Environnement pour le développement et la preuve de correction systématiques de programmes parallèles fonctionnels

Julien Tesson

UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE SCIENCES ET TECHNOLOGIES

Laboratoire d'Informatique Fondamentale d'Orléans

THÈSE présentée par :

Julien TESSON

soutenue le : **8 Novembre 2011**

pour obtenir le grade de : **Docteur de l'université d'Orléans**

Discipline/ Spécialité : **Informatique**

Environnement pour le développement et la preuve de correction systématiques de programmes parallèles fonctionnels

THÈSE dirigée par :

Frédéric LOULERGUE	Professeur des Universités, Université d'Orléans

RAPPORTEURS :

David CACHERA	Maître de conférence HDR, ENS Cachan Bretagne
Marco DANELUTTO	Associate Professor, Università di Pisa

JURY :

David CACHERA	Maître de conférence HDR, ENS Cachan Bretagne,	**Rapporteur**
Marco DANELUTTO	Associate Professor, Università di Pisa,	**Rapporteur**
Pascal FRADET	Chargé de recherche, INRIA Rhône-alpes	
Jacques JULLIAND	Professeur, Université de Franche-Comté,	**Président de jury**
Frédéric LOULERGUE	Professeur, Université d'Orléans,	**Directeur de thèse**
Pierre-Étienne MOREAU	Professeur, École des Mines de Nancy	
Jan-Georg SMAUS	Professeur, Université de Toulouse	

À Sonia

Remerciemements

> "Il est fort dangereux Frodon de sortir de chez soi. On prend la route et si l'on ne regarde pas où l'on met les pieds, on ne sait pas jusqu'où cela peut nous mener."
>
> Bilbon Sacquet

Je tiens tout d'abord à remercier Jacques Julliand de m'avoir fait l'honneur de présider mon jury de thèse, Marco Danelluto et David Cachera d'avoir accepté de rapporter ma thèse et Pascal Fradet, Pierre-Étienne Moreau et Jan-Georg Smaus d'avoir bien voulu participer à mon jury. Frédéric Loulergue m'a soutenu et guidé tout au long de cette thèse et je l'en remercie grandement. J'espère que l'on pourra continuer à travailler ensemble sur les passionnantes perspectives que promettent nos projets de recherches !

Je souhaite également remercier Armelle Bonenfant avec qui j'ai goûté pour la première fois à la recherche lors d'un stage de maîtrise, et dont les conseils ont mis mes pas sur un tortueux chemin qui m'a mené à cette thèse. J'ai adoré ce chemin.

Merci à toutes les personnes du LIFO et du département informatique, pour l'accueil qui m'a été fait et pour la place importante donnée aux doctorants. Je ne pourrai pas citer tous ceux avec qui j'ai eu plaisir à échanger, que ce soit sur des sujets de recherche, d'enseignement, ou sur n'importe quel autre sujet. Il est également plaisant de pouvoir discuter avec nos deux *grands chefs* en toute liberté et de savoir que leurs oreilles sont attentives et bien intentionnées. Merci également à Florence et Isabelle sans qui nous serions perdus !

Un grand merci à mes compagnons de voyage de l'ADSO et du LIFO, désolé de ne pouvoir tous vous citer, et plus particulièrement ceux avec qui j'ai partagé mon bureau : Claire, Yannick, Hélène et Simon ; la route est toujours plus agréable à faire à plusieurs. Je tiens à remercier plus particulièrement Matthieu : nous avons commencé le long périple de nos thèses ensemble, il était bon de pouvoir se plaindre mutuellement et de partager nos routes... et que de chemins parcourus ! ! Je suis heureux que nous parcourrions la dernière ligne droite de concert et j'espère que les chemins tortueux qui suivront se croiseront fréquemment.

Je remercie ma famille, à qui je dois ma manie de ne pas regarder où je met les pieds et de préférer regarder le paysage qui m'entoure à la recherche de nouveaux sentiers à explorer.

Et le meilleur pour la fin, toujours ...
Un incommensurable merci à Sonia pour m'avoir soutenu quand j'en avais le plus besoin, et pour m'avoir poussé, aussi, quand il le fallait. J'espère pouvoir te rendre la pareille bientôt ! Je souhaite que nos chemins restent longtemps entrelacés, que longtemps encore nous échangions nos visions, nos richesse et nos espérances.

Table des matières

Liste des figures

Introduction

1

Sommaire

« Le diable se cache dans les détails. » Cet adage s'applique particulièrement bien à la problématique que rencontre tout programmeur. Le diable, pour lui, c'est le *bug*, l'erreur qui fait dysfonctionner le programme. Généralement, cette erreur ne vient pas des grandes lignes de la conception d'un programme, aussi complexes soient elles. Au contraire, l'erreur vient souvent d'un infime détail : une erreur dans un nom de variable, une initialisation oubliée, une comparaison d'entiers inversée, le parcours d'un ensemble qui oublie le premier élément, une fonction utilisée dans un contexte pour lequel elle n'était pas prévue, *etc.*

Les langages de programmation ont évolué pour permettre la détection de certaines de ces erreurs et offrir de meilleures abstractions de la machine à programmer. L'utilisation d'un ramasse-miette [1] décharge le programmeur de la tâche de gestion de la mémoire, source d'innombrables erreurs. Les compilateurs modernes sont en mesure, après analyse statique du code source, d'avertir le programmeur de l'utilisation d'une variable non initialisée, de l'incompatibilité de types entre une opération et les expressions auxquelles elle est appliquée. Les erreurs les plus basiques peuvent ainsi être évitées. Les langages

1. ou encore glaneur de cellules, *Garbage Collector*, GC

orientés objet ou les langages fonctionnels offrent des paradigmes de programmation permettant un plus haut niveau d'abstraction et une plus grande réutilisabilité du code produit, permettant ainsi de capitaliser sur une base de code que l'on pourra constamment améliorer.

Toutes ces améliorations restent cependant limitées. Il est d'ailleurs encore couramment admis qu'un programme soit instable, qu'il "plante" de temps en temps. On se contente de redémarrer le programme, ou tout le système, et de se plaindre un peu. Mais avec la généralisation des systèmes informatiques, leur ubiquité, nous leur confions la gestion d'un nombre toujours croissant d'éléments de notre vie quotidienne. Ils gèrent notre musique, nos photos, toutes nos communications, nos informations personnelles, nos achats, nos factures, nos comptes bancaires, *etc.* Il est probable que les consommateurs de logiciels deviennent plus exigeants et réclament des garanties de stabilité. Les éditeurs de logiciels en mesure de fournir de telles garanties auront, à l'avenir, un avantage stratégique.

Conséquence de l'ubiquité des systèmes informatiques, les programmes se trouvent de plus en plus souvent en position de contrôler des systèmes critiques pour lesquels un défaut de conception peut avoir des conséquences importantes, voire vitales. Avec la montée en puissance de la domotique par exemple, on peut confier à un système informatique l'ouverture de toutes les issues d'un bâtiment, son système de chauffage, d'éclairage, *etc.* lorsqu'un tel système vient à tomber en panne les conséquences peuvent être graves. Au plus extrême, on pensera au pilotage automatique de moyens de transports collectifs, aux matériels médicaux ou encore aux centres de contrôle de centrales nucléaires, où un dysfonctionnement peut mettre en danger de nombreuses vies humaines. Dans ces cas extrêmes, l'erreur ne peut être tolérée.

1.1 Comment garantir la fiabilité d'un programme ?

Tout d'abord, un critère de correction doit être défini. Ce critère peut concerner la consommation de ressources (mémoire, temps, *etc.*), la réactivité, la sûreté d'exécution (le programme n'exécute pas des opérations interdites) ou le résultat du calcul. Le critère auquel nous nous intéressons est la correction du résultat du calcul. Pour vérifier que le résultat calculé par un programme est fiable, il faut être en mesure de décrire ce résultat. Une telle description du résultat s'appelle une spécification. Suivant le niveau de garantie que l'on souhaite et la méthode que nous utilisons pour l'obtenir, cette spécification prendra diverses formes.

Nous verrons que pour les méthodes de tests (section 1.1.1), il s'agit généralement d'un ensemble de paires (paramètres, résultat). Les méthodes formelles, qui visent à garantir une forme de correction d'un programme en s'appuyant sur un raisonnement logique, généralement mécanisé, utilisent souvent des langages d'expressivité variable pour exprimer les propriétés attendues du résultat en fonction des paramètres du programme : logiques temporelles, logiques du premier ordre, d'ordre supérieur, *etc.*. On peut distinguer dans les méthodes formelles deux familles d'approches :

– la preuve de correction *a posteriori* (section 1.1.2), où une spécification est élaborée indépendamment du programme puis on vérifie que celui-ci, ou un modèle abstrait de celui-ci correspond à la spécification ;
– les méthodes constructives (section 1.1.3), où une spécification est utilisée comme point de départ pour obtenir un programme par raffinements et/ou transformations successives, chaque étape devant garantir la préservation de la spécification.

1.1.1 Par des tests

Méthode la plus répandue aujourd'hui dans l'industrie, le test consiste à exécuter les programmes produits sur un certain nombre de valeurs d'entrées et de vérifier que le résultat obtenu, ou le comportement observé, est conforme aux attentes. La spécification du programme est dans ce cas un ensemble de paires associant les données d'entrées au résultat attendu pour celles-ci. Les tests peuvent être faits à l'échelle d'un programme complet, ou pour ses sous-composants (tests unitaires). Certains travaux visent à assurer une couverture maximale du code en analysant les portions de code testé par un jeu de données, d'autres à générer automatiquement des jeux de tests en se basant sur une analyse statique du code et une spécification sous une forme logique du résultat attendu. Cependant, même dans les cas où l'ensemble du code peut être couvert, il n'est testé que pour un nombre limité de valeurs. Le test ne donne aucune certitude pour les combinaisons de valeurs non-testées, et celles-ci peuvent être en nombre infini. Tester le code permet de détecter puis d'éliminer bon nombre d'erreurs à moindre coût, mais ne peut fournir qu'une garantie limitée sur la correction d'un programme. Pour garantir l'absence d'erreurs, il est nécessaire de faire appel aux méthodes formelles.

1.1.2 Par vérification *a posteriori*

Vérification de modèle

La vérification de modèle[2] [40, 119] n'est pas initialement une méthode destinée à vérifier des programmes mais plutôt à la vérification de systèmes dans le sens le plus général du terme (circuit électronique, protocoles de communication, *etc.*).

Le programme ou système est dans un premier temps modélisé sous forme d'un système d'états/transitions. Dans un second temps, le système d'états/transitions est transformé sous une forme permettant de vérifier automatiquement la satisfaction d'une propriété, généralement exprimée dans une logique temporelle. Les chemins dans le système d'état-transition ne vérifiant pas la propriété traduisent une faille du système. Cependant, une expertise est souvent nécessaire pour diagnostiquer la source de cette faille : s'agit-il d'un réel problème du système ou est-ce le modèle qui n'est pas vraiment fidèle à la réalité ?

L'avantage de cette méthode réside dans le fait qu'elle est complètement automatisable. L'inconvénient majeur reste que le nombre des états atteignables est sujet à une explosion combinatoire en fonction de la taille du système vérifié. De nombreuses

2. *Model checking*

méthodes ont été développées afin de pallier ce problème, notamment en utilisant des représentations symboliques des états. Cependant, quand le système représente un programme, le nombre d'états est rapidement trop important même pour des représentations compactes. Cette technique est inapplicable pour des systèmes où le nombre d'états atteignables est infini. Il faut dans ce cas construire par des méthodes d'abstraction un modèle approximatif du système représentable par un nombre fini d'états.

Une autre famille de méthodes formelles complètement automatisables est celle des analyses statiques.

Analyse statique

Les analyses statiques permettent de calculer des approximations des sémantiques des programmes. Elles ramènent l'étude d'une propriété non décidable à l'étude d'une *approximation décidable* de cette propriété.

L'interprétation abstraite [47] est un cadre générique basé sur les théories des treillis et des points fixes qui permet la construction systématique de sémantiques abstraites sur des domaines finis et infinis. L'interprétation abstraite se base sur la résolution d'un ensemble d'équations (d'un point fixe) et ses limites se situent, comme d'autres méthodes d'analyse statique, dans la possibilité ou non de résoudre ces équations. Les raisons de cette impossibilité peuvent être des problèmes de décidabilité ou des problèmes de complexité.

L'intervention humaine est nécessaire lorsqu'il n'existe pas de procédure de décision, que la précision des approximations nécessaires est trop importante (dans le pire cas, la sémantique concrète), ou tout simplement lorsque la preuve de la propriété recherchée repose sur une collection non bornée de cas particuliers de raisonnements.

Logiques de Hoare

Les logiques de Hoare [86] sont des logiques permettant de raisonner sur les propriétés des programmes. Il s'agit d'établir des jugements de la forme $\{\mathsf{P}\}\ \mathsf{p}\ \{\mathsf{Q}\}$, où P et Q sont des assertions logiques (en général en logique du premier ordre) et où p est un programme écrit dans un langage de programmation donné. Selon qu'on se place dans un cadre de correction totale ou de correction partielle, la signification de ce triplet de Hoare est différente.

En correction totale, $\{\mathsf{P}\}\mathsf{p}\{\mathsf{Q}\}$ signifie que si l'état de la mémoire vérifie l'assertion logique P, appelée *pré-condition* alors l'exécution du programme impératif p *termine* dans un état de la mémoire qui vérifie l'assertion logique Q, appelée *post-condition*. En correction partielle la terminaison n'est pas prouvée, et le triplet de Hoare signifie que si l'état de la mémoire vérifie la pré-condition P et que le programme p termine, alors l'état de la mémoire après l'exécution de p vérifie la post-condition.

Les logiques de Hoare sont définies par des systèmes d'inférence sur les triplets de Hoare. Les arbres de dérivation contiennent des triplets de Hoare pour chaque *instruction* du programme. Il s'agit donc d'une méthode qu'il est souhaitable d'automatiser au maximum. Par un calcul de pré-conditions les plus faibles [56], il est possible, partant d'un

programme correct et d'une post-condition, de calculer la pré-condition la plus faible qui permet d'établir le triplet de Hoare correspondant. Toutefois, ce calcul ne peut pas être complètement automatisé : pour certaines constructions des langages de programmation, il est nécessaire de donner des annotations intermédiaires, par exemple des invariants de boucle.

Pour réaliser des outils logiciels basés sur cette méthode de vérification de programme deux approches sont possibles :

- encoder directement une logique de Hoare dans un outil dédié,
- ou procéder en deux étapes :
  - générer les *obligations de preuve* (qui correspondent aux coupures introduites dans l'arbre de dérivation du triplet de Hoare) à l'aide d'outils dédiés appelés *Verification Condition Generators* ou VCG, qui sont basés sur un calcul de pré-conditions les plus faibles,
  - puis utiliser un ou des prouveurs automatiques ou interactifs non dédiés pour prouver les obligations de preuve.

C'est cette seconde méthode qui est la plus mise en pratique, par exemple dans les outils Why [59] ou Spec# [9].

L'avantage de ces outils est la forte automatisation lorsque les obligations de preuve peuvent être prouvées automatiquement ; le problème est que, lorsque les outils automatiques échouent, les obligations de preuve générées sont difficiles à lire et à prouver dans un prouveur interactif. Il est alors nécessaire de guider les prouveurs automatiques mais ceci demande également une grande expertise. L'art de l'utilisation de ces outils réside donc dans le fait de décomposer les étapes de raisonnement à l'aide de lemmes qui pourront être prouvés automatiquement et utilisés par les prouveurs pour résoudre les obligations de preuve.

Preuve de programme dans un assistant de preuve

Une autre solution pour prouver la correction d'un programme consiste à le représenter directement dans un assistant de preuve pour raisonner dessus. On distingue trois techniques :

- Le *plongement profond* du langage dans la logique de l'assistant de preuve : la syntaxe et la sémantique du langage sont décrites dans la logique de l'assistant de preuve, généralement sous la forme d'un type décrivant les arbres de syntaxe abstraite admissibles et de règles d'évaluation et de typage des arbres de syntaxe abstraite. Cette modélisation permet de montrer des propriétés sur le langage plongé et sur les termes de ce langage. Un inconvénient de cette méthode est la lourdeur de la représentation des programmes sous forme d'arbre de syntaxe abstraite et des propriétés sémantiques de ceux-ci. Le projet Bali [114] est un exemple d'utilisation de plongement profond. Une modélisation d'un sous-ensemble du langage Java dans l'assistant de preuve Isabelle/HOL y est utilisé pour prouver diverses propriété de ce langage.
- Le *plongement superficiel* profite de la forte expressivité des langages des assistants de preuves actuels pour utiliser la logique de l'assistant de preuve comme un langage

de programmation. L'avantage de cette technique est que le raisonnement sur les programmes peut profiter immédiatement de toute la mécanique de l'assistant de preuve, des bibliothèque existantes, *etc.* Le compilateur C certifié CompCert [96], principalement écrit à l'aide de Coq en est un exemple. La représentation de tous les programmes d'un langage de programmation complet dans le langage d'un assistant de preuve n'est toutefois pas toujours possible.

– La représentation d'un programme par une *formule caractéristique* [36], décrivant dans la logique d'ordre supérieur de l'assistant de preuve le comportement du programme. Cette formule caractéristique est construite directement à partir du code source du programme. Contrairement au plongement profond, elle ne fait pas directement référence à la construction syntaxique du programme. À la différence d'un plongement superficiel, la formule caractéristique ne repose pas directement sur la sémantique du langage dans lequel est plongé le langage initial, mais sur une modélisation de la sémantique du langage plongé.

Il est à noter que les méthodes de vérification présentées précédemment font elles-mêmes l'objet de vérification dans des assistants de preuve : le *model-checking* [118], les analyses statiques [48, 29, 30], la génération d'obligations de preuve [82, 73].

1.1.3 Par construction

Une autre famille d'approches consiste à s'assurer tout au long du développement que le programme en construction respecte bien sa spécification.

Algorithmique constructive

Lorsque le prix Turing est remis à John Backus en 1977, celui-ci propose un langage fonctionnel accompagné d'une algèbre de programme [5]. Le but est que :

« *le programmeur puisse utiliser des lois algébriques simples et un ou deux théorèmes fondamentaux pour résoudre des problèmes et produire des preuves dans le même style mécanique que nous utilisons pour résoudre des problèmes d'algèbre au lycée.* »

Bird et Meertens ont mis en pratique cette approche dans le formalisme qui porte à présent leur nom *Bird-Meertens Formalism*[3] (ou BMF) [12, 106]. Dans ce formalisme, les programmes sont initialement exprimés de façon naïve dans un langage fonctionnel d'ordre supérieur. Cette description sert de spécification et l'implantation des fonctions utilisées importe peu. Ensuite à l'aide de règles algébriques, ces spécifications sont transformées en un programme efficace. Initialement restreint aux listes, le formalisme à ensuite été étendu aux arbres binaires [14] puis aux arbres d'arité non-bornée [107]. Plus tard, Malcolm [101] permet la généralisation de ces algèbres à toute structure de données finie en s'appuyant sur les travaux de Hagino [76] sur les types de données catégoriques[4]. Cette généralisation, par une étude plus systématique des schémas de récursion utilisés en programmation fonctionnelle, a permis une généralisation des règles d'optimisation.

3. Ce formalisme est parfois appelé Squiggol du fait des nombreux symboles utilisées donnant parfois une impression de *gribouillis* (*squiggle*).

4. en anglais : *Categorical Data Types* ou CDT.

Parmi les plus simples, les catamorphismes sont les fonctions dont le schéma de récursion correspond aux constructeurs d'une structure de données : une valeur est calculée en *remplaçant* les constructeurs d'une structure de données par d'autres fonctions. Par exemple, les constructeurs habituels d'une liste sont `nil` d'arité 0 pour la liste vide et `cons` d'arité 2, le premier argument étant une valeur et le second une liste. Un catamorphisme `f` pour ce type de données sera une fonction qui pour `nil` renvoie une valeur `v` et pour (`cons a l`) retournera (`g a (f l)`). La fonction est donc définie de façon unique par v et `g` qui correspondent chacun à un constructeur ; on notera cette fonction $(\!|v,g|\!)$, ainsi la fonction calculant la longueur d'une liste est le catamorphisme $(\!|1,(+)|\!)$. Le formalisme BMF a également été généralisé à la dérivation de programmes à partir de relations [15]. L'utilisation d'une relation pour spécifier un résultat permet de le faire de façon non-déterministe. Une mécanisation de ce formalisme a récemment été développée dans le langage de programmation avec types dépendants Agda [111].

Affinement de spécification

Une autre approche consiste à partir d'une spécification sous la forme pré/post-conditions, puis de raffiner ces conditions jusqu'à l'obtention d'un programme exécutable. Le raffinement consiste à sur-spécifier afin d'obtenir une description plus précise du programme attendu. La méthode B [1] est un exemple d'application de ce type de raisonnement.

Les différents composants d'un programme sont décrits dans des machines abstraites. Une telle machine comprend un état interne, une initialisation, un invariant sur l'état et des opérations qui changent l'état. Des machines de plus en plus concrètes sont ensuite construites, en vérifiant à chaque étape que le raffinement est une implantation du comportement attendu de la machine précédente. À la dernière étape du raffinement, la machine décrit un programme exécutable pouvant être implanté directement dans un langage impératif comme Ada ou C. Cette méthodologie est reprise dans le projet FoCaLiZe [79] qui permet la dérivation de programmes vers un langage utilisant les paradigmes de la programmation objet et de la programmation fonctionnelle, en un mot, OCaml. La correction des étapes de dérivation est vérifiée par un prouveur automatique ; si celui-ci échoue, la preuve doit être faite en Coq.

1.2 Du parallélisme partout

Un phénomène nouveau vient encore compliquer la tâche des programmeurs : le parallélisme. Jusqu'à récemment, la rapidité des unités de calcul était en constante augmentation, et les programmes séquentiels, c'est à dire utilisant une seule unité de calcul, profitaient directement de cette accélération du matériel. Cependant, depuis quelques années maintenant, les constructeurs sont confrontés à des problèmes de dissipation de la chaleur et surtout de consommation d'énergie qui les ont poussés à ne plus augmenter la rapidité des unités de calcul mais à multiplier leur nombre sur une même puce. Les architectures parallèles, jusqu'à récemment confinées aux centres de calcul scientifique,

se retrouvent maintenant dans toutes les ordinateurs grand public, et jusque dans les téléphones portables.

1.2.1 Architectures parallèles

Les architectures parallèles sont traditionnellement classées selon la taxonomie de Flynn [60] qui distingue quatre types d'architecture en fonction de leur capacité à traiter en même temps un ou plusieurs flots de données, par un ou plusieurs flots d'instructions : l'architecture SISD (*Single Instruction, Simple Data*) correspond à l'architecture simple mono-processeur traitant une donnée à la fois. La plupart des processeurs modernes dispose de jeux d'instructions vectorielles entrant dans la catégorie SIMD (*Single Instruction Multiple Data*) où une même instruction est appliquée à plusieurs données simultanément. Les architectures MISD (*Multiple Instruction, Single Data*) sont généralement associées aux *pipelines* où une donnée subit une série de transformations successives. Enfin, l'architecture MIMD (*Multiple Instruction, Multiple Data*) correspond à la quasi totalité des architectures parallèles actuelles : elle est composée de multiples unités de calcul disposant de registres séparés et donc utilisant des données distinctes.

Cependant, le moindre ordinateur de bureau actuel regroupe tous ces modèles à différent niveaux. En effet, les processeurs modernes ont en interne un pipeline d'instructions. Ils offrent au programmeur un jeu d'instructions *vectorielles* opérant sur plusieurs entiers ou nombres à virgule flottante en même temps (SIMD). Enfin les processeurs multi-cœurs sont devenus la norme (MIMD).

Les architectures parallèles (MIMD) sont généralement divisées en deux catégories, les architectures à mémoire partagée, et les architectures à mémoire distribuée. Dans une machine multi-cœurs, tous les cœurs se partagent un même espace mémoire, tandis qu'une grappe d'ordinateurs ou les nœuds d'un centre de calcul sont globalement organisés suivant une architecture à mémoire distribuée (cependant, chaque nœud est souvent une machine MIMD à mémoire partagée).

1.2.2 Modèles de programmation parallèle répandus

Les machines parallèles requièrent des modèles de programmation adaptés. Les modèles les plus répandus aujourd'hui sont encore très proches des architectures pour lesquelles ils ont été conçus. On trouvera en effet des modèles de programmation à mémoire partagée, et des modèles de programmation par passage de message, initialement conçus pour la programmation de machines à mémoire distribuée.

Bien que quelques environnements essaient de porter les modèles de programmation à mémoire partagée sur des machines à mémoire distribuée, des problèmes de performances se posent pour le passage à l'échelle. À l'inverse, les modèles de programmation par passage de message peuvent profiter d'une mémoire partagée pour optimiser le passage de message et atteindre des performances en terme de temps d'exécution comparables à celles obtenues en utilisant le modèle de programmation à mémoire partagée (avec généralement un surcoût en terme de consommation mémoire).

Programmation par mémoire partagée

Le modèle le plus couramment utilisé aujourd'hui pour la programmation en mémoire partagée est celui des processus légers tels que les Pthreads [113] ou les processus Java. Avec Pthreads, les processus sont créés explicitement dans le programme par un appel système prenant une procédure en paramètre.

Dans ce modèle, chaque processus dispose d'un espace mémoire privé inaccessible aux autres processus, et d'un accès à un espace mémoire partagé. Les variables de l'espace partagé sont accessibles à tous les processus en lecture et en écriture. C'est au programmeur de gérer les accès concurrents aux ressources partagées. Le problème majeur de cette approche est la possibilité d'écrire des programmes se trouvant dans une situation de compétition pour l'accès à une donnée [5], menant à un comportement non déterministe.

Prenons le cas simple d'un même programme exécuté dans deux processus parallèles A et B accédant à une variable partagée x initialisée à 0 :

A	B	
if(x = 0)	if(x = 0)	(1)
x := x + 1	x := x + 1	(2)

Que vaut la variable x après l'exécution de ce programme ? Les deux processus ont dû accéder à la mémoire l'un après l'autre pour chacune de leurs instructions ; cependant il n'est pas possible de savoir dans quel ordre ces instructions ont été exécutées. Si l'ordre d'exécution est A1, A2, B1 le résultat est 1 ; mais si l'ordre est A1, B1, A2, B2, alors x vaut 0 lors des deux tests et les deux processus incrémentent donc la valeur de x. Suivant l'ordre d'exécution des instructions on obtiendra donc deux valeurs différentes. Un tel comportement non-déterministe est rarement souhaité (nous verrons lorsque nous aborderons la vérification de tels programmes que dans des cas un peu plus compliqués, le comportement de programmes avec processus légers n'est même pas défini). Il est donc nécessaire d'utiliser des mécanismes permettant de séquentialiser le comportement des processus pour l'accès à certaines variables.

Cette séquentialisation se fait par la mise en attente des processus souhaitant accéder à une ressource partagée. Cependant, pour que le programme reste parallèle, le blocage n'est pas global mais réalisé ressource par ressource. Or, il peut arriver qu'un processus bloquant l'accès à une ressource soit lui-même bloqué pour l'accès à une autre ressource, celle-ci étant bloquée par un processus lui-même bloqué pour accéder à une autre ressource, et ainsi de suite. Si ces dépendances aux ressources bloquées sont circulaires, on a une situation d'inter-blocage où plus aucun processus ne peut progresser.

Programmation par passage de messages

Le standard actuel pour la programmation par passage de messages est MPI [126]. Il existe deux implantations répandues de ce standard, OpenMPI et MPICH. De plus les constructeurs de machines massivement parallèles fournissent souvent une implantation spécifique tirant partie des particularités de leur architecture.

5. En anglais *data race condition*

MPI permet une certaine abstraction de la machine parallèle sur laquelle il est exécuté. Par exemple, le nombre de processus est décidé au moment de l'exécution du programme et n'est pas obligatoirement lié au nombre de processeurs disponibles. Chaque processus se voit attribuer un numéro d'identification, les processus peuvent ensuite être regroupés dans différents *communicateurs* afin de structurer les calculs.

Les processus communiquent entre eux en s'envoyant des messages, c'est-à-dire des paquets de données, à l'aide des primitives *send* et *receive*. Diverses variantes de ces primitives existent, suivant qu'elles sont bloquantes ou non, qu'elles utilisent une mémoire tampon ou non, *etc.* Ces primitives sont parfois comparées à l'instruction goto [37] car leur utilisation est complexe et sujette aux problèmes d'inter-blocage et de non-déterminisme.

Le non-déterminisme est provoqué par la possibilité pour un processus de recevoir des messages de deux processus différents. Ne sachant pas lequel va envoyer sa donnée le premier, il n'est pas possible de déterminer quel sera la valeur reçue. L'inter-blocage se produit lors de l'utilisation des méthodes bloquantes de communication. Ici encore, une dépendance circulaire entre les processus est possible et provoque le blocage de tous les processus impliqués.

Le standard MPI propose également quelques opérations collectives structurées[6] : diffusion collective, répartition de données parmi les processeurs d'un groupe, rassemblement des données d'un groupe sur un même processeur, calcul des préfixes pour un opérateur donné. Ces opérations présentent deux avantages : leur implantations peuvent tirer partie des spécificités d'une architecture et donc peuvent être plus performantes que la même opération implantée à l'aide des primitives *send* et *receive*. Mais, surtout, l'usage des opérations collectives évite la conception de schémas de communication complexes utilisant les primitives *send* et *receive*, susceptibles d'introduire des erreurs de communication, du non-déterminisme ou des inter-blocages. Ces opérations collectives sont un premier pas vers l'utilisation des squelettes algorithmiques que nous présentons à la section 1.3.3.

En MPI, les programmes sont généralement écrits suivant le principe programme unique, données multiples (*Single Program Multiple Data* ou SPMD). Cela veut dire que le même programme est écrit pour tous les processeurs de la machine et qu'il contient des expressions dépendantes du numéro du processeur (qui est lié extérieurement). Deux défauts majeurs en découlent :

- il peut être difficile pour le programmeur – et il est indécidable – de distinguer quelles parties du programme sont dépendantes du processeur sur lequel elles s'exécutent, de celles qui sont globales (c'est-à-dire indépendantes du processeur) et qui décrivent le comportement global de l'algorithme parallèle,
- l'ordre de lecture du programme peut ne plus correspondre à l'ordre d'exécution ce qui rend malaisée la mise au point.

6. respectivement *broadcast, scatter, gather, scan*

1.2.3 Comment garantir la fiabilité d'un programme parallèle ?

Nous l'avons vu, la conception de programmes parallèles est sujette à de multiples difficultés : en plus de celles inhérentes à la programmation séquentielle, le programmeur est confronté aux problèmes de gestion des communications entre processeurs ou d'accès concurrents à des mémoires partagées qui entraînent indéterminismes et inter-blocages. La vérification des programmes est donc d'autant plus nécessaire, et d'autant plus difficile.

Globalement, les mêmes méthodes que pour la vérification de programmes séquentiels sont utilisées. Elles nécessitent cependant des adaptations et de nouvelles propriétés doivent être vérifiées (déterminisme, absence d'inter-blocage). Nous ne présentons pas systématiquement les versions pour programmes parallèles des méthodes de la section 1.1, mais illustrons les principales difficultés rencontrées.

Programmation en mémoire partagée par processus légers

Prenons l'exemple de la composition de deux programmes parallèles A et B très simples afin d'observer les adaptations nécessaires :

$$\begin{array}{c|c|l} \mathsf{A} & \mathsf{B} & \\ \mathsf{l}_1 := \mathsf{G}_2; & \mathsf{l}_2 := \mathsf{G}_1; & (1) \\ \mathsf{G}_1 := 1 & \mathsf{G}_2 := 2 & (2) \end{array}$$

Les l_i sont des variables locales et les G_i des variables partagées (initialement à 0). Si l'on utilise une sémantique d'entrelacement, il faut ici considérer toutes les combinaisons possibles : $\mathsf{A1};\mathsf{B1};\mathsf{A2};\mathsf{B2}$ ou $\mathsf{A1};\mathsf{A2};\mathsf{B1};\mathsf{B2}$ ou $\mathsf{B1};\mathsf{A1};\mathsf{A2};\mathsf{B2}$ ou $\mathsf{B1};\mathsf{B2};\mathsf{A1};\mathsf{A2}$ ou $\mathsf{B1};\mathsf{A1};\mathsf{B2};\mathsf{A2}$. On voit ici que le nombre d'entrelacements possibles est combinatoire.

De plus, sur les processeurs modernes, deux instructions successives localement indépendantes peuvent être permutées pour des raisons d'optimisation. En outre, la hiérarchie mémoire des architectures multi-cœurs peut produire des comportements similaires à la permutation d'instructions. Il faudra donc considérer également les cas où $\mathsf{B2}$ est effectuée avant $\mathsf{B1}$. On constate toutefois que ce programme n'est pas bien synchronisé : un des processus écrit dans une variable globale tandis que l'autre la lit mais sans que la lecture et l'écriture soient ordonnées par une synchronisation. Afin de pouvoir considérer que le comportement d'un programme parallèle de ce type est l'entrelacement des comportements de A et B, le programme parallèle doit être bien synchronisé. Il existe des analyses pour le vérifier [48].

Le raisonnement sur les programmes parallèles avec mémoire partagée est rendu ardu par les interférences entre les processus légers qui s'exécutent simultanément. Owicki et Gries [116] introduisent le principe de non-interférence. Toutefois, dans ce cadre, il faut vérifier que les assertions sur un processus léger considéré seul sont conservées pour chaque autre processus léger avec lequel il est exécuté en parallèle.

La notion de *Rely/Guarantee* [93] offre une bonne solution pour décrire les interférences entre processus. En plus de la pré-condition, il faut exprimer un invariant sur l'environnement nécessaire à la préservation de la correction (*rely*). En plus de la post-condition, on prouve que le processus préserve un invariant sur l'environnement (*guaran-*

tee). Ce formalisme nécessite cependant d'exprimer des propriétés sur l'ensemble de l'environnement, ce qui alourdit notablement les preuves. La logique de séparation concurrente [115] permet de raisonner localement si les processus travaillent sur des parties distinctes du tas. Si les processus travaillent sur une même zone mémoire, un invariant doit spécifier l'interaction des processus. Il est difficile d'exprimer les interférences dans ce cas. Il y a donc une activité importante autour des recherches visant à combiner les avantages de ces deux approches [138].

Les travaux précédents sont en fait une simplification des programmes par processus légers réels puisqu'ils considèrent des programmes avec composition parallèle, ce qui introduit une certaine structure. De plus, ils considèrent un nombre borné de verrous et de processus, pré-alloués. Pour un programme avec Pthreads [71] ou Java avec processus légers [75], il est nécessaire de pouvoir gérer des verrous alloués dynamiquement sur le tas et un nombre quelconque de processus pouvant être créés et détruits dynamiquement.

Vérification de programmes MPI

À notre connaissance, la vérification de programmes MPI est réalisée pour le moment à l'aide de techniques de vérification de modèles.

Le projet Gauss a d'abord considéré la modélisation d'un petit sous-ensemble de MPI et la vérification de petits exemples caractéristiques de problèmes rencontrés en programmation MPI [69]. La vérification a été faite à l'aide notamment de SPIN [87]. Des sous-ensembles plus importants de MPI ont été par la suite modélisés. Dans ces travaux, les modèles sont soit écrits par l'utilisateur, soit générés à partir des programmes. La vérification *in-situ* a également été expérimentée [139]. Il s'agit de vérifier directement des programmes qui sont instrumentés par le biais d'une bibliothèque MPI instrumentée : un ordonnanceur enregistre la trace d'une exécution et contrôle si à certains points un autre entrelacement aurait dû être considéré. Si tel est le cas, l'exécution du programme est relancée pour obtenir une autre trace et ce jusqu'à ce que tous les entrelacements non équivalents soient couverts. La principale propriété vérifiée est l'absence d'interblocage. Une réduction d'ordre partiel dynamique a été conçue lorsqu'il s'est avéré que les réductions d'ordre partiel classiques (qui aident à réduire l'explosion combinatoire du nombre d'états des modèles à vérifier) ne sont pas adaptées [145].

Une autre équipe a considéré des sous-ensembles différents de MPI [121, 122]. Pour la vérification, une extension du langage de spécification de SPIN [120] plus adaptée pour écrire des modèles de programmes MPI est proposée. La vérification est faite avec SPIN mais les utilisateurs doivent écrire les modèles eux-mêmes.

D'autres approches, moins répandues, proposent des abstractions pour faciliter la programmation parallèle. En limitant le spectre des programmes parallèles exprimables, elles offrent au programmeur certaines garanties quant à leurs comportements. Nous présentons dans la section suivante certaines de ces approches de *parallélisme structuré*.

1.3 Approches de parallélisme structuré

Plusieurs formes de parallélisme structuré ont été proposées dans les années 90 pour pallier les problèmes soulevés par la programmation des architectures parallèles. Parmi ceux-ci nous nous intéresserons plus particulièrement au parallélisme de données [7], au modèle de programmation parallèle quasi-synchrone [8] (BSP) et aux squelettes algorithmiques. Les frontières entre ces modèles ne sont pas clairement définies ; il est tout à fait possible d'imaginer un ensemble de squelettes algorithmiques data-parallèles implantés suivant le modèle BSP. Nous essayons ici de faire ressortir les particularités de chacune d'elles. Nous revenons plus en détail sur ces trois approches au chapitre 2.

1.3.1 Parallélisme de données

Issu des langages impératifs, le parallélisme de données a pour but, comme son nom l'indique, de traiter des données en parallèle. Pour cerner la définition du parallélisme de données, on peut le mettre en opposition au parallélisme de tâches où des tâches différentes sont exécutées en même temps. Ici, le programme est écrit comme un programme séquentiel et la parallélisation se fait par la manipulation d'une structure de données répartie (généralement un tableau) à l'aide de primitives parallèles. On peut voir le parallélisme de données comme une généralisation du parallélisme SIMD au niveau du modèle de programmation, le modèle d'exécution d'un programme data-parallèle étant beaucoup plus asynchrone.

La structure de données peut être explicitement parallèle, et distinguée syntaxiquement des structures séquentielles, comme les *shape* de C* [137], ou être une structure du langage implicitement parallélisée comme les matrices en *High Performance Fortran* [95].

La création des processus et l'accès concurrent aux données sont entièrement à la charge du compilateur qui met en place les schémas de communication entre processus en assurant l'absence d'inter-blocage et de non-déterminisme. La répartition des structures de données est effectuée par le compilateur ; cependant, elle peut (et généralement, pour obtenir des performances raisonnables, doit) être guidée par le programmeur.

1.3.2 Parallélisme quasi-synchrone

Le modèle de programmation parallèle quasi-synchrone introduit par Valiant et affiné depuis [140, 105, 125, 16], est un modèle structuré de programmation parallèle qui vise à maximiser la portabilité des performances en ajoutant une notion de processus explicites au parallélisme de données. Il assure l'absence d'inter-blocage et limite fortement les possibilités de non-déterminisme des programmes.

Dans ce modèle, la machine parallèle est vue comme un ensemble homogène d'unités de calcul, reliées entre elles par un réseau de communication et une unité de synchronisation globale. Ce modèle peut être appliqué à la plupart des machines parallèles existantes,

7. En anglais *data-parallelism*
8. En anglais *Bulk Synchronous Parallelism*

d'une petite machine parallèle à mémoire partagée aux très grands centres de calculs. La synchronisation entre processeurs est généralement implantée au niveau logiciel.

Un programme BSP est une *séquence* de super-étapes, chaque super-étape étant composée d'une phase de calcul asynchrone, d'une phase de communications et d'une phase de synchronisation qui garantit que les données échangées sont arrivée à destination. Le modèle BSP propose un modèle de performances (dit aussi de coûts) qui est réaliste mais reste simple du fait de la structure contrainte des super-étapes.

Le modèle BSP a été utilisé avec succès pour une large variété de problèmes : le calcul scientifique [17, 88, 16], les algorithmes et la programmation génétiques [28, 57], les réseaux de neurones [117], les bases de données parallèles [8, 7], les solveurs de contraintes [72], la vérification [77, 64], *etc.*

1.3.3 Squelettes algorithmiques

Partant du constat que les langages *universels* ne permettaient pas d'avoir à la fois des programmes portables sur diverses architectures et de conserver des performances optimales, Cole a proposé de structurer la programmation parallèle à l'aide de squelettes algorithmiques [41].

Issus des approches de programmation fonctionnelle, les squelettes algorithmiques sont des fonctions d'ordre supérieur implantées en parallèle et fournies au programmeur. Ces fonctions permettent d'encapsuler les détails du parallélisme en proposant au programmeur des primitives correspondant à des motifs de calcul couramment utilisés, que ce soit pour le calcul sur des structures de données ou pour l'agencement de tâches de calcul. Différentes implantations d'un même squelette peuvent exister, permettant d'obtenir des performances optimales pour telle ou telle architecture.

Les squelettes offrant un parallélisme de données encaspulent une structure de données répartie et des primitives de manipulation de celles-ci. On est ici très proche de la définition donnée pour le paradigme du parallélisme de données. Bien que les squelettes algorithmiques classiques se restreignent souvent à des structures de données classiques telles que des collections ou des matrices, ils peuvent également servir pour la manipulation de structures moins *simples* à paralléliser telles que les arbres. Enfin, les approches par squelettes fournissent parfois une combinaison de squelettes pour le parallélisme de données et pour le parallélisme de tâches.

1.4 Contribution et structure du document

Notre travail a pour objectif de fournir un cadre pour le développement systématique de programmes parallèles corrects. Ces programmes sont écrits dans un langage fonctionnel parallèle quasi synchrone, *Bulk Synchronous Parallel ML* (ou BSML), qui permet de manipuler une structure de données répartie, les vecteurs parallèles. Langage fonctionnel d'ordre supérieur, BSML est un langage de choix pour l'implantation de squelettes algorithmiques data-parallèles. Il hérite également de l'absence d'inter-blocage

du modèle quasi-synchrone et son modèle de communication assure le déterminisme des programmes.

Pour développer ces programmes parallèles corrects, nous utilisons l'expressivité du langage de l'assistant de preuve Coq. La proximité de ce langage avec le langage OCaml permet d'intégrer naturellement un plongement superficiel du langage BSML. Les programmes sont donc écrits dans le langage de Coq, augmenté des primitives de BSML. Deux méthodes sont alors proposées pour obtenir un programme correct :

a posteriori après avoir écrit le programme parallèle, les propriétés sont exprimées sur celui-ci, et peuvent être prouvées en utilisant l'ensemble de la mécanique de Coq.

par dérivation une spécification est écrite sous la forme d'un programme séquentiel simple, ce programme peut ensuite être transformé pour correspondre à un squelette algorithmique (ou une composition de squelettes) dont il existe une parallélisation montrée correcte.

Une fois les programmes parallèles développés et montrés corrects, ils peuvent être extraits, puis compilés par le compilateur BSML, et enfin exécutés sur une grande variété d'architectures parallèles, allant d'une petite machine munie d'un processeur multi-cœur à un grand centre de calcul.

Le chapitre 2 présente un panorama des travaux existants sur la programmation parallèle structurée, la preuve et la dérivation de programmes parallèles structurés.

Le chapitre 3 présente plus en détails les deux outils utilisés tout au long de ce document, l'assistant de preuve Coq et le langage BSML.

Le plongement superficiel du langage BSML dans le langage de l'assistant de preuve Coq, conçu de façon à permettre l'extraction de programmes BSML, est décrit dans le chapitre 4.

Dans le chapitre 5, nous formalisons une notion de *parallélisation correcte composable* nous permettant de spécifier les programmes parallèles sous la forme d'un programme séquentiel simple. Cette notion est également formalisée en Coq.

Le chapitre 6 présente des outils pour le développement systématique de programmes parallèles corrects à l'aide de squelettes algorithmiques. Ces outils, basés sur les fonctionnalités de Coq permettent d'une part d'assister le programmeur dans la tâche de transformation d'une spécification vers une forme efficace et parallélisable, d'autre part de construire automatiquement l'implantation parallèle d'une spécification écrite sous la forme d'une composition de fonctions dont une parallélisation a été montrée correcte.

Le chapitre 7 traite de l'homomorphisme BSP (ou BH). Parmi les squelettes algorithmiques que nous utilisons, le plus expressif est le squelette BH, initialement proposé par Hu et élaboré plus en détails par Gesbert [89]. Nous avons étudié différentes propriétés de ce squelette, notamment la classe des fonctions qu'il peut paralléliser. Son implantation parallèle a été améliorée, puis nous avons montré qu'il s'agissait d'une parallélisation correcte pour un grand sous-ensemble des fonctions de la classe discutée.

Des applications développées avec notre environnement sont présentées dans le chapitre 8 ainsi que des expériences menées sur plusieurs machines parallèles.

Enfin nous concluons et dégageons des perspectives pour de futurs travaux dans le chapitre 9.

Des versions antérieures des travaux présentés ont fait l'objet de publications dans des conférences internationales :

1. Louis Gesbert, Zhenjiang Hu, Frédéric Loulergue, Kiminori Matsuzaki, et Julien Tesson. Systematic Development of Correct Bulk Synchronous Parallel Programs. Dans *The 11th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 334–340. IEEE Computer Society, 2010.
2. Julien Tesson, Hideki Hashimoto, Zhenjiang Hu, Frédéric Loulergue, et Masato Takeichi. Program Calculation in Coq. Dans *Thirteenth International Conference on Algebraic Methodology And Software Technology (AMAST2010)*, LNCS 6486, pages 163–179. Springer, 2010.
3. Julien Tesson et Frédéric Loulergue. A Verified Bulk Synchronous Parallel ML Heat Diffusion Simulation. Dans *11th International Conference on Computational Science (ICCS 2011)*, Procedia Computer Science, pages 36–45. Elsevier, 2011.

2 Développement et vérification de programmes parallèles structurés

Sommaire

Tous les modèles de parallélisme structuré présentés en introduction reposent sur la séparation entre le modèle d'exécution d'une part, et le modèle de programmation d'autre part. Dans [24, 25], Bougé propose que le modèle de programmation se base sur une *vue macroscopique* où le programme est vu comme une séquence d'étapes parallèles, tandis qu'à la compilation (dans le contexte de parallélisme de données), la vue microscopique doit être adoptée pour correspondre au modèle d'exécution où des processus effectuent en parallèle des séquences d'instructions. La vue macroscopique doit permettre au programmeur d'avoir une vision globale du programme parallèle. Bien que la présentation soit faite sous l'angle du parallélisme de données à grain fin utilisé en parallélisation automatique, elle correspond également aux autres approches de parallélisme structuré que nous avons présentées. Dans le modèle BSP, la séquence d'étapes parallèles correspond à la super étape, et le langage BSML donne une vision encore plus *macroscopique* par la vision globale qu'il offre du parallélisme en se démarquant du modèle SPMD. Les implantations actuelles de modèles BSP reposent, *in fine*, sur l'exécution de programmes

parallèles asynchrones communicants pour échanger des données et effectuer des barrières de synchronisation. La vision à deux niveaux se calque tout aussi bien sur les squelettes algorithmiques à parallélisme de données. Contrairement à la parallélisation automatique, le niveau microscopique est généralement traité au niveau de l'implantation du squelette et non de la compilation. Les squelettes à parallélisme de tâches ne sont plus exactement des séquences d'étapes parallèles ; la vue macroscopique reste cependant une vue globale d'où est orchestrée le parallélisme sans prendre en compte le modèle d'exécution.

2.1 Parallélisme de données

2.1.1 Sémantique des langages data-parallèles

Dans [26], les auteurs définissent un mini langage reprenant la structure courante des langages data-parallèles : un langage impératif dont les variables sont des vecteurs de données, l'application point-à-point d'opérations sur ceux-ci, une instruction Where permettant l'activation du calcul sur certaines localités uniquement, et une primitive de ré-arrangement des valeurs à l'intérieur d'un vecteur de données. La sémantique du langage est donnée au niveau macroscopique sous une forme dénotationnelle opérant sur un environnement (associant des valeurs aux vecteurs parallèles de données) et un *contexte d'activité* décrivant les processus actifs, non-masqués par une condition Where. Un calcul de pré-conditions les plus faibles est également défini pour un sous-ensemble du langage.

Dans [54], l'auteur étend un plongement hybride dans le langage de l'assistant de preuve Isabelle/HOL d'un sous ensemble du langage C par les primitives parallèles *Data Parallel C Extension*. Par plongement hybride, il faut comprendre que les structures de contrôle du langage sont représentées par un type de données (plongement profond) alors que les expressions sont représentées directement dans le langage d'Isabelle/HOL (plongement superficiel).

2.1.2 Représentation de la localité des données

Dans [78], les auteurs proposent un langage fonctionnel où le parallélisme de données est exprimé par des *data-fields*. Ces *data-fields* représentent la répartition des données par une fonction allant d'un domaine d'indices, à un domaine de données ; ils généralisent les *shapes* de C*. Cette notation élégante permet notamment d'exprimer l'application d'une fonction $f : A \to B$ en tout point d'un vecteur parallèle $\nu : I \to A$ par la composition $f \circ \nu : I \to B$. Les communications explicites sont exprimées par une transformation de l'ensemble des indices d'un *data-field*. Cette représentation ne permet toutefois pas de raisonner sur la correspondance entre l'indexation des *data-fields* et leur position en mémoire.

Violard généralise les *data-fields* par les *shaped data-fields* [142] qui permettent une représentation de la localité des valeurs prenant en compte les possibilités de duplication de valeurs pour un même indice, afin d'éviter les communications d'un processus à

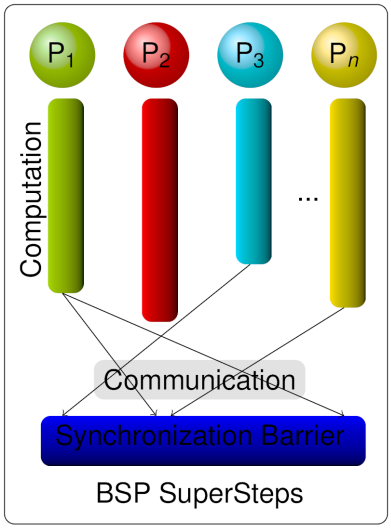


FIGURE 2.1 – *Super-étape BSP*

l'autre. Sa modélisation permet un découplage dans la représentation entre l'indexation des données (*data-fields*), utilisée par le programme, et la ou les localités effectives des données indexées (*shape*). Il offre ainsi une formalisation permettant de modifier la gestion de l'accès aux données réparties (réplications ou communications) en préservant la sémantique du programme.

2.1.3 Raffinement de programmes

PEI [143] est un formalisme déclaratif où les programmes sont exprimés sous forme d'équations entre des objets data-parallèles. Ces équations décrivent des relations et ne forment donc pas toujours un programme. Le formalisme PEI dispose d'un calcul formel pour le raffinement de programmes. Un système d'équations peut être transformé en un autre de deux façons : le nouveau système peut être plus spécifique du point de vue des solutions possibles, permettant ainsi de rapprocher la relation définie d'une fonction calculable ; ou bien, le nouveau système peut spécifier plus précisément la répartition des données ou la modifier. Un programme data-parallèle peut ainsi être produit par raffinements successifs, chaque étape de raffinement pouvant traiter soit du résultat calculé, soit du parallélisme utilisé pour le calcul.

2.2 PARALLÉLISME QUASI-SYNCHRONE

2.2.1 Le modèle

Un programme parallèle quasi-synchrone est une suite de super-étapes composées chacune de trois phases (représentées par la figure 2.1) : une phase de calcul asynchrone, où chaque unité effectue un calcul séquentiel et demande des échanges de données ; une

phase de communication durant laquelle les données sont échangées entre les unités ; enfin une phase de synchronisation à laquelle doivent participer toutes les unités. Les données communiquées ne sont disponibles qu'une fois la barrière de synchronisation effectuée. Les synchronisations concernant toujours l'ensemble des unités, il ne peut pas y avoir d'inter-blocage.

La machine parallèle est décrite par trois paramètres : p le nombre d'unités de calcul ; g un indicateur de la bande passante pour la communication entre les unités de calcul, généralement exprimé en secondes par mot, et L le nombre de cycles nécessaires pour synchroniser l'ensemble des processeurs. Ce dernier paramètre est souvent fonction du nombre de processeurs.

A l'aide de ces paramètres, il est possible de calculer le coût d'exécution d'un programme parallèle quasi-synchrone. Le coût d'une super-étape est la somme des coûts de ses trois phases :

$$Max_{i=0}^{p-1} w_i + g \times Max_{i=0}^{p-1} Max(h_i^+, h_i^-) + L$$

Le coût de la phase de calcul asynchrone est le maximum parmi les temps de travail locaux w_i des processeurs. Le coût des communications est le maximum des coûts de communication de chaque processeur. Le coût de communication pour un processeur i donné est le maximum entre la taille des données reçues par le processeur h_i^- et la taille de celles envoyées h_i^+, multiplié par la bande passante g. La phase de synchronisation a un coût constant L.

2.2.2 Langages et bibliothèques

Plusieurs bibliothèques de programmation SPMD suivant le modèle quasi-synchrone sont disponibles pour le langage de programmation Java [74] ; et pour le langage C : BSPlib [84], PUB [22] (Paderborn University BSP), BSPonMPI [132] et xBSP basé sur l'architecture de réseau Virtuel Interface Architecture (VIA) [94]. Ces bibliothèques fournissent des primitives de communication par passage de messages ou par accès "directs" aux mémoires distantes (qui doivent être déclarées *globales*) ; une primitive de synchronisation ; ainsi qu'une primitive donnant le numéro de processeur sur lequel elle est exécutée. Les échanges de messages ou les lectures/écritures des mémoires distantes ne sont effectifs qu'une fois passée la barrière de synchronisation.

Le langage Bulk Synchronous Parallel ML (BSML) [100] adapte le modèle de parallélisme quasi-synchrone au paradigme de la programmation fonctionnelle ; il offre une vue globale du programme parallèle et rend explicite les processus par la manipulation de vecteurs parallèles, évitant ainsi les écueils rencontrés par le paradigme SPMD. Une présentation détaillée du langage BSML est donnée à la section 3.2. Ce style de programmation a été repris pour des implantations de BSP dans les langages Python [85], C++ [58], et Haskell [108].

2.2.3 Sémantique, construction et vérification de programmes impératifs

Dans [130, 129], les auteurs donnent une sémantique axiomatique pour la composition parallèle de processus BSP. Une description axiomatique à la *Owicki - Gries* est donnée pour la composition parallèle de processus par une super-étape. Ici, un processus BSP ne peut opérer que sur des variables locales et communiquer avec les autres processus. La communication est modélisée par la construction d'un ensemble de messages. Une super-étape est représentée par l'état courant des variables et un ensemble de *messages*, chaque message associant une valeur à une variable (variable qui peut appartenir à un autre processus). La sémantique d'une composition parallèle est donnée par l'union des états et l'union des messages. La propriété de non-interférence entre les processus est assurée simplement en interdisant le partage de variables entre processus ; le seul moyen d'interaction avec les variables des autres processus étant les messages. Une barrière de synchronisation applique les substitutions décrites par les messages à l'union des états. Le cas non-déterministe où plusieurs messages correspondent à une même variable est considéré par une règle imposant que la post-condition soit vérifiée quel que soit le message utilisé. Cette axiomatisation adopte un point de vue macroscopique, le programme étant représenté par une succession de super-étapes.

Une sémantique est également donnée du point de vue microscopique (composition parallèle de processus BSP pouvant demander des synchronisations) en utilisant le principe de *Rely/Guarantee*. Les interactions avec l'environnement sont restreintes aux barrières de synchronisation. Une variable auxiliaire sert de compteur de barrières de synchronisation ; la composition de processus qui n'ont pas le même nombre de synchronisations est considérée incorrecte car le modèle BSP impose que tous les processus participent à chaque barrière de synchronisation.

Dans [131], la sémantique macroscopique est étendue pour permettre la désynchronisation de sous-groupes de processus. En effet, si l'on peut montrer statiquement que deux sous-groupes n'échangent aucune donnée lors d'une barrière de synchronisation, il n'est pas nécessaire que la synchronisation ait lieu globalement. Cette désynchronisation de sous-groupes de processus permet des gains de performance. Cette sémantique ouvre la voie à une optimisation sûre de programmes BSP.

Dans [38], Chen et Sanders proposent des lois de raffinement de spécifications LOGS vers des programmes BSP ; ces lois sont utilisées dans [37] pour construire des programmes BSPlib-C. Dans [128], une sémantique et des lois de raffinement permettent de construire des programmes BSP à *synchronisations partitionnées* (certaines synchronisations ne se font que pour des sous-groupes de processeurs).

Dans [61], les auteurs proposent BSP-Why, un langage BSP SPMD proche du langage Why du générateur d'obligations de preuves de même nom. Des annotations spécifiques sont introduites pour prendre en compte le parallélisme dans les spécifications. Les programmes BSP-Why et leurs annotations sont transformés dans le langage Why par une simulation séquentielle des super-étapes parallèles. Une sémantique opérationnelle du langage BSP-Why permet de montrer la correction de cette transformation.

2.3 Squelettes algorithmiques

De nombreuses bibliothèques et langages reposant sur le modèle de programmation par squelettes algorithmiques existent, un panorama récent de ceux-ci est dressé dans [68].

Nous commençons par un panorama de travaux sur l'algorithmique constructive pour le parallélisme puis décrivons quelques bibliothèques et langages de squelettes algorithmiques disposant d'une sémantique formelle ou qui sont utilisés en conjonction avec des travaux sur de la dérivation de programmes.

2.3.1 Méthodes constructives pour le parallélisme

Les travaux de Skillicorn [123, 124] ont mis en évidence l'intérêt des homomorphismes pour la conception de programmes parallèles. Les homomorphismes sont des fonctions dont le schéma de récursion se conforme aux constructeurs d'une structure de données. Si les constructeurs d'une structure de données modélisent bien sa répartition, les homomorphismes pour cette structure pourront être parallélisés de façon systématique.

Pour représenter la distribution des listes, il suffit de les considérer sous la forme de *join-list*, c'est à dire de listes construites par concaténation, dont les constructeurs sont :

$$\begin{cases} [\,] : \mathsf{list}\ A & \text{liste vide} \\ [.] : A \to \mathsf{list}\ A & \text{singleton} \\ (+\!\!+) : \mathsf{list}\ A \to \mathsf{list}\ A \to \mathsf{list}\ A & \text{concaténation.} \end{cases}$$

Cette représentation met en évidence la construction d'une liste à partir de sous-listes (et donc représente dans sa construction la possible répartition).

Un homomorphisme de *join-list* h est une fonction définie comme suit :

$$\begin{aligned} h\ [\,] &= \iota && \text{(Homo-nil)} \\ h\ [a] &= f\ a && \text{(Homo-Singleton)} \\ h\ (x +\!\!+ y) &= (h\ x)\ \odot\ (h\ y) && \text{(Homo-Concat)} \end{aligned}$$

Un tel homomorphisme, également appelé catamorphisme dans le cadre BMF, est défini uniquement par les éléments ι, f et $\odot$. il sera noté $(\!|\iota, f, \odot|\!)$. Comme la liste vide est neutre pour la concaténation, on a $h\ x = h\ (x +\!\!+ [\,]) = h\ (x) \odot h\ ([\,]) = h\ ([\,] +\!\!+ x) = h\ ([\,]) \odot h\ (x)$. ι est donc l'élément neutre associé à l'opérateur $\odot$, il est de ce fait souvent omis dans la notation des catamorphismes. Une autre propriété importante des homomorphismes de listes [1] est que l'opérateur $\odot$ doit être associatif (car la concaténation de liste est associative).

L'intérêt de tels homomorphismes pour la programmation parallèle est mis en avant par l'utilisation du premier théorème d'homomorphisme [13] qui énonce que tout homomorphisme $(\!|\iota, f, \odot|\!)$ peut être calculé en composant la réduction par $\odot$ à une application

1. dans le reste du document le terme *homomorphisme de listes* ou simplement *homomorphisme* fera toujours référence à un *homomorphisme de join-list*

de ($\mathtt{map}\ \mathtt{f}$). Autrement dit, l'homomorphisme est calculé sur une liste $\mathtt{l}$ par ($\mathit{fold\_left}$ $\odot$ ($\mathtt{map}\ \mathtt{f}\ \mathtt{l}$) ι). Cette décomposition montre le parallélisme intrinsèque des homomorphismes : la fonction *map* est purement parallèle et la réduction, comme l'opérateur $\odot$ est associatif, peut être effectuée localement pour chaque sous-liste, les résultats obtenus sur chaque processeur sont ensuite communiqués et combinés par l'opérateur $\odot$.

Cependant l'expressivité des homomorphismes de listes reste limitée et nombre de fonctions ne sont pas exprimables sous cette forme. Cole a montré dans [42] comment profiter de la parallélisation des homomorphismes tout en conservant une bonne expressivité en construisant des quasi-homomorphismes [2]. Un quasi-homomorphisme est une fonction construite par composition d'une fonction quelconque $\mathtt{f}$ avec un homomorphisme $\mathtt{h}$. Toute fonction $\mathtt{g}$ peut potentiellement s'écrire sous forme d'un quasi-homomorphisme : il suffit en effet de prendre l'homomorphisme $\mathtt{h} = (\!|[\,], [.], +\!|)$, c'est-à-dire l'identité sur les listes, et $\mathtt{f} = \mathtt{g}$. Cependant une telle construction n'apporte rien à la parallélisation de la fonction ; pour que la décomposition en quasi-homomorphisme soit efficace il faut que la fonction $\mathtt{f}$ effectue le minimum de *travail*. Dans les exemples pertinents, cette fonction $\mathtt{f}$ est généralement une projection. Dans [70], Gorlatch introduit les *homomorphismes distribués*, une adaptation de la caractérisation des homomorphismes au calcul dans le modèle de parallélisme hypercube. Hu [89] a proposé les *homomorphismes BSP*, ou BH, adaptés au modèle de parallélisme quasi synchrone. Gesbert [65] a proposé une implantation BSML d'un squelette algorithmique permettant calculer ces homomorphismes. Il a également modélisé cette implantation en Coq à l'aide du plongement superficiel de BSML dans la logique de l'assistant de preuve Coq et s'est servi de cette modélisation pour prouver en partie sa correction. Nous revenons plus en détails sur cet homomorphisme, son implantation et sa modélisation au chapitre 7.

2.3.2 Langages et bibliothèques

La bibliothèque de squelettes algorithmiques pour C++, dont la dernière version est méta-programmée, SkeTo [104, 102] est souvent utilisée pour l'implantation d'applications développées à l'aide de méthodes d'algorithmique constructive [103, 110, 109]. Cette bibliothèque offre des squelettes algorithmiques data-parallèles sur des tableaux (de taille fixe), des matrices et des arbres. OSL [91] partage avec SkeTo des squelettes sur les tableaux mais permet que ceux-ci ne soient pas uniformément répartis (ce qui permet par exemple de programmer un tri par échantillonnage régulier). Des squelettes supplémentaires sont fournis pour permettre ceci. OSL possède un modèle de programmation mécanisé en Coq [90] qui a été utilisé pour la preuve de correction d'une version OSL de la diffusion de chaleur [92].

Lithium [51] est une bibliothèque de squelettes algorithmiques pour le langage Java. Cette bibliothèque dispose d'une sémantique formelle qui offre simultanément un modèle de programmation et un modèle d'exécution [2]. Calcium et Skandium [97, 98] sont également des bibliothèques de squelettes pour le langage Java. Une sémantique opérationnelle et un système de type pour Calcium est présentée dans [32]. L'implantation

2. *near-homomorphism* en anglais

de Calcium est basée sur la bibliothèque ProActive [6] pour laquelle il existe un calcul formel des primitives pour le parallélisme et la répartition [31, 33]. Ce calcul a été mécanisé partiellement en Isabelle [80, 81]. Il n'existe toutefois pas à notre connaissance une modélisation de l'implantation des squelettes algorithmiques de Calcium à l'aide de cette formalisation en Isabelle, ni de preuve de dérivation ou d'extraction de programmes pour Calcium.

Le langage P3L [50, 3] dispose d'une sémantique formelle [34] sous forme d'*Abstract State Machine* [23]. Il existe une version de P3L pour OCaml nommée OCamlP3L [45]. Le calcul de tableaux multi-dimensionnels [55] est la base d'une future version d'OCamlP3L qui n'a pas encore été implantée.

Comme ProActive/Calcium et BSML/BH, Eden [99] est un langage de programmation parallèle fonctionnel qui est utilisé régulièrement pour implanter des squelettes algorithmiques [10]. Eden a une sémantique formelle [83] mais à notre connaissance non mécanisée et jamais utilisée pour prouver la correction de programmes Eden, et en particulier de squelettes.

Dans [62], Fradet et Mallet proposent un compilateur pour un langage fonctionnel du premier ordre, sans récursion, augmenté d'un ensemble de squelettes data-parallèles manipulant des vecteurs de données. Les programmes sont transformés vers une implantation en C+MPI. À la compilation, des implantations utilisant différents schémas réguliers de répartition des vecteurs sont générées. Les restrictions fortes sur le langage permettent d'obtenir statiquement un coût symbolique pour chacune de ces implantations. Les coûts ainsi obtenus peuvent être utilisés pour sélectionner une implantation plus efficace ou, lorsque le choix de l'implantation adéquat dépend d'informations disponibles uniquement à l'exécution, les coûts pré-calculés peuvent être utilisés dynamiquement pour le choix d'une implantation parmi plusieurs.

2.4 Discussion

Notre but est de permettre, dans un même environnement, de prouver la correction des résultats de programmes parallèles avec communications explicites, et de dériver des programmes parallèles corrects à l'aide de squelettes algorithmiques. Notre environnement permet ainsi de travailler à différents niveaux de détails, offrant à la fois une grande expressivité et un haut niveau d'abstraction vis-à-vis du parallélisme. Les travaux présentés précédemment font ici l'objet d'une discussion dans ce contexte.

Les sémantiques comme celle présentée dans [26], restreinte à un mini-langage dont il n'existe pas d'implantation, et non mécanisée, ne peuvent être utilisées que pour valider les grandes lignes d'implantation d'un programme, mais difficilement pour prouver complètement des programmes parallèles. De même, lorsqu'elles sont limitées à la sémantique de squelettes algorithmiques, mais ne permettent pas de raisonner sur les parties de code inséré, les sémantiques peuvent être (et sont) utilisées pour montrer des propriétés du langage de squelettes (telle que l'auto-réduction pour Calcium) ou la correction d'un ensemble fixé de transformations de squelettes (Lithium, compilateur de squelettes

de Fradet et Mallet), mais elles ne sont pas utilisables pour prouver la correction du résultat d'un programme. Les sémantiques des langages Eden et ProActive n'ont pas été utilisées pour faire des preuves de programme. La sémantique d'Eden n'est pas mécanisée, un sous ensemble de la sémantique de ProActive l'est depuis récemment mais il n'a été utilisé que pour montrer des propriétés du langage (abscence de *live-lock* dans le cas d'objets sans état).

Le plongement du langage Data Parallel C dans HOL permet *a priori* de prouver la correction de programmes parallèles, mais aucun exemple n'est disponible. Le formalisme PEI permet la construction de programmes corrects par raffinement. Ces deux formalismes sont dédiés à du parallélisme de données pour lequel les communications ne sont pas explicites contrairement à notre approche.

Les implantations SPMD du modèle BSP souffrent d'une absence de vision globale du programme parallèle, ce manque se retrouve dans les formalismes proposant des sémantiques pour la composition parallèle de processus, ainsi le raffinement de spécifications LOGS vers des programmes C-BSPlib produit des programmes pour un nombre fixé de processeurs. Cette méthode, de l'aveu même de l'auteur, a peu de chance de bien passer à l'échelle pour un grand nombre de processeurs.

BSP-Why permet de prouver des propriétés sur les programmes BSP indépendamment du nombre de processeurs, de façon automatisée. Le langage BSP-Why n'est cependant pas exécutable en parallèle et il n'existe pas pour le moment d'outil permettant de faire automatiquement le lien entre un programme BSP-Why et une implantation impérative SPMD (modèle utilisé par BSP-Why) effectivement exécutable en parallèle. De plus, le paradigme de génération d'obligations de preuve utilisé par BSP-Why ne permet pas la dérivation de programmes.

Les approches utilisant l'algorithmique constructive comme SkeTo reposent généralement sur des dérivations formelles pour définir de nouveaux squelettes ; cependant, ces dérivations ne sont pas mécanisées, et les implantations de ces squelettes dans des langages impératifs sont faiblement liées aux dérivations formelles qui ont donné naissance aux algorithmes.

Si l'on veut pouvoir construire des programmes parallèles avec un haut niveau de confiance dans leur résultat, il faut un environnement mécanisé permettant de définir de nouveaux programmes parallèles avec communications explicites et de prouver leur correction. Mais il faut également que cet environnement soit enrichi de squelettes algorithmiques ayant une sémantique fonctionnelle simple. De tels squelettes permettent de développer rapidement de nouveaux programmes parallèles et de mutualiser les efforts à fournir pour la preuve de correction de ceux-ci. De plus les programmes définis dans cet environnement doivent être directement exécutables : idéalement, ils doivent être automatiquement transformés en un code binaire de même sémantique par une chaîne entièrement certifiée. Au minimum la chaîne doit être composée d'éléments de confiance nous permettant de penser qu'aucune erreur ne sera introduite. C'est ce que nous proposons : une fois écrit un programme dans notre environnement, sa transformation en exécutable binaire ne passe que par trois maillons potentiellement faibles : l'extraction de Coq vers Ocaml, un peu de code permettant de lier le code extrait à l'implantation de

BSML, l'implantation de BSML elle-même, et le compilateur OCaml. Des travaux sont en cours sur la vérification de l'extraction [67], et il existe un compilateur certifié pour un sous-ensemble d'OCaml [52, 53].

Préliminaires

3

Sommaire

Dans ce chapitre, nous présentons l'assistant de preuve Coq et le langage de programmation parallèle BSML. Nous supposons que le lecteur est familier avec la programmation fonctionnelle. Les deux langages utilisés ayant une syntaxe à la ML, une familiarisation avec le langage OCaml facilitera la lecture du document. Le lecteur peut se référer à [46, 35].

3.1 Présentation de l'assistant de preuve Coq

Un assistant de preuve est un logiciel permettant d'exprimer, dans un langage plus ou moins expressif, des propositions logiques et de vérifier leur validité par une série d'interactions avec le logiciel.

Exprimer des propositions logiques complexes nécessite une logique expressive, or vérifier la validité des propositions exprimées dans une telle logique est souvent complexe. Les assistants de preuve sont utilisés lorsqu'une telle vérification ne peut se faire automatiquement. Contrairement à un prouveur automatique, l'assistant de preuve requiert l'intervention d'un humain pour construire la preuve de correction d'une proposition. Le

logiciel ne vérifie plus la validité d'une proposition logique elle-même mais la correction de la preuve construite par l'utilisateur. Il est en effet plus simple de s'assurer, par exemple, que tous les cas d'une preuve par induction ont été traités correctement, que de décider, pour résoudre un problème, de raisonner par induction.

Tout l'intérêt de l'assistant de preuve est de profiter de la créativité et de l'intelligence humaine pour la conception de preuve, et de la mécanique simple et rigoureuse d'un programme pour vérifier que tous les cas de figures ont été envisagés et que chaque étape du raisonnement est exempte de faille. Bien sûr, plus la mécanique de l'assistant de preuve est « intelligente », plus il est en mesure de construire des preuves complexes concernant les détails d'une étape de raisonnement, et plus l'être humain peut se consacrer à la production de preuves de haut niveau. Cependant, on sait aussi que plus la mécanique d'un logiciel est complexe et plus il est difficile d'avoir confiance en son comportement. Les assistants de preuve satisfaisant le critère de De Bruijn pallient ce problème en produisant des certificats vérifiables par un noyau relativement simple, en lequel il est possible d'avoir confiance. Les étapes de construction de la preuve peuvent ainsi utiliser différentes tactiques ou extension du langages non-sûr, efficaces mais susceptible d'introduire des erreurs dans les preuves. Les termes de preuve construits seront de toute façon, *in fine*, vérifiés par le noyau sûr du logiciel. L'assistant de preuve Coq satisfait ce critère.

Coq est basé sur le principe de la correspondance preuve-programme [1] qui établit une relation entre les preuves mathématiques formelles et les langages de programmation fonctionnelle.

Une proposition formelle mathématique correspond au type d'une fonction, sa preuve est une fonction prenant en paramètre des objets dont le type correspond aux prémisses de la proposition et retournant un objet typé par la conclusion. Suivant le calcul choisi, on obtiendra différentes logiques. Ainsi Howard a remarqué que les démonstrations de déduction naturelle intuitionniste peuvent être vues comme des termes du lambda-calcul simplement typé.

L'assistant de preuve Coq initialement basé sur le calcul des constructions utilise maintenant le calcul des constructions (co-)inductives. Nous ne présenterons pas ici l'intégralité du langage mais nous noterons qu'il s'agit d'un λ-calcul typé d'ordre supérieur avec types inductifs et co-inductifs. Chaque terme a un type et les types eux-mêmes sont des termes du calcul (ce sont des habitants de première classe du langage). Ainsi il est possible de définir des termes dépendants de types (polymorphisme), des types dépendants de types (opérateurs sur les types) et des types dépendants de termes (types dépendants). Le langage de spécification dans lequel sont écrits les termes en Coq s'appelle Gallina, les commandes permettant de contrôler le comportement de l'assistant de preuve font partie du langage Vernacular. Coq propose aussi un langage de tactiques, `Ltac` permettant de construire interactivement un terme de preuve. Par abus de langage nous parlerons souvent de Coq pour désigner l'un ou l'autre de ces langages ou encore le logiciel lui-même.

Bien qu'une preuve et un programme soient la même chose en Coq, construire l'une

1. Aussi appelée correspondance de Curry-Howard ou isomorphisme de Curry-Howard, voir [43] pour un historique des travaux sur cette correspondance et de leur influence sur les systèmes de preuve

ou écrire l'autre sont souvent deux activités bien différentes dans l'intention : lorsque nous voulons écrire une preuve, le calcul effectué par le terme produit nous importe peu, seul son type nous intéresse ; à l'inverse l'écriture d'un programme est peu guidée par le type de celui-ci mais beaucoup plus par la valeur calculée. Coq tient compte de cette différence d'intention en permettant une différenciation syntaxique et méthodologique entre la construction d'une preuve et l'écriture d'un programme. Cependant les termes obtenus sont identiques et il est tout à fait possible d'écrire un programme en utilisant la méthodologie destinée à construire des preuves et vice-versa. Nous présenterons ici les différentes approches proposée par Coq pour la construction de termes ainsi que certains mécanismes qui nous intéressent.

Cette présentation de Coq, faite pour que le lecteur dispose des éléments nécessaires à la compréhension de notre document, reste très partielle, le lecteur intéressé pourra utilement se référer au « Coq'Art »[11] pour une introduction plus complète à Coq, ainsi qu'au document « *Certified Programming with Dependent Types* » [39], explorant en profondeur l'utilisation des types dépendants de Coq, et également au manuel de référence de Coq.

Dans la suite de ce document, le code Coq est mis en valeur de deux façons différentes selon qu'il s'agit de présentations d'interactions avec l'assistant de preuve ou de définitions où l'interaction importe peu. Pour les interactions, les questions/réponses sont mises en forme comme suit :

```
Première ligne d'une définition
passée à l'assistant de preuve
et se terminant toujours par un point.
Réponse de la boucle interactive
qui est soit l'état d'une preuve
soit un simple message
```

Pour les parties de code où l'interaction importe moins, nous utiliserons deux mises en forme différentes, en fonction de la position du code dans le document (voir figure 3.1). De plus, quelques notations seront adoptées pour améliorer la lisibilité, ainsi `forall` et `exists` seront notés $\forall$ et $\exists$ et la négation de l'égalité écrite <> en Coq sera notée $\neq$.

3.1.1 Programmation en Coq

Pour écrire un programme l'utilisateur dispose d'une syntaxe à la OCaml. Comme son nom l'indique, le calcul des constructions inductives permet de construire des structures de données inductives. Les listes, par exemple, sont définies ainsi :

```
Inductive list (A :Type) :=
  nil : list A
| cons : A → list A → list A.
```

Une liste est soit une liste vide, obtenue par le constructeur nil, soit une liste construite à partir d'un élément et d'une sous-liste : cons *élément sous-liste*. Le type *list* est paramétré

Fonction du nom dans le code	mise en forme au fil du texte	mise en forme dans une définition séparée du texte
Mots clefs, Notations	`police Typewriter`	`police Typewriter rouge`
Inductifs, Enregistrements, Classes	police sans sérif	police sans sérif bleue
Définitions, Lemmes, Instances, Méthode	*police oblique*	*police oblique verte*
Constructeurs	police sans sérif	police sans sérif rouge foncée
Variables	*police oblique*	*police oblique violette*
Axiomes	*police oblique*	*police oblique verte foncée*
Modules	Petite majuscule	Petite majuscule verte

FIGURE 3.1 – *Mise en forme du code Coq dans le document*

par le type A des éléments de la liste, c'est un type polymorphe. Le polymorphisme en Coq étant explicite, les constructeurs cons et nil ont pour premier paramètre le type des éléments de la liste. Pour obtenir un comportement similaire au polymorphisme implicite des langages à la ML, il est possible de spécifier un paramètre comme étant implicite :

```
Implicit Arguments nil [A].
Implicit Arguments cons [A].
```

Il n'est maintenant plus nécessaire de fournir une valeur pour l'argument A, lorsque nous utilisons les constructeurs. Le système d'inférence de type tentera de deviner sa valeur à partir du contexte d'utilisation du constructeur. Il arrive cependant que le système ne puisse pas déduire cette valeur, il faut alors la fournir. Par exemple pour définir une liste d'entiers vide, si nous écrivons
`Definition l := nil.`

```
Coq < Toplevel input, characters 32-35:
> (* end hide*)  Definition l := nil.
>                                ^^^
Error: Cannot infer the implicit parameter A of nil.
```

Coq refuse de définir l car il ne sait pas quelle valeur utiliser pour le paramètre A, il faut donc spécifier le type en utilisant le symbole `@` qui rend explicite tous les paramètres du constructeur (ou de la fonction) qu'il préfixe.

```
Definition l := (@nil nat).
```

Pour calculer sur les listes, il est souvent nécessaire de définir des fonctions utilisant le filtrage par motif. Toutes les fonctions définies en Coq doivent être totales, tous les motifs possibles doivent donc être traités. La fonction *hd* qui renvoie la tête d'une liste doit donc prendre en paramètre une valeur à retourner au cas où la liste serait vide :

```
Definition hd (A :Type) (default : A) (l :list A) : A :=
```

```
match l with
  | nil ⇒ default
  | cons x _ ⇒ x
end.
```

La contrainte de totalité des fonctions est inhérente à la relation entre fonction et preuve. Puisqu'une fonction est une preuve de son type, si on autorise la fonction à n'être définie que partiellement, cela reviendrait à accepter une preuve de théorème qui échouerait dans certains cas. Ce n'est évidemment pas acceptable dans un assistant de preuve.

La fonction *hd* est une fonction polymorphe, paramétrée par le type des éléments de la liste. Comme pour les constructeurs, son paramètre A peut être rendu implicite afin de ne pas avoir à le fournir à chaque utilisation de la fonction.

```
Implicit Arguments hd [A].
```

On pourra ainsi écrire (*hd 0 l*) au lieu de (*hd* nat *0 l*) pour une liste d'entiers *l*.

Il est aussi possible de définir des fonctions récursives, cependant seules les récursions structurelles sont possibles afin d'assurer la terminaison des programmes. Cette contrainte de terminaison est, comme la contrainte de totalité, nécessaire pour assurer la cohérence de la logique de Coq. Ainsi pour définir la fonction *map*, qui applique une fonction à tous les éléments d'une liste, on procédera comme suit :

```
Fixpoint map (A B : Type) (f : A → B) (l :list A) {struct l} : list B :=
  match l with
    | nil ⇒ nil
    | cons h t ⇒ cons (f h) (map A B f t)
  end.
```

{`struct` *l*} indique que la récursion se fait structurellement sur *l*. Le système s'assure de la terminaison de notre fonction en vérifiant que l'appel récursif se fait bien sur un sous-terme strict de *l*. Du fait de cette contrainte de terminaison, Coq n'est pas Turing-complet, c'est un langage primitif récursif d'ordre supérieur. L'utilisation de fonctions d'ordre supérieur permet toutefois une grande expressivité, et la possibilité de définir des fonctions itérant sur des structures coinductives telles que les flots permet même de définir des fonctions non terminantes sous certaines contraintes que nous n'aborderons pas ici. De plus, il est possible de définir des récursions non-structurelles utilisant une preuve de décroissance des arguments suivant un ordre bien fondé.

L'interaction avec l'assistant de preuve lors de l'écriture de programmes est relativement limitée : l'utilisateur donne une définition, et l'assistant de preuve lui assure qu'elle est correctement définie et bien typée ou la rejette avec une explication. Nous allons présenter quelques exemples d'interactions. Afin d'améliorer la lisibilité du programme la notation infixe `::` désignant le constructeur cons sera utilisée. Elle est introduite par la commande suivante :

```
Notation " a :: l" := cons a l.
```

L'utilisateur souhaite définir la fonction *fold_left* qui prend une fonction, un élément et une liste, et applique la fonction à l'élément et à la tête de liste, puis s'appelle récursivement sur le reste de la liste en prenant pour élément le résultat obtenu précédemment. Arrivée à la fin de la liste, la fonction renvoie l'élément pris en paramètre.

```
  Fixpoint fold_left (A B : Type)
       (f : A->B->A) (l : list B) (a : A) {struct l}:=
  match l with
  | nil => a
  | b :: t => fold_left A B f l (f a b)
  end.
Error:
Recursive definition of fold_left is ill-formed.
In environment
fold_left : forall A B : Type, (A -> B -> A) -> list B -> A -> A
A : Type
B : Type
f : A -> B -> A
l : list B
a : A
b : B
t : list B
Recursive call to fold_left has principal argument equal to
"l" instead of "t".
Recursive definition is:
"fun (A B : Type) (f : A -> B -> A) (l : list B) (a : A) =>
 match l with
 | nil => a
 | b :: _ => fold_left A B f l (f a b)
 end".
```

Notre utilisateur maladroit s'est trompé, il a passé l en paramètre lors de l'appel récursif au lieu de la sous-liste t. Coq refuse de définir une fonction qui ne termine pas, il rejette donc la définition. Pour la définition corrigée de sa fonction, notre utilisateur peut s'abstenir de préciser quelle structure est utilisée pour la récursion car, dans ce cas simple, Coq arrive à le déduire automatiquement.

```
Fixpoint fold_left (A B : Type) (f : A->B->A)  (l : list B) (a : A) :=
  match l with
  | nil => a
  | b :: t => fold_left A B f t (f a b)
  end.
fold_left is recursively defined (decreasing on 4th argument)
```

L'interaction avec le système se limite, lors de la définition de programme, à une vérification de la syntaxe des programmes et à une vérification de la correction de leur type, définition après définition. Il en va tout autrement lors de la construction de preuve.

3.1.2 Preuves en Coq

L'énoncé d'un théorème est en général précédé du mot clef `Theorem` ou `Lemma` ou `Definition`. La différence avec la définition d'un programme est qu'ici, seul le type est donné. Par exemple, pour définir un lemme nommé *imp_trans*, affirmant la transitivité de l'implication, nous écrirons :

`Lemma` *imp_trans* $(P\ Q\ R : \texttt{Prop}) : (P \to Q) \to (Q \to R) \to P \to R$.

Après avoir donné ce lemme à coq, voici sa réponse :

```
Coq < 1 subgoal

  P : Prop
  Q : Prop
  R : Prop
  ============================
   (P -> Q) -> (Q -> R) -> P -> R
```

Elle signifie que le contexte contient les propositions P, Q et R, et qu'il reste un but à prouver, notre lemme. Nous utilisons le mot clef `Proof` pour annoncer que nous commençons à construire une preuve à l'aide du langage de tactique `Ltac`, puis nous utilisons la tactique `intros` qui introduit les hypothèses dans le contexte en leur donnant un nom.

```
    Proof.

    intros H H' p.
1 subgoal

  P : Prop
  Q : Prop
  R : Prop
  H : P -> Q
  H' : Q -> R
  p : P
  ============================
   R
```

ensuite, nous appliquons l'hypothèse *H'* au but.

```
      apply H'.
1 subgoal

  P : Prop
  Q : Prop
  R : Prop
  H : P -> Q
  H' : Q -> R
  p : P
  ============================
   Q
```

La conclusion de l'hypothèse *H'* est unifiée à notre but, et l'assistant de preuve nous demande de prouver ses prémisses. Nous répétons l'opération avec l'hypothèse *H* :

```
      apply H.
1 subgoal

  P : Prop
  Q : Prop
  R : Prop
  H : P -> Q
  H' : Q -> R
  p : P
  ============================
   P
```

Nous arrivons à un but dont l'énoncé est présent dans les hypothèses, nous pouvons maintenant utiliser différentes tactiques pour finir la preuve : `exact` *p* qui utilise l'hypothèse *p*, ou `auto`, `trivial`, `assumption` qui trouvent automatiquement l'hypothèse *p* et l'utilisent.

```
      exact p.
Proof completed.
```

La preuve est terminée, pour sauvegarder le terme de preuve qui a été construit à l'aide des tactiques, nous utilisons la commande `Qed` qui provoque une vérification du type du terme. S'il est correct, le terme est enregistré sous le nom que nous lui avons donné.

```
    Qed.
imp_trans is defined
```

Une fois la preuve sauvegardée, il est possible d'en afficher le corps, on voit ici le programme qui a été progressivement construit par la série de tactiques, ce programme

n'est pas très utile en soi mais il permet de mieux comprendre le lien entre preuve et programme.

```
     Print imp_trans.
imp_trans =
fun (P Q R : Prop) (H : P -> Q) (H' : Q -> R) (p : P) => H' (H p)
     : forall P Q R : Prop, (P -> Q) -> (Q -> R) -> P -> R
Argument scopes are [type_scope type_scope type_scope _ _ _]
```

La tactique **apply** correspond à l'application de fonction, la tactique **exact** reproduit le terme donné en paramètre. Le terme de preuve est donc l'application successive des fonctions H et H' à l'élément p.

Nous aurions pu également utiliser la commande **Defined** au lieu de **Qed**. L'effet immédiat est le même : typage et enregistrement du terme, la seule différence entre ces deux mots-clefs est l'opacité du terme produit. **Qed** produit un terme opaque qui ne peut pas être réduit. Il n'est donc pas possible de calculer avec ce terme, il ne pourra être utilisé que pour son type, dans le cadre de preuves. **Defined** produit un terme transparent qui peut être réduit. Il est donc possible de calculer avec ce terme. Si un terme est de grande taille, sa réduction peut être très longue, or certaines tactiques utilisent la réduction lors de leurs tentatives de résolution d'un but. Si un terme n'est présent que pour son type, il ne sert à rien de perdre du temps à le réduire, c'est pourquoi la plupart des lemmes et théorèmes sont définis opaques.

Passons à une preuve un peu plus technique[2], nous souhaitons prouver l'égalité suivante connue sous le nom de *map fusion* :

```
Lemma map_fusion : ∀ (A B C :Type) (f :A→B)(g :B→C) l,
                   map g (map f l) = map (fun x ⇒ g (f x)) l.
```

```
1 subgoal

  ============================
   forall (A B C : Type) (f : A -> B) (g : B -> C) (l : list A),
   map g (map f l) = map (fun x : A => g (f x)) l
```

Ce lemme est un exemple de type dépendant, le type *map* g (*map* f l) = *map* (**fun** $x \to g\ (f\ x)$) l dépend en effet des valeurs f, g et l. C'est pourquoi nous avons recours à la quantification universelle ∀ et non à la flèche → pour typer l'abstraction. ∀ a :A, B est équivalent à $A \to B$ si a n'apparaît pas dans B, c'est à dire que B ne dépend pas de a.

La preuve de ce lemme se fait par induction sur la structure de l :

2. pour les exemples qui suivent, nous utilisons la bibliothèque standard de Coq sur les listes, chargée par la commande (`Require Import List`)

```
  induction l.
2 subgoals

  A : Type
  B : Type
  C : Type
  f : A -> B
  g : B -> C
  ============================
   map g (map f nil) = map (fun x : A => g (f x)) nil
subgoal 2 is:
 map g (map f (a :: l)) = map (fun x : A => g (f x)) (a :: l)
```

Deux buts sont générés, un pour le cas où *l* est la liste vide, et un pour le cas inductif où la liste se compose d'une tête et d'une sous-liste.

Dans le cas de la liste vide, nous utilisons la tactique `simpl` qui applique une étape de réduction du terme,

```
  simpl.
2 subgoals

  A : Type
  B : Type
  C : Type
  f : A -> B
  g : B -> C
  ============================
   nil = nil
subgoal 2 is:
 map g (map f (a :: l)) = map (fun x : A => g (f x)) (a :: l)
```

puis la tactique `reflexivity` qui prouve les égalités simples, notre premier sous-but est alors résolu et le sous-but suivant nous est présenté.

```
  reflexivity.
1 subgoal

  A : Type
  B : Type
  C : Type
  f : A -> B
  g : B -> C
  a : A
  l : list A
  IHl : map g (map f l) = map (fun x : A => g (f x)) l
  ============================
   map g (map f (a :: l)) = map (fun x : A => g (f x)) (a :: l)
```

Dans le cas inductif, nous trouvons dans le contexte l'hypothèse d'induction *IHl* qui affirme que notre proposition est valable pour la sous-structure. Nous utilisons la tactique `simpl` pour dérouler une étape de calcul de la fonction *map*

```
  simpl.
1 subgoal

  A : Type
  B : Type
  C : Type
  f : A -> B
  g : B -> C
  a : A
  l : list A
  IHl : map g (map f l) = map (fun x : A => g (f x)) l
  ============================
   g (f a) :: map g (map f l) = g (f a) :: map (fun x : A => g (f x)) l
```

nous obtenons ainsi un terme dont une sous-partie est exactement la partie gauche de l'hypothèse d'induction, nous utilisons la tactique `rewrite` pour réécrire le sous-terme à l'aide de cette hypothèse. La tactique `rewrite` utilise une hypothèse dont la conclusion est une égalité et tente d'unifier un sous-terme du but avec la partie gauche de cette égalité ; si elle y parvient, elle remplace le sous-terme par la partie droite de l'égalité.

```
  rewrite IHl.
1 subgoal

  A : Type
  B : Type
  C : Type
  f : A -> B
  g : B -> C
  a : A
  l : list A
  IHl : map g (map f l) = map (fun x : A => g (f x)) l
  ============================
   g (f a) :: map (fun x : A => g (f x)) l =
   g (f a) :: map (fun x : A => g (f x)) l
```

Nous obtenons deux termes strictement égaux, nous pouvons utiliser la tactique `auto` pour finir la preuve. Cette tactique utilise tous les lemmes à sa disposition, ainsi que les hypothèses présentes dans le contexte, pour résoudre le but courant. Précédée du mot clef `info`, elle affiche ce qu'elle fait ; ici elle ne fait qu'appliquer *eq_refl*, le constructeur de l'égalité (le même résultat aurait pu être obtenu à l'aide de `reflexivity`).

```
  info auto.
 == apply eq_refl.

Proof completed.

Qed.
map_fusion is defined
```

Les preuves présentées ici utilisent les tactiques une à une pour bien montrer les différentes étapes de preuve, mais le langage `Ltac` permet d'autres constructions. Il est possible d'exécuter plusieurs tactiques en une seule fois en les combinant à l'aide de différentes tacticielles [3] :

3. traduction de *tacticals* proposée par Bertot et Castéran dans la version française du Coq'Art

```
Lemma map_fusion' : forall (A B C:Type)
                           (f:A->B)(g:B->C) l,
                          map g (map f  l) = map (fun x => g (f x)) l.
1 subgoal

  ============================
   forall (A B C : Type) (f : A -> B) (g : B -> C) (l : list A),
   map g (map f l) = map (fun x : A => g (f x)) l

Proof.
   intros A B C f g l;
   induction l;[
     reflexivity
     |
     simpl; rewrite IHl;reflexivity
   ].
Proof completed.

Qed.
map_fusion' is defined
```

Ici la tactique `intros` puis la tactique `induction` sont exécutées successivement car elles sont composées par un point-virgule. `induction` génère deux sous-buts, la syntaxe `[tac1 | tac2 ]` permet d'appliquer `tac1` au premier sous-but et `tac2` au deuxième sous-but. On voit à la position des réponses de la boucle interactive que Coq exécute en un seul bloc la suite de tactiques.

Au-delà de cette simple combinaison de tactiques, le langage `Ltac` propose également un filtrage par motif non-linéaire sur le but et les hypothèses et il permet de définir une procédure, nommée et paramétrée, dont le corps est une combinaison de tactiques. Cette procédure pourra ensuite être utilisée comme toute autre tactique. Pour plus de détails, nous suggérons au lecteur le chapitre dédié au langage de tactique dans le manuel de référence de Coq.

Nous avons vu ici comment une preuve peut être construite de façon incrémentale, en interaction avec l'assistant de preuve. Nous aurions aussi pu prouver directement les lemmes en fournissant le terme de preuve. Par exemple nous aurions pu définir le lemme *imp_trans* comme ceci :

```
Definition imp_trans' (P Q R : Prop) (H : P → Q) (H' : Q → R) (p : P) : R :=
H' (H p).
```

Le terme produit est exactement le même, cependant cette technique de preuve passe difficilement à l'échelle pour des énoncés plus complexes. Voici par exemple le terme de preuve du second lemme qui est pourtant encore relativement simple :

```
Definition map_fusion' (A B C :Type) (f :A→B)(g :B→C) l :
           map g (map f l) = map (fun x ⇒ g (f x)) l :=
  list_ind
  (fun l0 : list A ⇒ map g (map f l0) = map (fun x : A ⇒ g ( f x)) l0)
  (eq_refl nil)
  (fun (a : A) (l0 : list A)
    (IHl : map g (map f l0) = map (fun x : A ⇒ g (f x)) l0) ⇒
    eq_ind_r
    (fun l1 : list C ⇒
      g (f a) : : l1 = g (f a) : : map (fun x : A ⇒ g (f x)) l0)
    (eq_refl (g (f a) : : map (fun x : A ⇒ g (f x)) l0)) IHl)
  l.
```

list_ind est une fonction récursive, son type est le principe d'induction associé aux listes. Ce principe d'induction a été défini lors de la définition du type list.

```
  Check list_ind.
list_ind
     : forall (A : Type) (P : list A -> Prop),
       P nil ->
       (forall (a : A) (l : list A), P l -> P (a :: l)) ->
       forall l : list A, P l
```

Son premier argument est le type des éléments de la liste, le deuxième est la propriété à prouver, le troisième est la preuve de cette propriété pour le cas d'une liste vide, le quatrième est la preuve de la propriété pour une liste composée d'une tête *a* et d'une sous-liste *l* et enfin le dernier argument est la liste sur laquelle on souhaite raisonner par induction.

Prouver des propositions par la construction directe de tels termes de preuve est inenvisageable et l'utilisation interactive de Coq via `Ltac` prend ici tout son sens.

Nous avons eu ici un aperçu de l'utilisation de Coq pour définir des programmes ou des preuves, ainsi que les possibilités d'interactions avec l'assistant de preuve suivant les besoins. Les types utilisés pour les programmes étaient cependant relativement simples, voyons maintenant comment utiliser la richesse de Gallina pour typer nos programmes.

3.1.3 Programmes avec types dépendants

L'usage principal que nous avons des types dépendants lorsque nous définissons des programmes consiste à définir le sous-ensemble d'un type à l'aide d'une caractérisation des valeurs admissibles. Ainsi on pourra définir le type des listes comportant au moins un élément comme suit :

```
Definition nonEmptyList (A :Type) :={l: list A | l≠nil}.
```

Un élément de ce type est composé de deux parties, une liste et une preuve que cette liste est non-vide. La liste peut être obtenue à l'aide de la fonction de projection *proj1_sig*,

la preuve à l'aide de la projection *proj2_sig*. Ainsi pour une valeur l :*nonEmptyList* A, *proj1_sig* l est de type list A et *proj2_sig* l est de type $l \neq$ nil.

De tels types sont parfois appelés *sous-types*, car ils représentent un sous-ensemble des valeurs possibles d'un type, cependant nous préférons parler de *type fortement spécifié*. En effet, la dénomination sous-type laisse penser qu'un élément du sous-type est compatible avec le type initial, or du fait de la représentation de ces termes en Coq, il n'en est rien.

Le type *nonEmptyList* nous permet de redéfinir la fonction *hd* sans valeur par défaut puisque nous savons que la liste prise en paramètre est non vide.

Si on souhaite définir la fonction *hdStrong* qui prend une liste non-vide et retourne la tête de cette liste, il faut opérer un filtrage par motif sur la liste (obtenue par la projection *proj1_sig*). Le motif correspondant au cas où la liste contient un élément est traité simplement comme pour la fonction *hd*, mais le motif correspondant à la liste vide est plus problématique. Bien sûr ce cas est inaccessible dans les faits, puisqu'il est en contradiction avec la preuve, dont nous disposons, que la liste est non-vide. Cependant le filtrage par motif ne permet pas un filtrage partiel, il faut donc fournir un terme de type A pour le motif nil.

L'astuce ici consiste à utiliser le fait qu'à partir de *False*, il est possible de déduire n'importe quel type dans la logique de Coq. Nous allons devoir utiliser la preuve que la liste n'est pas vide, que nous obtenons en appliquant la projection *proj2_sig* sur notre paramètre. À partir de cette preuve et d'un terme de type l=nil il est possible de construire un terme de type False. Une fois ce terme obtenu, il est possible de construire un terme de n'importe quel type à l'aide de la fonction (*False_rect* A H) où H est de type False.

Comme le filtrage par motif standard *détruit* la variable filtrée, il faut utiliser un filtrage dépendant pour conserver un lien entre le terme de preuve établissant que la liste n'est pas vide et la liste elle-même. Pour plus de détails sur la syntaxe utilisée, nous renvoyons le lecteur au Manuel de référence de Coq [135, Section 2.2]. Voici le programme correspondant :

```
Definition hdStrong (A : Type) (l :{l:list A| l≠nil}) : A :=
  let l' := proj1_sig l in
    let l'_not_nil := proj2_sig l in
      match l' as l'' return (l' = l'' → A) with
        | nil ⇒ (fun Heq_l ⇒ False_rect A (l'_not_nil Heq_l))
        | (cons x _) ⇒ (fun _ ⇒ x)
      end (eq_refl l')
```

Ici, la majeure partie du terme est, dans l'intention, un programme, mais une petite partie est utilisée en tant que preuve. On voit bien que le style de définition directe des programmes ne convient pas, car la construction de la partie preuve est compliquée. Cependant le style interactif est encore moins satisfaisant car la lecture de la preuve permettant de définir ce programme ne nous permet que très difficilement de le comprendre :

```
Definition hdStrong'' (A :Type) (l :{l:list A| l≠nil}) : A .
  destruct l as [l' l_not_nil].
  destruct l' as [ |a t].
```

```
    contradict l_not_nil.
    reflexivity.

    exact a.
Defined.
```

`destruct` *l* `as` [*l'* *l_not_nil*] décompose *l* en une liste *l'* et la preuve qu'elle n'est pas vide *l_not_nil* ; `destruct` *l'* `as` [|*a t*] déconstruit *l'* suivant deux cas : le cas où *l'* vaut *nil* et le cas où *l'* se décompose en `cons` *a t*.

Le cas où *l'* vaut *nil*, est en contradiction avec l'hypothèse *l_not_nil*, dont le type est maintenant *nil* ≠ *nil*. *contradict l_not_nil* nous permet de prouver n'importe quel but si nous pouvons prouver la négation d'une hypothèse. Il s'agit ici de prouver la négation de *l_not_nil*, c'est à dire *nil*=*nil*, ce que nous faisons à l'aide de la tactique **`reflexivity`**.

Dans le cas où *l'* se décompose en `cons` *a t* nous fournissons l'élément de tête à l'aide de la tactique **`exact`** *a*, il s'agit de la valeur qui sera retournée par le programme ainsi construit. Même en ayant une bonne connaissance de la forme des termes produits par les tactiques `Ltac`, il est très difficile de se représenter l'algorithme implémenté à la simple lecture de cette définition. Nous pouvons faire mieux, en utilisant la tactique **`refine`**, qui permet de donner un terme de preuve contenant des trous puis de compléter ces trous à l'aide d'autres tactiques.

```
Definition hdStrong (A :Type) (l :{l:list A| l≠nil}) : A .
  refine ( let l' := proj1_sig l in
    let l'_not_nil := proj2_sig l in
      match l' as l'' return (l' = l'' → A) with
        | nil ⇒ _
        | (cons a t) ⇒ (fun _ ⇒ a)
      end (eq_refl l')).
  simpl in *.
  contradiction.
Defined.
```

L'algorithme implanté est plus lisible, cependant le résultat n'est toujours pas satisfaisant car le terme fournit doit être pensé pour les preuves à effectuer. De plus il n'y a pas de distinction claire entre les morceaux de preuves et la définition du programme.

3.1.4 Russel, ou la programmation confortable avec types dépendants

Une approche bien plus agréable pour l'activité de programmation avec type dépendant est apportée par le langage Russell [127].

Russel est intégré à Coq et permet de définir, en utilisant la syntaxe de Gallina, des programmes incomplets. Ceux-ci peuvent comporter des trous là où il est nécessaire d'insérer des preuves, retourner des éléments sans la partie preuve lorsqu'un élément de types fortement spécifié est attendu, ou inversement retourner un élément fortement spécifié là où est attendue une valeur simple. Les programmes sont transformés par le système pour être adaptés à l'insertion de preuve, certains trous sont laissés là où

des termes de preuves doivent être insérés et des projections sont insérées là où il faut retourner un élément sans sa spécification. Le système génère ensuite des obligations de preuves. L'utilisateur doit fournir les preuves manquantes, Russell se chargeant d'insérer les termes de preuve produits aux bons endroits dans le programme. Ainsi les activités de programmation et de preuve sont correctement séparées et l'utilisateur peut à nouveau profiter pleinement des deux paradigmes d'utilisation de Coq.

Voyons comment définir notre fonction *hdStrong* à l'aide de Russell. Afin de pouvoir présenter un exemple simple, nous avons désactivé la tactique de résolution automatique des obligations de preuve chargée avec le module PROGRAM qui permet d'utiliser Russell.

```
Obligation Tactic := idtac.
```

La tactique normalement chargée résout automatiquement les obligations de preuve générées par les exemples suivants.

Pour signifier que nous souhaitons utiliser Russell, il suffit de placer le mot clef `Program` devant notre définition. Nous laissons un trou (noté `_`) là où nous avons besoin de raisonner par preuve.

La structure du filtrage par motif que nous écrivons ici n'est pas construite pour permettre la preuve du cas absurde. En effet pour montrer que le motif `nil` est un cas absurde, il faut pouvoir lier la variable *l* à ce motif, il faudrait donc utiliser un filtrage dépendant. Russell s'occupe de transformer le filtrage en filtrage dépendant pour nous.

```
Program Definition  hdStrong (A:Type)  (l :{lst:list A| lst<>nil}) : A :=
  match l with
    | nil =>  _
    |  x::t => x
  end.
hdStrong has type-checked, generating 1 obligation(s)
Solving obligations automatically...
1 obligation remaining
Obligation 1 of hdStrong:
forall (A : Type) (l : {lst : list A | lst <> nil}),
let filtered_var := proj1_sig l in nil = filtered_var -> A.
```

Le système a généré une obligation de preuve, nous demandant de fournir un élément de type A dans le cas où la variable filtrée vaut `nil`. Pour résoudre cette obligation de preuve, nous utilisons les mots clefs `Next Obligation`. Le système affiche alors le but à prouver :

```
  Next Obligation.
1 subgoal

  ============================
   forall (A : Type) (l : {lst : list A | lst <> nil}),
   let filtered_var := proj1_sig l in nil = filtered_var -> A
```

Toutes les tactiques `Ltac` sont disponibles pour construire une preuve. Ici nous commençons par introduire toutes les hypothèses dans l'environnement, nous décomposons ensuite la liste non vide *l* en une liste *l*, et la preuve qu'elle n'est pas vide *l_not_nil*, et nous appliquons une étape de réduction dans toutes les hypothèses afin d'éliminer les projections.

```
intros A l filtered_var Heq_l; destruct l as [l l_not_nil]; simpl in *.
1 subgoal

  A : Type
  l : list A
  l_not_nil : l <> nil
  filtered_var := l : list A
  Heq_l : nil = filtered_var
  ============================
   A
```

Nous inversons l'ordre des éléments dans l'égalité *Heq_l* afin qu'ils soient dans le même ordre que l'inégalité *l_not_nil* et nous utilisons la tactique `contradiction` afin de prouver le but grâce à la contradiction entre *Heq_l* et *l_not_nil*.

```
  symmetry in Heq_l; contradiction.
Proof completed.

Qed.
hdStrong_obligation_1 is defined
No more obligations remaining
hdStrong is defined
```

Une fois la preuve effectuée et enregistrée, Russell place le terme construit dans le trou correspondant du programme. Lorsque tous les trous du programmes sont complétés, Russell soumet celui-ci à Coq, qui vérifie le type du programme. Cette étape nous assure que le terme construit par Russell est un programme Coq valide. Le programme ainsi défini n'est pas exactement celui que nous avons écrit, il a subi quelques transformations nécessaires à la création et à la résolution des obligations de preuve :

```
  Print   hdStrong.
hdStrong =
fun (A : Type) (l : {lst : list A | lst <> nil}) =>
let filtered_var := proj1_sig l in
let branch_0 :=
  fun Heq_l : nil = filtered_var => hdStrong_obligation_1 A l Heq_l in
let branch_1 :=
  fun (x : A) (t : list A) (_ : x :: t = filtered_var) => x in
match filtered_var as l0 return (l0 = filtered_var -> A) with
| nil => branch_0
| x :: t => branch_1 x t
end (eq_refl filtered_var)
     : forall A : Type, {lst : list A | lst <> nil} -> A
Argument scopes are [type_scope _]
```

Le filtrage par motif se fait sur une variable à laquelle a été affectée la projection sur le premier élément de l, de type list A, ce qui correspond aux motifs que nous avons proposés. On peut noter aussi l'apparition de *hdStrong_obligation_1*, qui correspond à l'obligation de preuve que nous avons résolue.

Nous aurions aussi pu écrire *hdStrong* avec une spécification forte de son type de retour. Par exemple, si nous voulons retourner, en plus de l'élément de tête de la liste, une preuve que le résultat obtenu est le même que celui qui aurait été obtenu par la fonction *hd* :

```
Program Definition  hdStrong' (A:Type)  (l :{lst:list A| lst<>nil}) :
                               {a:A | forall default, hd default l = a} :=
  match l with
    | nil =>  _
    | x::t => x
  end.
hdStrong' has type-checked, generating 2 obligation(s)
Solving obligations automatically...
2 obligations remaining
Obligation 1 of hdStrong':
forall (A : Type) (l : {lst : list A | lst <> nil}),
let filtered_var := proj1_sig l in
nil = filtered_var -> {a : A | forall default : A, hd default nil = a}.

Obligation 2 of hdStrong':
forall (A : Type) (l : {lst : list A | lst <> nil}),
let filtered_var := proj1_sig l in
forall (x : A) (t : list A),
x :: t = filtered_var -> forall default : A, hd default (x :: t) = x.
```

Nous ne nous préoccupons toujours pas de l'aspect preuve dans la définition de notre programme. Russell nous demande cette fois de résoudre deux obligations de preuves.

La première est presque la même que précédemment sauf qu'elle nous demande de fournir un terme de type $\{a : A \mid \forall\ default,\ hd\ default\ \mathsf{nil} = a\}$ et non de type A. La preuve précédente fonctionne tout aussi bien puisque la contradiction est toujours valable.

```
Next Obligation.
1 subgoal

  ============================
   forall (A : Type) (l : {lst : list A | lst <> nil}),
   let filtered_var := proj1_sig l in
   nil = filtered_var ->
   {a : A | forall default : A, hd default nil = a}

  intros; destruct l; simpl in * ; symmetry in Heq_l ;contradiction.
Proof completed.

Qed.
hdStrong'_obligation_1 is defined
1 obligation remaining
```

Une fois cette obligation terminée et sauvegardée, Russell nous avertit qu'il reste une obligation de preuve.

La seconde obligation correspond à la deuxième branche du filtrage par motif et nous demande de prouver que le terme que nous retournons est bien égal au résultat de *hd*. Cette preuve est suffisamment simple pour être résolue par `reflexivity`.

```
Next Obligation.
1 subgoal

  ============================
   forall (A : Type) (l : {lst : list A | lst <> nil}),
   let filtered_var := proj1_sig l in
   forall (x : A) (t : list A),
   x :: t = filtered_var -> forall default : A, hd default (x :: t) = x

  intros;  reflexivity.
Proof completed.

Qed.
hdStrong'_obligation_2 is defined
No more obligations remaining
hdStrong' is defined
```

Pour récapituler, le code source de notre programme est le suivant :

```
Program Definition hdStrong' (A :Type) (l :{lst:list A| lst≠nil}) :
                                    {a:A | ∀ default, hd default l = a} :=
  match l with
    | nil ⇒ _
    | cons x _ ⇒ x
  end.
Next Obligation.
  intros; destruct l; simpl in *; symmetry in Heq_l;contradiction.
Qed.
Next Obligation.
  intros; reflexivity.
Qed.
```

La définition du programme et les preuves sont clairement découplées. L'algorithme développé est lisible, et nous avons pu profiter pleinement de l'interactivité de Coq pour prouver les compatibilités entre notre algorithme et les types manipulés. En pratique les deux obligations de preuve sont résolues automatiquement par la tactique chargée par `Program` que nous avons désactivé et il n'est besoin que de donner la définition du programme.

Le programme généré par Russell est le suivant :

```
Definition hdStrong' (A : Type) (l : {lst : list A | lst ≠ nil}) :
                                    {a : A | ∀ default : A, hd default (proj1_sig l) = a}
  :=
  let filtered_var := proj1_sig l in
    let branch_0 :=
      fun Heq_l : nil = filtered_var ⇒ hdStrong'_obligation_1 A l Heq_l in
        let branch_1 :=
          fun (x : A) (t : list A) (Heq_l : x :: t = filtered_var) ⇒
            exist (fun a : A ⇒ ∀ default : A, hd default (x :: t) = a) x
            (hdStrong'_obligation_2 A l x t Heq_l) in
            match filtered_var as l0 return
              (l0 = filtered_var →
                {a : A | ∀ default : A, hd default l0 = a})
              with
              | nil ⇒ branch_0
              | x :: t ⇒ branch_1 x t
            end (eq_refl filtered_var).
```

Le programme généré est plus compliqué que précédement, nous nous contenterons de noter que les valeurs retournées dans chaque branche sont construites à l'aide de nos obligations de preuve et que le système a inséré une projection *proj1_sig* sur *l* dans le type de retour, cette projection était nécessaire car *hd* prend un paramètre de type list et non de type *nonEmptyList*.

Ainsi Russell nous permet de programmer avec des types fortement spécifiés comme s'il s'agissait réellement de sous-types, convertibles avec le type initial. Le gain en confort d'utilisation des types dépendants est très appréciable. Il faut cependant rester conscient, lors de l'utilisation de ce langage, que Russell effectue des transformations sur les programmes et les types que nous écrivons. Il introduit donc un décalage entre la représentation des types et des programmes données dans le code source et la forme réelle des termes produits. Ce décalage permet cependant de conserver un code source plus lisible, permettant une meilleur compréhension de l'intention des programmes écrits.

3.1.5 Exécution de programmes dans l'assistant de preuve

Nous avons vu qu'il est possible d'appliquer une étape de réduction sur un terme Coq à l'aide de la tactique `simpl`. Il est aussi possible de réduire complètement un terme à l'aide de la tactique `cbv`, le terme est alors réduit suivant une stratégie d'appel par valeur, ou `lazy`, la réduction se fait alors par appel par nom. Ces tactiques doivent être suivies du type de réduction à effectuer : `beta` pour la β-réduction correspondant à la substitution de variable lors de l'application fonctionnelle, `iota` pour la ι-réduction, correspondant à l'évaluation d'un filtrage par motif, `delta` pour la δ-réduction, correspondant au remplacement du nom d'une fonction par son corps, et enfin `zeta` pour la ζ-réduction, correspondant à l'expansion des définitions locales (`let` x := ... `in` ...) . La tactique `compute`, équivalente à `cbv beta delta iota zeta`, permet de réduire un terme à sa forme normale. Toutes ces tactiques d'évaluation sont en fait utilisables en dehors des preuves à l'aide de la commande `Eval`.

Par exemple, si nous souhaitons appliquer la fonction successeur à tous les entiers contenus dans une liste, nous procéderons comme suit :

```
Eval compute in (map  (fun x => x+1) [ 0 ; 1 ; 2 ; 3 ; 4 ; 5 ; 6 ]).
     = [1; 2; 3; 4; 5; 6; 7]
     : list nat
```

Le terme résultat de ce calcul peut être enregistré comme tout terme :

```
Definition ma_fonction :=(map  (fun x => x+1) [ 0 ; 1 ; 2 ; 3 ; 4 ; 5 ; 6 ]).
ma_fonction is defined

Print ma_fonction.
ma_fonction =
map (fun x : nat => x + 1) [0; 1; 2; 3; 4; 5; 6]
     : list nat

Definition ma_liste := Eval compute in ma_fonction.
ma_liste is defined

Print ma_liste.
ma_liste = [1; 2; 3; 4; 5; 6; 7]
     : list nat
```

Il est aussi possible de limiter la δ-réduction à certaines fonctions (ici *ma_fonction* puis *ma_fonction* et *map*) :

```
Eval cbv beta  iota zeta delta [ma_fonction] in ma_fonction.
     = map (fun x : nat => x + 1) [0; 1; 2; 3; 4; 5; 6]
     : list nat

Eval cbv beta iota zeta delta [ma_fonction map] in ma_fonction.
     = [0 + 1; 1 + 1; 2 + 1; 3 + 1; 4 + 1; 5 + 1; 6 + 1]
     : list nat
```

ou encore d'interdire l'expansion de certaines fonctions (ici *plus*) :

```
Eval cbv beta iota zeta delta -[plus] in ma_fonction.
     = [0 + 1; 1 + 1; 2 + 1; 3 + 1; 4 + 1; 5 + 1; 6 + 1]
     : list nat
```

Bien sûr, si notre terme contient des axiomes, il ne pourra être complètement réduit puisque les axiomes sont des éléments avec un type mais pas de corps.

```
Parameter fonction_axiome : nat -> nat.
fonction_axiome is assumed

Eval compute in (map fonction_axiome [ 0 ; 1 ; 2 ; 3 ; 4 ; 5 ; 6 ]).
     = [fonction_axiome 0; fonction_axiome 1; fonction_axiome 2;
       fonction_axiome 3; fonction_axiome 4; fonction_axiome 5;
       fonction_axiome 6]
     : list nat
```

3.1.6 Extraction de programmes

Exécuter les programmes dans l'assistant de preuve est intéressant mais peu efficace. Les calculs sont coûteux d'abord parce qu'ils sont interprétés à la volée et aussi parce que les termes de preuves sont également réduits s'ils ne sont pas opaques. Ces termes de preuve ne sont là que pour assurer la correction du typage mais ils n'apportent rien du point de vue algorithmique, leur élimination avant l'exécution est donc souhaitable. En plus d'une évaluation coûteuse, les programmes développés en Coq ne peuvent gérer ni les entrées/sorties, ni les interactions avec l'utilisateur ou d'autres programmes, et sont de ce fait très limités.

Le mécanisme d'extraction de programmes vers d'autres langages permet d'utiliser les programmes développés en Coq de façon efficace, dans des programmes plus larges, permettant des interactions avec le système et l'utilisateur plus complexes que l'usage de la boucle interactive de Coq. Les programmes peuvent être extraits vers OCaml, Haskell et Scheme.

La première phase de cette extraction est la même quel que soit le langage choisi. Elle supprime du corps du programme tous les termes de preuve ne servant pas au calcul. Il est possible d'indiquer à Coq qu'un terme doit être considéré comme élément de preuve ou comme algorithme de calcul en choisissant le type de son type parmi `Type`, `Prop` ou `Set`. Tout élément dont le type est un habitant de `Prop` est considéré comme un élément de preuve et est éliminé si possible au moment de l'extraction. Tout élément dont le type est un habitant de `Set`, au contraire, est considéré comme un élément de calcul et est conservé au moment de l'extraction. Afin d'assurer la cohérence des programmes extraits, il n'est pas possible d'utiliser un filtrage par motif sur une variable dont le type appartient à `Prop` pour construire un élément dont le type n'est pas dans `Prop`. La variable étant éliminée à l'extraction, il ne serait plus possible de retourner une valeur.

Il arrive qu'un élément de type `Prop` ne soit pas entièrement éliminé parce qu'il apparaît en temps qu'élément d'une structure de donnée appartenant à `Set`, dans ce cas il est remplacé à l'extraction par un élément `__` qui n'effectue aucun calcul.

L'extraction récursive permet d'extraire un terme ainsi que tous les termes dont il dépend. Ainsi pour notre programme *hdStrong'* cela donne :

```
  Recursive Extraction hdStrong'.
let __ = let rec f _ = Obj.repr f in Obj.repr f
type 'a list =
  | Nil
  | Cons of 'a * 'a list
type 'a sig0 = 'a
  (* singleton inductive, whose constructor was exist *)
(** val hdStrong'_obligation_1 : 'a1 list -> 'a1 **)
let hdStrong'_obligation_1 l =
  assert false (* absurd case *)
(** val hdStrong' : 'a1 list -> 'a1 **)
let hdStrong' l =
  let branch_0 = fun _ -> hdStrong'_obligation_1 l in
  let branch_1 = fun x t -> x in
  (match l with
     | Nil -> branch_0 __
     | Cons (x, t) -> branch_1 x t)
```

la définition `__` est complexe et sort du cadre de ce document, on se contentera de dire qu'elle remplace les termes de preuve là où ils n'ont pas pu être entièrement éliminés pour des raisons de correction du typage.

La définition inductive du type list est donnée. L'obligation de preuve correspondant au cas où la liste est vide lève une erreur car il s'agit d'un cas inatteignable dans la définition Coq. Ici cependant la preuve que la liste est non-vide est éliminée à l'extraction, il faudra donc faire attention à bien utiliser le programme dans des conditions satisfaisant les pré-conditions imposées lors de la définition en Coq. Dans le corps de *hdStrong'*, tous les éléments correspondant à des preuves sont éliminés.

La définition donnée pour le type liste n'est pas celle des listes OCaml, or l'implantation OCaml des listes est optimisée puisqu'elles sont au cœur du langage. Nous souhaiterions profiter de ces optimisations, nous allons donc associer le type de liste Coq au type de liste OCaml à l'aide d'une directive d'extraction :

```
Extract Inductive list => "list" [ "[]" "(::)" ].
```

le type inductif *list* de Coq est associé au type `list` OCaml, les deux constructeurs de listes Coq associés à leur équivalent en OCaml : `[]` pour nil et l'opérateur infixe `(::)` pour cons.

```
Recursive Extraction hdStrong'.
let __ = let rec f _ = Obj.repr f in Obj.repr f
type 'a sig0 = 'a
  (* singleton inductive, whose constructor was exist *)
(** val hdStrong'_obligation_1 : 'a1 list -> 'a1 **)
let hdStrong'_obligation_1 l =
  assert false (* absurd case *)
(** val hdStrong' : 'a1 list -> 'a1 **)
let hdStrong' l =
  let branch_0 = fun _ -> hdStrong'_obligation_1 l in
  let branch_1 = fun x t -> x in
  (match l with
     | [] -> branch_0 __
     | x::t -> branch_1 x t)
```

L'extraction ne donne maintenant plus de définition des listes, et les listes OCaml sont utilisées à la place.

Pour l'extraction de bibliothèques Coq entières, nous utiliserons la commande :

```
Recursive Extraction Library maBilbiothèque.
```

Elle crée un fichier ml par fichier de bibliothèque Coq. Grâce au mot-clef `Recursive`, toutes les bibliothèques dont dépend maBilbiothèque seront extraites.

Après extraction, ces bibliothèques pourront être réutilisées dans des développements utilisant toutes les possibilités du langage cible et compilées afin d'obtenir un programme efficace.

3.1.7 Classes de type

La surcharge d'opérateur, également appelée polymorphisme *ad hoc* peut être introduite dans les langages fonctionnels grâce aux classes de type [144]. Celles-ci ont été récemment introduites dans Coq par Sozeau [127]. Une classe regroupe un ensemble de

signatures d'opérations, éventuellement accompagnées d'une définition. Elle est généralement paramétrée par un ou plusieurs types. Elle permet ainsi de déclarer l'existence de ces opérations.

Prenons l'exemple de l'addition. Cette opération est définie en Coq pour plusieurs types différents, avec des noms différents : *plus* pour le type des entiers naturels nat, *Zplus* pour le type des entiers relatifs Z, *Qplus* pour le type des nombres rationnels Q, etc. Elle n'est cependant pas disponible pour tous les types existants, et pour chaque type disposant de cette opération, elle sera définie différemment. Nous ne pouvons donc pas la définir comme une opération polymorphe au sens du polymorphisme paramétrique usuel.

Pour utiliser un même nom de fonction pour toutes les opérations d'addition, nous définissons une classe de type nommée Addition, paramétrée par un type *number* pour lequel nous décrivons la signature de l'addition : `Class Addition` (*number* : `Type`) :={ *add* : *number* → *number* → *number*}.

Nous pouvons définir une notation infixe pour la méthode *add* afin de rendre son usage plus confortable.

Infix " + " := *add*.

Pour le moment *add* n'est définie pour aucun type. Nous devons déclarer une instance de la classe pour associer une définition à cette opération [4] :

`Instance` *Addition_Nat* : Addition nat :={*add* :=*plus*}.

`Instance` *Addition_Z* : Addition Z :={*add* :=*Zplus*}.

`Instance` *Addition_Q* : Addition Q :={*add* :=*Qplus*}.

La définition d'une instance est identique à une définition normale mise à part l'utilisation du mot clef `Instance` au lieu de `Définition`. Il est donc possible de donner explicitement le corps du terme, ou de se limiter à donner le type, puis utiliser `Ltac` pour construire le corps de l'instance. Une classe peut être vue comme un type enregistrement, pour définir un terme d'un tel type on peut utiliser la notation {*champ1* :=*corps1* ; *champ2* :=*corps2* ; ... }.

Nous avons ici déclaré trois instances de notre classe, *Addition_Nat* est une instance de la classe Addition pour le type nat. Cette instance définit la méthode *add* comme étant l'opération *plus*. *Addition_Z* et *Addition_Q* sont des instances définies de la même manière pour les types Z et Q.

Nous pouvons à présent utiliser notre méthode *add*, notée +, pour calculer l'addition indifféremment sur les naturels, les relatifs ou les rationnels [5] :

4. Les exemples présentés ici utilise les bibliothèques *QArtih Zarith* et *Reals*

5. Ici la même notation est utilisée pour les nombres, quelque soit leur type. Il s'agit d'un artifice syntaxique implanté par les développeurs de Coq dans des modules chargés en même temps que les bibliothèques définissant les différentes arithmétiques sur ces types. La surcharge de notre opération n'utilise, elle, que le mécanisme des classes de type.

```
Eval compute in (let x:= (1)%nat in x+x).
     = 2
     : nat

Eval compute in (let x:= (1)%Z in x + x  ).
     = 2%Z
     : Z

Eval compute in (let x:= (1)%Q in x + x  ).
     = 2 # 1
     : Q
```

Notre addition ne marchera pas pour les réels car nous n'avons pas déclaré d'instance pour ce type :

```
Eval compute in (let x:= (1)%R in x+x).
Toplevel input, characters 34-37:
> Eval compute in (let x:= (1)%R in x+x).
>                                  ^^^
Error: Cannot infer the implicit parameter Addition of
add.
Could not find an instance for "Addition R" in environment:
x := 1%R : R
```

Le système nous dit qu'il rejette notre terme car il ne peut pas trouver d'instance de notre classe correspondant au type R. Lorsqu'une méthode de classe est utilisée, une recherche d'instance est déclenchée au moment de l'inférence de type. Cette recherche prend en compte les contraintes imposées par le type des paramètres afin d'obtenir une instance compatible. L'instance trouvée servira à définir le corps de la fonction à utiliser.

Notre classe ne fait que suggérer l'existence d'une opération nommée *add* pour un type *number*, nous pourrions aller plus loin en précisant par exemple que l'addition est une opération associative. L'associativité étant une propriété commune à plusieurs opérateurs, autant la définir à l'aide d'une classe de type également :

```
Class Associative (A :Type) (op : A → A →A) :={
  associativity : ∀ a b c, op a (op b c) = op (op a b) c
}.
```

Cette classe définit une méthode *associativity* dont le type exprime l'associativité de l'opérateur *op* paramètre de la classe. Nous redéfinissons notre classe Addition pour imposer que l'opérateur *add* soit associatif :

```
Class Addition (number : Type) :={
  add : number → number → number ;
  add_assoc :> Associative number add
}.
```

La notation :> déclare le champ *add_assoc* comme une instance de la classe Associative. Nous construisons cette fois notre instance de la classe Addition pour les naturels en utilisant en partie Ltac :

```
Instance addition_nat : Addition nat :={add :=plus}.
Proof.
constructor.
auto with arith.
Qed.
```

Le champ *add* est définit directement, le champ *add_assoc* est défini à l'aide du langage de tactique. Nous pouvons maintenant utiliser la méthode *associativity* sur une combinaison d'additions sur les naturels :

```
Goal forall i j k l : nat, i+(j+(k+l)) = i+j+k+l.
1 subgoal

  ============================
   forall i j k l : nat, i + (j + (k + l)) = i + j + k + l

  intros i j k l;
  repeat rewrite associativity.
1 subgoal

  i : nat
  j : nat
  k : nat
  l : nat
  ============================
   i + j + k + l = i + j + k + l

  reflexivity.
Proof completed.

Qed.
Unnamed_thm is defined
```

On voit ici que l'utilisation des classes de types est autant profitable dans l'activité de preuve que pour l'écriture de programme. En effet, cela permet de regrouper sous une même appellation des propriétés similaires sur des types différents. Ici, il n'est plus nécessaire de se souvenir du nom du terme donnant la propriété d'associativité d'un opérateur particulier pour utiliser cette propriété, le système ira le chercher pour nous.

Une instance de classe peut également être paramétrée par une autre instance de classe. Lorsque l'algorithme de recherche d'instances essaiera d'utiliser cette instance, il tentera de compléter ses paramètres, si ceux-ci sont typés par une classe de type, l'algorithme lancera une recherche d'instance pour ces paramètres.

Cette technique de recherche d'instance fonctionne comme la stratégie de recherche

de Prolog en profondeur et avec retour arrière [6] utilisé habituellement en programmation logique.

Prenons pour exemple une classe décrivant les listes non-vides.

```
Class NonEmpty (A :Type) (l :list A) := {
  NonEmptyList : l≠ nil
}.
```

Une instance triviale est celle définie sur une liste construite par cons :

```
Instance consNonEmpty (A :Type) (a :A) (l :list A) : NonEmpty (a::l).
```

Il est aussi possible de construire [7], à partir d'une liste non-vide, une nouvelle liste non-vide en la concaténant à n'importe quelle liste (à droite ou à gauche), l'opérateur de concaténation est noté ++ :

```
Instance appNonEmptyLeft (A :Type) (l1 l2 :list A) {nvl : NonEmpty l1}
                                    : NonEmpty (l1 ++ l2).
Instance appNonEmptyRight (A :Type) (l1 l2 :list A) {nvl : NonEmpty l2}
                                    : NonEmpty (l1 ++ l2).
```

On peut également prouver que l'application de *map* sur une liste non-vide construit une liste non-vide :

```
Instance mapNonEmpty (A B :Type) (f :A→B) (l :list A) {nvl :NonEmpty l}
                                                  : NonEmpty(map f l).
```

Utilisons cette classe pour redéfinir la fonction *hdStrong* vue précédemment :

```
Program Definition hdStrongClass {A :Type} (l :list A) {nvl : NonEmpty l} :A :=
  match l with
    nil ⇒ _
    | h :: t ⇒ h
  end.
```

La fonction a pour paramètre implicite et *maximalement inséré* une instance de la classe NonEmpty appliquée à *l*. Le caractère *maximalement inséré* d'un paramètre est marqué par l'utilisation d'accolades au lieu de parenthèses pour le délimiter : {*nvl* : NonEmpty *l*}. Lorsqu'une définition dépend d'un paramètre *maximalement inséré* typé par une classe, l'algorithme d'inférence de type effectue une recherche d'instance à chaque fois qu'il la rencontre. S'il ne trouve pas d'instance, l'inférence de type échoue.

Si par exemple on utilise *hdStrongClass* avec une liste quelconque, le système ne pourra trouver aucune instance de la classe NonEmpty pour cette liste,

6. backtracking

7. Nous ne détaillons pas ici les preuves permettant de définir ces instances, seule leur spécification est donnée. Le code de cet exemple est disponible à l'adresse `http://tesson.julien.free.fr/research/project/sdpp.php`

```
Check (fun l => hdStrongClass  l).
Toplevel input, characters 16-29:
> Check (fun l => hdStrongClass  l).
>                 ^^^^^^^^^^^^^
Error: Cannot infer the implicit parameter nvl of
hdStrongClass.
Could not find an instance for "NonEmpty l" in environment:
l : list?88
With the following constraints:
?88 == Type
```

par contre si la liste est définie par le constructeur cons, l'instance *consNonEmpty* sera automatiquement utilisée.

```
Check (fun A (a:A) l => hdStrongClass  (a::l)).
fun (A : Type) (a : A) (l : list A) => hdStrongClass (a :: l)
     : forall A : Type, A -> list A -> A
```

La recherche d'instances peut-être poussée plus loin en utilisant les instances qui dépendent d'autres instances :

```
Typeclasses eauto := debug.

Check (fun A B (a:A) (f: A -> B) l =>
                      hdStrongClass (map f ((l++(a::nil))++l))).
1.1: apply mapNonEmpty on
(NonEmpty (map f ((l ++ a :: nil) ++ l)))
1.1.1.1: apply appNonEmptyLeft on
(NonEmpty ((l ++ a :: nil) ++ l))
1.1.1.1.1.1: apply appNonEmptyLeft on
(NonEmpty (l ++ a :: nil))
1.1.1.1.1.1.1: no match for (NonEmpty l)
3 possibilities
1.1.1.1.1.2: apply appNonEmptyRight on
(NonEmpty (l ++ a :: nil))
1.1.1.1.1.2.1.1: apply consNonEmpty on
(NonEmpty (a :: nil))
no backtrack on (NonEmpty (a :: nil))
no backtrack on (NonEmpty (a :: nil))
no backtrack on (NonEmpty (l ++ a :: nil))
no backtrack on (NonEmpty ((l ++ a :: nil) ++ l))
fun (A B : Type) (a : A) (f : A -> B) (l : list A) =>
hdStrongClass (map f ((l ++ a :: nil) ++ l))
     : forall A B : Type, A -> (A -> B) -> list A -> B
```

Nous avons ici activé l'affichage de la trace de la recherche d'instances afin de suivre le chaînage qui a mené à la construction de l'instance de la classe NonEmpty.

Cette instance utilise *mapNonEmpty*, puis *appNonEmptyLeft*, suivi de *appNonEmptyRight*, pour finir avec *consNonEmpty*. Cette série d'instances a été trouvée par un mécanisme de recherche avec retour arrière utilisant les bases d'instances de chacune des classes pour lesquelles une instance est recherchée. Si, dans un contexte donné, plusieurs instances sont utilisables, la dernière instance déclarée sera choisie. Il est cependant possible d'affecter une priorité à une instance, ce qui donne un certain contrôle sur la recherche d'instance.

Ainsi les classes peuvent être utilisées pour la construction automatique d'un terme lors de l'inférence de type à partir des instances d'une ou plusieurs classes.

3.1.8 Résumé

Si nous prenons le point de vue d'un programmeur, Coq offre un langage fonctionnel primitif récursif d'ordre supérieur, supportant un polymorphisme explicite ; un polymorphisme ad-hoc grâce aux classes de types ; la spécification forte des fonctions et des valeurs grâce aux types dépendants. Il offre également un mécanisme de modules et de foncteurs que nous n'avons pas présenté ici, similaires à ceux d'Objective Caml. Ces foncteurs permettent la généricité du code par rapport à un ensemble d'opérations, de preuves, etc.

Du point de vue de l'activité de preuve Coq propose : un langage de spécification expressif ; un mécanisme de réécriture ; un langage pour la construction interactive de preuves ; la possibilité de construire nos propres tactiques de preuve par une composition de tactiques existantes.

3.2 Le langage Bulk Synchronous Parallel ML

3.2.1 Description générale du langage

Le langage BSML est une extension du langage OCaml permettant de définir des programmes parallèles quasi-synchrone.

Il existe, pour le langage BSML, différentes implantation des primitives parallèle. Une implantation séquentielle permet la simulation séquentielle de programme BSML. Elle est notamment utilisée par la boucle interactive BSML qui permet la définition rapide de petits prototypes. D'autres implantations utilisent diverses bibliothèques pour la communication entre processus (TCP/IP, MPI, PVM, PUB,...). Pour toutes ces implantations, il existe un compilateur utilisant le compilateur OCaml. L'abstraction apportée par le modèle de programmation nous a permis d'exécuter un même programme parallèle sur des machines aussi diverses que notre machine SPEED qui utilise en mémoire partagée 4 processeurs ayant 12 cœurs chacun, le Centre de Calcul Scientifique région Centre (CCSC), où nous avons pu réserver jusqu'à 128 coeurs répartis sur 16 nœuds biprocesseur reliés en InfiniBand, et le Très Grand Centre de Calcul Curie où nous avons

pu utiliser jusqu'à 2048 cœurs répartis sur 64 nœuds de 4 processeurs, les nœuds étant inter-connectés par une liaison InfiniBand.

Contrairement au paradigme de programmation SPMD proposé par les implantations impératives courantes du modèle quasi-synchrone, le langage BSML donne une vision explicite du parallélisme. Deux niveaux d'exécutions peuvent être distingués à la lecture d'un programme BSML : un niveau global où l'ensemble des valeurs définies est accessible à tous les processeurs, et un niveau local où les calculs se font sur un seul processeur de la machine BSP et où les valeurs calculées ne sont accessibles qu'à ce processeur. Les valeurs locales doivent être explicitement communiquées pour être utilisées au niveau global ou par un autre processeur.

Le programme principal est défini au niveau global et manipule une structure nommée *vecteur parallèle* à l'aide des primitives parallèles du langage. Ce vecteur parallèle contient une valeur locale par processeur. Rappelons que dans le langage OCaml une valeur peut aussi bien être une fonction qu'une donnée. Il est ainsi possible de créer des vecteurs parallèles de fonctions et donc de manipuler une représentation explicite des processus parallèles.

Le vecteur parallèle est représenté par un type polymorphe, `'a par`. C'est un vecteur de taille constante `p` (le nombre de processeurs de la machine BSP) contenant des termes du langage OCaml. Nous décrirons informellement une telle structure par $\langle x_0 \ldots x_{p-1} \rangle$, ou x_i est la valeur présente sur le $i^{\text{ème}}$ processeur. L'imbrication de vecteurs parallèles est interdite, un système de type rejetant les programmes contenant des imbrications de vecteurs a été définie dans [65].

3.2.2 Primitives du langage

Le langage BSML donne également accès aux paramètres de la machine BSP, le plus utilisé étant le nombre de processeurs `p` accessible via la valeur `bsp_p`.

Création d'un vecteur parallèle

La création d'un vecteur parallèle se fait à l'aide de la primitive ***mkpar***. Cette primitive prend une fonction `f` en paramètre et construit un vecteur contenant localement, à chaque processeur `i`, la valeur `(f i)`. La signature de cette primitive est ***mkpar***: `(int→'a) →'a par`. Sa sémantique informelle peut être décrite ainsi :

$$\textbf{\textit{mkpar}}\ \mathtt{f} = \langle \mathtt{f}\ 0, \ldots, \mathtt{f}\ (\mathtt{p}-1) \rangle.$$

Dans la boucle interactive, sur une machine à six processeurs, on obtient :

```
# let this = mkpar (fun i →i);;
val this : int par = <0, 1, 2, 3, 4, 5>
```

`#` est l'invite de commande, l'expression **`let this =`** définit la valeur `this` en appliquant la primitive ***mkpar*** à la fonction identité. La seconde ligne est la réponse de la boucle interactive, signifiant que la valeur nommée `this` est ⟨ 0,1,2,3,4,5⟩, de type

vecteur parallèle d'entiers (*int par*). Un vecteur parallèle à donc été créé avec, à chaque processeur, sont identifiant.

Une autre fonction simple utilisant ***mkpar*** est la fonction `replicate` qui place la même valeur en chaque point du vecteur :

```
# let replicate = fun x →mkpar(fun _→x);;
val replicate : 'a →'a par = <fun>

# let hello = replicate "Hello";;
hello : string par = <"Hello", "Hello", "Hello",
                      "Hello", "Hello", "Hello">
```

L'évaluation de l'application de ***mkpar*** à une fonction `f` ne nécessite aucune communication ni synchronisation car toutes les valeurs OCaml définies au niveau global, telle que la fonction identité ou la chaîne de caractère "hello" dans les exemples précédents, sont répliquées sur tous les processeurs. Cette évaluation est donc faite durant la phase de calcul asynchrone d'une super-étape. Le coût d'évaluation de ***mkpar*** `f` est $\max_{0\leq i<p} \overline{f\ i}$ où $\overline{e}$ est le coût de l'évaluation de l'expression e.

Application parallèle point à point

Le type *par* est polymorphe, il est donc possible de construire un vecteur parallèle de fonctions. Par exemple, la fonction suivante construit un vecteur qui contient en chaque point une fonction qui concatène son argument et le numéro de processeur local :

```
# let vf = mkpar(fun i→ fun x →x ^ (string_of_int i));;
val vf : (string →string) par = <<fun>, ..., <fun>>
```

Un tel vecteur de fonctions n'est utile que s'il peut être appliqué point à point à un vecteur de valeurs. Cette application n'est pas l'application de fonction intrinsèque du langage OCaml, il faut donc utiliser la primitive ***apply***. Sa signature est : ***apply***: `('a→'b)`*par*`→'a` *par*`→'b` *par*.

Par exemple, nous pouvons appliquer ce vecteur de fonctions `vf` au vecteur de valeurs `hello` :

```
# let hello_i = apply vf hello;;
val hello_i : string par = <"Hello0", "Hello1", "Hello2",
                            "Hello3", "Hello4", "Hello5">
```

La sémantique informelle de ***apply*** peut être vue comme :

$$\textbf{\textit{apply}}\ \langle f_0, ..., f_{p-1}\rangle\ \langle x_0, ..., x_{p-1}\rangle = \langle f_0\ x_0, ..., f_{p-1}\ x_{p-1}\rangle$$

L'évaluation de cette primitive ne nécessite pas de communication ni de synchronisation ; son coût d'évaluation est $\max_{0\leq i<p} \overline{f_i\ x_i}$.

À partir des fonctions ***mkpar*** et ***apply*** il est possible d'implanter tous les programmes parallèles ne nécessitant pas de communication. Dans la bibliothèque standard de programmation de BSML, on trouvera notamment la très utile fonction

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

appliquant `f` à tous les éléments d'un vecteur parallèle, ainsi que ses variantes `parfunN` pour appliquer à N vecteurs parallèles une fonction à N arguments.

Communications

Pour les communications entre processus, BSML propose deux primitives, ***put*** et ***proj***, qui provoquent également une synchronisation. Leur utilisation déclenche les deux dernières phases d'une super-étape.

La primitive ***put*** est utilisée pour échanger des données entre processeurs. Elle prend et retourne un vecteur de fonctions. Sa signature est ***put*** : (*int*→'a) *par* →(*int*→'a) *par*. Les fonctions du vecteur d'entrée décrivent, pour chaque processeur, les messages à envoyer aux autres processeurs. Une fois les communications effectuées, la primitive retourne un vecteur de fonctions encodant, à chaque processeur, les données reçues de tous les processeurs. La sémantique informelle de ***put*** est

$$\textbf{\textit{put}}\ \langle f_0, \ldots, f_{p-1}\rangle = \langle \lambda i.f_i 0, \ldots, \lambda i.f_i(p-1)\rangle.$$

Au processeur `i` , (f_i `j`) est le message à envoyer au processeur `j`. La fonction construite par ***put*** au processeur `i` associe à un numéro de processeur `j` la valeur reçue de celui-ci, c'est à dire (f_j `i`).

Lorsque aucune valeur ne doit être communiquée à un processeur `j`, (f_i `j`) doit être le premier constructeur sans argument d'un type union. Par exemple, pour le type `'a option = None | Some of 'a`, la valeur `None` représente l'absence de message, pour les listes on utilisera la liste vide `[]`, *etc.* La fonction encodant les messages reçus utilisera cette même valeur pour signifier qu'aucun message n'a été reçu pour un processeur donné.

Par exemple, si nous souhaitons diffuser la valeur disponible depuis un processeur à tous les autres processeurs, il faudra utiliser deux fonctions, l'une retournant la valeur à envoyer, quelque soit le processeur donné en paramètre, l'autre n'envoyant aucun message, à aucun processeur. La première fonction sera utilisée par le processeur à l'origine de la diffusion, l'autre par tous les autres processeurs. Pour pouvoir encoder l'absence de message tout en gardant une fonction de communication indépendante du type des éléments échangés, on encapsule ces derniers dans une liste. L'élément envoyé sera une liste singleton et l'absence de message sera représenté par la liste vide.

```
let choice root pid value =
  let toall = fun dst →[value]
  and nothing = fun dst →[] in
        if pid=root then toall else nothing
```

La fonction `choice`, pour une racine donnée (`root` : processeur contenant la valeur à diffuser) construit la fonction encodant les messages à envoyer depuis le processeur `pid`. Elle peut être utilisée pour construire un vecteur parallèle décrivant les communications

nécessaire à la diffusion d'une valeur (valeur `msg` dans la fonction broadcast ci-dessous). Ce vecteur sera donné en argument de la primitive ***put*** pour effectuer les communications.

Pour obtenir le message diffusé depuis `root` chaque processeur doit appliquer la fonction résultante de la communication (contenue dans le vecteur `recv`) au numéro de processeur `root`. En utilisant la fonction `List.hd`, qui renvoie la tête de la liste si elle contient un élément et lève une erreur sinon, on obtient l'élément initialement présent au processeur `root` à la place de la liste singleton reçue.

```
# let broadcast root data =
   let msg = apply (mkpar(choice root)) data in
   let recv=put msg in
   parfun List.hd (apply (recv) (replicate root));;
val broadcast : int →'a par →'a par = <fun>
```

Si on applique cette fonction de diffusion au vecteur `hello_i` défini précédemment, on obtient le résultat suivant :

```
# let v3 = broadcast 3 hello_i;;
val v3 : string par = <"Hello3", "Hello3", "Hello3",
                       "Hello3", "Hello3", "Hello3">
```

Pour mieux comprendre les différentes étapes, on remarquera que lors de la construction de `v3`, le vecteur `msg` a pris la valeur

```
<fun dst→[], fun dst→[], fun dst→[],
       fun dst→["Hello3"], fun dst→[], fun dst→[] >
```

et après l'application de ***put*** à ce vecteur, `recv` est un vecteur contenant en tout point la fonction

```
fun from →
match from with
   3 →["Hello3"]
   | _→[]
```

L'évaluation d'une application de ***put*** à un vecteur $\langle f_0, \ldots, f_{p-1} \rangle$ demande une super-étape complète : d'abord les messages à envoyer sont calculés de manière asynchrone, puis les messages sont communiqués, et enfin une synchronisation clôt la super étape afin de rendre disponible les messages échangés. Son coût est

$$\max_{0 \leq i < p} \Big(\sum_{0 \leq j < p} \overline{f_i\ j} \Big) + g \times \max_{0 \leq i < p} (max(h_i^+, h_i^-)) + L$$

où g est la bande passante du réseau, L est la latence nécessaire à une synchronisation, $\sum_{0 \leq j < p} \overline{f_i\ j}$ est le coût de calcul des messages à envoyer depuis le processeur i, h_i^+ est la taille totale des données envoyées par le processeur i, définie par $h_i^+ = \sum_{0 \leq j < p} |\, f_i\ j \,|$, et h_i^- est la taille des données reçues par le processeur i, $h_i^- = \sum_{0 \leq j < p} |\, f_j\ i \,|$. La notation $|\, f\ a \,|$ représente la taille de la valeur obtenue par l'application de f à a.

La primitive de communication ***proj*** de signature ***proj*** `: 'a` *par* `→(`*int*`→'a)` permet d'amener au niveau global du programme les valeurs contenues au niveau local dans un vecteur parallèle. C'est une sorte de fonction duale de la fonction ***mkpar***.

Cette primitive effectue un échange total des valeurs présentes dans le vecteur parallèle qui lui est donné, puis elle crée une fonction au niveau globale pour accéder à ces valeurs. Cette fonction associe à un numéro de processeur la valeur initialement présente à ce processeur dans le vecteur donné. Par exemple si on applique ***proj*** au vecteur `hello_i` on obtient la fonction suivante :

```
fun pid →match pid with
     0→ "hello0" | 1→ "hello1" | 2→ "hello2"
   | 3→ "hello3" | 4→ "hello4" | 5→ "hello5"
```

Le coût de l'évaluation de la primitive ***proj*** appliquée à un vecteur de valeurs x_i est

$$\max(\max_{0\leq i<p}(\sum_{j\neq i}|x_i|)\ ,\ \max_{0\leq i<p}(|x_i|\times(p-1)))+L.$$

3.2.3 Intérêts de ce langage dans notre contexte

Avec le langages Bulk Synchronous ML, nous n'avons pas à nous préoccuper de l'architecture sous-jacente, nous profitons d'un modèle de programmation simple et structuré pour écrire des programmes fonctionnels en utilisant explicitement le parallélisme d'une machine quasi-synchrone. Nos programmes sont déterministes et sans inter-blocage et leurs performances sont prévisibles si nous disposons des paramètres BSP de la machine que nous exploitons. La vue globale du programme parallèle apportée par le vecteur permet de raisonner clairement sur la localité des données.

Cette vue globale, le déterminisme des programmes ainsi que l'utilisation du paradigme de la programmation fonctionnelle permettent le plongement superficiel du langage BSML dans la logique de l'assistant de preuve Coq, c'est ce que nous présentons au chapitre suivant.

BSML en Coq

4

Sommaire

Ce chapitre présente la base de notre environnement de programmation parallèle certifiée. Cet environnement repose sur une axiomatisation de BSML en Coq, présentée dans la première section. Les sections suivantes sont consacrées aux modalités d'utilisation de cette axiomatisation pour définir des programmes BSML fortement spécifiés depuis l'assistant de preuve (sect. 4.2) et les exécuter dans l'assistant de preuve ou après extraction (sect. 4.3). Un tableau récapitulatif des différents modules Coq présentés ici se trouve à l'annexe B.

4.1 Axiomatisation de BSML

L'axiomatisation que nous présentons ici est un plongement superficiel du langage BSML dans Gallina. Ce plongement est forcément partiel car le langage de Coq n'est pas Turing-complet, certaines fonctions récursives de BSML ne pourront donc pas être plongées dans Gallina. Nous verrons cependant au cours de cette thèse qu'une grande variété de programmes peut être écrite dans ce langage. Notre implantation est une évolution des axiomatisations proposées par Gava [63] puis Gesbert [65]. L'implantation de Gava utilisait la version 7.3 de Coq, la syntaxe du langage a changé au passage à la version 8. Dans les versions de Gesbert et Gava, les axiomes ne sont pas encapsulés dans un module, or sans cette encapsulation il ne nous parait pas possible d'exécuter les programmes extraits sans les modifier. Gesbert propose une réalisation des axiomes mais

celle-ci nécessite l'axiome de *non-pertinence des termes de preuves*[1] qui affirme l'égalité de deux termes de même type. Bien que cet axiome soit cohérent avec la logique de Coq, sa combinaison avec d'autres axiomes peut rendre la logique incohérente.

Dans notre environnement, la sémantique de BSML est modélisée dans un type de module Coq nommé SPECIFICATION.PRIMITIVES. Cette modélisation est écrite de façon minimaliste, elle ne décrit que : la définition du type des noms de processeurs *processor*, le type *par* représentant les vecteurs parallèles ainsi qu'un descripteur *get* permettant d'accéder à la valeur d'un tel vecteur à un processeur donné, les axiomes décrivant les primitives du langage. Les propriétés intéressantes déductibles de cette modélisation sont décrites dans un type de module séparé, nommé PROPERTIES.TYPE.

Tous nos développements sont ensuite effectués dans des *types de modules* et non dans des modules afin de permettre les importations de module *en diamant* : si deux modules B et C importent un module A, et qu'un module D importe les modules B et C, le module A importé dans B sera connu comme B.A, et celui importé dans C comme C.A. Un élément *a* de A existera dans le module D sous la forme B.A.*a* et C.A.*a*, et Coq ne les reconnaît pas comme identiques. Pour pallier ce problème, il faut définir A comme un type de module, les modules B et C sont alors des foncteurs et D les appliquera à un module de type A. Cette fois Coq sera en mesure de reconnaître que l'élément *a* utilisé dans B et C vient du même module. Cet astuce alourdit la signature des modules, mais s'avère vite indispensable dans notre développement. Les types de module peuvent contenir les mêmes constructions que les modules.

Les modules décrivant des programmes parallèles seront donc des foncteurs (ou des types de foncteur) prenant en argument un module de type SPECIFICATION.PRIMITIVES et souvent une application à ce module d'un foncteur de type PROPERTIES.TYPE.

Les foncteurs contenant du code BSML devront ensuite être appliqués à un module contenant une réalisation de la sémantique pour être utilisés. En Coq, nous proposons une telle réalisation ne dépendant que d'un paramètre, un entier strictement positif représentant le nombre de processeurs, mais n'utilisant aucun autre axiome. Ce module est une implantation séquentielle des primitives BSML, les foncteurs seront donc rarement appliqués à celui-ci. En BSML, nous fournissons un module qui utilise les primitives parallèles du langage et dont la signature (le type du module) correspond à SPECIFICATION.PRIMITIVES. Un foncteur défini en Coq, une fois extrait, pourra être appliqué à ce module pour obtenir des programmes effectivement parallèles.

Le type de module SPECIFICATION.PRIMITIVES déclare en premier lieu l'existence d'un entier naturel, le nombre de processeurs de la machine BSP, et un axiome assurant que ce nombre est strictement positif :

```
Parameter bsp_p : nat.
Axiom bsp_pLtZero : 0 < bsp_p.
```

Ensuite, le type *processor* définit le type des noms de processeur. Ce sont des entiers strictement inférieurs à *bsp_p* :

1. En Anglais *proof irrelevance*

```
Definition processor : Type := { pid: nat | pid < bsp_p }.
```

Le type des vecteurs parallèles est

```
Parameter par : Type → Type.
```

C'est un type explicitement polymorphe, prenant en argument le type des valeurs contenues dans le vecteur. Pour décrire les valeurs locales d'un vecteur parallèle, une fonction *get* avec la spécification suivante est supposée définie :

```
Parameter get : ∀ A : Type, par A → processor → A.
```

Cette fonction a uniquement un rôle de descripteur permettant de raisonner sur le contenu d'un vecteur parallèle dans une spécification, elle ne doit pas être utilisée dans le corps des programmes parallèles car elle ne correspond à aucune primitive BSML. $(get\ V\ i)$ représente la valeur présente au $i^{\text{ème}}$ processeur du vecteur V. Le paramètre A est implicite, il est inféré par le système à partir du type de V. Ce paramètre sera omis dans la suite, toutes les primitives sont polymorphes et dépendent d'un tel type.

Notre modélisation donne la sémantique formelle des primitives via une description des valeurs présentes à chaque processeur après application de la primitive en question.

La création de vecteur parallèle est décrite ainsi :

```
Parameter mkpar :
  ∀ f : processor → A,
    { V: par A | ∀ i : processor, get V i = f i }.
```

Le résultat de l'application de *mkpar* à une fonction f produit un vecteur parallèle V accompagné d'une preuve qu'à tout processeur i, V a pour valeur $f\ i$.

De même l'application point à point :

```
Parameter apply :
  ∀ (B : Type) (vf : par ( A→ B)) (vx : par A),
    { V: par B | ∀ i : processor, get V i = (get vf i) (get vx i) }.
```

et les primitives de communication :

```
Parameter put :
  ∀ (vf : par (processor → A)),
    { V: par (processor → A) | ∀ i j : processor, get V i j = get vf j i }.

Parameter proj :
  ∀ (v : par A),
    { V: processor → A | ∀ i : processor, V i = get v i }.
```

sont décrites par leur signature. Hormis le typage fort de la valeur de retour des primitives, leur signature correspond bien à celle de l'implantation du langage BSML. Il est impossible de calculer le résultat de ces fonctions puisqu'elles sont ici définies sous forme d'axiomes, cependant le type de retour permet de spécifier fortement le résultat que l'on attendrait d'une implantation de ces fonctions. Cette spécification nous permet

de raisonner sur les résultats des programmes BSML, comme nous le verrons à la section 4.2.2.

Le module PROPERTIES.TYPE contient divers lemmes permettant de manipuler les représentations des processeurs. Il définit également une base de réécriture permettant l'évaluation symbolique des primitives BSML à l'aide de leur spécification. Un exemple d'utilisation des primitives et de leur spécification est montré dans la section suivante.

4.2 ÉCRITURE DE PROGRAMMES PARALLÈLES ET CORRECTION LOCALE

4.2.1 Utilisation des primitives

Pour être en mesure d'écrire un programme BSML il faut avant tout se placer dans un module paramétré par la spécification des primitives :

```
Module Foo (Bs : Specification.PRIMITIVES)(Props :Properties.TYPE  Bs) .
```

Le module SPECIFICATION.PRIMITIVES apporte les primitives présentées ci-dessus. Bien que le module PROPERTIES.TYPE ne soit pas *stricto sensu* nécessaire pour définir des programmes BSML, il devient rapidement indispensable dès qu'il s'agit de raisonner sur ceux-ci.

Pour éliminer la spécification du résultat obtenu par l'application d'une primitive et ne garder que le vecteur résultat (ou la fonction dans le cas de *proj*) il suffit d'appliquer une projection sur le premier élément (*proj1_sig*). La projection sur le deuxième élément (*proj2_sig*) fournit un terme de preuve spécifiant ce résultat par rapport aux paramètres d'entrée. Pour *mkpar* par exemple, la projection sur le premier élément, *proj1_sig*(*mkpar* f), avec f de type *processor* $\rightarrow A$, nous donne le vecteur parallèle résultat, de type *par* A, tandis que la projection sur le deuxième élément, *proj2_sig*(*mkpar* f), donne un terme spécifiant le résultat, de type $\forall\ (i : \textit{processor}),\ \textit{get}\ (\textit{proj1\_sig}\ (\textit{mkpar}\ f))\ i = f\ i$.

Les primitives manipulent des vecteurs parallèles, sans spécification. Lorsqu'on compose ces primitives il faut donc intercaler entre chaque appel une projection sur le premier élément du résultat de l'application d'une primitive afin que celui-ci soit compatible avec le type de paramètre attendu par la fonction suivante. Nous utilisons `Program` pour faire ceci automatiquement, ainsi nous pouvons continuer à écrire nos programmes comme nous le ferions en BSML.

Prenons l'exemple simple d'un programme appliquant en tout point la fonction identité à un vecteur contenant à chaque processeur l'identifiant de celui-ci. Sa définition à l'aide de `Program` est celle que l'on donnerait en BSML :

```
Program Definition foo:=
        apply (mkpar (fun _ a => a)) (mkpar (fun proc => proc)).
```

Cependant si on regarde le terme produit, on peut voir que des projections ont été insérées :

```
Print  foo.
foo =
apply (proj1_sig (mkpar (fun _ a : processor => a)))
  (proj1_sig (mkpar (fun proc : processor => proc)))
     : {X : par processor |
       forall i : processor,
       get X i =
       get (proj1_sig (mkpar (fun _ a : processor => a))) i
         (get (proj1_sig (mkpar (fun proc : processor => proc))) i)}
```

En utilisant les spécifications des primitives, il est possible de réécrire un terme utilisant le contenu d'un vecteur à un processeur donné et donc d'obtenir une évaluation symbolique locale des primitives BSML.

Par exemple, pour prouver qu'à tout processeur le résultat de *foo* est bien l'identifiant du processeur local, posons la proposition suivante :

```
  Goal forall proc,
         get (proj1_sig foo) proc = proc.

  intros proc ; unfold foo.
1 subgoal

  proc : processor
  ============================
   get
     (proj1_sig
        (apply (proj1_sig (mkpar (fun _ a : processor => a)))
           (proj1_sig (mkpar (fun proc0 : processor => proc0))))) proc =
   proc
```

Commençons par utiliser la spécification de *apply*.

```
  rewrite (proj2_sig (apply _ _)).
1 subgoal

  proc : processor
  ============================
   get (proj1_sig (mkpar (fun _ a : processor => a))) proc
     (get (proj1_sig (mkpar (fun proc0 : processor => proc0))) proc) =
   proc
```

On obtient bien, après réécriture, l'application de la fonction présente au processeur *proc* à la valeur présente au même processeur dans le deuxième vecteur.

Il suffit maintenant d'utiliser la spécification de *mkpar* pour simplifier l'expression :

```
  rewrite (proj2_sig (mkpar _)).
1 subgoal

  proc : processor
  ============================
   get (proj1_sig (mkpar (fun proc0 : processor => proc0))) proc = proc

  rewrite (proj2_sig (mkpar _)).
1 subgoal

  proc : processor
  ============================
   proc = proc
```

on obtient bien le résultat escompté.

Le module PROPERTIES.TYPE définit une base de réécriture *bsml* regroupant les spécifications des primitives. À l'aide de la tactique `autorewrite` cette base permet une évaluation symbolique locale puissante et simple d'utilisation. À titre d'exemple, reprenons la caractérisation précédente de la fonction *foo* :

```
1 subgoal

  proc : processor
  ============================
   get
     (proj1_sig
        (apply (proj1_sig (mkpar (fun _ a : processor => a)))
           (proj1_sig (mkpar (fun proc0 : processor => proc0))))) proc =
   proc

  autorewrite with bsml.
1 subgoal

  proc : processor
  ============================
   proc = proc
```

Plus besoin ici d'énoncer une à une les propriétés à appliquer, la réécriture se fait automatiquement.

4.2.2 Correction locale, exemples de la bibliothèque standard

Pour profiter pleinement de l'expressivité du langage Coq, il est souhaitable d'écrire tous les programmes sous une forme fortement spécifiée, où les valeurs du vecteur résultat attendues sont décrites dans le type de retour.

Cette spécification est construite à l'aide du descripteur *get* et donne une caractérisation locale du vecteur, comme pour la spécification des primitives du langage.

Les programmes que nous écrirons dorénavant, si leur valeur de retour est un vecteur parallèle, auront tous un type de retour de la forme `{ Vr : par A | ∀ (i :processor), get Vr i = SpecLocale }` où *SpecLocale* est une description du résultat attendu au processeur i dépendante des paramètres de notre programme.

Par exemple la fonction `replicate` de la bibliothèque `Bsmlbase`, qui construit un vecteur parallèle en répliquant une même valeur à tous les processeurs (définie en BSML à la section 3.2.2) est définie comme suit en Coq :

```
Definition replicate (A :Type)(a :A) :
  { Vr : par A | ∀ (i :processor), get Vr i = a } :=
  mkpar(fun _ ⇒ a).
```

Le type de retour est un vecteur parallèle accompagné d'une preuve que le vecteur contient à tout processeur la valeur à répliquer. Pour cette fonction très simple le type de retour indiqué est parfaitement compatible avec celui inféré pour le corps de la fonction et nous n'avons pas besoin des fonctionnalités de `Program`.

Partant de *replicate* il est à présent possible de définir *parfun*, qui applique une même fonction en tout point du vecteur :

```
Program Definition parfun(A B : Type)(f :A→B)(v :par A) :
 { vr: par B | ∀ (i :processor), get vr i = f (get v i) } :=
 apply (replicate f) v.

Next Obligation.
 autorewrite with bsml; reflexivity.
Defined.
```

Ici nous utilisons `Program` pour deux raisons :

Premièrement, comme pour notre premier exemple de programme, il faut insérer automatiquement les projections là où elles sont nécessaires.

Deuxièmement le type de retour de la fonction *parfun* porte une spécification qui n'est pas directement compatible avec le type de retour inféré par Coq à partir du corps de sa définition. En effet le type inféré pour le corps de la fonction *parfun* est

```
{X : par B | ∀ i : processor, get X i = get ( proj1_sig (replicate f)) i (get v i)}.
```

`Program` génère une obligation de preuve nous demandant de prouver que la spécification est déductible à partir de ce type. La résolution de cette preuve est une simple évaluation symbolique à l'aide de notre base de réécriture *bsml*.

Pour qu'à leur tour, les fonctions que nous implantons soient facilement réutilisables dans un contexte où l'on souhaite raisonner sur le contenu des vecteurs parallèles produits, leur spécification est elle aussi ajoutée à une base de réécriture (*bsmlbase* pour la bibliothèque `Base` définissant des combinaisons de primitives BSML fréquemment utilisées, *bsmlcomm* pour la bibliothèque `Comm` implantant divers schémas de communication, ...).

4.3 Exécution des programmes BSML

À présent, dans tout foncteur prenant en paramètre un module typé par la spécification des primitives BSML, nous pouvons écrire un programme BSML. Il ne pourra évidement pas être exécuté en parallèle depuis Coq, cependant une implantation séquentielle des primitives BSML a été réalisée et il est donc possible d'exécuter ces programmes dans l'assistant de preuve.

Cette implantation est un module ne dépendant que d'un paramètre, un entier strictement positif : *bsp_p*. Ce module respecte la signature Specification.PRIMITIVES et ne contient aucun axiome. Il s'agit donc d'une réalisation de notre axiomatisation de BSML.

Les vecteurs parallèles sont implantés par une structure de données de type vector. vector est un type inductif identique à une liste à ceci près que la taille est fixée par un des paramètres du type lui même. Ici nous fixons la taille du vecteur à *bsp_p* :

```
Definition par (A :Type) := vector A bsp_p.
```

Le descripteur *get* est défini par la fonction retournant le $\mathtt{i}^{\text{ème}}$ élément du vecteur.

La fonction *mkpar* f est une application de la fonction f en chaque point d'un vecteur contenant la séquence des processeurs (dans l'ordre croissant).

La fonction *apply* combine deux vecteurs en un vecteur de paires d'éléments (fonction *combine*) puis applique, en tout point, le premier élément au deuxième (à l'aide d'une fonction map).

La fonction *put* est implantée comme une transposition de matrice. La matrice des éléments à envoyer est une matrice carrée, implantée par un vecteur de taille *bsp_p* contenant des vecteurs de taille *bsp_p*. En tout point du vecteur, le vecteur des valeurs à envoyer est construit en appliquant la valeur (fonction) locale à tous les points d'un vecteur contenant la séquence des noms de processeurs. La matrice des valeurs transmises est ensuite transposée et *put* renvoie un vecteur de fonctions, chacune associant à un numéro de processeur i la $\mathtt{i}^{\text{ème}}$ valeur du vecteur local (*reçue* de i).

La fonction *proj* est exactement définie par *get*, bien qu'ils soient très différents dans l'intention. *proj* est une primitive du langage qui effectue un échange total des données et renvoie une fonction indexant ces données par leur numéro de processeur ; *get* est un axiome de la modélisation permettant de raisonner sur les valeurs présentes en chaque point d'un vecteur parallèle.

Ainsi toutes les primitives du langage sont implantées et peuvent être utilisées pour effectuer un calcul. Le fait qu'une telle réalisation de notre axiomatisation existe nous

assure que celle-ci n'introduit pas d'incohérence dans la logique de Coq. Il est maintenant possible d'exécuter séquentiellement les programmes BSML dans l'assistant de preuve.

Par exemple, si l'on instancie le module Foo, contenant la fonction *foo* définie précédemment, avec notre implantation des primitives pour un nombre de processeurs fixé (ici 8), il est possible de calculer le résultat de cette fonction de la façon suivante :

```
Eval compute in (proj1_sig FooImplem.foo).
   = << 0 -| 1 -| 2 -| 3 -| 4 -| 5 -| 6 -| 7 - >>
   : BsmlImplementation_8.par BsmlImplementation_8.processor
```

4.3.1 Extraction

Ce qui nous intéresse est de pouvoir exécuter nos programmes en parallèle. Pour cela nous utilisons la fonctionnalité d'extraction de Coq, présentée à la section 3.1.6. qui permet d'obtenir un code OCaml correspondant à nos modules Coq, que nous compilons ensuite à l'aide du compilateur BSML. Lors de l'extraction nous avons la possibilité d'associer des structures de données Coq à leur équivalent OCaml. Ceci nous permet, par exemple, d'utiliser les listes et les booléens OCaml au lieu d'extraire les types inductifs correspondants définis en Coq. Cela permet de gagner en lisibilité et également en espace mémoire lors du calcul, les structures de données OCaml étant implantées de façon plus efficace.

Le code de la bibliothèque `Bsmlbase` et la version extraite depuis Coq sont très similaires. Le module Base extrait, tout comme le module `Bsmlbase` de la bibliothèque standard BSML, est un foncteur prenant en paramètre un module contenant les primitives BSML. Cependant, dans la version extraite, les processeurs doivent être désignés par des entiers de Peano là où le module BSML fournissant les primitives du langage utilise des entiers relatifs binaires codés sur 31 ou 62 bits suivant l'architecture sur laquelle est compilé le programme.

Un module BSML nommé `BsmlNat`, respectant la signature attendue par le foncteur Base, a été implanté pour faire le lien entre le code extrait depuis Coq et les signatures des primitives BSML. Le passage d'un codage de nom de processeur à l'autre se fait par une fonction de conversion utilisant un cache afin d'éviter un trop grand nombre de conversion. Cet encodage reste correct pour nommer les processeurs à condition de ne pas utiliser plus de 2^{31} (2147483648) processeurs ou 2^{62} (4611686018427387904) pour les machines 64 bits, une condition raisonnable pour quelques années encore. Le risque de dépasser ces limites augmente lorsqu'on effectue des calculs sur les numéros de processeur.

Performance des programmes BSML extraits

Le codage sous forme d'entier de Peano pour les noms de processeur a été choisie pour permettre l'utilisation de tous les développements Coq existant autour de l'arithmétique

sur ceux-ci. Cette simplification des preuves se paie par un sur-coût à l'exécution dès que des opérations ont lieu sur les processeurs.

Nous avons mesurés sur le Très Grand Centre de Calcul Curie les différences de temps d'exécution entre un décalage à droite circulaire des valeurs d'un vecteur parallèle, écrit en BSML et le programme équivalent extrait depuis Coq, utilisant les primitives BSML implantées dans `BsmlNat`. En utilisant jusqu'à 2048 processeurs pour des listes de petite taille, nous n'avons pas mesuré de différence de temps d'exécution, l'impact de l'encapsulation des primitives dans le module `BsmlNat` est donc négligeable.

4.3.2 Tests de la chaîne de production

À ce stade il convient tout de même de s'interroger sur la correction des programmes extraits.

Premièrement, le code extrait dépend du module `BsmlNat` qui n'est pas certifié ; par exemple, la fonction *get* est implantée par une fonction levant toujours une exception. Ainsi, la *fuite* dans un code extrait de cette fonction réservée à la spécification et au raisonnement sur les programmes ne provoquera pas un échange total, l'utilisateur sera informé d'un usage non-conforme de cette fonction. C'est pour le moment au programmeur de vérifier qu'il ne l'utilise pas dans son développement. Ceci pourrait cependant être vérifié automatiquement sans grande difficulté par analyse syntaxique du code.

Ensuite l'extraction des programmes de Coq vers OCaml n'est pas une opération certifiée, des erreurs sont donc susceptibles d'apparaître à ce stade. Des travaux en cours dans l'équipe de développement de Coq visent à certifier cette opération [67]. De plus, le compilateur de BSML n'est pas certifié, la conversion du code source au code machine n'est donc pas une opération sûre.

La spécification en Coq des primitives BSML ne représente donc que ce que nous avons estimé être leur sémantique, mais rien ne la lie formellement à l'implantation du langage BSML en OCaml ni au code binaire obtenu après compilation.

Pour augmenter notre confiance dans la chaîne de production des programmes, nous avons procédé par test, en comparant les résultats d'exécutions dans l'assistant de preuve avec les résultats obtenus pour le même programme exécuté en utilisant les primitives implantées dans `BsmlNat`.

Cette opération se fait en trois étapes : d'abord des fonctions à tester sont écrites dans un module Coq paramétré par les spécifications de BSML ; ensuite une instance de ce module est créée à l'aide de l'implantation séquentielle des primitives en Coq et le résultat de chacune des fonctions est évalué dans l'assistant de preuve ; enfin l'ensemble des résultats ainsi calculés est passé à un troisième module, celui-ci est paramétré par la spécification de BSML et par une fonction d'affichage. Après extraction, la valeur calculée séquentiellement dans Coq est comparée à celle calculée en parallèle, et le résultat de la comparaison est affiché. La comparaison, ainsi que l'affichage, seront effectués au moment de l'exécution en parallèle du module extrait, qu'on aura instancié à l'aide de `BsmlNat`.

Ces comparaisons testent à la fois la chaîne de production du fichier binaire exécutable et l'axiomatisation proposée. Les seuls éléments écrits hors de Coq pour ces tests sont le module contenant la fonction d'affichage, le module `BsmlNat` ainsi que l'instanciation

des modules de tests. Tous les tests effectués ont donné les résultats attendus, ce qui ne prouve rien formellement, mais permet de penser que notre axiomatisation correspond bien au comportement des primitives.

4.4 Conclusion

Notre axiomatisation propose un plongement superficiel de BSML dans Coq. Il est à présent possible de définir des programmes BSML dans l'assistant de preuve et de raisonner sur leur sémantique à l'aide de la forte spécification de leur type de retour. Ici, l'intérêt d'un plongement superficiel de notre langage dans le langage de Coq est évident : la partie fonctionnelle séquentielle du langage BSML est exprimable directement et peut être extraite directement en OCaml. Ainsi, nous avons gratuitement à notre disposition, pour décrire des programmes BSML dans l'assistant de preuve, les structures d'enregistrement, les fonctions d'ordre supérieur, les structures de module, etc. Ce que nous perdons par rapport à un plongement profond, c'est la possibilité de raisonner et de calculer directement sur la structure des programmes, c'est à dire leur arbre de syntaxe abstraite. De tels raisonnements sont intéressants, par exemple, si l'on souhaite obtenir des garanties sur les consommations de ressources (temps, mémoire). Cependant, dans notre cadre, nous recherchons principalement à nous assurer que les programmes que nous écrivons produisent des résultats corrects, nous n'avons pas, pour cela, besoin de raisonner sur la structure des programmes.

Par rapport à celles de Gava et Gesbert, notre axiomatisation est mieux adaptée à l'extraction de programmes grâce à sa structuration sous forme de modules, de plus l'axiomatisation est encapsulée dans un type de module minimaliste, réduisant au minimum le code non-certifié qui devra être utilisé pour la parallélisation du code extrait. Enfin une implantation séquentielle en Coq de ce type de module a été réalisée afin d'assurer la cohérence de l'axiomatisation. Contrairement à celles proposées par Gava et Gesbert, notre réalisation a été faite sans aucun axiome. De plus nous avons utilisé cette implantation séquentielle certifiée pour valider expérimentalement la modélisation.

Nous avons vu dans ce chapitre comment définir dans l'assistant de preuve des programmes parallèles BSML à l'aide de notre plongement superficiel, comment les exécuter et comment prouver des propriétés locales sur les éléments des vecteurs parallèles. Cependant cette description locale des vecteurs est-elle la plus à même d'exprimer clairement la correction d'un programme ? Quand on parle de la correction d'un programme, il est question de son comportement global. Bien sûr, il est toujours possible de raisonner sur le résultat global d'un programme parallèle à partir des composants du vecteur résultat mais il s'agit d'une tâche complexe.

Une approche plus simple pour appréhender le comportement d'un programme parallèle est de considérer le vecteur résultat comme une vue particulière d'une structure de données séquentielle, et le programme parallèle comme l'équivalent d'une fonction (séquentielle) transformant cette structure de données. C'est l'approche que nous présentons au chapitre suivant.

Parallélisation correcte

5

Sommaire

Au chapitre précédent, nous avons mis en place les outils nécessaires pour définir des programmes BSML dans l'assistant de preuve. Nous pouvons à présent raisonner sur leurs résultats en observant les valeurs présentes à chaque processeur.

Cependant, raisonner ainsi sur les résultats des programmes parallèles impose de prendre en compte la localité des éléments répartis. Ces détails peuvent nuire à la compréhension globale du programme et compliquent les preuves. Lorsqu'il n'est pas nécessaire de prendre en compte la répartition des données, il est utile de pouvoir s'abstraire du parallélisme et de raisonner sur des structures de données séquentielles. Une fois clairement spécifié le résultat séquentiel attendu, il faut cependant pouvoir le lier au résultat d'un programme parallèle.

Nous proposons dans ce chapitre de formaliser le lien entre les fonctions séquentielles utilisant des structures de données séquentielles et les programmes parallèles manipulant des structures réparties. Les fonctions séquentielles servent ici de spécifications, décrivant les valeurs attendues, et les programmes parallèles sont des réalisations de ces spécifications, parallélisant le calcul.

5.1 Spécification séquentielle, programme parallèle

5.1.1 Répartition des données

Pour paralléliser un algorithme séquentiel, il faut en général répartir les données sur lesquelles il calcule. Il nous faut donc une notion de répartition de structure de données. La répartition d'une structure de données de type A doit pouvoir se faire par une fonction totale vers un type $A_{/\!/}$. Ce type parallèle peut être une composition de plusieurs types, dont l'un au moins est un type de vecteur parallèle de la forme $(par\ A_L)$ [1]. C'est ce dernier qui donne son caractère parallèle au type $A_{/\!/}$.

Nous prendrons ici l'exemple de la fonction *filter* qui prend une fonction *select* de type $E \to \mathsf{bool}$ et une liste (de type $\mathsf{list}\ E$) et construit une liste de même type en éliminant de la liste initiale toutes les valeurs pour lesquelles *select* renvoie false. Voici sa définition

```
Fixpoint filter (A : Type) (f : A → bool) (l : list A) : list A :=
  match l with
  | nil ⇒ nil
  | x :: l0 ⇒ if f x then x :: filter A f l0 else filter A f l0
  end.
Implicit Arguments filter [A].
```

Pour cet exemple, la structure de données manipulée est une liste, il nous faut donc une fonction de répartition des listes. Nous ne détaillons pas ici le code de la fonction que nous utilisons, nous nous contentons d'indiquer que cette fonction, que nous nommons $partition_{\mathrm{list}}$, découpe une liste en p morceaux de même taille (à un élément près) et les répartit sur les p processeurs, dans l'ordre croissant de leurs identifiants. Cette répartition est schématisée par la figure 5.1.

5.1.2 Parallélisation correcte

Une fois la liste répartie, nous utilisons la fonction suivante pour calculer *filter* en parallèle :

```
Program Definition filterPar (E :Type) (f :E → bool) (pl : par(list E)) : par (list E) :=
  parfun (filter f) pl.
```

Si nous disposons d'une fonction de répartition pour un type A, appelons la $partition_A$, nous pouvons calculer en parallèle sur les mêmes données qu'une spécification traitant des données de type A. Il suffit de répartir les données avant le calcul. Cependant le résultat de notre programme parallèle sera certainement une structure de données répartie, or pour vérifier sa correction, nous allons devoir comparer cette valeur

1. où A_L est un type séquentiel quelconque.

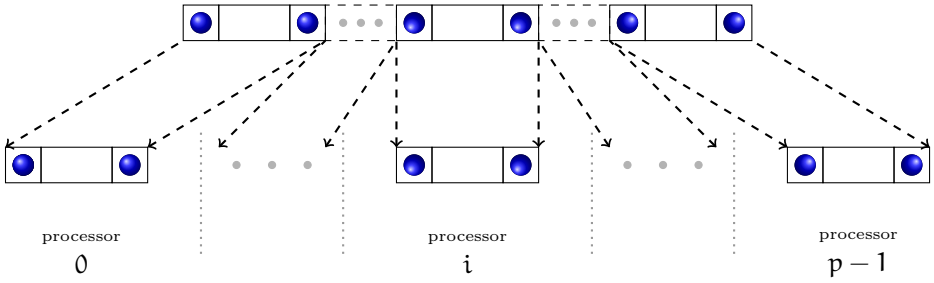


FIGURE 5.1 – *Partitionnement d'une liste séquentielle*

au résultat de la spécification séquentielle. Si le type résultat de la spécification a lui-même une fonction de répartition, nous pouvons répartir ce résultat et considérer comme corrects uniquement les programmes parallèles fournissant un résultat équivalent.

Une formulation de ceci consiste à dire que le diagramme suivant doit commuter, c'est à dire que pour tout $a : A$, $\mathit{partition}_B \circ f\ a = f_{/\!/} \circ \mathit{partition}_A\ a$.

$$\begin{array}{ccc} A_{/\!/} & \xrightarrow{f_{/\!/}} & B_{/\!/} \\ {\scriptstyle \mathit{partition}_A}\uparrow & & \uparrow{\scriptstyle \mathit{partition}_B} \\ A & \xrightarrow{f} & B \end{array}$$

Ce raisonnement n'est cependant valide que si une donnée séquentielle n'a qu'une répartition possible.

Dans le cas de nos fonctions *filter* et *filterPar*, il est évident, vu l'implantation de *filterPar*, que le résultat ne sera pas toujours une liste équitablement répartie entre les processeurs. Il suffit que plus d'éléments soient éliminés sur un processeur que sur un autre pour déséquilibrer la répartition. Le résultat de *filterPar* ne sera donc pas égal à la répartition du résultat de *filter* car notre fonction de répartition des listes ne construit que des listes équitablement réparties. Pour rendre correct notre programme *filterPar* il faudrait le composer avec une fonction de rééquilibrage afin de construire une forme normale de liste parallèle compatible avec la fonction de répartition.

Pour éviter qu'un simple déséquilibre dans le résultat nous empêche de montrer correct notre résultat parallèle, nous allons le séquentialiser avant de le comparer au résultat de la spécification séquentielle. La fonction de séquentialisation d'un type $A_{/\!/}$, que nous appellerons join_A, devra être totale et invariante à la distribution. Nous imposons de plus que $\mathit{join}_A \circ \mathit{partition}_A = \mathit{id}_A$. Cette contrainte permet d'assurer que la fonction identité sur $A_{/\!/}$ ($\mathit{id}_{A_{/\!/}}$) soit une parallélisation correcte de l'identité sur A (id_A). Pour notre exemple, nous avons besoin d'une fonction de séquentialisation des listes réparties :

```
Program Definition listOfParList (A :Type)(v :par (list A)) : list A :=
  flatten (map (get v) processors).
```

La figure 5.2 schématise l'opération effectuée par *listOfParList*. Cette fonction construit

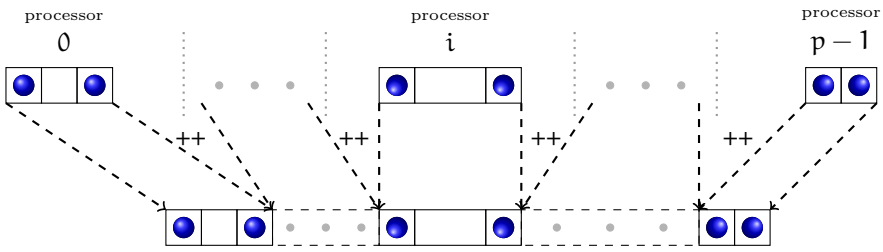


FIGURE 5.2 – *Séquentialisation d'une liste répartie*

la liste des listes présentes à chaque processeur, dans l'ordre croissant des numéros de processeurs, puis les concatène à l'aide de *flatten*.

La fonction *listOfParList* utilise *get* et non *proj* car nous l'utiliserons dans notre cadre en Coq pour spécifier des programmes parallèles. Séquentialiser une liste répartie est une opération coûteuse en temps de calcul qui doit être effectuée en pleine connaissance de cause, il n'est donc pas souhaitable qu'une telle opération soit utilisée par inadvertance. Si le programmeur souhaite réellement séquentialiser une liste répartie (c'est-à-dire en faire une valeur répliquée et donc effectuer un échange total des données), il devra le faire en utilisant explicitement une fonction faite pour cela, implantée à l'aide de *proj*.

Nous pouvons maintenant définir la correction d'une parallélisation comme la commutation du diagramme suivant :

$$\begin{array}{ccc} A_{/\!/} & \xrightarrow{f_{/\!/}} & B_{/\!/} \\ {\scriptstyle partition_A}\uparrow & & \downarrow{\scriptstyle join_B} \\ A & \xrightarrow{f} & B \end{array}$$

Autrement dit, $f = join_B \circ f_{/\!/} \circ partition_A$.

L'identité sur $A_{/\!/}$ est dans ce contexte une parallélisation correcte de l'identité sur A. Mais qu'en est-il de la composition de fonctions ? Supposons que $f : A \to B$ est correctement parallélisée par $f_{/\!/}$ et $g : B \to C$ est correctement parallélisée par $g_{/\!/}$. Nous souhaiterions pouvoir dire que $g_{/\!/} \circ f_{/\!/}$ est une parallélisation correcte de $g \circ f$. Or nous avons :

$g \circ f$
$=$ {par définition de la parallélisation correcte de f}
$g \circ join_B \circ f_{/\!/} \circ partition_A$
$=$ {par définition de la parallélisation correcte de g}
$join_C \circ g_{/\!/} \circ partition_B \circ join_B \circ f_{/\!/} \circ partition_A$

Pour en déduire que $g_{/\!/} \circ f_{/\!/}$ est une parallélisation correcte de $g \circ f$, il faut soit avoir $partition_B \circ join_B = id_{B_{/\!/}}$, soit montrer que $partition_B \circ join_B \circ f_{/\!/} \circ partition_A = f \circ partition_A$, soit montrer de façon ad-hoc que $f_{/\!/} \circ g_{/\!/}$ est une parallélisation correcte de $f \circ g$.

Avoir $partition_B \circ join_B = id_{B_{/\!/}}$ signifie que les éléments de B ont une seule répartition possible, ce qui est beaucoup trop restrictif. Il n'est possible de montrer

$\mathit{partition}_B \circ \mathit{join}_B \circ f_{/\!/} \circ \mathit{partition}_A = f \circ \mathit{partition}_A$ que si $f_{/\!/}$ produit un résultat dont la répartition est toujours identique à celle obtenue par la fonction $\mathit{partition}_B$. Nous avons vu que dans le cas de notre fonction *filterPar* ce n'est pas le cas. Si une telle preuve n'est pas possible, il ne reste plus que la possibilité de faire une preuve ad-hoc à chaque composition avec une nouvelle fonction.

De telles restrictions ne sont pas souhaitables, nous allons donc généraliser notre propriété de parallélisation correcte afin que celle-ci soit préservée par la composition.

5.1.3 Définition d'une relation de parallélisation correcte composable

Pour cela nous imposons qu'une parallélisation soit correcte quelque soit la distribution des données en entrée, ce que nous exprimons par la définition suivante :

Définition 1 (Parallélisation correcte composable) *La parallélisation $f_{/\!/} : A_{/\!/} \rightarrow B_{/\!/}$ d'une fonction $f : A \rightarrow B$ est correcte (au sens composable) si pour tout $a_{/\!/}$ de type $A_{/\!/}$, on a $(\mathit{join}_B \circ f_{/\!/})\ a_{/\!/} = (f \circ \mathit{join}_A)\ a_{/\!/}$.*

Autrement dit, le diagramme suivant commute :

$$\begin{array}{ccc} A_{/\!/} & \xrightarrow{f_{/\!/}} & B_{/\!/} \\ {\scriptstyle \mathit{join}_A}\downarrow & & \downarrow{\scriptstyle \mathit{join}_B} \\ A & \xrightarrow{f} & B \end{array}$$

Cette relation de parallélisation correcte entre f et $f_{/\!/}$ sera notée $f \mathrel{/\!/\!\!\Longrightarrow} f_{/\!/}$. Cette notion de correction généralise la relation vue précédemment :

Lemme 5.1 *(généralisation) Pour toutes fonctions f et $f_{/\!/}$, si $f \mathrel{/\!/\!\!\Longrightarrow} f_{/\!/}$ alors pour tout $a : A$, $\mathit{join}_B \circ f_{/\!/} \circ \mathit{partition}_A\ a = f\ a$.*

Démonstration

$f \mathrel{/\!/\!\!\Longrightarrow} f_{/\!/}$

$\equiv$ {par définition de $\mathrel{/\!/\!\!\Longrightarrow}$}

$\forall\ a_{/\!/} : A_{/\!/},\ (\ \mathit{join}_B \circ f_{/\!/}\ )\ a_{/\!/} = (\ f \circ \mathit{join}_A\ )\ a_{/\!/}$

$\Longrightarrow$ {par définition, $\mathit{partition}_A$ étant totale, on a $\forall\ a : A,\ \exists\ a_{/\!/} : A_{/\!/} |\ a_{/\!/} = \mathit{partition}_A\ a$ }

$\forall\ a : A,\ (\ \mathit{join}_B \circ f_{/\!/}\ )\ (\ \mathit{partition}_A\ a\ ) = (\ f \circ \mathit{join}_A\ )\ (\ \mathit{partition}_A\ a\ )$

$\equiv$ {par associativité de la composition }

$\forall\ a : A,\ (\ \mathit{join}_B \circ f_{/\!/} \circ \mathit{partition}_A)\ a = (\ f \circ \mathit{join}_A \circ \mathit{partition}_A)\ a$

$\equiv$ {par construction $\mathit{join}_A \circ \mathit{partition}_A = \mathit{id}_A$ }

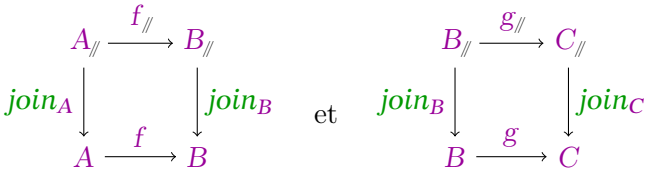
$\forall\ a : A,\ (\ \mathit{join}_B \circ f_{/\!/} \circ \mathit{partition}_A\ )\ a = (\ f \circ \mathit{id}_A\ )\ a\ )$

$\equiv$ {par définition de l'identité }

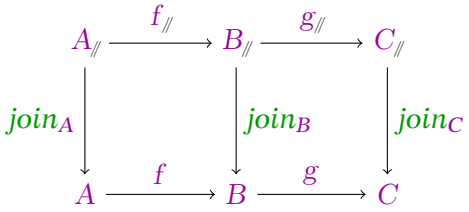
$\forall\ a : A,\ (\ \mathit{join}_B \circ f_{/\!/} \circ \mathit{partition}_A\ )\ a = f\ a$

□

Ainsi notre notion de parallélisation correcte est une généralisation de la précédente. Dans la suite du document, nous utiliserons le terme de parallélisation correcte pour désigner la relation de correction composable, sauf si nous précisons explicitement qu'il s'agit de correction *non composable*. Montrons maintenant que la correction est préservée par composition : la preuve est directe par composition de diagramme car on sait que si



commutent, alors leur composition commute :



la preuve peut aussi être faite ainsi :

$g \mathrel{\#\!\!\Longrightarrow} g_{/\!/}$
$\equiv$ {par définition de la relation $\mathrel{\#\!\!\Longrightarrow}_{/\!/}$}
$join_C \circ g_{/\!/} \circ f_{/\!/} = g \circ f \circ join_A$
$\equiv$ {par la relation $g \mathrel{\#\!\!\Longrightarrow} g_{/\!/}$}
$g \circ join_B \circ f_{/\!/} = g \circ f \circ join_A$

$\equiv$ {par la relation $f \mathrel{\#\!\!\Longrightarrow} f_{/\!/}$}
$g \circ f \circ join_A = g \circ f \circ join_A$
□

5.1.4 Support en Coq

Déclarer l'existence des fonctions de répartition et de séquentialisation se fait en Coq par la définition d'une instance de la classe de type Partitionnable. Cette classe est paramétrée par le type d'une structure de données séquentielle et contient quatre champs correspondant au type de la structure de données parallèle, aux deux fonctions de répartition et de séquentialisation et enfin la preuve de *compatibilité* de ces deux fonctions, assurant que $join_A \circ partition_A = id_A$:

```
Class Partitionnable (A : Type) :=
{
  parallel_type : Type;
  join : parallel_type → A;
  partition : A → parallel_type;
  join_part_match : ( ∀ a : A, join ( partition a ) = a )
}.
```

Comme il n'est pas possible en Coq de raisonner sur la structure d'un type, nous ne pouvons pas imposer que le type parallèle déclaré *contienne* effectivement un type réparti, de la forme *par A*. Il revient au programmeur de décider qu'un type est parallèle.

Cette définition est accompagnée d'une notation : le type parallèle correspondant à un type A, désigné par le champ *parallel_type* d'une instance de Partitionnable est noté $//A\backslash\backslash$.

La relation de parallélisation correcte entre une fonction séquentielle f et une fonction parallèle $f_{/\!/}$ se déclare à l'aide d'une instance de la classe suivante :

```
Class Parallel {A B : Type} {Ap : Partitionnable A} {Bp : Partitionnable B}
                            (f : A → B)
                            (f// : //A\\ → //B\\)
 :={
  parallel_spec_match : ∀ a//, join ( fp a//) = f (join a//)
}.
```

L'existence d'une instance de cette classe paramétrée par f et $f_{/\!/}$ nous assure que $f_{/\!/}$ est bien une parallélisation correcte de f. Le caractère *parallèle* de la fonction $f_{/\!/}$ est donné par le fait qu'elle manipule des éléments dont le type est déclaré parallèle (par les instances *Ap* et *Bp*). Comme nous faisons référence au type des arguments de la fonction, il n'est pas possible de les représenter pour une arité de fonction quelconque, nous nous limitons donc à des fonctions d'arité un. Il est néanmoins possible de décurryfier les fonctions afin de les rendre compatibles avec cette classe. Nous avons défini des instances de la classe Partitionnable permettant de construire automatiquement une instance de Partitionnable pour un type construit par le produit de plusieurs types dont l'un au moins est Partitionnable, ainsi le type $(A_1 \times A_2 \times A_3)$ sera associé au type parallèle $(A_1 \times A_{2/\!/} \times A_3)$ si le type A_2 est partitionnable ou plus prioritairement à $(A_1 \times A_{2/\!/} \times A_{3/\!/})$ si A_3 est également partitionnable. Décurryfier les fonctions nuit cependant à la lisibilité et quand un seul des paramètres est une donnée répartie, nous préférerons construire une instance de Parallel paramétrée par les valeurs répliquées, comme nous le faisons dans la suite pour *filter*.

Pour revenir à notre exemple, la répartition du type des listes est déclarée comme suit [2] [3] :

```
Instance list_partitionnable {A : Type} : Partitionnable (list A) :={
    parallel_type := par (list A);
      join := (fun l ⇒ listOfParList l);
      partition := (fun l ⇒ partitionList l)
}.
```

Il nous faut maintenant prouver que pour toute fonction *select*, *filter* est correctement parallélisé par *filterPar* : (*filter select*) $\Longrightarrow\!\!\!\!\!\!/$ (*filterPar select*). L'idée de la preuve est de raisonner par induction sur la liste des sous-listes présentes à chaque processeur. En effet, si l'on concatène les n premières listes locales résultantes de l'application de *filterPar*, on obtient bien le même résultat que si on applique *filter* à la concaténation des n premières listes locales initiales.

Une telle induction n'est pas possible directement car *listOfParList* utilise la liste de tous les processeurs, se terminant sur le dernier processeur, défini à partir de *bsp_p* -1. On ne peut raisonner directement par induction sur ce numéro de processeur. Cela génère des erreurs de typage car d'autres termes dans l'environnement ont un type dépendant de *bsp_p*.

Il est donc nécessaire de construire un lemme généralisant le lemme courant, en remplaçant la liste de tous les processeurs par la liste de processeurs jusqu'à un certain processeur *i*. Notre lemme généralisé sera quantifié universellement par *i*. Dans la preuve de ce lemme, il sera possible de raisonner par induction sur l'entier correspondant à *i*. Une fois le lemme prouvé, il suffit de l'appliquer en prenant pour le paramètre *i* la valeur *lastProcessor* qui correspond au dernier processeur.

Ce schéma de raisonnement est courant pour les preuves de correction d'une parallélisation de fonction sur les listes. Nous avons donc implanté une tactique qui génère automatiquement le lemme généralisé à partir d'un lemme utilisant *listOfParList*, et lance une induction sur le numéro de processeur.

Il est maintenant possible de montrer que notre fonction *filterPar* est une parallélisation correcte de la fonction *filter*, en définissant l'instance suivante :

```
Instance FilterParCorrect (E :Type)(select :E→bool) :Parallel(filter select)(filterPar select).
```

La fonction *filter* a pour type d'entrée et de sortie le type (list E), les paramètres *Ap* et *Bp* de notre classe étant implicites et maximalement insérés, l'instance *list_partitionnable* est automatiquement utilisée pour définir le type attendu pour la fonction parallèle,

2. Les eta-expansions utilisées ici sont nécessaires pour la compilation du code extrait. En effet, bien que ces fonctions ne soient pas utilisées par les programmes que nous définirons, elles apparaissent dans la bibliothèque de programmation extraite et doivent donc être compilées. Dans ce cas précis, si l'eta-expansion n'est pas utilisée le type du terme extrait est déclaré polymorphe dans l'interface extraite alors que le type inféré par OCaml est un type faible (non-polymorphe et donc non-compatible).

3. Pour compléter la définition de cette instance, Coq nous demande de prouver que *join* ∘ *partition*=*id*. Nous omettons ici la preuve.

qui ici s'unifie bien avec le type de *filterPar*. Nous commençons avec quelques tactiques permettant d'exhiber la présence de *listOfParList* :

```
  constructor;intros ap; simpl.
1 subgoal

  E : Type
  select : E -> bool
  ap : // list E \\
  ============================
   listOfParList (filterPar select ap) =
   filter select (listOfParList ap)
```

Puis nous utilisons notre tactique qui simule une induction sur une liste construite par morceaux.

```
  listOfParListInduction.
2 subgoals

  E : Type
  select : E -> bool
  ap : // list E \\
  pProc : 0 < bsp_p
  ============================
   get (filterPar select ap) firstProcessor =
   filter select (get ap firstProcessor)
subgoal 2 is:
 flatten (map (get (filterPar select ap)) (processorsUpTo p')) ++
 get (filterPar select ap) S_p =
 filter select
   (flatten (map (get ap) (processorsUpTo p')) ++ get ap S_p)
```

Le premier cas correspond à la sous-liste du premier processeur. La définition de la fonction *filterPar* est dépliée, et nous utilisons la base de réécriture de la bibliothèque standard de BsmlBase pour effectuer une évaluation symbolique de *parfun*. Le sous-but est ensuite résolu par `reflexivity`.

```
  unfold filterPar;
  autorewrite with  bsmlbase.
2 subgoals

  E : Type
  select : E -> bool
  ap : // list E \\
  pProc : 0 < bsp_p
  ============================
   filter select (get ap firstProcessor) =
   filter select (get ap firstProcessor)
subgoal 2 is:
 flatten (map (get (filterPar select ap)) (processorsUpTo p')) ++
 get (filterPar select ap) S_p =
 filter select
   (flatten (map (get ap) (processorsUpTo p')) ++ get ap S_p)

  reflexivity.
```

Dans le deuxième sous-but généré, l'égalité est supposée prouvée pour la concaténation des p' premières sous-listes des listes réparties (hypothèse *IHp*'). Il nous est demandé de prouver l'égalité pour la concaténation des p'+1 premières sous-listes. Nous utilisons un lemme *filter_app* pour distribuer *filter* sur la concaténation. La tactique `f_equal` nous propose de prouver que les arguments de la concaténation sont les mêmes de chaque coté de l'égalité. Les arguments gauches correspondent à l'hypothèse d'induction *IHp*' que nous appliquons, ceux de droite sont montrés égaux de la même façon que pour le premier processeur.

```
  rewrite filter_app.
1 subgoal

  E : Type
  select : E -> bool
  ap : // list E \\
  p : nat
  pProc : S p < bsp_p
  p' := % p  : {n : nat | n < bsp_p}
  S_p := % S p  : {n : nat | n < bsp_p}
  IHp' : flatten (map (get (filterPar select ap)) (processorsUpTo p')) =
         filter select (flatten (map (get ap) (processorsUpTo p')))
  ============================
   flatten (map (get (filterPar select ap)) (processorsUpTo p')) ++
   get (filterPar select ap) S_p =
   filter select (flatten (map (get ap) (processorsUpTo p'))) ++
   filter select (get ap S_p)
```

```
  f_equal;[
     apply IHp'
     |
     unfold filterPar;
         autorewrite with  bsmlbase;
            reflexivity
            ].
Proof completed.

  Qed.

FilterParCorrect is defined
```

Composition correcte

Nous avons montré que la composition de fonction préservait la correction de la parallélisation, ceci est exprimé par l'instance suivante [4] :

```
Instance correctComposition (A B C : Type)
  (APart : Partitionnable A) (BPart : Partitionnable B) (CPart : Partitionnable C)
  (f : B → C) fp (parf : Parallel f fp)
  (g : A → B) gp (parg : Parallel g gp) : Parallel (f :o: g) (fp :o: gp).
```

Comme dans notre preuve papier, la preuve se fait ici en deux réécritures utilisant la relation $\not\Longrightarrow$(définie en coq par *parallel_spec_match*) :

4. `:o:` est une notation pour la composition de fonction

```
  constructor;intros par_a;unfold comp.
correctComposition < 1 subgoal

                ⋮
  APart : Partitionnable A
  BPart : Partitionnable B
  CPart : Partitionnable C
  f : B -> C
  fp : // B \\ -> // C \\
  parf : Parallel f fp
  g : A -> B
  gp : // A \\ -> // B \\
  parg : Parallel g gp
  par_a : // A \\
  ============================
   join (fp (gp par_a)) = f (g (join par_a))

  rewrite parallel_spec_match.
1 subgoal

                ⋮
  ============================
   f (join (gp par_a)) = f (g (join par_a))

  rewrite parallel_spec_match.
1 subgoal

                ⋮
  ============================
   f (g (join par_a)) = f (g (join par_a))
```

Pas besoin ici de préciser quelle spécification est utilisée lors de la réécriture, le système de recherche d'instance utilise automatiquement les instances (*parf* et *parg*) dont les preuves de correction s'unifient avec le but courant.

Cette formalisation de la correction de parallélisation nous permet de raisonner sur les résultats des programmes parallèles comme s'il s'agissait de programmes séquentiels, manipulant des structures de données séquentielles. Bien sûr, tous les programmes parallèles ne peuvent pas entrer dans ce cadre. Par exemple un programme lançant en parallèle des travaux indépendants, sur des structures de données distinctes, ne pourrait pas être spécifiés de cette façon. On peut, dans ce cas, utiliser la méthode de spécification proposée au chapitre précédent 4 ou spécifier chacun des travaux séparément. Un grand nombre de programmes parallèles peuvent cependant être spécifiés sous la forme que nous proposons ici. Nous allons maintenant en présenter quelques exemples.

5.2 Exemple de programme correct - Diffusion de chaleur

Dans cette section, nous présentons la preuve de correction d'un programme parallèle calculant la diffusion de la chaleur dans un corps unidimensionnel homogène.

5.2.1 Spécification du problème

La diffusion de la chaleur dans un matériau unidimensionnel de longueur 1 est décrite par l'équation différentielle :

$$\frac{\delta u}{\delta t} - \kappa \frac{\delta^2 u}{\delta^2 x} = 0$$

où u, t, x et κ sont des réels représentant respectivement la température, le temps, la position et un coefficient de diffusion (constant). Nous supposons les conditions aux bords du matériau constantes ($u(0,t) = l$ et $u(1,t) = r$ avec l et r constantes).

Pour donner une approximation de la fonction u, il est possible de discrétiser cette équation par la méthode des différences finies. Pour une étape de temps dt et une approximation utilisant un nombre fini d'éléments à distance dx les uns des autres, on obtient l'équation suivante :

$$u(x, t + dt) = \kappa \times \frac{dt}{dx^2} \times \big(u(x + dx, t) + u(x - dx, t) - 2 \times u(x, t)\big) + u(x, t) \qquad (5.1)$$

La première étape de notre développement Coq est d'écrire cette spécification. Nous mettons les $\frac{1}{dx} - 1$ valeurs de u (hors extrémités) dans une liste. L'équation (5.1) décrit les valeurs de la liste obtenue à partir de u après une étape de simulation pour les éléments ne se situant pas aux bords. Pour ces derniers, l'équation est

$$u(x_g, t + dt) = \kappa \times \frac{dt}{dx^2} \times \big(u(x_g + dx, t) + l - 2 \times u(x_g, t)\big) + u(x_g, t)$$

pour l'élément le plus à gauche et

$$u(x_d, t + dt) = \kappa \times \frac{dt}{dx^2} \times \big(r + u(x_d - dx, t) - 2 \times u(x_d, t)\big) + u(x_d, t)$$

pour l'élément le plus à droite.

Pour la modélisation de cette spécification, nous supposons l'existence d'un type *number* accompagné des opérations binaires `+`, `-`, $\times$ et `/`. Nous ne supposons rien sur la sémantique de ces opérations.

```
Definition heatEquationFormula (dt dx κ uI uIMinusOne uIPlusOne : number)
                                                              : number :=
  κ × dt / (dx × dx) × ( uIPlusOne + uIMinusOne - uI - uI) + uI.
```

Cette formule spécifie la valeur d'un élément de $\mathfrak{u}$ en fonction de la distance à ses voisins (dx), du pas de temps dt, et des valeurs précédentes des voisins gauche ($uIMinusOne$) et droit ($uIPlusOne$), et de l'élément (uI).

La spécification d'une étape de simulation peut maintenant être définie :

```
Definition heatEquationStepSpecification
    (step : number→number→number→number→number→list number→list number)
                                                                : Prop :=
  ∀ (u : list number)(Hu : u≠[])(leftBound rightBound dt dx κ : number)
    (i : nat)(Hi :i<length u)(default : number),
    nth i (step leftBound rightBound dt dx κ u) default
   =
    heatEquationFormula dt dx κ (nth i u default)
        (if i== 0 then leftBound else nth (i-1) u default) (nth (i+1) u rightBound).
```

Nous utilisons la fonction *nth* pour obtenir le $\mathfrak{i}^{\text{ème}}$ élément de $\mathfrak{u}$. Cette spécification affirme qu'à toute position $\mathfrak{i}$, le résultat d'une étape de simulation doit être égale au résultat de la formule ci-dessus. Les conditions particulières des bornes sont traitées en adaptant les paramètres de la formule. Là où la valeur de l'élément de gauche doit être fournie, on donne la constante correspondante si $\mathfrak{i}$ vaut 0. Pour l'élément de droite *nth* $(i+1)$ *u rightBound* renvoie l'élément à droite de la position $\mathfrak{i}$ sauf si $\mathfrak{i}$ est le dernier élément de la liste, dans ce cas $\mathfrak{i}+1$ étant en dehors de la liste, c'est l'argument donnant la valeur par défaut *rightBound* qui est utilisé.

5.2.2 Implantations séquentielle et parallèle

Nous écrivons ensuite un programme séquentiel satisfaisant cette spécification :

```
Fixpoint heatEquationSeq (lBound rBound dt dx κ : number)(u : list number)
                                                          : list number :=
  match u with
    | [] ⇒ []
    | uI :: u' ⇒
      match u' with
        | [] ⇒ [ heatEquationFormula dt dx κ uI lBound rBound ]
        | uIPlusOne:: _ ⇒
          (heatEquationFormula dt dx κ uI lBound uIPlusOne)::
          (heatEquationSeq uI rBound dt dx κ u')
      end
  end.
```

La preuve que cette implantation vérifie la spécification est faite par induction sur la liste $\mathfrak{u}$ puis sur l'indice $\mathfrak{i}$. Elle repose sur le lemme suivant qui a été prouvé par induction sur la liste *u1* :

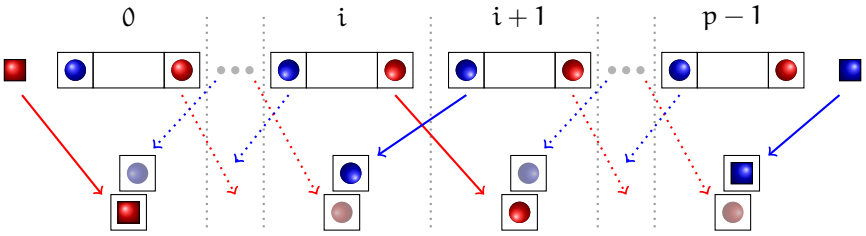


FIGURE 5.3 – *Schéma de communication de la fonction* **getBounds**

```
Lemma heatEquationSeqApp :
  ∀ (leftBound rightBound dt dx κ : number)(u1 u2 : list number),
    heatEquationSeq leftBound rightBound dt dx κ (u1 ++ u2)
   =
    (heatEquationSeq leftBound (hd rightBound u2) dt dx κ u1) ++
    (heatEquationSeq (last u1 leftBound) rightBound dt dx κ u2).
```

Ce lemme permet de décomposer le résultat du programme séquentiel sur une concaténation de listes. Nous ne détaillons pas plus ces preuves qui ne présentent pas un grand intérêt en soi. À présent nous allons définir un programme parallèle effectuant ce même calcul. Notre programme prend en paramètre un vecteur parallèle de listes de nombres. Pour simplifier les communications de valeurs, nous imposons qu'il y ait au moins une valeur à chaque processeur (paramètre *Hu*).

```
Program Definition heatEquationPar
                   (leftBound rightBound dt dx κ : number)(u : par(list number))
                   (Hu : ∀ i, get u i ≠ nil) : par(list number) :=
let bounds := getBounds leftBound rightBound Hu in
 apply
  (parfun2 (fun l r ⇒ heatEquationSeq l r dt dx κ) (fst bounds) (snd bounds))
  u.
```

La fonction **getBounds** effectue les communications en utilisant la primitive *put* selon le schéma de la figure 5.3. Elle renvoie deux vecteurs parallèles, l'un contenant la valeur reçue de droite, l'autre la valeur reçue de gauche. Depuis tous les processeurs, sauf le premier et le dernier, elle envoie le premier élément de la liste locale au processeur de gauche et le dernier élément au processeur de droite. Si le processeur est le premier ou le dernier, une des communications n'est pas effectuée, et la constante adéquate est utilisée (*leftbound* ou *rightBound*). Afin de pouvoir vérifier la spécification de notre programme, la fonction **getBounds** est fortement spécifiée, voici son type :

```
Program Definition getBounds (A :Type)(leftBound rightBound : A)(v : par(list A))
                                                    (H : ∀ i, get v i ≠ nil) :
  { vr : par A | ∀ (i :processor),
    get vr i = if ( i == firstProcessor ) then leftBound else sLast (get v (i-1)) }
  ×
  { vr : par A | ∀ (i :processor),
    get vr i = if ( i == lastProcessor) then rightBound else sHead (get v (min (i+1)
                                                                    lastProcessor)) }
```

Pour plus de commodité, le programme a été défini dans le langage Russell (décrit section 3.1.4). Les fonctions *sLast* et *sHead* retournent le dernier élément ou la tête d'une liste prouvée non-vide. Russell se sert de la preuve H pour compléter les appels à *sLast* et *sHead*. Il permet également d'effectuer des opérations sur les numéros de processeurs comme s'il s'agissait d'entiers. Lorsque des entiers sont utilisés comme numéro de processeur, des obligations de preuves sont générées pour s'assurer qu'ils sont bien inférieurs au nombre de processeurs disponibles.

5.2.3 Preuve de correction

Maintenant que notre programme parallèle est défini, nous pouvons poser un théorème décrivant son lien avec la fonction séquentielle.

```
Theorem heatEquationParSeqEq :
  ∀ (leftBound rightBound dt dx κ : number)
    (u : par(list number))(Hu : ∀ i, get u i ≠ nil),
    heatEquationSeq leftBound rightBound dt dx κ (listOfParList u)
  =
    listOfParList(heatEquationPar leftBound rightBound dt dx κ Hu).
```

Ce théorème est prouvé à l'aide d'un lemme intermédiaire qui décrit le résultat au processeur `i`. Ce résultat correspond à l'application d'une étape de simulation séquentielle sur la liste initialement présente sur le processeur `i`. Les valeurs utilisées aux bornes sont les valeurs retournées par *getBounds* :

```
Program Definition heatParSeqEqAtI :
  ∀ (leftBound rightBound dt dx κ : number)(u : par(list number))
    (Hu : ∀ i, get u i ≠ nil)(i :processor),
    (get (heatEquationPar leftBound rightBound dt dx κ Hu) i)
  =
    heatEquationSeq (leftBoundAtI leftBound Hu i) (rightBoundAtI rightBound Hu i)
                    dt dx κ (get u i).
```

La fonction *leftBoundAtI* désigne soit le dernier élément de la liste du processeur à gauche de `i`, soit la valeur *leftBound* au premier processeur (*rightBoundAtI* est définie de façon similaire). Ces fonctions de description des valeurs reçues utilisent la fonction de description des vecteurs parallèles *get*, elles ne doivent donc pas être utilisées dans le programme extrait :

```
Program Definition leftBoundAtI (leftBound : number)(u : par(list number))
                                (Hu : ∀ i, get u i ≠ nil)(i :processor)
                                : number :=
  if ( (‘i) == (‘firstProcessor) ) then leftBound else sLast (get u (i-1)).
```

La preuve du lemme intermédiaire se fait en distinguant trois cas : `i` est le premier processeur, `i` est le dernier processeur, ou `i` est un autre processeur. Elle est ensuite résolue par une évaluation symbolique à l'aide des bases de réécriture liées à la formalisation de BSML. Il ne reste plus qu'à combiner le résultat obtenu par *heatParSeqAtI* pour tous les processeurs. Pour cela, nous utilisons la tactique d'induction sur les listes réparties présentée à la section précédente.

Le théorème de correction de notre implantation parallèle impose cependant une restriction sur la répartition des données. Cette restriction nous permet d'avoir un schéma de communication très simple, facile à implanter et à montrer correct, mais elle nous empêche de montrer que *heatEquationPar* est une parallélisation correcte de *heatEquationSeq*. En fait, notre parallélisation est correcte pour toute liste de taille supérieure au nombre de processeurs. Nous commençons donc par montrer que le type $\{l : \mathsf{list}\ A \mid length\ l \geq bsp\_p\}$ peut être réparti en un vecteur de type $\{pl : \mathsf{list}\ A \mid \forall\ i, get\ pl\ i \neq \mathsf{nil}\}$.

La répartition d'un élément de type $\{l : \mathsf{list}\ A \mid length\ l \geq bsp\_p\}$ est définie à l'aide de Russell par la fonction de répartition des listes normales. Il suffit ensuite de prouver qu'à tout processeur la liste n'est pas vide. On utilisera pour cela le fait que la liste initiale a plus d'éléments qu'il n'y a de processeurs et qu'elle produit une répartition équilibrée. De même, en séquentialisant une liste répartie dont les partitions sont non-vides, on obtient bien une liste dont la taille est supérieure à *bsp_p*, le nombre de partitions.

Il nous reste à prouver que nos fonctions sont bien compatibles (champs *join_part_match* de la classe Partitionnable). Cette formalisation nécessite malheureusement l'usage d'un axiome affirmant que les termes prouvant ($length\ l1 \geq bsp\_p$) et ($length\ l2 \geq bsp\_p$) sont égaux si $l1{=}l2$. Il s'agit d'une restriction de l'axiome de *non-pertinence des termes de preuves*[5] aux seules preuves sur la longueur de nos listes. Une fois que nous disposons de cet axiome, la preuve se résume à l'application de la preuve de compatibilité que nous avons faite pour les listes simples.

Maintenant que nous disposons d'un type parallèle correspondant à notre programme parallèle et d'un type séquentiel *Partitionnable* correspondant, nous définissons dans le langage Russell une spécification restreinte. Celle-ci prend en paramètre une liste de taille supérieure à *bsp_p* et renvoie également une liste de ce même type :

```
Program Definition heatEquationSeqRestricted
  (leftBound rightBound dt dx κ : number)(u : {l: list number | length l ≥ bsp_p}) :
{l: list number | length l ≥ bsp_p} :=
  heatEquationSeq leftBound rightBound dt dx κ u.
```

Une obligation de preuve est générée pour le type de retour, nous demandant de prouver que la liste retournée est bien de taille supérieure à *bsp_p*. Pour cela nous avons

5. En anglais, *proof irrelevance*

prouvé dans un lemme auxiliaire que *heatEquationSeq* retourne une liste de même taille que la liste initiale. L'obligation de preuve est alors triviale à résoudre.

De même, nous utilisons Russell pour définir *heatEquationParRestricted*, un équivalent de notre fonction parallèle qui prend en entrée le bon type de donnée et retourne également une valeur de ce même type. Le système nous demande alors de prouver qu'à tous les processeurs, le vecteur résultat contient une liste non-vide. La preuve est faite par évaluation symbolique et à l'aide du lemme affirmant que *heatEquationSeq* préserve la longueur de la liste.

Nous pouvons maintenant prouver que notre programme séquentiel restreint est correctement parallélisé par notre programme parallèle grâce à l'instance suivante de la classe Parallel :

```
Instance heatEquationParallel (leftBound rightBound dt dx κ : number) :
                    Parallel
                    (heatEquationSeqRestricted leftBound rightBound dt dx κ)
                    (heatEquationParRestricted leftBound rightBound dt dx κ).
```

La preuve consiste à appliquer l'axiome de non-pertinence des termes de preuves, puis le théorème heatEquationParSeqEq.

Nous avons utilisé nos types de listes fortement spécifiées comme type de retour afin que la fonction de calcul de la chaleur puisse être composée avec elle-même. Nous aurions pu nous contenter d'une spécification séquentielle retournant une simple liste et d'avoir pour type de retour une liste répartie sans contrainte de distribution pour notre programme parallèle. Cependant le programme de calcul de chaleur est destiné à être composé plusieurs fois avec lui-même jusqu'à l'équilibre, nous souhaitons donc que cette composition soit également correcte.

5.2.4 Mesures de performance

L'implantation Coq du calcul de diffusion de la chaleur est codée dans un foncteur prenant deux modules en paramètres, le module contenant les primitives BSML et un module contenant une définition du type *number* et les opérations associées. Pour les mesures de performance, le foncteur BSML extrait est appliqué au module `BsmlNat` présenté au chapitre précédent et à un module définissant *number* par le type `float` et ses opérations. Cette encapsulation des opérations introduit un niveau d'indirection lors de l'appel à celles-ci et surtout empêche le compilateur d'utiliser une optimisation nommée *unboxing* qui améliore nettement les performances pour les calculs arithmétiques. C'est pourquoi nous avons aussi mesuré les performances d'une version *défonctorisée* de notre code. La défonctorisation consiste ici à remplacer l'application d'un foncteur à un module par le code correspondant au résultat de cette application. Cette opération peut être vue comme une opération d'*inlining* au niveau des foncteurs et non des fonctions. Elle est ici effectuée "à la main" et n'est donc pas une opération sûre. Il est cependant envisageable d'automatiser cette opération et de vérifier sa correction.

Les mesures ont été effectuées dans des conditions similaires pour la version extraite du programme, la version défonctorisée, et une version non prouvée, écrite directement

en BSML. Les temps de calculs ont été mesurés sur la grappe de calcul MIREV pour des listes de tailles croissantes. Les temps de calculs sont donnés pour 200 itérations de la simulation avec une collecte complète de la mémoire [6] après chaque itération. La figure 5.4 montre les temps d'exécutions sur la machine MIREV, une fois défonctorisée. La version extraite se comporte aussi bien que la version écrite directement en BSML.

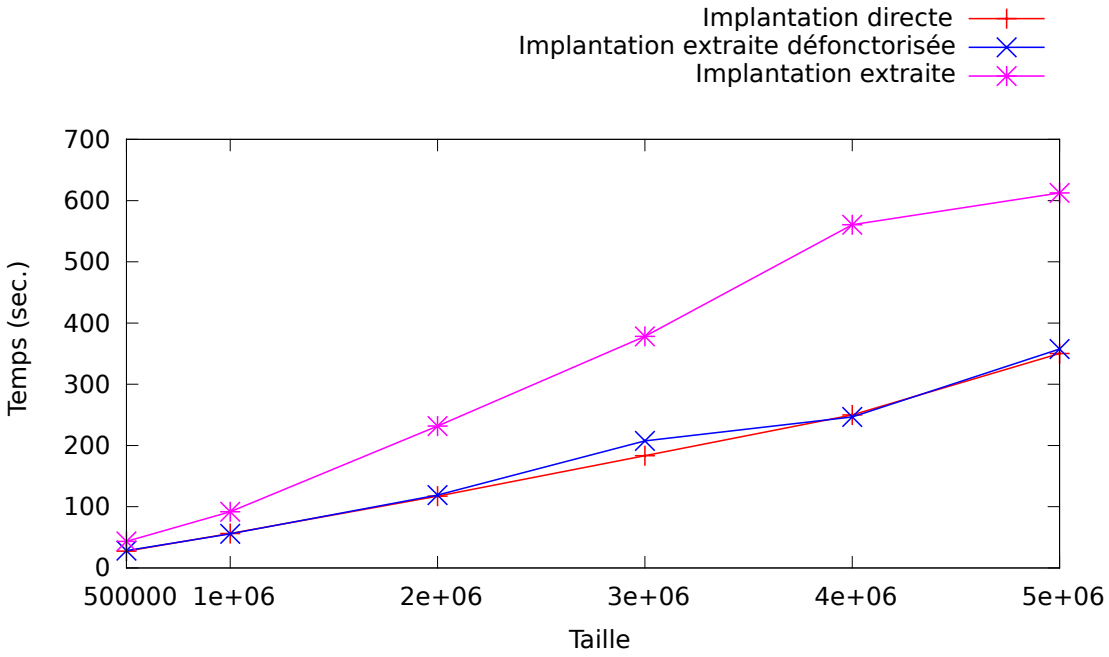
5.3 Conclusion

Nous avons mis en place un cadre formel pour décrire ce qu'est la parallélisation correcte d'un programme séquentiel. Grâce à cette formalisation, nous pouvons écrire un programme séquentiel, prouver des propriétés sur le résultat de ce programme sans s'encombrer des détails du parallélisme, puis écrire une implantation parallèle du programme. L'implantation parallèle est ensuite prouvée correcte par rapport au programme séquentiel, ce qui signifie que les propriétés prouvées sur le résultat de ce dernier seront valables pour la séquentialisation du résultat du programme parallèle. Nous avons ainsi pu obtenir une implantation parallèle prouvée correcte d'un algorithme de calcul de la diffusion de la chaleur dans un corps unidimensionnel homogène. Afin de pouvoir implanter ce programme plus facilement en parallèle, nous avons ajouté une pré-condition sur la répartition des données. De ce fait, le programme n'était plus une parallélisation correcte du programme séquentiel initial. Nous avons dû définir un nouveau programme séquentiel à partir du précédent avec une pré-condition compatible avec celle de l'implantation parallèle. Cette re-formulation du programme initial est de notre point de vue nécessaire car elle explicite les pré-conditions nécessaires pour que la parallélisation soit correcte.

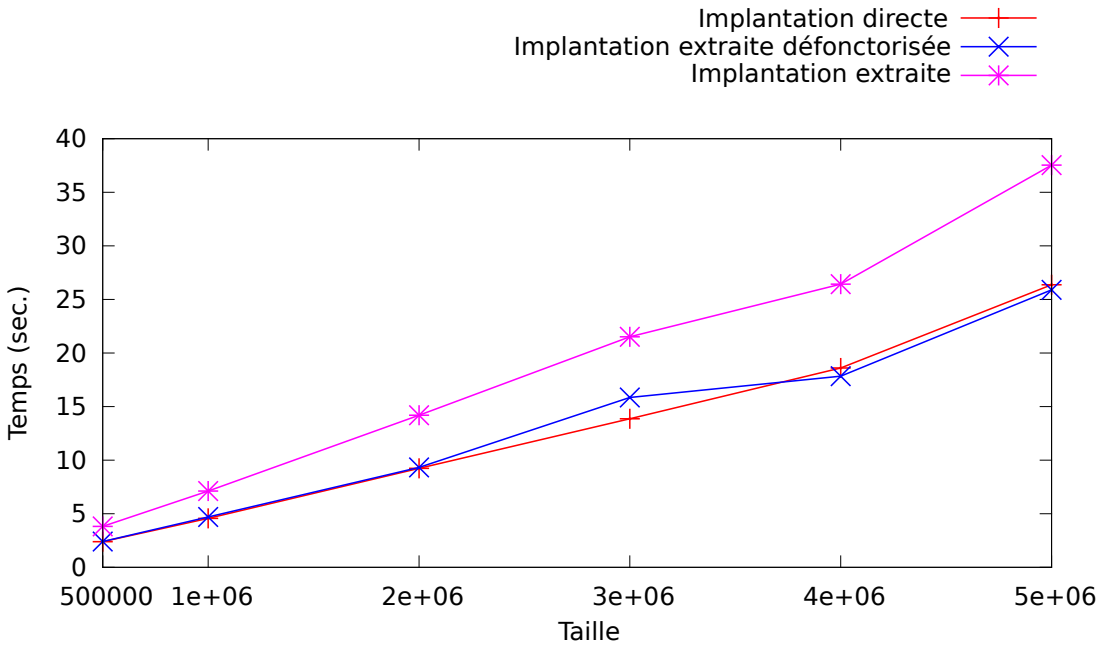
Prouver la correction de notre implantation parallèle, sans compter la définition des types fortement spécifiés et l'instance de Partitionnable correspondante, aura cependant demandé la définition d'une dizaine de fonctions auxiliaires, environ autant de lemmes et environ 300 lignes de code de preuves compactes. Obtenir une implantation parallèle correcte demande donc un effort non-négligeable, et dont une petite partie seulement est réutilisable pour définir d'autres programmes.

Pour simplifier le problème du développement à plus large échelle de programmes parallèles prouvés corrects, nous proposons dans les chapitres suivants d'utiliser la programmation par squelettes algorithmiques et l'algorithmique constructive afin de mutualiser les efforts de preuve fournis pour prouver la correction d'un programme.

6. En anglais *full major garbage collection*



(a) sur 1 processeur



(b) sur 16 processeurs (8 nœuds)

FIGURE 5.4 – *Mesure du temps de calcul et de collecte mémoire sur la machine MIREV*

Dérivation de programmes

6

Sommaire

Au chapitre précédent, nous avons vu comment spécifier des programmes parallèles en formalisant leur lien avec les programmes séquentiels. Cependant écrire la parallélisation d'un programme et prouver sa correction reste une tâche complexe qui nécessite beaucoup de temps et de bonnes connaissances en algorithmique parallèle.

Nous proposons d'utiliser Coq comme un outil pour la dérivation de programmes parallèles corrects à partir de spécifications. Ici une spécification sera une combinaison de fonctions simples, facilement compréhensibles et représentant bien les étapes du raisonnement qui sous-tend la spécification. Cette fonction ne prend pas en compte l'efficacité du calcul et encore moins le parallélisme. Dans cette spécification, nous pouvons utiliser des fonctions simples dont nous comprenons bien la sémantique, comme par exemple *map*, *filter*, *fold_left*. Ces fonctions d'ordre supérieur peuvent en fait être vues comme des squelettes algorithmiques. Nous avons présenté au chapitre 5 une implantation parallèle correcte de *filter*, la fonction *map* est parallélisée trivialement par la fonction suivante :

```
Program Definition mapPar A B (f : A→B) (l : par (list A)) : par (list B) :=
  parfun (map f) l
```

Pour ce qui est de la réduction *fold_left*, nous verrons par la suite comment la paralléliser lorsque l'opérateur utilisé est associatif et que l'élément initial utilisé est l'élément neutre pour cette opération. Nous supposons pour le moment l'existence de *fold_leftPar*. C'est là l'avantage de penser en terme de squelettes algorithmiques : nous savons qu'une

implantation parallèle existe pour un squelette donné, mais nous n'avons pas à nous préoccuper des détails de son implantation.

6.1 Construction de programmes parallèles corrects par composition de squelettes

6.1.1 Un exemple de composition

Pour introduire notre démarche, nous utiliserons l'exemple suivant : supposons que l'on dispose d'une fonction *size* de type $A \rightarrow \mathsf{nat}$, nous donnant la taille d'un élément de type A. Nous voulons compter le nombre d'éléments dont la taille est supérieure à 0 dans une liste [1]. Il faut en premier lieu, calculer la taille de tous les éléments. On utilise pour cela la fonction *map*. Ensuite nous éliminons de la liste obtenue tous les éléments de taille nulle à l'aide de la fonction *filter*, puis, pour compter les éléments validant notre condition, nous remplaçons toutes les tailles non-filtrées par 1 et nous faisons la somme des éléments de la liste à l'aide de la fonction *fold_left*. La spécification s'écrit ainsi [2] :

```
Definition counting (A :Type) (size : A→ nat) :=
(fun l ⇒ fold_left plus l 0):o:(map (fun x ⇒ 1)):o:
                                    (filter (fun x ⇒ x > 0)):o:(map size).
```

Notre fonction de comptage est une composition de squelettes algorithmiques pour lesquels existent des implantations parallèles prouvées correctes. Aussi, pour obtenir une implantation parallèle de notre spécification, il suffit de remplacer les appels aux squelettes par leur implantation parallèle :

```
Definition countingPar (A :Type) (size : A→ nat) :=
  (fun l ⇒ fold_leftPar plus l 0):o:(mapPar (fun x ⇒ 1)):o:
                                    (filterPar (fun x ⇒ x > 0)):o:(mapPar size).
```

Pour prouver que cette implantation est une parallélisation correcte de la précédente, nous construisons une instance de la classe parallèle à l'aide de l'instance *correctComposition* donnée à la section 5.1.4 (page 85).

```
Instance countingParCorrect A (size : A → nat) : Parallel (counting A size) (countingPar A size) :=
  correctComposition
     (f :=(fun l ⇒ fold_left plus l 0))
     (parg :=correctComposition
        (f :=(map (fun x ⇒ 1)))
        (parg := correctComposition(f :=(filter (fun x ⇒ x > 0)))(g := (map size)))).
```

Les mécanismes d'inférence de paramètres implicites et de recherche d'instances nous permettent de ne préciser qu'un minimum d'informations pour construire l'instance de

1. Bien sûr cet exemple est très simple, trop peut-être, mais un exemple plus compliqué n'apporterait pas grand chose quant à la présentation de l'approche utilisée, et encombrerait le document d'explications inutiles.
2. La notation `:o:` définit dans notre cadre la composition de fonctions.

Parallel. L'instance *correctComposition* est bien de type Parallel. Ici, six de ses paramètres nous intéressent : les fonctions séquentielles f et g, leurs homologues parallèles $f_{/\!/}$ et $g_{/\!/}$, et les deux instances de la classe Parallel, *parf* et *parg*, montrant que f (respectivement g) est correctement parallélisée par $f_{/\!/}$ (respectivement $g_{/\!/}$). Pour chaque composition, nous donnons une valeur pour le paramètre f, élément droit de la composition, et pour le paramètre *parg*, sauf pour la dernière composition où nous donnons la valeur pour g. La valeur de g est déduite à partir du paramètre *parg* par le mécanisme d'inférence des paramètres implicites. Les paramètres *parf* (et *parg*) sont de type Parallel. Le mécanisme de recherche d'instance est déclenché pour ces paramètres lors de l'inférence des paramètres implicites car Parallel est une classe de types. L'instance obtenue, correspondant à f ou à g, nous fournit également la valeur à utiliser pour $f_{/\!/}$ ou $g_{/\!/}$.

6.1.2 Automatisation de la construction

Les mécanismes d'inférence de paramètres implicites et de recherche d'instances de classes nous ont permis de prouver simplement que notre parallélisation était correcte. La construction de la fonction parallèle à partir de la spécification séquentielle est triviale puisque nous n'utilisons que des squelettes algorithmiques, la preuve est également très simple car il s'agit de composition de fonctions correctement parallélisées. Devant tant de simplicité nous aimerions que le système fasse le travail pour nous ! Nous voudrions par exemple pouvoir écrire

```
Definition countingParAuto (A :Type)(size :A→nat) := parallel (f :=counting A size).
```

Pour pouvoir faire ceci, nous avons défini la classe Parallelisable suivante :

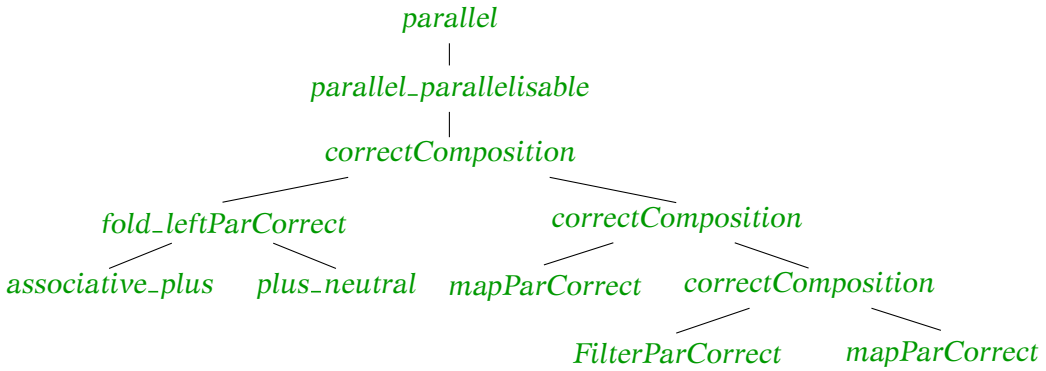
```
Class Parallelisable {A B :Type } (f : A → B)
                     {APart :Partitionnable A} {BPart :Partitionnable B} :={
  parallel : //A\\ → //B\\;
  spec_match : ∀ a_//, join ( parallel a_//)) = f (join a_//)
}.
```

La différence avec la classe Parallel réside dans le fait que la fonction parallèle n'est plus un paramètre de la classe mais un champ. L'utilisation de ce champ lancera donc une recherche d'instance. En spécifiant le paramètre f on obtient automatiquement la parallélisation de f si elle existe. Afin de profiter de toutes les fonctions que l'on a déclarées parallèles, on définit l'instance suivante qui permet d'aller chercher dans les instances de Parallel une parallélisation pour n'importe quelle fonction séquentielle :

```
Instance parallel_parallelisable (A B :Type)(f :A →B) fp {H :Parallel A B f fp} :
                                                          Parallelisable f :=
 {parallel := fp}.
```

Pour tous nos squelettes, il existe une instance de Parallel et nous disposons également d'une instance de Parallel pour la composition de fonctions correctement parallélisées. Il est donc possible de construire automatiquement le programme parallèle correspondant à une composition des squelettes algorithmiques.

La fonction *countingParAuto* se construit automatiquement à l'aide de ce champ de la classe Parallelisable. Le chaînage d'instances utilisées est représenté par l'arbre suivant :



Comme dit précédemment, pour pouvoir paralléliser *fold_left*, il faut que l'opérateur utilisé pour la réduction soit associatif et que l'élément utilisé initialement pour la réduction soit neutre pour l'opérateur utilisé. L'instance *fold_leftParCorrect* de la classe de type Parallel dépend donc d'une instance de la classe Associative définie en introduction (section 3.1.7) et d'une instance d'une classe Neutral assurant que 0 est l'élément neutre pour l'addition. Cet exemple intéressant nous montre comment la parallélisation d'une fonction peut-être conditionnée par les propriétés algébriques d'un opérateur.

Si ce terme est extrait directement, il contiendra un certain nombre d'artifices qui court-circuitent le système de type d'OCaml car les classes de types ne sont pas supportées par celui-ci. Pour obtenir un terme n'utilisant plus le mécanisme de classe de type, nous réduisons le terme par la stratégie définie par la tactique `simpl` :

```
Definition countingParExtractible (A :Type) (size : A→ nat) :=
      Eval simpl in parallel (f :=counting A size).
```

Le terme obtenu est exactement celui de que nous avons écrit pour définir *countingPar*.

Dans cette section nous avons décrit un mécanisme de classe de type qui nous permet de construire automatiquement un programme parallèle à partir de composition de fonctions d'ordre supérieur dont il existe une implantation parallèle prouvée correcte. Cette construction automatique est capable d'utiliser les propriétés déclarées des paramètres des fonctions utilisées comme dans la cas de la parallélisation de *fold_left*. Cette possibilité, couplée à la possibilité de définir un ordre de priorité lors de la définition d'instance, devrait permettre de définir plusieurs implantations parallèles pour une même fonction [3]. Ces différentes implantations utilisant différentes propriétés des paramètres pourraient ensuite être ordonnées suivant leur efficacité. On obtiendrait ainsi une parallélisation optimale d'un squelette pour une utilisation spécifique.

3. À titre d'exemple, la réduction d'une liste de gauche à droite par un opérateur $\oplus$ et de droite à gauche par un opérateur $\otimes$ peut être calculée en un seul parcours de liste de gauche à droite si l'opérateur $\otimes$ est commutatif et associatif.

6.2 Support pour la transformation de programmes

Pouvoir construire automatiquement nos programmes parallèles est une bonne chose, mais encore faut-il que le programme séquentiel ait une forme parallélisable (composition de squelettes) et efficace. Notre fonction *counting* est bien sous une forme directement parallélisable mais elle n'est pas efficace, de nombreuses listes intermédiaires sont générées et re-parcourues. Cette inefficacité se retrouve au niveau parallèle. Nous avons donc besoin d'un moyen de transformer un programme en un autre tout en ayant l'assurance de ne pas en changer le résultat. Pour cela nous avons développé un ensemble de tactiques permettant d'écrire nos transformations dans un style d'écriture classique en preuve par calcul de programmes[4] :

Expression
={ justification de la transformation}
Nouvelle expression

Ce style a l'avantage d'améliorer la lisibilité des preuves par son caractère déclaratif et d'être dirigé par la forme que nous souhaitons obtenir, et non par l'effet plus ou moins prévisible de tactiques sur le but.

6.2.1 Preuve d'équivalence de programmes par transformation

Voici un exemple de preuve Coq écrite à l'aide de ces tactiques. La preuve est plus longue que si nous l'écrivions directement avec les tactiques standard, cependant notre but ici est d'exhiber dans le code source les différentes étapes de la preuve, les transformations successives des programmes. La preuve faite ici montre que la fonction *filterMap* (dont nous ne donnons pas le code ici) donne le même résultat que la fonction *filter* composée à la fonction *map*.

```
Lemma filter_map_fusion (A B :Type) (select : A → bool) (f : B → A) :
  ∀ l, filter select (map f l) = filterMap select f l.
Proof.
  Begin.
  induction l.
  check (filter select (map f []) = filterMap select f []) is GOAL.
      GOAL
      ={unfold filter, map, filterMap}
      (@nil A= @nil A).
   [].
  check ( filter select (map f (a :: l)) = filterMap select f (a :: l) ) is GOAL.
      LHS
      ={unfold map}
      ( filter select ((f a)::(map f l))).
      ={unfold filter}
```

4. Initialement proposée par Feijen, cette notation a été introduite dans la communauté de calcul de programme à la BMF par Backhouse [4]

```
      (if select (f a)
        then f a :: filter select (map f l)
        else filter select (map f l)).
      RHS
      ={unfold filterMap}
      (if select (f a)
        then f a :: filterMap select f l
        else filterMap select f l).
      ={rewrite IHl}
      (if select (f a)
        then f a :: filter select (map f l)
        else filter select (map f l)).
  [].
Qed.
```

L'implantation de ces tactiques est basée sur la possibilité de définir des notations personnalisées pour les tactiques. Nous ne présenterons pas ici les détails techniques de l'implantation de celles-ci, nous expliquerons seulement leur utilisation. Le lecteur intéressé pourra lire [133] ou consulter le code de la bibliothèque *Program Calculation in Coq*, relativement court [136].

La tactique `Begin` prépare le contexte et le but pour une preuve utilisant nos tactiques, ici elle ne fait qu'introduire les variables dans le contexte.

Nous ne disposons pas de syntaxe appropriée pour le raisonnement par induction, nous utilisons donc la tactique d'induction `Ltac` de façon classique. Pour que les cas à prouver soient lisibles dans le code source, nous utilisons la tactique `check (`*terme*`) is GOAL`. Cette tactique échoue si *terme* n'est pas unifiable avec le but courant.

Nous utilisons ensuite la notation `GOAL ={tac}(`*terme*`)` pour transformer le but courant en *terme*. L'équivalence est montrée par la tactique `tac` qui peut être n'importe quelle tactique du langage `Ltac`. La tactique `unfold` utilisée ici ne fait que déplier les définitions des fonctions citées, on pourrait lire ceci comme *par définition de filter, map et filterMap*.

Une fois atteinte une égalité suffisamment simple (prouvable par le système à l'aide de la tactique `reflexivity`), nous utilisons la tactique `[]` qui termine la preuve du but courant.

Pour le cas inductif, nous utilisons `LHS` pour ne transformer que la partie gauche (Left Hand Side) d'une équation, et `RHS` pour transformer la partie droite (Right Hand Side). La tactique donnée entre accolades sert alors à prouver que le terme donné est bien égal à la partie de l'équation sélectionnée.

Ces tactiques permettent d'avoir une preuve facile à lire, ici seule l'utilisation de l'hypothèse *IHl* n'est pas très explicite, il s'agit de l'hypothèse d'induction. Il serait intéressant d'avoir un meilleur support de l'induction dans notre syntaxe.

Cet exemple ne présente cependant pas la construction d'un programme par transformation mais uniquement une preuve d'équivalence entre deux programmes. Cette activité

n'est pas très prospective. Voyons comment faire pour construire par transformations successives à partir d'une spécification un programme équivalent.

6.2.2 Construction de programme par transformation

Nous avons l'intuition que notre spécification *counting* n'est pas optimale, mais nous n'avons pas d'idée précise de la forme qu'elle devrait avoir.

Nous commençons donc par écrire une spécification de notre fonction optimisée sans en donner le corps. Son type devient alors un but à prouver. Il exprime l'existence d'une fonction f dont le résultat est égal à la fonction *counting* appliquée aux variables A, *size* et l. L'arrière plan du code source est ici grisé afin de faciliter la distinction avec les commentaires.

```
Definition optimization (A :Type) (size :A→ nat)(l : list A) :{f| counting A size l=f}.
```

Nous utilisons à présent notre tactique **Begin** pour commencer la construction d'un programme à partir de cette spécification. Le but est de la forme *counting A size l* = *?i*. La variable *?i* est désignée par le signe *?* suivi d'un nombre. Il s'agit d'une variable existentielle [5] dont l'instanciation a été retardée. Si, au moment de sauvegarder le terme, cette variable n'a été instanciée par aucune valeur, Coq rejettera le terme.

```
Begin.
```

Il n'est pas possible de faire directement réference aux variables existentielles dans le code, pour pallier ce problème nous avons une tactique **BeginEvar** qui attribue un nom à celles-ci. Le mécanisme sous-jacent impose d'utiliser une tactique **endEvar** une fois le but courant résolu.

```
BeginEvar.
```

Nous pouvons maintenant utiliser la tactique **check** pour montrer dans le code le but courant.

```
check (counting A size l = EVAR) is GOAL.
```

5. le terme evar est souvent utilisé pour désigner de telles variables

Nous souhaitons transformer la partie gauche de l'équation, nous la sélectionnons par `LHS`. Pour l'instant nous déplions uniquement la définition de la spécification par la tactique `unfold` pour exhiber sa définition, que nous écrivons également. Le code source présenté ici permet ainsi de suivre les étapes du raisonnement.

```
   LHS
={unfold counting}
  (fold_left plus (map (fun x ⇒ 1) (filter (fun x ⇒ x > 0) (map size l))) 0).
```

Nous allons maintenant utiliser un lemme de fusion de *fold_left* et *map* pour transformer notre fonction

```
={rewrite fold_left_map_fusion}
  (fold_left (fun d e ⇒ plus d 1) (filter (fun x ⇒ x > 0) (map size l)) 0).
```

Puis nous fusionnons *filter* et *map* à l'aide du lemme *filterMapFusion* défini précédement

```
={rewrite filter_map_fusion}
  (fold_left (fun d e ⇒ plus d 1) (filterMap (fun x ⇒ x > 0) size l) 0).
```

Nous sommes maintenant satisfait de la forme de notre programme, pour arrêter la dérivation, il suffit d'utiliser la tactique de fin de transformation :

```
[].
```

Celle-ci instancie la variable existentielle f par la fonction courante. par unification de la variable existentielle (membre droit de l'équation) avec le membre gauche de l'équation que nous venons de définir. Notre définition est maintenant terminée, nous utilisons `endEvar` puis `Defined` pour enregistrer notre terme.

```
 endEvar.
Defined.
```

Notre fonction est maintenant définie, le terme *Optimisation* est composé de notre fonction optimisée et de la preuve qu'elle donne le même résutat que *counting*. Nous pouvons obtenir la fonction optimisée ainsi construite en utilisant la projection *proj1_sig*

```
Definition optimized (A :Type) (size : A→ nat) (l : list A) :=
  Eval simpl in (proj1_sig (optimization A size l)).
```

Les optimisations présentées ici ne sont pas très compliquées, et nous aimerions que ce genre de détails soient automatisés, afin de pouvoir nous intéresser aux problèmes plus complexes. Nos lemmes de transformations qui correspondent à des optimisations ont été enregistrés dans une base de réécriture nommée *optimisation*. Nous pouvons donc laisser Coq s'occuper de ces détails seul.

```
Definition optimization' (A :Type)(size : A→ nat)(l : list A) :{f|counting A size l=f}.
Begin.
LHS
={unfold counting}
(fold_left plus (map (fun x ⇒ 1) (filter (fun x ⇒ x > 0) (map size l))) 0).
autorewrite with optimization.
```

Et comme nous apprécions que le code source de notre développement soit lisible, une fois le résultat de l'optimisation obtenu, nous utilisons la tactique **check** pour garder une trace du programme obtenu par réécriture.

```
check ( fold_left (fun a _ : nat ⇒ a + 1) (filterMap (fun x : nat ⇒ x > 0) size l) 0) is
LHS .
[].
Defined.
```

Si après avoir développé ce code nous modifions notre base de réécriture, et que cette modification a une incidence sur le terme produit ici, la tactique **check** échouera. Nous aurons dans notre code source une mémoire du programme que nous avions produit, nous pourrons donc choisir entre re-prouver l'équivalence de celui-ci avec la spécification ou utiliser le nouveau programme.

Nos tactiques sont un simple sucre syntaxique, mais elles permettent d'utiliser le langage **Ltac** dans un style plus lisible, adapté à la transformation de programme. Dans une certaine mesure, nos tactiques ont un but similaire à celui du langage C-zar [44], qui propose, en remplacement du langage procédural **Ltac**, un langage de tactique déclaratif grâce auquel le code source ressemble à une preuve mathématique standard. Cependant notre implantation est beaucoup plus légère et ne se propose que de faciliter la transformation de programme.

6.3 Conclusion

Nous avons présenté dans ce chapitre deux mécanismes indépendants mais complémentaires.

Le premier s'appuie sur le mécanisme de classes de types pour construire automatiquement une parallélisation correcte par composition de squelettes algorithmiques pour lesquels une parallélisation correcte a été définie. Le second utilise la souplesse syntaxique du langage `Ltac` pour obtenir un mécanisme de transformation de programme permettant de faciliter la construction incrémentale d'une fonction optimisée à partir d'une implantation naïve utilisée comme spécification.

Quelle que soit la syntaxe utilisée, la définition d'une spécification et sa transformation ne peuvent être menées à bien simplement que si nous disposons de nombreuses définitions et de nombreux lemmes prouvant des transformations tels que le lemme *filterMapFusion*. Nous avons développé une bibliothèque contenant environ 120 définitions et 350 preuves qui viennent compléter la bibliothèque standard de Coq. L'assistant de preuve Coq dispose heureusement de mécanismes [6] permettant de lister toutes les définitions ou lemmes utilisant une définition particulière ou encore s'unifiant avec un motif donné. Ces mécanismes s'avèrent vite indispensables lorsqu'il est nécessaire de trouver un lemme adéquat pour une transformation.

Les mécanismes que nous avons présentés, associés aux mécanismes de Coq, fournissent un environnement idéal pour la dérivation de programmes parallèles corrects par construction.

6. `SearchAbout`, `SearchPattern`

Homomorphisme BSP

7

Sommaire

Nous avons présenté jusque là deux squelettes algorithmiques prouvés corrects, *filter* et *map*. Ces deux squelettes sont simples, mais nous avons besoin de squelettes plus expressifs pour être en mesure de développer des programmes intéressants.

Pour ceci, il nous faut un squelette suffisamment général pour exprimer de nombreux problèmes, mais dont la structure permet une parallélisation efficace.

Nous reprenons et étendons dans ce chapitre les travaux de Hu [89] et Gesbert [65] définissant un homomorphisme particulier, l'homomorphisme BSP (BH), adapté à la parallélisation de fonctions suivant le modèle de parallélisme quasi-synchrone. Nous verrons que ce squelette peut être utilisé pour implanter des exemples non-triviaux.

7.1 BH : l'homomorphisme BSP

7.1.1 Définition

La définition d'un homomorphisme BSP (ou BH) proposée par Hu et Gesbert est la suivante :

Définition 2 (fonction BH) *Une fonction* $\mathtt{bh}$ *est un homomorphisme BSP, ou BH, si elle peut être exprimée sous la forme*

$$\begin{cases} \mathtt{bh}\ [\mathtt{a}]\ \mathtt{l}\ \mathtt{r} = [\mathtt{k}\ \mathtt{a}\ \mathtt{l}\ \mathtt{r}] & \text{(BH-Singleton)} \\ \mathtt{bh}\ (\mathtt{x} +\!\!+ \mathtt{y})\ \mathtt{l}\ \mathtt{r} = \mathtt{bh}\ \mathtt{x}\ \mathtt{l}\ ((\mathtt{g_r}\ \mathtt{y}) \otimes_\mathtt{r} \mathtt{r})\ +\!\!+ \\ \qquad\qquad\qquad\qquad \mathtt{bh}\ \mathtt{y}\ (\mathtt{l} \oplus_\mathtt{l} (\mathtt{g_l}\ \mathtt{x}))\ \mathtt{r} & \text{(BH-Concat)} \end{cases}$$

pour certains $\mathtt{k}$, $\mathtt{g_l}$, $\mathtt{g_r}$, $\oplus_\mathtt{l}$ *et* $\otimes_\mathtt{r}$.

Le but de cette définition est de permettre le calcul parallèle de toute fonction BH sur une liste répartie par un programme quasi-synchrone. Comme pour la parallélisation d'homomorphismes de liste initialement proposée par Skillicorn, la répartition est représentée implicitement dans cette définition par le fait de *distribuer* la fonction considérée, et donc le calcul, sur la concaténation de listes.

L'homomorphisme BSP, pour calculer sur une sous-liste, utilise ces deux paramètres $\mathtt{l}$ et $\mathtt{r}$ comme des résumés des listes à gauche ($\mathtt{l}$) et à droite ($\mathtt{r}$) de la sous-liste considérée. Ces résumés de listes sont calculés par les fonction $\mathtt{g_r}$ et $\mathtt{g_l}$, les opérateurs $\otimes_\mathtt{r}$ et $\oplus_\mathtt{l}$ sont utilisés pour adjoindre à ces résumés les valeurs initiales des paramètres $\mathtt{l}$ et $\mathtt{r}$.

La définition donnée n'est cependant pas claire sur les propriétés des opérations $\mathtt{k}$, $\mathtt{g_l}$, $\mathtt{g_r}$, $\oplus_\mathtt{l}$ et $\otimes_\mathtt{r}$, et cette définition ne traite pas explicitement du cas de la liste vide. Les deux sous-sections suivantes présentent les clarifications que j'apporte sur ces deux points.

- Après avoir exploré les propriétés déductibles des fonctions BH selon la définition ci-dessus, je propose que la définition soit complétée pour traiter explicitement le cas de la liste vide.
- Je définis ensuite les propriétés des différents opérateurs issus de la décomposition d'une fonction BH nécessaires pour assurer la possibilité d'une parallélisation systématique de ces fonctions (équations $\mathtt{g_r}$-Concat et $\mathtt{g_l}$-Concat).
- Je propose ensuite deux conditions plus faibles mais indépendantes de la définition de la fonction BH à paralléliser (équation $\mathtt{g_r}$-Concat-faible et $\mathtt{g_l}$-Concat-faible).
- Je montre ensuite que le cas particulier utilisé dans nos développements [1], où $\mathtt{g_l}$ (resp. $\mathtt{g_r}$) est un homomorphisme de liste dont l'opérateur associé est $\oplus_\mathtt{l}$ (resp. $\otimes_\mathtt{r}$), vérifie ces conditions.

L'homomorphisme BSP et la liste vide

Le cas de la liste vide n'est pas traité dans la définition proposée, il est pourtant important car le découpage de la liste peut mener à une concaténation de sous-listes parmi lesquelles se trouve la liste vide. Il faut donc, pour définir le squelette algorithmique correspondant aux fonctions BH, être en mesure de traiter ce cas.

Pour faire apparaître la liste vide dans la définition, on peut utiliser la définition de la concaténation qui nous donne $[\mathtt{a}] = [\,] +\!\!+ [\mathtt{a}] = [\mathtt{a}] +\!\!+ [\,]$. On obtient donc :

1. cette restriction est discutée plus en détails à la section 7.1.6

$$
\begin{array}{l}
\quad [\mathtt{k}\ \mathtt{a}\ \mathtt{l}\ \mathtt{r}] \\
=\{\text{par l'équation BH-Singleton}\} \\
\quad \mathtt{bh}\ ([] +\!\!+ [\mathtt{a}])\ \mathtt{l}\ \mathtt{r} \\
=\{\text{par l'équation BH-Concat}\} \\
\quad \mathtt{bh}\ []\ \mathtt{l}\ (\mathtt{r} \otimes_{\mathtt{r}} (\mathtt{g}_{\mathtt{r}}\ [\mathtt{a}])) +\!\!+ \\
\qquad \mathtt{bh}\ [\mathtt{a}]\ (\mathtt{l} \oplus_{\mathtt{l}} (\mathtt{g}_{\mathtt{l}}[]))\ \mathtt{r} \\
=\{\text{par l'équation BH-Singleton}\} \\
\quad \mathtt{bh}\ []\ \mathtt{l}\ (\mathtt{r} \otimes_{\mathtt{r}} (\mathtt{g}_{\mathtt{r}}\ [\mathtt{a}])) +\!\!+ \\
\qquad [\mathtt{k}\ \mathtt{a}\ (\mathtt{l} \oplus_{\mathtt{l}} (\mathtt{g}_{\mathtt{l}}[]))\ \mathtt{r}]
\end{array}
\qquad
\begin{array}{l}
\quad [\mathtt{k}\ \mathtt{a}\ \mathtt{l}\ \mathtt{r}] \\
=\{\text{par l'équation BH-Singleton}\} \\
\quad \mathtt{bh}\ ([\mathtt{a}] +\!\!+ [])\ \mathtt{l}\ \mathtt{r} = \\
=\{\text{par l'équation BH-Concat}\} \\
\quad \mathtt{bh}\ [\mathtt{a}]\ \mathtt{l}\ ((\mathtt{g}_{\mathtt{r}}[]) \otimes_{\mathtt{r}} \mathtt{r})\ +\!\!+ \\
\qquad \mathtt{bh}\ []\ (\mathtt{l} \oplus_{\mathtt{l}} (\mathtt{g}_{\mathtt{l}}\ [\mathtt{a}]))\ \mathtt{l} \\
=\{\text{par l'équation BH-Singleton}\} \\
\quad [\mathtt{k}\ \mathtt{a}\ \mathtt{l}\ ((\mathtt{g}_{\mathtt{r}}[]) \otimes_{\mathtt{r}} \mathtt{r})]\ +\!\!+ \\
\qquad \mathtt{bh}\ []\ (\mathtt{l} \oplus_{\mathtt{l}} (\mathtt{g}_{\mathtt{l}}\ [\mathtt{a}]))\ \mathtt{l}
\end{array}
$$

On peut donc déduire de la définition donnée pour les fonctions BH que les résultats de $\mathtt{bh}\ []\ (\mathtt{l} \oplus_{\mathtt{l}} (\mathtt{g}_{\mathtt{l}}\ [\mathtt{a}]))\ \mathtt{l}$ et $\mathtt{bh}\ []\ \mathtt{l}\ (\mathtt{r} \otimes_{\mathtt{r}} (\mathtt{g}_{\mathtt{r}}\ [\mathtt{a}]))$ sont des listes vides.

De plus, on peut déduire des dérivations précédentes la caractérisation suivante pour les différentes fonctions permettant d'exprimer une fonction BH :

$$\forall\ \mathtt{a}\ \mathtt{l}\ \mathtt{r},\ \mathtt{k}\ \mathtt{a}\ \mathtt{l}\ \mathtt{r}\ =\ \mathtt{k}\ \mathtt{a}\ \mathtt{l}\ ((\mathtt{g}_{\mathtt{r}}[]) \otimes_{\mathtt{r}} \mathtt{r}) = \mathtt{k}\ \mathtt{a}\ (\mathtt{l} \oplus_{\mathtt{l}} (\mathtt{g}_{\mathtt{l}}[]))\ \mathtt{r}. \qquad \text{(BH-nil-k)}$$

Pour satisfaire cette condition, il suffit d'utiliser une fonction $\mathtt{g}_{\mathtt{l}}$ (resp. $\mathtt{g}_{\mathtt{r}}$) dont l'application à la liste vide retourne l'élément neutre à droite (resp. à gauche) pour l'opérateur $\oplus_{\mathtt{l}}$ (resp. $\otimes_{\mathtt{r}}$).

Afin de simplifier la définition des fonctions BH, il est raisonnable d'imposer que pour tout élément $\mathtt{l}$ et $\mathtt{r}$,

$$\mathtt{bh}\ []\ \mathtt{l}\ \mathtt{r} = []. \qquad \text{(BH-nil)}$$

Ainsi quelle que soit la liste considérée, l'application d'une fonction BH produira une liste de même longueur.

Décomposition de BH

Pour paralléliser une fonction BH sur un nombre quelconque de processeurs, il faut pouvoir découper la liste en un nombre quelconque de morceaux. Considérons l'application de $\mathtt{bh}$ à la liste $(\mathtt{x} +\!\!+ (\mathtt{y} +\!\!+ \mathtt{z}))$:

$$
\begin{array}{l}
\quad \mathtt{bh}\ (\mathtt{x} +\!\!+ (\mathtt{y} +\!\!+ \mathtt{z}))\ \mathtt{l}\ \mathtt{r} \\
= \\
\quad \mathtt{bh}\ \mathtt{x}\ \mathtt{l}\ ((\mathtt{g}_{\mathtt{r}}\ (\mathtt{y} +\!\!+ \mathtt{z})) \otimes_{\mathtt{r}} \mathtt{r})\ +\!\!+\ \mathtt{bh}\ (\mathtt{y} +\!\!+ \mathtt{z})\ (\mathtt{l} \oplus_{\mathtt{l}} (\mathtt{g}_{\mathtt{l}}\ \mathtt{x}))\ \mathtt{r} \\
= \\
\quad \mathtt{bh}\ \mathtt{x}\ \mathtt{l}\ ((\mathtt{g}_{\mathtt{r}}\ (\mathtt{y} +\!\!+ \mathtt{z})) \otimes_{\mathtt{r}} \mathtt{r})\ +\!\!+ \\
\quad \mathtt{bh}\ \mathtt{y}\ (\mathtt{l} \oplus_{\mathtt{l}} (\mathtt{g}_{\mathtt{l}}\ \mathtt{x}))\ ((\mathtt{g}_{\mathtt{r}}\ \mathtt{z}) \otimes_{\mathtt{r}} \mathtt{r})\ +\!\!+ \mathtt{bh}\ \mathtt{z}\ ((\mathtt{l} \oplus_{\mathtt{l}} (\mathtt{g}_{\mathtt{l}}\ \mathtt{x})) \oplus_{\mathtt{l}} (\mathtt{g}_{\mathtt{l}}\ \mathtt{y}))\ \mathtt{r}
\end{array}
$$

cependant, la concaténation étant une opération associative, on a également

$$
\begin{array}{l}
\quad \mathtt{bh}\ (\mathtt{x} +\!\!+ (\mathtt{y} +\!\!+ \mathtt{z}))\ \mathtt{l}\ \mathtt{r} \\
= \\
\quad \mathtt{bh}\ ((\mathtt{x} +\!\!+ \mathtt{y}) +\!\!+ \mathtt{z})\ \mathtt{l}\ \mathtt{r}
\end{array}
$$

=

$\mathtt{bh}\ (\mathtt{x} +\!\!+ \mathtt{y})\ \mathtt{l}\ ((\mathtt{g_r}\ \mathtt{z}) \otimes_\mathtt{r} \mathtt{r})\ +\!\!+\ \mathtt{bh}\ \mathtt{z}\ (\mathtt{l} \oplus_\mathtt{l} (\mathtt{g_l}\ (\mathtt{x} +\!\!+ \mathtt{y}))\ \mathtt{r}$

=

$\mathtt{bh}\ \mathtt{x}\ \mathtt{l}\ ((\mathtt{g_r}\ \mathtt{y}) \otimes_\mathtt{r} ((\mathtt{g_r}\ \mathtt{z}) \otimes_\mathtt{r} \mathtt{r}))\ +\!\!+\ \mathtt{bh}\ \mathtt{y}\ (\mathtt{l} \oplus_\mathtt{l} (\mathtt{g_l}\ \mathtt{x}))\ ((\mathtt{g_r}\ \mathtt{z}) \otimes_\mathtt{r} \mathtt{r})\ +\!\!+$
$\mathtt{bh}\ \mathtt{z}\ (\mathtt{l} \oplus_\mathtt{l} (\mathtt{g_l}\ (\mathtt{x} +\!\!+ \mathtt{y}))\ \mathtt{r}$

Pour que la fonction $\mathtt{bh}$ soit décomposable sur n'importe quelle répartition d'une liste, il faut donc qu'elle vérifie, pour tout $\mathtt{l}$, $\mathtt{r}$, $\mathtt{x}$, $\mathtt{y}$ et $\mathtt{z}$ l'équation suivante :

$$\mathtt{bh}\ \mathtt{x}\ \mathtt{l}\ ((\mathtt{g_r}(\mathtt{y} +\!\!+ \mathtt{z})) \otimes_\mathtt{r} \mathtt{r}) +\!\!+ \mathtt{bh}\ \mathtt{y}\ (\mathtt{l} \oplus_\mathtt{l} (\mathtt{g_l}\ \mathtt{x}))\ ((\mathtt{g_r}\ \mathtt{y}) \otimes_\mathtt{r} \mathtt{r}) +\!\!+ \mathtt{bh}\ \mathtt{z}\ ((\mathtt{l} \oplus_\mathtt{l} (\mathtt{g_l}\ \mathtt{x})) \oplus_\mathtt{l} (\mathtt{g_l}\ \mathtt{y}))\ \mathtt{r}$$
$$=$$
$$\mathtt{bh}\ \mathtt{x}\ \mathtt{l}\ ((\mathtt{g_r}\ \mathtt{y}) \otimes_\mathtt{r} ((\mathtt{g_r}\ \mathtt{z}) \otimes_\mathtt{r} \mathtt{r})) +\!\!+ \mathtt{bh}\ \mathtt{y}\ (\mathtt{l} \oplus_\mathtt{l} (\mathtt{g_l}\ \mathtt{x}))\ ((\mathtt{g_r}\ \mathtt{z}) \otimes_\mathtt{r} \mathtt{r}) +\!\!+ \mathtt{bh}\ \mathtt{z}\ (\mathtt{l} \oplus_\mathtt{l} (\mathtt{g_l}(\mathtt{x} +\!\!+ \mathtt{y}))\ \mathtt{r}$$

Cette équation est équivalente aux deux équations suivantes [2] :

$$\mathtt{bh}\ \mathtt{x}\ \mathtt{l}\ ((\mathtt{g_r}\ (\mathtt{y} +\!\!+ \mathtt{z})) \otimes_\mathtt{r} \mathtt{r}) = \mathtt{bh}\ \mathtt{x}\ \mathtt{l}\ ((\mathtt{g_r}\ \mathtt{y}) \otimes_\mathtt{r} ((\mathtt{g_r}\ \mathtt{z}) \otimes_\mathtt{r} \mathtt{r})) \qquad (\mathtt{g_r}\text{-Concat})$$
$$\mathtt{bh}\ \mathtt{z}\ ((\mathtt{l} \oplus_\mathtt{l} (\mathtt{g_l}\ \mathtt{x})) \oplus_\mathtt{l} (\mathtt{g_l}\ \mathtt{y}))\ \mathtt{r} = \mathtt{bh}\ \mathtt{z}\ (\mathtt{l} \oplus_\mathtt{l} (\mathtt{g_l}\ (\mathtt{x} +\!\!+ \mathtt{y}))\ \mathtt{r} \qquad (\mathtt{g_l}\text{-Concat})$$

Pour satisfaire ces équations, il suffit que les égalités suivantes soient vérifiées :

$$(\mathtt{g_r}\ (\mathtt{y} +\!\!+ \mathtt{z})) \otimes_\mathtt{r} \mathtt{r} = (\mathtt{g_r}\ \mathtt{y}) \otimes_\mathtt{r} ((\mathtt{g_r}\ \mathtt{z}) \otimes_\mathtt{r} \mathtt{r}) \qquad (\mathtt{g_r}\text{-Concat-faible})$$
$$\textit{et}$$
$$\mathtt{l} \oplus_\mathtt{l} (\mathtt{g_l}\ (\mathtt{x} +\!\!+ \mathtt{y})) = (\mathtt{l} \oplus_\mathtt{l} (\mathtt{g_l}\ \mathtt{x})) \oplus_\mathtt{l} (\mathtt{g_l}\ \mathtt{y}) \qquad (\mathtt{g_l}\text{-Concat-faible})$$

Une condition suffisante pour satisfaire les équations BH-nil-k, $\mathtt{g_r}$-Concat-faible et $\mathtt{g_l}$-Concat-faible consiste à dire que $\mathtt{g_l}$ et $\mathtt{g_r}$ sont des homomorphismes dont les opérateurs associés sont respectivement $\oplus_\mathtt{l}$ et $\otimes_\mathtt{r}$. Autrement dit, il existe deux fonctions $\mathtt{f_l}$ et $\mathtt{f_r}$ telles que $\mathtt{g_l} = (\!|\mathtt{f_l}, \oplus_\mathtt{l}|\!)$ et $\mathtt{g_r} = (\!|\mathtt{f_r}, \otimes_\mathtt{r}|\!)$. En effet, dans ce cas

$(\mathtt{l} \oplus_\mathtt{l} (\mathtt{g_l}\ \mathtt{x})) \oplus_\mathtt{l} (\mathtt{g_l}\ \mathtt{y})$
=\{par la propriété d'associativité des opérateurs d'homomorphismes \}
$\mathtt{l} \oplus_\mathtt{l} ((\mathtt{g_l}\ \mathtt{x}) \oplus_\mathtt{l} (\mathtt{g_l}\ \mathtt{y}))$
=\{par la propriété d'homomorphisme de $\mathtt{g_l}$\}
$\mathtt{l} \oplus_\mathtt{l} ((\mathtt{g_l}\ (\mathtt{x} +\!\!+\ \mathtt{y}))$

Le même raisonnement tient pour $\mathtt{g_r}$ et $\otimes_\mathtt{r}$. L'équation BH-nil-k est vérifiée car, par définition des homomorphismes, $\mathtt{g_l}$ (resp. $\mathtt{g_r}$) appliquée à la liste vide retourne l'élément neutre pour l'opérateur $\oplus_\mathtt{l}$ (resp. $\otimes_\mathtt{r}$).

7.1.2 Calcul des fonctions BH par un squelette algorithmique

Gesbert propose également un squelette algorithmique capable de calculer les homomorphismes BSP sous certaines conditions et une certification en Coq d'une implantation parallèle de ce squelette en BSML. Cette certification procède de la façon suivante :

2. La preuve de ceci repose sur le fait que toute fonction BH produit une liste de même longueur que la liste prise en paramètre, et pour toute liste $\mathtt{l_1}, \mathtt{l_2}, \mathtt{l'_1}, \mathtt{l'_2}$, si $\mathtt{l_1}$ et $\mathtt{l'_1}$ sont de même longueur alors $\mathtt{l_1} +\!\!+ \mathtt{l_2} = \mathtt{l'_1} +\!\!+ \mathtt{l'_2}$ est équivalent à $\mathtt{l_1} = \mathtt{l'_1}$ et $\mathtt{l_2} = \mathtt{l'_2}$.

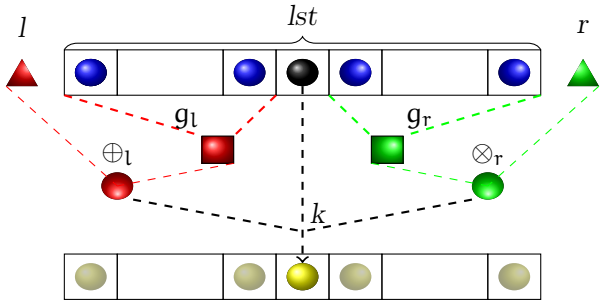


FIGURE 7.1 – *Résultat en un point de la liste de l'application du squelette algorithmique BH*

d'abord une spécification décrit la valeur du $n^{\text{ème}}$ élément de la liste résultante d'un calcul d'une fonction BH ; ensuite une fonction calculant le résultat de cette spécification en tout point d'une liste est écrite ; une implantation parallèle, utilisant une spécification des primitives BSML est écrite ; enfin une preuve d'équivalence de la fonction parallèle avec la fonction séquentielle est donnée. Cette dernière preuve est une preuve de parallélisation correcte non-composable (telle que définie au chapitre 5.1.2).

Je présente ici en détails sa formalisation, puis mes apports qui consistent en :

- une optimisation du calcul effectué par l'implantation parallèle du squelette ;
- deux généralisations de la preuve de correction de l'implantation parallèle : la première relative aux paramètres admissibles par la preuve de correction non-composable, la seconde consistant en une nouvelle preuve, montrant que l'implantation parallèle est une parallélisation correcte *composable* d'une fonction *bh_comp* spécifiant le calcul des fonctions BH.

Afin de simplifier l'écriture des fonctions et des lemmes, les fonctions k, g_l, $\oplus_l$, g_r, et $\otimes_r$ correspondant aux paramètres du squelette BH sont définies comme variables globales, ainsi que les types L, R et A correspondant respectivement aux types des éléments l, r et a de la définition 2 et le type B des éléments de la liste résultant de l'application de BH. Tous les lemmes et les fonctions présentés ici sont donc quantifiés universellement par ces paramètres.

La spécification décrivant le $n^{\text{ème}}$ élément du résultat du calcul d'une fonction BH est représentée graphiquement à la figure 7.1. Sa définition en Coq est la suivante :

```
Definition bh_nth (za : A) (l :L) (lst : list A) (r :R) (n :nat) :=
  k ( l⊕l (gl(firstn n lst))) (nth n lst za) ( (gr(skipn (S n) lst)) ⊗r r).
```

firstn n *lst* construit une liste ne contenant que les n premiers éléments de *lst*, *skipn* $(S\ n)$ *lst* construit une liste en éliminant les $n+1$ premiers éléments de *lst*. *za* sert de valeur par défaut pour *nth* lorsque n n'est pas un indice valide de la liste.

Cette spécification est utilisée pour définir la fonction *bh_comp*, qui calcule la valeur attendue de l'application d'une fonction BH sur une liste complète :

```
Definition bh_comp (l :L) (lst : list A) (r :R) : list B :=
  match lst with
    | nil ⇒ []
```

```
  | a :: lst0 ⇒ map (bh_nth a l (a :: lst0) r) (seq 0 (length (a :: lst0)))
end.
```

Cette fonction, pour une liste donnée, construit la séquence d'entiers correspondant aux positions de la liste (fonction *seq*), puis applique la spécification précédente en chaque point de la séquence. Il est ainsi très simple de prouver que cette fonction respecte bien la spécification donnée.

Une preuve est donnée également que cette fonction se décompose bien suivant le schéma de BH pour une concaténation de listes si la fonction g_l (resp. g_r) est un homomorphisme dont l'opérateur associé est $\oplus_l$ (resp. $\otimes_r$) :

```
Theorem bh_hom (l :L) (x y : list A) (r :R) :
  (is_homomorphism g_l ⊕_l) → (is_homomorphism g_r ⊗_r) →
  bh_comp l (x++y) r = bh_comp l x ( (g_r y) ⊗_r r) ++ bh_comp ( l ⊕_l(g_l x)) y r.
```

Dans [65], cette preuve est présentée comme une preuve que *bh_comp* correspond bien à BH. La terminologie utilisée est un peu floue. Il me semble nécessaire de clarifier que cette preuve ne montre pas que toute fonction BH peut-être calculée par *bh_comp*, elle montre simplement que *bh_comp* est une fonction BH. Je montre à la section 7.1.5 que *bh_comp* permet effectivement de calculer les fonctions BH (sous la même condition sur les opérateurs).

La fonction *bh_comp* est conçue pour spécifier le résultat attendu d'une fonction BH, sans souci d'efficacité. Une autre fonction est proposée, calculant de façon plus efficace le même résultat :

```
Definition bh_seq (l :L) (lst : list A) (r :R) :=
  map (k_tripl l r) (inits_id_tails lst).
```

La fonction *inits_id_tails* remplace chaque élément de la liste par un triplet contenant la liste des éléments qui le précèdent, l'élément lui même, et la liste des éléments qui le suivent. La fonction *k_tripl* utilise ce triplet pour construire les paramètres à passer à la fonction *k*. La fonction g_l est appliquée à l'élément de gauche du triplet, le résultat est combiné à l'élément *l* par l'opérateur $\oplus_l$, on obtient ainsi le deuxième paramètre de *k*. Son premier paramètre est l'élément courant, deuxième élément du triplet. Le résultat de l'application de g_r au troisième élément du triplet est combiné par l'opérateur $\otimes_r$ à *r* pour donner le troisième paramètre de la fonction *k*.

La fonction *bh_seq* est accompagnée d'une preuve qu'elle calcule effectivement le même résultat que *bh_comp* :

```
Lemma bh_seq_bh : ∀ l lst r, bh_seq l lst r = bh_comp l lst r.
```

L'implantation parallèle de BH proposée est la suivante :

```
Definition bh_bsml_comp (l :L) (vl : par (list A)) (r :R) :=
  let comms := bsml_comp_bh_comm l vl r in
    parfun3 (bh_seq)
      (loc_lefts l comms) vl (loc_rights r comms).
```

Le fonctionnement de cette implantation est schématisé par la figure 7.2. La fonction

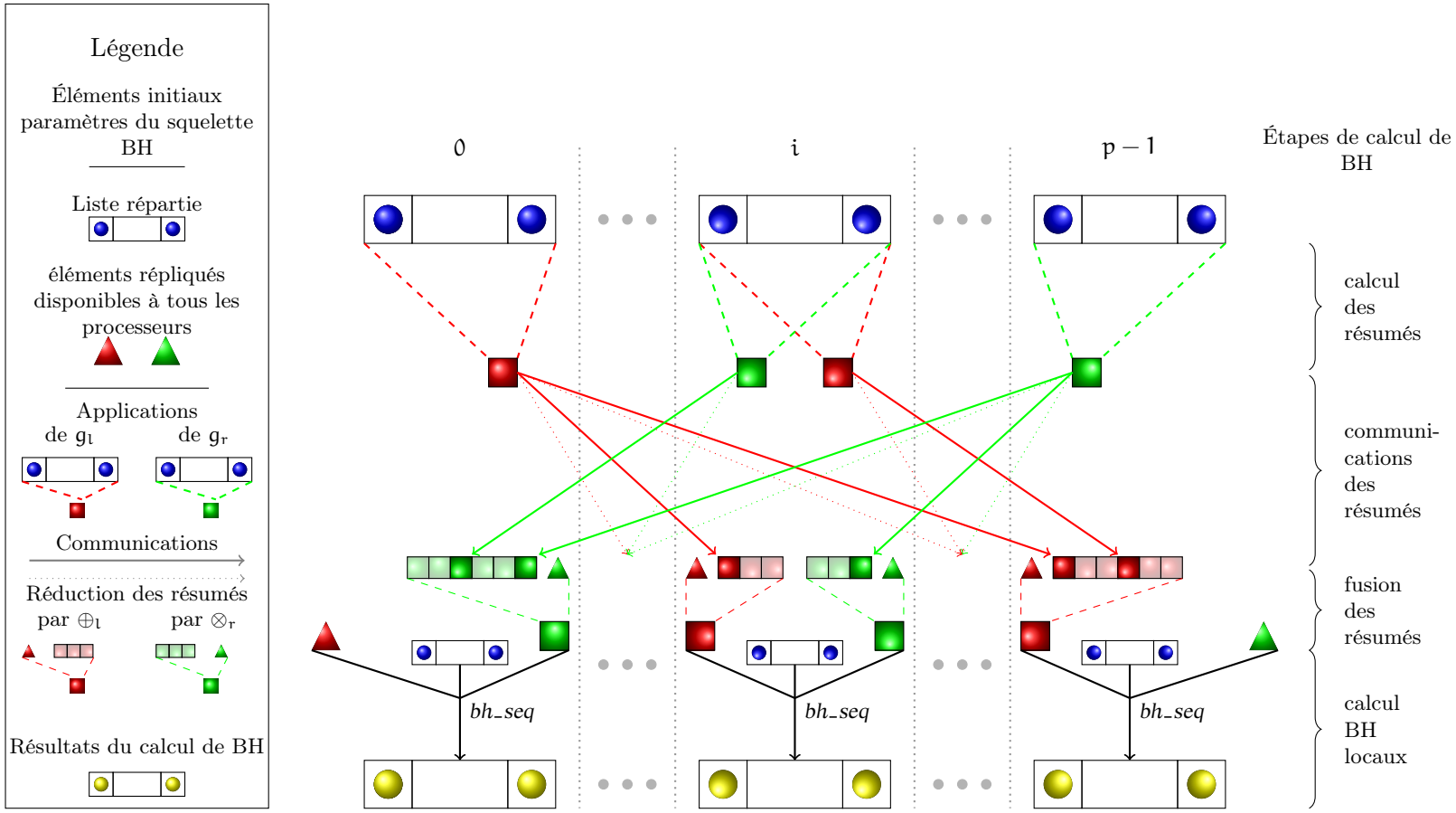


FIGURE 7.2 – *Implantation parallèle du squelette algorithmique BH*

bsml_comp_bh_comm implante les calculs des résumés locaux par application de g_l et g_r à la liste d'éléments, et l'étape de communication (étape 1 et 2 de la figure 7.2). Les fonctions *loc_left* et *loc_right* fusionnent les valeurs reçues à l'aide des opérateurs $\oplus_l$ et $\otimes_r$ (étape 3 de la figure 7.2).

La preuve de correction de l'implantation parallèle suit la notion de parallélisation correcte non-composable, elle est exprimée comme suit :

```
Theorem bh_bsml_bh lst :
 (is_homomorphism gl ⊕l) → (is_homomorphism gr ⊗r) →
   partition_merge (proj' (bh_bsml_comp (gl nil) scatter lst (gr nil))) =
   bh_comp (gl nil) lst (gr nil).
```

La composition des fonctions *partition_merge* et *proj'* est équivalente à la fonction *listOfParList*, la fonction *scatter* répartit la liste de la façon décrite au chapitre précédent. La preuve est donc bien de la forme $join_{list} \circ bh\_bsml\_comp \circ partition_{list} = bh\_comp$.

On remarquera que la preuve est limitée pour les valeurs droite et gauche de l'algorithme BH à l'application des fonctions g_l et g_r à la liste vide. Les valeurs ainsi obtenues sont les neutres des opérateurs $\oplus_l$ et $\otimes_r$.

7.1.3 Optimisation de l'algorithme

L'algorithme séquentiel de calcul de BH n'est pas efficace car il génère la liste des préfixes et des suffixes de la liste à traiter et ensuite réduit ces suffixes et préfixes à l'aide des fonctions g_l et g_r. En fait, cette implantation n'utilise pas le fait que g_l et g_r sont des homomorphismes. En effet, cette propriété nous permet de dire que la réduction du préfixe à une position *i* de la liste peut être calculée à partir de la réduction du préfixe à la position *i-1* et de l'élément à la position *i* : supposons que lst_i soit le préfixe de e_i, le préfixe de e_{i+1} est alors $lst_i +\!\!+ [e_i]$, calculer g_l sur celui-ci revient à calculer $(g_l\ lst_i) \oplus_l (g_l\ [e_i])$ par la propriété d'homomorphisme de g_l. Or, avant de calculer la valeur de g_l pour le préfixe de e_{i+1}, nous avons parcouru récursivement la liste et calculé le résultat de g_l sur les préfixes des éléments précédents, il est donc inutile de le calculer à nouveau, il suffit, lors de l'appel récursif, de passer la valeur calculée précédemment en argument.

Ma version optimisée se sert de la propriété des homomorphismes pour calculer en un parcours la liste des réductions de préfixes en réutilisant la valeur calculée précédemment (réduction de gauche à droite *fold_left*) et, en un autre parcours, la réduction des suffixes (réduction de droite à gauche *fold_right*). Le résultat final est obtenu en appliquant la fonction [3] (*map3 k*) à la liste des éléments et aux deux listes contenant les sommes des préfixes et des suffixes.

J'ai prouvé que la fonction *bh_seq_opt* calcule le même résultat que la fonction *bh_seq*, elle calcule donc le même résultat que la fonction *bh_comp*. La preuve est faite par induction et met un jeu de nombreux lemmes sur les opérations de base utilisées.

La table de la figure 7.3 reprend les différentes implantations du squelette BH.

3. la fonction *map3* prend une fonction f à trois arguments et trois listes $[e_1^1,\ldots, e_n^1]$, $[e_1^2,\ldots, e_n^2]$ et $[e_1^3,\ldots, e_n^3]$ et construit la liste $[f\ e_1^1\ e_1^2\ e_1^3,\ldots, f\ e_n^1\ e_n^2\ e_n^3]$

Implantations séquentielles	spécification	*bh_comp*
	version initialement utilisée pour calculer localement	*bh_seq*
	version optimisée	*bh_seq_opt*
Implantations parallèles	initialement utilisée	*bh_bsml_comp*
	version optimisée utilisant localement *bh_seq_opt*	*bh_bsml_opt_comp*

FIGURE 7.3 – *Différentes implantations du squelette BH*

7.1.4 Preuves de correction généralisées

Les preuves de corrections présentées jusque là reposent sur le fait que les fonctions g_l et g_r sont des homomorphismes dont les opérateurs associés sont $\oplus_l$ et $\otimes_r$. J'ai fait le même pré-supposé dans mes développements, ce choix est discuté à la section 7.1.6.

Généralisation de la preuve de correction non-composable

J'ai initialement étendu la preuve de correction non-composable de la parallélisation de *bh_comp* par *bh_bsml_comp* afin de prendre en compte l'utilisation de valeurs quelconques pour les éléments gauche et droit (paramètres l et r de BH). Ce faisant, j'ai trouvé une erreur dans l'implantation de l'algorithme de calcul parallèle. Une fonction calculant la réduction des résumés reçus prenait deux fois en compte la valeur du paramètre droit r. Tant que cet élément était neutre pour l'opération $\otimes_r$, cela ne posait pas de problème, mais si la version erronée de l'algorithme avait été utilisée avec des éléments gauche et droit non-neutres, le résultat obtenu aurait alors été faux.

Une fois l'algorithme corrigé, j'ai pu généraliser la preuve de correction non-composable à des paramètres l et r quelconques. Le théorème ainsi prouvé restait cependant spécifique à la répartition des données obtenue par la fonction *scatter*.

C'est à partir de ces travaux que s'est posé le problème de la correction de composition de fonctions correctement parallélisées, me menant à la définition de parallélisation correcte composable définie au chapitre 5. J'ai ensuite prouvé que l'algorithme parallèle de calcul de BH était une parallélisation correcte composable de *bh_comp*.

Preuve de correction composable de la parallélisation de BH

Le théorème de parallélisation de BH est le suivant :

```
Theorem bh_bsml_opt_comp_bh : ∀ (l : L) (r : R) (plst : par (list A)),
 (is_homomorphism g_l ⊕_l) → (is_homomorphism g_r ⊗_r) →
   listOfParList (bh_bsml_opt_comp l plst r) =
   bh_comp l (listOfParList ( plst)) r.
```

La fonction *bh_bsml_opt_comp* est une version de la fonction *bh_bsml_comp* où la fonction de calcul locale de BH, *bh_seq*, a été remplacée par la version optimisée présentée à la section 7.1.3.

Contrairement à la version précédente de la preuve de correction de l'algorithme, nous n'avons pas directement accès à la liste séquentielle des éléments. La preuve est quantifiée universellement sur une liste répartie *plst*.

La première étape du raisonnement consiste donc à rendre explicite l'existence de la liste séquentialisée, par l'introduction d'une variable *lst* définie par ***listOfParList*** *plst*. La preuve se fait ensuite par induction sur cette liste *lst*. La difficulté ici consiste à exhiber le lien entre la liste séquentielle sur laquelle nous raisonnons par induction et sa représentation parallèle.

Pour le cas de la liste vide, nous utilisons le fait que si la séquentialisation d'une liste parallèle donne une liste vide alors toutes les sous-listes, à tous les processeurs, sont des listes vides. Il est alors simple de montrer qu'à tous les processeurs la valeur obtenue par le calcul local est une liste vide. La séquentialisation de la liste parallèle résultat est la liste vide, ce qui correspond au résultat obtenu par ***bh_comp***. Notre fonction est donc correcte dans ce cas.

Pour le cas où la liste n'est pas vide, (*lst*=*a*::*lst'*), l'hypothèse d'induction est la suivante :

$$\forall\ (plst : par\ (\mathsf{list}\ A))\ (l : L)\ (r : R),\ \ lst' = listOfParList\ plst \rightarrow \\ listOfParList\ (bh\_bsml\_opt\_comp\ l\ plst\ r) = bh\_comp\ l\ lst'\ r$$

le but est de la forme :

$$\forall\ (plst : par\ (\mathsf{list}\ A))\ (l : L)\ (r : R),\ \ a::lst' = listOfParList\ plst \rightarrow \\ listOfParList\ (bh\_bsml\_opt\_comp\ l\ plst\ r) = bh\_comp\ l\ (a::lst')\ r$$

en utilisant le fait que *a*::*lst'*=*[a]*+⁺ *lst'* et l'équation BH-Concat sur la partie droite, nous obtenons :

$$listOfParList\ (bh\_bsml\_opt\_comp\ l\ plst\ r) = \\ bh\_comp\ l\ [a]\ r\ +\!\!+\!\!^{+}\ \ bh\_comp\ (l \oplus_{\mathsf{l}} (\ g_{\mathsf{l}}\ [a]\ ))\ lst'\ r$$

Pour pouvoir utiliser l'hypothèse d'induction, il nous faut une liste parallèle telle que sa séquentialisation est égale à *lst'*. Cette liste est construite par le lemme suivant :

$$\mathsf{Lemma}\ sub\_par\_list\ (plst : par\ (\mathsf{list}\ A))\ (a\ lst') : \\ listOfParList\ plst = a::lst' \rightarrow \\ \exists\ plst', \exists\ i, \exists\ sublst, \\ \quad listOfParList\ plst' = lst' \\ \quad \wedge \\ \quad \forall\ i', \\ \quad\quad (\ `i' < `i \rightarrow (get\ plst\ i' = \mathsf{nil} \wedge get\ plst'\ i' = \mathsf{nil})) \\ \quad\quad \wedge\ (\ `i' = `i \rightarrow (get\ plst\ i' = a::sublst \wedge get\ plst'\ i' = sublst)) \\ \quad\quad \wedge\ (`i < `i' \rightarrow get\ plst\ i' = get\ plst'\ i').$$

La notation $`i$ représente l'entier correspondant au processeur i.

Ce lemme affirme l'existence d'un processeur i contenant la tête de la liste séquentialisée et tel que tous les processeurs le précédant contiennent une liste vide. Il décrit également une liste répartie *plst'* construite à partir de la liste *plst* selon le schéma de la

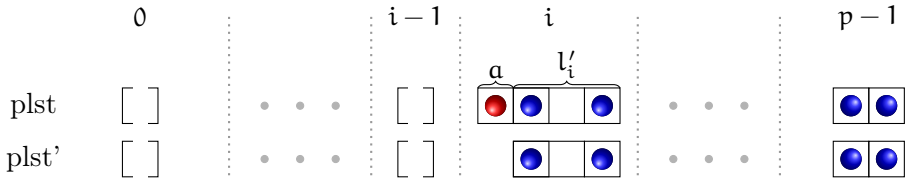


FIGURE 7.4 – *Construction d'une liste parallèle plst' identique à plst sans sa tête a*

figure 7.4. L'hypothèse d'induction appliquée à cette liste *plst'* permet de réécrire le but sous la forme suivante :

listOfParList (*bh_bsml_opt_comp l plst r*) =
bh_comp l [a] ((g_r *lst'*) $\otimes_r$ *r*) ++
listOfParList (*bh_bsml_opt_comp* (*l* $\oplus_l$ (g_l [a])) *plst' r*)

Une tactique que j'ai développée permet ensuite de décomposer les séquentialisations de listes en la concaténation de trois morceaux : le premier morceau est la concaténation des valeurs contenues aux processeurs avant *i*, le second morceau est la valeur contenue au processeur *i* et le troisième correspond aux processeurs supérieurs à *i*.

Par construction, la liste répartie *plst* a pour valeur *a::l'*$_i$ au processeur *i* et *plst'* a pour valeur *lst'*$_i$ à ce même processeur. Tous les processeurs inférieurs à *i* contiennent une liste vide pour *plst* et *plst'*. Pour les processeurs supérieurs à *i*, les valeurs sont identiques sur *plst* et *plst'*.

Il est simple de montrer que le résultat de l'algorithme parallèle sur les premiers processeurs, contenant des listes vides, est une liste vide. La concaténation des valeurs jusqu'au processeur *i* donne donc une liste vide.

Il reste à montrer qu'à partir du processeur *i*, l'algorithme parallèle de calcul de BH donne, après application au vecteur *plst*, la même chose qu'après application au vecteur *plst'* mais avec *l* $\oplus_l$(g_l *a*) en valeur gauche et préfixé par (*bh_comp l [a]* ((g_r *lst'*)$\otimes_r$*r*)). Schématiquement cela donne [4] :

get (*bh_bsml_opt_comp l plst r*) *i* ++
++ $^{p-1}_{i+1}$ *bh_bsml_opt_comp l plst r*
=
bh_comp l [a] ((g_r lst') $\otimes_r$ *r*) ++
get (*bh_bsml_opt_comp* (*l* $\oplus_l$ (g_l [a])) *plst' r*) *i* ++
++ $^{p-1}_{i+1}$ *bh_bsml_opt_comp* (*l* $\oplus_l$ (g_l [a])) *plst' r*

Cette égalité est montrée en deux temps : dans un premier temps on considère la valeur au processeur *i* et on montre que

get (*bh_bsml_opt_comp l plst r*) *i*
=
bh_comp l [a] ((g_r lst') $\otimes_r$ *r*) ++

4. ++ $^{p-1}_{i}$ *plst* représente la concaténation des éléments d'une vecteur parallèle de liste, du processeur *i* au processeur *p-1*

$$get\ (bh\_bsml\_opt\_comp\ (l \oplus_l (\ g_l\ [a]\ ))\ plst'\ r)\ i$$

On montre ensuite que pour tout processeur i' supérieur à i on a

$$get\ (bh\_bsml\_opt\_comp\ l\ plst\ r)\ i'$$
$$=$$
$$get\ (bh\_bsml\_opt\_comp\ (l \oplus_l (\ g_l\ [a]\ ))\ plst'\ r)\ i'$$

Il est ensuite relativement simple de conclure la preuve du théorème.

Pour prouver les deux dernières égalités, un lemme auxiliaire est utilisé, qui décrit les valeurs calculées à un processeur i quelconque.

```
Lemma bh_bsml_opt_comp_bh_aux : ∀ l r plst i,
  (is_homomorphism gl ⊕l) → (is_homomorphism gr ⊗r) →
  get (bh_bsml_opt_comp l plst r) i =
  bh_comp ( l ⊕l (gl(++ 0^(i-1) plst)))
          (get plst i)
          ((gr(++ (i+1)^(p-1) plst)) ⊗r r).
```

Ce lemme est prouvé en utilisant l'évaluation symbolique des primitives BSML et deux lemmes prouvant la correction des communications :

```
Lemma loc_lefts_def' l (plst : par (list A)) (i :processor) :
    (is_homomorphism gl ⊕l) →
      get (loc_lefts l (bsml_comp_bh_comm plst)) i = (l ⊕l (gl(++ 0^(i-1) plst))).
Lemma loc_rights_def' r (plst : par (list A)) (i :processor) :
    (is_homomorphism gr ⊗r) →
      get (loc_rights r(bsml_comp_bh_comm plst )) i = ((gr(++ (i+1)^(p-1) plst))⊗r r).
```

Les preuves de ces deux lemmes se font par induction sur la liste des processeurs à gauche (resp. à droite) du processeur observé. Elles utilisent également l'évaluation symbolique des primitives BSML ainsi que de nombreuses propriétés des opérations séquentielles utilisées pour assembler les résultats des communications.

7.1.5 Support Coq pour la parallélisation de programmes BH

Pour permettre la parallélisation automatique d'une fonction définie à l'aide de *bh_comp*, j'ai défini une instance de la classe Parallel liant la fonction séquentielle *bh_comp* et son implantation parallèle *bh_bsml_opt_comp*.

```
Instance bh_compParallel{L A R B : Type}
  {k : L → A → R → B}
  {gl : list A → L} {opl : L → L → L} {hl :Homomorphism' opl gl}
  {gr : list A → R} {opr : R → R → R} {hr :Homomorphism' opr gr }
   : Parallel
  (fun lst ⇒bh_comp k gl opl gr opr l lst r)
  (fun lst ⇒ bh_bsml_opt_comp k gl opl gr opr l lst r).
```

Cette instance est paramétrée par deux instances d'une classe Homomorphism' qui assurent que les fonctions g_l et g_r sont des homomorphismes dont les opérateurs associés sont $\oplus_l$ et $\otimes_r$, conditions nécessaires à la preuve de correction de la parallélisation. Ainsi, lorsque la méthode de parallélisation décrite au chapitre précédent est utilisée sur une application de la fonction *bh_comp*, le mécanisme de recherche d'instances utilisera l'instance *bh_compParallel* s'il trouve des instances de la classe Homomorphism' pour les opérateurs utilisés.

Comme nous le verrons dans la suite, la définition de BH est très expressive, des fonctions d'ordre supérieur très générales peuvent être parallélisées avec BH et être utilisées ensuite elles-même comme squelettes. Afin que l'intention première de ces fonctions puisse être comprise à la lecture de leur implantation, elles ne sont pas écrites initialement à l'aide de la fonction *bh_comp*.

Je propose deux méthodologies pour montrer qu'une fonction est un homomorphisme BSP : montrer, par transformation, que le résultat de la fonction peut être calculé par la fonction *bh_comp*, ou montrer que la fonction se décompose bien suivant les équations caractérisant les fonctions BH (BH-nil, BH-Singleton, BH-Concat).

Pour chacune de ces méthodologies, des classes de types supports permettent de guider la preuve en réunissant dans une classe l'ensemble des faits à prouver pour pouvoir correctement paralléliser une fonction à l'aide de BH. Une fois les preuves faites pour construire une instance de l'une de ces classes pour une fonction donnée, l'implantation parallèle de celle-ci est construite automatiquement à l'aide de *bh_bsml_opt_comp*.

Parallélisation par transformation

Dans le cas où l'on souhaite paralléliser une fonction f en montrant qu'elle peut s'écrire à l'aide de *bh_comp*, on dispose de deux classes, suivant que les paramètres *gauche* et *droit* de *bh_comp* sont utilisés par la fonction ou non. Si ces paramètres sont utilisés, la fonction f doit être montrée équivalente à *bh_comp* pour tous paramètres `l` (résumé initial de gauche), `r` (résumé initial de droite), et `lst` (la liste à traiter). Une instance de la classe BH_General suivante doit être construite :

```
Class BH_General {L A R B : Type} (f : L → R → list A → list B) :=
{
  BHG_k : L → A → R → B;
  BHG_gl : list A → L;
  BHG_opl : L → L → L;
  BHG_gr : list A → R;
  BHG_opr : R → R → R;
  BHG_spec_match : ∀ l r lst ,
      f l r lst = bh_comp BHG_k BHG_gl BHG_opl BHG_gr BHG_opr l lst r;
  BHG_parallelisation : L → R → par(list A) →par( list B) :=
    (fun l r lst ⇒
     bh_bsml_opt_comp BHG_k BHG_gl BHG_opl BHG_gr BHG_opr l lst r);
  BHG_left_homomorphism : Homomorphism' BHG_opl BHG_gl;
  BHG_right_homomorphism : Homomorphism' BHG_opr BHG_gr
```

```
}.
```

Cette classe regroupe dans son corps toutes les informations nécessaires pour obtenir une parallélisation correcte de la fonction f. Les cinq premiers champs correspondent aux opérations issues de la décomposition de la fonction. Le champ *BHG_spec_match* est une preuve que la fonction f est équivalente à la fonction *bh_comp*. Le champ *BHG_Parallelisation* construit une implantation parallèle à l'aide des éléments précédaents. Les deux derniers champs correspondent aux conditions sur les opérateurs nécessaires pour prouver correcte la parallélisation.

Certaines fonctions sont équivalentes à *bh_comp* pour des arguments *gauche* et *droit* initiaux fixés. Je propose donc une classe support paramétrée par ces constantes :

```
Class BH_Special {A B L R : Type} (f : list A → list B) (l :L) (r :R) :=
 { BHS_k : L → A → R → B;
   BHS_gl : list A → L;
   BHS_opl : L → L → L;
   BHS_gr : list A → R;
   BHS_opr : R → R → R;
   BHS_spec_match : ∀ lst : list A,
     f lst = bh_comp BHS_k BHS_gl BHS_opl BHS_gr BHS_opr l lst r;
   BHS_parallelisation : par(list A) →par( list B) :=
    (fun plst ⇒
     bh_bsml_opt_comp BHS_k BHS_gl BHS_opl BHS_gr BHS_opr l plst r);
   BHS_left_homomorphism : Homomorphism' BHS_opl BHS_gl;
   BHS_right_homomorphism : Homomorphism' BHS_opr BHS_gr
 }.
```

Dans ces deux classes, rien n'assure que la fonction parallèle construite soit une parallélisation correcte de la fonction initiale. Pour s'en assurer, j'ai construit pour chacune d'elles une instance de la classe Parallel. Ces instances de Parallel sont paramétrées par une fonction f et une instance de la classe BH_General ou BH_Special pour cette fonction. Pour montrer correcte la parallélisation de f, il suffit d'utiliser le champ *BHG_spec_match* ou *BHS_spec_match* pour réécrire la fonction en un appel à *bh_comp* puis d'appliquer le théorème *bh_bsml_opt_comp_bh* de parallélisation de BH.

Grâce à ces instances de Parallel paramétrées par une instance de BH_General ou de BH_Special, la recherche automatique d'implantation parallèle présentée au chapitre précédent pourra obtenir l'implantation parallèle d'une fonction si celle-ci à été montrée BH à l'aide de l'une de ces deux classes.

Parallélisation par décomposition

Pour paralléliser une fonction suivant la deuxième méthodologie, j'ai défini une classe BH_PROP décrivant les différentes propriétés que doit respecter une fonction f pour être un homomorphisme BSP parallélisable par notre squelette algorithmique :

```
Class BH_PROP (f : list A → L → R → list B) :=
  {
```

```
  bh_k : L → A → R → B;
  bh_gl : list A → L;
  bh_opl : L → L → L;
  bh_gr : list A → R;
  bh_opr : R → R → R;
  bh_nil : ∀ l r,   f [] l r = [];
  bh_singleton : ∀ a l r,   f [a] l r = [ bh_k l a r ];
  bh_append : ∀ (x y : list A) l r,
      f (x++y) l r =
      f x l (bh_opr (bh_gr y) r) ++ f y (bh_opl l (bh_gl x) ) r
  bh_left_homomorphism : Homomorphism' bh_opl bh_gl;
  bh_right_homomorphism : Homomorphism' bh_opr bh_gr;
}.
```

Définir une instance de cette classe pour une fonction f revient à montrer qu'il existe cinq fonctions *bh_k*, *bh_gl*, *bh_opl*, *bh_gr*, *bh_opr* telles que la fonction f se décompose à l'aide de ces fonctions suivant les équations BH-nil, BH-Singleton et BH-Concat de la définition des homomorphismes BSP. De plus, les deux derniers champs imposent que *bh_gl* (respectivement *bh_gr*) soit un homomorphisme défini à l'aide de l'opérateur *bh_opl* (resp. *bh_opr*), assurant ainsi la correction de la parallélisation.

J'ai prouvé que toute fonction vérifiant ces propriétés peut être parallélisée par *bh_bsml_opt_comp*.

```
Instance BH_PROP_Parallel (f :list A → L → R → list B){bh_prop : BH_PROP f} l r
    : Parallel
      (fun lst ⇒f lst l r)
      (fun lst ⇒ bh_bsml_opt_comp bh_k bh_gl bh_opl bh_gr bh_opr l lst r).
```

La preuve est faite en montrant que la fonction f est calculable par *bh_comp* (induction sur la liste et utilisation des propriétés d'homomorphismes) et en utilisant la preuve de parallélisation correcte de *bh_comp* par *bh_bsml_opt_comp*.

Cette preuve, bien que simple à réaliser, est très importante car elle assure que le squelette algorithmique *bh_comp* et son implantation parallèle sont effectivement en mesure d'implanter toute fonction exprimable suivant la définition 2 des fonctions BH lorsque les propriétés sur les opérateurs sus-cités sont vérifiées.

Cette preuve permet également de définir une instance de la classe Parallel paramétrée par une instance de la classe BH_PROP. Ainsi, toute fonction montrée BH grâce à la classe BH_PROP sera automatiquement parallélisée à l'aide de *bh_bsml_opt_comp*.

Pour paralléliser une fonction intrinsèquement BH, nous disposons maintenant de trois méthodes : soit la fonction est directement écrite à l'aide de *bh_comp* ; soit elle est transformée en un appel à *bh_comp* ; soit on prouve que la fonction se décompose bien selon la caractérisation des fonctions BH (définition 2 + équation BH-nil), et que ses opérations vérifient des conditions suffisantes pour que la parallélisation soit correcte.

La première méthode permet de n'avoir aucune preuve à faire. Cependant, la spécifi-

cation du résultat est donnée directement par une application de *bh_comp* ce qui implique une bonne compréhension de sa sémantique. Transformer une fonction déjà définie en un appel à *bh_comp* nécessite de faire des preuves de transformations qui peuvent s'avérer fastidieuses. La dernière méthodologie proposée simplifie ce problème en effectuant cette transformation automatiquement à partir de la preuve que la fonction se décompose bien suivant le schéma des fonctions BH.

7.1.6 Limitation de ce cadre de développement

Au moment où le code Coq concernant la parallélisation de BH a été écrit, il semblait raisonnable de supposer que les fonctions $\mathtt{g_l}$ et $\mathtt{g_r}$ soient des homomorphismes dont les opérateurs associés sont $\oplus_\mathtt{l}$ et $\otimes_\mathtt{r}$. Cependant, cette restriction semble maintenant excessive.

Prenons l'exemple où $\oplus_\mathtt{l}$ est la soustraction et $(\mathtt{g_l\ l})$ calcule la somme des entiers contenus dans la liste $\mathtt{l}$, c'est à dire $\mathtt{g_l}{=}(\!|\mathtt{id}, (+)|\!)$. L'égalité suivante est vérifiée :

$$\mathtt{l} - (\mathtt{g_l\ x}) - (\mathtt{g_l\ y}) = \mathtt{l} - ((\mathtt{g_l\ x}) + (\mathtt{g_l\ y})) = \mathtt{l} - (\mathtt{g_l\ (x +\!\!+ y)}).$$

L'équation ($\mathtt{g_l}$-Concat-faible) est donc satisfaite par cet exemple simple dans lequel $\oplus_\mathtt{l}$ n'est pas l'opérateur associé à l'homomorphisme $\mathtt{g_l}$. Les opérations $\oplus_\mathtt{l}$ et $\mathtt{g_l}$ sont donc, *a priori*, utilisables pour décomposer une fonction BH et la paralléliser, cependant les restrictions que nous avons utilisées dans nos preuves ne permettent pas d'utiliser ces opérations telles quelles.

Il est cependant possible d'amener une telle instance de BH au cas restreint. De manière générale, si $\mathtt{g_l}$ est un homomorphisme, $\mathtt{g_l}$ appliqué à la liste vide est un élément neutre pour l'opérateur associé à $\mathtt{g_l}$ (appelons le $\oplus_\mathtt{l}$). Calculer $(\mathit{bh\_comp}\ k\ \mathtt{g_l} \odot \mathtt{g_r} \otimes_\mathtt{r} \mathtt{l\ lst\ r})$ revient à calculer la fonction $(\mathit{bh\_comp}\ k'\ \mathtt{g_l} \oplus_\mathtt{l} \mathtt{g_r} \otimes_\mathtt{r} \iota_{\oplus_\mathtt{l}}\ \mathtt{lst\ r})$ où l'opérateur $\oplus_\mathtt{l}$ utilisé au lieu de $\odot$, k' est défini par $(\mathtt{fun\ l'\ lst'\ r'} \Rightarrow k\ (\mathtt{l} \oplus_\mathtt{l} \mathtt{l'})\ \mathtt{lst'\ r'})$ et $\iota_{\oplus_\mathtt{l}}$ est $(\mathtt{g_l}\ [])$.

La preuve de correction de BH pourrait être généralisée afin de découpler les homomorphismes $\mathtt{g_l}$ et $\mathtt{g_r}$ des opérateurs $\oplus_\mathtt{l}$ et $\otimes_\mathtt{r}$.

La nécessité pour $\mathtt{g_l}$ et $\mathtt{g_r}$ d'être des homomorphismes est plus difficile à éliminer car l'optimisation de la fonction *bh_seq* repose sur cette propriété.

7.1.7 Coût BSP de l'implantation parallèle de BH

Nous utilisons ici le modèle de coût de BSP pour discuter de la complexité de notre implantation parallèle. L'implantation BSML de BH appliquée aux paramètres $\mathtt{k}$, $\mathtt{gl}$, $\oplus_\mathtt{l}$, $\mathtt{g_r}$, $\otimes_\mathtt{r}$, $\mathtt{l}$, $\mathtt{r}$ et $\mathtt{lst}$ (supposés être des valeurs) a la complexité suivante : le calcul est fait en une super-étape complète suivie de calculs locaux asynchrones. Chaque processeur $\mathtt{i}$ a une partie contiguë de la liste $\mathtt{lst}$, qui sera notée $\mathtt{lst_i}$.

Le coût BSP de l'application de BH est :

$$seq_1 + \max_{0 \leq \mathtt{i} < \mathtt{p}} (\max(\mathtt{h_i^+}, \mathtt{h_i^-})) \times \mathtt{g} + \mathtt{L} + seq_2$$

La première phase calcule sur chaque processeur, les résumés des valeurs détenues par le processeur : l'un à envoyer aux processeurs à sa gauche (calculé par $\mathtt{gl}$), l'autre à envoyer aux processeurs de droite (calculé par $\mathtt{gr}$) :

$$seq_1 = \max_{0 \le i < p} (\overline{g_l\ lst_i} + \overline{g_r\ lst_i}).$$

Où $\overline{e}$ est le coût du calcul de l'expression e.

Ensuite, chaque processeur i envoie $l_i = (g_l\ lst_i)\downarrow$ aux processeurs de gauche et $r_i = (g_l\ lst_i)\downarrow$ aux processeurs de droite ($(e)\downarrow$ est la valeur obtenue après réduction de l'expression e). Ainsi la taille des données échangées est :

$$\begin{cases} h_i^+ &= i \times |l_i| + (p-1-i) \times |r_i| \\ h_i^- &= \sum_{j=0}^{i-1} |r_j| + \sum_{j=i+1}^{p-1} |l_j| \end{cases}$$

Le reste du calcul asynchrone se fait de la façon suivante : premièrement, sur chaque processeur la liste des résumés de gauche (resp. de droite) reçus est réduite par l'opérateur $\oplus_l$ (resp. $\otimes_r$). Ensuite un calcul local, séquentiel, de BH est effectué comme décrit précédemment. Le coût à un processeur i est donc celui de la réduction des résumés reçus des processeurs de gauche que l'on notera $\overline{/\oplus_{li}}$ plus la réduction des résumés reçus des processeurs de droites $\overline{/\otimes_{ri}}$ plus le coût de calcul de *bh_seq_opt* sur la sous-liste locale avec les résumés locaux $\overline{\mathtt{bh\_seq\_opt}\ l_i\ lst_i\ r_i}$. On a donc

$$seq2 = \max_{0 \le i < p} \overline{/\oplus_{li}} + \overline{/\otimes_{ri}} + \overline{\mathtt{bh\_seq\_opt}\ l_i\ lst_i\ r_i}$$

Dans le cas où $\oplus_l$, $\otimes_r$, et k ont une complexité constante, et g_r et g_l ont une complexité linéaire par rapport à la taille de la liste traitée, seq_1 et seq_2 ont pour complexité $\mathcal{O}(\max_{0 \le i < p} \mathtt{length}\ lst_i)$.

7.2 Dérivation de squelettes plus spécifiques vers BH

Le squelette algorithmique calculant les fonctions BH est un squelette très expressif, permettant de calculer de nombreuses fonctions. Gesbert propose ainsi la construction d'autres squelettes, plus spécifiques, à l'aide de BH. J'ai prouvé correctes ces constructions dans notre formalisation Coq. Cette section décrit les squelettes proposés ainsi que les preuves de correction.

7.2.1 mapAround

Le premier squelette proposé, *mapAround*, est très général. La seule différence avec le squelette BH est qu'il n'utilise pas de valeur l et r. La sémantique informelle du squelette *mapAround* est la suivante :

$$mapAround\ f\ [x_1, \ldots, x_i, \ldots, x_n] = [\ldots, f([x_1, \ldots, x_{i-1}], x_i, [x_{i+1}, \ldots, x_n]), \ldots]$$

Séquentiellement, le squelette est implanté par la fonction suivante :

```
Definition mapAround (f : list A × A × list A→ B) (lst : list A) : list B :=
map f (zip3 (inits lst) lst (tails lst)).
```

où la fonction *zip3* construit une liste de triplet à partir de trois listes, *inits* construit la liste des préfixes stricts d'une liste ($\texttt{inits}\ [x_1, x_2, \ldots, x_n] = [[], [x_1], \ldots, [x_1, \cdots, x_{n-1}]]$), et *tails* la liste des suffixes stricts ($\texttt{tails}\ [x_1, x_2, \ldots, x_n] = [[x_2, \cdots, x_n], \ldots, [x_1], []]$).

La parallélisation par le squelette BH proposée procède comme suit :

Lemme 7.1 (Parallélisation de *mapAround* par BH) *Pour une fonction* $h = mapAround\ f$ *si nous pouvons décomposer* f *en* $f\ (ls, x, rs) = k\ (g_1\ ls, x, g_2\ rs)$, *où* k *est une fonction quelconque et* g_i *est un quasi-homomorphisme, composition d'une fonction* π_i *avec un homomorphisme* $h_i = (\!|k_i, \oplus_i|\!)$, *alors*

$$h\ xs = \mathrm{BH}\ k'\ (\!|k_2, \oplus_2|\!)\ (\!|k_1, \oplus_1|\!)\ xs\ \iota_{\oplus_1}\ \iota_{\oplus_2}$$

$$\text{où} \begin{cases} k'\ (l, x, r) = k(\pi_1\ l, x, \pi_2\ r), \\ \iota_{\oplus_1}\ \textit{est l'élément neutre (à gauche) de}\ \oplus_1, \\ \iota_{\oplus_2}\ \textit{est l'élément neutre (à droite) de}\ \oplus_2\,. \end{cases}$$

La preuve a été faite en Coq par cas sur la liste d'entrée xs. Pour les éléments neutres $\iota_{\oplus_i}$, il suffit de prendre les valeurs $(h_i\ [])$ puisque $\oplus_i$ est l'opérateur associé à l'homomorphisme h_i.

Support Coq pour *mapAround*

Afin de pouvoir paralléliser simplement des fonctions exprimées à l'aide de *mapAround*, j'ai défini une classe support, MapAround_BHable, pour la parallélisation de fonctions à l'aide de *mapAround*. Cette classe fonctionne de la même façon que la classe BH_Special de parallélisation par le squelette *bh_comp*. Elle décrit les conditions nécessaires pour qu'une fonction calculable par le squelette *mapAround* soit parallélisée comme montré dans le lemme 7.1. La construction de l'implantation parallèle de la fonction repose sur une instance de la classe BH_Special paramétrée par la classe MapAround_BHable.

7.2.2 Parallélisation de la somme des préfixes

Le squelette algorithmique *mapAround* peut, à son tour, être utilisé pour la construction du squelette algorithmique *scan* qui calcule la somme des préfixes généralisée à tout opérateur $\odot$ associatif. Ce squelette est utile pour de nombreuses applications (recensées dans [18]). Informellement, la fonction *scan* produit le résultat suivant :

$$scan\ (\odot)\ [x_1, x_2, \ldots, x_n] = [x_1, x_1 \odot x_2, \ldots, x_1 \odot x_2 \odot \cdots \odot x_n]$$

C'est un cas particulier de *mapAround* qui n'utilise pas la liste à droite d'un élément, en effet $scan\ (\odot) = mapAround\ f$ où la fonction f est définie par $f(ls, x, rs) = k\ (sum\ \odot\ \iota_\odot\ ls\ ,\ x\ ,\ (\texttt{fun}\ \_\ \Rightarrow\ [])rs)$.

La fonction $\mathtt{sum}$ calcule la réduction d'une liste par un opérateur : $\mathtt{sum}\ \odot\ \iota_\odot = (\!|\iota_\odot, \mathtt{id}, \odot|\!)$; la fonction $\mathtt{k}$ est définie par $k\ x\ y\ z\ =\ x \odot y$. Bien que la liste à droite de l'élément ne soit jamais utilisée, il faut fournir un homomorphisme pour la traiter. Les valeurs calculées par cet homomorphisme seront transférées d'un processeur à l'autre, c'est pourquoi nous prenons la fonction renvoyant toujours une liste vide afin de minimiser le coût des communications.

Dans le développement Coq la fonction *scan* est définie comme suit :

```
Definition scan (lst : list A) : list A :=
  map (sum ⊙ ι⊙) ( prefix lst).
```

La fonction *prefix* calcul la liste des préfixes en incluant l'élément courant ($\mathtt{prefix}\ [x_1, x_2, \ldots, x_n] = [[x_1], [x_1, x_2], \ldots, [x_1, \cdots, x_n]]$). Une instance de la classe MapAround_BHable est ensuite construite. Cette instance est paramétrée par une instance de la classe Associative pour l'opérateur $\odot$ et une instance d'une classe Neutral permettant de s'assurer que $\iota_\odot$ est bien neutre pour l'opérateur $\odot$.

L'instance de la classe Homomorphism' montrant que *sum* est un homomorphisme, nécessaire à la construction de l'instance de MapAround_BHable, est construite à l'aide des instances des classes Associative et Neutral.

La fonction *scan* peut donc être parallélisée automatiquement (à l'aide de *mapAround*, et donc de *bh_bsml_opt_comp*) pour tout opérateur associatif muni d'un élément neutre.

Coût BSP de l'implantation de *scan* par BH

En utilisant la formule de coût donnée à la section 7.1.7, il est possible d'obtenir un modèle de coût pour la fonction *scan*. L'homomorphisme calculant les valeurs envoyées à gauche retourne toujours la liste vide en temps constant. La liste vide n'étant pas communiquée par BSML, la taille des messages envoyés à droite est nulle. Le coût de la première phase de calcul est donc

$$seq_1 = \max_{0 \le i < p} (\mathtt{sum}\ \odot\ \iota_\odot\ lst_i)$$

et le coût de la phase de communication est

$$\begin{cases} h_i^+ &= i \times |s_i| \\ h_i^- &= \sum_{j=0}^{i-1} |s_j| \end{cases}$$

où $s_i = \mathtt{sum}\ \odot\ \iota_\odot\ lst_i$.

Si l'opération $\odot$ est calculée en temps constant, le coût BSP de *scan* est :

$$\mathcal{O}(\max_{0 \le i < p} \mathtt{length}\ lst_i + \max_{0 \le i < p} \max(i \times |s_i|, \sum_{j=0}^{i-1} |s_j|) \times g + L)$$

Ce coût correspond au coût de l'implantation usuelle en une super-étape de la fonction *scan*.

7.2.3 Parallélisation des homomorphismes à l'aide de BH

Gesbert propose de paralléliser tout homomorphisme $\mathtt{h} = (\!|\odot, \mathtt{k}|\!)$ à l'aide de BH. Le calcul proposé est le suivant :

$$\mathtt{h} = last \circ mapAround\ \mathtt{f}$$

où $\mathtt{f}(ls, \mathtt{x}, rs) = (\mathtt{h}\ \mathtt{ls}) \odot (\mathtt{k}\,\mathtt{x})$ et *last* retourne le dernier élément d'une liste.

En Coq, l'implantation séquentielle de *last* prend une valeur par défaut à retourner dans le cas où la liste est vide. Pour le calcul de l'homomorphisme il suffit d'utiliser $\iota_\odot$ comme valeur par défaut.

```
Lemma last_homomorphism_prefixes_correct :
 ∀ (A B :Type) (op :B→B→B) h, Homomorphism' op h →
   ∀ l, h l = last (homomorphism_prefixes op h l) (h []).
```

La fonction *homomorphism_prefixes* est définie par *mapAround f*, elle calcule la liste des valeurs de h pour l'ensemble des préfixes d'une liste.

De plus, pour que la parallélisation de l'homomorphisme soit correcte, il faut avoir une implantation parallèle correcte *lastPar* de la fonction *last*.

Pour la parallélisation de *last*, le problème se pose de savoir comment doit être distribuée la valeur résultat à la fin du calcul de *lastPar*. Dans le cas général, on ne connaît pas la structure du type résultat, nommons le A et on ne sait pas comment il se répartit. Pour pouvoir intégrer la fonction *lastPar* dans notre cadre, il faut toutefois disposer d'une instance de Partitionnable pour A. Pour cela, nous construisons une instance un peu artificielle qui prend comme fonction de séquentialisation la fonction qui renvoie l'élément présent au premier processeur, et pour fonction de parallélisation, la fonction qui place l'élément à répartir sur le premier processeur, et place sur tous les autres processeurs l'élément $\iota_\odot$. La fonction *lastPar* envoie le dernier élément de la liste sur le premier processeur, tous les autres processeurs prennent la valeur $\iota_\odot$ à la fin du calcul.

Cette astuce permet de construire l'instance suivante de la classe Parallel :

```
Instance homomorphism_Parallel {homo_h :Homomorphism' op h } :
      Parallel h (homoPar).
```

où la fonction *homoPar* est la composition de la fonction *lastPar* et de la parallélisation de *homomorphism_prefixes* à l'aide BH. Ainsi, tout homomorphisme pourra être parallélisé.

Par exemple, la fonction *fold_left*, lorsqu'elle est utilisée avec un opérateur associatif et un élément initial neutre pour l'opérateur utilisé, est un homomorphisme. Ainsi, l'instance *fold_leftParCorrect* utilisée à la section 6.1 est construite à l'aide de l'instance *homomorphism_Parallel*.

Cette implantation n'est certes pas très efficace en terme de consommation mémoire, puisqu'elle conserve la liste de tous les résultats intermédiaires en mémoire avant d'appliquer *lastPar*, mais elle nous permet d'avoir très facilement une implantation certifiée correcte pour tout homomorphisme de liste.

7.3 Conclusion

Nous disposons avec BH d'un squelette algorithmique très expressif. Ma contribution, présentée dans ce chapitre, a consisté à clarifier la définition des fonctions BH afin de pouvoir paralléliser ces fonctions de manière systématique. J'ai corrigé l'implantation parallèle proposée par Gesbert, puis montré qu'il s'agissait d'une parallélisation correcte composable. J'ai défini des classes de type permettant de guider la parallélisation de fonctions à l'aide de BH. L'une de ces classes reprend la définition des fonctions BH sous la forme d'un système d'équation. J'ai montré que la définition d'une instance de cette classe pour une fonction `f` permettait de construire une parallélisation correcte de `f` en passant par le squelette *bh_comp*. Cette preuve montre que si les opérateurs utilisés vérifient certaines conditions, *bh_comp* est effectivement en mesure d'exprimer toutes les fonctions BH et que toute fonction BH est parallélisable par *bh_bsml_opt_comp*. Enfin j'ai implanté en Coq la parallélisation automatique des squelettes *mapAround*, *scan* ainsi que la parallélisation de tout homomorphisme de liste.

Le chapitre suivant montre des exemples d'utilisations de ces squelettes, de la spécification d'un programme à son exécution sur une machine parallèle.

Applications et Expérimentations

8

Sommaire

Dans ce chapitre, nous présentons la construction de programmes parallèles corrects à l'aide des outils mis en place dans les chapitres précédents. Les programmes construits sont ensuite extraits vers OCaml puis importés dans un programme mesurant leur temps d'exécution. Nous commençons par donner quelques détails sur le protocole expérimental suivi avant de passer aux différents exemples.

8.1 Protocole de mesure de performance

Nos programmes sont toujours implantés en Coq dans un foncteur prenant en paramètre un module contenant l'implantation des primitives BSML. Après avoir extrait ce foncteur en OCaml, il sera appliqué au module `BsmlNat` présenté à la section 3.1.6.

Le foncteur contenant le code peut également être paramétré par un module fournissant des types et des opérations paramétrant le problème traité. Par exemple, le programme de diffusion de la chaleur présenté au chapitre 5.2 est paramétré par un module contenant une représentation des nombres, et les opérations usuelles sur ceux-ci. L'abstraction ainsi obtenue après application du foncteur fournit des programmes dont la

structure algorithmique a été prouvée correcte, mais qui utilisent des types et des opérations non disponibles en Coq, comme la fonction d'affichage utilisée dans le test de la chaîne de production de programmes (section 4.3.2) ou plus simplement les nombres flottants et leur opérations associées.

Une fois le foncteur appliqué à tous les modules nécessaires, l'implantation parallèle qu'il contient est utilisée dans une boucle sur des données générées aléatoirement dont la taille est fixée au lancement du programme. Tous les programmes sur lesquels nous avons expérimenté calculent sur des listes et ont une complexité indépendante de la valeur des données contenues dans la liste, nous n'avons donc pas à prêter particulièrement attention aux valeurs générées, seule la taille de la liste est significative. En plus de cette boucle interne au programme, celui-ci est exécuté à plusieurs reprises afin de calculer un temps moyen.

8.1.1 Gestion de la mémoire

Nous avons initialement essayé de compter séparément le temps de collecte mémoire [1], pour cela une collecte mémoire complète [2] est effectuée après chaque itération interne. Le temps d'exécution de ces collectes a été mémorisé séparément lors des expériences, cependant la collecte mémoire peut être déclenchée à tout moment de l'exécution lorsqu'une allocation mémoire à lieu. De ce fait, il est impossible d'avoir une mesure correctement séparé du temps de collecte, c'est pourquoi les temps que nous présentons sont la somme du temps d'exécution de la fonction et du temps d'exécution de la collecte mémoire déclenchée après le calcul.

Afin d'éviter de déclencher le ramasse-miette d'OCaml trop souvent, nous augmentons la taille du tas mineur [3] à une valeur proche de la taille de la mémoire disponible.

8.1.2 Gestion des architectures hétérogènes

Dans nos expériences sur la grappe de calcul MIREV et sur le centre de calcul CCSC, lorsque plusieurs nœuds sont utilisés et que, sur ces nœuds, plus d'un cœur par processeur est utilisé, les performances s'effondrent. Ce problème semble dû à l'implantation de MPI ou des drivers réseaux qui gèrent mal les accès concurrents au réseau par plusieurs cœurs d'un même processeur. Nous utilisons donc dans nos mesures un seul cœur par processeur lorsque plusieurs nœuds sont utilisés.

8.2 Diffusion de la chaleur - Implantation à l'aide de BH

À la section 5.2, nous avons présenté l'implantation d'un calcul de diffusion de la chaleur dans un corps uni-dimensionnel. Afin de simplifier les preuves de correction des

1. garbage collection
2. full major G.C.
3. minor heap

communications, cette implantation est restreinte à une certaine forme de distribution de données. Nous allons voir ici comment le problème de diffusion de la chaleur peut être implanté à l'aide du squelette algorithmique BH, puis nous discuterons de l'efficacité de cette implantation.

8.2.1 Définition à l'aide de BH

Du point de vue séquentiel, nous reprenons exactement les définitions de la section 5.2, le programme *heatEquationSeq* nous sert de spécification séquentielle. En observant le lemme *heatEquationSeqApp*, on peut voir que sa forme est très proche de la forme de l'équation BH-Concat. La fonction *heatEquationSeq* est décomposée sur une concaténation de listes et calcule sur chaque sous-liste la même fonction avec simplement une variation sur la borne gauche ou droite. Ces valeurs aux bornes sont re-calculées en fonction de la sous-liste gauche ou droite à l'aide des fonctions *last* et *hd*. Nous serions donc tenté d'appliquer directement la définition des fonctions BH en prenant pour le paramètre g_l la fonction *last* et pour le paramètre g_r la fonction *hd*, mais ces fonctions ne sont pas des homomorphismes. Cependant, les fonctions *last_option* et *hd_option* sont des homomorphismes. Ces deux fonctions n'utilisent pas de valeur par défaut pour traiter le cas de la liste vide mais retournent un élément de type option. Ainsi la fonction *last_option* définie par

```
Definition last_option (A :Type) (lst : list A) :=
  match lst with
    [] ⇒ None
    | a::l ⇒ Some (List.last lst a )
  end.
```

est un homomorphisme associé à l'opérateur suivant :

```
Definition right_option A (a : option A) (b :option A) :=
  match b with
    None ⇒ a
    | Some _ ⇒ b
  end.
```

On définit de façon similaire l'homomorphisme *hd_option* associé à l'opérateur *left_option*. L'élément neutre pour *left_option* et *right_option* est la valeur None.

Pour éliminer l'encapsulation d'une valeur par le type option, nous définissons l'opération *no_some_default* qui utilise une valeur par défaut pour traiter le cas None :

```
Definition no_some_default (A :Type) (a : option A) (d : A) :=
  match a with
    None ⇒ d
    | Some a ⇒ a
  end.
```

Il est maintenant possible de définir une fonction compatible avec la définition des fonctions BH :

```
Definition heatEquationSeqCompat (gbL gbR : number) (l r :option number)
                                                    dt dx κ u :=
  heatEquationSeq (no_some_default l gbL) (no_some_default r gbR) dt dx κ u.
```

Cette fonction utilise deux paramètres de type option, elle peut donc se décomposer en utilisant *last_option* et *hd_option*. Pour éliminer l'encapsulation du type option afin de passer les paramètres à la fonction *heatEquationSeq*, deux variables globales *gbL* et *gbR* sont utilisées pour les cas où l'un des éléments est de type None. Pour que cette nouvelle fonction calcule la même chose que *heatEquationSeq*, il faudra appeler avec *l*=Some *gbL* et *r*=Some *gbR*.

La décomposition de cette fonction sur la concaténation de liste est définie comme suit :

```
Lemma heatEquationSeqCompatApp (gbL gbR : number) :
 ∀ (l r : option number) (dt dx κ : number)(u1 u2 : list number),
  heatEquationSeqCompat gbL gbR l r dt dx κ (u1 ++ u2)
 =
  (heatEquationSeqCompat gbL gbR l (left_option (hd_option u2) r) dt dx κ u1)
 ++
 (heatEquationSeqCompat gbL gbR (right_option l (last_option u1)) r dt dx κ u2).
```

La preuve de cette décomposition réutilise le lemme *heatEquationSeqApp* et procède par cas sur les résultats de (*hd_option u2*) et (*last_option u1*).

Nous pouvons maintenant construire une instance de la classe BH_PROP pour la fonction *heatEquationSeqCompat*. Nous utilisons pour cela notre système de preuve par calcul de programme. La tactique `Begin` permet de retarder l'instanciation des champs *bh_k*, *bh_gl*, *bh_opl*, *bh_gr*, *bh_opr*. Cinq sous-buts sont générés : les trois premiers correspondent à la décomposition de la fonction suivant les équations BH-nil, BH-Singleton, BH-Concat, les deux sous-buts suivants consistent à prouver que les fonctions utilisées dans la décomposition sont des homomorphismes (champs *bh_left_homomorphism* et *bh_right_homomorphism* de la classe).

```
Instance heatEqCompat_bh (left right dt dx κ : number) :
 BH_PROP
  (fun u lBound rBound⇒
    heatEquationSeqCompat left right lBound rBound dt dx κ u).
Begin.

check GOAL is (heatEquationSeqCompat left right l r dt dx κ [] = []).
[].

BeginEvar.
check GOAL is
```

```
    (heatEquationSeqCompat left right l r dt dx κ [a] = [EVAR l a r]).
  LHS
=={unfold heatEquationSeqCompat}
  (heatEquationSeq (no_some_default l left) (no_some_default r right) dt dx κ [a]).
=={simpl}
  ([heatEquationFormula dt dx κ a (no_some_default l left)
                                  (no_some_default r right)]).
=={simpl}
  ([(fun l a r ⇒ heatEquationFormula dt dx κ a (no_some_default l left)
                                               (no_some_default r right)) l a r]).
[].
endEvar.

BeginEvar.
check GOAL is (
  heatEquationSeqCompat left right l r dt dx κ (x ++ y)
  =
  heatEquationSeqCompat left right l (EVAR (EVAR0 y) r) dt dx κ x
  ++
  heatEquationSeqCompat left right (EVAR1 l (EVAR2 x)) r dt dx κ y
).
  LHS
=={rewrite heatEquationSeqCompatApp}
  ( heatEquationSeqCompat left right l
    (left_option (hd_option y) r) dt dx κ x
    ++
  heatEquationSeqCompat left right
    (right_option l (last_option x)) r dt dx κ y).
[].
endEvar.
   (* Il reste ensuite à indiquer que les opérations utilisées sont bien
des homomorphismes. Les instances de Homomorphism' correspondantes aux
opérations utilisées sont définies dans notre bibliothèque, elles sont
utilisées à l'aide de la tactique de recherche d'instances de classes :
*)
  typeclasses eauto.
  typeclasses eauto.
Qed.
```

Cette instance de BH_PROP nous permet de construire une instance de la classe Parallelizable. Nous utilisons partiellement notre système de preuve par calcul de programme.

```
Instance heatEqParConstr (leftBound rightBound dt dx κ : number) :
  Parallelizable (fun u ⇒ heatEquationSeq leftBound rightBound dt dx κ u).
Begin.
```

```
BeginEvar.
check GOAL is
    (join (EVAR par_a) = heatEquationSeq leftBound rightBound dt dx κ (join par_a)).
RHS
={simpl}
( heatEquationSeqCompat leftBound rightBound (Some leftBound) (Some rightBound)
dt dx κ (join par_a)).
rewrite ← (spec_match (Parallelizable :=(BH_PROP_Parallelizable _ _ _))).
[]. endEvar.
Defined.
```

La tactique `Begin` introduit en tant que variable exisentielle le champ *parallel* qui contiendra l'implantation parallèle de la fonction *heatEquationSeq*. Le sous-but généré nous demande de prouver la propriété *spec_match* de la classe Parallelizable. Nous transformons d'abord notre fonction séquentielle en un appel à *heatEquationSeqCompat* pour pouvoir utiliser la parallélisation par BH_PROP construite précédemment. Nous utilisons ensuite le champ *spec_match* de la classe Parallelizable pour réécrire la partie droite de l'équation en un appel à la fonction *parallel*. Cette transformation utilise l'instance *heatEqCompat_bh* pour paralléliser la fonction *heatEquationSeqCompat*.

Ici la recherche d'instance n'arrive pas à construire automatiquement l'instance de Parallelizable correspondant à la fonction *heatEquationSeqCompat*, il faut la guider en lui donnant l'instance de la classe Parallelizable paramétrée par une instance de BH_PROP. Le mécanisme va ensuite chercher l'instance *heatEqCompat_bh* et l'utilise avec les paramètres appropriés.

Nous n'utilisons pas nos tactiques de calcul de programme pour cette dernière transformation. Beaucoup de paramètres sont ici instanciés automatiquement lors de la réécriture grâce à l'unification avec le but courant, or notre syntaxe impose ici de fournir explicitement tous ces paramètres du fait du découplage qu'elle impose entre le but proposé par Coq et le terme fournit par l'utilisateur. L'écriture de la preuve s'en trouve extrêmement alourdie.

La dernière étape consiste à récupérer l'implantation parallèle construite :

```
Definition HeatEqPar (leftBound rightBound dt dx κ : number) :=
  Eval simpl in
    parallel (f := (fun u ⇒ heatEquationSeq leftBound rightBound dt dx κ u)).
```

Nous utilisons une réduction du terme par la tactique `simpl` afin d'éliminer par réduction tous les appels de fonctions utilisant le mécanisme de classe de type. Cela permet d'obtenir, après extraction en OCaml, un terme dont le type peut être inféré dans le système de type d'OCaml qui n'a pas de support pour les classes de types.

8.2.2 Mesures de performance

Les temps de calcul ont été mesurés de la même façon que pour la version n'utilisant pas BH (voir section 5.2.4).

La figure 5.4 montre les temps de calcul du programme extrait implanté à l'aide de BH, du programme extrait implanté sans utiliser BH, des versions défonctorisées de ces deux programmes et de l'implantation directe en BSML non certifiée. La version implantée à l'aide de BH est environ trois fois plus lente que la version implantée sans BH. Ce problème vient de l'utilisation de la fonction *bh_seq_opt*. En effet, bien qu'elle ait été optimisée par rapport à la version pré-existante, le calcul fait ici est un cas pathologique où décomposer le programme pour l'adapter à BH augmente la complexité du calcul. Dans ce cas, il faudrait tout simplement utiliser *heatEquationSeqCompat* pour effectuer les calculs locaux, la correction du programme serait bien sûr préservée, et des tests préliminaires laissent espérer des performances très proches de la version extraite implantée sans BH.

Nous prévoyons de développer une implantation certifiée du squelette BH qui utilisera, pour le calcul local, non pas la fonction *bh_seq_opt* mais directement la fonction séquentielle que nous cherchons à paralléliser. Si, comme nous le pensons, les performances sont similaires aux performances obtenue par l'implantation non certifiée, nous aurons obtenu une parallélisation efficace de la fonction de calcul de diffusion de la chaleur prouvée correcte sans avoir, dans la chaîne de preuve, à raisonner sur le parallélisme. Le développement et la preuve de ce programme parallèle s'en trouvent donc grandement simplifiés.

8.3 Construction de tours

Le problème de construction de tours est une extension du problème des lignes de vues pour laquelle Gesbert propose une dérivation à l'aide du squelette *mapAround*. Nous reprenons ici la définition du problème, montrons comment il a été implanté dans notre environnement, puis présentons des mesures de performances.

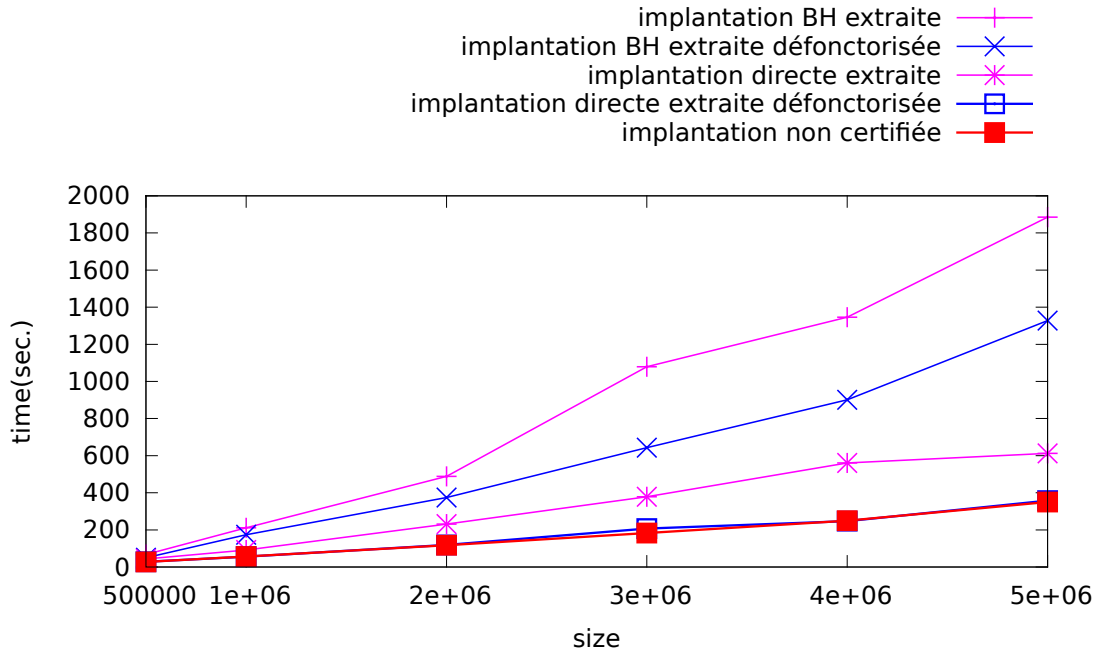
8.3.1 Définition

Considérons une liste d'emplacements le long d'une ligne, en montagne. Les emplacements sont désignés par leur position et leur hauteur sur la ligne : $[(x_1, h_1), \ldots, (x_i, h_i), \ldots, (x_n, h_n)]$ [4]. Prenons deux points particuliers (x_G, h_G) et (x_D, h_D) sur cette même ligne, respectivement à gauche et à droite de ceux désignés dans la liste. Le problème consiste à trouver toutes les positions où il est possible de construire une tour de hauteur h du haut de laquelle il sera possible de voir les deux points. Si nous ne nous préoccupons pas de parallélisme, ni d'efficacité, le problème peut aisément être résolu en vérifiant que, pour chaque emplacement (x_i, h_i), la tour peut être vue depuis (x_G, h_G) et (x_D, h_D).

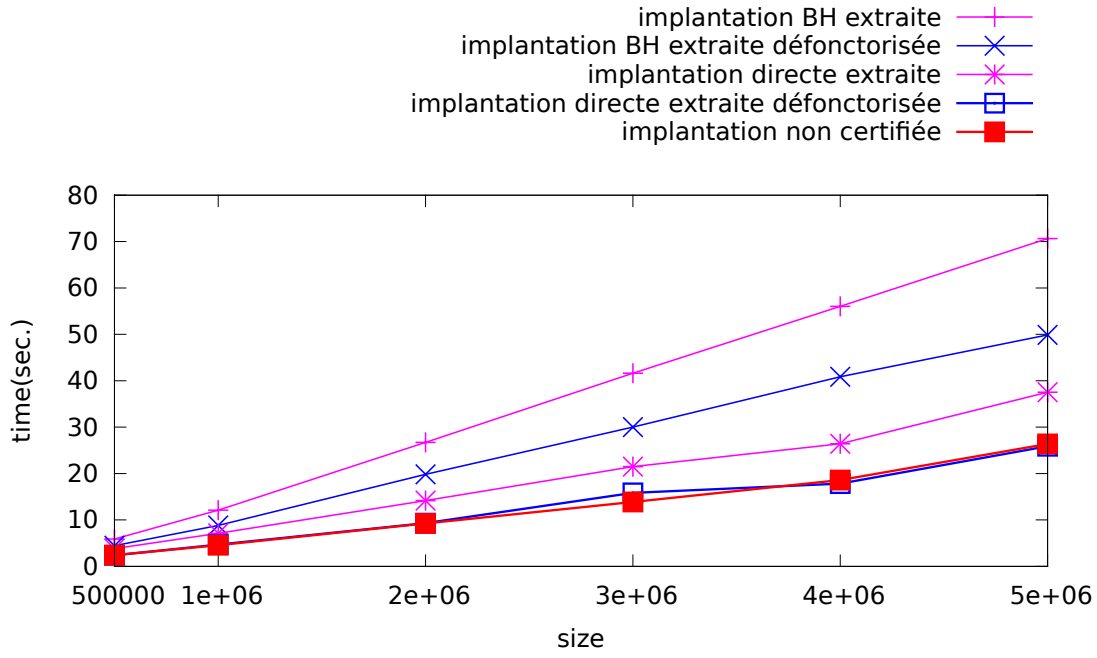
La tour peut être vue si elle n'est cachée par aucune des autres positions, ce qui s'exprime par le fait que pour tout $k = 1, 2, \ldots, i-1$ l'inéquation suivante est vérifiée

$$\frac{h_k - h_G}{x_k - x_G} < \frac{h + h_i - h_G}{x_i - x_G}$$

4. $\forall i, x_i < x_{i+1}$



(a) sur 1 processeur



(b) sur 16 processeurs (8 nœuds)

FIGURE 8.1 – *Mesure du temps de calcul et de collecte mémoire pour 200 itérations sur la machine MIREV*

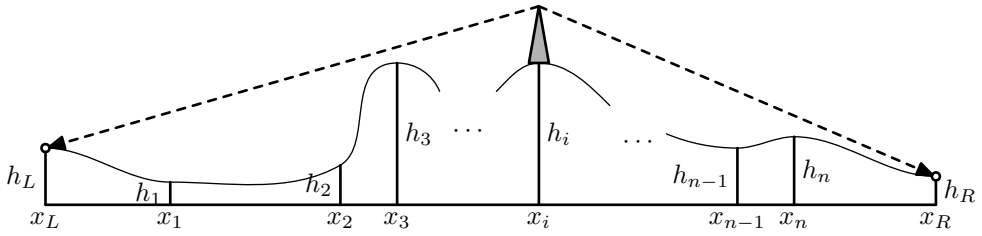


FIGURE 8.2 – *Problème de construction de tours*

et de même pour tout $\mathtt{k} = \mathtt{i}+1, \ldots, \mathtt{n}$ on doit avoir

$$\frac{\mathtt{h_k} - \mathtt{h_D}}{\mathtt{x_D} - \mathtt{x_k}} < \frac{\mathtt{h} + \mathtt{h_i} - \mathtt{h_D}}{\mathtt{x_D} - \mathtt{x_i}}.$$

Spécification à l'aide de *mapAround*

Nous pouvons résoudre directement le problème en utilisant `mapAround`. À chaque position, il suffit d'utiliser les informations des deux cotés pour décider si une tour peut être construite à cet endroit. Ainsi notre spécification peut être définie directement comme suit :

$$\begin{array}{l} \mathit{tower}\ (\mathtt{x_L}, \mathtt{h_L})\ (\mathtt{x_R}, \mathtt{h_R})\ \mathit{xs} = \mathit{mapAround}\ \mathit{visibleLR}\ \mathit{xs} \\ \quad \text{où } \mathit{visibleLR}\ (\mathit{ls}, (\mathtt{x_i}, \mathtt{h_i}), \mathit{rs}) = \\ \qquad\qquad \mathit{visibleL}\ \mathit{ls}\ \mathtt{x_i} \wedge \mathit{visibleR}\ \mathit{rs}\ \mathtt{x_i} \\ \qquad \mathit{visibleL}\ \mathit{ls}\ \mathtt{x_i} = \mathit{maxAngleL}\ \mathit{ls} < \frac{\mathtt{h}+\mathtt{h_i}-\mathtt{h_L}}{\mathtt{x}-\mathtt{x_L}} \\ \qquad \mathit{visibleR}\ \mathit{rs}\ \mathtt{x_i} = \mathit{maxAngleR}\ \mathit{rs} < \frac{\mathtt{h}+\mathtt{h_i}-\mathtt{h_R}}{\mathtt{x_R}-\mathtt{x}} \end{array}$$

La fonction *maxAngleL* calcule si la position $(\mathtt{x_L}, \mathtt{h_L})$ est visible. elle est définie ainsi :

$$\begin{array}{lcl} \mathit{maxAngleL}\ [\,] & = & -\infty \\ \mathit{maxAngleL}\ ([(\mathtt{x}, \mathtt{h})] +\!\!+ \mathtt{xs}) & = & \frac{\mathtt{h}-\mathtt{h_L}}{\mathtt{x}-\mathtt{x_L}} \uparrow \mathit{maxAngleL}\ \mathtt{xs} \end{array}$$

$\mathtt{a} \uparrow \mathtt{b}$ retourne le maximum de $\mathtt{a}$ et $\mathtt{b}$. La fonction *maxAngleR* calcule la visibilité de la position $(\mathtt{x_R}, \mathtt{h_R})$:

$$\begin{array}{lcl} \mathit{maxAngleR}\ [\,] & = & -\infty \\ \mathit{maxAngleR}\ ([(\mathtt{x}, \mathtt{h})] +\!\!+ \mathtt{xs}) & = & \frac{\mathtt{h}-\mathtt{h_R}}{\mathtt{x_R}-\mathtt{x}} \uparrow \mathit{maxAngleL}\ \mathtt{xs}. \end{array}$$

Les fonctions *maxAngleL* et *maxAngleR* sont des homomorphismes, *visibleL* et *visibleL* sont donc des quasi-homomorphismes. La fonction *tower* peut donc être parallélisée par BH.

Implantation Coq

Nous avons montré comment le problème de construction de tours pouvait être implanté à l'aide du squelette *mapAround*. Nous allons maintenant donner plus de détails sur la façon dont cette implantation est faite en Coq.

Le foncteur contenant le code prend en paramètre un module contenant le type utilisé pour définir les positions (qui sera instancié par *float* à l'exécution) ; une relation d'ordre total sur ce type ; les fonctions *angleL* et *angleR*, qui effectuent les calculs d'angles et une fonction *preprocess* qui permet d'ajouter la hauteur de la tour à la position testée.

Pour pouvoir définir les fonctions *maxAngleL* et *maxAngleR*, il faut disposer d'un élément neutre pour la comparaison utilisée. Pour cela nous encapsulons les valeurs dans un type inductif pour l'étendre avec un élément min et un élément max ; la relation d'ordre est étendue à ce type.

Les fonctions *maxAngleL* et *maxAngleR* retournent un élément de ce type étendu. Pour chacune de ces deux fonctions on montre qu'il s'agit d'un homomorphisme dont l'opérateur associé est la fonction ($\uparrow$).

La fonction *tower* est définie comme présenté ci-dessus. Une instance de la classe MapAround_BHable est ensuite définie pour cette fonction puis la fonction de calcul du problème est parallélisée en utilisant la fonction *parallel* appliquée à la fonction *tower*.

Complexité de l'algorithme de construction de tours

Dans ce programme, les opérations $\oplus_{\mathtt{l}}$ et $\otimes_{\mathtt{r}}$ du squelette sont instanciées par la fonction $\uparrow$ de complexité constante (si la comparaison d'éléments se fait en temps constant). *maxAngleL* et *maxAngleR* ont la même complexité :

$$\overline{\mathit{maxAngleL}\ \mathtt{lst}} = \overline{\mathit{maxAngleR}\ \mathtt{lst}} \in \mathcal{O}(\mathtt{length\ lst}).$$

En reprenant la formule de coût pour le squelette BH donnée à la section 7.1.7, on obtient la complexité suivante pour l'algorithme de construction de tours extrait :

$$\max_{0 \leq \mathtt{i} < \mathtt{p}} \mathcal{O}(\mathtt{length\ lst_i}) + \mathtt{g} * \mathcal{O}(\mathtt{p}) + \mathtt{L}$$

et si la liste lst est équitablement distribuée, la complexité est :

$$\mathcal{O}(\mathtt{length\ lst/p}) + \mathtt{g} * \mathcal{O}(\mathtt{p}) + \mathtt{L}.$$

8.3.2 Mesures de performance

Nous avons fait des mesures de temps d'exécution de l'implantation extraite de la dérivation en Coq et les avons comparées à une implantation écrite directement en BSML. L'expérimentation a été menée sur le Centre de Calcul Scientifique de la Région Centre (CCSC). Nous n'avons pu réserver que 18 noeuds de la grappe pour nos expériences.

La figure 8.3 montre le temps de calcul sur un processeur pour différentes tailles de données. La figure 8.4 montre le temps de calcul sur 32 processeurs pour ces mêmes tailles de données.

Nous pouvons voir que le temps d'exécution (et de collecte mémoire) augmente linéairement avec la quantité de données. La version extraite est plus lente que la version directe avec un facteur de 1 pour 2,5.

Cette différence de performance peut s'expliquer en grande partie par l'encapsulation du type des positions pour avoir une borne min et max dans l'implantation extraite. Dans l'implantation écrite en BSML, le type *float* est utilisé directement et la valeur `min_float` est utilisée comme borne inférieure, ce qui économise un certain nombre d'appels de fonctions.

Comme montré à la figure 8.5, l'accélération des deux implantations par rapport au nombre de processeurs, pour un nombre fixé de données (5.120.000 éléments), est super-linéaire. Cette accélération est due à l'augmentation de la mémoire disponible qui permet un déclenchement moins fréquent du ramasse-miette.

8.4 Somme de préfixe maximum

Le calcul de la somme maximale de préfixe est une restriction du problème de segment de somme maximale, où l'on ne considère que les segments commençant au début de la liste, c'est à dire les préfixes.

8.4.1 Définition

Informellement, cela donne

$$\mathrm{mps}[1,-1,2,-2] = 1 \uparrow (1-1) \uparrow (1-1+2) \uparrow (1-1+2-2).$$

L'ensemble du développement est paramétré par un module contenant un type muni d'un ordre total, d'une borne inférieure pour cet ordre, et d'un opérateur $\odot$ utilisé pour faire la somme des éléments, ainsi qu'un élément neutre pour cet opérateur,

Nous définissons la fonction *mps* en Coq comme suit :

```
Definition mps (l :list A) := (fun l ⇒ fold_left Max l min) :o: (scan A ⊙ ι⊙) l.
```

La fonction *scan* calcule la somme des préfixes, puis la fonction *fold_left* est utilisée pour obtenir la valeur maximale parmi les sommes de préfixes calculées à l'aide d'une fonction *Max* définie à partir de l'ordre total fourni.

Nous avons vu au chapitre précédent la parallélisation de la fonction *scan* ainsi que des homomorphismes de listes. La fonction *fold_left* est un homomorphisme si elle est utilisée avec un opérateur associatif et un élément neutre pour cette opérateur en valeur initiale. C'est exactement le cas ici, la fonction *mps* est donc une composition de fonctions parallélisables, sa définition parallèle se fait comme suit :

```
Definition mps_bsml := Eval compute
[parallel_type list_partitionnable parallel
  parallelComposition BHS_Parallelizable BHS_parallelisation
  parallel_parallelizable
  BHS_k BHS_gl BHS_opl BHS_gr BHS_opr
  mapAround_to_BH
```

```
    homoPar
    homomorphismParallelKernel]
  in parallel (f :=mps).
```

La réduction à l'aide de la commande `Eval compute`, en restreignant la δ-réduction aux noms des instances et méthodes impliquées dans la parallélisation permet d'éliminer toutes les références aux classes de types.

8.4.2 Mesures de performance

Les performances du programme extrait sont comparées à une implantation utilisant la fonction *scan* de la bibliothèque BSML. Les deux programmes sont appliqués sur des listes de *float* et utilisent l'addition comme opérateur. Les figure 8.6 et 8.7 montrent les temps d'exécution de l'implantation extraite et de l'implantation non certifiée pour des listes de taille croissante sur 1 et 128 processeurs sur le Centre de Calcul Scientifique de la région Centre (CCSC) . Le temps de calcul est linéaire par rapport à la taille de la liste dans les deux cas. L'implantation extraite est 5 fois plus lente que l'implantation non certifiée. Cet écart de performances important est très certainement dû à l'implantation des homomorphismes de listes par BH. Malgré sa forte expressivité, BH n'est pas l'outil adapté pour ce calcul et il serait judicieux de développer un squelette spécifique pour paralléliser les homomorphismes de façon plus efficace.

L'accélération des deux programmes pour un nombre de processeurs croissants est donnée à la figure 8.8. La version extraite et la version non certifiée ont une accélération similaire.

8.5 Conclusion

Nous avons présenté dans ce chapitre trois exemples. Pour chaque exemple, une implantation séquentielle sert de spécification, puis une implantation parallèle montrée correcte est construite à l'aide du squelette BH et des mécanismes présentés au chapitre 6. Les foncteurs contenant les programmes parallèles ainsi construits sont extraits en OCaml, puis appliqués au module `BsmlNat`. Les programmes obtenus sont alors directement exécutables en parallèle, sans intervention sur le code obtenu. Ainsi, aucune erreur ne peut être introduite dans le code montré correct si l'on considère que l'extraction est un mécanisme sûr.

Les mesures effectuées en appliquant les programmes extraits à des données de tailles croissantes montrent des temps d'exécution de 2,5 à 5 fois supérieurs aux temps d'exécutions obtenus par les implantations non-certifiées. Bien que l'assurance obtenue par la preuve de correction vaut bien un léger sur-coût à l'exécution, ceux obtenus ici sont excessifs. Nous avons cependant vu qu'ils peuvent être largement diminués de diverses manières : la défonctorisation du code, le remplacement de *bh_seq_opt* par une implantation optimisée pour un problème spécifique, le développement de squelettes plus adaptés pour certains problèmes.

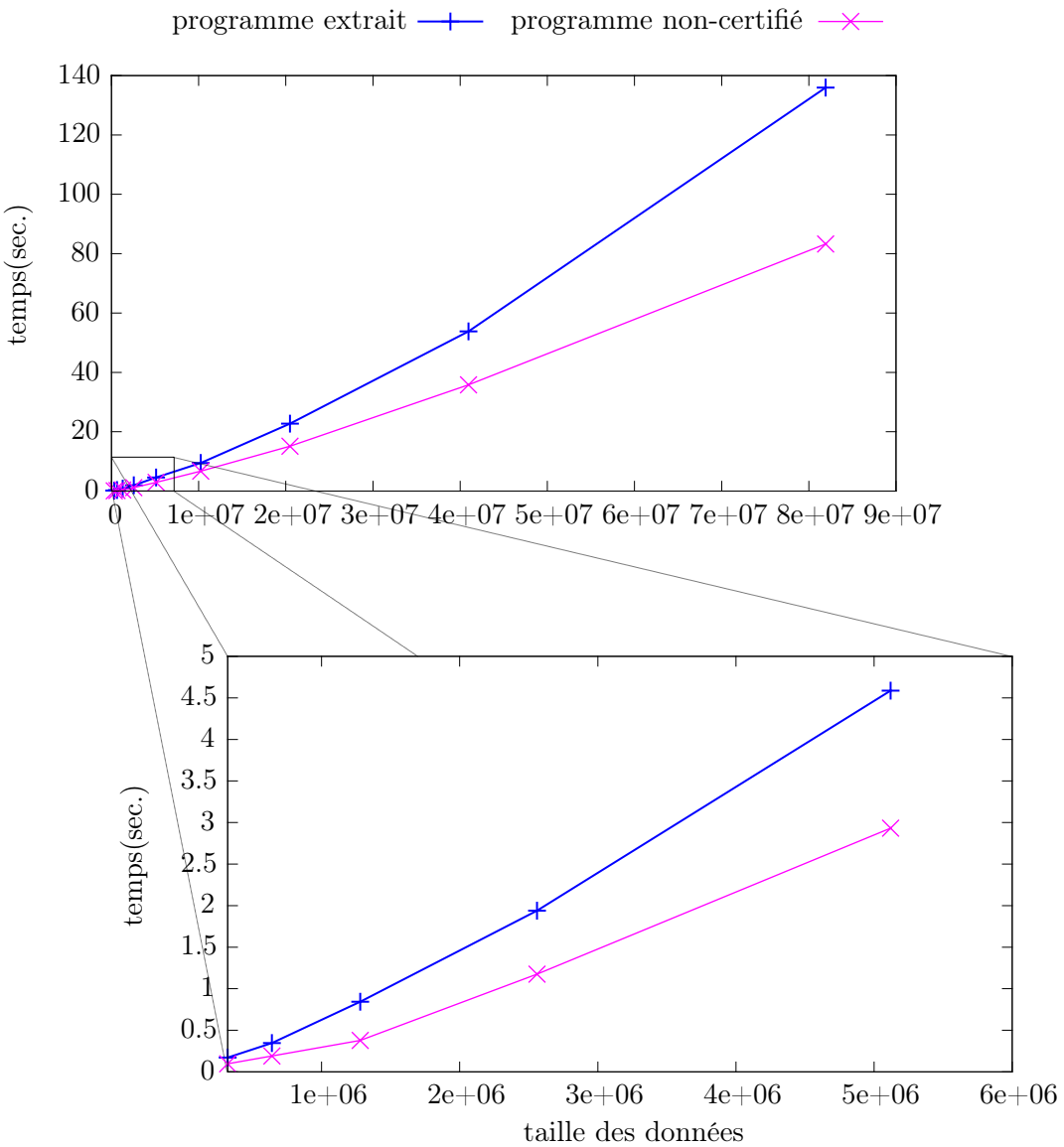


FIGURE 8.3 – *Temps d'exécution du programme de construction de tours sur 1 processeur pour une liste de taille croissante sur le CCSC.*

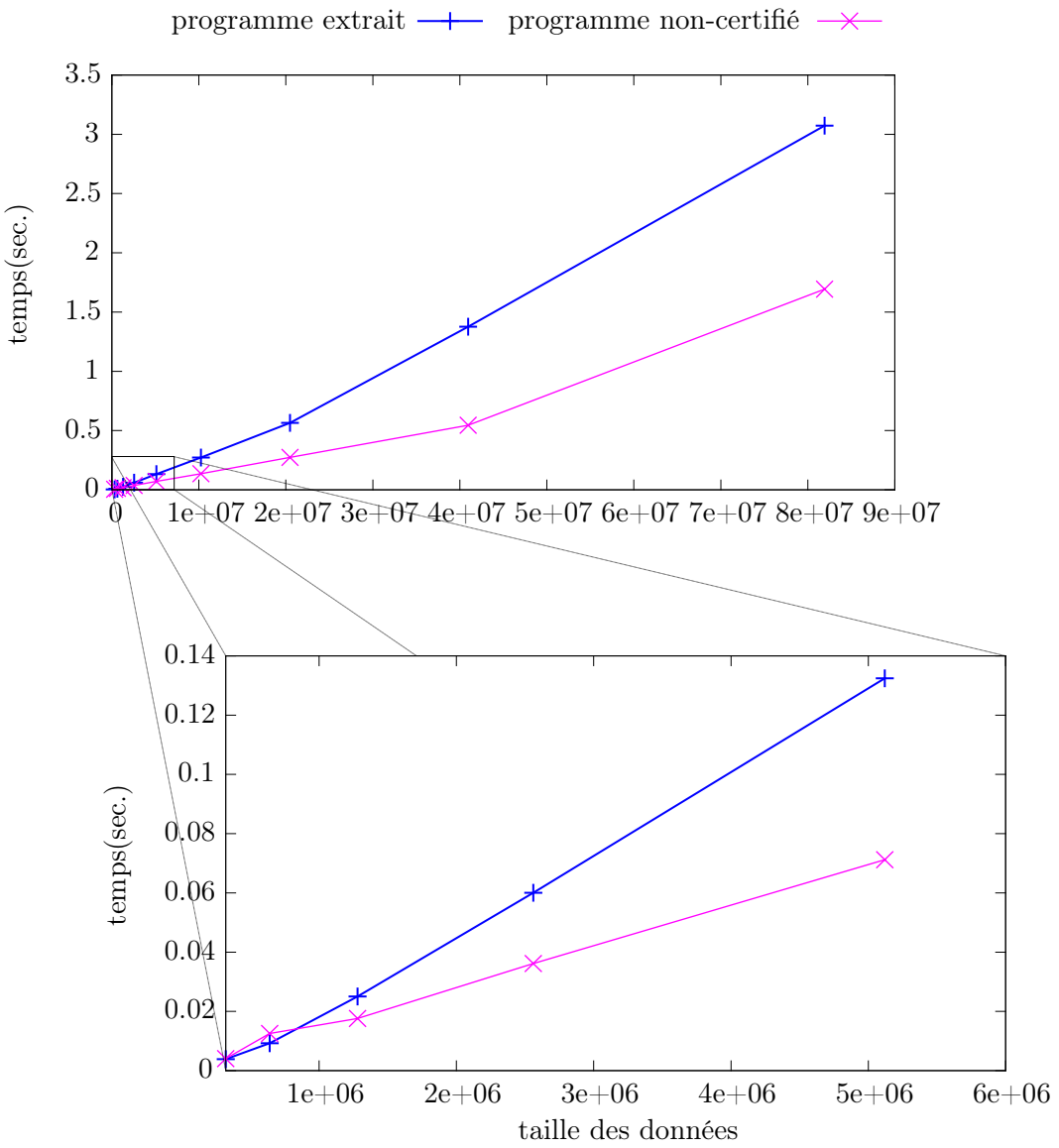


FIGURE 8.4 – *Temps d'exécution du programme de construction de tours sur 32 processeurs pour une liste de taille croissante sur le CCSC.*

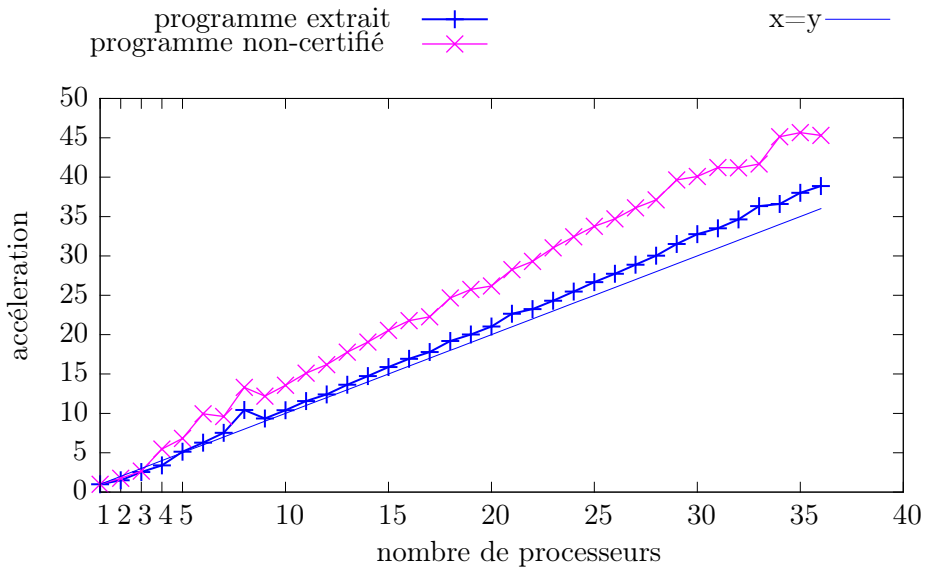


FIGURE 8.5 – *Accélération du programme de construction de tours pour un nombre croissant de processeurs (pour une liste de 5120000 éléments, sur le CCSC).*

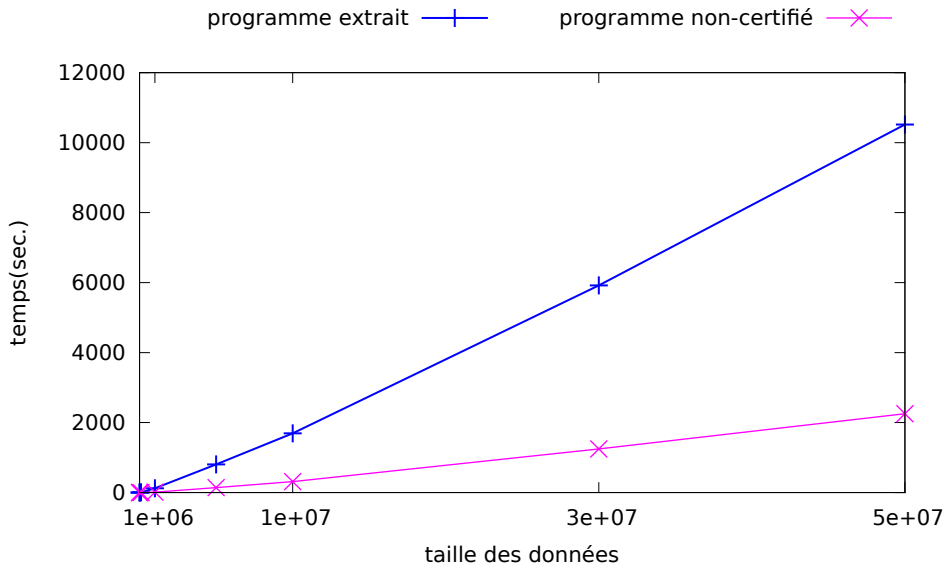


FIGURE 8.6 – *Temps de calcul de la somme maximale de préfixe sur 1 processeur (CCSC).*

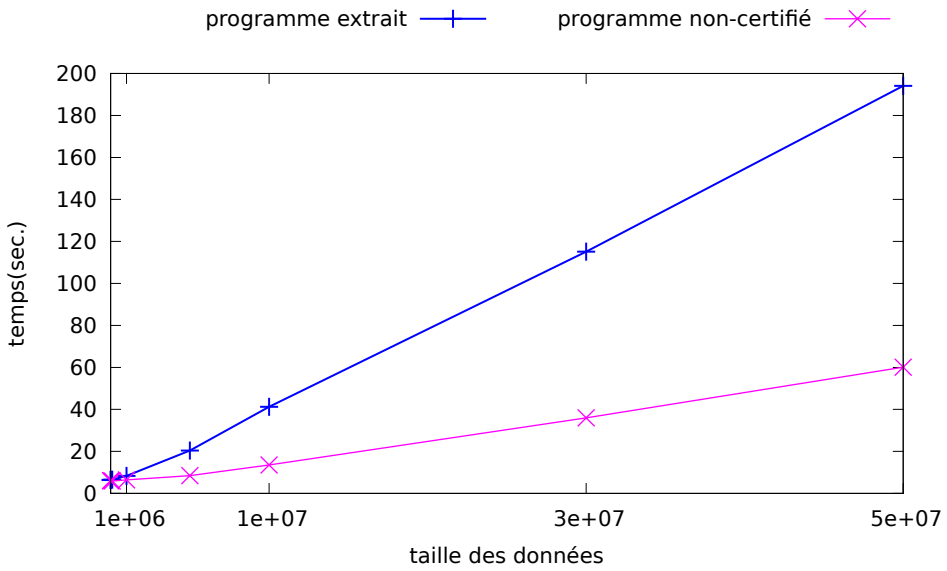


FIGURE 8.7 – *Temps de calcul de la somme maximale de préfixe sur 128 processeurs (CCSC).*

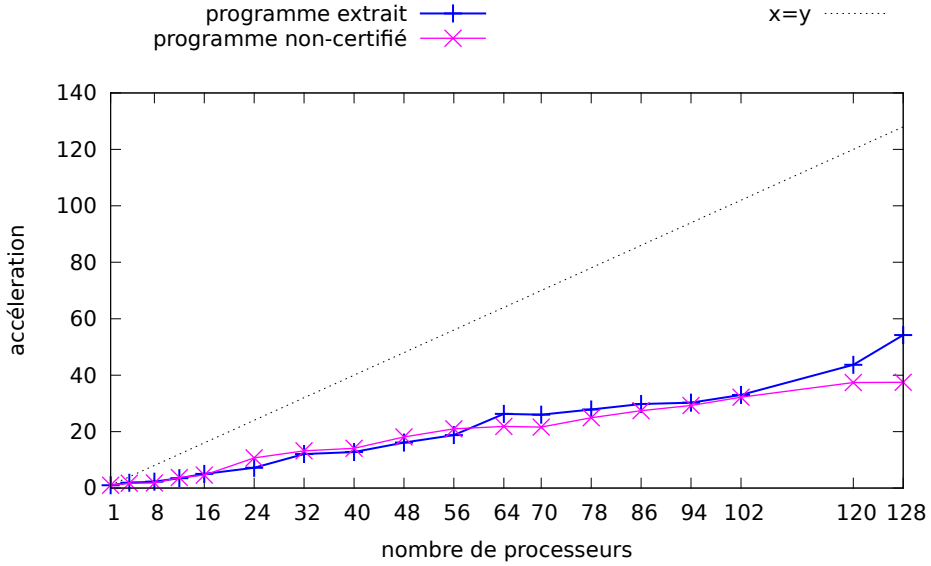


FIGURE 8.8 – *Accélération du calcul de la somme maximale de préfixe pour une liste de* 5×10^7 *(CCSC).*

Conclusion et perspectives

9

Sommaire

9.1 Bilan

Cette thèse propose un environnement pour le développement et la preuve de correction de programmes parallèles BSML dans l'assistant de preuve Coq. Nous proposons pour cela un plongement superficiel du langage BSML dans le langage de l'assistant de preuve Coq. La structure de données parallèle de BSML et la sémantique de ses primitives parallèles sont axiomatisées sous forme d'un type de module. Il est ainsi possible d'écrire et de raisonner sur des programmes BSML en Coq dans des foncteurs prenant en paramètre un module respectant la spécification des vecteurs parallèles et des primitives associées. Une réalisation séquentielle des axiomes nous assure de la cohérence de ceux-ci avec la sémantique de Coq. Nous avons également pu tester sur quelques exemples simples la conformité de cette réalisation des spécifications des primitives avec leurs implantations séquentielle et parallèle en OCaml. **Cette axiomatisation nous permet donc de définir des programmes BSML dans l'assistant de preuve Coq et de raisonner sur leurs résultats, puis de les extraire vers OCaml. Ils peuvent ensuite être compilés à l'aide du compilateur BSML et exécutés sur une machine parallèle.** Développer directement des programmes parallèles, énoncer des propriétés sur ces programmes et les montrer corrects n'est cependant pas une tâche aisée. Nous proposons donc que le développement de programme soit fait en deux étapes :

- les programmes sont initialement implantés sous une forme séquentielle naïve, simple à comprendre, que nous prenons comme spécification,
- puis une implantation parallèle efficace est construite et montrée équivalente.

Pour guider cette démarche, **nous définissons formellement la *parallélisation correcte* d'un programme séquentiel par un programme parallèle**, et pour cela

nous formalisons également la relation entre structures de données séquentielles et structures de données parallèles. La relation de parallélisation correcte définie a la bonne propriété d'être préservée par la composition de fonction.

Notre spécification est un programme calculant une *vue séquentielle* du résultat attendu du programme parallèle, des propriétés peuvent être prouvées sur cette *vue* du résultat sans s'encombrer des détails liés au parallélisme.

Cependant, montrer qu'un programme est la parallélisation correcte d'une spécification reste une tâche parfois ardue. Il est donc nécessaire de pouvoir capitaliser les efforts fournis. C'est précisément ce que nous permet la programmation par squelette en définissant des fonctions de haut niveau, implantées en parallèle et réutilisables dans de nombreuses circonstances.

Nous proposons un mécanisme permettant de faciliter, voire d'automatiser la parallélisation de fonctions écrites à l'aide de squelettes dont il existe une parallélisation connue.

Un atout important de notre approche réside dans la possibilité de conditionner la parallélisation d'une fonction par un squelette aux propriétés des opérations utilisées dans la définition de cette fonction.

Pour donner une représentation claire du résultat attendu d'un programme, une spécification n'est pas toujours représentée directement sous une forme efficace et directement parallélisable par un squelette algorithmique. Afin de parvenir à une telle forme, **nous proposons un ensemble de tactiques permettant de faciliter la transformation d'une spécification en un programme efficace et parallélisable**. Enfin, un squelette algorithmique efficace et expressif proposé par Hu et Gesbert, le squelette BH, vient compléter notre environnement. Notre contribution sur les homomorphismes BSP a consisté à clarifier la définition de cette classe de fonctions de façon à assurer la possibilité d'une parallélisation systématique de celle-ci. Nous avons également fourni une implantation parallèle corrigée et plus efficace du squelette algorithmique correspondant et montré qu'il s'agissait d'une parallélisation correcte des fonctions de cette classe sous certaines conditions. Par ailleurs, ce squelette a été utilisé pour paralléliser des squelettes algorithmiques moins généraux, dont la sémantique est plus proche de problèmes algorithmiques.

Finalement, **des exemples concrets montrent que notre environnement permet de simplifier grandement le développement de programmes parallèles corrects.** Les applications utilisant des squelettes algorithmiques ne nécessitent, par exemple, aucune preuve sur les communications. Par ailleurs des expériences menées sur plusieurs machines parallèles et massivement parallèles montrent que les programmes BSML extraits à partir de ces développements Coq fournissent des résultats encourageants en terme de performances.

9.2 Perspectives

9.2.1 Étendre la classe de programmes traités dans l'environnement

Nous avons déjà mentionné à la section 8.5 les améliorations possibles des squelettes disponibles. Considérer de nouveaux squelettes algorithmiques, voire d'autres structures de données que les listes, telles que les matrices, les arbres [109] ou les graphes permettrait d'étendre le champ d'applicabilité de notre environnement.

Dans les applications présentées, nous n'avons pas prouvé ni utilisé de propriétés sur les représentations des nombres utilisées et les opérations associées. Par exemple dans le cas de la diffusion de chaleur, la spécification est une formulation de l'équation discrétisée de la chaleur qui manipule des nombres dont la représentation ne fait l'objet d'aucune hypothèse. Si la spécification est une équation continue de la chaleur, les solutions sont des fonctions sur $\mathbb{R}$. Il faudrait d'abord pouvoir borner l'erreur de méthode, c'est-à-dire l'erreur introduite par la méthode de discrétisation, puis il faudrait pouvoir borner les erreurs d'arrondis introduites par le passage de valeurs sur $\mathbb{R}$ à des nombres à virgule flottante. Les travaux menés dans le cadre du projet Fost [1] ont considéré une équation d'ondes et borné les deux types d'erreur [20, 19]. Les développements Coq, qui ont en particulier utilisé l'outil Gappa [21], pourraient être utilisés pour fournir de telles garanties dans le cas de l'équation de la chaleur. Les techniques pour borner les erreurs d'arrondis en vue de prouver la correction d'un programme itératif écrit en C pour résoudre l'équation discrétisée poseraient deux difficultés en général pour la preuve de programmes BSML. La première est que ces travaux ont été réalisés dans le cadre de la vérification d'obligations de preuve générées par l'outil Why/Caduceus et non dans un cadre d'extraction de code. La seconde est que les opérations sur les nombres flottants ne sont pas associatives et qu'établir des bornes raisonnables lorsque l'associativité est utilisée semble difficile. Or dans le cas d'une réduction en parallèle par exemple, l'associativité est utilisée massivement. Notons toutefois que le cas particulier de l'équation de la chaleur ne pose pas ce problème : les expressions arithmétiques sont évaluées de la même façon en séquentiel et en parallèle. Dans le cas contraire, pour prouver l'équivalence des versions séquentielles et parallèles nous aurions du faire des hypothèses sur le type des nombres que nous manipulons.

Enfin le langage BSML complet dispose, comme OCaml, de traits impératifs. Il serait intéressant de voir dans quelle mesure ils pourraient être intégrés à notre modélisation. Dans ce contexte, la possible extension de Ynot [112] pourrait être étudiée.

9.2.2 Raisonner sur le coût des programmes

Un autre point qu'il serait nécessaire de développer est le raisonnement sur les coûts des programmes. Nous avons vu que le modèle BSP est équipé d'un modèle de coût. Il serait intéressant de pouvoir produire des preuves mécanisées des coûts des programmes développés dans notre environnement.

1. `http://fost.saclay.inria.fr`

Quand il est question de parallélisme et de coût, la question de la répartition des données est centrale. Nous avons effectué quelques travaux préliminaires, dans lesquels nous avons modélisé la répartition des données et nous avons pu prouver que l'implantation parallèle de BH préserve la distribution des données.

Pour pouvoir raisonner sur les coûts, deux grandes approches semblent envisageables :

- utiliser des monades de coût, ce qui implique d'équiper les primitives parallèles avec ces monades mais également les opérations séquentielles les plus déterminantes dans le coût, c'est l'approche suivie par [141] dans un cadre séquentiel et purement fonctionnel ;
- raisonner sur la structure même des programmes.

Dans ce dernier cas, nous atteignons là les limites qu'imposent un plongement superficiel, il n'est pas possible en Coq de raisonner directement et de manière générale sur la structure d'un terme. Néanmoins, la structure de Coq est conçue pour pouvoir ajouter simplement des extensions dans lesquelles la structure des termes est accessible. Grâce à ce genre d'extension, on pourrait construire, à partir d'un terme Coq, une représentation de ce terme dans un type inductif correspondant à un plongement profond. Un tel mécanisme permettrait de conserver la facilité de programmation de notre environnement tout en autorisant des preuves pour de nouvelles classes de propriétés basées sur la structure des programmes, telle que les coûts d'exécution.

Une alternative serait d'utiliser l'approche par formule caractéristique. Dailler [49] a étudié l'ajout de crédit-temps pour la preuve de correction et de complexité de programmes séquentiels impératifs.

9.2.3 Vérifier l'implantation de BSML

Bien entendu, l'assurance que les programmes BSML extraits de développement réalisés dans notre environnement s'exécutent de façon conforme à leur spécification repose sur un certain nombre d'hypothèses incluant : la correction de Coq et en particulier de son mécanisme d'extraction, la correction du compilateur OCaml, et enfin la correction de la bibliothèque BSML (les deux dernières hypothèses pouvant être remplacées par la correction d'un compilateur BSML si celui-ci existait). CompCertML [52, 53] est un compilateur vérifié pour un sous-ensemble d'OCaml. Des travaux sur la vérification en Coq de l'extraction sont en cours [67].

L'ajout des vecteurs parallèles et des primitives BSML à CompCertML permettrait de compiler les applications considérées dans cette thèse. La vérification d'un tel compilateur ou de *l'implantation parallèle* des primitives BSML en tant que bibliothèque nécessiterait d'avoir des modèles d'exécution formels de ces primitives et de prouver leur équivalence avec le modèle de programmation, c'est-à-dire l'axiomatisation des primitives BSML que nous avons proposée. On retrouve la distinction vue macroscopique et vue microscopique en parallélisme de données [24]. Ceci a déjà été fait pour des sémantiques papier de BSML [63] mais n'a jamais été vérifié en Coq et ne concerne qu'un noyau extrêmement réduit de ML.

La vérification de l'implantation d'une bibliothèque ou d'un compilateur BSML nécessite de fixer le niveau auquel on axiomatise. En effet, l'implantation actuelle de la

bibliothèque BSML peut au choix utiliser le module `Unix` d'OCaml pour des communications TCP/IP, ou un interfaçage avec une bibliothèque MPI. Dans le premier cas, il serait nécessaire d'axiomatiser les appels systèmes TCP/IP, dans le second cas il faudrait axiomatiser la sémantique des fonctions utilisées de la bibliothèque MPI. Il pourrait ensuite être souhaitable de prouver la correction de l'implantation de ces fonctions C de MPI (qui, *in fine*, reposent sur des appels systèmes aux fonctionnalités réseau) ce qui serait un travail titanesque.

Développer un compilateur (vérifié) spécifique à BSML serait un travail beaucoup plus important que vérifier l'implantation de la bibliothèque BSML mais offrirait d'autres avantages :

- la possibilité de concevoir des optimisations spécifiques aux programmes BSML qui prennent en compte le niveau global, ce que ne peut faire l'implantation actuelle,
- la possibilité de concevoir d'autres modèles d'exécution, en particulier étudier une implantation de BSML en mémoire partagée.

Annexes

Machines parallèles

A

Sommaire

A.1 Très Grand Centre de Calcul Curie

Nombre de nœuds	360
Processeurs par nœud	4
Cœurs par processeur	8
Mémoire vive	128Go
nombre total de cœurs	11520
type de processeurs	Intel® Nehalem-EX X7560 cadencés à 2.26 GHz
interconnexion	InfiniBand QDR Full Fat
version de MPI	bullxmpi 1.1.8.1

Le très grand centre de calcul Curie est un des premiers supercalculateurs européens du partenariat P.R.A.C.E. (Partnership for Advanced Computing in Europe). Nous l'avons utilisé lors de la première phase de sa conception.

La machine est formée de 360 nœuds S6010 bullx, chaque nœud dispose de 4 processeurs octo-cœurs Intel® Nehalem-EX X7560 cadencés à 2.26 GHz, 128 Go de ram, 1 disque local de 2To. Les noeuds sont reliés entre eux par un réseau InfiniBand QDR Full Fat Tree.

Les nœuds sont sous GNU/linux noyau 2.6.32-71.14.1.el6.Bull.20.x86_64, les programmes sont compilés à l'aide d'une version de MPI adaptée par Bull pour le centre de calcul.

Les unités de calcul sont réservées à l'aide du système de gestion de cluster SLURM. Pour toutes les expériences, les réservations ont été faites pour un usage exclusif des nœuds afin d'éviter les interférences pour les accès mémoire et réseau.

A.2 Centre de Calcul Scientifique de la région Centre

Nombre de nœuds	42
Processeurs par nœud	2
Cœurs par processeur	4
Mémoire vive	8Go
nombre total de cœurs	336
type de processeurs	Xeon E5450 cadencés à 2.26 GHz
interconnexion	InfiniBand
version de MPI	Open MPI 1.4.2

A.3 Cluster Mirev

Nombre de nœuds	8
Processeurs par nœud	2
Cœurs par processeur	4
Mémoire vive	16Go
nombre total de cœurs	64
type de processeurs	AMD Opteron QuadCore 2376 cadencés à 2.3 GHz
interconnexion	controleurs Go Ethernet Cuivre
version de MPI	Open MPI 1.4.2

A.4 Machine speed

Nombre de nœuds	1
Processeurs par nœud	4
Cœurs par processeur	12
Mémoire vive	64Go
nombre total de cœurs	48
type de processeurs	AMD Opteron Processor 6174 cadencés à 2.2 GHz
interconnexion	-
version de MPI	MPICH 2

Développement Coq

B

Sommaire

B.1 Principaux modules pour la modélisation de BSML

Modules ou type de module Coq	Description
Specification.PRIMITIVES	Axiomatisation des primitives BSML
BsmlImplementation	Module de type Specification.PRIMITIVES, implantant les primitives BSML par des opérations sur vecteur séquentiel de type Vector
Properties.TYPE	Propriétés découlant de l'axiomatisation
BsmlBase	Modélisation de la bibliothèque standard `Base` de BSML (combinaison de primitives parallèles usuelles sans communication)
BsmlComm	Modélisation d'une partie de la bibliothèque standard `Comm` de BSML dédiée au schémas usuels de communication

B.2 Nombre de ligne de code Coq

		Spécification	Preuve
Modélisation de BSML	Primitives	30	0
	Propriétés & stdlib	216	464
	Implantation séquentielle	60	35
Parallélisation correcte		91	15
	Répartition listes	622	602
Squelettes	BH	456	884
	Autres(map,filter,last,...)	403	226
Applications	Heat equation	199	363
	Heat equation BH	186	57
	Comptage	35	0
	Construction de tours	105	59
	Maximum prefix sum	110	0
Séquentiel	Bibliothèque Coq LIFO : lists, vector, algebra	1995	2827
Total		4508	5532

Bibliographie

[1] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
Cité page 7.

[2] M. Aldinucci and M. Danelutto. Skeleton-based parallel programming : Functional and parallel semantics in a single shot. *Computer Languages, Systems and Structures*, 33(3-4) :179–192, 2007.
Cité page 23.

[3] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L : a structured high-level parallel language, and its structure support. *Concurrency : Practice and Experiences*, 7(3) :225–255, May 1995.
Cité page 24.

[4] R. C. Backhouse. An exploration of the Bird-Meertens formalism. Technical report, STOP Summer School on Constructive Algorithmics, Abeland, 1988.
Cité page 99.

[5] J. Backus. Can programming be liberated from the Von Neumann style : A functional style and its algebra of programs. *Communications of the ACM*, 22(8) :613–641, August 1978. Turing Award Lecture.
Cité page 6.

[6] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. Programming, Deploying, Composing, for the Grid. In J. Cunha and O. F. Rana, editors, *Grid Computing : Software Environments and Tools*. Springer, 2006.
Cité page 24.

[7] M. Bamha and M. Exbrayat. Pipelining a Skew-Insensitive Parallel Join Algorithm. *Parallel Processing Letters*, 13(3) :317–328, 2003.
Cité page 14.

[8] M. Bamha and G. Hains. Frequency-adaptive join for Shared Nothing machines. *Parallel and Distributed Computing Practices*, 2(3) :333–345, 1999.
Cité page 14.

[9] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system : An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, LNCS 3362, pages 49–69. Springer, 2004.
Cité page 5.

[10] J. Berthold, M. Dieterle, O. Lobachev, and R. Loogen. Parallel FFT with Eden Skeletons. In V. Malyshkin, editor, *Parallel Computing Technologies*, LNCS 5698, pages 73–83. Springer, 2009.
Cité page 24.

[11] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development.* Springer, 2004.
Cité page 29.

[12] R. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4) :487–504, Oct. 1984.
Cité page 6.

[13] R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987.
Cité page 22.

[14] R. Bird. Lectures on Constructive Functional Programming. In M. Broy, editor, *Constructive Methods in Computing Science*, number 55 in {NATO} {ASI}, Marktoberdorf, {BRD}, 1988. Springer.
Cité page 6.

[15] R. Bird and O. de Moor. *Algebra of Programming.* International series in computer science. Prentice-Hall, 1997.
Cité page 7.

[16] R. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004.
Cité pages 13 et 14.

[17] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. In B. Pehrson and I. Simon, editors, *Technology and Foundations : Information Processing '94, Vol. I*, volume 51 of *IFIP Transactions A*, pages 509–514. Elsevier Science Publishers, Amsterdam, 1994.
Cité page 14.

[18] G. E. Blelloch. Prefix Sums and Their Applications. Technical report, School of Computer Science, Carnegie Mellon University, 1990.
Cité page 122.

[19] S. Boldo. Floats and Ropes : A Case Study for Formal Numerical Program Verification. In S. Albers, A. Marchetti-Spaccamela, Y. Matias, S. E. Nikoletseas, and W. Thomas, editors, *36th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 5556, pages 91–102. Springer, 2009.
Cité page 145.

[20] S. Boldo, F. Clément, J.-C. Filliâtre, M. Mayero, G. Melquiond, and P. Weis. Formal Proof of a Wave Equation Resolution Scheme : The Method Error. In M. Kaufmann and L. C. Paulson, editors, *First International Conference on Interactive Theorem Proving (IFP)*, LNCS 6172, pages 147–162. Springer, 2010.
Cité page 145.

[21] S. Boldo, J.-C. Filliâtre, and G. Melquiond. Combining Coq and Gappa for Certifying Floating-Point Programs. In J. Carette, L. Dixon, C. Sacerdoti Coen, and S. M. Watt, editors, *16th Symposium on Intelligent Computer Mathematics, (Calculemus)*, LNCS 5625, pages 59–74. Springer, 2009.
Cité page 145.

[22] O. Bonorden, B. Judoiink, I. von Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2) :187–207, 2003.
Cité page 20.

[23] E. Börger and R. Stärk. *Abstract State Machines*. Springer, 2003.
Cité page 24.

[24] L. Bougé. Le modèle de programmation à parallélisme de données : une perspective sémantique. *RAIRO Technique et Science Informatiques*, 12(5), 1993.
Cité pages 17 et 146.

[25] L. Bougé. The Data-Parallel Programming Model : A Semantic Perspective. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, LNCS 1132, pages 4–26. Springer, 1996.
Cité page 17.

[26] L. Bougé, D. Cachera, Y. L. Guyadec, G. Utard, and B. Virot. Formal Validation of Data–Parallel Programs : a Two–Component Assertional Proof System for a Simple Language. *Theoretical Computer Science*, 189(1-2) :71–107, 1997.
Cité pages 18 et 24.

[27] L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors. *Euro-Par'96 Parallel Processing*, number 1123–1124 in LNCS, Lyon, August 1996. LIP-ENSL, Springer.
Cité pages 161 et 164.

[28] A. Braud and C. Vrain. A parallel genetic algorithm based on the BSP model. In *Evolutionary Computation and Parallel Processing GECCO & AAAI Workshop*, Orlando (Florida), USA, 1999.
Cité page 14.

[29] D. Cachera and D. Pichardie. A certified denotational abstract interpreter. In M. Kaufmann and L. C. Paulson, editors, *First International Conference on Interactive Theorem Proving (ITP 2010)*, volume 6172 of *LNCS*, pages 9–24. Springer, 2010.
Cité page 6.

[30] D. Cachera and D. Pichardie. Programmation d'un interpréteur abstrait certifié en logique constructive. *Technique et Science Informatiques*, 30(4) :381–408, 2011.
Cité page 6.

[31] D. Caromel and L. Henrio. *A Theory of Distributed Objects*. Springer, 2004.
Cité page 24.

[32] D. Caromel, L. Henrio, and M. Leyton. Type safe algorithmic skeletons. In *16th Euromicro International Conference on Parallel, Distributed and Network-Based*

Processing (PDP 2008), pages 45–53. IEEE Computer Society, 2008.
Cité page 23.

[33] D. Caromel, L. Henrio, and B. Serpette. Asynchronous and Deterministic Objects. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM Symposium on Principles of Programming Languages*, pages 123–134. ACM Press, 2004.
Cité page 24.

[34] A. Cavarra, E. Riccobene, and A. Zavanella. A formal model for the parallel semantics of P3L. In *SAC'00 : Proceedings of the 2000 ACM symposium on Applied computing*, pages 804–812. ACM, 2000.
Cité page 24.

[35] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000.
Cité page 27.

[36] A. Chargéraud. Program verification through characteristic formulae. In P. Hudak and S. Weirich, editors, *15th ACM SIGPLAN international conference on Functional programming (ICFP 2010)*, pages 321–332. ACM, 2010.
Cité page 6.

[37] Y. Chen and J. W. Sanders. Logic of global synchrony. *ACM Transaction on Programming Languages and Systems*, 26(2) :221–262, 2004.
Cité pages 10 et 21.

[38] Y. Chen and W. Sanders. Top-Down Design of Bulk-Synchronous Parallel Programs. *Parallel Processing Letters*, 13(3) :389–400, 2003.
Cité page 21.

[39] A. Chlipala. Certified Programming with Dependent Types. http ://adam.chlipala.net/cpdt, 2010.
Cité page 29.

[40] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2001.
Cité page 3.

[41] M. Cole. *Algorithmic Skeletons : Structured Management of Parallel Computation*. MIT Press, 1989. Available at `http://homepages.inf.ed.ac.uk/mic/Pubs`.
Cité page 14.

[42] M. Cole. Parallel Programming with List Homomorphisms. *Parallel Processing Letters*, 5(2) :191–203, 1995.
Cité page 23.

[43] R. L. Constable. The trimuph of types : Creating a logic of computational reality. In *Principia Mathematica anniversary symposium*, Nov. 2011.
Cité page 28.

[44] P. Corbineau. A Declarative Language for the Coq Proof Assistant. In M. Miculan, I. Scagnetto, and F. Honsell, editors, *Types for Proofs and Programs, International Conference, TYPES 2007, Cividale des Friuli, Italy, May 2-5, 2007, Revised Selected Papers*, volume 4941 of *LNCS 4941*, pages 69–84. Springer, 2008.
Cité page 103.

[45] R. D. Cosmo, Z. Li, S. Pelagatti, and P. Weis. Skeletal Parallel Programming with OcamlP3l 2.0. *Parallel Processing Letters*, 18(1) :149–164, 2008.
Cité page 24.

[46] G. Cousineau and M. Mauny. *Approche Fonctionnelle de la Programmation*. Ediscience International, 1995.
Cité page 27.

[47] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'77)*, pages 238–252. ACM, 1977.
Cité page 4.

[48] F. Dabrowski and D. Pichardie. A Certified Data Race Analysis for a Java-like Language. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *LNCS 5674*, pages 212–227. Springer, 2009.
Cité pages 6 et 11.

[49] S. Dailler. Preuve mécanisée de correction et de complexité pour un algorithme impératif. Master's thesis, University Paris Diderot, 2011.
Cité page 146.

[50] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for data parallelism in p3l. In *Proceedings of the Third International Euro-Par Conference on Parallel Processing*, Euro-Par '97, pages 619–628. Springer-Verlag, 1997.
Cité page 24.

[51] M. Danelutto and P. Teti. Lithium : A Structured Parallel Programming Environment in Java. In P. M. A. Sloot, C. J. K. Tan, J. Dongarra, and A. G. Hoekstra, editors, *Computational Science - ICCS 2002, International Conference, Amsterdam, The Netherlands, April 21-24, 2002. Proceedings, Part II*, LNCS 2330, pages 844–853. Springer, 2002.
Cité page 23.

[52] Z. Dargaye. *Vérification formelle d'un compilateur pour langages fonctionnels*. PhD thesis, Université Paris 7 Diderot, July 2009.
Cité pages 26 et 146.

[53] Z. Dargaye and X. Leroy. A verified framework for higher-order uncurrying optimizations. *Higher-Order and Symbolic Computation*, 22(3) :199–231, 2009.
Cité pages 26 et 146.

[54] M. Daum. Reasoning on Data-Parallel Programs in Isabelle/Hol. In *C/C++ Verification Workshop*, 2007.
Cité page 18.

[55] R. Di Cosmo, S. Pelagatti, and Z. Li. A calculus for parallel computations over multidimensional dense arrays. *Computer Language Structures and Systems*, 33(3-4) :82–110, 2007.
Cité page 24.

[56] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *CACM*, 18(8) :453–457, 1975.
Cité page 4.

[57] D. C. Dracopoulos and S. Kent. Speeding up genetic programming : A parallel BSP implementation. In *First Annual Conference on Genetic Programming*. MIT Press, July 1996.
Cité page 14.

[58] J. Falcou, K. Hamidouche, and D. Etiemble. Hybrid Bulk Synchronous Parallelism Library for Clustered SMP Architectures. In *4th workshop on High-Level Parallel Programming and Applications (HLPP)*, pages 55–62. ACM, 2010.
Cité page 20.

[59] 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, 2007.
Cité page 5.

[60] M. Flynn. Some computer organizations and their effectiveness. In *Trans. on Computers*, volume C-21(9), pages 948–960. IEEE, 1972.
Cité page 8.

[61] J. Fortin and F. Gava. Bsp-why : an intermediate language for deductive verification of bsp programs. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, HLPP '10, pages 35–44. ACM, 2010.
Cité page 21.

[62] P. Fradet and J. Mallet. Compilation of a specialized functional language for massively parallel computers. *Journal of Functional Programming*, 10(6) :561–605, 2000.
Cité page 24.

[63] F. Gava. *Approches fonctionnelles de la programmation paralléle et des méta-ordinateurs. Sémantiques, implantations et certification.* PhD thesis, University Paris Val-de-Marne, LACL, 2005.
Cité pages 63 et 146.

[64] F. Gava, M. Guedj, and F. Pommereau. A BSP algorithm for the state space construction of security protocols. In *Ninth International Workshop on Parallel and Distributed Methods in Verification (PDMC 2010)*, pages 37–44. IEEE, 2010.
Cité page 14.

[65] L. Gesbert. *Développement systématique et sûreté d'exécution en programmation parallèle structurée.* PhD thesis, University Paris Est, LACL, 2009.
Cité pages 23, 58, 63, 105 et 110.

[66] L. Gesbert, Z. Hu, F. Loulergue, K. Matsuzaki, and J. Tesson. Systematic Development of Correct Bulk Synchronous Parallel Programs. In *The 11th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 334–340. IEEE Computer Society, 2010.

[67] S. Glondu. Extraction certifiée dans Coq-en-Coq. In A. Schmitt, editor, *Vingtièmes Journées Francophones des Langages Applicatifs (JFLA 2009)*, volume 7.2 of *Studia Informatica Universalis*, pages 383–410, 2009.
Cité pages 26, 72 et 146.

[68] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks : high-level structured parallel programming enablers. *Software, Practrice & Experience*, 40(12) :1135–1160, 2010.
Cité page 22.

[69] G. Gopalakrishnan and R. M. Kirby. Toward reliable and efficient message passing software through formal analysis. In *International Parallel and Distributed Processing Symposium (IPDPS 2006)*. IEEE, 2006. doi :10.1109/IPDPS.2006.1639578.
Cité page 12.

[70] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In Bougé et al. [27].
Cité page 23.

[71] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local Reasoning for Storable Locks and Threads. In Z. Shao, editor, *Programming Languages and Systems*, volume 4807 of *LNCS*, pages 19–37. Springer Berlin / Heidelberg, 2007.
Cité page 12.

[72] L. Granvilliers, G. Hains, Q. Miller, and N. Romero. A system for the high-level parallelization and cooperation of constraint solvers. In Y. Pan, S. G. Akl, and K. Li, editors, *Proceedings of International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 596–601, Las Vegas, USA, 1998. IASTED/ACTA Press.
Cité page 14.

[73] B. Grégoire and J. L. Sacchini. Combining a verification condition generator for a bytecode language with static analyses. In *Trustworthy Global Computing, Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers*, volume 4912 of *Lecture Notes in Computer Science*, pages 23–40. Springer, 2007.
Cité page 6.

[74] Y. Gu, B.-S. Lee, and W. Cai. JBSP : A BSP programming library in Java. *Journal of Parallel and Distributed Computing*, 61(8) :1126–1142, 2001.
Cité page 20.

[75] C. Haack, M. Huisman, and C. Hurlin. Permission-based separation logic for multithreaded java programs. *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica*, 15 :13–23, 2011.
Cité page 12.

[76] T. Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.
Cité page 6.

[77] K. Hamidouche, A. Borghi, P. Esterie, J. Falcou, and S. Peyronnet. Three High Performance Architectures in the Parallel APMC Boat. In *Ninth International Workshop on Parallel and Distributed Methods in Verification (PDMC 2010)*, pages 20–27, 2010.
Cité page 14.

[78] P. Hammarlund and B. Lisper. On the relation between functional and data parallel programming languages. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 210–219, New York, NY, USA, 1993. ACM.
Cité page 18.

[79] T. Hardin, F. Pessaux, P. Weis, and D. Doligez. *FoCaLiZe - Reference Manual*. LIP6 - CEDRIC, dec 2009. Version 0.6.
Cité page 7.

[80] L. Henrio and F. Kammüller. Functional Active Objects : Typing and Formalisation. *ENTCS*, 255 :83–101, 2009.
Cité page 24.

[81] L. Henrio, F. Kammüller, and B. Lutz. Aspfun : A typed functional active object calculus. *Science of Computer Programming*, In Press, Corrected Proof, 2011.
Cité page 24.

[82] P. Herms. Certification of a chain for deductive program verification. In Y. Bertot, editor, *2nd Coq Workshop, satellite of ITP'10*, 2010.
Cité page 6.

[83] M. Hidalgo-Herrero and Y. Ortega-Mallén. An Operational Semantics for the Parallel Language Eden. *Parallel Processing Letters*, 12(2) :211–228, 2002.
Cité page 24.

[84] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib : The BSP Programming Library. *Parallel Computing*, 24 :1947–1980, 1998.
Cité page 20.

[85] K. Hinsen. High-Level Parallel Software Development with Python and BSP. *Parallel Processing Letters*, 13(3) :461–472, 2003.
Cité page 20.

[86] C. A. R. Hoare. An axiomatic basis for computer programmation. *Communication of the ACM*, 12(10) :576–580, 1969.
Cité page 4.

[87] J. Holzmann. *The Spin Model Checker*. Addison Wesley, 2004.
Cité page 12.

[88] G. Horvitz and R. H. Bisseling. Designing a BSP version of ScaLAPACK. In B. H. et al., editor, *Proceedings Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Philadelphia, PA, 1999.
Cité page 14.

[89] Z. Hu. Towards Systematic Derivation of BSP Parallel Algorithms. Workshop on Parallelism Oblivious Programming, The University of Tokyo, July 2007.
Cité pages 15, 23 et 105.

[90] N. Javed and F. Loulergue. A Formal Programming Model of Orléans Skeleton Library. In V. Malyshkin, editor, *11th International Conference on Parallel Computing Technologies (PaCT)*, LNCS 6873. Springer, 2011.
Cité page 23.

[91] N. Javed and F. Loulergue. Parallel Programming and Performance Predictability with Orléans Skeleton Library. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 257–263. IEEE, 2011.
Cité page 23.

[92] N. Javed and F. Loulergue. Verification of a Heat Diffusion Simulation written with Orléans Skeleton Library. In *9th International Conference on Parallel Processing and Applied Mathematics (PPAM)*, LNCS. Springer, 2011. to appear.
Cité page 23.

[93] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5 :596–619, October 1983.
Cité page 11.

[94] Y. Kee and S. Ha. An Efficient Implementation of the BSP Programming Library for VIA. *Parallel Processing Letters*, 12(1) :65–77, 2002.
Cité page 20.

[95] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.
Cité page 13.

[96] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7) :107–115, 2009.
Cité page 6.

[97] M. Leyton, L. Henrio, and J. M. Piquer. Exceptions for algorithmic skeletons. In P. D'Ambra, M. R. Guarracino, and D. Talia, editors, *16th International Euro-Par Conference*, LNCS 6272, pages 14–25. Springer, 2010.
Cité page 23.

[98] M. Leyton and J. M. Piquer. Skandium : Multi-core Programming with Algorithmic Skeletons. In *PDP*, pages 289–296. IEEE, 2010.
Cité page 23.

[99] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3) :431–475, 2005.
Cité page 24.

[100] F. Loulergue, F. Gava, and D. Billiet. Bulk Synchronous Parallel ML : Modular Implementation and Performance Prediction. In V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *International Conference on Computational Science (ICCS)*, LNCS 3515, pages 1046–1054. Springer, 2005.
Cité page 20.

[101] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2-3) :255–279, Oct. 1990.
Cité page 6.

[102] K. Matsuzaki and K. Emoto. Implementing Fusion-Equipped Parallel Skeletons by Expression Templates. In *21st International Workshop on Implementation and Application of Functional Languages (IFL)*, LNCS. Springer, 2009.
Cité page 23.

[103] K. Matsuzaki, Z. Hu, K. Kakehi, and M. Takeichi. Systematic Derivation of Tree Contraction Algorithms. *Parallel Processing Letters*, 15(3) :321–336, 2005.
Cité page 23.

[104] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In *InfoScale'06 : Proceedings of the 1st international conference on Scalable information systems*. ACM Press, 2006.
Cité page 23.

[105] W. F. McColl. Universal computing. In Bougé et al. [27].
Cité page 13.

[106] L. Meertens. Algorithmics – Towards Programming as a Mathematical Activity. In *Proceedings of CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.
Cité page 6.

[107] L. Meertens. First Steps Towards the Theory of Rose Trees. CWI, Amsterdam, IFIP Working Group 2.1 Working paper 592 ROM-25, 1988.
Cité page 6.

[108] Q. Miller. BSP in a Lazy Functional Context. In *Trends in Functional Programming*, volume 3. Intellect Books, may 2002.
Cité page 20.

[109] A. Morihata, K. Matsuzaki, Z. H. Hu, and M. Takeichi. The third homomorphism theorem on trees : downward & upward lead to divide-and-conquer. In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*, pages 177–185. ACM Press, 2009.
Cité pages 23 et 145.

[110] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. Automatic Inversion Generates Divide-and-Conquer Parallel Programs. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 146–155. ACM Press, 2007.
Cité page 23.

[111] S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of programming using dependent types. In P. Audebaud and C. Paulin-Mohring, editors, *Mathematics of Program Construction*, volume 5133 of *LNCS*, pages 268–283. Springer Berlin / Heidelberg,

2008.
Cité page 7.

[112] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot : dependent types for imperative programs. In J. Hook and P. Thiemann, editors, *13th ACM SIGPLAN international conference on Functional programming (ICFP 2008)*, pages 229–240. ACM, 2008.
Cité page 145.

[113] B. Nichols, D. Buttlar, and J. Proulx Farrell. *Pthreads Programming : A POSIX Standard for Better Multiprocessing*. O'Reilly, 1996.
Cité page 9.

[114] T. Nipkow, D. von Oheimb, and C. Pusch. µJava : Embedding a programming language in a theorem prover. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, volume 175 of *NATO Science Series F : Computer and Systems Sciences*, pages 117–144. IOS Press, 2000. `http://isabelle.in.tum.de/Bali/papers/MOD99.html`.
Cité page 5.

[115] P. W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375 :271–307, 2007.
Cité page 12.

[116] S. Owicki and D. Gries. Verifying properties of parallel programs : an axiomatic approach. *Communications of the ACM*, 19(5) :279–285, 1976.
Cité page 11.

[117] R. O. Rogers and D. B. Skillicorn. Using the BSP cost model to optimise parallel neural network training. *Future Generation Computer Systems*, 14(5-6) :409–424, 1998.
Cité page 14.

[118] A. Schimpf, S. Merz, and J.-G. Smaus. Construction of Büchi Automata for LTL Model Checking Verified in Isabelle/HOL. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *LNCS*, pages 424–439. Springer, 2009.
Cité page 6.

[119] P. Schnoebelen, editor. *Vérification de logiciels : techniques et outils du model-checking*. Vuibert, 2000.
Cité page 3.

[120] S. F. Siegel. Verifying Parallel Programs with MPI-Spin. In F. Cappello, T. Hérault, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI 2007)*, LNCS 4757, pages 13–14. Springer, 2007. doi :10.1007/978-3-540-75416-9_8.
Cité page 12.

[121] S. F. Siegel and G. S. Avrunin. Verification of MPI-Based Software for Scientific Computation. In S. Graf and L. Mounier, editors, *Model Checking Software,*

11th International SPIN Workshop, LNCS 2989, pages 286–303. Springer, 2004. doi :10.1007/b96721.
Cité page 12.

[122] S. F. Siegel and G. S. Avrunin. Modeling wildcard-free MPI programs for verification. In K. Pingali, K. A. Yelick, and A. S. Grimshaw, editors, *Symposium on Principles and Practice of Parallel Programming (PPoPP 2005)*, pages 95–106. ACM, 2005. doi :10.1145/1065944.1065957.
Cité page 12.

[123] D. Skillicorn. Parallelism and the Bird-Meertens Formalism. FTP from qucis.queensu.ca in /pub/skill, 1992.
Cité page 22.

[124] D. B. Skillicorn. *Foundations of Parallel Programming.* Number 6 in International Series on Parallel Computation. Cambridge University Press, 1994.
Cité page 22.

[125] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3) :249–274, 1997.
Cité page 13.

[126] M. Snir and W. Gropp. *MPI the Complete Reference.* MIT Press, 1998.
Cité page 9.

[127] M. Sozeau. *Un environnement pour la programmation avec types dépendants.* PhD thesis, University Paris South, LRI, 2008.
Cité pages 42 et 51.

[128] A. Stewart. A programming model for BSP with partitioned synchronisation. *Formal Aspects of Computing*, 23(4) :421–432, 2011.
Cité page 21.

[129] A. Stewart and M. Clint. BSP-style Computation : a Semantic Investigation. *The Computer Journal*, 44(3) :174–185, 2001.
Cité page 21.

[130] A. Stewart, M. Clint, and J. Gabarró. Axiomatic Frameworks for Developing BSP-Style Programs. *Parallel Algorithms and Applications*, 14 :271–292, 2000.
Cité page 21.

[131] A. Stewart, M. Clint, and J. Gabarró. Barrier synchronisation : Axiomatisation and relaxation. *Formal Aspects of Computing*, 16(1) :36–50, 2004.
Cité page 21.

[132] W. J. Suijlen. BSPonMPI. http ://bsponmpi.sourceforge.net.
Cité page 20.

[133] J. Tesson, H. Hashimoto, Z. Hu, F. Loulergue, and M. Takeichi. Program Calculation in Coq. In *Thirteenth International Conference on Algebraic Methodology And Software Technology (AMAST2010)*, LNCS 6486, pages 163–179. Springer, 2010.
Cité page 100.

[134] J. Tesson and F. Loulergue. A Verified Bulk Synchronous Parallel ML Heat Diffusion Simulation. In *11th International Conference on Computational Science (ICCS 2011)*, Procedia Computer Science, pages 36–45. Elsevier, 2011.

[135] The Coq Development Team. The Coq Proof Assistant. `http://coq.inria.fr`.
Cité page 41.

[136] The SDPP Development Team. Program Calculation In Coq. `http://traclifo.univ-orleans.fr/SDPP`, 2011.
Cité page 100.

[137] *C* Programming Guide*. Thinking Machine Corporation, 1990.
Cité page 13.

[138] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In L. Caires and V. Vasconcelos, editors, *CONCUR 2007 – Concurrency Theory*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer Berlin / Heidelberg, 2007.
Cité page 12.

[139] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby. ISP : a tool for model checking MPI programs. In *PPoPP '08 : Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 285–286. ACM, 2008.
Cité page 12.

[140] L. G. Valiant. A bridging model for parallel computation. *Comm. of the ACM*, 33(8) :103, 1990.
Cité page 13.

[141] E. van der Weegen and J. McKinna. A Machine-checked Proof of the Average-case Complexity of Quicksort in Coq. In S. Berardi, F. Damiani, and U. de'Liguoro, editors, *Types for Proofs and Programs, International Conference (TYPES 2008)*, LNCS 5497, pages 256–271. Springer, 2008.
Cité page 146.

[142] E. Violard. A semantic framework to address data locality in data parallel languages. *Parallel Computing*, 30(1) :139–161, 2004.
Cité page 18.

[143] E. Violard, S. Genaud, and G.-R. Perrin. Refinement of data parallel programs in PEI. ICPS - Univ. Louis Pasteur, Strasbourg, 1997.
Cité page 19.

[144] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.
Cité page 51.

[145] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient Stateful Dynamic Partial Order Reduction. In K. Havelund, R. Majumdar, and J. Palsberg, editors, *Model Checking Software, 15th International SPIN Workshop*, LLNCS 5156, pages

288–305. Springer, 2008.
Cité page 12.

Index des définitions Coq

Ce document a été préparé à l'aide de l'éditeur de texte GNU Emacs, des outils coqdoc et coq-tex ainsi que du logiciel de composition typographique LaTeX 2ε.

Julien TESSON

Résumé :

Concevoir et implanter des programmes parallèles est une tâche complexe, sujette aux erreurs. La vérification des programmes parallèles est également plus difficile que celle des programmes séquentiels.
Pour permettre le développement et la preuve de correction de programmes parallèles, nous proposons de combiner le langage parallèle fonctionnel quasi-synchrone BSML, les squelettes algorithmiques - qui sont des fonctions d'ordre supérieur sur des structures de données réparties offrant une abstraction du parallélisme - et l'assistant de preuve Coq, dont le langage de spécification est suffisamment riche pour écrire des programmes fonctionnels purs et leurs propriétés.
Nous proposons un plongement des primitives BSML dans la logique de Coq sous une forme modulaire adaptée à l'extraction de programmes. Ainsi, nous pouvons écrire dans Coq des programmes BSML, raisonner dessus, puis les extraire et les exécuter en parallèle. Pour faciliter le raisonnement sur ceux-ci, nous formalisons le lien entre programmes parallèles, manipulant des structures de données distribuées, et les spécifications, manipulant des structures séquentielles. Nous prouvons ainsi la correction d'une implantation du squelette algorithmique BH, un squelette adapté au traitement de listes réparties dans le modèle de parallélisme quasi synchrone.
Pour un ensemble d'applications partant d'une spécification d'un problème sous forme d'un programme séquentiel simple, nous dérivons une instance de nos squelettes, puis nous extrayons un programme BSML avant de l'exécuter sur des machines parallèles.

Mots clés : Vérification formelle, squelettes algorithmiques, parallélisme quasi-synchrone, dérivation de programmes, programmation parallèle, programmation fonctionnelle, assistant de preuve

Environment for the systematic development and proof of correction of functional parallel programs

Abstract :

Parallel program design and implementation is a complex, error prone task. Verifying parallel programs is also harder than verifying sequential ones.
To ease the development and the proof of correction of parallel programs, we propose to combine the functional bulk synchronous parallel language BSML ; the algorithmic skeleton, that are higher order function on distributed data structures which offer an abstraction of the parallelism ; and the Coq proof assistant, who's specification language is rich enough to write purely functional programs together with their properties.
We propose an embedding of BSML primitives in the Coq logic in a modular form, adapted to program extraction. So we can write BSML programs in Coq, reason on them, extract them and then execute them in parallel. To ease the specification of these programs, we formalise the relation between parallel programs using distributed data structures and specification using sequential data structure. We prove the correctness of an implementation of the BH skeleton. This skeleton is devoted to the treatment of distributed lists in the BSP model. For a set of application, starting from a sequential specification of a problem, we derive an instance of our skeletons, then extract a BSML program which is executed on parallel machines.

Keywords : Verification, algorithmic skeletons, bulk synchronous parallelism, program derivation, parallel programming, functional programming, proof assistant

Laboratoire d'Informatique Fondamentale d'Orléans
(LIFO)
Université d'Orléans - Faculté des Sciences
Bâtiment IIIA
Rue Léonard de Vinci
B.P. 6759
F-45067 ORLEANS Cedex 2, France