour BSML s'appuient sur [Gav05], où l'auteur propose plusieurs sémantiques (naturelle, à petits pas et distribuée) à substitutions explicites, prouve leur équivalence et va jusqu'à prouver la correction d'une machine virtuelle parallèle. C'est une base solide et raisonnable pour construire notre travail de certification.

À l'autre extrémité, la spécification de l'algorithme sera fournie de manière déclarative, sous une forme définie clairement. Un algorithme séquentiel simple est proposé, et son équivalence avec l'algorithme BSML prouvée.

Comme pour tout système de preuves, il y a des limites et il reste des éléments en lesquels on devra faire confiance : ces éléments sont ici le prouveur Coq lui-même, la machine physique bien sûr, ainsi que l'extraction de programmes OCaml et le compilateur OCaml. En ce qui concerne BSML, l'utilisateur devra encore avoir confiance en son implantation et en la sous-couche de communications utilisée s'il souhaite bénéficier de l'extraction de programmes.

6.2 Squelettes algorithmiques

6.2.1 Généralités

L'apparition des squelettes algorithmiques vient de deux constatations, la première étant que programmer les détails de l'implantation des algorithmes parallèles est généralement fastidieux, et la seconde que des motifs de programmation récurrents émergent de ces implantations [Col89, Col04b]. D'où l'idée de construire une boîte à outils de combinateurs, prenant en charge les détails du parallélisme, et qui peuvent être associés entre eux pour implanter des algorithmes de façon parallèle.

Ces combinateurs, que l'on peut considérer comme des primitives de haut niveau, forment les squelettes algorithmiques. Dans les approches strictes, tout parallélisme est invisible aux yeux de l'utilisateur et géré de façon interne par les squelettes : l'effort de parallélisation est alors une réécriture de l'algorithme sous une forme utilisant les squelettes disponibles, par exemple des opérations de *folding*, *map*, *etc.* Les squelettes ayant pu être optimisés pour bénéficier au maximum des possibilités de parallélisation, une réécriture judicieuse donnera une implantation parallèle efficace dans beaucoup de cas.

Les squelettes, de plus, peuvent présenter une interface générique et portable puisque les programmes ne reposent sur aucune caractéristique matérielle : des implantations optimisées suivant les particularités de différentes architectures sont du ressort du langage de squelettes. L'implantation est donc plus simple (d'un niveau d'abstraction plus élevé), plus portable et réutilisable, plus performante en général, et, étant plus structurée, peut ouvrir la voie à des optimisations plus avancées.

Ces optimisations, cependant, peuvent parfois se révéler délicates. Le problème de la composition de squelettes, en particulier, est un sujet de recherche actif [HIT02]. Se pose aussi le problème de l'étendue de la bibliothèque de squelettes : il est souvent difficile d'implanter un algorithme quelconque sous forme de squelettes, et cela requiert une bonne connaissance de la bibliothèque de la part de l'utilisateur. Un ensemble «minimal» de squelettes qui permettraient le meilleur gain de performances possible quelle que soit l'architecture reste encore à définir.

Ces dernières années ont vu apparaître un grand nombre de bibliothèques de squelettes [CMV+06, CLPW08, FSCL06, BL05, Col04a, Alt07, GSF+07] ; mais leur place dans le développement d'applications parallèles reste à faire. Cela est du, principalement, à la perte de généralité et au besoin de se conformer à une approche inhabituelle pour le programmeur. Les squelettes se révèlent une solution idéale, par exemple, dans le cas du traitement d'un flux de données par étapes successives (*pipelining*) mais peuvent parfois se révéler très contraignants, tout en limitant les possibilités offertes au programmeur, en matière de partage des données par exemple.

Les squelettes sont aussi parfois considérés comme les héritiers de l'échec de la parallélisation automatique : en demandant à l'utilisateur de suivre une structure particulière, le compilateur sera capable de paralléliser de manière optimale le code, en fonction de l'architecture cible.

Dans les langages de squelettes purs, les squelettes en eux-mêmes ont la forme de primitives particulières, qui ne se mêlent pas de façon uniforme au reste du langage. BSML se révèle particulièrement adapté à l'implantation de squelettes [GG08], et permet de définir des squelettes sous forme de simples fonctions, tout en offrant, s'il en est besoin, un contrôle manuel du parallélisme à l'utilisateur par l'intermédiaire des vecteurs parallèles.

6.2.2 Cas d'application

Face à la grande variété de squelettes existants, et aux différentes approches auxquelles ils correspondent, nous allons devoir restreindre notre étude à une certaine classe de squelettes : les squelettes de données. Ceux-ci se spécialisent dans l'étude de structures de données particulières (arbres [MH06, Rei93], matrices [EMHT06], listes [Col93]...), par opposition aux squelettes de structure qui déterminent un schéma d'exécution particulier, et sont plutôt utilisés pour le traitement de flux (*pipeline*, *farm*) [CLPW08].

Notre méthodologie, afin d'analyser en détail un cas concret, sera développée sur les squelettes de listes. La distribution d'une liste ne posant pas de problème – contrairement à celle d'un arbre, par exemple – nos exemples se baseront sur une liste distribuée entre les processeurs, à la façon de la fonction select_list du 2.4.1.

La généralisation de la méthode à d'autres structures de données plus complexes demande un travail supplémentaire, mais traiter des squelettes de listes nous permettra d'analyser les difficultés conceptuelles principales.

6.3 Environnement de preuve de programmes BSML

6.3.1 L'assistant de preuves Coq

Coq est un système d'aide à la preuve basé sur le Calcul des Constructions Inductives (CCI), qui est un λ-calcul typé dans lequel des types sont des termes comme les autres. Il fournit des mécanismes pour écrire des définitions et pour faire des preuves formelles.

Le système Coq est un programme interactif permettant à l'utilisateur de construire des preuves. Cela permet de :

1. Définir des termes, c'est-à-dire les objets (mathématiques) du discours, comme les entiers, les listes *etc.*, et des propriétés sur ces objets puisque celles-ci sont des types et que types

et termes ne sont pas différenciés ;

2. Construire des démonstrations de ces propriétés (qui sont également des termes) à l'aide d'un langage de *tactiques* permettant de fabriquer interactivement un terme d'un type donné ;
3. Vérifier les preuves ainsi construites, en demandant au programme de les typer.

Le mécanisme de construction des preuves n'a pas besoin d'être certifié puisque le noyau du système Coq, c'est-à-dire la fonction de typage vérifie les preuves. Le noyau fait en revanche l'objet de certifications et de vérifications [Bar99]. Ajoutons que la représentation concrète des preuves a d'autres avantages : par exemple, elle rend possible l'écriture d'un vérificateur de type indépendant de Coq, et elle autorise également l'emploi de toutes sortes d'outils pour engendrer des preuves que Coq pourra vérifier *a posteriori* [BC04, Fil03, FM04, FM07].

Il n'y a pas d'inférence de type en Coq, celle-ci n'étant pas décidable : cela correspondrait en effet à un prouveur automatique «universel», ce qui est impossible. Il est cependant laborieux d'annoter toutes les définitions par des informations de type quand celles-ci semblent pouvoir être aisément déduites du contexte. Coq propose donc un mécanisme simple d'inférence, qui, bien sûr, peut échouer. Il fournit également un mécanisme d'arguments implicites et il s'efforce aussi de les inférer.

Nous nous intéressons donc, dans cette section, au système d'aide à la preuve Coq. Nous n'avons pas pour but de remplacer la documentation accompagnant le système. Le lecteur pourra donc consulter en complément : le tutoriel et le livre [BC04] pour une introduction plus graduelle ainsi que le manuel de référence[14] pour une description formelle et exhaustive de Coq.

6.3.2 Assistant de preuves

L'objectif premier de Coq est d'être un assistant de preuves, et donc de permettre de formaliser le raisonnement mathématique. Prenons un énoncé basique : $\forall A, A \Rightarrow A$. Cet énoncé, qui se lit «pour toute proposition A (c'est-à-dire objet A de type Prop), A implique A», peut être directement transcrit dans le système Coq comme suit :

```
Lemma easy: forall A: Prop, A → A.
Proof.
```

Cette portion de script commence par le nom et l'énoncé de notre lemme, où Prop est l'ensemble des propositions logiques. Le système entre alors dans son mode *interactif* qui permet de bâtir cette preuve étape par étape. Pour cela, le système utilise des directives appelées *tactiques*. Celles-ci reflètent la structure de la preuve en déduction naturelle. Nous avons donc :

```
1 subgoal

  ============================
   forall A:Prop, A→ A

easy < intros.
```

[14] Le manuel de référence ainsi que le tutoriel sont librement téléchargeables à `http://coq.inria.fr`.

En entrant la tactique intros (introduire les hypothèses), le système affiche l'état courant du ou des buts à prouver. Les hypothèses passent au-dessus de la double-barre, le but à prouver reste en-dessous :

```
1 subgoal

  A : Prop
  H : A
  ============================
   A
```

```
easy < apply H.
Proof completed.
```

Il s'agit alors de trouver une preuve de A sous l'hypothèse A. La tactique **apply** H permet d'appliquer cette hypothèse, ce qui termine dans ce cas la preuve. Il ne reste plus qu'à faire vérifier l'ensemble de la preuve au système Coq (vérification des types de tous les sous-termes) à l'aide du mot-clef Qed, qui enregistrera le lemme easy dans le système.

Coq met à la disposition de l'utilisateur une grande variété de tactiques. Ici par exemple, pour un énoncé aussi simple, les tactiques de recherche automatique telles que auto ou trivial auraient été suffisantes.

La représentation interne d'une preuve est un λ-terme. Le système logique sous-jacent à Coq est le CCI (Calcul des Constructions Inductives) qui est un λ-calcul typé où les énoncés exprimables en Coq sont des types du CCI. En application de l'isomorphisme de Curry-Howard [GLT89], vérifier si t est bien une preuve valide de l'énoncé T consiste à vérifier que le type T est bien un type légal du λ-terme t. C'est ce qui est effectué par le système lors de l'utilisation de Qed. On peut par exemple obtenir la représentation interne du lemme easy à l'aide de la directive Print :

```
easy = fun (A:Prop) (H:A) ⇒ H
     : forall A:Prop, A → A
```

où **fun** x:X => e est la notation Coq pour l'abstraction typée. Celle-ci est l'effet de la tactique intro. Quant à **apply**, son effet est l'utilisation de la variable de contexte H. De façon générale, une tactique contribue à construire petit à petit le λ-terme CCI de la preuve. Notons que si l'on connaît à l'avance le λ-terme complet, on peut le donner directement sous la forme d'une Definition, ce qui donne :

```
Definition easy: forall A:Prop, A → A :=
  fun (A:Prop) (H:A) ⇒ H
```

La quantification universelle $\forall x : X, T$ est appelé *produit* ou *type dépendant*, puisque, le plus souvent, le corps T du produit dépend de la variable x de la tête du produit. Cette quantification n'étant pas restreinte, le CCI est donc une logique d'ordre supérieur. Notons que la syntaxe A→ B est un sucre syntaxique de Coq pour forall _:A, B.

6.3.3 Langage de programmation

On peut également, et cela nous sera d'un intérêt particulier, aborder Coq depuis l'autre versant de l'isomorphisme de Curry-Howard : considérer Coq non pas comme un système logique mais plutôt commme un λ-calcul, c'est-à-dire un langage de programmation purement fonctionnel. Par exemple, notre lemme easy devient une fonction d'identité sur les propositions logiques.

En tant que langage de programmation, il permet la définition de types de données. Les types en Coq sont définis par induction, par exemple le type des booléens s'obtient par :

```
Inductive bool:Set := true:bool | false:bool
```

Cette déclaration crée un nouveau type, nommé bool, ainsi que deux nouveaux constructeurs. L'annotation Set permet de désigner quel va être le type du type bool. Set est le type des objets calculatoires (à la différence de Prop qui est le type des objets logiques). De même, les entiers de Peano sont définis de la manière suivante :

```
Inductive nat:Set := O: nat | S: nat → nat.
```

où O est le zéro et S le successeur d'un nat. Un dernier exemple usuel est celui des listes paramétriques :

```
Inductive list (A:Set): Set := nil: list A | cons: A → list A → list A
```

où la liste dépend d'un paramètre A. Ce paramètre peut être fourni en argument lors de l'utilisation de la structure de données, mais Coq est généralement capable de l'inférer d'après les éléments placés dans la liste. Une liste d'entiers a ainsi pour type list nat.

L'entrelacement des parties logiques (des propositions) et des parties purement calculatoires est possible dans le système Coq. Cela permet notamment d'enrichir un terme calculatoire avec des pré- et post-conditions, ou encore d'utiliser une récursion bien fondée en justifiant la décroissance d'une mesure à chaque appel récursif.

Outre le besoin d'exprimer la spécification d'une fonction sous la forme de pré- et de post-conditions, les pré-conditions logiques vont également apporter une solution au problème de la définition des fonctions partielles. Considérons, par exemple, une fonction de prédécesseur sur les entiers naturels pred:nat→ nat, qui n'est pas définie quand son argument est nul. On peut alors exprimer, par une pré-condition logique, le fait que cet argument doit être non nul si l'on veut utiliser cette fonction. Le type de pred devient alors forall n:nat, n<>0 → nat. On notera que le type de l'argument n'est plus un type flèche (un produit anonyme) mais un produit nommé par n afin de s'y référer dans l'assertion logique. La fonction pred n'est alors plus définie en dehors du domaine de validité de l'assertion logique. La contre-partie est que l'on doit toujours fournir une preuve logique de non-nullité à chaque appel à pred.

Combiner pré- et post-conditions spécificie une fonction dans un style à la Hoare [Hoa69]. Ainsi, une fonction de type A→ B de pré-condition P et de post-condition Q correspond à la preuve constructive : $\forall x : A,\ (P\ x) \rightarrow \exists y : B,\ (Q\ x\ y)$. Ceci est exprimable en Coq à l'aide d'un type inductif sig :

```
Inductif sig (A:Set)(P:A→ Prop): Set :=
    exist: forall x:A, (P x)→ (sig A P).
```

qui s'écrit également avec un sucre syntaxique {x:A | (P x)}. Une spécification complète en Coq de la fonction de prédécesseur entière s'écrit alors :

```
Definition pred: forall n:nat, n<>0 → {q:nat | (S q)=n}.
```

Il ne reste alors qu'à donner au système un λ-terme répondant à cette spécification (un λ-terme ayant ce type). On peut le construire à l'aide de tactiques de filtrage par cas comme destruct, qui énumèrent les constructeurs possibles pour ce type.

```
Definition pred: forall n:nat, n<>0 → {q:nat | (S q)=n}.
  intros.
  destruct n.
    absurd (0<>0). auto. assumption. (* cas n=0 *)
    exists n. reflexivity. (* cas n<>0 *)
Defined.
```

Le premier but se résout facilement par l'absurde et le second est une trivialité. Nous pouvons maintenant afficher le λ-terme CCI complet de notre fonction :

```
pred =
fun (n : nat) (H : n <> 0) ⇒
match n as n0 return (n0 <> 0 → {q : nat | S q = n0}) with
| 0 ⇒
    fun H0 : 0 <> 0 ⇒
    False_rec {q : nat | S q = 0}
      (let H1 :=
          False_ind (0 <> 0)
            (let H1 := H0 in
             (let H2 := fun _ : 0 = 0 → False ⇒ H0 (refl_equal 0) in
              fun H3 : 0 <> 0 ⇒ H2 H3) H1) in
       (let H2 := fun _ : 0 = 0 → False ⇒ H0 (refl_equal 0) in
        fun H3 : 0 <> 0 ⇒ H2 H3) H1)
| S n0 ⇒
    fun _ : S n0 <> 0 ⇒
    exist (fun q : nat ⇒ S q = S n0) n0 (refl_equal (S n0))
end H
     : forall n : nat, n <> 0 → {q : nat | S q = n}
```

Un dernier exemple montrant la définition d'un lemme un peu plus étendu nous sera utile dans la suite. La fonction nth est définie dans la bibliothèque Coq standard et a le comportement suivant ; nth n lst d renvoie l'élément d'indice n dans la liste lst si cet indice est valable, d sinon. Un lemme utile permettant de prouver les égalités de listes s'énonce comme suit :

```
Lemma nth_equal (l1 l2: list A):
  forall d,
  length l1 = length l2 →
    (forall n, n < length l1 → nth n l1 d = nth n l2 d) →
      l1 = l2.
```

La preuve se fait par double induction sur l1 et l2. On omettra dans de nombreux cas le script de preuve complet des lemmes, propositions et théorèmes comme on vient de le faire ici, pour se

contenter d'exprimer les idées principales guidant la preuve : les scripts sont en général longs et peu lisibles et surtout, étant construits de manière interactive, ils présentent peu d'intérêt si on ne peut pas suivre leur éxécution pas à pas dans Coq.

Notons pour finir que Coq permet l'extraction de programmes vers un langage externe [Let04] : on peut ainsi, à partir d'un programme en Coq sur lequel des propriétés mathématiques ont été prouvées, extraire un programme OCaml – ou BSML dans le cas qui nous concerne. Si l'on reprend un exemple ci-dessus :

```
Coq < Extraction pred.

(** val pred : nat → nat **)

let pred = function
  | O → assert false (* cas absurde *)
  | S n0 → n0
```

6.3.4 Définition de BSML en Coq

Les preuves se basent sur une définition préalable des primitives BSML en Coq. Cette définition repose sur des paramètres, nous permettant de nous abstraire de l'implantation problement dite (une instance simple de ces paramètres, utilisant des listes, nous suffit à prouver leur cohérence). De précédentes versions de la description de BSML en Coq [Gav03] se basaient sur des axiomes, et étaient donc moins générales. La spécification décrite ici diffère en un certain nombre d'autres points, comme la définition de l'ensemble des identifiants de processeurs : ces modifications se sont imposées à l'usage, et facilitent l'utilisation de ce formalisme dans les preuves.

Afin de ne pas surcharger le manuscrit, seul un aperçu du formalisme utilisé est donné : l'ensemble des définitions et des preuves décrits ici sont disponibles au lecteur à l'adresse `http://gesbert.fr/coq`.

La première chose à définir est le paramètre **bsp_p**. Comme celui-ci doit être strictement positif, on le définit comme successeur d'un paramètre entier naturel max_pid.

```
Parameter max_pid : nat.
Definition bsp_p := S max_pid.
Lemma bsp_p_gt_0 : 0 < bsp_p.
Proof. unfold bsp_p. auto with arith. Qed.
```

On définit ensuite l'ensemble des processeurs, dont dépendront les primitives. C'est un sous-ensemble des entiers naturels :

```
Definition procs := { pid: nat | pid < bsp_p }.

Definition nat_of_proc (p: procs) := let (n,_) := p in n.
Definition proc_of_nat (n: nat) (P: n < bsp_p) : procs :=
  exist (fun x ⇒ x < bsp_p) n P.
```

Les deux convertisseurs nat_of_proc et proc_of_nat nous seront utiles par la suite.

Il nous sera utile dans nos preuves de pouvoir utiliser la propriété

```
Lemma procs_eq : forall (i j: procs),
  nat_of_proc i = nat_of_proc j → i = j.
```

c'est-à-dire que deux processeurs ayant le même pid sont égaux. Nous adoptons pour cela l'axiome d'indiscernabilité des preuves (*proof-irrelevance* : ∀ A:Prop, ∀ p q:A, p=q), non contradictoire et largement utilisé dans les développements Coq [BC04]. La preuve de ce lemme ne pose pas de problème particulier en utilisant cet axiome.

Le type abstrait des vecteurs parallèles, et l'accesseur att permettant d'en récupérer une composante sont donnés comme suit :

```
Parameter Vector : Set → Set.
Parameter att : forall A: Set, Vector A → forall i: procs, A.
```

att nous permettra de spécifier les propriétés locales de nos opérateurs parallèles. Afin de pouvoir prouver l'égalité de vecteurs, on suppose également donnée la caractérisation suivante, qui spécifie simplement que deux vecteurs dont toutes les composantes sont égales sont égaux :

```
Parameter vec_equal : forall (A: Set) (v v': Vector A),
    (forall (i: procs), att A v i = att A v' i) → v = v'.
```

Tous les éléments sont posés pour la définition de nos primitives ; on ne donne aucune spécification ici sur la forme des vecteurs ou des calculs, mais on précise les propriétés que doivent assurer chacune des primitives :

```
Variable A : Set.

Parameter mkpar_p : forall f: procs → A,
  { X: Vector A | forall i: procs, att X i = f i }.

Parameter apply_p : forall (B: Set) (vf: Vector (A → B)) (vx: Vector A),
  { X: Vector B | forall i: procs, att X i = (att vf i) (att vx i) }.

Parameter put_p : forall (vf: Vector (procs → A)),
  { X: Vector (procs → A) | forall i: procs, att X i = fun j ⇒ att vf j i }.

Parameter proj_p : forall (v: Vector A),
  { X: procs → A | forall i: procs, X i = att v i }.
```

En prenant l'exemple de **mkpar**, le paramètre mkpar_p devra fournir, pour toute fonction des processeurs vers un ensemble A, un vecteur X ainsi qu'une preuve de la propriété att X i = f i pour tout processeur i. Cette propriété correspond à notre définition sémantique de mkpar : on pourra ainsi l'utiliser dans les preuves, tout en restant abstrait par rapport à son implantation.

De ces paramètres, on extrait trivialement les primitives ainsi que leurs propriétés de réductions. Ils seront ainsi sous une forme directement utilisable. Dans le cas de **mkpar** (les autres sont similaires) :

```
Definition mkpar (f: procs → A) := let (x,p) := mkpar_p f in x.

Lemma mkpar_def (f: procs → A) : forall i: procs, att (mkpar f) i = f i.
```

```
Proof.
  intros. unfold mkpar. destruct (mkpar_p f). apply e.
Qed.
```

Enfin, on peut simplifier la réécriture de programmes utilisant nos primitives à l'aide de

```
Hint Rewrite mkpar_def apply_def put_def proj_def : bsml_def.
```

Des versions de nos primitives utilisables directement avec le type `nat` plutôt que `procs` sont également fournies. Il est en particulier souvent plus aisé de fournir une fonction sur `nat` quand on utilise **mkpar**.

Exemples

Il est très simple, à l'aide de ces outils, de prouver la dualité de **proj** et **mkpar**.

```
Lemma proj_mkpar:
  forall (i:procs) (f:procs → A), proj (mkpar f) i = f i.
Proof.
  intros.
  autorewrite with bsml_def.
  reflexivity.
Qed.
```

L'égalité est ici donnée «pour tout i», où i est un processeur valide : on a en effet pas l'égalité des fonctions à proprement parler[15], l'une mettant en particulier en œuvre le parallélisme. Ce qui nous intéresse est de toute manière de pouvoir prouver l'égalité des résultats.

En revanche, l'égalité des vecteurs nous est fournie par le paramètre `vec_equal` :

```
Lemma mkpar_proj: forall (v: Vector A), mkpar (proj v) = v.
Proof.
  intros.
  apply vec_equal. intro.
  autorewrite with bsml_def.
  reflexivity.
Qed.
```

Existence d'une instance des paramètres

Bien sûr, la validité de ces définitions est dépendante de l'existence de paramètres satisfaisant ces conditions. On propose par exemple des listes de taille fixe :

```
Definition Vector A := { l: list A | length l = bsp_p }.

Definition att : Vector A → forall i: procs, A.
  intros v i.
```

[15] à moins de supposer l' «extensionalité» fonctionnelle, considérée comme cohérente dans Coq mais non sans conséquences, en particulier sur l'extraction de programmes [Hof95, Our05]. Nous nous en dispenserons ici.

```
  destruct v as (l,Hl).
  refine (nth (nat_of_proc i) l _).
  destruct l.
    absurd_hyp Hl. unfold bsp_p. auto with arith. assumption.
  exact a.
Defined.
```

Notre définition de att correspond simplement à la fonction nth sur les listes ; la preuve sépare la liste l de la propriété sur sa longueur Hl. La fin dé-construit la liste l en s'appuyant sur le fait qu'elle a au moins un élément (**bsp_p** > 0), afin de fournir une valeur par défaut à nth.

Nous prouvons également la propriété vec_equal en nous basant sur le lemme nth_equal et sur proof_irrelevance.

Il est ensuite relativement simple de définir les primitives parallèles, par exemple, pour **mkpar** :

```
Definition mkpar (f: procs → A) : Vector A.
  intro f.
  refine (vec_of_list (map f procs_list) _).
  rewrite map_length.
  apply procs_list_length.
Defined.
```

La preuve ici concerne la longueur de la liste renvoyée, afin d'en faire un vecteur. La preuve de la propriété requise par mkpar_p, que nous omettons ici, est plus longue mais ne présente aucune difficulté particulière.

Cette instanciation de nos paramètres ne sera pas utilisée dans la suite : elle assure juste que notre théorie est cohérente. Les programmes extraits de coq utiliseront les primitives abstraites **mkpar**, **apply**, **proj** et **put** qui correspondront à notre implantation concrète (et parallèle) du langage.

6.4 Méthodologie

À partir des éléments préliminaires que nous avons décrits, il devient possible d'implanter des squelettes *certifiés* en BSML : ceux-ci fournissent une base idéale pour la preuve d'applications parallèles. En effet, depuis le programme utilisateur, les squelettes sont spécifiés en Coq en tant que combinateurs mathématiques ne laissant pas apparaître le parallélisme sous-jacent à leur implantation. La preuve de ce programme en Coq est alors de la même complexité que la preuve d'un programme séquentiel habituel utilisant ces opérations.

En utilisant des méthodes d'algorithmique constructive telles que celles développées dans [HIT97], cette approche peut être rendue encore plus haut niveau et il nous devient possible de masquer toute la complexité des preuves techniques à l'utilisateur. En effet, il s'avère qu'il est souvent délicat d'utiliser les squelettes de structure de façon directe depuis un programme, ceux-ci nécessitant parfois le calcul de fonctions auxiliaires afin de gérer le découpage des données : c'est le cas de l'exemple des m-ponts donné en 5.6.

L'algorithmique constructive permet justement de dériver ces fonctions depuis une spécification initiale du problème, liée à la structure de données concernée, et de définition mathématique

beaucoup plus simple. Bien qu'on y perde en généralité, l'usage en est beaucoup plus simple, et l'efficacité de l'implantation garantie. Le développement se déroule comme suit :

1. L'utilisateur écrit une spécification de son algorithme, basée sur des opérations dont l'expression mathématique est simple.
2. Un algorithme utilisant les squelettes correspondants est dérivé de cette spécification.
3. Ces mêmes squelettes sont implantés en BSML, ce qui nous fournit une implantation de l'algorithme d'origine.

Le mérite de cette approche est qu'elle être prouvée de haut en bas, de manière semi-automatique (ici en Coq) : la transformation initiale depuis la spécification peut être prouvée, et l'implantation des squelettes également – toute propriété que l'utilisateur aura prouvée sur la spécification d'origine est alors garantie dans l'implantation finale, parallèle et optimisée.

Dans le chapitre suivant, nous nous attacherons à montrer une application de cette méthodologie dans le cadre de squelettes sur les listes, en fournissant la théorie sous-jacente et de nombreux exemples.

Chapitre 7

Cas d'application : le squelette BH

Le squelette BH a été conçu pour former la passerelle idéale entre opérations sur les listes, basées sur les homomorphismes, et calcul BSP sur une liste distribuée. Après avoir établi un premier exemple, qui sera développé au fil du chapitre, nous allons définir les propriétés de ce squelette et montrer comment la méthodologie que nous avons énoncée peut être appliquée de façon semi-automatique. Comme pour le chapitre précédent, les développements formels peuvent être téléchargés à l'adresse `http://gesbert.fr/coq/`.

7.1 Exemple : construction de tours

On développera dans la suite des exemples basés sur le problème de construction de tours, une variante du problème des lignes de vues [BHS+94]. Dans ce problème, on suppose données les hauteurs de points le long d'une ligne droite sous la forme d'une liste.

$$[(x_1, h_1), \ldots, (x_i, h_i), \ldots, (x_n, h_n)]$$

Les x_i, qu'on suppose ordonnés, correspondent à l'abscisse des points sur la ligne, les h_i à l'altitude correspondante. Deux points particuliers, $L = (x_L, h_L)$ et $R = (x_R, h_R)$ sont donnés à gauche et à droite de cette liste (c'est-à-dire que $x_L < x_1$ et $x_n < x_R$). Le problème consiste à déterminer, parmi les points de la liste, ceux où la construction d'une tour de hauteur h donnée permettrait d'apercevoir à la fois L et R.

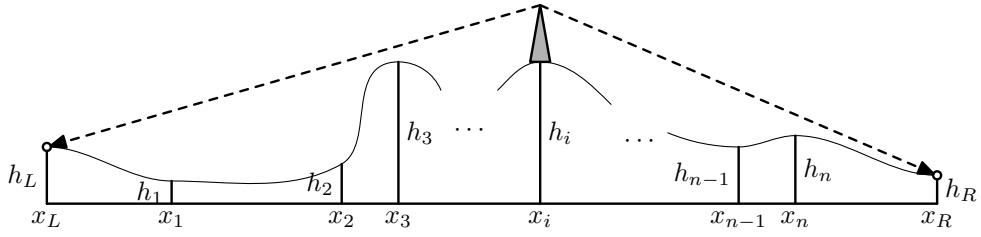


FIG. 7.1 – Problème de construction de tour

Un point (x_i, h_i) est visible depuis L et R si aucun des points intermédiaires ne fait obstacle :

$$\forall k < i, \frac{h_k - h_L}{x_k - x_L} < \frac{h_i + h - h_L}{x_i - x_L} \qquad \forall k > i, \frac{h_k - h_R}{x_R - x_k} < \frac{h_i + h - h_R}{x_R - x_i}$$

Une solution non parallèle simple consiste à tester pour chaque point si ces conditions sont validées. Cela nous donne une spécification claire et relativement simple. Ce n'est pas le cas du programme parallèle BSML suivant, permettant de résoudre le problème à partir des coordonnées sous forme de liste distribuée lv :

```
val procs_left : int list par
let procs_left = ≪ sequence 0 ($this$ − 1) ≫

val procs_right : int list par
let procs_right = ≪ sequence ($this$ + 1) (bsp_p − 1) ≫

val bsml_towers :
  float ∗ float → (float ∗ float) list par → float ∗ float → float → bool list par

let bsml_towers (xl,hl) lv (xr,hr) htower =
  let tan_l (x,h) = (h −. hl) /. (x −. xl) in
  let tan_r (x,h) = (h −. hr) /. (xr −. x) in
  let maxtans_l =
    ≪ fold_left (fun (m,acc) p → let m = max (tan_l p) m in m, m :: acc)
                (neg_infinity,[]) $lv$ ≫ in
  let maxtans_r =
    ≪ fold_right (fun p (m,acc) → let m = max (tan_r p) m in m, m :: acc)
                 $lv$ (neg_infinity,[]) ≫ in
  let maxtan_l,maxtanlist_l = ≪ fst $maxtans_l$ ≫ , ≪ rev (snd $maxtans_l$) ≫ in
  let maxtan_r,maxtanlist_r = ≪ fst $maxtans_r$ ≫ , ≪ snd $maxtans_r$ ≫
  in
  let comms = put ≪ fun dest → if dest < $this$ then $maxtan_r$
                               else if dest > $this$ then $maxtan_l$
                               else 0. ≫
  in
  let maxl = ≪ fold_left (fun acc j → max acc ($comms$ j)) neg_infinity $procs_left$ ≫ in
  let maxr = ≪ fold_left (fun acc j → max acc ($comms$ j)) neg_infinity $procs_right$ ≫
  in
    ≪ map3 (fun tan_l (x,h) tan_r → (h +. htower −. hl) /. (x −. xl) > max tan_l $maxl$
                                 && (h +. htower −. hr) /. (xr −. x) > max tan_r $maxr$)
           $maxtanlist_l$ $lv$ $maxtanlist_r$ ≫
```

Le programme procède en calculant localement la tangente maximale par rapport à L et R pour chaque sous-liste, en échangeant ces résultats puis en rassemblant, sur chaque processeur, le maximum à gauche et à droite afin de terminer le calcul localement. Il renvoie une liste distribuée de booléens ayant la même structure que la liste initiale.

Il est difficile à première vue de s'assurer de la correction de ce programme, ne serait-ce

que d'un point de vue algorithmique : c'est une raison supplémentaire qui justifie une approche certifiée.

7.2 Homomorphismes

Les homomorphismes jouent un rôle particulier par rapport au parallélisme, spécialement en ce qui concerne les squelettes : ils offrent en effet une structure générale qui permet de les paralléliser naturellement, et regroupent nombre d'opérations usuelles. Si on a un type τ muni d'un opérateur de composition $+\!\!+$, on peut en effet calculer le résultat de l'homomorphisme sur une composition à partir de ses résultats sur les éléments initiaux : en matière de parallélisme, on peut donc obtenir le résultat global à partir de résultats préliminaires locaux.

Si l'on choisit les listes comme structure de données, exemple que nous allons développer dans la suite, les homomorphismes se définissent de la façon suivante, où $+\!\!+$ est la concaténation de listes :

Définition 7.2.1 (*Homomorphisme de listes*)

La fonction h est un homomorphisme de listes s'il existe une fonction f et un opérateur associatif $\odot$ tels que :

$$h\,[a] = f\,a$$
$$h\,(x +\!\!+\, y) = h(x) \odot h(y)$$

f et $\odot$ sont suffisants pour définir l'homomorphisme. On notera $(\!|f, \odot|\!)$ pour désigner un tel homomorphisme. Une conséquence de cette définition est que h appliqué à la liste vide est une unité pour l'opérateur $\odot$ – nous noterons $1_\odot$. En effet, d'après la deuxième équation,

$$h(x) = h([] +\!\!+\, x) = h([]) \odot h(x) = h(x +\!\!+\, []) = h(x) \odot h([])$$

map et reduce sont probablement les homomorphismes de listes les plus couramment utilisés en programmation :

- map f est purement locale et se définit comme $(\!|f, +\!\!+|\!)$. map $f\,[x_1, \ldots, x_n] = [f\,x_1, \ldots, f\,x_n]$
- reduce $(\odot)$ pour sa part peut se définir comme $(\!|id, \odot|\!)$ (où id est l'identité) – c'est un équivalent de fold qui n'impose pas l'ordre de réduction, et donc destiné aux opérateurs associatifs. fold $(\odot)\,[x_1, \ldots, x_n] = x_1 \odot \ldots \odot x_n$

Il est ainsi possible de définir tout homomorphisme $(\!|f, \odot|\!)$ sous la forme d'une composition (reduce $\odot$) $\circ$ (map f), c'est-à-dire de séparer le calcul de f et de $\odot$ en deux phases entièrement distinctes.

Les homomorphismes, en tant que tels, restent trop simples pour représenter une variété réellement intéressante de fonctions, mais il est assez facile de les étendre pour y remédier en conservant leurs propriétés majeures. Les *quasi-homomorphismes* [Col93] construisent des fonctions renvoyant des tuples qui peuvent être utilisées comme homomorphismes, et ajoutent une opération simple de projection pour extraire le résultat final.

Exemple 7.2.2 (*mss*)

Le problème du segment de somme maximum (mss)*, consiste à trouver les éléments consé-*

cutifs dont la somme est maximale dans une liste de nombres relatifs. Prenons la liste $[4;5;-20;6;5;-20]$ *: la somme de segment maximale est* 11*, correspondant au segment* $[6;5]$*. On ne peut pas déduire ce résultat des résultats sur* $[4;5;-20;6]$ *et* $[5;-20]$ *(respectivement* 9 *pour le segment* $[4;5]$ *et* 5 *pour le segment* $[5]$*) : ceux-ci ne donnent pas suffisamment d'informations sur la disposition des éléments dans la sous-liste. Il reste cependant possible de définir une fonction hmss qui renvoie un quadruplet* (sl, ms, sr, ts) *donnant la somme maximale de segment adjacent à gauche, la somme maximale de segment, la somme maximale de segment adjacent à droite et le total du segment : on a ainsi suffisamment d'informations sur la sous-liste pour pouvoir composer les résultats. Cette fonction est un homomorphisme : on peut définir* $\odot$ *tel que*

$$\begin{aligned}&(sl, ms, sr, ts) \odot (sl', ms', sr', ts')\\ &= (\max(sl, ts + sl'), \max(ms, sr + sl', ms'), \max(sr + ts', sr'), ts + ts')\end{aligned}$$

Une fois le calcul terminé, on a le résultat dans la deuxième composante du quadruplet, et on l'extrait par π*. De ce quasi-homomorphisme se déduit directement un algorithme parallèle efficace.*

Cette technique est générale. En effet, toute fonction f sur une liste peut s'exprimer par :

$$\begin{aligned}g(x) &= (f(x), x)\\ (a, x) \odot (b, y) &= (f(x +\!\!+ y), x +\!\!+ y)\\ \pi(a, x) &= a\end{aligned}$$

$\pi \circ (\!| g, \odot |\!)$ est un quasi-homomorphisme qui calcule f. Ce dernier n'a cependant que peu d'intérêt vis-à-vis du parallélisme, les résultats intermédiaires $f\,x$ n'étant pas réutilisés pour le calcul du résultat sur une concaténation : la totalité du calcul est refaite à chaque étape. La définition d'un quasi-homomorphisme intéressant pour le parallélisme nécessite donc un choix judicieux de résultats intermédiaires et de projection, afin de maximiser l'utilisation de calculs locaux.

Parmi les autres propriétés utiles des homomorphismes, le «théorème de l'homomorphisme» [Gib96, MMM+07] joue un rôle clef dans leur extraction à partir de problèmes.

Théorème 7.2.3 (*Théorème de l'homomorphisme*)

Si une fonction f *sur les listes peut être calculée par composition à droite et par composition à gauche, c'est-à-dire qu'il existe des opérateurs* $\oplus_{\mathrm{l}}, \oplus_{\mathrm{r}}$ *tels que*

$$\begin{aligned}f([a] +\!\!+ y) &= a \oplus_{\mathrm{l}} f\,y \qquad\qquad \textit{et}\\ f(x +\!\!+ [b]) &= f\,x \oplus_{\mathrm{r}} b\end{aligned}$$

alors f *est un homomorphisme* $(\!| f', \odot |\!)$*, où* $f'(x) = f([x])$ *et, si* g *est un inverse droit de* f *(c'est-à-dire tel que* $f \circ g = id$*),* $\odot$ *est défini par* $a' \odot b' = f(g(a') +\!\!+ g(b'))$*.*

On montre l'utilité de ce théorème à travers l'exemple d'un sous-problème du problème de construction de tour :

Exemple 7.2.4

À partir de données similaires à celles du problème déjà énoncé, on cherche cette fois à savoir si du haut d'une tour de hauteur h en R on pourra apercevoir le point L.

Le problème peut s'exprimer sous la forme suivante :

$$\begin{aligned} visible\, xs = maxTanL\, xs &< \frac{h_R + h - h_L}{x_R - x_L} \\ maxTanL\, [] &= -\infty \\ maxTanL\, ([(x,h)] +\!\!+\, xs) &= \max(\frac{h - h_L}{x - x_L}, maxTanL\, xs) \end{aligned}$$

Une autre définition de $maxTanL$ est possible, ce qui nous permettra d'utiliser le théorème de l'homomorphisme :

$$maxTanL\, (xs +\!\!+ [(x,h)]) = \max(maxTanL\, xs, \frac{h - h_L}{x - x_L})$$

En remarquant qu'on peut obtenir simplement un inverse droit pour $maxTanL$ avec :

$$g\, a = [(x_L + 1, h_L + a)]$$

on peut appliquer le théorème et en déduire $maxTanL = (\!|f, \odot|\!)$ avec

$$\begin{aligned} a \odot b &= maxTanL\, [g\, a +\!\!+\, g\, b] \\ &= maxTanL\, [(x_L + 1, h_L + a), (x_L + 1, h_L + b)] \\ &= \max(a, b) \end{aligned}$$

et $f = maxTanL\, [(x,h)] = \frac{h - h_L}{x - x_L}$.

7.3 BH : homomorphisme BSP

7.3.1 Définition

Les homomorphismes ont l'intérêt de décrire directement des algorithmes parallèles par l'intermédiaire de leur écriture sous forme de map/reduce. Afin de décrire plus précisément des algorithmes BSP en vue de les prouver, on définit un homomorphisme particulier, BH.

Définition 7.3.1 (*BH*)

h est un «homomorphisme BSP», ou BH, si on peut l'exprimer sous la forme :

$$\begin{aligned} h\, [a]\, l\, r &= [k\, a\, l\, r] \\ h\, (x +\!\!+\, y)\, l\, r &= h\, x\, l\, (g_r\, y \oplus_{\mathrm{r}} r) +\!\!+\, h\, y\, (l \oplus_{\mathrm{l}} g_l\, x)\, r \end{aligned}$$

pour certains $k, g_l, g_r, \oplus_{\mathrm{l}}, \oplus_{\mathrm{r}}$.

BH nous offre pont entre la réalité des communications et des super-étapes, et une abstraction de plus haut niveau. On note $BH(k, (\!|g_l, \oplus_{\mathrm{l}}|\!), (\!|g_r, \oplus_{\mathrm{r}}|\!))$.

Les paramètres l et r figurent les informations qui sont nécessaires pour permettre le calcul final (local) de h sur un élément ; ces paramètres sont calculés respectivement par les fonctions de propagation à droite $(g_l, \oplus_{\mathrm{l}})$ et à gauche $(g_r, \oplus_{\mathrm{r}})$: ils nous permettent de figurer les communications qui auront lieu lors du calcul. L'idée est qu'on puisse se ramener, sur chaque processeur, au calcul de BH sur un segment x pour lequel les valeurs l et r sont connues – celles-ci étant obtenues par g_l et g_r, les communications, puis $\oplus_{\mathrm{l}}$ et $\oplus_{\mathrm{r}}$.

BH renvoie une liste de même taille que la liste d'entrée, et plus précisément conserve la structure du découpage initial. Cela a son importance lorsqu'on travaille sur une liste distribuée, car ainsi l'équilibre des données sera conservé.

Le calcul de BH sur la machine parallèle peut s'effectuer de la façon suivante, en supposant une liste $l = x_0 +\!\!+ \cdots +\!\!+ x_{p-1}$ distribuée (préférablement de façon équilibrée) sur p processeurs (soit en BSML le vecteur $\langle x_i \rangle_i$) :

1. calcul local de g_l et g_r sur chacun des x_i
2. communication vers le processeur i de tous les $g_l\, x_j$ pour $j < i$ et de tous les $g_r\, x_k$ pour $i < k$, ce qui correspond à une fin de super-étape
3. combinaison des données $g_l\, x_0, \ldots, g_l\, x_{i-1}, g_r\, x_{i+1}, \ldots, g_r\, x_{p-1}$ reçues sur chaque processeur i à l'aide de $\oplus_{\mathrm{l}}$ et $\oplus_{\mathrm{r}}$ afin d'obtenir les l et r correspondant au segment local.
4. calcul local de h sur chaque x_i à l'aide des l et r.

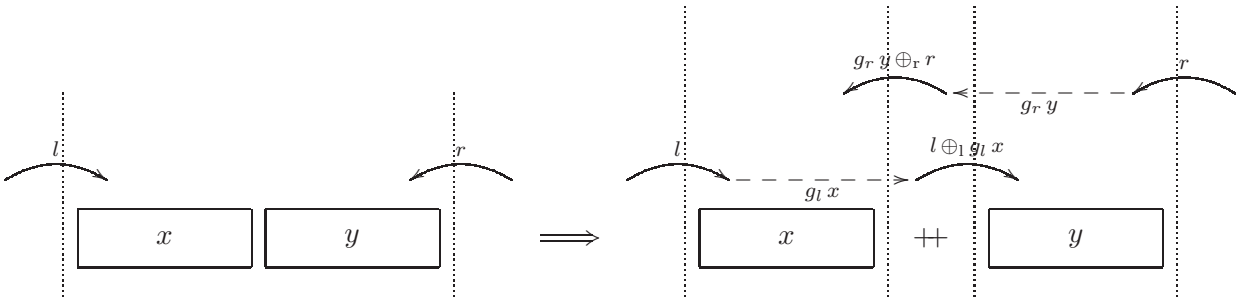


FIG. 7.2 – Propagation de l'information pour BH

L'efficacité de l'implantation dépendra de deux critères, la durée des calculs locaux et la quantité de données à échanger. Attardons-nous sur ce dernier, qui sera plus limitatif pour la parallélisation à grande échelle. On peut estimer le coût des communications en fonction de la taille relative de $g_l(x)$ et x (et $g_r(x)$ et x). Chaque processeur i émet i fois $g_l(x_i)$ et $p - i - 1$ fois $g_r(x_i)$ et réceptionne une quantité de données équivalente. La taille maximale des données émises ou reçues est ainsi atteinte par le processeur 0 ou par le processeur $p-1$, et on a un coût en communications de

$$g \times \max\left(\sum_{i<p-1} \mathit{taille}(g_l(x_i)), \sum_{i>0} \mathit{taille}(g_r(x_i))\right)$$

Si on suppose la liste distribuée de façon équitable, et en notant $taille_f(l) = taille(f(x))$ si x est de taille l , on peut simplifier la formule en :

$$g \times (p-1) \times \max\left(taille_{g_l}(\frac{taille(lst)}{p}), taille_{g_r}(\frac{taille(lst)}{p})\right)$$

Le facteur de réduction de la taille de g_l et g_r est donc déterminant pour les performances de BH : pour le problème des plus proches valeurs inférieures, $taille_{g_l}$ et $taille_{g_r}$ sont des logarithmes, pour le problème de construction des tours ce sont des fonctions constantes.

7.3.2 Spécification Coq

Plusieurs définitions de BH nous seront utiles pour nos différentes applications ; on a en particulier une définition plus mathématique utilisée dans les preuves, et une définition utilisant les primitives BSML et correspondant à l'implantation. La preuve d'équivalence de ces différentes versions, dont nous reparlerons, est une des pierres angulaires de cette méthode.

Donnons ici une définition simple basée sur la fonction nth de la bibliothèque standard de Coq.

k,gl,opl,gr,opr sont des paramètres définis préalablement, et correspondant respectivement à $k, g_l, (\oplus_{\mathrm{l}}), g_r, (\oplus_{\mathrm{r}})$ dans la définition précédente.

```
Definition bh_nth (za: A) (l:L) (lst: list A) (r:R) (n:nat) :=
  k (opl l (gl (firstn n lst))) (nth n lst za) (opr (gr (skipn (S n) lst)) r).

Definition bh_comp (l:L) (lst: list A) (r:R) : list B :=
  match lst with
    | nil ⇒ nil
    | a::lst ⇒ map (bh_nth a l (a::lst) r) (seq 0 (length (a::lst)))
  end.
```

bh_nth calcule l'image du n^e élément de la liste lst par BH à partir de l'image de g_l sur ses prédecesseurs (firstn n lst) et de l'image de g_r sur ses successeurs (skipn (S n) lst). bh_comp est ensuite défini comme un *map* de cette fonction sur la liste des indices de la liste d'origine.

À titre d'exemple, et en anticipant légèrement sur la suite, voici une implantation de BH en BSML : d'abord une version séquentielle, puis une version parallèle prenant en paramètre une liste distribuée.

```
let bh_seq (k: 'l → 'a → 'r → 'o)
    (gl: 'a list → 'l) (opl: 'l → 'l → 'l)
    (gr: 'a list → 'r) (opr: 'r → 'r → 'r) =
  fun (l: 'l) (lst: 'a list) (r: 'r) →
    let rights = fold_left (fun (r::rs) x → (opr (gr [x]) r)::r::rs) [r] (rev lst)
    in
    let rec aux acc l lst rs = match (lst,rs) with
      | (x::tl,r::rs) → aux (k l x r :: acc) (opl l (gl [x])) tl rs
      | ([],[]) → acc
```

```
    in
      rev (aux [] l lst (tl rights))

type ('l,'r) comms = Ncx | Lcx of 'l | Rcx of 'r

let bh_par
    (k: 'l → 'a → 'r → 'o)
    (gl: 'a list → 'l) (opl: 'l → 'l → 'l)
    (gr: 'a list → 'r) (opr: 'r → 'r → 'r) =
  fun (l: 'l) (lst: 'a list par) (r: 'r) →
  let accl = ≪ gl $lst$ ≫
  and accr = ≪ gr $lst$ ≫
  in
    let comms = put ≪ fun receiver →
                        if $this$ < receiver then Lcx $accl$
                        else if $this$ > receiver then Rcx $accr$
                        else Ncx ≫
  in
  (* calcul des valeurs l et r locales à l'aide des valeurs reçues *)
    let lv = ≪ fold_left
      (fun acc i → opl acc (match $comms$ i with Lcx x → x)) l $procs_left$ ≫
    and rv = ≪ fold_right
      (fun i acc → opr (match $comms$ i with Rcx x → x) acc) $procs_right$ r ≫
  in
  (* calcul sur la sous-liste locale à l'aide de la version séquentielle de BH *)
    ≪ bh_seq k gl opl gr opr $lv$ $lst$ $rv$ ≫
```

bh_seq calcule BH séquentiellement sur une liste en deux étapes : d'abord la liste des r est calculée par un fold_left, puis une double récursivité sur cette liste et celle d'origine permet de calculer BH en chaque point.

Lors de l'exécution de bh_par, chaque processeur calcule d'abord g_l et g_r sur sa sous-liste locale, puis envoie les résultats du premier vers tous les processeurs à sa droite et les résultats du second vers tous les processeurs à sa gauche. Les informations reçues sont ensuite combinées à l'aide de $\oplus_l$ et $\oplus_r$ afin d'obtenir les valeurs de l et r correspondant à la sous-liste locale, sur laquelle le calcul est fait par appel à bh_seq.

7.4 Dérivation de BH

7.4.1 Spécification

Afin de permettre la transformation semi-automatique du programme initial, celui-ci doit se conformer à un certain format de spécification. Cette spécification ne sera pas purement mathématique, mais elle est construite à partir d'éléments suffisamment simples pour être facilement vérifiables (au contraire du programme parallèle final). En particulier, les propriétés que l'utilisateur souhaite certifier sur son programme pourront être prouvées en Coq par rapport à cette

spécification, et ainsi validées sur l'implantation parallèle.

La spécification, par souci de simplicité, se fait en Coq. On la suppose fonctionnelle, et faite à partir d'opérations parallèles primitives sur les listes de la forme suivante :

- opérateurs collectifs (homomorphismes, *scan*, *mapAround*)
- fonctions récursives gauches ou droites
- fonction de communications (*permute*, *shiftL*, *shiftR*, ...)

Les opérateurs collectifs permettent de décrire les opérations simples sur les listes. *scan*se définit informellement de la façon suivante (on suppose $\odot$ associatif) :

$$scan\,(\odot)\,[x_1, x_2, \ldots, x_n] = [x_1, x_1 \odot x_2, \ldots, x_1 \odot x_2 \odot \ldots \odot x_n]$$

mapAround est une version étendue de map, beaucoup plus générique : elle permet de calculer l'image de chaque élément d'une liste en fonction de ses prédecesseurs, l'élément lui-même et ses successeurs.

$$mapAround\,f\,[x_1, \ldots, x_i, \ldots, x_n] = [\ldots, f\,([x_1, \ldots, x_{i-1}], x_i, [x_{i+1}, \ldots, x_n]), \ldots]$$

Il est utile de pouvoir utiliser aussi des fonctions récursives dans la spécification, dans le cas où des homomorphismes n'apparaissent pas de façon évidente. Comme on l'a vu, on pourra utiliser le théorème de l'homomorphisme si ces fonctions sont définies à la fois vers la gauche et vers la droite.

Les fonctions de communication capturent un autre comportement, et permettent de redistribuer les éléments de la liste. *permute*réordonne les éléments suivant un index indiquant leur position finale souhaitée dans la liste :

$$permute[(2, a), (1, b), (5, c), (3, d), (4, e)] = [b, a, d, e, c]$$

et pour finir, *shiftL*et *shiftR*permettent de décaler les éléments de la liste :

$$shiftL\,a\,[x_1, \ldots, x_n] = [x_2, \ldots, x_n, a]$$
$$shiftR\,a\,[x_1, \ldots, x_n] = [a, x_1, \ldots, x_{n-1}]$$

Exemple 7.4.1 (*spécification du problème des tours*)

La fonction mapAround est particulièrement adaptée à la spécification de ce problème, qui se calcule pour chaque point en fonction des éléments à sa gauche et à sa droite :

$$tower\,(x_L, h_L)\,xs\,(x_R, h_R) = mapAround\,visibleLR\,xs$$

Où vsibleLR est défini de la façon suivante :

$$visibleLR\,(ls,(x_i,h_i),rs) = visibleL\,ls\,x_i \wedge visibleR\,rs\,x_i$$
$$visibleL\,ls\,x_i = maxTanL\,ls < \frac{h_i + h - h_L}{x - x_L}$$
$$visibleR\,rs\,x_i = maxTanR\,rs < \frac{h_i + h - h_R}{x_R - x}$$

Nous avons déjà défini maxTanL, maxTanR se définit de façon similaire par :

$$maxTanR\,[\,] = -\infty$$
$$maxTanR\,([x,h] +\!\!+\, xs) = \max\left(\frac{h - h_R}{x_R - x}, maxTanR\,xs\right)$$
$$maxTanR\,(xs +\!\!+ [x,h]) = \max\left(maxTanR\,xs, \frac{h - h_R}{x_R - x}\right)$$

Exemple 7.4.2 (*spécification de mps*)

Le problème de la somme maximale de préfixe (mps) [] est similaire au problème du segment de somme maximum, à la différence près qu'on force à ce que le segment choisi soit un préfixe de la liste. Par exemple $mps[2,-1,3,-2]$ *est 4, qui correspond à la somme du préfixe* $[2,-1,3]$.

Il y a plusieurs moyens de définir mps : par composition de fonctions, en calculant d'abord la somme de tous les préfixes, puis en prenant le maximum.

$$mps = maximum \circ psums$$
$$maximum = (\!|id, \max|\!)$$
$$psums = scan\,(+)$$

Ou bien, par exemple, à l'aide de la fonction récursive bidirectionnelle suivante (sum étant définie simplement, de la même façon) :

$$mps\,[\,] = 0$$
$$mps\,([a] +\!\!+\, x) = \max(0, a + mps\,x)$$
$$mps\,(x +\!\!+ [a]) = \max(mps\,x, (sum\,x) + a)$$

Exemple 7.4.3 (*spécification du rassemblement de tableau*)

*Le problème du rassemblement de tableau (*array packing*) est courant dans la littérature sur les algorithmes parallèles, et est utilisé dans de nombreux algorithmes, tel le calcul d'enveloppe convexe. Il consiste à rassembler, dans une liste de valeurs booléennes* true *et* false*, tous les* true *en tête du tableau. On le spécifie par composition :*

$$arrPack = permute \circ compIndex$$

une première étape, compIndex, se charge de calculer l'index désiré pour chaque élément,

et une seconde replace les éléments par permute. L'index peut être calculé simplement à l'aide de mapAround : si l'élément est true*, son index correspond au nombre de* true *à sa gauche, et dans le cas inverse il correspond au nombre de* true *total plus le nombre de* false *à sa gauche (soit la taille de la liste moins le nombre de* false *à sa droite). On utilise les homomorphismes ctrue et cfalse qui comptent le nombre d'occurrence de* true *et* false *dans une liste pour*

$$compIndex = mapAround\, indexing\, xs$$

$$indexing\,(ls, x, rs) = \begin{array}{ll} ctrue\, ls + 1, x & si\ x = \mathsf{true} \\ ctrue\, ls + ctrue\, rs + cfalse\, ls + 1, x & sinon \end{array}$$

7.4.2 Théorèmes de transformation

On se propose, à partir de la spécification du problème, de dériver un programme BSML certifié. Cela peut être assez simple pour des fonctions de communication telles que *permute*, qui correspondent directement à une fonction dans le langage ; cependant les définitions récursives et les autres opérateurs peuvent être imbriqués et ne pas fournir directement un programme parallèle de façon évidente. Nous allons transformer ces définitions en appels à BH, dont nous fournirons ensuite une implantation certifiée.

Théorème 7.4.4 (*Parallélisation de mapAround par BH*)

Soit $h = mapAround\, f$. On suppose que f se décompose de la façon suivante :

$$f\,(ls, x, rs) = k\,(g_l\, ls, x, g_r\, rs)$$

où g_l et g_r sont des quasi-homomorphismes $\pi_l \circ (\!|k_l, \oplus_{\mathrm{l}}|\!)$ et $\pi_r \circ (\!|k_r, \oplus_{\mathrm{r}}|\!)$. Alors

$$h\, xs = BH(k', (\!|g_l, \oplus_{\mathrm{l}}|\!), (\!|g_r, \oplus_{\mathrm{r}}|\!))\, xs\, 1_{\oplus_{\mathrm{l}}}\, 1_{\oplus_{\mathrm{r}}}$$

avec k' combinant k et les opérations de projection : $k'\,(l, x, r) = k\,(\pi_l\, l, x, \pi_r\, r)$.

Démonstration : Par induction sur la liste d'entrée (xs) ; la preuve en Coq est détaillée en 7.4.3. ■

Corollaire 7.4.5 (*Parallélisation de scan par BH*)

Le calcul de $scan\,(\odot)$ est un BH.

Démonstration : Il suffit de remarquer que *scan*peut s'exprimer en termes de *mapAround* : $scan\,(\odot) = mapAround\, f$ avec

$$f\,(ls, x, rs) = k\,((\!|id, \odot|\!)\, ls, x, [])$$

où $k\,(a, b, c) = a \odot b$. ■

Corollaire 7.4.6 (*Parallélisation d'homomorphisme par BH*)

~~Tout~~ homomorphisme $(\!|f, \odot|\!)$ peut être exprimé par un BH.

Démonstration : On définit g sous la forme suivante :

$$g\,(ls, x, rs) = k\,([], f\,x, (\!|f, \odot|\!)\,rs)$$

avec $k\,(a, b, c) = b \odot c$. On a alors $head \circ mapAround\,g = (\!|f, \odot|\!)$, où *head* renvoie la tête de la liste. ■

Exemple 7.4.7 (*Dérivation de BH pour le problème des tours*)

D'après la spécification précédente, on peut définir le problème à partir de :

$$visibleLR\,(ls, x, rs) = k\,(maxTanL\,ls, x, maxTanR\,rs)$$
$$k\,(m_l, (x, h_x), m_r) = \left(m_l < \frac{h_x + h - h_L}{x - x_L}\right) \wedge \left(m_r < \frac{h_x + h - h_R}{x_R - x}\right)$$

maxTanL et maxTanR étant comme nous avons vu des homomorphismes $(\!|k_l, \max|\!)$ *et* $(\!|k_r, \max|\!)$*, avec* $k_l(x, h_x) = \frac{h_x - h_L}{x - x_L}$ *et* $k_r(x, h_x) = \frac{h_x - h_R}{x_R - x}$*. On peut donc appliquer le théorème 7.4.4 :*

$$tower\,(x_L, h_L)\,xs\,(x_R, h_R) = BH\,(k, (\!|k_l, \max|\!), (\!|k_r, \max|\!))\,xs\,(-\infty)\,(-\infty)$$

Exemple 7.4.8 (*Dérivation de BH pour mps*)

mps est la composition d'un homomorphisme et d'un scan : mps = maximum ∘ psums. La dérivation de BH pour chacun d'entre eux est directe :

$$\begin{aligned} psums\,xs &= scan\,(+)\,xs \\ &= BH\,(k, (\!|id, +|\!), [])\,xs\,0\,0 && \text{avec } k\,(a, b, c) = a + b \\ maximum\,xs &= (\!|id, \max|\!)\,xs \\ &= (head \circ BH\,(k', [], (\!|id, \max|\!)))\,xs\,(-\infty)\,(-\infty) && \text{avec } k'\,(a, b, c) = \max(b, c) \end{aligned}$$

Exemple 7.4.9 (*Dérivation de BH pour* Array Packing)

On a la spécification arrPack = permute ∘ compIndex. On suppose une implantation certifiée de permute donnée en tant que telle, il nous reste donc à dériver un BH pour compIndex, de la même manière que nous l'avons fait pour le problème de construction de tours. Le fait que deux fonctions différentes soient appliquées à ls dans indexing (ctrue et cfalse) pourrait paraître un obstacle, mais il est aisément contourné par la construction d'un couple [HITT97] : on définit $g_l\,xs = (ctrue\,xs, cfalse\,xs)$*, ce qui correspond à l'homomorphisme* $(\!|k_l, \oplus_l|\!)$

$$(t_1, f_1) \oplus_l (t_2, f_2) = (t_1 + t_2, f_1 + f_2)$$
$$k_l x = \begin{array}{l} (1, 0)\ si\ x\ est\ \mathsf{true} \\ (0, 1)\ si\ x\ est\ \mathsf{false} \end{array}$$

La suite est semblable au problème des tours.

7.4.3 Théorie Coq

On peut définir *mapAround* en Coq de la façon suivante :

```
Definition mapAround (f: list A * A * list A → B) (lst: list A) : list B :=
  map f (zip3 (inits lst) lst (tails lst)).
```

Où zip3 combine trois listes en une liste de triplets, inits est la liste des préfixes d'une liste et tails la liste des suffixes : de la sorte, chaque élément de la liste «zippée» est de la forme (préfixe,x,suffixe) et on peut calculer *mapAround* par l'intermédiaire d'un simple map. Les fonctions auxiliaires sont définies de la sorte :

```
(* zip3 fusionne trois listes de même taille en une liste de triplets *)
Fixpoint zip3 (L C R: Type) (l: list L) (c: list C) (r: list R) {struct l} : list (L*C*R) :=
  match l,c,r with
    |a::l',b::c',c::r' ⇒ (a,b,c)::(zip3 l' c' r')
    |_,_,_⇒ nil
  end.

(* tails calcule la liste des suffixes *)
Fixpoint tails (l:list A) :=
  match l with
    | nil ⇒ nil
    | a::r ⇒ r::(tails r)
  end.

(* inits calcule la liste des préfixes *)
Definition inits (l:list A) :=
  rev (map (rev(A:=A)) (tails (rev(A:=A) l))).
```

De nombreux lemmes sont prouvés sur ces fonctions afin de permettre de les manipuler dans les preuves. Les plus importants sont ceux qui nous permettent d'obtenir des information sur le n^e élément d'une liste :

```
Lemma tails_nth (l:list A) (n:nat) :
  nth n (tails l) nil = skipn (S n) l.
Proof.
  intros. generalize l.
  induction n.
  intro l'. destruct l'. simpl. reflexivity. simpl. reflexivity.
  intro l'. destruct l'. simpl. reflexivity.
  simpl nth. change (skipn (S (S n)) (a::l')) with (skipn (S n) l').
  apply IHn.
Qed.

Lemma inits_nth (l:list A) (n:nat) :
  nth n (inits l) l = firstn n l.

Lemma zip3_nth :
  forall L C R (l: list L) (c: list C) (r: list R) n d dl dc dr,
```

```
  n < length (zip3 l c r) →
    nth n (zip3 l c r) d = ((nth n l dl), (nth n c dc), (nth n r dr)).
```

Nous avons ici omis les preuves de inits_nth (par une longue série de réécriture et quelques propriétés intermédiaires, elle repose sur celle de tails_nth) et de zip3_nth (par simple induction sur n).

Le théorème 7.4.4 s'énonce pour sa part de la façon suivante :

```
Variable (L R : Set).
Variable (opl: L → L → L).
Variable (opr: R → R → R).

Theorem mapAround_to_bh :
  forall (f :((list A)*A*(list A)) → B) (k :L → A → R → B) (gl: list A → L) (gr: list A → R) xs ,
    (forall ls x rs,
      f (ls,x,rs) = k (gl ls) x (gr rs))
    → is_homomorphism A L gl opl
    → is_homomorphism A R gr opr
    → mapAround f xs = bh_comp k gl opl gr opr (gl nil) xs (gr nil).
```

(gl nil) et (gr nil) correspondent à $1_{\oplus_l}$ et $1_{\oplus_r}$; La preuve fait une cinquantaine de lignes (hors lemmes utilitaires, nombreux) et est basée sur l'application du lemme nth_equal, d'où l'utilité des lemmes inits_nth, tails_nth et zip3_nth.

7.5 Implantation certifiée de BH

Nous avons donné une première définition bh_comp pour BH. On peut montrer que cette définition correspond bien à BH :

```
  Theorem bh_hom (l:L) (x y: list A) (r:R):
    (is_homomorphism A L gl opl) →
    (is_homomorphism A R gr opr) →
    bh_comp l (x++y) r = bh_comp l x (opr (gr y) r) ++ bh_comp (opl l (gl x)) y r.
```

La preuve est longue en raison du grand nombre de cas à traiter, mais sans difficulté conceptuelle particulière : elle repose sur nth_equal et des suites de réécritures.

La définition du calcul parallèle de BH est plus longue à définir. Elle reprend le déroulement de la version OCaml présentée p.135.

```
Definition bh_bsml_comp (l:L) (vl: Vector (list A)) (r:R) :=
  let comms := bsml_comp_bh_comm l vl r in
    apply
      (apply (apply (replicate bh_comp) (loc_lefts l vl r comms)) vl)
      (loc_rights l vl r comms).
```

Ce programme calcule d'abord les valeurs préliminaires et les communique à l'aide de bsml_comp_bh_comm, puis rassemble les résultats à l'aide de $\oplus_l$ et $\oplus_r$ par l'intermédiaire des fonctions

loc_lefts et loc_rights. Il se termine par un appel local à bh_comp sur le segment concerné. Les communications sont définies de la façon suivante :

```
Inductive t_comm : Set :=
  | lcx : L → t_comm
  | rcx : R → t_comm
  | ncx : t_comm.

Definition comm_fun sender l r receiver :=
  match nat_compare sender receiver with
    | Lt ⇒ lcx l
    | Eq ⇒ ncx
    | Gt ⇒ rcx r
  end.

Definition bsml_comp_bh_comm (l:L) (lst: Vector (list A)) (r:R) :=
  put_nat (apply (apply (mkpar_nat comm_fun) (apply (replicate gl) lst))
                 (apply (replicate gr) lst)).
```

On définit, comme dans la version OCaml, un type pour les communications suivant leur direction, et une fonction de communication qui servira de paramètre à put_nat. bsml_comp_bh_comm commence par faire une application locale de gl et gr puis effectue les communications par un appel à **put**. Ces résultats sont ensuite réduits localement par :

```
Definition loc_lefts (l:L) (lst: Vector (list A)) (r:R) (comms: Vector (procs → t_comm)) :=
  apply
    (mkpar (fun i com ⇒ fold_range_left_procs (rem_lcx_comm com l) l (nat_of_proc i)))
    comms.

Definition loc_rights (l:L) (lst: Vector (list A)) (r:R) (comms: Vector (procs → t_comm)) :=
  apply
    (mkpar (fun i com ⇒ fold_range_right_procs (rem_rcx_comm com r) r (nat_of_proc i)))
    comms.
```

ce qui est équivalent à nos utilisation de fold_left sur $procs_left$ et fold_right sur $procs_right$ dans la version OCaml (on omet les définitions des fonctions utilitaires fold_range_ et rem_, peu intéressantes).

La difficulté est ensuite de prouver que bh_bsml_comp calcule bien le même résultat que bh_comp, c'est-à-dire que :

```
Theorem bh_bsml_bh lst:
  partition_merge (proj' (bh_bsml_comp (gl nil) (scatter lst) (gr nil))) =
    bh_comp k gl opl gr opr (gl nil) lst (gr nil).
```

Où **proj**' est similaire à **proj** mais renvoie une fonction de nat plutôt que de procs. Ce théorème repose sur une théorie permettant le partitionnement et le rassemblement de listes, et qui nous permet de déterminer la façon dont la liste d'origine serra distribuée sur la machine parallèle :

```
Definition partition (l: list A) (i:nat):=
  let len := length l in
```

```
  list_part l (cut_border max_pid len i) (cut_border max_pid len (S i)).

Definition scatter (B: Set) (l: list B) := mkpar_nat (partition l).

Definition partition_merge (l: nat → list A) :=
  fold_procs _ (fun x i ⇒ x++(l i)) nil.
```

partition l i renvoie la sous-liste de l destinée au processeur i ; scatter utilise cette fonction pour distribuer une liste sur la machine parallèle, et partition_merge rassemble les éléments d'une partition pour reconstituer la liste d'origine. scatter et partition_merge∘**proj**' correspondent à select_list et gather_list que nous avons donnés comme exemples lors de la description de BSML.

Deux résultats préliminaires majeurs sont nécessaires à la preuve du théorème bh_bsml_bh : ils assurent que les résultats des communications et de loc_lefts et loc_rights correspondent bien aux résultats attendus, c'est-à-dire g_l et g_r appliqués aux sous-listes des éléments à gauche et à droite (respectivement) de la sous-liste locale. Ils s'énoncent en Coq comme suit :

```
Lemma loc_lefts_def (lst: list A) (p:procs) :
  proj (loc_lefts (gl nil) (scatter lst) (gr nil)
                  (bsml_comp_bh_comm (gl nil) (scatter lst) (gr nil))) p
  = gl (firstn (cut_border max_pid (length lst) (nat_of_proc p)) lst).

Lemma loc_rights_def (lst: list A) (p:procs) :
  proj (loc_rights (gl nil) (scatter lst) (gr nil)
                   (bsml_comp_bh_comm (gl nil) (scatter lst) (gr nil))) p
  = gr (skipn (cut_border max_pid (length lst) (S (nat_of_proc p))) lst).
```

La preuve de ces lemmes est très technique, car le résultat découle de plusieurs étapes de découpage et combinaison de sous-listes. On obtient le résultat en décomposant la liste des processeurs sous la forme (l1++p::nil)++l2 puis par induction, sur l1 dans le cas de loc_lefts_def et sur l2 dans le cas de loc_rights_def.

On est ensuite en mesure de prouver le résultat final sur une sous-liste locale :

```
Theorem bh_bsml_bh_pre lst:
  forall n, n<bsp_p →
    proj' (bh_bsml_comp (gl nil) lst (gr nil)) n =
    partition (bh_comp k gl opl gr opr (gl nil) lst (gr nil)) n.
```

par une série de réécritures et l'utilisation de nos hypothèses sur les homomorphismes. Le théorème final en découle directement par application d'un lemme de réunification de partition.

On a donc la preuve de la validité de notre squelette BH, et ainsi la preuve de validité de l'implantation parallèle de tout algorithme spécifié en Coq selon le format que nous avons donné. Outre les exemples déjà présentés, nous avons pu obtenir de la sorte une implantation certifiée d'un algorithme de multiplication d'une matrice creuse et d'un vecteur, et d'une solution au problème des valeurs inférieures les plus proches (*all nearest smaller values*).

Il serait intéressant d'étendre ces travaux à d'autres structures de données et à d'autres squelettes. En particulier, l'approche s'adapterait tout à fait aux squelettes sur les arbres découpés par m-ponts tels que mentionnés en 5.6

Chapitre 8

Conclusion

Trouver les paradigmes les plus adaptés à la programmation parallèle, et donc vraisemblablement à la programmation de demain, est un des défis de la recherche actuelle. Si le parallélisme bénéficie d'un long héritage qu'il ne faut pas abandonner dans le domaine du calcul scientifique, le problème commence à se poser en termes différents du fait de la diversité de ses nouvelles applications.

Une grande variété d'approches, anciennes et récentes, existent : de la programmation concurrente à la programmation de très haut niveau par pure combinaison de squelettes ou par parallélisation automatique. Les débats font rage entre les tenants des différentes approches, et ressemblent par certains aspects à ceux qui ont opposé autrefois les tenants du `goto` des langages séquentiels primitifs à ceux d'approches plus modernes mais ne permettant pas une utilisation optimale des ressources de calcul [Gor04]. BSML occupe une position unique dans ce paysage, en se plaçant comme langage de haut niveau tout en conservant le contrôle précis du parallélisme : il impose une structure suffisante au parallélisme pour garantir la sûreté d'exécution, mais ne crée pas d'abstraction implicite entre la machine réelle et l'utilisateur, s'opposant par là aux langages à génération dynamique de processus légers.

Le travail développé ici vise à démontrer l'intérêt de l'équilibre que nous avons choisi pour BSML. En particulier, celui-ci couvre une échelle d'utilisations, entre haut et bas niveau, qu'aucun autre langage parallèle n'a atteinte jusqu'alors. Traits impératifs et extensions syntaxiques en font un langage généraliste, complet et largement utilisable ; nous montrons qu'ils sont compatibles avec notre modèle et en fournissons une implantation. Les définitions formelles des sémantiques et du système de types en font un langage sûr (dans le sens de Milner, c'est-à-dire sans erreur d'exécution) bien qu'il soit parallèle. Le langage reste bas niveau et efficace, muni d'un système de coûts précis et simple, détaillé dans [Gav05]. Enfin, on montre en deuxième partie qu'il est possible de faire de la programmation de haut niveau par squelettes en BSML et de se baser sur le langage pour établir une parallélisation semi-automatique par squelettes ; la programmation parallèle certifiée à ce niveau n'est fournie par aucun autre langage.

Le choix original derrière le modèle de BSML lui permet donc de s'adapter à une grande étendue d'applications, couvrant tous les besoins habituels du programmeur : c'est une conclusion majeure de ce travail. En effet, il s'agit là à notre avis d'un critère particulièrement important dans la recherche du domaine.

Enfin, nous espérons que les travaux présentés ici auront leur place dans la construction

des futurs modèles et des futurs langages qui permettront la généralisation du parallélisme. La plus grande partie de ces travaux pourraient, à notre avis, être adaptés à d'autres contextes : le populaire langage Java, par exemple, a un système d'exceptions très proche de celui d'OCaml ; le système de types avec effets permet de ne distinguer code local et global que là où la distinction est nécessaire, et pourrait ainsi bénéficier à d'autres approches utilisant un parallélisme à deux niveaux.

Quels qu'ils soient, ces modèles devront fournir la sûreté d'exécution – notion que nous étendons au parallélisme en assurant l'absence d'interblocages et d'indéterminisme – et préserver les traits les plus utiles des langages modernes.

Nous avons fourni des bases qui permettront d'atteindre ce résultat.

Bibliographie

[ABC+06] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research : A View from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.

[Alt07] M. Alt. *Using Algorithmic Skeletons for Efficient Grid Computing with Predictable Performance*. PhD thesis, Universität Münster, 2007.

[Bal92] H. E. Bal. Fault-tolerant parallel programming in argus. *Concurrency : Practice and Experience*, 4(1) :37–55, 1992.

[Bam00] M. Bamha. *Parallélisme et équilibrage de charges dans le traitement de la jointure et de la multi-jointure sur des architectures SN*. PhD thesis, LIFO, Université d'Orléans, 2000.

[Bar99] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université de Paris VII, 1999.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[BH99] M. Bamha and G. Hains. Frequency-Adaptive Join for Shared Nothing Machines. *Journal of Parallel and Distributed Computing Practices*, 2(3) :333–345, 1999.

[BHM00] N. Benton, J. Hughes, and E. Moggi. Monads and effects. In *International Summer School On Applied Semantics (APPSEM'2000)*, pages 42–122. Springer-Verlag, 2000.

[BHS+94] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1) :4–14, 1994.

[Bir01] R. S. Bird. Maximum marking problems. *Journal of Functional Programming*, 11(4) :411–424, 2001.

[Bis04] R. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI*. Oxford University Press, 2004.

[BJvOR03] O. Bonorden, B. Juurlink, I. von Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2) :187–207, 2003.

[BL05] J. Berthold and R. Loogen. Skeletons for recursively unfolding process topologies. In *International Conference on Parallel Computing (PARCO'05)*, pages 835–842, 2005.

[BLH99] O. Ballereau, F. Loulergue, and G. Hains. High-level BSP Programming : BSML and BSλ. In P Trinder and G. Michaelson, editors, *Scottish Functional Programming Workshop (SFP'99)*, pages 43–52, Édimbourg, august 1999. Heriot-Watt University.

[BLKC05] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello. Coordinated Checkpoint versus Message log for Fault Tolerant MPI. *Journal of High Performance Computing and Networking (IJHPCN)*, 2005.

[CDT05] A. Chan, F. Dehne, and R. Taylor. Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines. *Journal of High Performance Computing Applications*, 2005.

[CJvdP07] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP : Portable Shared Memory Parallel Programming.* Scientific and Engineering Computation Series. The MIT Press, 2007.

[CLPW08] R. Di Cosmo, Z. Li, S. Pelagatti, and P. Weis. Skeletal Parallel Programming with OcamlP3L 2.0. *Parallel Processing Letters*, 18(1), 2008.

[CMV+06] F. Clément, V. Martin, A. Vodicka, R. Di Cosmo, and P. Weis. Domain Decomposition and Skeleton Programming with OCamlP3l. *Parallel Computing*, 32 :539–550, 2006.

[Col89] M. Cole. *Algorithmic Skeletons : Structured Management of Parallel Computation.* MIT Press, 1989. Disponible à l'adresse `http://homepages.inf.ed.ac.uk/mic/Pubs`.

[Col93] M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problem. In *International Conference on Parallel Computing (PARCO'93)*, Elsevier Series in Advances in Parallel Computing, 1993.

[Col04a] M. Cole. Bringing Skeletons out of the Closet : A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3) :389–406, 2004.

[Col04b] M. Cole. Why skeletal parallel programming matters. In M.Danelutto, M. Vanneschi, and F. Laforenza, editors, *Euro-Par*, volume 3149 of *LNCS*, page 37. Springer-Verlag, 2004.

[CS03] Y. Chen and W. Sanders. Top-Down Design of Bulk-Synchronous Parallel Programs. *Parallel Processing Letters*, 13(3) :389–400, 2003.

[Dab03] F. Dabrowski. Pattern Matching and Exceptions Handling for Bulk Synchronous Parallel ML. Master's thesis, Université Paris Val de Marne, 2003.

[DDK86] J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language : Mini-ML. In *Logic In Comp. Science*, pages 13–27. ACM Press, 1986.

[DdR07] N. Pouillard D. de Rauglaudre. The OCaml Pre-Processor-Pretty-Printer, Camlp4, 2007. Page web : `http://brion.inria.fr/gallium/index.php/Camlp4`.

[DFRC96] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3) :379–400, 1996.

[DLG03] F. Dabrowski, F. Loulergue, and F. Gava. Pattern Matching of Parallel Values in Bulk Synchronous Parallel ML. In W. Dosch and R. Y. Lee, editors, *International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'03)*, pages 301–308. ACIS, 2003.

[DM69] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Ninth Annual ACM Symposium on Principles of Programming Languages (POPL'82)*, pages 207–212, New York, USA, 1969. Association for Computing Machinery, ACM Press.

[EMHT06] K. Emoto, K. Matsuzaki, Z. Hu, and M. Takeichi. Surrounding theorem : Developing parallel programs for matrix-convolutions. In *Euro-Par*, volume 4128/2006 of *Lecture Notes in Computer Science*, pages 605–614. Springer, 2006.

[Fil03] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4), 2003.

[FM04] J.-C. Filliâtre and C. Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, pages 15–29. Springer-Verlag, 2004. `http://why.lri.fr/caduceus/`.

[FM07] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, LNCS. Springer-Verlag, 2007.

[FRR+07] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore : A heterogeneous parallel language. In *Declarative Aspects of Multicore Programming (DAMP'07)*, 2007.

[FRRS08] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in manticore. *SIGPLAN Notices*, 43(9) :119–130, 2008.

[FSCL06] J. Falcou, J. Serot, T. Chateau, and J. T. Lapreste. QUAFF : Efficient C++ Design for Parallel Skeletons. *Parallel Computing*, 32(7-8) :604–615, 2006.

[Gar04] J. Garrigue. Relaxing the value restriction. In *International Symposium on Functional and Logic Programming*, number 2998 in LNCS. Springer-Verlag, April 2004.

[Gav03] F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 13(3) :365–376, 2003.

[Gav04] F. Gava. Design of Deparmental Metacomputing ML. In M. Bubak, D. van Albada, P. Sloot, and J. Dongarra, editors, *The International Conference on Computational Science (ICCS'04)*, LNCS, pages 50–53. Springer-Verlag, 2004.

[Gav05] F. Gava. *Approches fonctionnelles de la programmation paralléle et des méta-ordinateurs. Sémantiques, implantations et certification.* PhD thesis, LACL, Université Paris Val-de-Marne, 2005.

[Gav08] F. Gava. BSP Functional Programming ; Examples of a cost based methodology. In M. Bubak, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, editors, *The International Conference on Computational Science (ICCS'08), Part I*, volume 5101 of *LNCS*, pages 375–385. Springer-Verlag, 2008.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* Scientific and Engineering Computation Series. MIT Press, 1994.

[GF08] F. Gava and J. Fortin. Formal Semantics of a Subset of the Paderborn's BSPlib. In *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'08)*, pages 269–276, Washington, USA, 2008. IEEE Computer Society.

[GF09] F. Gava and J. Fortin. Two Formal Semantics of a Subset of the Paderborn University BSPlib. In *International Conference on Parallel, Distributed and Network-Based Processing (PDP 2009)*. IEEE Press, 2009. à paraître.

[GG08] I. Garnier and F. Gava. New Implementation of a Parallel Composition Primitive for a Functionnal BSP Language with application to the implementation of

P3L's algorithmic skeletons. Technical Report 5, LACL, Université Paris Est, 2008. Disponible à l'adresse `http://lacl.univ-paris12.fr//Labo/TechReports/2008/TR-LACL-2008-5.pdf`.

[Gib96] J. Gibbons. The third homomorphism theorem. *Journal of Functional Programming*, 6(4) :657–665, 1996.

[GK07] G. Gopalakrishnan and R. M. Kirby. Formal methods for mpi programs. *Electron. Notes Theor. Comput. Sci.*, 193 :19–27, 2007.

[GL03] F. Gava and F. Loulergue. Verifying Functional Bulk Synchronous Parallel Programs Using the Coq System. Technical Report 2003-02, LACL, Université Paris 12, 2003.

[GL04] F. Gava and F. Loulergue. Semantics of a Functional Bulk Synchronous Parallel Language with Imperative Features. In G. Joubert, W. Nagel, F. Peters, and W. Walter, editors, *International Conference on Parallel Computing (PARCO'04)*, pages 95–102, Dresden, 2004. North Holland/Elsevier.

[GL05] F. Gava and F. Loulergue. A Functional Language for Departmental Metacomputing. *Parallel Processing Letters*, 15(3) :289–304, 2005.

[GLD03] F. Gava, F. Loulergue, and F. Dabrowski. A Parallel Categorical Abstract Machine for Bulk Synchronous Parallel ML. In W. Dosch and R. Y. Lee, editors, *International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'03)*, pages 293–300. ACIS, 2003.

[GLT89] J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[Goo75] J. B. Goodenough. Exception handling : issues and a proposed notation. *Communications of the ACM*, 18(12) :683–696, 1975.

[Gor04] S. Gorlatch. Send-receive considered harmful : Myths and realities of message passing. *Transactions on Programming Languages and Systems (TOPLAS)*, 26(1) :47–56, 2004.

[GSF+07] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou. Ct : A Flexible Parallel Programming Model for Tera-scale Architectures. Technical report, Intel Research, 2007.

[Hai94] G. Hains. Parallel functional languages should be strict. In *International Federation for Information Processing (IFIP) Congress*, pages 527–532, 1994.

[HFA+96] P. H. Hartel, M. Feeley, M. Alt, L. Augustsson, P. Baumann, M. Beemster, E. Chailloux, C. H. Flood, W. Grieskamp, J. H. G. van Groningen, K. Hammond, B. Hausman, M. Y. Ivory, R. E. Jones, J. Kamperman, P. Lee, X. Leroy, R. D. Lins, S. Loosemore, N. Röjemo, M. Serrano, J.-P. Talpin, J. Thackray, S. Thomas, P. Walters, P. Weis, and P. Wentworth. Benchmarking implementations of functional languages with "Pseudoknot", a float-intensive benchmark. *Journal of Functional Programming*, 6(4) :621–655, 1996.

[HIT97] Z. Hu, H. Iwasaki, and M. Takeichi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *Transactions on Programming Languages and Systems (TOPLAS)*, 19(3) :444–461, 1997.

[HIT02] Z. Hu, H. Iwasaki, and M. Takeichi. An accumulative parallel skeleton for all. In *European Symposium on Programming (ESOP)*, number 2305 in LNCS, pages 83–97. Springer, 2002.

[HITT97] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *International Conference on Functional Programming (ICFP'97)*, pages 164–175, Amsterdam, Pays-bas, June 1997. ACM Press.

[HM05] J.-M. Hélary and A. Milani. About the efficiency of partial replication to implement distributed shared memory. Technical Report 1727, IRISA, 2005.

[HMC07] G. Henry, M. Mauny, and E. Chailloux. Typer la désérialisation sans sérialiser les types. *Technique et science informatiques*, 26 :1067–1090, 9 2007.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programmation. *Communication of the ACM*, 1969.

[Hof95] M. Hofmann. *Extensional concepts in intensional type theory.* PhD thesis, Laboratory for Foundations of Computer Science, The University of Edinburgh, 1995. Disponible à l'adresse `http://www.lfcs.informatics.ed.ac.uk/reports/95/ECS-LFCS-95-327/`.

[HS86] W. D. Hillis and G. L. Steele. Data parallel algorithms. *Communications of the ACM*, 1986.

[JMC96] H. Jifeng, Q. Miller, and L. Chen. Algebraic Laws for BSP Programming. In L. Bouge and Y. Robert, editors, *Euro-Par*, number 1124 in LNCS, pages 359–368. Springer, 1996.

[Kae92] S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Conference on LISP and Functional Programming (LFP'92)*, pages 193–204, New York, USA, 1992. ACM Press.

[Kru08] P. Krusche. Experimental Evaluation of BSP Programming Libraries. *Parallel Processing Letters*, 18(1) :7–21, 2008.

[Las99] I. Guérin Lassous. *Algorithmes parallèles de traitement de graphes : une approche basée sur l'analyse expérimentale.* PhD thesis, Université de Paris VII, 1999.

[LDG+07] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml System release 3.10, 2007. Pages web : `http://caml.inria.fr`.

[Let04] P. Letouzey. *Programmation fonctionnelle certifée – L'extraction de programmes dans l'assistant Coq.* PhD thesis, Université Paris-Sud, 2004.

[LGB05] F. Loulergue, F. Gava, and D. Billiet. Bulk Synchronous Parallel ML : Modular Implementation and Performance Prediction. In V. S. Sunderam, G. Dick van Albada, P. M. A. Sloot, and J. Dongarra, editors, *The International Conference on Computational Science (ICCS'05), Part II*, number 3515 in LNCS, pages 1046–1054. Springer, 2005.

[LHF98] F. Loulergue, G. Hains, and C. Foisy. A Calculus of Recursive-Parallel BSP Programs. In S. Gorlatch, editor, *International Workshop on Constructive Methods for Parallel Programming (CMPP'98)*, Research Report MIP-9805, pages 59–70. University of Passau, May 1998.

[LHF00] F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1–3) :253–277, 2000.

[LO06] I. Lynce and J. Ouaknine. Sudoku as a sat problem. In *9 th International Symposium on Artificial Intelligence and Mathematics (AIMATH'2006).* Springer, 2006.

[LOMP05] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3) :431–475, 2005.

[Lou98] F. Loulergue. BSML : Programmation BSP purement fonctionnelle. In D. Méry and G.-R. Perrin, editors, *Dixièmes Rencontres Francophones du Parallélisme (Renpar'10)*, pages 243–246, Strasbourg, june 1998.

[Lou01] F. Loulergue. Distributed Evaluation of Functional BSP Programs. *Parallel Processing Letters*, (4) :423–437, 2001.

[Lou03] F. Loulergue. Parallel Superposition for Bulk Synchronous Parallel ML. In P. M. A. Sloot and al., editors, *The International Conference on Computational Science (ICCS'03), Part I*, number 2659 in LNCS, pages 223–232. Springer-Verlag, june 2003.

[LP00] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *Transactions on Programming Languages and Systems (TOPLAS)*, 22(2) :340–377, 2000.

[McC94] W. F. McColl. Scalable parallel computing : A grand unified theory and its practical development. In B Pehrson and I Simon, editors, *Proc. 13th IFIP World Computer Congress. Volume 1 (Invited Paper)*. Elsevier, 1994.

[McC95] W. F. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today : Recent Trends and Developments*, volume 1000 of *LNCS*, pages 46–61. Springer-Verlag, 1995.

[McC96a] W. F. McColl. Scalability, portability and predictability : The BSP approach to parallel programming. *Future Generation Computer Systems (FGCS'96)*, 12 :265–272, 1996.

[McC96b] W. F. McColl. Universal computing. In L. Bouge and al., editors, *Euro-Par*, volume 1123 of *LNCS*, pages 25–36. Springer-Verlag, 1996.

[MH06] K. Matsuzaki and Z. Hu. Efficient implementation of tree skeletons on distributed-memory parallel computers. Technical Report METR06-65, Department of Mathematical Informatics, Graduate School of Information Science and Technology, The University of Tokyo, 2006.

[MHL01] A. Merlin, G. Hains, and F. Loulergue. A SPMD Environment Machine for Functional BSP Programs. In *Scottish Functional Programming Workshop (SFP'01)*, august 2001.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3) :348–375, December 1978.

[Mil02] Q. Miller. *BSP in a lazy functional context*, pages 37–50. Intellect Books, Exeter, UK, UK, 2002.

[MJM01] S. Marlow, S. P. Jones, and A. Moran. Asynchronous exceptions in haskell. In *Conference on Programming Languages Design and Implementation (PLDI'01)*, pages 274–285. ACM Press, 2001.

[MM08] L. Mandel and L. Maranget. Programming in JoCaml. In *17th European Symposium on Programming (ESOP'08)*, LNCS. Springer-Verlag, 2008.

[MMM+07] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In *Conference on Programming Language Design and Implementation (PLDI'07)*, pages 146–155. ACM Press, June 2007.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. Macqueen. *The Definition of Standard ML - Revised*. The MIT Press, May 1997.

[OSW99] M. Odersky, M. Sulzmann, and M. Wehr. Type Inference with Constrained Types. *Theory and Practice of Object Systems*, 5(1) :35–55, 1999.

[Our05] N. Oury. Extensionality in the calculus of constructions. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOL'05)*, volume 3603 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2005.

[OWW95] M. Odersky, P. Wadler, and M. Wehr. A second look at overloading. In *Conference on Functional Programming and Computer Architecture*, pages 135–146. ACM Press, June 1995.

[PR05] F. Pottier and D. Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[RDKT01] A. B. Romanovsky, C. Dony, J. Knudsen, and A. Tripathi, editors. *Advances in Exception Handling Techniques*, volume 2022 of *LNCS*. Springer, 2001.

[Rei93] J. H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, USA, 1993.

[Rém89] D. Rémy. Records and variants as a natural extension of ML. In *Sixteenth Annual Symposium on Principles Of Programming Languages (POPL'89)*, pages 77–88, Austin, Texas, USA, 1989. ACM Press.

[Rep99] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[RK01] A. Romanovsky and J. Kienzle. *Action-oriented exception handling in cooperative and competitive concurrent object-oriented systems*, pages 147–164. Springer-Verlag, New York, USA, 2001.

[SC01] A. Stewart and M. Clint. BSP-style Computation : a Semantic Investigation. *The Computer Journal*, 44(3) :174–185, 2001.

[SCG00] A. Stewart, M. Clint, and J. Gabarró. Axiomatic Frameworks for Developing BSP-Style Programs. *Parallel Algorithms and Applications*, 14 :271–292, 2000.

[SCG04] A. Stewart, M. Clint, and J. Gabarr. Barrier synchronisation : Axiomatisation and relaxation. *Formal Aspects of Computing*, 16(1) :36–50, april 2004.

[SG98] M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.

[Ski98] D.B. Skillicorn. Building BSP Progams Using the Refinement Calculus. In D. Merry, editor, *Workshop on Formal Methods for Parallel Programming : Theory and Applications*, pages 790–795. Springer, 1998.

[Smo95] G. Smolka. The oz programming model. In *Computer Science Today, Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, 1995.

[Smo01] G. Smolka. A semi-syntactic soundness proof for hm(x). Research Report 4150, INRIA, March 2001. Disponible à l'adresse `ftp://ftp.inria.fr/INRIA/publication/RR/RR-4150.ps.gz`.

[SP02] C. Skalka and F. Pottier. Syntactic type soundness for HM(X). In *Workshop on Types in Programming (TIP'02)*, volume 75 of *Electronic Notes in Theoretical Computer Science*, Dagstuhl, Germany, July 2002.

[SSB+95] T. L. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf : A parallel workstation for scientific computation. In *International Conference on Parallel Processing (ICPP'95)*, pages 11–14, 1995.

[Tal93] J.-P. Talpin. *Aspects Théoriques et Pratiques de l'Inférence de Type et d'Effets*. Thèse de doctorat, Université Paris VI, École Nationale Supérieure des Mines de Paris, Mai 1993.

[Tis98] A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms*. PhD thesis, Oxford University Computing Laboratory, 1998.

[TJ92] J.-P. Talpin and P. Jouvelot. The type and effect discipline. Rapport technique EMP-CRI A/206, École des Mines de Paris, 1992.

[TL07] J. Tesson and F. Loulergue. Formal Semantics for the DRMA Programming Style Subset of the BSPlib Library. In J. Weglarz, R. Wyrzykowski, and B. Szymanski, editors, *Seventh International Conference on Parallel Processing and Applied Mathematics (PPAM'07)*, LNCS. Springer, 2007.

[Tri99] P. W. Trinder. Motivation for Glasgow distributed Haskell, a non-strict Functional Language. In T. Ito and T. Yuasa, editors, *Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA'99)*, pages 72–81, Sendai, Japan, 1999. World Scientific.

[TW04] J. L. Traeff and J. Worringen. Verifying Collective MPI Calls. In *The 11th EuroPVM/MPI conference*, LNCS. Springer, 2004.

[Val90] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8) :103–111, August 1990.

[Wad92] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2 :461–493, 1992.

[WF94] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115 :38–94, 1994.

[Wri92] A. K. Wright. Typing references by effect inference. In *4th European Symposium on Programming (ESOP'92)*, pages 473–491. Springer-Verlag, 1992.

Résumé

Exprimer le parallèlisme dans la programmation de manière simple et performante est un défi auquel l'informatique fait face, en raison de l'évolution actuelle des architectures matérielles. BSML est un langage permettant une programmation parallèle de haut niveau, structurée, qui participe à cette recherche. En s'appuyant sur le cœur du langage existant, cette thèse propose d'une part des extensions qui en font un langage plus général et plus simple (traits impératifs tels que références et exceptions, syntaxe spécifique...) tout en conservant et étendant sa sûreté (sémantiques formelles, système de types...), et d'autre part une méthodologie de développement d'applications parallèles certifiées.

Mots-clés: parallélisme, programmation de haut niveau, programmation fonctionnelle, sémantique, sûreté des langages, preuves de programmes, typage, BSP

Abstract

Finding a good paradigm to represent parallel programming in a simple and efficient way is a challenge currently faced by computer science research, mainly due to the evolution of machine architectures towards multi-core processors. BSML is a high level, structured parallel programming language that takes part in this research in an original way. By building upon existing work, this thesis extends the language and makes it more general, simple and usable with added imperative features such as references and exceptions, a specific syntax, etc. The existing formal and safety characteristics of the language (semantics, type system...) are preserved and extended. A major application is given in the form of a methodology for the development of fully proved parallel programs.

Keywords: parallelism, high level programming, functional programming, semantics, safety of programs, program proofs, type systems, BSP