Méthodes et outils pour la spécification et la preuve de propriétés difficiles de programmes séquentiels

Martin Clochard

université PARIS-SACLAY

NNT : 2018SACLS071

Thèse de doctorat

UNIVERSITÉ PARIS SUD
Comprendre le monde, construire l'avenir

Méthodes et outils pour la spécification et la preuve de propriétés difficiles de programmes séquentiels

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université Paris-Sud

École doctorale n°580 : sciences et technologies de l'information et de la communication
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Orsay, le 30/03/2018, par
Martin Clochard

Composition du Jury :

François Pottier Directeur de recherche, Inria Paris (Inria Paris)	Président
Sandrine Blazy Professeure, Université de Rennes 1 (Institut de Recherche en Informatique et Systèmes Aléatoires)	Rapporteure
Alexandre Miquel Professeur, Universidad de la República, Montevideo, Uruguay (Instituto de Matemática y Estadística)	Rapporteur
Hubert Comon-Lundh Professeur, ENS Paris-Saclay (Lab. Spécification et Vérification)	Examinateur
Claude Marché Directeur de recherche, Université Paris-Sud (Inria Saclay)	Directeur de thèse
Andrei Paskevich Maître de Conférences, Université Paris-Sud (Lab. de Recherche en Informatique)	Co-Directeur de thèse

Remerciements

Je tiens tout d'abord à remercier mes encadrants de thèse, Claude Marché et Andrei Paskevich, pour leur soutien constant durant mon stage de master 2 puis ces années de thèses.

Je souhaite également remercier les rapporteurs, Sandrine Blazy et Alexandre Miquel, pour avoir accepté de relire et de rapporter ma thèse, ainsi que pour leurs remarques. Je remercie également François Pottier et Hubert Comon d'avoir accepté de faire partie de mon Jury.

Je remercie aussi les membres de l'équipe VALS, actuels et passé, pour avoir créé un cadre de recherche très vivant. L'ambiance de l'équipe a été un élément moteur pour moi. Je remercie en particulier Léon Gondelman et Mário Pereira, qui ont partagé leur bureau avec moi. Merci également à Jean-Christophe Filliâtre pour son enthousiasme communicatif lors des discussions autour des mathématiques et de l'informatique.

Enfin, je remercie ma famille, qui m'a soutenue tout au long de mon parcours. Je n'en serais sans doute pas arrivé là sans l'aide de mes parents.

Table des matières

Table des figures

Chapitre 1

Introduction

Cette thèse se positionne dans le domaine de la vérification de programmes. Le but d'une telle vérification est de démontrer qu'un programme fait bien ce que l'on attend de lui. Ce problème est critique dans le cas de certains logiciels, comme ceux embarqués dans les avions ou ceux utilisés dans une centrale nucléaire. Toute déviation du comportement vis-à-vis du comportement attendu peut entraîner des conséquences potentiellement désastreuses. Nous pouvons citer, parmi de nombreux exemples de telles conséquences, la célèbre explosion du lanceur Ariane 5 lors de son vol inaugural du 4 juin 1996, due à une conversion arithmétique invalide.

Notons que le problème de la vérification possède deux facettes. Premièrement, il faut arriver à traduire l'intuition du comportement attendu du programme, c'est-à-dire « ce que l'on attend de lui », en un énoncé formel qui reflète ce comportement. Cette étape correspond à la *spécification* du programme. Elle peut être source d'erreur si l'énoncé formel obtenu ne correspond pas au comportement intuitivement attendu. Deuxièmement, il faut ensuite arriver à établir cet énoncé formel, ce qui correspond à la *vérification* proprement dite. Cette vérification peut être difficile selon le degré de complexité de la spécification et du programme correspondant.

Étant donnée la dépendance de plus en plus importante envers des systèmes informatiques qui sont eux-mêmes de plus en plus complexes, il y a également un besoin de méthodes efficaces pour vérifier ces systèmes, y compris leur composants les plus compliqués. Ces méthodes peuvent aussi bien porter sur la partie vérification que sur la partie spécification. L'amélioration des méthodes de vérification permet de simplifier l'effort de vérification total, tandis que l'utilisation d'un format de spécification adapté au type de comportement attendu permet de limiter les erreurs de spécification. De plus, la mise en œuvre d'une méthode de vérification peut être considérablement facilitée si la spécification du programme est écrite sous une forme

adéquate.

1.1 Méthodes de vérification pour les programmes

Un certain nombre de méthodes ont été développées pour vérifier qu'un programme se comporte comme prévu. Parmi celles-ci, nous pouvons notamment mentionner l'interprétation abstraite, la vérification de modèles (*model checking*), et la vérification déductive de programmes.

L'interprétation abstraite [34, 33] est une méthode qui consiste essentiellement à effectuer une sur-approximation des états possibles d'un programme. Cette sur-approximation est calculée par un outil, l'interpréteur abstrait, à partir de ce que l'on appelle un « domaine abstrait ». Les domaines abstraits permettent de représenter de manière symbolique un ensemble d'états du programme. La sur-approximation est alors calculée dans le domaine abstrait, et peut être utilisée pour vérifier des propriétés de sûreté, c'est-à-dire qu'il est impossible d'atteindre un état désigné comme « mauvais ». Le domaine abstrait est choisi en fonction du type de propriétés à vérifier. Par exemple, pour montrer que des valeurs entières respectent certaines bornes, ces valeurs sont typiquement représentées par des intervalles ou des polytopes convexes.

La vérification de modèles [73, 20] consiste à vérifier si un modèle d'un système informatique satisfait une propriété donnée. Dans ce contexte, les modèles sont typiquement représentés par des automates finis ou une de leurs variantes, et les propriétés par des formules de la logique temporelle. Ces méthodes fonctionnent essentiellement par exploration exhaustive de l'espace d'états, en utilisant des techniques symboliques pour réduire l'espace de recherche. Une autre méthode de réduction est celle de la vérification de modèles bornée, qui consiste à restreindre l'analyse aux k premières étapes d'exécution.

Enfin, la vérification déductive de programmes consiste à transformer une propriété à vérifier sur un programme en un énoncé logique, pour ensuite démontrer cet énoncé. Il s'agit de l'approche à laquelle nous nous intéressons ici.

1.2 Vérification déductive de programmes

Historiquement, le premier exemple significatif de méthode de vérification déductive est donné par la logique de Floyd-Hoare [45, 49], que nous appellerons par la suite logique de Hoare. L'idée derrière cette méthode est de caractériser le comportement du programme par deux formules logiques. La première, appelée la *précondition*, caractérise les états de départ pour lesquels le programme doit se comporter correctement. La seconde, appelé la *post-condition*, correspond aux propriétés attendues des états finaux du programme après son exécution. Pour un programme c, donner un tel couple pré/post-condition (P, Q) revient à donner une spécification formelle de son comportement. Un programme équipé d'une telle spécification est alors appelé un *triplet de Hoare*, représenté par la notation $\{P\}c\{Q\}$.

Un triplet de Hoare $\{P\}c\{Q\}$ est alors dit *valide* si pour tout état de départ satisfaisant P, si le programme c termine, alors son état final satisfait Q. Cette définition correspond plus précisément à la *correction partielle* du programme. Nous pouvons également interpréter la validité d'un triplet de Hoare par la notion de *correction*

totale, qui correspond à la correction partielle à laquelle on ajoute la terminaison garantie du programme. La logique de Hoare consiste alors en un ensemble de règles d'inférence correct et *relativement complet* [32] pour la validité des triplets de Hoare, fournissant ainsi une méthode de preuve pour la vérification des programmes.

Le développement des méthodes de vérification déductive a donné naissance à un certain nombre d'outils de vérification. Ces outils possèdent deux avantages principaux. Le premier est l'automatisation partielle ou totale de la méthode de preuve sous-jacente. Le deuxième est un gain de sécurité potentiel, puisque la vérification du programme est effectuée de manière mécanisée. Cela permet de réduire le risque d'erreur due à une preuve manuelle.

Il existe de nombreux outils de vérification déductive, lesquels utilisent différentes méthodes de vérification. La première est d'utiliser un système de déduction pour une logique de programme. Les outils de cette première catégorie sont typiquement implémentés par une bibliothèque dans un assistant de preuve, laquelle fournit une logique de programme avec ses règles d'inférence. Les bibliothèques couramment utilisées correspondent en général à une variante de la logique de séparation introduite par Reynolds [74], une extension de la logique de Hoare destinée à faciliter le raisonnement sur la structure de la mémoire. Nous pouvons par exemple mentionner la bibliothèque utilisée dans le projet seL4 [80] pour la vérification de code C, basée sur Isabelle/HOL [70]. Nous pouvons également mentionner, dans l'assistant de preuve Coq [11], les bibliothèques Ynot [69] et CFML [17] ou la formalisation de la logique Iris [53].

Une autre catégorie est celle des outils basés sur la génération de *conditions de vérification* (souvent notées VC). Le principe est de partir d'un programme annoté, par exemple par des pré/post-conditions, et de générer des formules mathématiques correspondant à la correction du programme. Ces dernières sont ensuite démontrées par l'outil, typiquement par envoi à un démonstrateur automatique. Une méthode classique de génération est celle du calcul de plus faible pré-condition introduit par Dijkstra [40]. Étant donnés un programme annoté c et une post-condition Q, il est possible de calculer la formule $WP(c, Q)$ la plus faible telle que le triplet $\{WP(c, Q)\}c\{Q\}$ soit valide. En particulier, la validité d'un triplet de Hoare $\{P\}c\{Q\}$ peut se réduire à la formule $(P \Rightarrow WP(c, Q))$.

L'approche à base de calcul de plus faible pré-condition est celle utilisée par les outils de vérification déductive ESC/Java [31], Frama-C/WP [35], Boogie [8], Why3 [43, 15] F* [78] et OpenJML [30]. Par extension, il s'agit également de la méthode utilisée par tous les vérificateurs qui fonctionnent par encodage vers ces outils, comme Chalice [58], Dafny [56], Spec# [9] et VCC [29] pour Boogie, ou Frama-C/Jessie [64], Krakatoa [65] et SPARK2014 [67] pour Why3.

Une autre approche de génération de conditions de vérification est celle de l'exécution symbolique, principalement utilisée pour la logique de séparation. Cette approche est notamment utilisée dans les outils KeY [4], VeriFast [52] ou Viper [68].

Enfin, une dernière catégorie est formée par les outils basés sur le raffinement. Le principe du raffinement, originellement introduit par Dijkstra [39] et formalisé par Back [7], est de partir d'une version abstraite du programme, qui sert de spécification, puis d'en dériver successivement des versions de plus en plus concrètes, jusqu'à arriver au programme voulu. Le programme obtenu est alors correct par construction, puisque celle-ci assure l'équivalence entre le programme final et sa version abstraite. La notion de raffinement est notamment au cœur de la méthode

B [1], notamment implémentée dans l'outil Atelier B, ou de son extension Event-B implémentée dans l'outil Rodin [2].

1.3 Problématique de cette thèse

La vérification effective d'un programme peut poser de nombreuses difficultés pratiques. Nous pouvons citer un certain nombre d'exemples classiques qui rendent la phase de vérification difficile, comme l'utilisation de certaines constructions telles que le parallélisme ou l'ordre supérieur, ou encore la présence de structures de données basées sur les pointeurs. Nous pouvons également avoir des problèmes à cause de la taille du programme à vérifier. Mais même lorsque l'on met ces cas de côté, les concepts mis en jeu derrière le programme peuvent suffire à faire obstacle à la vérification.

En effet, certains programmes peuvent être assez courts et n'utiliser que des constructions simples, et pourtant s'avérer très difficiles à vérifier. Nous pouvons prendre l'exemple des algorithmes de calcul numérique efficace, comme l'algorithme de multiplication de grands entiers de Karatsuba ou l'algorithme de multiplication de matrices de Strassen. Ces algorithmes sont récursifs, et fonctionnent en réduisant la multiplication à plusieurs multiplications sur des entrées plus petites. Par des manipulations algébriques, le nombre de multiplications récursives effectuées à chaque appel est réduit de un par rapport à une multiplication récursive directe, ce qui se traduit par un gain de complexité asymptotique.

Ces deux algorithmes peuvent être implémentés en utilisant des tableaux pour représenter les grands entiers ou les matrices, c'est-à-dire une structure de données très simple. De plus, une implémentation directe de ces algorithmes n'utilise pas de constructions plus compliquées qu'une fonction récursive principale. Il n'est pas difficile de donner une spécification exprimant que les programmes obtenus implémentent bien la multiplication d'entiers/de matrices. Cependant, obtenir une preuve mécanisée que ces programmes satisfont effectivement cette spécification est loin d'être immédiat. Pour obtenir ladite preuve, nous devons tout d'abord corréler chacune des opérations effectuées sur les tableaux avec des opérations sur les entiers/matrices. Nous devons donc assister le processus de vérification en indiquant la nature de cette corrélation. Ensuite, nous devons vérifier que les opérations effectuées sur les entiers/matrices représentés coïncident bien avec la multiplication que l'on cherche à effectuer, ce qui revient ainsi à appliquer un certain nombre de propriétés algébriques. En pratique, il est très difficile, voire quasi impossible pour un algorithme de vérification d'effectuer la totalité de ce travail automatiquement. Cela induit donc un effort de vérification manuel conséquent, qui ne provient pas des constructions utilisées par le programme, mais de sa spécification.

Un exemple encore plus flagrant est celui d'une fonction de compilation simple. Une telle fonction traduit un programme d'un langage de programmation dit « source » vers un programme d'un langage de programmation dit « cible » tout en préservant le comportement observable du programme. Le programme source est typiquement représenté par un arbre de syntaxe abstraite, représentant les constructions syntaxiques du langage source. Le programme cible peut être également représenté par un tel arbre, ou par une séquence d'instructions dans le cas d'un langage type assembleur. Une telle fonction de compilation est typiquement implémentée par

récursivité sur l'arbre de syntaxe abstraite du programme source.

Si l'on utilise des types algébriques pour représenter les arbres de syntaxe abstraite, nous remarquons que cet exemple n'utilise aucune construction vraiment compliquée. Pourtant, la spécification et la vérification d'une telle procédure de compilation sont ardues. En effet, nous ne pouvons même pas spécifier un tel programme sans tout d'abord définir la sémantique des langages de programmation source et cible. Ensuite, la spécification naturelle indiquant que les deux programmes ont le même comportement observable n'est pas toujours facile à formaliser. Cela dépend notamment des sémantiques choisies, ainsi que de ce qui est considéré comme observable. Enfin, la préservation du comportement observable n'est pas une propriété facile à établir de manière automatique, car elle demande de mettre en relation de manière fine les comportements du programme source et du programme produit par la fonction de compilation. Cette dernière partie demande un effort manuel très important. Certains compilateurs ont été entièrement prouvés, comme CompCert [60] pour le langage C ou CakeML [54] pour Standard ML, mais cela représente le travail de plusieurs années et de plusieurs personnes, elles-mêmes des développeurs experts de l'outil de vérification utilisé.

Nous observons donc qu'il y a un grand nombre de cas où la vérification mécanisée d'un programme demande un certain effort manuel, tout simplement car les outils automatiques échouent à saisir les concepts sous-jacents à un programme. Dans un environnement de vérification déductive basé sur les démonstrateurs automatiques, cela se traduit par un échec de ces démonstrateurs à démontrer les conditions de vérification obtenues pour un programme, même si celles-ci sont vraies. Nous nous posons donc la question suivante : dans le contexte d'un tel environnement, quelles méthodes appliquer quand les démonstrateurs échouent ?

Cette question n'est toutefois pas assez précise. En effet, il est typiquement possible dans ces environnements d'insérer des assertions intermédiaires dans le programme. Cela permet d'énoncer des propriétés qui doivent être vraies à un point donné du programme, lesquelles doivent alors être vérifiées en arrivant au point associé, et sont considérées comme établies au niveau des points subséquents. Nous pouvons employer ces assertions pour fragmenter la condition de vérification, ce qui permet généralement de la mettre à portée des démonstrateurs automatiques. Il s'agit donc d'une méthode parfaitement applicable en cas d'échec des démonstrateurs automatiques. Cependant, cette approche est très manuelle et montre rapidement ses limites. De plus, son usage systématique tend en pratique à créer un effet de saturation contre-productif pour les démonstrateurs automatiques, ce qui impose une fragmentation encore plus importante. L'objectif serait donc plutôt de trouver des méthodes pour réduire l'effort manuel nécessaire à une telle vérification.

De plus, dans un environnement de vérification déductive, le problème de la vérification dépend également du format de spécification choisi. En effet, le choix d'un format de spécification plutôt qu'un autre pour les fragments internes du programme peut lourdement impacter les conditions de vérification générées par l'outil, et donc la possibilité de démontrer celles-ci de manière automatique ou non.

Nous obtenons ainsi la question à laquelle nous tentons de répondre dans cette thèse :

> Dans le contexte d'un environnement de vérification déductive de programmes basé sur les démonstrateurs automatiques, quelles méthodes

appliquer pour réduire l'effort nécessaire à la fois pour spécifier des comportements attendus complexes, ainsi que pour démontrer qu'un programme respecte ces comportements attendus ?

Comme nous avons observé que ces difficultés peuvent apparaître même pour des programmes basés sur des constructions simples, nous avons principalement restreint notre attention à ce type de programmes. Nous nous sommes en particulier restreints aux programmes séquentiels.

1.4 Contributions de cette thèse

Pour mener notre étude, nous nous sommes placés dans le cadre de l'environnement de vérification déductive de programmes Why3. La vérification de programmes en Why3 est basée sur la génération de conditions de vérification, et l'usage de démonstrateurs externes pour les prouver, que ces démonstrateurs soient automatiques ou interactifs. Nous avons développé plusieurs méthodes, certaines générales et d'autres spécifiques à des classes de programmes, pour réduire l'effort manuel. Pour pouvoir appliquer certaines de ces méthodes, nous avons ajouté de nouvelles fonctionnalités dans Why3.

Nous présentons dans le chapitre 2 l'environnement de preuve de programmes Why3, ainsi que les fonctionnalités particulières que nous exploitons ou avons implémentées pour assister le processus de vérification de programmes. Nous proposons notamment un mécanisme léger de preuve déclarative, basé sur la notion d'indicateurs de coupures [23], pour écrire les étapes d'une preuve de manière plus efficace que par l'usage d'assertions intermédiaires brutes. Nous avons implémenté ce mécanisme en Why3. Nous ajoutons également une fonctionnalité permettant d'effectuer les raisonnements par induction directement dans Why3, sans passer par un assistant de preuve interactif. Enfin, nous démontrons à travers deux études de cas comment les fonctionnalités de calcul de Why3 peuvent être exploitées pour simplifier considérablement certaines preuves. Notre première étude de cas est un compilateur pour un langage `While` [26], où nous effectuons la preuve par le biais d'une variante du calcul de plus faible pré-condition sur le langage cible. Notre seconde étude de cas est une vérification de l'algorithme de multiplication de matrices de Strassen [27], pour laquelle nous effectuons une preuve par réflexion.

Nous nous intéressons ensuite dans le chapitre 3 au problème des débordements arithmétiques. Il s'agit d'un phénomène qui peut se produire lorsque l'on effectue des calculs avec des entiers de taille fixe. Un débordement arithmétique se produit lorsque le résultat d'une opération est trop grand ou trop petit pour être stocké dans la taille donnée, ce qui donne lieu soit à un plantage immédiat ou à la création d'une valeur incorrecte. Dans les deux cas, l'issue est souvent fatale. La vérification d'un programme manipulant des entiers de taille fixe passe donc par la preuve d'absence de débordements arithmétiques. Nous avons développé en collaboration avec Jean-Christophe Filliâtre et Andrei Paskevich une nouvelle méthode [25] pour ce problème. Celle-ci permet de traiter très facilement l'absence de débordement arithmétique pour une classe particulière d'utilisation des entiers de taille fixe, qui est typiquement difficile à traiter par les méthodes standards. Notre méthode consiste grossièrement à borner la taille de certains entiers par le temps d'exécution, et à relâcher le critère d'absence de débordement arithmétique, en prouvant l'absence

de débordement seulement pour les premières centaines d'années de l'exécution du programme. Ce relâchement du critère n'a aucune incidence pour toute utilisation en pratique.

Enfin, nous nous intéressons au développement d'une bibliothèque générique pour la spécification et la preuve de programmes générateurs de code. Nous reprenons pour cela la méthode employée pour la preuve de compilateur effectuée dans le chapitre 2. Le principe de cette méthode est de spécifier la génération de code par la production d'un triplet de Hoare valide portant sur le code produit, et à établir cette spécification par le biais d'une variante du calcul de plus faible pré-condition. Dans le chapitre 4, nous généralisons la notion de triplet de Hoare pour un langage particulier à une notion générique de *garanties* sur des jeux. Le choix des jeux nous permet de traiter de manière uniforme deux interprétations — existentielle et universelle — de la logique de Hoare, correspondant respectivement aux propriétés d'accessibilité (l'existence d'un comportement qui atteint un état donné) et de vivacité (la garantie que tout comportement atteigne un état avec de « bonnes » propriétés). Nous généralisons également le cadre de travail de manière à pouvoir capturer précisément les comportements des programmes qui ne terminent pas. Pour cela, nous caractérisons l'état d'un programme après une infinité d'étapes comme la borne supérieure des états atteints en temps fini. Nous donnons des règles de raisonnement basiques permettant de travailler avec la notion de garantie, notamment un théorème crucial de simulation entre deux jeux. Visant l'objectif final d'une bibliothèque générique de spécification et de preuve, nous avons entièrement formalisé en Why3 les notions développées dans le chapitre 4.

Dans le chapitre 5, nous développons une logique complète pour les garanties introduites dans le chapitre 4, ainsi qu'un analogue du calcul de plus faible précondition dédié à la preuve de garanties. La logique pour les garanties est inspirée de la logique de Hoare, mais avec des règles adaptées au cas des jeux. En particulier, les règles pour les garanties ne sont pas dirigées par une quelconque syntaxe de programme, car la notion de jeu n'en fournit pas. Nous adaptons les règles correspondant aux boucles et à la récursivité pour pouvoir gérer finement le cas des comportements qui ne terminent pas. Nous montrons ensuite comment obtenir un système de preuve pratique pour les garanties, basé sur la combinaison de *transformateurs de post-conditions*. Ce système est fortement inspiré du calcul de plus faible pré-condition. De même que notre logique, ce calcul n'est pas guidé par une syntaxe de programme. Les règles de combinaison de transformateurs doivent être utilisées explicitement, mais en contrepartie elles peuvent être utilisées de manière plus flexible. Toujours dans l'objectif d'obtenir une bibliothèque générique de spécification et de preuve, nous avons formalisé la logique proposée pour les garanties en Why3. À l'heure de la rédaction de cette thèse, la formalisation en Why3 des transformateurs de post-conditions est en cours.

Finalement, nous montrons dans le chapitre 6 comment utiliser les notions de jeux, de garanties, et de transformateurs de post-conditions pour vérifier un compilateur plus conséquent que celui vérifié pour un simple langage `While`. Nous montrons notamment comment utiliser ces notions pour vérifier la compilation de fonctions récursives, et comment prendre en compte la préservation des comportements qui ne terminent pas.

Chapitre 2

Facilités pour les preuves

Au cours de cette thèse, nous nous sommes rapidement rendu compte des limitations d'une approche complètement automatique de la vérification de programme. En effet, ce type d'approche est loin de fonctionner de manière systématique. Nous avons donc été amenés à développer des outils et méthodes pour assister le processus automatique. L'objectif de ce chapitre est de présenter ces outils, ainsi que quelques exemples que ceux-ci nous ont permis de vérifier.

```
theory AtLeastTwoElements
  type t                    (* Déclaration d'un type abstrait *)
  constant a : t            (* Déclaration d'une constante abstraite *)
  constant b : t
  axiom distinct : a ≠ b (* Déclaration d'un axiome *)
end
```

FIGURE 2.1 – Exemple de théorie axiomatique

Nous présentons tout d'abord l'outil Why3, qui est le cadre dans lequel nous avons mené nos expériences de preuve de programme. Nous présentons ensuite une transformation d'induction permettant d'effectuer des raisonnements par récurrence sur les prédicats inductifs. Cette transformation a été développé en collaboration avec Léon Gondelman. Dans la troisième section, nous présentons une extension que nous avons apporté à Why3 pour lui ajouter un mécanisme de preuve léger de preuve déclarative [23]. Dans la quatrième section, nous présentons une transformation de calcul intégrée à Why3. Cette transformation nous a permis de simplifier considérablement la preuve de certains programmes. Nous présentons deux exemples dans les deux dernières sections. Le premier est une preuve d'un petit compilateur [26], effectuée avec Léon Gondelman. Le second est une preuve de l'algorithme de Strassen [27], effectuée avec Léon Gondelman et Mário Pereira. Notre contribution principale à ces deux exemples est la méthodologie de preuve, qui s'appuie sur la transformation de calcul combiné à un étiquetage statique bien choisi des données.

2.1 Cadre de travail : l'environnement Why3

Nous menons nos expériences de preuve de programme dans le cadre de l'outil Why3. Celui-ci propose un langage riche, appelé WhyML, pour la spécification et l'écriture de programmes vérifiés. Il repose sur l'utilisation de démonstrateurs externes, qu'ils soient automatiques ou interactifs, pour obtenir les preuves des lemmes auxiliaires ainsi que des conditions de vérification.

Le fragment logique de WhyML [14], utilisé pour annoter les programmes, est une extension de la logique du premier ordre comprenant notamment des types polymorphes à la ML, des types algébriques, des définitions inductives et récursives, ainsi que certaines constructions d'ordre supérieur [24]. Les constructions comme le filtrage et la liaison peuvent être employées directement dans les formules et les termes. Ce fragment sert également à définir ou à axiomatiser des théories logiques, qui servent de base pour la spécification et la vérification des programmes. La figure 2.1 donne un exemple de théorie complètement axiomatique, et la figure 2.2 d'une théorie complètement définie. Seule la seconde génère des conditions de vérification, une par lemme.

Comme la majorité des démonstrateurs employés ne supportent pas toutes les constructions du langage, Why3 utilise durant le processus de génération des obligations de preuve une série de transformations pour éliminer celles qui ne sont pas supportées. Celles-ci peuvent également être employées explicitement par l'utilisateur pour changer la forme des obligations générées, *a priori* dans le but de faciliter

```
theory Peano
  type t = O | S t
  function add (n m: t) : t =
    match n with
    | O → m
    | S x → add x (S m)
    end
  predicate non_zero (n:t) = n ≠ O
  inductive even t =
    | Base : even O
    | Succ : even x → even (S (S x))
  lemma double_even : ∀a. even (add a a)
  lemma add_non_zero : ∀a b. non_zero a → non_zero (add a b)
end
```

FIGURE 2.2 – Exemple de théorie définie

```
module M

  use import int.Int
  use import ref.Ref

  let is_even (n: int) : bool
    requires { n ≥ 0 }
    ensures { result ↔ ∃k. n = 2 * k }
  = let r = ref n in
    let ghost k = ref 0 in
    while !r > 1 do
      invariant { n - !r = 2 * !k ∧ !r ≥ 0 }
      variant { !r }
      r := !r - 2; k := !k + 1
    done;
    !r = 0
end
```

FIGURE 2.3 – Exemple de programme WhyML

la tâche aux démonstrateurs automatiques. Une de nos contributions dans cette thèse est d'ajouter ou de modifier des transformations existantes pour faciliter cette assistance.

Le langage de programmation lui-même [44] est un dialecte de ML avec un certain nombre de restrictions pour rendre la preuve automatique viable. Par exemple, les procédures d'ordre supérieur ne sont pas acceptées. Le langage impose également la restriction que tous les alias d'une variable mutable doivent être connus statiquement. Pour décrire le comportement attendu d'un programme, les défini-

```
type list 'a =
  | Nil
  | Cons 'a (list 'a)

function size (l:list 'a) = match l with
  | Nil → 0 | Cons _ q → 1 + size l
  end

let rec lemma f (l: list 'a)
  requires { l ≠ Nil }
  ensures { size l ≥ 1 }
  variant { l }
= match l with
  | Nil → absurd
  | Cons _ q → if q ≠ Nil then f q
  end
(* Les conditions de vérification suivantes ont
   axiom f_lemma : ∀ l:list 'a. l ≠ Nil → size l ≥ 1
   comme hypothèse *)
```

FIGURE 2.4 – Exemple de fonction-lemme

tions de fonctions WhyML sont annotées par des contrats, sous la forme de pré- et post-conditions. Les boucles sont également munies d'invariants. Dans le but de vérifier la terminaison, les définitions récursives ainsi que les boucles peuvent également être munies de variants, c'est-à-dire des valeurs qui décroissent à chaque itération pour un ordre bien fondé. Nous pouvons également écrire des assertions vérifiées statiquement à des emplacements arbitraires du programme. L'outil Why3 utilise ces annotations pour générer des conditions de vérification via un calcul de plus faible pré-condition. Nous donnons un exemple de code WhyML dans la figure 2.3. La construction `use import int.Int` sert à importer la théorie prédéfinie des entiers, la suivante celle des références. Pour ce programme particulier, les conditions de vérification générées par Why3 sont démontrés quasi instantanément par les démonstrateurs automatiques.

Le langage WhyML permet également d'écrire du code pour assister la vérification. En plus des assertions, nous pouvons notamment écrire du code fantôme [42], c'est-à-dire des calculs et données qui peuvent être retirés du programme sans changer son comportement observable. Un usage typique du code fantôme est de propager les témoins de propriétés existentielles. Il s'agit précisément du rôle de la variable `k` dans le programme de la figure 2.3.

Nous pouvons également utiliser le langage de programmation de Why3 pour prouver des propriétés auxiliaires via le mécanisme des fonctions-lemmes. Ces objets sont des procédures annotées via le mot-clé `lemma`. L'outil Why3 impose alors la contrainte supplémentaire qu'une telle procédure soit sans résultat, que sa terminaison soit garantie, et ne produise aucun effet observable. La validité du contrat d'une telle procédure signifie alors que la pré-condition implique la post-condition pour

toute assignation des variables. L'outil Why3 transforme alors cette implication en lemme auxiliaire, utilisé pour démontrer toute obligation subséquente. Un exemple de fonction-lemme est donné en figure 2.4. Celle-ci permet d'établir que toute liste non vide est de taille au moins un. L'appel récursif correspond alors à l'utilisation de l'hypothèse de récurrence. Le mécanisme des fonctions-lemme permet notamment de simuler le raisonnement par récurrence sur les données, que les démonstrateurs automatiques sont généralement incapables d'effectuer.

L'outil Why3 fournit également un mécanisme d'extraction vers OCaml, ce qui permet d'exécuter les programmes vérifiés. En particulier, ce mécanisme est responsable de l'élimination complète du code fantôme, qui n'apparaît pas dans le programme extrait.

Pour une présentation plus complète de Why3 et WhyML, nous invitons le lecteur à se référer à la page web du projet http://why3.lri.fr, qui propose une introduction détaillée ainsi qu'une grande collection d'exemples.

2.2 Récurrence sur les prédicats inductifs

Nous avons rapidement identifié le raisonnement inductif comme étant un point bloquant pour la preuve automatique. Ce type de raisonnement est en effet peu ou pas supporté par les démonstrateurs automatiques génériques. De plus, les constructions inductives de Why3 ne sont pas toujours traduisibles correctement vers le langage de ces démonstrateurs. Nous devons donc avoir un autre moyen de faire un tel raisonnement. Bien entendu, Why3 nous donne la possibilité de décharger les obligations de preuves impliquées via un assistant de preuve interactif comme Coq [11], mais ce n'est pas toujours une solution satisfaisante. Nous sommes alors contraint de faire ces preuves intégralement au travers de l'assistant, même si la majorité du raisonnement est dans le fragment à la portée des démonstrateurs automatiques.

Dans le cadre de l'induction sur les valeurs d'un type algébrique, l'environnement Why3 nous fournit plusieurs mécanismes alternatifs. Par exemple, nous avons vu dans la section 2.1 que l'utilisation de fonction-lemmes récursives permet de simuler le raisonnement par récurrence. L'outil Why3 permet également l'application manuelle d'une transformation d'induction sur les valeurs, basée sur les mêmes heuristiques que celles proposées pour le langage Dafny [57]. Cependant, ces méthodes ne sont pas prévues pour traiter le cas des prédicats inductifs. Nous avons donc implémenté une nouvelle transformation `induction_pr` pour appliquer le raisonnement par induction dans le cas d'un prédicat inductif.

Les prédicats inductifs de Why3 sont définis comme le plus petit point fixe d'un ensemble de clauses, autrement dit comme la propriété la plus forte parmi ces points fixes. Par exemple, nous pouvons définir la clôture réflexive/transitive d'une relation de la manière suivante :

```
type t
predicate r t t
inductive closure t t =          (* Définition inductive *)
  | Refl : ∀x. closure x x
  | Step : ∀x y z. closure x y ∧ r y z → closure x z
```

De manière formelle, les clauses définissant un prédicat inductif I doivent prendre

la forme

$$\bigwedge_{c\in C}\left(\forall\overline{x_c}.F_c(I,\overline{x_c})\Rightarrow I(\overline{f_c}(\overline{x_c}))\right) \qquad (\mathcal{C})$$

sous la contrainte que les $(F_c)_{c\in C}$ soient des opérateurs croissant vis-à-vis de leur premier paramètre. La contrainte de monotonicité est la condition habituelle de Tarski pour garantir l'existence du plus petit point fixe, et montre également que I est alors la propriété la plus forte satisfaisant cet ensemble de clauses. En particulier, toute propriété satisfaisant la condition $(\mathcal{C})$ est conséquence de I, ce qui correspond exactement au principe d'induction.

Notre transformation d'induction exploite ce principe de la manière suivante. Étant donné un but, nous commençons par en extraire une hypothèse $I(\overline{f}(\overline{y}))$ où I est un prédicat inductif. Cela revient à réarranger le but sous la forme :

$$\forall\overline{xy}.\left(\bigwedge_{H\in\Gamma}H(\overline{x},\overline{y})\right)\Rightarrow I(\overline{f}(\overline{y}))\Rightarrow G(\overline{x},\overline{y})$$

où Γ correspond à l'ensemble des hypothèses qui se trouvaient avant l'hypothèse inductive de manière syntaxique. Intuitivement, Γ est alors le contexte de la formule, et G la propriété à prouver par récurrence.

Ensuite, nous sélectionnons certaines hypothèses de Γ et les déplaçons dans G. Notons que ce processus est conservatif, au sens que cela rend les clauses associées à G plus faibles. Nous pourrions donc sélectionner toutes les hypothèses de Γ. Cependant, ce choix peut faire énormément grossir les formules, et les rendre à la fois moins naturelles et moins faciles à prouver par le biais d'un outil. Nous procédons donc à une sélection via l'heuristique suivante : une hypothèse est déplacée de Γ vers G si elle dépend syntaxiquement de $\overline{y}$, ce qui la lie à l'induction. Le but à montrer prend alors la forme :

$$\forall\overline{x}.\left(\bigwedge_{H\in\Gamma}H(\overline{x})\right)\Rightarrow\forall\overline{y}.I(\overline{f}(\overline{y}))\Rightarrow G(\overline{x},\overline{y})$$

Enfin, nous introduisons des variables supplémentaires et des égalités pour mettre le but sous une forme qui permette d'appliquer le principe de récurrence :

$$\forall\overline{x}.\Gamma(\overline{x})\Rightarrow\forall\overline{z}.I(\overline{z})\Rightarrow(\forall\overline{y}.(\overline{z}=\overline{f}(\overline{y}))\Rightarrow G(\overline{x},\overline{y}))$$

Le but revient alors à énoncer que sous les hypothèses de Γ, la propriété

$$\varphi_{\overline{x}}(\overline{z})\triangleq\forall\overline{y}.(\overline{z}=\overline{f}(\overline{y}))\Rightarrow G(\overline{x},\overline{y})$$

est une conséquence logique de I. Par le principe d'induction, il suffit de démontrer que $\varphi_{\overline{x}}(\overline{z})$ satisfait les clauses définissant I. Comme il est fréquent de vouloir réutiliser I parmi les hypothèses de récurrence, nous renforçons cette propriété en $I\wedge\varphi_{\overline{x}}$. Nous renvoyons alors comme résultat final de la transformation un ensemble de buts correspondant à la validité de la condition $(\mathcal{C})$ pour $I\wedge\varphi_{\overline{x}}$:

$$\bigwedge_{c\in C}\left(\forall\overline{xx_c}.\left(\bigwedge_{H\in\Gamma}H(\overline{x})\right)\Rightarrow F_c(I\wedge\varphi_{\overline{x}},\overline{x_c})\Rightarrow\varphi_x(\overline{f_c}(\overline{x_c}))\right)$$

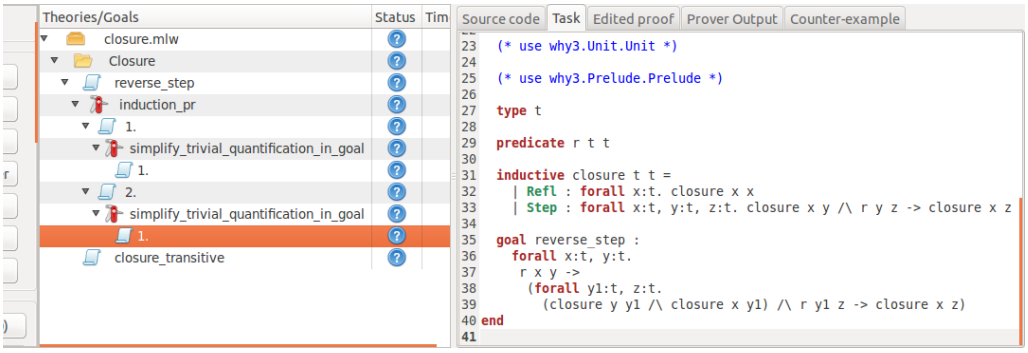

FIGURE 2.5 – Induction dans l'interface graphique de Why3

Nous avons délibérément omis $I(\overline{f_c}(\overline{x_c}))$ de la conclusion des clauses car nous savons que cette instance du prédicat inductif est vérifiée sous les hypothèses fournies. En effet, elle est conséquence de la propriété $F_c(I, \overline{x_c})$, qui suit de la croissance de F_c.

Armés de cette transformation, la preuve des buts où le raisonnement inductif est l'argument principal devient immédiate. Par exemple, reprenons l'exemple de la clôture transitive. Nous l'avons définie en accumulant les étapes à droite, mais nous pouvons démontrer que l'on peut également accumuler les étapes à gauche :

```
goal reverse_step : ∀x y z. r x y → closure y z → closure x z
```

Nous démontrons cette propriété en appliquant la transformation d'induction à ce but depuis l'interface graphique de Why3. Nous utilisons également une seconde transformation pour simplifier les quantifications du type $\forall x.(x = t) \Rightarrow P(x)$ en $P(t)$, obtenant ainsi le résultat de la figure 2.5. Nous appliquons cette seconde transformation car notre mécanisme d'induction introduit un grand nombre d'égalités, qui dans la vaste majorité des cas peuvent être éliminés de cette manière. Le résultat brut de l'induction consiste en effet en les deux sous-buts suivants, beaucoup moins lisibles :

```
goal reverse_step_1 : ∀x:t, y:t. r x y →
  ∀x1:t. ∀z:t. z = x1 → y = x1 → closure x z
goal reverse_step_2 : ∀x:t, y:t. r x y →
  ∀x1:t, y1:t, z:t.
     (closure x1 y1 ∧ ∀z1:t. z1 = y1 → y = x1 → closure x z1)
     ∧ r y1 z → ∀z1:t. z1 = z → y = x1 → closure x z1
```

Dans le cas où plusieurs choix d'inductions sont possibles, notre transformation sélectionne l'hypothèse syntaxiquement la plus à gauche, ce qui n'est pas nécessairement le choix voulu. Nous avons donc ajouté la possibilité de contrôler quelle hypothèse utiliser via le placement d'une étiquette `"induction"`, qui rend l'hypothèse marquée prioritaire pour cette sélection.

```
goal closure_transitive : ∀x y z.
  closure x y → ("induction" closure y z) → closure x z
```

Notre transformation d'induction a été utilisée dans plusieurs études de cas uti-

lisant Why3. Nous pouvons notamment mentionner une preuve d'un algorithme de résolution de chemin [19] qui fait un usage important de cette transformation. Dans une moindre mesure, l'exemple présenté dans la section 2.5 emploie également cette transformation.

2.3 Indicateurs de coupures

Le raisonnement par induction n'est pas le seul point bloquant pour la preuve automatique. Nous avons rapidement obtenu des obligations de preuves trop complexes pour être vérifiées par des démonstrateurs automatiques en temps raisonnable. Dans le cas de la vérification de programmes pour lesquels la preuve de correction est difficile, cela peut même devenir le cas de la majorité des conditions de vérification. Bien sûr, nous pouvons alors décharger ces obligations via un assistant de preuve interactif, mais ce n'est généralement pas une solution satisfaisante. En effet, l'envoi des obligations vers un assistant de preuve externe impose une traduction de la logique, ce qui a tendance à polluer la preuve. De plus, cette solution nous force à basculer entre les outils, ce qui devient rapidement gênant lorsque l'on doit traiter un grand nombre de buts.

Notons que nous pouvons contourner l'utilisation d'un assistant de preuve. Nous pouvons par exemple ajouter au contexte logique des lemmes intermédiaires pour faciliter la preuve. Cependant, cette solution ne passe pas à l'échelle. Nous avons constaté que les démonstrateurs automatiques ont tendance à saturer lorsque le contexte dépasse un seuil critique, à partir duquel ils ne prouvent plus que les formules les plus simples. Cela vient principalement du fait que la portée des lemmes intermédiaires dépasse largement leur cas attendu d'utilisation.

Pour pallier à ce problème, nous avons introduit un mécanisme léger de preuve déclarative au sein de l'outil Why3, permettant de guider la preuve automatique. Ce mécanisme prend la forme de deux nouveaux connecteurs logiques appelés *indicateurs de coupure*. L'idée générale est d'utiliser un connecteur, nommé `by`, pour attacher localement une assertion intermédiaire à la formule que l'on souhaite prouver. Autrement dit, le connecteur `by` indique que pour prouver une formule donnée, il faut d'abord prouver l'assertion intermédiaire attachée puis prendre cette assertion comme hypothèse supplémentaire. Nous introduisons également le connecteur dual `so` pour attacher un corollaire à une formule. Ces deux connecteurs permettent notamment de traduire des preuves données en langage naturel sans pour autant apporter de changement majeur à Why3, comme l'ajout d'un langage de preuve dédié.

Pour traiter ces indicateurs en pratique, nous avons implémenté dans l'outil Why3 un algorithme d'élimination. Celui-ci transforme une formule en un ensemble de buts correspondant aux différentes étapes de preuves induites par les indicateurs, qui peuvent ensuite être envoyés à divers démonstrateurs externes. Nous avons intégré cet algorithme au sein de la transformation `split_goal_wp`, qui découpe également les buts conjonctifs en un ensemble de sous-buts. Notons que cette transformation offrait déjà auparavant une légère aide à la preuve en exploitant les connecteurs asymétriques de Why3. En effet, le connecteur de conjonction asymétrique `A && B` est traité comme un raccourci pour `A` $\wedge$ `(A` $\rightarrow$ `B)`, tandis que le connecteur asymétrique `A || B` est traité comme `A` $\vee$ `((not A)` $\wedge$ `B)`.

Dans cette section, nous démontrons le fonctionnement de ces nouveaux connecteurs par le biais d'un exemple avant de les formaliser par des règles de raisonnement. Enfin, nous présentons également l'algorithme de traitement que nous avons implémenté dans l'outil Why3 pour éliminer les indicateurs.

2.3.1 Exemple d'utilisation de `by` et `so`

Nous montrons à l'aide d'un exemple comment traduire des preuves données en langage naturel en utilisant les connecteurs `by` et `so`. Cette traduction est basée sur l'équivalence intuitive de ces deux connecteurs avec les conjonctions de coordination "car" et "donc" en langage naturel. Nous employons donc le connecteur `by` pour exprimer la justification, tout comme "car" en langage naturel. De manière similaire, nous traduisons la notion de conséquence exprimée par "donc" en utilisant le connecteur `so`.

Nous prenons comme exemple la preuve de l'irrationalité de la racine carrée de 2, exprimée de la manière suivante : pour tout rationnel $\frac{p}{q}$, $(\frac{p}{q})^2 \neq 2$. Autrement dit, pour toute paire d'entiers p, q avec $q \neq 0$, $p^2 \neq 2q^2$. Cette proposition est représentée en Why3 comme suit :

```
lemma racine_2_irrationelle : ∀p q. q ≠ 0 → p * p ≠ 2 * q * q
```

Rappelons tout d'abord une manière standard de prouver cette proposition.
— Soit p, q deux entiers tels que $q \neq 0$. On cherche à montrer que $p^2 \neq 2q^2$.
— Par contradiction, on suppose $p^2 = 2q^2$.
— Soit d le plus grand diviseur commun de p et q.
— Il existe donc a et b tels que $da = p$ et $db = q$.
— Donc a, b sont premiers entre eux.
— En divisant par d^2 dans l'équation initiale, $a^2 = 2b^2$.
— Donc a est pair
— Donc il existe c tel que $a = 2c$.
— Donc $b^2 = 2c^2$.
— Donc b est également pair, ce qui contredit a, b premiers entre eux. □

En utilisant les principes mentionnés au début de cette section, cette preuve se traduit comme suit :

```
lemma racine_2_irrationelle : ∀p q. q ≠ 0 → p * p ≠ 2 * q * q
  by (p * p = 2 * q * q → false          (* Par contradiction *)
    by let d = gcd p q in                (* Soit d ... *)
      ∃a b. p = d * a ∧ q = d * b        (* Il existe a,b ... *)
        so gcd a b = 1                   (* a, b sont ... *)
        so a * a = 2 * b * b             (* a² = 2b² *)
        so divides 2 a                   (* a pair *)
        so ∃c. a = 2 * c                 (* Il existe c ... *)
        so b * b = 2 * c * c             (* b² = 2c² *)
        so divides 2 b)                  (* 2 divise b *)
```

Syntaxiquement, il faut lire cette formule en considérant que les connecteurs `by` et `so` sont associatifs à droite, et plus prioritaires que les constructions introduisant des variables (ici ∃ et `let`).

Nous retrouvons bien la même structure dans les énoncés des deux preuves. En effet, si l'on lit la formule en remplaçant `by` par le mot "car" et `so` par le mot "donc", on retrouve pratiquement le texte initial. Cependant, la preuve présentée jusqu'ici est insuffisamment détaillée, au sens que les démonstrateurs automatiques utilisés par Why3 n'arrivent pas à justifier certaines étapes de raisonnement, notamment celles qui impliquent de l'arithmétique non linéaire. Il faut donc ajouter des sous-preuves pour ces étapes :

— Justifier que $\gcd(a, b) = 1$. Cela vient du fait que

$$d = \gcd(da, db) = d \times \gcd(a, b),$$

comme de plus $d \neq 0$ nous pouvons simplifier.

— Justifier que $a^2 = 2b^2$. Cette équation est valide car $d^2(a^2 - 2b^2) = p^2 - 2q^2 = 0$ et $d \neq 0$

— Justifier que pour n'importe quel a, b tels que $a^2 = 2b^2$, 2 divise a. Dans le cas contraire, a est impair de la forme $2k + 1$, d'où une contradiction immédiate en développant a dans l'équation $a^2 = 2b^2$. De manière plus concise, c'est vrai car a ne peut pas être de la forme $2k + 1$.

L'ajout de ces sous-preuves peut se traduire par :

```
lemma racine_2_irrationelle : ∀p q. q ≠ 0 → p * p ≠ 2 * q * q
  by (p * p = 2 * q * q → false
    by let d = gcd p q in
      ∃a b. p = d * a ∧ q = d * b
    so (gcd a b = 1 by d * gcd a b = d
                    so d * (gcd a b - 1) = 0 ∧ d ≠ 0)
    so (a * a = 2 * b * b by (d*d) * (a*a-2*b*b) = 0 ∧ d ≠ 0)
    so (divides 2 a by not ∃k. a = 2*k+1)
    so ∃c. a = 2 * c
    so b * b = 2 * c * c
    so (divides 2 b by not ∃k. b = 2*k+1))
```

Exprimée ainsi, nous pouvons vérifier la preuve en appliquant la transformation `split_goal_wp`. La transformation produit 16 obligations de preuves, correspondant précisément aux étapes de raisonnement attendues. Nous utilisons ensuite une combinaison de démonstrateurs automatiques pour prouver chacun des buts, en l'occurrence Alt-Ergo [13], CVC4 [10] et E [77].

Notons que cet exemple court aurait pu être prouvé sans les indicateurs de coupure, en énonçant une cascade de lemmes intermédiaires. Cependant, ces lemmes intermédiaires seraient restés visibles par la suite, ce qui conduit à la saturation des démonstrateurs pour des exemples plus larges.

2.3.2 Interprétation et règles de raisonnement

Les connecteurs `by` et `so` diffèrent des connecteurs logiques habituels en ce qu'ils ne construisent pas véritablement de nouvelles formules. Nous définissons en effet l'interprétation des formules A `by` B et A `so` B comme étant la même que A. Nous avons fait ce choix car utiliser le connecteur `by` pour introduire la preuve d'une formule ne doit pas changer le sens de la formule en question, et il en va de même

pour `so` par dualité. Autrement dit, ces deux connecteurs sont des indicateurs qui servent uniquement à attacher syntaxiquement une formule à une autre, et nous pouvons les effacer sans changer le sens de la formule.

Par contre, la présence d'un indicateur de coupure altère l'utilisation de la formule dans une preuve. Nous souhaitons que la preuve d'une formule $A \,\texttt{by}\, B$ se fasse en prouvant d'abord B, puis ensuite A avec B dans le contexte. La formule B sert donc d'assertion intermédiaire dans la preuve de A. De manière duale, la preuve d'une formule avec $A \,\texttt{so}\, B$ parmi ses hypothèses doit commencer par dériver B comme hypothèse supplémentaire à partir de A. Nous disons donc que B sert d'hypothèse compagnon à A. Ces deux comportements correspondent tous les deux à l'introduction d'une *coupure* sur la formule B dans le raisonnement. En nous plaçant dans le calcul des séquents **LK** [46], cela revient à indiquer l'utilisation d'instances spécifiques de la règle de coupure :

$$\textsc{Cut}\frac{\Gamma \vdash B, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma \vdash \Delta}$$

Nous formalisons donc le comportement de ces nouveaux connecteurs par l'ajout des règles suivantes :

$$\textsc{By-r}\frac{\Gamma \vdash B, A, \Delta \qquad \Gamma, B \vdash A, \Delta}{\Gamma \vdash A \,\texttt{by}\, B, \Delta} \qquad \textsc{By-l}\frac{\Gamma, A \vdash \Delta}{\Gamma, A \,\texttt{by}\, B \vdash \Delta}$$

$$\textsc{So-l}\frac{\Gamma, A \vdash B, \Delta \qquad \Gamma, A, B \vdash \Delta}{\Gamma, A \,\texttt{so}\, B \vdash \Delta} \qquad \textsc{So-r}\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \,\texttt{so}\, B, \Delta}$$

Les règles By-r et So-l sont les instances attendues de la règle de coupure, tandis que les autres règles correspondent à l'interprétation triviale des connecteurs. Pour se convaincre de la correction de ces règles, il est suffisant de montrer que si un séquent est prouvable alors sa version "effacée", où l'on a éliminé les indicateurs de coupure, l'est également. C'est immédiat car l'effacement rend toutes les règles supplémentaires admissibles dans **LK**. Remarquons par ailleurs qu'en employant la règle d'affaiblissement, nous pouvons également montrer l'équiprouvabilité du séquent et de sa version effacée.

Notons cependant que la règle By-r ne correspond pas tout à fait à l'intuition donnée pour le connecteur `by`. La règle correspondant à l'intuition est la suivante, dérivée immédiatement par affaiblissement de By-r :

$$\textsc{By-r'}\frac{\Gamma \vdash B, \Delta \qquad \Gamma, B \vdash A, \Delta}{\Gamma \vdash A \,\texttt{by}\, B, \Delta}$$

La formule supplémentaire A dans la règle By-r permet de conserver la possibilité de prouver directement A au lieu de l'assertion intermédiaire B, ce qui est nécessaire pour obtenir l'équiprouvabilité entre les formules $A \,\texttt{by}\, B$ et A. Nous pouvons également voir la règle By-r comme une preuve par contradiction puisque c'est équivalent à supposer $\neg A$ pour prouver B. Nous pourrions de manière similaire définir une règle duale So-l' qui correspondrait à l'oubli de l'hypothèse A dans la prémisse $\Gamma, A, B \vdash \Delta$ de la règle So-l, mais elle n'a pas d'intérêt pratique.

Remarquons que toute preuve en calcul des séquents **LK** standard peut être complètement encodée dans les formules via le connecteur `by`. Nous définissons cet encodage des preuves par :

$$T\left(\frac{(\pi_i)_{i\in I}}{\Gamma \vdash \Delta}\right) = \left(\forall(x)_{x\in fv(\Gamma\cup\Delta)}.\bigwedge\Gamma \to \bigvee\Delta\right)\,\mathtt{by}\bigwedge_{i\in I} T(\pi_i)$$

De par l'interprétation donnée à `by`, il est immédiat que la formule résultante est équivalente à la conclusion de la preuve encodée. De plus, l'application répétée de la règle By-R' et de la règle de conjonction à droite sur le séquent $\vdash T(\pi)$ construit un arbre de preuve dont les feuilles correspondent précisément aux nœuds de π. En particulier, il est possible d'utiliser les indicateurs de coupure pour donner toute la structure d'une preuve si nécessaire.

2.3.3 Élimination des indicateurs : algorithme théorique

Nous présentons maintenant un algorithme de traitement des indicateurs de coupure. L'objectif est d'éliminer les connecteurs `by` et `so` d'une formule arbitraire sans perdre l'information donnée par les indicateurs. Il faut bien évidemment que la formule initiale soit une conséquence logique de la formule obtenue par l'algorithme d'élimination, de façon à pouvoir réduire la preuve de la formule initiale à celle après élimination. L'implication réciproque n'est pas nécessaire, et ne sera pas vraie en général.

Pour représenter fidèlement les informations données par les indicateurs de coupure, la formule obtenue doit regrouper de manière conjonctive les feuilles de l'arbre de preuve obtenu par l'application des règles de **LK** étendu. Par exemple, pour la formule $A\,\mathtt{by}(B\,\mathtt{so}\,C)$ on souhaite obtenir une formule isomorphe à $B \wedge (B \to C) \wedge (B \wedge C \to A)$, qui correspond à la structure de preuve suivante :

$$\dfrac{\dfrac{\dfrac{\vdash B}{\vdash B\,\mathtt{so}\,C}}{\vdash B\,\mathtt{so}\,C, A} \qquad \dfrac{\dfrac{B \vdash C}{B \vdash C, A} \qquad B, C \vdash A}{B\,\mathtt{so}\,C \vdash A}}{\vdash A\,\mathtt{by}(B\,\mathtt{so}\,C)}$$

Remarquons que l'arbre de preuve donné ci-dessus contient deux affaiblissements sur la partie droite, dont l'un correspondant à la règle dérivée By-R'. Nous avons fait le choix d'ajouter ces affaiblissements pour que les formules obtenues par notre algorithme soient plus naturelles. En effet, laisser plusieurs formules à droite du séquent reviendrait à systématiquement donner la possibilité de faire une preuve par contradiction sur la formule que l'on cherche à prouver. Nous pouvons par ailleurs nous permettre de faire de tels affaiblissements car nous voulons seulement que la formule finale implique la formule de départ. Dans le cas particulier où l'on souhaite faire une preuve par contradiction, nous avons toujours la possibilité de le faire explicitement via la construction $\neg A \to (\bot\,\mathtt{by}\,B)$.

Les formules traitées par l'algorithme d'élimination sont décrites par la syntaxe suivante :

$$\begin{array}{rcl}
\varphi & ::= & A \\
& \mid & \top \mid \bot \\
& \mid & \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \to \varphi \\
& \mid & \neg\varphi \\
& \mid & \varphi\,\mathtt{by}\,\varphi \mid \varphi\,\mathtt{so}\,\varphi \\
& \mid & \forall x.\varphi \mid \exists x.\varphi
\end{array}$$

$$
\begin{array}{lcll}
e^s(\varphi) & = & e^s(\varphi) & (\varphi, s) \in \{A, \top, \bot\} \times \{-, +\} \\
e^s(\varphi_1 \,\mathtt{op}\, \varphi_2) & = & e^s(\varphi_1) \,\mathtt{op}\, e^s(\varphi_2) & (\mathtt{op}, s) \in \{\wedge, \vee\} \times \{-, +\} \\
e^s(Qx.\varphi) & = & Qx.e^s(\varphi) & (Q, s) \in \{\forall, \exists\} \times \{-, +\} \\
e^+(\neg\varphi) & = & \neg e^-(\varphi) & \\
e^-(\neg\varphi) & = & \neg e^+(\varphi) & \\
e^+(\varphi_1 \to \varphi_2) & = & e^-(\varphi_1) \to e^+(\varphi_2) & \\
e^-(\varphi_1 \to \varphi_2) & = & e^+(\varphi_1) \to e^-(\varphi_2) & \\
e^+(\varphi_1 \,\mathtt{by}\, \varphi_2) & = & e^+(\varphi_2) & \\
e^-(\varphi_1 \,\mathtt{by}\, \varphi_2) & = & e^-(\varphi_1) & \\
e^+(\varphi_1 \,\mathtt{so}\, \varphi_2) & = & e^+(\varphi_1) & \\
e^-(\varphi_1 \,\mathtt{so}\, \varphi_2) & = & e^-(\varphi_1) \wedge e^-(\varphi_2) &
\end{array}
$$

$$
\begin{array}{lcll}
c^s(\varphi) & = & \top & (\varphi, s) \in \{A, \top, \bot\} \times \{-, +\} \\
c^s(\varphi_1 \,\mathtt{op}\, \varphi_2) & = & c^s(\varphi_1) \wedge c^s(\varphi_2) & (\mathtt{op}, s) \in \{\wedge, \vee\} \times \{-, +\} \\
c^s(Qx.\varphi) & = & \forall x.c^s(\varphi) & (Q, s) \in \{\forall, \exists\} \times \{-, +\} \\
c^+(\neg\varphi) & = & c^-(\varphi) & \\
c^-(\neg\varphi) & = & c^+(\varphi) & \\
c^+(\varphi_1 \to \varphi_2) & = & c^-(\varphi_1) \wedge (e^-(\varphi_1) \to c^+(\varphi_2)) & \\
c^-(\varphi_1 \to \varphi_2) & = & c^+(\varphi_1) \wedge c^-(\varphi_1) \wedge (e^-(\varphi_1) \to c^-(\varphi_2)) & \\
c^+(\varphi_1 \,\mathtt{by}\, \varphi_2) & = & c^+(\varphi_2) \wedge c^-(\varphi_2) \wedge (e^-(\varphi_2) \to e^+(\varphi_1) \wedge c^+(\varphi_1)) & \\
c^-(\varphi_1 \,\mathtt{by}\, \varphi_2) & = & c^-(\varphi_1) & \\
c^+(\varphi_1 \,\mathtt{so}\, \varphi_2) & = & c^+(\varphi_1) & \\
c^-(\varphi_1 \,\mathtt{so}\, \varphi_2) & = & c^-(\varphi_1) \wedge (e^-(\varphi_1) \to e^+(\varphi_2) \wedge c^+(\varphi_2) \wedge c^-(\varphi_2)) &
\end{array}
$$

FIGURE 2.6 – Définition des fonctions d'éliminations des indicateurs de coupure

L'idée intuitive de l'algorithme d'élimination est de collecter les différentes étapes de preuves induites par les indicateurs de coupures comme des conditions auxiliaires, puis de construire une version de la formule principale qui est logiquement plus faible lorsque ces conditions sont vraies. Comme la polarité de la formule peut inverser le sens dans lequel le traitement doit être effectué, cela nous conduit naturellement à quatre fonctions auxiliaires mutuellement récursives e^+, e^-, c^+, c^-, qui doivent respecter les propriétés ci-dessous :

(1) si $c^+(\varphi)$ est vérifiée, alors $e^+(\varphi)$ implique φ.
(2) si $c^-(\varphi)$ est vérifiée, alors φ implique $e^-(\varphi)$.

Nous donnons les définitions effectives de ces quatre fonctions dans la figure 2.6. L'algorithme final consiste à calculer $e^+(\varphi) \wedge c^+(\varphi)$. L'idée intuitive est la suivante : e^+ remplace les sous-formules $A \,\mathtt{by}\, B$ qui apparaissent en position positive par B. Cela revient à remplacer le but A par l'assertion intermédiaire B. De manière duale, e^- remplace $A \,\mathtt{so}\, B$ en position négative par $A \wedge B$, ajoutant ainsi l'hypothèse compagnon B au contexte. Les deux fonctions c^+ et c^- collectent les conditions sous lesquelles ces remplacements sont possibles, respectivement $B \to A$ et $A \to B$ dans les cas ci-dessus. On appelle ces conditions "conditions de bord" dans ce qui suit. Ces conditions correspondent précisément à la conjonction des étapes de raisonnement induites par les indicateurs de coupure dans **LK** étendu.

Nous pouvons aisément vérifier les propriétés de corrections (1) et (2) par récur-

rence sur la formule. Par exemple, considérons le traitement d'une formule existentielle $\exists x.\varphi$ en position négative. Nous devons donc démontrer qu'il existe un témoin x_1 pour $e^-(\varphi)$, sachant que $\forall x.c^-(\varphi)$ est vraie et qu'il existe un témoin x_0 pour φ. Par hypothèse de récurrence, comme $c^-(\varphi)$ est vraie pour $x = x_0$, x_0 est le témoin voulu. Remarquons qu'il est nécessaire de transformer la quantification existentielle en quantification universelle dans les conditions de bord pour pouvoir effectuer ce raisonnement. De manière générale, les conditions de bord s'accumulent de manière conjonctive.

Notons que l'algorithme que nous avons décrit traite les opérateurs binaires $\wedge$ et $\vee$ de manière différente de l'implication pour les conditions de bord. En effet, l'implication préserve $e^-(\varphi_1)$ en tant que contexte quand c'est possible pour les conditions de bord, tandis que les autres opérateurs ne préservent pas de contexte. Nous avons fait ce choix pour préserver la nature symétrique de $\wedge$ et $\vee$. Cependant, si l'on considère que $\wedge$ et $\vee$ sont asymétriques alors d'autres définitions sont possibles, comme par exemple :

$$\begin{array}{lcl} c^+(\varphi_1 \wedge \varphi_2) & = & c^+(\varphi_1) \wedge c^-(\varphi_1) \wedge (e^-(\varphi_1) \to c^+(\varphi_2)) \\ c^-(\varphi_1 \wedge \varphi_2) & = & c^-(\varphi_1) \wedge (e^-(\varphi_1) \to c^-(\varphi_2)) \\ c^+(\varphi_1 \vee \varphi_2) & = & c^+(\varphi_1) \wedge (e^+(\varphi_1) \vee c^+(\varphi_2)) \\ c^-(\varphi_1 \vee \varphi_2) & = & c^-(\varphi_1) \wedge c^+(\varphi_1) \wedge (e^+(\varphi_1) \vee c^-(\varphi_2)) \end{array}$$

Dans ce contexte, le traitement de l'implication n'est alors qu'un cas particulier de traitement asymétrique de la disjonction. Remarquons qu'il n'est pas possible de préserver le contexte des deux côtés à la fois. Dans le cas de la conjonction, cela correspond au fait qu'il faut prouver l'une des deux formules en premier pour pouvoir l'utiliser comme contexte dans l'autre.

Une autre variation possible dans cet algorithme est de systématiquement autoriser le raisonnement par l'absurde au niveau du connecteur `by`. Nous obtenons ce comportement en changeant la définition de e^+ en :

$$e^+(\varphi_1 \texttt{ by } \varphi_2) = e^+(\varphi_1) \vee e^+(\varphi_2)$$

2.3.4 Algorithme implémenté dans Why3

Pour traiter les indicateurs de coupure dans Why3, nous avons intégré un algorithme d'élimination des indicateurs de coupure au sein de la transformation `split_goal_wp`. Cette transformation est utilisée pour transformer un but à vérifier en une collection de sous-buts indépendants dont la conjonction implique le but initial. Elle consiste de manière effective à éliminer les indicateurs, puis à découper ensuite la structure conjonctive de la formule obtenue pour obtenir les différents sous-buts. L'algorithme utilisé pour éliminer les indicateurs diffère cependant légèrement de l'algorithme théorique pour des raisons de complexité, le calcul des conditions de bord pouvant sinon créer des formules de taille exponentielle.

Le phénomène se produit à cause de la duplication des conditions de bord dans le traitement des connecteurs binaires asymétriques, comme l'implication. En effet, pour une formule $\varphi_1 \to \varphi_2$ les conditions de bord négatives de φ_1 sont propagées à la fois dans les conditions de bord positives et négatives de $\varphi_1 \to \varphi_2$. Par ailleurs, comme le connecteur `by` donne un contexte différent aux conditions de bord négatives et positives, il n'est pas possible de traiter le problème simplement par regroupement

des branches conjonctives identiques. Une famille de formules $(\varphi_n)_{n\in\mathbb{N}}$ exhibant un comportement pathologique est définie par :

$$\varphi_n \triangleq \begin{cases} n = 0 & : \quad A \\ n = m+1 & : \quad (\varphi_m \to U_m)\,\texttt{by}\,T_m \end{cases}$$

où $(U_n, T_n)_{n\in\mathbb{N}}$ sont des atomes distincts. On peut montrer que le nombre de branches conjonctives dans les conditions de bord majore la suite de Fibonacci.

Pour pallier au problème, nous avons renforcé les conditions de bord du connecteur `by` de manière à toujours donner un contexte identique aux conditions de bord positives et négatives provenant d'une même formule. Nous remplaçons donc la définition de c^+ pour l'opérateur `by`

$$c^+(\varphi_1\,\texttt{by}\,\varphi_2) = c^+(\varphi_2) \wedge c^-(\varphi_2) \wedge (e^-(\varphi_2) \to e^+(\varphi_1) \wedge c^+(\varphi_1))$$

par la nouvelle définition suivante :

$$c^+(\varphi_1\,\texttt{by}\,\varphi_2) = c^+(\varphi_2) \wedge c^-(\varphi_2) \wedge c^+(\varphi_1) \wedge (e^-(\varphi_2) \to e^+(\varphi_1))$$

De cette façon, nous pouvons traiter les conditions de bord de manière uniforme. Nous calculons donc directement $c^{all}(\varphi) = c^+(\varphi) \wedge c^-(\varphi)$ en simplifiant les duplications quand elles apparaissent, ce qui rend la taille des formules générées polynomiale. En calculant simultanément c^{all}, e^+ et e^- on obtient un algorithme d'élimination en temps polynomial. Nous renforçons alors le résultat final de l'élimination en $c^{all}(\varphi) \wedge e^+(\varphi)$.

Pour terminer la description de l'algorithme, il nous reste à préciser si nous utilisons les règles de calcul symétriques ou asymétriques pour la conjonction et la disjonction. Comme Why3 supporte des variantes asymétriques de ces deux connecteurs, le choix se fait simplement selon leur nature.

Les restrictions présentent néanmoins deux défauts. Premièrement, comme on ne peut plus distinguer les conditions de bord positives et négatives, on est obligé de toutes les inclure au résultat final. Ce n'est cependant pas un problème en pratique car les formules sur lesquelles l'élimination est utilisée ont généralement des conditions de bord négatives triviales, ou déjà incluses dans les conditions de bord positives. Deuxièmement, comme on a restreint les conditions de bord générées pour le connecteur `by`, il y a des cas assez rares où des éléments de contexte désirables disparaissent. Le premier exemple pratique que nous ayons rencontré est une formule de la forme $((A\,\texttt{by}\,B) \vee (C\,\texttt{by}\,D))\,\texttt{by}\,E$, où l'hypothèse E n'est visible que pour prouver $B \vee D$.

2.4 Transformation de calcul

Dans cette section, nous décrivons un dernier outil que nous utilisons pour assister les démonstrateurs automatiques. Il s'agit d'une famille de transformations qui nous permet de faire une phase de précalcul avant d'envoyer les obligations de preuve aux démonstrateurs automatiques.

Plus précisément, la transformation `compute` applique un système de réécriture aux buts jusqu'à normalisation. Ces règles de réécriture contiennent par défaut la réduction des opérations primitives, comme les opérateurs entiers et booléens ainsi

que le filtrage sur les types algébriques. Elles comprennent aussi le déroulage des définitions de fonctions, y compris récursives. Ainsi, le but suivant :

```
function length (l:list 'a) : int = match l with
  | Nil → 0
  | Cons _ q → 1 + length q
  end
goal G : ∀x. x = 3 → length (Cons 1 (Cons 2 (Cons 3 Nil))) = x
```

est réécrit en `∀x. x = 3 → 3 = x`. Cette transformation permet également d'ajouter des propriétés comme règles de réécriture additionnelles, via une déclaration `meta`. Cela permet par exemple de résoudre complètement le but suivant via `compute`.

```
function fact int : int
axiom fact_rule : ∀n.
  fact n = if n ≤ 0 then 1 else n * fact (n-1)
meta "rewrite" prop fact_rule
goal G : fact 4 = 24 ∧ fact (fact 3) = 720
```

Le processus de normalisation peut a priori ne pas terminer. Pour contourner ce problème, la transformation de calcul est paramétrée par un nombre maximal d'étapes de réductions, via le meta `compute_max_steps`. De plus, les règles de réécriture ne sont pas appliquées dans les branches des constructions conditionnelles (`if` et `match`), pour éviter un déroulement illimité des fonctions récursives.

Telle quelle, la transformation `compute` est un mécanisme de simplification extrêmement agressif, qui déroule parfois trop de définitions. Cependant, un contrôle plus fin des règles de réécriture utilisées et des termes impliqués permet d'utiliser la transformation pour effectuer une phase de précalcul arbitraire avant l'envoi aux démonstrateurs, mettant ainsi certains buts à leur portée. De plus, nous pouvons implémenter l'équivalent de transformations auxiliaires directement dans les sources des fichiers à vérifier, voire une procédure de décision complète comme démontré dans la section 2.6. Contrairement à l'ajout de transformation ad-hoc au sein de l'outil lui-même, cette méthodologie ne peut pas introduire d'incohérence si les règles de réécriture sont démontrées.

Pour obtenir ce contrôle, nous avons introduit une nouvelle transformation `compute_specified`, qui ne déroule que les définitions enregistrés comme règles de réécriture via le meta `rewrite_def`. De plus, nous avons également pris en compte les propriétés marquées par l'étiquette `"rewrite"` comme règles de réécriture. Ce second mécanisme nous permet de générer localement des règles de réécriture via les post-conditions des procédures.

Nous avons utilisé cette transformation en combinaison avec le code fantôme pour développer une nouvelle méthodologie de preuve : équiper les données avec des annotations dont le seul but est d'aider les démonstrateurs, et dont la valeur est déterminée par calcul. Ces annotations représentent typiquement une abstraction des données sous-jacentes. Comme ces annotations sont fantômes, elles n'interviennent que pour la vérification.

Nous démontrons cette méthodologie sur un exemple simple. Nous étiquetons des entiers avec un majorant pour maintenir des bornes statiques sur les entiers du programme :

```
type t = { n : int; ghost m: int }
predicate c (t:t) = t.n ≤ t.m
let add (x y:t) : t
  requires { c x ∧ c y }
  ensures { c result }
  ensures { "rewrite" result.m = x.m + y.m }
= { n = x.n + y.n; m = x.m + y.m }
```

Les conditions de vérification d'une procédure appelant `add` contiennent naturellement ces égalités dans leurs hypothèses, ce qui permet d'obtenir la valeur des bornes par calcul. En particulier, si les bornes des valeurs initiales sont explicites, nous obtenons des bornes explicites sur les valeurs calculées via `add`. Faire dépendre les obligations de preuves de cette annotation permet donc de bénéficier de ce calcul pour la vérification. Dans notre exemple, nous pouvons vérifier un nouveau majorant directement par calcul, comme pour l'assertion suivante :

```
let f (x y:t)
  requires { c x ∧ c y }
  requires { "rewrite" x.m = 1 }
  requires { "rewrite" y.m = 1 }
= let z = add (add y x) (add x y) in
  assert { z.m ≤ 5 }
```

Plus précisément, nous établissons l'assertion en appliquant la séquence de transformation suivante. Nous commençons par appliquer la transformation `split_goal_wp` à la condition de vérification de `f` pour isoler les différentes obligations de preuves. Nous appliquons ensuite la transformation `introduce_premises` au but représentant l'assertion. Cette transformation déplace les parties gauche d'implication dans le contexte global. Nous devons l'appliquer pour que les hypothèses marquées `"rewrite"` puissent être détectées comme règles de réécriture. Enfin, nous appliquons la transformation de calcul, laquelle réduit le but à `true` et donc l'élimine.

Bien entendu, il ne s'agit ici que de démontrer l'esprit de cette méthodologie. Les démonstrateurs automatiques arrivent parfaitement à vérifier ce type de propriété sans aide extérieure. Nous donnons dans les sections suivantes deux exemples où cette méthode d'étiquetage fantôme combiné avec le calcul facilite considérablement la vérification.

2.5 Application : preuve d'un petit compilateur

Dans cette section, nous présentons une expérience visant à certifier un petit compilateur de manière la plus automatique possible. Avec d'un côté les progrès récents dans la preuve formelle des compilateurs, et de l'autre côté l'essor des démonstrateurs automatiques, il semble qu'il s'agisse d'une question d'actualité. À l'heure actuelle, l'exemple le plus impressionnant de compilateur certifié est Compcert [60], développé avec l'assistant de preuve Coq. Majoritairement manuelle, cette preuve a également nécessité un effort considérable. Il est donc naturel de chercher à augmenter le degré d'automatisation d'une telle preuve.

À cette fin, nous avons choisi un exemple jouet, présenté par Xavier Leroy lors

d'un cours à l'OPLSS (Oregon Programming Languages Summer School) [1] sur la vérification mécanisée des compilateurs, qui effectue la traduction d'un langage impératif simple vers une machine virtuelle. Bien que relativement modeste, ce choix suffit à illustrer la complexité des problèmes rencontrés. Un autre intérêt est que ce compilateur est accompagné d'une preuve Coq qui nous fournit un élément de comparaison.

Pour augmenter le degré d'automatisation, la solution qui vient immédiatement à l'esprit est d'employer les démonstrateurs automatiques. Le cadre de Why3 nous a donc semblé adapté à cette expérience. Nous avons d'abord tenté une approche directe, dans l'esprit de la preuve Coq. Cependant, elle s'est montrée peu satisfaisante : les démonstrateurs automatiques s'avèrent incapables d'effectuer les preuves sans aide manuelle conséquente.

Nous avons alors considéré la preuve d'un compilateur comme la preuve, une fois pour toutes, d'une spécification pour le code généré par le compilateur. La spécification en question est de se comporter comme le code source. Cette vision nous permet d'employer les méthodes de preuve de programme au niveau du code généré. Plus concrètement, nous avons défini un calcul de plus faible pré-condition pour le langage cible du compilateur. Cette approche simplifie considérablement les obligations de preuves générées pour le compilateur puisque celles-ci correspondent alors à des obligations de preuves pour le code généré, similaires à celles générées par Why3 lui-même. Comme ce nouveau calcul de plus faible pré-condition a lieu à un autre niveau que celui de Why3, nous nous reposons de manière critique sur la transformation de calcul pour rendre ce calcul effectif. Le développement complet fait partie de la galerie d'exemples de Why3 [2].

2.5.1 Langage source

Le langage source est un langage impératif simple, communément appelé Imp. Chaque programme de Imp est composé d'une séquence de commandes qui servent à manipuler l'état global de l'exécution à travers des variables globales. On définit l'état et les variables avec les types suivants :

```
type id = Id int          (* les noms des variables *)
type state = map id int   (* l'état global *)
```

Les variables, dénotées par des entiers, correspondent toujours à des valeurs entières. Outre les variables, à l'intérieur d'une commande on manipule des expressions arithmétiques ou booléennes, dont les définitions sont données dans la figure 2.7. Les commandes sont constituées de la commande vide, de l'affectation, de la séquence, de la conditionnelle, et de la commande de boucle. La représentation en Why3 des commandes du langage Imp est donnée dans la figure 2.8.

Comme exemple de programme Imp, voici comment on peut écrire la factorielle :

```
X := 1; WHILE ¬(Y ≤ 0) DO X := X * Y; Y := Y - 1 DONE
```

programme qui est représenté en Why3 par la valeur suivante :

1. http://pauillac.inria.fr/~xleroy/courses/Eugene-2011/
2. http://toccata.lri.fr/gallery/double_wp.fr.html

```
type aexpr =
  | Anum int          (*  n ∈ ℤ  *)
  | Avar id           (*    X    *)
  | Aadd aexpr aexpr  (* a1 + a2 *)
  | Asub aexpr aexpr  (* a1 - a2 *)
  | Amul aexpr aexpr  (* a1 * a2 *)

type bexpr =
  | Btrue             (*   vrai   *)
  | Bfalse            (*   faux   *)
  | Band bexpr bexpr  (* b1 ∧ b2 *)
  | Bnot bexpr        (*   ¬b     *)
  | Beq aexpr aexpr   (* a1 = a2 *)
  | Ble aexpr aexpr   (* a1 ≤ a2 *)
```

FIGURE 2.7 – Expressions du langage Imp

```
type com =
  | Cskip               (*          SKIP           *)
  | Cassign id aexpr    (*        X := a           *)
  | Cseq com com        (*        c1 ; c2          *)
  | Cif bexpr com com   (*  IF b THEN c1 ELSE c2  *)
  | Cwhile bexpr com    (*   WHILE b DO c DONE     *)
```

FIGURE 2.8 – Commandes du programmes Imp

```
Cseq (Cassign (Id 1, Anum 1),
      Cwhile (Bnot Ble(Var(Id 2), Anum 0),
              Cseq (Cassign (Id 1, Amul (Var(Id 1), Var(Id 2))),
                    Cassign (Id 2, Asub (Var(Id 2), Anum 1)))))
```

La sémantique du langage Imp est une sémantique opérationnelle standard à grands pas. Nous la formalisons en Why3 de la même façon que dans le développement Coq qui nous sert de référence, à savoir par des fonctions d'évaluation `aeval` et `beval` pour les expressions, respectivement arithmétiques et booléennes, et un prédicat inductif `ceval` pour les commandes. Les définitions sont données dans la figure 2.9.

Enfin, remarquons que cette sémantique est déterministe. La transformation d'induction (cf. section 2.2) nous permet de réduire cette propriété vers des conditions faciles à vérifier pour les démonstrateurs automatiques.

```
lemma ceval_deterministic :
  ∀c mi mf1 mf2. ceval mi c mf1 →  ceval mi c mf2 → mf1 = mf2
```

```
function aeval (st: state) (e: aexpr) : int = match e with
  | Anum n      → n
  | Avar x      → st[x]
  | Aadd e1 e2  → aeval st e1 + aeval st e2
  | Asub e1 e2  → aeval st e1 - aeval st e2
  | Amul e1 e2  → aeval st e1 * aeval st e2
  end

function beval (st:state) (b:bexpr) : bool = match b with
  | Btrue      → true
  | Bfalse     → false
  | Bnot b'    → notb (beval st b')
  | Band b1 b2 → andb (beval st b1) (beval st b2)
  | Beq a1 a2  → aeval st a1 = aeval st a2
  | Ble a1 a2  → aeval st a1 ≤ aeval st a2
  end

inductive ceval (mi: state) (cmd: com) (mf: state) =
  | E_Skip : ∀m. ceval m Cskip m
  | E_Ass  : ∀m a x. ceval m (Cassign x a) m[x ← aeval m a]
  | E_Seq  : ∀cmd1 cmd2 m0 m1 m2.
      ceval m0 cmd1 m1 → ceval m1 cmd2 m2 →
      ceval m0 (Cseq cmd1 cmd2) m2
  | E_IfTrue : ∀m0 m1 cond cmd1 cmd2. beval m0 cond →
      ceval m0 cmd1 m1 → ceval m0 (Cif cond cmd1 cmd2) m1
  | E_IfFalse : ∀m0 m1 cond cmd1 cmd2. not beval m0 cond →
      ceval m0 cmd2 m1 → ceval m0 (Cif cond cmd1 cmd2) m1
  | E_WhileEnd : ∀cond m body. not beval m cond →
      ceval m (Cwhile cond body) m
  | E_WhileLoop : ∀mi mj mf cond body. beval mi cond →
      ceval mi body mj → ceval mj (Cwhile cond body) mf →
      ceval mi (Cwhile cond body) mf
```

FIGURE 2.9 – Sémantique à grands pas de Imp

```
type ofs   = int
type instr =
  | Iconst int   (* empile un entier                          *)
  | Ivar id      (* empile une variable                       *)
  | Isetvar id   (* dépile n, assigne la variable à n         *)
  | Ibranch ofs  (* saute le nombre d'instructions donné      *)
  | Iadd         (* dépile deux valeurs, empile leur somme    *)
  | Isub         (* dépile n2 puis n1, empile n1 - n2         *)
  | Imul         (* dépile deux valeurs, empile leur produit  *)
  | Ibeq ofs     (* dépile n2 puis n1, Ibranch ssi n1 = n2    *)
  | Ibne ofs     (* dépile n2 puis n1, Ibranch ssi n1 <> n2   *)
  | Ible ofs     (* dépile n2 puis n1, Ibranch ssi n1 <= n2   *)
  | Ibgt ofs     (* dépile n2 puis n1, Ibranch ssi n1 > n2    *)
  | Ihalt        (* arrêt de la machine                       *)

type code = list instr
```

FIGURE 2.10 – Instructions de la machine virtuelle

2.5.2 Langage cible

Le langage cible est un langage bas niveau interprété par une machine virtuelle. Chaque programme est composé d'une séquence d'instructions. La machine virtuelle est équipée d'une pile et d'une mémoire :

```
type pos = int                    (* position de la tête de lecture *)
type stack = list int                              (* pile d'entiers *)
type machine_state = VMS pos stack state                    (* état  *)
```

À chaque étape, la machine procède en exécutant l'instruction qui se trouve au niveau de la tête de lecture, puis avance celle-ci d'un cran. Le jeu d'instructions est décrit en Why3 par le type `instr`, dont nous donnons la définition dans la figure 2.10.

Remarque 2.1. Le jeu d'instructions que nous présentons est légèrement différent de celui utilisé dans le développement Coq. Comme nous utilisons des entiers relatifs, nous avons fusionné les instructions de saut avant/arrière en une seule (`Ibranch`).

Voici l'exemple de la factorielle réécrit en langage cible :

```
(*  X = Id 0, Y = Id 1 *)
[ Iconst 1; Isetvar X; Ivar Y   ; Iconst 0     ; Ible 9;
  Ivar X  ; Ivar Y   ; Imul     ; Isetvar X    ; Ivar Y;
  Iconst 0; Isub     ; Isetvar Y; Ibranch (-12); Ihalt ]
```

Un autre exemple de programme est la construction la plus courte d'une boucle infinie, par la séquence constituée de l'unique instruction `Ibranch (-1)`.

La sémantique de la machine est une sémantique opérationnelle à petits pas ce que nous formalisons en Why3 par un prédicat inductif `transition`, dont la définition est donnée dans la figure 2.11. Celui-ci représente le changement d'état lors d'une étape de calcul de la machine virtuelle. Nous obtenons alors la représenta-

tion de l'existence d'une séquence arbitraire de transition par sa clôture transitive `transition_star`.

2.5.3 Structure du compilateur

Le compilateur que nous cherchons à vérifier est composé de trois fonctions de programme récursives qui compilent respectivement les trois catégories syntaxiques du langage Imp :

```
let rec compile_aexpr (a: aexpr) : code = ...
let rec compile_bexpr (b: bexpr) (cond: bool) (ofs: ofs) : code = ...
let rec compile_com   (cmd: com) : code = ...
```

Le schéma de compilation d'une expression arithmétique consiste à mettre sa valeur au sommet de la pile. Voici par exemple comment on compile `Asub a1 a2` :

```
  (* ++ représente la concaténation *)
  compile_aexpr a1 ++ compile_aexpr a2 ++ Cons Isub Nil
```

Les expressions booléennes sont elles traduites vers des branchements. Si une telle expression s'évalue à `cond`, l'exécution du code compilé effectue un saut de `ofs`. Sinon, il n'effectue pas de saut.

Enfin, les commandes sont traduites par une séquence d'instructions qui change l'état de la même manière. La fonction principale consiste alors simplement à ajouter une instruction d'arrêt pour finir l'exécution.

```
let compile_program (prog : com) : code =
  compile_com prog ++ Cons Ihalt Nil
```

2.5.4 Énoncé de correction du compilateur

Intuitivement, un compilateur est correct si, que l'on exécute le programme de départ P ou le programme compilé $\widehat{P}$, on obtient le même résultat. Concrètement, nous allons montrer un résultat de *simulation en avant*, c'est-à-dire que si P termine dans un état donné, l'exécution de $\widehat{P}$ avec les mêmes conditions initiales termine dans le même état. Nous exprimons cela en donnant à la fonction principale `compile_program` la post-condition suivante :

```
let compile_program (prog: com) : code
  ensures { ∀mi mf.
    ceval mi prog mf → vm_terminates result mi mf }
```

Ici, `vm_terminates` exprime la terminaison de la machine, c'est-à-dire que la tête de lecture est positionnée sur une instruction `Ihalt`, et que la pile, utilisée pour les calculs intermédiaires, est laissée vide.

```
(* Les instructions en minuscules (iconst n,ivar x,...)
   sont des abbréviations pour la liste singleton formée
   de cette instruction (Cons (Iconst n) Nil,Cons (Ivar x) Nil ,...) *)
inductive transition (c: code) (msi msj: machine_state) =
  | trans_const : ∀c p n. codeseq_at c p (iconst n) →
      ∀s m. transition c (VMS p s m) (VMS (p + 1) (Cons n s) m)
  | trans_var : ∀c p x. codeseq_at c p (ivar x) →
      ∀s m. transition c (VMS p s m) (VMS (p + 1) (Cons m[x] s) m)
  | trans_set_var: ∀c p x. codeseq_at c p (isetvar x) →
      ∀n s m. transition c (VMS p (Cons n s) m) (VMS (p + 1) s m[x←n])
  | trans_add : ∀c p. codeseq_at c p iadd →
      ∀n1 n2 s m. transition c (VMS p (Cons n2 (Cons n1 s)) m)
                               (VMS (p + 1) (Cons (n1 + n2) s) m)
  | trans_sub : ∀c p. codeseq_at c p isub →
      ∀n1 n2 s m. transition c (VMS p (Cons n2 (Cons n1 s)) m)
                               (VMS (p + 1) (Cons (n1 - n2) s) m)
  | trans_mul : ∀c p. codeseq_at c p imul →
      ∀n1 n2 s m. transition c (VMS p (Cons n2 (Cons n1 s)) m)
                                 (VMS (p + 1) (Cons (n1 * n2) s) m)
  | trans_beq: ∀c p1 ofs. codeseq_at c p1 (ibeq ofs) →
      ∀s m n1 n2. transition c (VMS p1 (Cons n2 (Cons n1 s)) m)
                  (VMS (if n1 = n2 then p1 + 1 + ofs else p1 + 1) s m)
  | trans_bne: ∀c p1 ofs. codeseq_at c p1 (ibne ofs) →
      ∀s m n1 n2. transition c (VMS p1 (Cons n2 (Cons n1 s)) m)
                   (VMS (if n1 = n2 then p1 + 1 else p1 + 1 + ofs) s  m)
  | trans_ble: ∀c p1 ofs. codeseq_at c p1 (ible ofs) →
      ∀s m n1 n2. transition c (VMS p1 (Cons n2 (Cons n1 s)) m)
                  (VMS (if n1 ≤ n2 then p1 + 1 + ofs else p1 + 1)  s m)
  | trans_bgt: ∀c p1 ofs. codeseq_at c p1 (ibgt ofs) →
      ∀s m n1 n2. transition c (VMS p1 (Cons n2 (Cons n1 s)) m)
                  (VMS (if n1 ≤ n2 then p1 + 1 else p1 + 1 + ofs) s  m)
  | trans_branch: ∀c p ofs. codeseq_at c p (ibranch ofs) →
      ∀s m. transition c (VMS p s m) (VMS (p + 1 + ofs) s m)
```

FIGURE 2.11 – Sémantique à petit pas de la machine virtuelle

```
(* le code c se trouve à la position p dans le code c_glob *)
inductive codeseq_at (c_glob: code) (p: pos) (c: code) =
  | codeseq_at_intro :
    ∀c1 c2 c3. codeseq_at (c1 ++ c2 ++ c3) (length c1) c2

predicate vm_terminates (c: code) (mi mf: state) =
  ∃p. codeseq_at c p (Cons Ihalt Nil) ∧
    transition_star c (VMS 0 Nil mi) (VMS p Nil mf)
```

Remarquons au passage que nous nous intéressons ici uniquement aux comportements qui terminent. L'énoncé de correction choisi ne donne aucune garantie si le programme de départ diverge.

2.5.5 Échec de l'approche naïve

Nous avons tout d'abord essayé de prouver le compilateur par une approche directe, mais cela c'est avéré peu fructueux. Commençons tout d'abord par essayer de prouver la correction pour la compilation des expressions arithmétiques. On exprime cela en donnant à `compile_aexpr` la post-condition suivante :

```
let rec compile_aexpr (a: aexpr) : code
  ensures { ∀c: code, p: pos, m: state, s: stack.
    codeseq_at c p result →
    transition_star c (VMS p s m)
                      (VMS (p + length result)
                           (Cons (aeval m a) s)
                           m) }
```

Autrement dit, quel que soit l'endroit où le code compilé se trouve dans le résultat final, son exécution déplace la tête de lecture du bon nombre d'instructions et met la valeur de l'expression arithmétique au sommet de la pile.

La preuve de ce programme se découpe en 5 cas, un par construction syntaxique. Les cas correspondant aux variables et aux constantes entières sont démontrés automatiquement, car l'énoncé de correction correspond alors parfaitement aux règles de transition de la machine virtuelle pour les instructions générées.

Malheureusement, le cas des opérations binaires ne se passe pas aussi bien. Focalisons-nous par exemple sur la cas de la soustraction.

```
  | Asub a1 a2 →
      compile_aexpr a1 ++ compile_aexpr a2 ++ Cons Isub Nil
```

Prouver la correction de cette traduction revient à montrer l'existence d'une séquence de transitions partant de l'état `VMS p s m` vers l'état

```
VMS (p + length a1 + length a2 + 1)
    (Cons (aeval m a1 - aeval m a2) s)
    m
```

On obtient cette séquence en appliquant l'énoncé de correction sur les sous-expressions, ce qui nous donne les états intermédiaires suivants :

```
let c1 = compile_aexpr a1 in let c2 = compile_aexpr a2 in
let isub = Cons Isub Nil in let c12 = c1 ++ c2 in
let res = c12 ++ isub in
  assert { ∀c: code, p: pos, m: state, s: stack.
    codeseq_at c p res →
    let st1 = VMS p s m in  let s2 = Cons (aeval m a1) s in
    let st2 = VMS (p + length c1) s2 m in
    let st3 = VMS (p + length c12) (Cons (aeval m a2) s2) m in
    let st4 = VMS (p + length res) (Cons (aeval m a) s) m in
    transition_star c st1 st4
    by transition_star c st1 st2
    so codeseq_at c (p + length c1) c2 &&
    so transition_star c st2 st3
    so transition_star c st3 st4 };
res
```

FIGURE 2.12 – Preuve directe de la correction de la compilation d'une soustraction

```
VMS (p + length a1)
    (Cons (aeval m a1) s)
    m
VMS (p + length a1 + length a2)
    (Cons (aeval m a2) (Cons (aeval m a1) s))
    m
```

Nous avons observé que parmi la douzaine de démonstrateurs automatiques que nous avons essayés, aucun n'est parvenu à trouver ces états [3]. On pourrait penser que la difficulté survienne de la nécessité d'exhiber des instances du prédicat `codeseq_at` pour chaque transition intermédiaire. Or, quelques changements pour contourner cette difficulté n'ont pas amélioré la situation. Il est donc plus que probable que le problème réside dans la construction de ces états intermédiaires.

Une première solution est d'utiliser un assistant de preuve interactif, comme Coq, mais cela n'apporterait évidemment rien en termes d'automatisation vis-à-vis d'une preuve effectuée dans un tel environnement. Une autre possibilité serait d'aider les démonstrateurs automatiques en exhibant les états intermédiaires via les assertions de Why3. Dans le cas de la soustraction, on obtient l'assertion donnée dans la figure 2.12.

Cependant, écrire une telle assertion revient à placer une preuve explicite complète dans le code Why3, et n'apporte donc rien non plus à l'automatisation de la preuve. De plus, la nécessité de devoir aller aussi loin pour la compilation d'une opération aussi simple que la soustraction n'est pas un bon signe pour la suite.

À l'état actuel de la preuve automatique, il semble que l'on ne puisse pas prouver directement l'énoncé de correction (du moins en temps raisonnable). Une telle approche nécessite une quantité trop importante de travail manuel pour mériter le nom de preuve automatique. Nous allons donc aborder la vérification du compila-

3. dans un délai de deux minutes

teur sous un angle différent, en utilisant un calcul de plus faible pré-condition pour construire ces états intermédiaires de manière mécanique.

2.5.6 Spécification via des triplets de Hoare

Nous définissons une logique de Hoare adaptée au code non structuré de la machine virtuelle. Cette logique nous permet alors de donner des spécifications des fonctions intermédiaires en termes de triplets valides pour le code généré. Plus précisément, nous modifions nos fonctions pour renvoyer des triplets de Hoare valides, dont les pré-conditions et post-conditions correspondent au comportement du code source. Nous voyons donc un compilateur certifié comme un compilateur produisant du code certifié. De plus, l'utilisation de ces annotations nous permet d'employer des règles de combinaison qui simplifient les obligations de preuve.

Le changement de spécification est aisé car les énoncés de correction ont naturellement une telle structure. Par exemple, dans le cas des expressions arithmétiques, la pré-condition serait simplement le fait que la tête de lecture est au début du code généré, tandis que la post-condition correspondrait à l'empilement du résultat de l'expression.

Nous commençons donc par définir la forme de ces triplets de Hoare en Why3 :

```
type pre  = pos → machine_state → bool
type post = pos → machine_state → machine_state → bool
type hl   = { code: code; ghost pre: pre; ghost post: post }
```

Comme on peut le remarquer, les pré- et post-conditions sont paramétrées par une position. Celle-ci indique la position à laquelle se trouvent les instructions spécifiées par rapport au programme global. On remarque également que la post-condition parle de deux états machine, le premier étant l'état initial. Notons enfin que ces annotations font partie du code fantôme et sont donc uniquement des outils de spécification. En particulier, elles n'interviendront pas à l'exécution du compilateur.

Remarque 2.2. Au lieu de paramétrer les spécifications par une position *absolue*, nous aurions pu donner les positions d'une manière *relative* dans les spécifications, autrement dit comme si les instructions spécifiées se trouvaient à la position zéro. Néanmoins, il serait alors nécessaire d'introduire explicitement des décalages de la tête de lecture dans toutes les annotations lorsque le code est placé à une autre position que zéro. Nous avons choisi la première solution car elle nous a paru moins intrusive.

Le sens que nous attribuons à ces triplets correspond à la notion de *correction totale*. Autrement dit, quel que soit l'état initial, si la pré-condition est vérifiée, on atteindra un état final où la post-condition est vérifiée. Cette propriété exprime bien la correction totale car la machine virtuelle est déterministe. Nous assurons que tout élément du type `hl` représente bien un triplet de Hoare valide en équipant ce type d'un invariant qui exprime cette propriété (cf. figure 2.13). Nous devrons par la suite établir cet invariant à chaque fois que nous construisons un nouvel élément de ce type. Notons que comme les types de Why3 sont non vides, nous devons fournir une garantie d'habitation pour pouvoir ajouter cet invariant, que nous obtenons en donnant explicitement un témoin.

Armés de ces triplets, nous pouvons maintenant exprimer la spécification des fonctions de compilation de la manière suivante : elles renvoient un triplet, dont

```
predicate contextual_irrelevance
            (c: code) (p: pos) (ms1 ms2: machine_state) =
  ∀c_g. codeseq_at c_g p c → transition_star c_g ms1 ms2

type hl = { code: code; ghost pre: pre; ghost post: post }
  invariant { ∀p ms. pre p ms →
    ∃ms'. post p ms ms' ∧ contextual_irrelevance code p ms ms' }
  (* témoin d'habitation *)
  by { code = Nil; pre = (λ_ _. false); post = λ_ _ _. true }
```

FIGURE 2.13 – Invariant ajouté au type `hl`

la pré-condition et la post-condition sont des fonctions connues du code source. Le triplet renvoyé est forcément valide à cause de l'invariant associé au type `hl`. Les énoncés précis sont donnés en figure 2.14. Étant donné que la structure des énoncés de correction est similaire, nous pouvons sans difficulté dériver des spécifications plus naturelles pour ces trois procédures.

Remarquons que pour pouvoir construire notre compilateur certifié en combinant des triplets, nous avons également besoin d'établir des triplets basiques pour les instructions de la machine virtuelle. Nous introduisons donc pour chaque instruction une opération qui l'encapsule dans un triplet. Comme la validité de chacun de ces triplets correspond exactement aux règles de transition de la machine virtuelle, vérifier ces triplets de manière automatique ne pose pas de problème particulier. Pour la soustraction, nous obtenons la spécification donnée dans la figure 2.15.

La pré-condition des opérations binaires requiert que deux opérandes soient présents au sommet de la pile, et la post-condition exprime que ceux-ci ont été remplacés par le résultat de l'opération.

2.5.7 Calcul de plus faible pré-condition

La simple utilisation des triplets ne suffit évidemment pas à faciliter la preuve du compilateur. Nous avons donc besoin d'une méthode pour combiner ces triplets entre eux. La logique de Hoare telle quelle n'est pas très satisfaisante car elle requiert un grand nombre d'annotations. En effet, il faut typiquement ajuster explicitement les prédicats via des règles comme l'affaiblissement pour pouvoir appliquer les règles d'inférence. Pour éviter ce problème, nous utilisons le calcul de plus faible pré-condition pour effectuer la combinaison.

Comme pour les triplets, nous commençons par introduire une forme d'annotation du code qui correspond aux *transformateurs de prédicat en arrière (backward predicate transformers)*.

```
type pred = machine_state → bool
type wp_trans = pos → pred → pred
type wp = { wcode : code ; ghost wp : wp_trans }
  invariant { ∀p post ms. (wp p post) ms →
    ∃ms'. post ms' ∧ contextual_irrelevance wcode p ms ms' }
  by { wcode = Nil; wp = λ_ _ _. false }
```

```
(* Pré-condition triviale: tête de lecture au début du code *)
function trivial_pre : pre =
  λp ms. let VMS p' _ _ = ms in p = p'

(* Post-condition pour le code généré associé à
   une expression arithmétique: tête de lecture à la fin
   du code, expression en sommet de pile, mémoire inchangée *)
function aexpr_post (a: aexpr) (len: pos) : post =
  λp ms ms'. let VMS _ s m = ms in
    ms' = VMS (p+len) (Cons (aeval m a) s) m

let rec compile_aexpr (a: aexpr) : hl
  ensures { result.pre = trivial_pre }
  ensures { result.post = aexpr_post a result.code.length }

(* Post-condition pour le code généré associé à une
   expression booléenne: pile et mémoire inchangées,
   tête de lecture positionnée selon le résultat. *)
function bexpr_post (b: bexpr) (c: bool) (o_t o_f:ofs) : post =
  λp ms ms'. let VMS _ s m = ms in if beval m b = c
    then ms' = VMS (p+o_t) s m
    else ms' = VMS (p+o_f) s m

let rec compile_bexpr (b: bexpr) (c: bool) (ofs: ofs) : hl
  ensures { result.pre = trivial_pre }
  ensures { let len = result.code.length in
    result.post = bexpr_post b c (len + ofs) len }

(* Pré-condition pour le code généré associé à une commande:
   tête de lecture au début du code, et le programme source
   termine *)
function com_pre (cmd: com) : pre =
  λp ms. let VMS p' _ m = ms in p = p' ∧ ∃m'. ceval m cmd m'

(* Post-condition pour le code généré associé à une commande:
   tête de lecture à la fin du code, pile inchangée,
   et mémoire modifiée comme pour la commande. *)
function com_post (cmd: com) (len: pos) : post =
  λp ms ms'. let VMS _ s m = ms in let VMS p' s' m' = ms' in
    p' = p + len ∧ s' = s ∧ ceval m cmd m'

let rec compile_com (cmd: com) : hl
  ensures { result.pre = com_pre cmd }
  ensures { result.post = com_post cmd result.code.length }
```

FIGURE 2.14 – Spécification via une logique de Hoare

```
constant ibinop_pre : pre =
  λp ms. ∃n1 n2 s m. ms = VMS p (Cons n2 (Cons n1 s)) m
function ibinop_post (op: int → int → int) : post =
  λp ms ms'. ∀n1 n2 s m. ms = VMS p (Cons n2 (Cons n1 s)) m →
                         ms' = VMS (p+1) (Cons (op n1 n2) s) m
let isubf () : hl
  ensures { result.pre = ibinop_pre }
  ensures { result.post = ibinop_post (λx y. x - y) }
  ensures { result.code.length = 1 }
```

FIGURE 2.15 – Spécification de l'instruction de soustraction

Étant donnée une propriété Q portant sur les états machine, un tel transformateur (de type `wp_trans`) renvoie une condition suffisante sur l'état de départ pour que la machine finisse par arriver dans un état vérifiant Q. Cette propriété est précisément fournie par l'invariant de type associé au type `wp`. Comme pour les triplets, ce transformateur est paramétré par la position absolue du code généré dans le code final.

Maintenant que nous avons deux formes différentes d'annotations, il est nécessaire d'introduire des convertisseurs de l'une vers l'autre. Le premier convertisseur (`$`) correspond exactement au calcul de plus faible pré-condition pour un appel de fonction.

```
function towp_wp (pr: pre) (ps: post) : wp_trans =
  λp q ms. pr p ms && ∀ms'. ps p ms ms' → q ms'
let ($_) (c: hl) : wp      (* Déclaration d'un opérateur unaire $ *)
  ensures  { result.wcode.length = c.code.length }
  ensures  { result.wp = towp_wp c.pre c.post }
= { wcode = c.code; wp = towp_wp c.pre c.post }
```

Le convertisseur dans l'autre sens correspond alors à la vérification d'un tel contrat vis-à-vis de la pré-condition calculée, comme l'indique sa pré-condition.

```
let hoare (ghost pre: pre) (c: wp) (ghost post: post) : hl
  (* correction du contrat *)
  requires { ∀p ms. pre p ms → (c.wp p (post p ms)) ms }
  ensures  { result.pre = pre }
  ensures  { result.post = post }
  ensures  { result.code.length = c.wcode.length }
= { code = c.wcode; pre = pre; post = post }
```

Ces deux convertisseurs nous permettent de basculer entre deux mondes : le monde des triplets de Hoare, qui permet de donner des spécifications précises, et celui des transformateurs, qui permettent de faire des combinaisons intermédiaires. Le premier type de combinaison que nous pouvons effectuer correspond à la mise en séquence de deux listes d'instructions consécutives.

```
let rec compile_aexpr (a:aexpr) :  hl
  ensures { result.pre = trivial_pre }
  ensures { result.post = aexpr_post a result.code.length }
  variant { a }
= let c = match a with
    | Anum n     → $ iconstf n
    | Avar x     → $ ivarf x
    | Aadd a1 a2 →
      $ compile_aexpr a1 -- $ compile_aexpr a2 -- $ iaddf ()
    | Asub a1 a2 →
      $ compile_aexpr a1 -- $ compile_aexpr a2 -- $ isubf ()
    | Amul a1 a2 →
      $ compile_aexpr a1 -- $ compile_aexpr a2 -- $ imulf ()
    end in
  hoare trivial_pre c (aexpr_post a c.wcode.length)
```

FIGURE 2.16 – Compilateur des expressions arithmétiques, avec transformateurs

```
function seq_wp (l1: int) (w1 w2: wp_trans) : wp_trans =
  λp q ms. w1 p (w2 (p + l1) q) ms
let (--) (s1: wp) (s2: wp) : wp
  ensures  {
    result.wcode.length = s1.wcode.length + s2.wcode.length }
  ensures  { result.wp = seq_wp s1.wcode.length s1.wp s2.wp }
= { wcode = s1.wcode ++ s2.wcode;
    wp = seq_wp s1.wcode.length s1.wp s2.wp }
```

Cet opérateur correspond à la règle de calcul usuelle pour la plus faible pré-condition d'une séquence. Comme nous l'avons mentionné au début de cette section, le choix des positions absolues nous épargne des décalages explicites de la tête de lecture dans la condition suffisante ainsi calculée.

Les opérateurs définis jusqu'ici suffisent pour traiter les opérations arithmétiques. Nous réécrivons alors le compilateur des expressions arithmétiques sous la forme donnée dans la figure 2.16.

La partie non triviale de la preuve se réduit alors à vérifier la pré-condition de l'opérateur `hoare` à la fin de la fonction de compilation. Concrètement, celle-ci correspond précisément à des conditions de vérification pour le code généré. Le problème des états intermédiaires est alors résolu car le combinateur de séquence fait automatiquement le lien entre les différentes étapes. Cependant, nous ne sommes pas encore au bout : les démonstrateurs automatiques peinent à vérifier ces conditions.

Ce dernier phénomène s'explique par la structure de la condition de vérification, laquelle est une combinaison d'objets d'ordre supérieur, en particulier l'application du transformateur final. Il faut réduire cette application un certain nombre de fois pour obtenir une condition de vérification exploitable, mais les démonstrateurs automatiques ne le font pas forcément, ou pas de manière prioritaire. Cette réduction correspond au calcul de plus faible pré-condition au sens effectif du terme, et est

donc une tâche idéale pour la transformation de calcul `compute`.

Pour effectuer ce calcul, nous utilisons les règles de réécriture suivantes :

— Les définitions de tous les opérateurs de combinaisons. Ici, cela revient à dire que nous marquons les définitions des symboles `to_wp` et `seq_wp` comme règles de réécriture.

— Les définitions de toutes les pré/post-conditions.

— Les égalités définissant toutes les annotations fantômes dans les post-conditions des fonctions Why3. Par exemple, une égalité définissant une pré-condition prend la forme `"rewrite" result.pre = pre`.

Cela permet d'utiliser la transformation de calcul pour éliminer complètement les objets d'ordre supérieurs introduits pour la mise en place du calcul de plus faible pré-condition, réduisant ainsi les conditions de vérification au premier ordre. Après cette transformation, la totalité des obligations de preuve obtenues pour `compile_aexpr` sont vérifiées automatiquement, en moins de 5 secondes de temps cumulé.

Armés de ces nouveaux outils, nous sommes également capable de vérifier quasi intégralement le compilateur des expressions booléennes. Le seul cas qui résiste est celui de la conjonction, car celle-ci est traitée de manière paresseuse. En effet, les opérateurs présentés jusqu'à présent ne permettent de vérifier que du code en ligne droite. Or, le traitement paresseux introduit un comportement conditionnel. Nous avons donc naturellement introduit un nouvel opérateur pour refléter ce comportement :

```
function fork_wp (w: wp_trans) (cond: pre) : wp_trans =
  λp q ms. (cond p ms → w p q ms) ∧ (not cond p ms → q ms)
let (%) (s: wp) (ghost cond: pre) : wp
  ensures  { result.wp = fork_wp s.wp cond }
  ensures  { result.wcode.length = s.wcode.length }
= { wcode = s.wcode; wp = fork_wp s.wp cond }
```

Cet opérateur permet d'ignorer le comportement du code sous-jacent lorsque la condition est fausse, auquel cas nous supposons alors qu'aucune instruction n'est exécutée. Nous pouvons alors vérifier la compilation paresseuse de la conjonction par la combinaison suivante :

```
function exec_cond (b: bexpr) (c: bool) : pre =
  λp ms. let VMS _ _ m = ms in beval m b = c
        ⋮
  | Band b1 b2 →
      let c2 = $ compile_bexpr b2 c ofs % exec_cond b1 true in
      let len = length c2.wcode in
      let ofs = if c then len else ofs + len in
      $ compile_bexpr b1 false ofs -- c2
```

Ce cas est alors lui aussi prouvé automatiquement.

2.5.8 Extension de la méthode aux commandes

Pour étendre notre méthode à la compilation des commandes, nous avons dû étendre la portée de notre calcul de plus faible pré-condition. Sans surprise, le cas de la boucle `WHILE` est non trivial, et demande notamment un opérateur de boucle.

```
type pred = machine_state → bool
type pre 'a  = 'a → pos → pred
type post 'a = 'a → pos → machine_state → pred
type hl 'a  = { code: code; ghost pre: pre 'a; ghost post: post 'a }
  invariant { ∀x:'a,p ms. pre x p ms →
      ∃ms'. post x p ms ms' ∧ contextual_irrelevance code p ms ms' }
  by { code = Nil; pre = (λ_ _ _. false); post = λ_ _ _ _. true }
type wp_trans 'a = 'a → pos → pred → pred
type wp 'a = { wcode : code; ghost wp: wp_trans 'a }
  invariant { ∀x:'a,p post ms. (wp x p post) ms →
      ∃ms'. post ms' ∧ contextual_irrelevance wcode p ms ms' }
  by { wcode = Nil; wp = λ_ _ _ _. false }
```

FIGURE 2.17 – Code annoté, avec paramètres auxiliaires

De manière plus surprenante, le cas d'une construction conditionnelle `Cif b c1 c2` ne peut pas être traité directement non plus. En effet, le problème est que nous mettons les codes compilés pour les commandes `c1` et `c2` en séquence. Comme l'état mémoire peut avoir changé après l'exécution de `c1`, nous n'avons plus accès aux informations nécessaires pour savoir si `c2` doit être exécuté ou non. En effet, cette information dépend de l'état mémoire initial, que nous ne pouvons pas reconstruire.

La solution que nous proposons pour éviter ce problème est d'ajouter des variables auxiliaires, autrement dit de paramétrer nos triplets et transformateurs par des données supplémentaires. Ces données nous permettent notamment de stocker un état précédent, ce qui règle le problème observé pour la conditionnelle. Remarquons que ces données jouent un rôle proche de celui du code fantôme dans Why3.

Concrètement, nous ajoutons un paramètre aux pré-conditions, post-conditions et transformateurs. Comme le type des données que l'on souhaite stocker est a priori inconnu, nous le représentons par un paramètre de type. Les triplets et transformateurs sont donc redéfinis en ajoutant ce paramètre aux membres d'ordre supérieur, par les définitions données dans la figure 2.17.

La correction des triplets (resp. transformateurs) est maintenant exprimée en quantifiant universellement sur les variables auxiliaires. Au vu de ces nouvelles définitions, nous observons que le paramètre de position joue en fait le rôle d'une variable auxiliaire, et nous pourrions le traiter comme tel. En pratique, il est plus simple de le traiter séparément du fait de son omniprésence.

Ce changement fait, nous adaptons facilement nos opérateurs à ces nouvelles définitions. Le seul opérateur que nous changeons de manière significative est l'opérateur de mise en séquence. Nous l'adaptons de manière à automatiquement 'photographier' l'état initial, qui devient alors visible dans la seconde partie.

```
function seq_wp (l1: int) (w1: wp_trans 'a)
                (w2: wp_trans ('a, machine_state)) : wp_trans 'a
  = λx p q ms. w1 x p (w2 (x, ms) (p+l1) q) ms
```

En particulier, nous avons maintenant accès à l'état initial pour écrire la condition sous laquelle le deuxième cas de la conditionnelle est exécuté. Cela nous permet

```
function loop_wp (w:wp_trans 'a)
                 (inv cont:pre 'a) (var:post 'a) : wp_trans 'a =
  λx p q ms. inv x p ms && acc (var x p) ms
    && ∀ms'. inv x p ms' →
        if cont x p ms'
        then w x p (λy. inv x p y && var x p y ms') ms'
        else w x p q ms'
let make_loop (c:wp 'a) (ghost inv cont:pre 'a)
                            (ghost var:post 'a) : wp 'a
  ensures { result.wp = loop_wp w inv cont var }
  ensures { result.wcode.length = c.wcode.length }
= let wpt = loop_wp inv cont var in
  assert { ∀x p q ms0. wpt x p q ms0 →
    ∀ms. inv x p ms → acc (var x p) ms →
      ∃ms'. contextual_irrelevance c.wcode p ms ms' ∧ q ms'
  };
  { wcode = c.wcode; wp = wpt }
```

FIGURE 2.18 – Combinateur de boucle

de vérifier la compilation de cette construction automatiquement aussi.

Nous nous intéressons maintenant au cas de la boucle. Sans surprise, nous avons besoin de pouvoir exprimer la notion d'exécution répétée, ce qui nous oblige à introduire un nouvel opérateur sur les transformateurs de prédicats. Nous définissons un combinateur inspiré du calcul de plus faible pré-condition pour une boucle, paramétré par un prédicat qui teste si la boucle doit continuer, un invariant, et un variant. Ce dernier ingrédient est nécessaire pour garantir la terminaison de la boucle

Commençons par expliquer le variant. Il s'agit dans notre cas d'une relation qui doit être bien fondée sur les états qui seront atteints durant l'itération. Nous devons donc définir la notion de bonne fondation. Nous utilisons pour cela le prédicat usuel d'accessibilité, qui caractérise l'ensemble des éléments pour lesquels la relation est bien fondée :

```
inductive acc ('a → 'a → bool) 'a =
  | Acc : ∀r, x:'a. (∀y. r y x → acc r y) → acc r x
```

Nous définissons ensuite le combinateur de boucle en utilisant cette notion (cf. figure 2.18). La pré-condition calculée par le combinateur de boucle exprime que l'invariant de boucle choisi est vérifié pour l'état initial, que cet état est dans la partie bien fondée du variant, et qu'à chaque itération, l'exécution progresse vers la post-condition. Autrement dit, à l'issue d'une itération, soit la post-condition est établie et la boucle est terminée, soit l'invariant de la boucle est établi de nouveau et l'état devient « plus petit » au sens du variant. Nous utilisons le test de continuation pour identifier le cas correspondant à un état donné.

Sans surprise, les démonstrateurs automatiques ne prouvent pas directement la validité de ce combinateur à cause de sa structure inductive. Nous devons donc manuellement ajouter une courte assertion pour expliciter l'induction justifiant sa

```
| Cwhile test body  → let code_body = compile_com body in
    let body_length = length code_body.code + 1 in
    let code_test = compile_bexpr test false body_length in
    let ofs = length code_test.code + body_length in
    let wp_while = $ code_test --
        ($ code_body -- $ ibranchf (- ofs)) % exec_cond test true in
    let ghost inv  = loop_invariant cmd in
    let ghost var  = loop_variant body test  in
    $ inil () -- make_loop wp_while inv (exec_cond test true) var
```

FIGURE 2.19 – Compilation vérifiée de la construction de boucle

validité. Une fois la transformation d'induction (cf. section 2.2) appliquée à l'obligation de preuve associée à l'assertion, la preuve devient complètement automatique.

Nous utilisons maintenant ce combinateur pour vérifier le cas de la boucle `Cwhile b c0`. Pour cela, nous définissons d'abord les paramètres effectifs du combinateur.

```
(* c = Cwhile b c0 *)
function loop_invariant (c: com) : pre ('a, machine_state) =
  λx p msi. let VMS _ s0 m0 = snd x in let VMS pi si mi = msi in
    pi = p ∧ s0 = si ∧ ∃mf. ceval m0 c mf ∧ ceval mi c mf
(* c = c0, test = b *)
function loop_variant (c: com) (test: bexpr) : post 'a =
  λ_ _ msj msi. let VMS pj sj mj = msj in let VMS pi si mi = msi in
     pj = pi ∧ sj = si ∧ ceval mi c mj ∧ beval mi test
```

L'invariant spécifie que la construction de boucle évalue les états intermédiaires et l'état initial en le même état final. Le variant correspond lui à une itération du corps de la boucle. La compilation de la boucle consiste donc à générer le code pour une itération, puis à appliquer le combinateur de boucle avec les paramètres ci-dessus au résultat. Nous donnons le code Why3 résultant dans la figure 2.19. La seule surprise dans ce fragment est l'ajout d'un code vide en tête du code généré. Il s'agit d'une astuce pour ajouter l'état initial aux variables auxiliaires sans avoir à ajouter un nouveau combinateur pour « photographier » cet état.

Qu'en est-il de la vérification des buts générés ? Malheureusement, une obligation de preuve résiste aux démonstrateurs automatiques. De manière peu surprenante, il s'agit de la vérification que le variant est bien fondé, ce qui nécessite une preuve par induction non triviale sur la sémantique de Imp. Nous avons effectué cette preuve par le biais d'un lemme sur la sémantique du langage source, qui énonce que tout état qui s'évalue à travers une boucle est bien accessible pour notre variant. Nous utilisons alors la transformation d'induction pour démontrer ce lemme, en conjonction avec l'indicateur de coupure `by` (cf. section 2.3) pour fournir des indications aux démonstrateurs automatiques.

```
lemma loop_variant_acc : ∀c test,x:'a,p mi mj.
  let wh = Cwhile test c in let var = loop_variant c test x p in
    (ceval mi wh mj → ∀pi si. acc var (VMS pi si mi))
    by ∀pi si mi mj mf. ceval mi c mj ∧ beval mi test →
      ceval mj wh mf ∧ (∀pj sj. acc var (VMS pj sj mj)) →
      acc var (VMS pi si mi)
      by (∀pk sk mk. var (VMS pk sk mk) (VMS pi si mi) → mk = mj)
```

La démonstration de ce lemme clôt la preuve de ce petit compilateur.

2.5.9 Comparaison avec une preuve directe

Le point de départ de notre expérience était la preuve directe du même petit compilateur en Coq, plus précisément la partie concernant la simulation en avant. Avons-nous gagné en automatisation vis-à-vis de cette preuve ? En termes de la preuve proprement dite, nous avons réussi à la rendre quasi-automatique. En revanche, notre approche induit un coût non négligeable en spécifications. Concrètement, notre développement représente un peu moins de 500 lignes de code, ce qui est comparable à la partie correspondante du développement Coq (un peu plus de 400 lignes). Cependant, la formalisation de la logique de Hoare et des spécifications des instructions de la machine virtuelle, qui représente 60% de notre développement, est indépendante du compilateur proprement dit. Elle pourrait être réutilisée telle quelle pour un autre compilateur vers cette même machine, ce qui représente un avantage vis-à-vis d'une preuve directe.

2.6 Application : preuve de l'algorithme de multiplication de matrices de Strassen par réflexion

Dans cette section, nous présentons une preuve de l'algorithme de multiplication de Strassen qui utilise de manière importante la transformation de calcul. L'algorithme de Strassen est un algorithme de multiplication de matrices dont la complexité est meilleure que l'algorithme naïf, plus précisément $O(n^{\log_2(7)})$ pour multiplier deux matrices carrées $n \times n$ au lieu de $O(n^3)$. L'algorithme que nous présentons ici fonctionne (et est vérifié) également pour les matrices rectangulaires.

La preuve que nous présentons est un fragment d'une solution complète au premier problème de la compétition VerifyThis 2016 [4]. Cette solution fait partie de la galerie d'exemples de Why3 [5], et est décrite en tant que telle dans [27]. Nous présentons ici l'utilisation combinée du code fantôme et de la transformation de calcul pour démontrer les identités sur lesquelles l'algorithme de Strassen est basé. Comme ces identités sont entièrement démontrées par la transformation, il s'agit d'une preuve par réflexion. Elle suit la méthodologie présentée par Bertot et Castéran [11, chapitre 16]. À notre connaissance, il s'agit de la première utilisation de la preuve par réflexion dans le cadre d'un environnement principalement basé sur les démonstrateurs automatiques.

4. http://etaps2016.verifythis.org/
5. http://toccata.lri.fr/gallery/verifythis_2016_matrix_multiplication.en.html

```
type mat 'a
function rows (mat 'a) : int
function cols (mat 'a) : int
axiom rows_and_cols_nonnegative:
  ∀m: mat 'a. 0 ≤ rows m ∧ 0 ≤ cols m
function get (mat 'a) int int : 'a
predicate (==) (m1 m2: mat 'a) =
  rows m1 = rows m2 ∧ cols m1 = cols m2 ∧
  ∀i j: int. 0 ≤ i < rows m1 → 0 ≤ j < cols m1 →
    get m1 i j = get m2 i j
axiom extensionality: ∀m1 m2: mat 'a. m1 == m2 → m1 = m2
function create (r c: int) (f: int → int → 'a) : mat 'a
axiom create_rows:
  ∀r c: int, f: int → int → 'a.
    0 ≤ r → rows (create r c f) = r
axiom create_cols:
  ∀r c: int, f: int → int → 'a.
    0 ≤ c → cols (create r c f) = c
axiom create_get:
  ∀r c: int, f: int → int → 'a, i j: int.
    0 ≤ i < r → 0 ≤ j < c → get (create r c f) i j = f i j
```

FIGURE 2.20 – Noyau axiomatique de la théorie des matrices

Notons qu'il y a eu d'autres travaux de vérification de l'algorithme de Strassen. Notre preuve est notamment assez proche de celle de Dénès et al [38] dans l'assistant de preuve Coq, qui utilise également une preuve par réflexion via la tactique `ring`. Cependant, cette tactique limite leur preuve au cas des matrices carrées.

Nous pouvons également citer une preuve de l'algorithme de Strassen donnée dans l'archive des preuves formelles [79]. Celle-ci traite le cas des matrices rectangulaires, et utilise le système de simplification intégré à l'assistant Isabelle pour prouver les identités de matrices.

Enfin, remarquons qu'une preuve très automatique de l'algorithme de Strassen a été effectuée dans le système ACL2 [71], via l'emploi d'un système de réécriture bien choisi. Cependant, les choix de spécification des auteurs limitent cette preuve au cas des matrices carrées de taille une puissance de 2.

2.6.1 Spécification via une théorie des matrices

Pour spécifier l'algorithme de Strassen, nous avons écrit une théorie des matrices regroupant les opérations arithmétiques, l'extraction de sous-matrices et les relations algébriques entre ces différentes opérations. Ces opérations et relations sont toutes respectivement définies et prouvées à partir d'un noyau axiomatique restreint, présenté dans la figure 2.20. Ce noyau comprend notamment une fonction d'ordre supérieur `create` qui permet de définir une matrice par compréhension, ce que nous faisons pour toutes les opérations.

L'utilisation de l'ordre supérieur nous a ainsi permis de réduire considérablement la partie axiomatique de cette théorie. Par exemple, nous pouvons directement définir la multiplication de la façon suivante :

```
function mul_atom (a b: mat int) (i j:int) : int → int =
  λk. get a i k  * get b k j
function mul_cell (a b: mat int): int → int → int =
  λi j. sum (mul_atom a b i j) 0 (cols a)
function mul (a b: mat int) : mat int =
  create (rows a) (cols b) (mul_cell a b)
```

Nous utilisons également l'ordre supérieur au sein du symbole `sum`, qui correspond au symbole mathématique Σ et fait partie de la bibliothèque standard de Why3.

Pour prouver les relations algébriques comme le caractère associatif des opérateurs d'addition et de multiplication, nous avons principalement utilisé les propriétés de la function `sum` et l'extensionnalité de l'égalité des matrices. Nous avons utilisé à la fois le mécanisme des fonctions-lemmes et des indicateurs de coupures pour assister les démonstrateurs automatiques lorsque nécessaire.

Cette théorie nous fournit un modèle des matrices. Pour implémenter l'algorithme de Strassen, nous avons utilisé les matrices fournies par la bibliothèque standard de Why3. Ces dernières correspondent à des tableaux impératifs bidimensionnels, pour laquelle la bibliothèque fournit les opérations de création, d'accès et de mise à jour :

```
type matrix 'a model {
  rows: int;
  columns: int;
  mutable elts: map int (map int 'a)
}
```

Nous lions les matrices fournies par la bibliothèque standard de Why3 à notre modèle par une fonction dédiée `mdl` (« modèle ») définie par compréhension :

```
function mdl (m: matrix 'a) : mat 'a =
  create m.rows m.cols (λi j. m.elts[i][j])
```

2.6.2 Algorithme de Strassen

Le principe de base de l'algorithme de Strassen est d'utiliser la multiplication par bloc de taille 2×2 récursivement, en ne faisant que sept multiplications récursives par appels au lieu des huit multiplications du schéma naturel. Plus précisément, l'algorithme consiste en premier lieu à partitionner les matrices d'entrées A, B et de sortie M en quatre matrices de même taille.

$$A = \left[\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array}\right] \qquad B = \left[\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \end{array}\right] \qquad M = \left[\begin{array}{c|c} M_{1,1} & M_{1,2} \\ \hline M_{2,1} & M_{2,2} \end{array}\right]$$

Ensuite, les composantes de M sont obtenues par combinaison linéaire de sept matrices intermédiaires

$$\begin{array}{lcllcl} M_{1,1} & = & X_1 + X_4 - X_5 + X_7 & \qquad M_{2,1} & = & X_2 + X_4 \\ M_{1,2} & = & X_3 + X_5 & \qquad M_{2,2} & = & X_1 - X_2 + X_3 + X_6 \end{array}$$

lesquelles sont elle-même obtenues par le biais de combinaisons linéaires et d'un total de sept multiplications

$$\begin{array}{rclcrcl} X_1 & = & (A_{1,1}+A_{2,2})\times(B_{1,1}+B_{2,2}) & & X_2 & = & (A_{2,1}+A_{2,2})\times B_{1,1} \\ X_3 & = & A_{1,1}\times(B_{1,2}-B_{2,2}) & & X_4 & = & A_{2,2}\times(B_{2,1}-B_{1,1}) \\ X_5 & = & (A_{1,1}+A_{1,2})\times B_{2,2} & & X_6 & = & (A_{2,1}-A_{1,1})\times(B_{1,1}+B_{1,2}) \\ X_7 & = & (A_{1,2}-A_{2,2})\times(B_{2,1}+B_{2,2}) & & & & \end{array}$$

Ce schéma récursif ne fonctionne que lorsque les dimensions des matrices d'entrée sont paires. Il y a plusieurs méthodes pour gérer le cas des dimensions impaires. Principalement, nous pouvons soit agrandir les matrices avec des 0, soit faire un produit par blocs brut pour enlever quelques lignes et/ou colonnes. Nous avons choisi de procéder de la manière suivante. Lorsque notre fonction récursive est appelée avec des matrices dont au moins une dimension est impaire, nous ajoutons tout d'abord autant de lignes/colonnes nulles que nécessaire pour rendre la totalité des dimensions paires. Ensuite, nous rappelons immédiatement notre fonction récursive sur les matrices ainsi agrandies, puis nous extrayons du résultat le bloc correspondant au produit des matrices initiales.

Enfin, pour garantir la terminaison il faut utiliser un autre algorithme quand les dimensions deviennent suffisamment petites. Nous utilisons pour cela un algorithme naïf dès que l'une des dimensions passe en-dessous d'une certaine constante `cut_off`. Comme n'importe quelle valeur strictement positive fonctionne, nous avons gardée cette constante abstraite de manière à ce que la preuve soit indépendante de sa valeur concrète.

Le code que nous avons prouvé est donné en figure 2.21. En plus de la multiplication naïve (`mul_naive`), il fait usage des routines suivantes, dont le code et la preuve sont complètement standard :

— Addition (`add`) et soustraction (`sub`) de matrices
— Extraction de blocs (`block`) et copie bloc-vers-bloc (`blit`).
— Augmentation par des zéros (`padding`).

Notre preuve de cet algorithme est divisée en trois axes. Le premier est de démontrer le produit par bloc naturel, autrement dit

$$\begin{array}{rclcrcl} M_{1,1} & = & A_{1,1}B_{1,1}+A_{1,2}B_{2,1} & & M_{2,1} & = & A_{2,1}B_{1,2}+A_{2,2}B_{2,1} \\ M_{1,2} & = & A_{1,1}B_{1,2}+A_{1,2}B_{2,2} & & M_{2,2} & = & A_{2,1}B_{1,2}+A_{2,2}B_{2,2} \end{array}$$

Notre théorie des matrices ne contient pas cette propriété telle quelle, mais nous permet de la dériver via un effort raisonnable. Le second axe est la terminaison, qui est plus subtile qu'il n'y paraît à cause de l'augmentation potentielle des dimensions lors d'un appel récursif. Le troisième est de démontrer que les quatre matrices $M_{i,j}$ calculées par le schéma récursif de Strassen sont bien celle attendues pour un produit par bloc, ce qui revient à démontrer quatre identités de matrices. Il s'agit de l'axe sur lequel nous allons nous concentrer ici.

En effet, nous n'avons pas réussi à démontrer une seule de ces identités sans devoir apporter une assistance considérable aux démonstrateurs automatiques. En utilisant les indicateurs de coupures pour fournir les étapes, nous avons été obligé d'écrire presque toutes les étapes de développement et de simplification, ainsi que d'expliciter un certain nombre de dimensions pour les matrices intermédiaires. Une explication possible est que les démonstrateurs n'identifient pas les propriétés algébriques des opérateurs de matrices comme telles, et les utilisent donc comme des

```
constant cut_off : int
axiom cut_off_positive : cut_off > 0
let rec strassen (a b: matrix int) : matrix int
= let (rw, md, cl) = (a.rows, a.columns, b.columns) in
  if rw ≤ cut_off || md ≤ cut_off || cl ≤ cut_off
  then mul_naive a b else
  let (qr, rr) = (div rw 2, mod rw 2) in
  let (qm, rm) = (div md 2, mod md 2) in
  let (qc, rc) = (div cl 2, mod cl 2) in
  if rr ≠ 0 || rm ≠ 0 || rc ≠ 0 then begin (* Augmentation *)
    let (rw', md', cl') = (rw + rr, md + rm, cl + rc) in
    let ap = padding a rw' md' in
    let bp = padding b md' cl' in
    let m  = strassen ap bp in
    block m 0 rw 0 cl
  end else begin (* Schéma recursif *)
    let (m11, m12, m21, m22) =
      let a11 = block a 0 qr 0 qm in
      let a12 = block a 0 qr qm qm in
      let a21 = block a qr qr 0 qm in
      let a22 = block a qr qr qm qm in
      let b11 = block b 0 qm 0 qc in
      let b12 = block b 0 qm qc qc in
      let b21 = block b qm qm 0 qc in
      let b22 = block b qm qm qc qc in
      let x1  = strassen (add a11 a22) (add b11 b22) in
      let x2  = strassen (add a21 a22) b11 in
      let x3  = strassen a11 (sub b12 b22) in
      let x4  = strassen a22 (sub b21 b11) in
      let x5  = strassen (add a11 a12) b22 in
      let x6  = strassen (sub a21 a11) (add b11 b12) in
      let x7  = strassen (sub a12 a22) (add b21 b22) in
      let m11 = add (sub (add x1 x4) x5) x7 in
      let m12 = add x3 x5 in
      let m21 = add x2 x4 in
      let m22 = add (add (sub x1 x2) x3) x6 in
      (m11, m12, m21, m22) in
    let res = make a.rows b.columns 0 in
    blit m11 res 0 0 qr 0 0 qc;
    blit m12 res 0 0 qr 0 qc qc;
    blit m21 res 0 qr qr 0 0 qc;
    blit m22 res 0 qr qr 0 qc qc;
    res
  end
```

FIGURE 2.21 – Implémentation de l'algorithme de Strassen

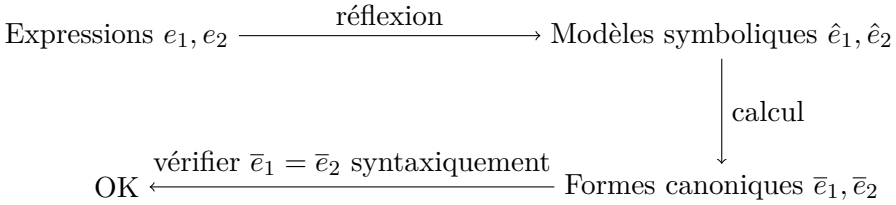

FIGURE 2.22 – Preuve par réflexion

axiomes génériques. De plus, ces axiomes sont sujets à des contraintes de dimensions, ce qui peut expliquer pourquoi nous avons dû en fournir certaines.

Comme fournir explicitement toutes ces étapes est long et pénible, nous avons choisi une autre approche. Nous avons écrit une procédure de normalisation des expressions algébriques directement dans Why3, et l'avons exécutée via la transformation de calcul. La méthodologie que nous avons utilisée pour arriver à nos fins correspond précisément à la preuve par réflexion.

Notons que nous aurions également pu utiliser un assistant de preuve possédant une telle procédure de normalisation. Nous aurions alors effectué la preuve en faisant correspondre nos matrices à l'interface de l'assistant. Cependant, il n'était pas évident que cette correspondance soit aisée à mettre en place car la logique doit être traduite avant d'être envoyée à un assistant de preuve, et de plus nous étions curieux de voir si nous pouvions obtenir le même résultat dans Why3 lui-même.

2.6.3 Preuve d'identités de matrices par réflexion

L'essence de la preuve par réflexion consiste à exécuter une procédure de décision à l'intérieur de la logique de l'environnement de preuve. Dans notre cas, il s'agit plus précisément d'une procédure de mise en forme canonique pour les expressions matricielles. Nous prouvons ensuite les identités simplement en comparant les formes canoniques. Cependant, nous ne pouvons pas définir une telle procédure directement sur les matrices car cela ne nous donne aucune information sur la structure des expressions impliquées. La première étape est donc de remplacer ces expressions par des arbres de syntaxe abstraite, qui permettent d'effectuer un calcul symbolique.

Plus précisément, nous fabriquons un modèle symbolique des expressions de matrices. Ce modèle symbolique est associée à une interprétation $[\![\cdot]\!]$. qui traduit les expressions vers des matrices. Nous avons également besoin d'un mécanisme de réflexion pour transformer des expressions e_i (au sens du langage logique) en objets symboliques $\hat{e}_i$ et un environnement ρ tels que $[\![\hat{e}_i]\!]_\rho = e_i$. La figure 2.22 résume schématiquement ce procédé.

Nous représentons les expressions par une cascade d'opérateurs symboliques $\hat{+}, \hat{\times}, \ldots$ qui maintiennent les expressions en forme canonique. Autrement dit, nous représentons $(a + b) \times c$ par $(\hat{a}\hat{+}\hat{b})\hat{\times}\hat{c}$. De plus, ce choix d'utiliser une cascade de fonctions plutôt qu'un véritable arbre de syntaxe abstraite permet d'obtenir la forme canonique simplement par calcul des opérateurs symboliques. Cela signifie donc qu'il nous suffit de définir les opérateurs symboliques pour définir la procédure de mise en forme canonique.

Les expressions en forme normale sont représentées par une liste de monômes,

lesquels consistent eux-mêmes en un produit de variables avec un signe explicite. Nous considérons qu'une expression est en forme canonique si les monômes sont triés et qu'il n'y a pas deux monômes opposés dans la liste. Pour faciliter l'élimination des termes opposés et la soustraction, nous avons choisi l'ordre sur les monômes de sorte que deux monômes opposés soient consécutifs. Les variables sont quant à elles représentées par des entiers.

```
type  var = int
type mono = { m_prod: list var; m_pos: bool; }
type expr = { e_body: list mono; e_rows: int; e_cols: int; }
```

Comme nous ne nous sommes pas focalisé sur l'efficacité de la procédure de mise en forme canonique, l'écriture les opérateurs symboliques dans la logique est un simple exercice de programmation fonctionnelle. L'addition correspond à une fusion de liste triées mélangée avec l'élimination des termes opposés. Nous obtenons l'opposé d'une expression en changeant les signes des monômes, car nous avons choisi l'ordre de manière à ce que cela préserve les invariants d'une expression en forme canonique. Enfin, nous réduisons la multiplication à l'addition par développements successifs.

Cependant, nous devons également prouver que notre procédure de décision est correcte. Plus précisément, nous avons besoin de théorèmes de commutation pour remplacer les opérateurs normaux par leurs pendants symboliques. Ces théorèmes ont typiquement la forme $[\![\hat{e}_1 \hat{+} \hat{e}_2]\!]_\rho = [\![\hat{e}_1]\!]_\rho + [\![\hat{e}_2]\!]_\rho$. Remarquons que nous n'avons pas besoin de prouver que nos opérateurs maintiennent la forme canonique des expressions. En effet, cela ne met pas en danger la correction de la preuve obtenue. Le seul risque encouru est que les formes calculées ne soient pas canoniques, c'est-à-dire que deux expressions qui auraient dû avoir la même forme canonique sont transformées en deux expressions syntaxiquement distinctes, auquel cas la preuve échoue. Nous ne maintenons donc aucun invariant particulier sur les formes canoniques, excepté les contraintes dimensionnelles nécessaires pour que l'expression ait un sens.

Les définitions de la fonction d'interprétation $[\![\cdot]\!]_\cdot$ et de l'invariant de bonne formation d'une forme canonique sont standards. Nous donnons ces définitions complètes dans la figure 2.23. Remarquons toutefois que ces définitions dépendent d'un paramètre d'environnement utilisé pour lier les variables à de véritables matrices. Nous modélisons cet environnement par un objet d'ordre supérieur.

En utilisant ces définitions, nous pouvons prouver les théorèmes de commutation sur les opérateurs symboliques. Notons que nous pouvons voir chacun de ces théorèmes comme la post-condition d'un opérateur symbolique. Nous démontrons donc ces opérateurs par le biais de procédures fantômes, dont le corps simule celui de l'opérateur correspondant. Nous pourrions également utiliser des fonctions-lemmes, mais dans notre cas nous allons voir que de simples procédures fantômes suffisent. Par exemple, le code donné en figure 2.24 montre comment nous obtenons le théorème de commutation pour la multiplication. La fonction `lm_merge` implémente l'addition, et `lm_distribute` la multiplication. Nous ajoutons une déclaration `meta` pour enregistrer la fonction parmi celles à dérouler par calcul (cf. section 2.4). Nous obtenons la correction de ces procédures comme une conséquence immédiate des propriétés algébriques fournies par la théorie des matrices, en l'occurrence la distributivité pour

```
function l_mdl (f: int → mat int) (l: list int) : mat int =
  match l with
  | Nil → ... (* Cas non pertinent *)
  | Cons x Nil → f x
  | Cons x q → mul (f x) (l_mdl f q)
  end

function m_mdl (f: int → mat int) (m: mono) : mat int =
  let m0 = l_mdl f m.m_prod in
  if m.m_pos then m0 else opp m0

function lm_mdl (f: int → mat int)
                (r c: int) (l: list mono) : mat int =
  match l with
  | Nil → zero r c
  | Cons x q → add (lm_mdl f r c q) (m_mdl f x)
  end

function e_mdl (f: int → mat int) (e: expr) : mat int =
  lm_mdl f e.e_rows e.e_cols e.e_body

predicate l_vld (f: int → mat int) (r c: int) (l: list int) =
  match l with
  | Nil → false
  | Cons x Nil → rows (f x) = r ∧ cols (f x) = c
  | Cons x q → rows (f x) = r ∧ l_vld f (cols (f x)) c q
  end

predicate m_vld (f: int → mat int) (r c: int) (m: mono) =
  l_vld f r c m.m_prod

predicate lm_vld (f: int → mat int) (r c: int) (l:list mono) =
  match l with
  | Nil → true
  | Cons x q → m_vld f r c x ∧ lm_vld f r c q
  end

predicate e_vld (f:int → mat int) (e:expr) =
  e.e_rows ≥ 0 ∧ e.e_cols ≥ 0 ∧
  lm_vld f e.e_rows e.e_cols e.e_body
```

FIGURE 2.23 – Interprétation et invariant pour les formes canoniques

```
function lm_distribute (l1 l2: list mono) : list mono =
  match l1 with
  | Nil → Nil
  | Cons x q →
    lm_merge Nil (m_distribute x l2) (lm_distribute q l2)
  end

meta rewrite_def function lm_distribute

let rec ghost lm_distribute_ok (f: int → mat int) (r k c: int)
                               (l1 l2: list mono) : list mono
  requires { r ≥ 0 ∧ k ≥ 0 ∧ c ≥ 0 }
  requires { lm_vld f r k l1 ∧ lm_vld f k c l2 }
  ensures  { result = lm_distribute l1 l2 }
  ensures { lm_vld f r c result }
  ensures  { lm_mdl f r c result =
               mul (lm_mdl f r k l1) (lm_mdl f k c l2) }
  variant  { l1 }
= match l1 with
  | Nil → Nil
  | Cons x q →
    lm_merge_ok f r c Nil (m_distribute_ok f r k c x l2)
                          (lm_distribute_ok f r k c q l2)
  end
```

FIGURE 2.24 – Preuve d'un théorème de commutation

la procédure ci-dessus.

Pour terminer, il ne nous manque maintenant plus qu'un ingrédient : le processus de réflexion lui-même, c'est-à-dire le remplacement d'une expression par sa variante symbolique. Malheureusement, il n'y a pour l'instant aucun moyen évident d'effectuer un tel remplacement dans Why3. Pour éviter de réécrire deux fois les mêmes expressions sous des formes différentes, nous construisons alors les versions symboliques des expressions en même temps que nous construisons leur version réelle. Nous arrivons à ce résultat par le mécanisme de l'étiquetage fantôme. Plus précisément, nous ajoutons aux matrices manipulées par notre programme un champ fantôme contenant une version symbolique. Comme ces étiquettes sont fantômes, elles n'ajoutent aucun surcoût à l'exécution.

```
type with_symb = { phy : matrix int;
                   ghost sym : expr; (* réflexion *) }
predicate with_symb_vld (f:int → mat int) (ws:with_symb) =
  e_vld f ws.sym ∧                    (* invariant *)
  e_mdl f ws.sym = ws.phy.mdl ∧       (* mêmes modèles *)
  ws.sym.e_rows = ws.phy.mdl.rows ∧   (* mêmes dimensions *)
  ws.sym.e_cols = ws.sym.mdl.cols
```

```
let add_ws (ghost env:env) (a b:with_symb) : with_symb
  requires { a.phy.mdl.rows = b.phy.mdl.rows }
  requires { a.phy.mdl.cols = b.phy.mdl.cols }
  requires { with_symb_vld env a ∧ with_symb_vld env b }
  ensures { result.phy.mdl = add a.phy.mdl b.phy.mdl }
  ensures { result.sym --> symb_add a.sym b.sym }
  ensures { with_symb_vld env result }
= { phy = add a.phy b.phy;
    sym = ghost symb_add env a.sym b.sym }
```

FIGURE 2.25 – Addition appariée

Nous pouvons alors facilement construire ces paires via des opérateurs qui mettent à jour les étiquettes correctement. Chaque opération arithmétique est naturellement associée à sa version symbolique. De plus, nous pouvons utiliser les post-conditions de ces opérateurs pour fournir directement les instances pertinentes des théorèmes de commutations, et donc obtenir les égalités caractérisant la réflexion sans utiliser les démonstrateurs. Nous obtenons ces instances par appel des procédures fantômes qui les prouvent. Nous donnons par exemple la définition de l'addition appariée sur les matrices et les expressions symboliques dans la figure 2.25

Il reste cependant deux points subtils. Premièrement, remarquons que certaines des expressions impliquées sont construites uniquement en tant qu'objets logiques pour la preuve. Pour notre preuve de l'algorithme de Strassen, il s'agit des expressions correspondant au produit par bloc naturel. Pour exploiter les opérateurs qui maintiennent l'étiquetage, nous construisons ces expressions une fois pour toutes dans le programme, et remplaçons leurs occurrences dans les preuves par les modèles de ces expressions. Cette méthodologie évite bien la duplication, mais semble a priori présenter un énorme coût supplémentaire à l'exécution. Nous évitons également ce surcoût en remarquant que ces opérations supplémentaires n'influent pas sur le résultat du programme, et peuvent donc être rendues intégralement fantômes.

La deuxième subtilité est le traitement des variables. Comme nous avons précisément une variable par bloc, nous apparions leur introduction avec l'extraction de blocs. Cependant, nous devons alors changer l'environnement pour y ajouter cette variable. Pour prendre en compte ces changements, nous représentons l'environnement par un objet mutable, contenant l'environnement effectif et un générateur de symbole. Ce dernier élément assure que l'on génère systématiquement une variable fraîche. Bien entendu, nous utilisons cet environnement exclusivement dans du code fantôme. Nous donnons la définition de l'environnement et de l'extraction de bloc conjointe à l'introduction dans la figure 2.26.

Nous utilisons les étiquettes `"rewrite"` pour enregistrer les définitions de champs comme des règles de réécriture pour la transformation de calcul, ce qui permet de dérouler complètement la définition de l'environnement. Bien que la mutabilité de l'environnement rende la génération de variable facile, cette méthode a le défaut d'invalider a priori les invariants de représentation pour les valeurs déjà construites. En effet, changer l'environnement n'a a priori pas de raison de préserver ces invariants. Nous parvenons néanmoins à les établir de nouveau en les déroulant via la

```
type env = { mutable ev_f: int → mat int; mutable ev_c: int; }
function extends
  (f: int → mat int) (c: int) (v: mat int) : int → mat int =
    λn. if n ≠ c then f n else v
meta rewrite_def function extends
let block_ws (ghost env:env) (a:matrix int)
             (r dr c dc: int) : with_symb
  requires { 0 ≤ r ≤ r + dr ≤ a.mdl.F.rows }
  requires { 0 ≤ c ≤ c + dc ≤ a.mdl.F.cols }
  ensures { let rm = result.phy.mdl in
    rm = block a.mdl r dr c dc ∧ rm.rows = dr ∧ rm.cols = dc }
  ensures { "rewrite"
    result.sym = symb_mat result.phy.mdl (old env.ev_c) }
  ensures { "rewrite"
    env.ev_f = old (extends env.ev_f env.ev_c result.phy.mdl) }
  ensures { "rewrite" env.ev_c = old env.ev_c + 1 }
  ensures { with_symb_vld env.ev_f result }
```

FIGURE 2.26 – Environnement et introduction de variables

transformation de calcul. En effet, cela les remplace par une conjonction d'égalités entre dimensions, dont la preuve est immédiate pour les démonstrateurs.

Pour prouver une égalité `e1 = e2`, nous procédons comme suit :

— Premièrement, nous introduisons un environnement vide pour enregistrer les variables.

— Nous construisons ensuite les expressions `e1` et `e2` via les opérateurs doubles.

— Nous écrivons une assertion qui correspond à l'égalité des deux modèles de `e1` et `e2`, e.g. `e_mdl env.ev_f e1.sym = e_mdl env.ev_f e2.sym`.

— Nous utilisons finalement la transformation de calcul sur l'obligation de preuve associée. Si les deux expressions sont algébriquement égales, les deux termes `e1.sym` et `e2.sym` dans l'assertion ci-dessus sont réduits vers une même valeur `e`. Comme la transformation de calcul détecte les égalités syntaxiques, l'assertion est alors réduite vers `true`, et l'obligation est éliminée sans même être envoyée aux démonstrateurs automatiques.

Cette méthode nous permet de vérifier sans difficulté que le schéma récursif de Strassen coïncide avec la multiplication par blocs naturelle. Nous présentons la partie du code source impactée dans la figure 2.27. L'assertion finale est celle prouvée par calcul. Remarquons que contrairement aux autres opérateurs servant à maintenir l'appariement, la multiplication est introduite localement dans l'algorithme. De cette manière, cet opérateur encapsule les appels récursifs.

```
let ghost f = symb_env () in (* environnement vide *)
let mul_ws (a b: with_symb) : with_symb
  requires { with_symb_vld f.ev_f a ∧ with_symb_vld f.ev_f b }
  requires { a.phy.mdl.F.rows = qr ∧ a.phy.mdl.F.cols = qm }
  requires { b.phy.mdl.F.rows = qm ∧ b.phy.mdl.F.cols = qc }
  ensures { with_symb_vld f.ev_f result }
  ensures { result.phy.mdl = mul a.phy.mdl b.phy.mdl }
  ensures { "rewrite" result.sym = symb_mul a.sym b.sym }
= { phy = strassen a.phy b.phy;
    sym = ghost symb_mul e a.sym b.sym }
in
let a11 = block_ws f a 0 qr 0 qm in
let a12 = block_ws f a 0 qr qm qm in
let a21 = block_ws f a qr qr 0 qm in
let a22 = block_ws f a qr qr qm qm in
let b11 = block_ws f b 0 qm 0 qc in
let b12 = block_ws f b 0 qm qc qc in
let b21 = block_ws f b qm qm 0 qc in
let b22 = block_ws f b qm qm qc qc in
let x1  = mul_ws (add_ws f a11 a22) (add_ws f b11 b22) in
let x2  = mul_ws (add_ws f a21 a22) b11 in
let x3  = mul_ws a11 (sub_ws f b12 b22) in
let x4  = mul_ws a22 (sub_ws f b21 b11) in
let x5  = mul_ws (add_ws f a11 a12) b22 in
let x6  = mul_ws (sub_ws f a21 a11) (add_ws f b11 b12) in
let x7  = mul_ws (sub_ws f a12 a22) (add_ws f b21 b22) in
let m11 = add_ws f (sub_ws f (add_ws f x1 x4) x5) x7 in
let m12 = add_ws f x3 x5 in
let m21 = add_ws f x2 x4 in
let m22 = add_ws f (add_ws f (sub_ws f x1 x2) x3) x6 in
let ghost gm11 = add_ws f (mul_ws e_a11 e_b11)
                          (mul_ws e_a12 e_b21) in
let ghost gm12 = add_ws f (mul_ws e_a11 e_b12)
                          (mul_ws e_a12 e_b22) in
let ghost gm21 = add_ws f (mul_ws e_a21 e_b11)
                          (mul_ws e_a22 e_b21) in
let ghost gm22 = add_ws f (mul_ws e_a21 e_b12)
                          (mul_ws e_a22 e_b22) in
assert {
  e_mdl f.ev_f m11.sym = e_mdl f.ev_f gm11.sym
  ∧ e_mdl f.ev_f m12.sym = e_mdl f.ev_f gm12.sym
  ∧ e_mdl f.ev_f m21.sym = e_mdl f.ev_f gm21.sym
  ∧ e_mdl f.ev_f m22.sym = e_mdl f.ev_f gm22.sym
};
```

FIGURE 2.27 – Code intégrant la preuve des identités

Chapitre 3

Absence de débordements arithmétiques par croissance lente

Nous nous intéressons ici au problème de la preuve d'absence de débordements arithmétiques. Il s'agit d'un cas particulier de vérification à effectuer pour vérifier la sûreté d'un programme. Bien qu'un débordement arithmétique ne cause généralement pas de plantage immédiat, il résulte en une valeur incorrecte, laquelle peut avoir des conséquences fatales durant la suite de l'exécution. Un exemple bien connu de débordement arithmétique est celui de l'algorithme de recherche par dichotomie de la bibliothèque standard de Java, décrit en détail dans le billet de Joshua Bloch *Nearly All Binary Searches and Mergesorts are Broken* [12]. En cas d'usage sur de

grands tableaux, le calcul du point médian de deux entiers `low` et `high` par l'expression `(low + high) / 2` peut déborder au niveau de l'addition, résultant en un indice négatif et donc un plantage lors de l'accès au tableau.

Dans le contexte de la vérification déductive, nous pouvons par exemple modéliser les entiers machine par un type spécifique, en ajoutant aux opérations artihmétiques des pré-conditions garantissant l'absence de débordements. Pour ce qui est de la recherche par dichotomie, nous pouvons aisément vérifier que le calcul du point médian entre `low` et `high` est effectué sans débordement par l'expression `low + (high - low) / 2`.

Cependant, il y a de nombreuses situations dans lesquelles il est très difficile voire impossible de vérifier l'absence de tels débordements. Par exemple, considérons le cas d'un générateur de symboles frais. Un tel générateur est typiquement réalisé par un compteur global, incrémenté à chaque fois qu'un nouveau symbole est demandé. En principe, nous pourrions presque toujours appliquer les approches habituelles, comme celle à base de pré-conditions dans le cadre de la vérification déductive. Cette approche est néanmoins très envahissante, puisqu'il faut propager les pré-conditions de bornes à toutes les fonctions utilisant ce compteur. De plus, nous devrions trouver les bornes adéquates, ce qui est loin d'être toujours évident. Il ne s'agit donc pas d'une approche applicable en pratique. Enfin, pour certaines applications particulières comme les démonstrateurs automatiques, il n'est même pas possible de trouver une borne puisque l'on ne peut pas garantir la terminaison.

Nous observons toutefois que si ce compteur est représenté par un nombre de bits suffisant, par exemple 64, et part de 0, alors il faut effectuer 2^{64} opérations d'incrémentation pour arriver à un débordement effectif. Même si nous effectuions un milliard d'incrémentations par seconde, il nous faudrait plus de 584 ans pour atteindre cette limite. À moins de prévoir une exécution longue de plusieurs centaines d'années, nous pouvons être sûr que ce compteur ne débordera pas. Le point crucial de cet argument est que ce compteur ne peut pas augmenter de plus de 1 par unité de temps, autrement dit que la croissance du compteur est suffisamment lente. En particulier, nous ne pouvons pas effectuer d'additions ou de multiplications arbitraires. Nous montrons dans ce chapitre comment formaliser cet argument, et nous en démontrons la correction. Nous proposons également une implémentation de cette approche dans l'outil Why3.

Il est important de remarquer que cette approche n'est pas un remplacement des méthodes traditionnelles pour la preuve d'absence de débordements arithmétiques, car celles-ci visent un usage différent des entiers. Nous proposons de combiner les différentes méthodes pour vérifier un programme donné. Nous donnons un exemple d'une telle combinaison de méthodes au sein de ce chapitre.

3.1 Champ d'application : exemples

Nous avons déjà mentionné auparavant le cas d'un générateur de symboles. Un générateur de symboles est un programme renvoyant un entier différent à chaque appel. Nous implémentons typiquement un tel programme à l'aide d'une variable globale, comme décrit dans le Programme 3.1. Notons qu'une implémentation plus robuste cacherait d'une manière ou d'une autre la variable globale s, par exemple en utilisant une variable statique en C ou une variable privée dans un langage objet.

Programme 3.1 Générateur de symboles

```
1: s ← 0
2: function GENSYM
3:     s ← s + 1
4:     return s
5: end function
```

Supposons que nous voulions vérifier, d'une manière standard, que l'opération d'incrémentation dans la fonction GENSYM du Programme 3.1 ne fasse pas déborder le compteur. Dans le contexte de la vérification déductive, nous devrions ajouter une pré-condition à GENSYM pour garantir que s soit bien strictement inférieur au plus grand entier représentable. Pour que cette pré-condition soit garantie, nous devons également ajouter une pré-condition du même type à chaque fonction appelant directement ou indirectement GENSYM. De plus, nous devons obtenir les bornes correspondantes, ce qui revient essentiellement à obtenir une borne sur le nombre d'appels à GENSYM. Dans le meilleur des cas, ces annotations additionnelles vont simplement augmenter la taille des spécifications. En pratique, ces annotations peuvent également perturber la vérification, et l'inférence des bornes peut être extrêmement difficile, voire tout simplement impossible.

Cependant, si nous supposons que s est un entier 64-bits non signé, et si nous acceptons de nous restreindre à l'absence de débordements *pendant les premières centaines d'années* de l'exécution du programme, alors nous n'avons plus rien à vérifier pour le programme GENSYM. Nous devons simplement nous assurer que le compteur s n'est pas modifié de manière dangereuse (par exemple, doublé) à un autre emplacement du programme. Cette vérification est particulièrement immédiate si le compteur est rendu inaccessible d'une manière ou d'une autre.

Une grande catégorie d'exemples pour laquelle nous pouvons appliquer la même méthode est celle des programmes calculant la taille d'une structure de données. Par exemple, considérons le calcul de la taille d'une liste chaînée. Nous pouvons implémenter ce calcul aussi bien récursivement (Programme 3.2) que de manière itérative (Programme 3.3). Dans les deux cas, il s'agit d'une suite d'incrémentations pour chaque élément de la liste, en partant de zéro. Comme pour le cas du générateur de symboles, si la longueur est calculée sur 64 bits, il faudra trop de temps pour qu'un débordement puisse avoir lieu.

Programme 3.2 Longueur d'une liste, version récursive

```
1: function LENGTH(l)
2:     if l = null then return 0
3:     else return 1 + LENGTH(l.next)
4:     end if
5: end function
```

Dans cette catégorie, nous pouvons également donner l'exemple du calcul de la taille d'un arbre (Programme 3.4). Cet exemple est légèrement différent car nous effectuons la somme des tailles des sous-arbres. Cependant, cet exemple est encore équivalent à une séquence d'incrémentations par 1. Nous pouvons rendre cette équivalence évidente en réécrivant le programme via un compteur global, correspondant

Programme 3.3 Longueur d'une liste, version itérative

```
1: function LENGTH(l)
2:     len ← 0
3:     while l ≠ null do
4:         len ← len + 1
5:         l ← l.next
6:     end while
7:     return len
8: end function
```

à une accumulation de sommes.

Programme 3.4 Taille d'un arbre

```
1: function SIZE(t)
2:     if t = empty then return 0
3:     else return 1 + SIZE(t.left) + SIZE(t.right)
4:     end if
5: end function
```

Le cas des structures de données avec partage est particulier. Par exemple, intéressons-nous à l'arbre de profondeur h montré sur la droite. Sa taille est de $2^h - 1$. Si nous exploitons le partage pour accélérer le calcul de la taille, en utilisant potentiellement plusieurs fois la même valeur, notre argument de croissance lente ne fonctionne plus. Dans ce cas précis, la réutilisation des résultats permet de simuler l'opération de multiplication par 2, et donc d'atteindre très rapidement la limite de capacité des entiers.

Cependant, si nous ne réutilisons pas les résultats, l'argument temporel fonctionne même pour les arbres avec partage. En effet, si nous appelons la fonction SIZE du Programme 3.4 sur un arbre du type proposé avec h suffisamment grand pour causer un débordement, le programme ne terminera tout simplement pas en pratique. Cela démontre que notre argument est bien basé sur le temps d'exécution, et non sur l'espace occupé par la structure de données.

Une autre catégorie de programmes où nous pouvons appliquer une telle approche est celle des programmes qui stockent des entiers pour la gestion interne d'une structure de donnée. Nous trouvons notamment dans cette catégorie l'information de rang utilisé pour la structure d'*union-find*. Cette structure représente la partition d'un ensemble en classes disjointes, ou de manière équivalente une relation d'équivalence sur cet ensemble. Les classes d'équivalences sont représentées par les arbres d'une forêt, où chaque nœud est équipé d'un pointeur *link* vers son père, ainsi que d'un rang entier r représentant un majorant de la distance du nœud à une feuille. Pour effectuer la réunion de deux classes représentées par les nœuds racines a et b, nous lions le nœud de rang inférieur en dessous du nœud de rang supérieur. Si les deux nœuds ont le même rang, nous effectuons un choix arbitraire et incrémentons de 1 le rang de la racine. Comme les rangs ne sont obtenus que par incréments, il n'y a en pratique pas de risque de débordement arithmétique.

Dans le même esprit, nous pourrions également considérer le cas des indices de De Bruijn [36], une solution technique pour la représentation des lieurs en calcul

Programme 3.5 Union-Find avec rangs

```
1: procedure UNION(a, b)
2:     if a ≠ b then
3:         if a.r < b.r then a.link ← b else b.link ← a end if
4:         if a.r = b.r then a.r ← a.r + 1 end if
5:     end if
6: end procedure
```

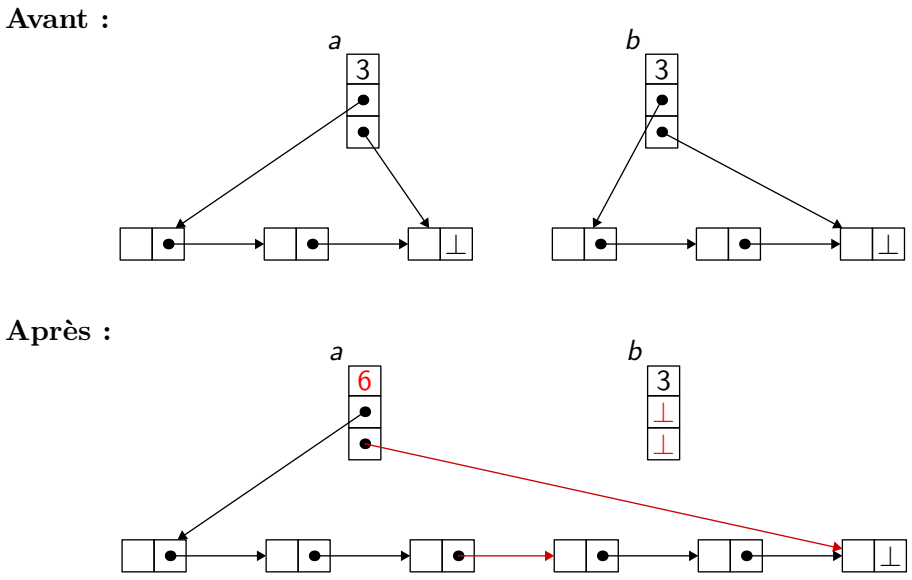

FIGURE 3.1 – Concaténation destructive de listes simplement chaînées

symbolique. Une variable est représentée par un entier, qui compte le nombre de lieurs en dessous desquels elle apparaît. Quand l'on effectue des substitutions, les indices doivent être incrémentés lors du passage sous les lieurs. Ces incréments n'ont lieu que un par un, et ne posent donc pas de risque pratique de débordement.

Enfin, un dernier exemple dans cette catégorie est celui des listes simplement chaînées avec concaténation destructive, comme présenté dans la figure 3.1. Chaque liste contient sa longueur et deux pointeurs, respectivement vers son premier et dernier élément. La concaténation est effectuée en faisant pointer le dernier élément de la première liste sur le premier élément de la seconde. Ensuite, la longueur de la première liste et son dernier élément sont mis à jour, puis la seconde liste est invalidée en assignant ses pointeurs à *null*.

Programme 3.6 Concaténation destructive de listes simplement chaînées

```
1: procedure APPEND(a, b)
2:     assert a ≠ b ∧ a.first ≠ null ∧ b.first ≠ null
3:     a.size ← a.size + b.size
4:     a.last.next ← b.first
5:     a.last ← b.last
6:     b.first ← null
7:     b.last ← null
8: end procedure
```

Comme dans l'exemple du calcul de taille d'arbre, l'addition effectuée dans le Programme 3.6 n'est pas dangereuse. En effet, la taille de chacune des listes est limitée par le temps passé à les construire. Comme l'une des listes est invalidée par la concaténation, nous déplaçons les « crédits temps » gagné par sa construction dans la longueur de la liste agrandie, ce qui ne pose donc pas de risque de débordement en temps raisonnable.

Programme 3.7 Nombre de solutions au problème des N reines

```
1: function N-QUEENS(b)
2:     if l'échiquier b contient N reines then return 1
3:     else
4:         r ← 0
5:         for n'importe quel échiquier valide b′ obtenu par ajout
6:                 d'une reine sur la première ligne vide de l'échiquier b do
7:             r ← r + N-QUEENS(b′)
8:         end for
9:         return r
10:    end if
11: end function
```

Enfin, une dernière catégorie est celle des programmes combinatoires qui comptent le nombre de solutions à un problème par énumération exhaustive. Prenons par exemple le cas du problème des N reines, où l'on cherche à obtenir le nombre de moyens de placer N reines sur un échiquier de taille $N \times N$ sans qu'une reine puisse en attaquer une autre. Nous donnons dans le Programme 3.7 le schéma d'une fonction N-QUEENS qui ajoute des reines à un échiquier jusqu'à trouver une solution. Les nombres manipulés par ce programme sont bornés par le nombre de solutions déjà énumérées, et donc par le temps d'exécution.

Un aspect important de cet exemple est que l'on ne connaît pas de borne raisonnable du résultat, et donc il n'est pas possible de vérifier l'absence de débordements arithmétiques sans d'une manière ou d'une autre exécuter un programme du même type.

3.2 Solution proposée

Nous proposons la méthode suivante pour vérifier l'absence de débordements arithmétiques dans les programmes du type de ceux énumérés dans la section pré-

cédente. Tout d'abord, nous identifions au sein du programme les valeurs entières pour lesquelles nous voulons appliquer notre approche, par exemple parce qu'il n'est pas réaliste d'expliciter des bornes pour ces valeurs. Pour toutes ces valeurs, représentées par un entier machine sur n bits, nous relâchons la propriété d'absence de débordements arithmétiques à l'absence durant les 2^n premières étapes d'exécution. Ces étapes doivent correspondre à une sémantique opérationnelle raisonnable, de manière à pouvoir les traduire en un nombre proportionnel de cycles du processeur.

Nous laissons à l'utilisateur de la méthode la responsabilité de vérifier que la propriété relâchée est suffisante, c'est-à-dire qu'une durée d'exécution de cet ordre de grandeur dépasse toute utilisation potentielle. Nous pensons que les entiers 64 bits sont actuellement parfaits pour notre cas d'utilisation. En effet, ces entiers sont supportés nativement par les processeurs modernes, tandis que nous n'avons jamais entendu parler de la moindre application où un programme séquentiel pourrait s'exécuter pendant plusieurs centaines d'années, c'est-à-dire assez longtemps pour pouvoir causer un débordement arithmétique juste avec des incrémentations par 1.

Dans le but d'assurer la propriété de sûreté relâchée pour les valeurs entières sélectionnées, nous restreignons les opérations disponibles pour manipuler ces entiers. Nous limitons ainsi la vitesse de croissance de ces valeurs. Selon le langage de programmation et les méthodes de vérification, nous avons de nombreux moyens de garantir ces restrictions. Par exemple, nous pouvons introduire de nouveaux types de données pour ces « entiers restreints », et laisser le système de types vérifier que seules les opérations autorisées sont employées. Nous pourrions également effectuer une analyse statique pour détecter les valeurs respectant ces restrictions. Remarquons au passage que cette analyse est une forme d'inférence de types.

Nous démontrons ici l'approche par typage. Nous allons décrire deux types d'entiers restreints (qui doivent bien entendu être compilés vers des entiers machine). Les premiers sont les *entiers de Peano*, où seule l'opération d'incrémentation est permise pour créer de nouveaux entiers. En utilisant les entiers de Peano, nous pouvons garantir l'absence de débordement arithmétique dans les programmes 3.1 (générateur de symboles), 3.2 et 3.3 (longueur d'une liste), et 3.5.

Nous introduisons la seconde classe dans le but de permettre certaines additions. Essentiellement, une addition ne pose pas de danger si ses arguments ne sont additionnés qu'une seule fois. Nous introduisons donc les *entiers à usage unique*, une classe d'entiers où l'addition invalide ses arguments. En utilisant les entiers à usage unique, nous pouvons garantir l'absence de débordement dans les programmes restant, c'est-à-dire les programmes 3.4 (taille d'un arbre), 3.6 (longueur d'une liste) et 3.7 (N reines).

3.2.1 Entiers de Peano

Les entiers de Peano sont introduits par un nouveau type *peano*. Ces entiers sont modélisés par des entiers mathématiques. Essentiellement, les opérations disponibles se réduisent à la constante *zero* et à l'opération successeur *succ*, d'où le nom choisi pour ces entiers.

$$\begin{aligned} &\mathit{zero} : \mathit{peano} \\ &\mathit{succ}(p : \mathit{peano}) : \mathit{peano} \end{aligned}$$

Cependant, d'autres opérations sont possibles sans pour autant invalider l'argument de vitesse de croissance limitée. Par exemple, nous pouvons avoir une fonction

de conversion vers des entiers de précision arbitraire, renvoyant le modèle d'un entier de Peano :

$$to\_int(p : peano) : \mathbb{Z}$$

ou encore la construction d'un entier de Peano depuis un entier arbitraire x, qui est valide tant que x est positif et dominé par un entier de Peano déjà construit :

$$cap(x : \mathbb{Z},\ p : peano) : peano$$

Cette liste d'opérations n'est pas exhaustive. Par exemple, il est tout à fait possible de comparer deux entiers de Peano, de calculer leur minimum ou leur maximum, etc.

3.2.2 Entiers à usage unique

Les entiers à usage unique sont introduits par un nouveau type *onetime*. Pour représenter l'idée que les entiers à usage unique sont bien utilisés de cette manière, chaque entier à usage unique est équipé d'un drapeau booléen en plus de sa valeur entière. Le drapeau de validité est mutable, et est invalidé lorsque l'entier à usage unique est utilisé pour en créer un nouveau. La valeur elle-même de l'entier à usage unique est immuable, et les opérations permises génèrent de nouveaux entiers à usage unique, distincts de tous ceux générés jusqu'à présent.

Comme pour les entiers de Peano, les entiers à usage unique disposent d'opérations correspondant à la constante nulle et au successeur :

$$fresh\_zero() :\ onetime$$
$$succ(p : onetime) : onetime$$

Cependant, ces opérations se comportent de manière différente. Par exemple, *fresh_zero* n'est pas une constante, puisque elle doit renvoyer un entier à usage unique différent à chaque fois. La fonction *succ* est également différente de son pendant pour les entiers de Peano. Elle ne fonctionne que pour des arguments valides, et les invalide pour garantir la validité de son résultat.

L'intérêt principal des entiers à usage unique est la fonction d'addition :

$$add(x : onetime,\ y : onetime) : onetime$$

Comme la fonction *succ*, la fonction *add* prend en entrée des entiers à usage unique valides et les invalide. De plus, ces deux entiers doivent être distincts, pour éviter qu'il soit possible de directement doubler un entier à usage unique.

Enfin, nous fournissons également une opération non destructive de conversion vers les entiers de Peano :

$$to\_peano(x : onetime) : peano$$

Cette opération n'a pas besoin d'invalider son argument puisqu'il n'y a aucun moyen de reconstruire un entier à usage unique depuis un entier de Peano.

Remarquons par ailleurs que nous n'avons fourni aucune opération pour inspecter le drapeau de validité. Ce choix nous permet de compiler les entiers à usage unique vers des entiers machine, où le drapeau n'existe pas. Pour les mêmes raisons, nous n'avons fourni aucune fonction permettant de vérifier si deux entiers à usage unique occupent le même espace mémoire.

v	::=	$\langle p \rangle$	entier de Peano ($p \in \mathbb{N}$)
	|	a	adresse mémoire
	|	$\bot \mid \top$	valeur booléenne
	|	n	entier de taille arbitraire ($n \in \mathbb{Z}$)
e	::=	x	variable
	|	v	valeur
	|	$succ(e)$	opération successeur
	|	$fresh\_zero()$	création d'un entier à usage unique
	|	$add(e, e)$	addition pour les entiers à usage unique
	|	$to\_int(e)$	conversion Peano/$\mathbb{Z}$
	|	$cap(e, e)$	conversion partielle $\mathbb{Z}$/Peano
	|	$\{f = e, \ldots, f = e\}$	construction d'un nouvel enregistrement
	|	$e.f$	accès à un champ
	|	...	connecteurs booléens, opérateurs arithmétiques sur $\mathbb{Z}$, etc.
s	::=	$skip$	skip
	|	$x \leftarrow e$	affectation de variable
	|	$e.f \leftarrow e$	affectation d'un champ
	|	$s\,;s$	séquence
	|	`if` e `then` s `else` s	conditionnelle
	|	`while` e `do` s `done`	boucle

FIGURE 3.2 – Syntaxe abstraite pour un petit langage de programmation

3.2.3 Formalisation

Pour prouver la correction de notre approche, nous considérons un petit langage `While` avec des enregistrements alloués sur le tas. La figure 3.2 présente la syntaxe abstraite de ce langage, qui distingue valeurs v, expressions e, et commandes s. Les valeurs sont ou bien des entiers de Peano, ou bien des adresses mémoire, ou bien des booléens, ou bien des entiers de précision arbitraire. L'ensemble des adresses est supposé être infini. Un entier à usage unique est représenté par un enregistrement à deux champs : un champ otP contenant sa valeur en tant qu'entier de Peano ; et un champ otV contenant son drapeau de validité. L'opération de conversion d'un entier à usage unique vers un entier de Peano est donc simplement l'accès au champ otP. L'accès au champ otV est interdit, pour les raisons expliquées précédemment.

Nous adjoignons à notre langage une sémantique opérationnelle à petits pas. Un tas Σ est défini comme une fonction à support fini des paires adresse/nom de champ vers les valeurs. Nous définissons alors un état du programme comme un triplet (V, Σ, s), où V est une fonction des variables vers les valeurs, et s une commande représentant le reste du code à exécuter. Nous définissons notre sémantique à petits pas par le biais d'une relation d'exécution en un pas, définie aussi bien pour les expressions que pour les commandes, et notée $V, \Sigma, s \to V', \Sigma', s'$, où s/s' peuvent être

des commandes ou des expressions. Cette relation est définie de manière standard, par le biais de règles de réduction de tête (Fig. 3.3–3.5) et de contextes de réduction. Remarquons au passage que les règles de réduction interdisent la construction et la mutation directe d'entiers à usage unique (voir règles ALLOC et MEM-ASSIGN).

De manière informelle, nous pouvons énoncer le théorème principal de la manière suivante. Durant les n premières étapes d'exécution d'un programme dont l'état initial ne contient pas d'entiers de Peano différents de 0, la valeur des entiers de Peano/à usage unique ne dépassera pas n. Pour rendre cet énoncé formel, nous introduisons la notion d'état programme borné :

Définition 3.1. Un état (V, Σ, s) est n-borné si :
- tout entier de Peano/à usage unique apparaissant dans l'état est positif ou nul.
- tout entier de Peano apparaissant dans l'état, y compris parmi les valeurs appartenant au programme s, n'est pas plus grand que n ;
- la somme de tous les entiers à usage unique valides alloués dans Σ ne dépasse pas n.

Nous pouvons alors énoncer le résultat principal. Nous utilisons la notion d'état 0-borné pour représenter le fait que tous les entiers de Peano, constantes comprises, ont pour valeur 0 au début de l'exécution d'un programme.

Théorème 3.2. *Soit V, Σ, s un état 0-borné. Alors tout état V', Σ', s' atteignable après n étapes d'exécution est n-borné.*

Démonstration. Nous procédons par récurrence sur n. Comme le cas de base est trivial, il suffit de montrer que pour tout état n-borné V, Σ, s qui se réduit en une étape en un état V', Σ', s', l'état V', Σ', s' est $(n+1)$-borné. Il nous suffit maintenant de faire une disjonction de cas sur la règle de réduction de tête :
- PEANO-SUCC : comme l'entier de Peano est borné par n, son successeur est borné par $n+1$. Les autres parties de l'état ne changeant pas, l'état résultant est bien $(n+1)$-borné.
- ONE-TIME-SUCC : nous appliquons le même raisonnement que pour les entiers de Peano, à ceci près que la somme totale des entiers à usage unique valides change aussi. Cependant, elle n'augmente que de un, donc l'état résultant est bien $(n+1)$-borné.
- FRESH_ZERO : l'entier à usage unique nouvellement introduit valant 0, la somme totale reste inchangée et donc dominée par n. Comme $0 \leq n$, l'état résultant est bien n-borné, et donc $(n+1)$-borné.
- CAP : par hypothèse, cette opération crée un entier de Peano plus petit qu'un entier existant, et donc respectant la borne.
- ADD : la somme totale des entiers à usage unique valides reste inchangée, et donc ne dépasse pas n. Comme l'entier à usage unique résultant fait partie de cette somme, et que tous les termes de la somme sont positifs, il est également inférieur ou égal à n. En particulier, cette règle de réduction préserve le caractère n-borné.
- MEM-ASSIGN : Les contraintes sur cette règle font qu'elle n'introduit aucun entier restreint et qu'elle n'a pas le droit de les modifier. En particulier, elle préserve le caractère n-borné.

$$\text{VAR}\ \frac{}{V,\ \Sigma,\ x\ \rightarrow\ V,\ \Sigma,\ V(x)} \qquad \text{FIELD}\ \frac{a.f \in \Sigma \qquad f \neq otV}{V,\ \Sigma,\ a.f\ \rightarrow\ V,\ \Sigma,\ \Sigma(a.f)}$$

$$\text{ALLOC}\ \frac{a \notin \Sigma \qquad \forall i.\ f_i \notin \{otP, otV\}}{V,\ \Sigma,\ \{f_1 = v_1, \ldots, f_n = v_n\}\ \rightarrow\ V,\ \Sigma[a.f_1 \leftarrow v_1, \ldots, a.f_n \leftarrow v_n],\ a}$$

FIGURE 3.3 – Règle de réduction pour les expressions

$$\text{SKIP}\ \frac{}{V,\ \Sigma,\ (\mathit{skip}\,;s)\ \rightarrow\ V,\ \Sigma,\ s} \qquad \text{ASSIGN}\ \frac{}{V,\ \Sigma,\ x \leftarrow v\ \rightarrow\ V[x \leftarrow v],\ \Sigma,\ \mathit{skip}}$$

$$\text{MEM-ASSIGN}\ \frac{a.f \in \Sigma \qquad f \notin \{otP, otV\}}{V,\ \Sigma,\ a.f \leftarrow v\ \rightarrow\ V,\ \Sigma[a.f \leftarrow v],\ \mathit{skip}}$$

$$\text{IF-TRUE}\ \frac{}{V,\ \Sigma,\ (\texttt{if}\ \top\ \texttt{then}\ s\ \texttt{else}\ s')\ \rightarrow\ V,\ \Sigma,\ s}$$

$$\text{IF-FALSE}\ \frac{}{V,\ \Sigma,\ (\texttt{if}\ \bot\ \texttt{then}\ s\ \texttt{else}\ s')\ \rightarrow\ V,\ \Sigma,\ s'}$$

$$\text{WHILE}\ \frac{}{V,\ \Sigma,\ (\texttt{while}\ e\ \texttt{do}\ s\ \texttt{done})\ \rightarrow\ V,\ \Sigma,\ (\texttt{if}\ e\ \texttt{then}\ s\,;\ \texttt{while}\ e\ \texttt{do}\ s\ \texttt{done}\ \texttt{else}\ \mathit{skip})}$$

FIGURE 3.4 – Règle de réduction pour les commandes

$$\text{PEANO-SUCC}\ \frac{}{V,\ \Sigma,\ \mathit{succ}(\langle p\rangle)\ \rightarrow\ V,\ \Sigma,\ \langle p+1\rangle}$$

$$\text{ONE-TIME-SUCC}\ \frac{\Sigma(a) = \{otP = \langle p\rangle,\ otV = \top\} \qquad a' \notin \Sigma}{V,\ \Sigma,\ \mathit{succ}(a)\ \rightarrow\ V,\ \Sigma[a.otV \leftarrow \bot,\ a'.otP \leftarrow \langle p+1\rangle,\ a'.otV \leftarrow \top],\ a'}$$

$$\text{FRESH\_\_ZERO}\ \frac{a \notin \Sigma}{V,\ \Sigma,\ \mathit{fresh\_\_zero}()\ \rightarrow\ V,\ \Sigma[a.otP \leftarrow \langle 0\rangle,\ a.otV \leftarrow \top],\ a}$$

$$\text{TO\_\_INT}\ \frac{}{V,\ \Sigma,\ \mathit{to\_\_int}(\langle n\rangle)\ \rightarrow\ V,\ \Sigma,\ n} \qquad \text{CAP}\ \frac{0 \leq n \leq m}{V,\ \Sigma,\ \mathit{cap}(n, \langle m\rangle)\ \rightarrow\ V,\ \Sigma,\ \langle n\rangle}$$

$$\text{ADD}\ \frac{\Sigma(a_1) = \{otP = \langle n\rangle,\ otV = \top\} \quad \Sigma(a_2) = \{otP = \langle m\rangle,\ otV = \top\} \quad a_1 \neq a_2 \quad a_3 \notin \Sigma}{V,\ \Sigma,\ \mathit{add}(a_1, a_2)\ \rightarrow\ V,\ \Sigma[a_1.otV \leftarrow \bot,\ a_2.otV \leftarrow \bot,\ a_3.otP \leftarrow \langle n+m\rangle,\ a_3.otV \leftarrow \top],\ a_3}$$

FIGURE 3.5 – Règles de réduction pour les opérations sur les entiers restreints

$$
\begin{array}{rcl}
C_e & ::= & \Box \mid succ(C_e) \\
& \mid & add(C_e, e) \mid add(e, C_e) \\
& \mid & int(C_e) \\
& \mid & cap(C_e, e) \mid cap(e, C_e) \\
& \mid & \{f = e, \ldots, f = C_e, \ldots, f = e\} \\
& \mid & C_e.f \\
& \mid & \ldots \quad \text{(autres contextes de réduction habituels, pour les connecteurs booléens, etc)}
\end{array}
$$

$$
\begin{array}{rcl}
C_s & ::= & \Box \mid x \leftarrow C_e \\
& \mid & C_e.f \leftarrow e \mid e.f \leftarrow C_e \\
& \mid & C_s; s \\
& \mid & \texttt{if } C_e \texttt{ then } s \texttt{ else } s \\
& \mid & \texttt{while } C_e \texttt{ do } s \texttt{ done}
\end{array}
$$

FIGURE 3.6 – Contextes de réduction

— ALLOC : comme cette règle ne permet pas de créer d'entiers à usage unique, et n'en modifie pas, elle préserve le caractère n-borné.
— autres règles : même raisonnement que pour MEM-ASSIGN/ALLOC, à ceci près que ces règles n'ont pas été contraintes pour obtenir ces propriétés.

□

Remarquons par ailleurs que dans la preuve ci-dessus, la borne ne peut augmenter qu'après l'application d'une opération successeur. Nous obtenons donc le corollaire suivant :

Corollaire 3.3. *Pour toute exécution $V, \Sigma, s \rightarrow^* V', \Sigma', s'$ où l'état initial est 0-borné, l'état final est borné par le nombre d'étapes successeurs employées dans l'exécution (règles* PEANO-SUCC *et* ONE-TIME-SUCC*).*

3.3 Implémentation dans l'outil Why3

3.3.1 Modélisation

Nous proposons au sein de Why3 une implémentation des entiers restreints à base de typage. Les entiers de Peano et à usage unique sont introduits par les deux types suivants :

```
type peano = private { ghost v : int }

type onetime = private {
  ghost v : int;
  ghost mutable valid : bool
}
```

Le type `int` correspond au type des entiers mathématiques de précision arbitraire, autrement dit $\mathbb{Z}$. Nous introduisons les deux types comme des types privés, ce qui revient à donner un modèle à ces deux types. En effet, les valeurs des types privés

```
val constant zero : peano
  ensures { result.v = 0 }
val succ (x: peano) : peano
  ensures { result.v = x.v + 1 }
val to_int (x: peano) : int
  ensures { result = x.v }
val cap (x: int) (p: peano) : peano
  requires { 0 ≤ x ≤ p.v }
  ensures  { result.v = x }
val zero () : onetime
  ensures { result.v = 0 ∧ result.valid }
val succ (x: onetime) : onetime
  requires { x.valid }
  writes   { x }
  ensures  { result.v = x.v + 1 ∧ result.valid ∧ not x.valid }
val to_peano (x:onetime) : peano
  ensures { result.v = x.v }
val add (x y: t) : t
  requires { x.valid ∧ y.valid }
  writes   { x, y }
  ensures  { result.v = x.v + y.v ∧ result.valid }
  ensures  { not x.valid ∧ not y.valid }
```

FIGURE 3.7 – Fonctions abstraites sur les entiers restreints

ne peuvent pas être construites ni modifiées directement. Nous pouvons néanmoins accéder aux champs déclarés, ce qui permet d'écrire des spécifications. Comme ces champs sont fantômes, leurs valeurs ne peuvent pas être utilisées directement pour affecter le déroulement du programme. Le code client doit donc utiliser les fonctions abstraites fournies avec le type privé pour pouvoir manipuler les données associées, ce qui correspond exactement au type de restrictions que nous souhaitons imposer.

Nous fournissons également les opérations qui doivent accompagner les entiers restreints (cf. figure 3.7). La spécification de ces opérations correspond précisément au comportement annoncé. Notons que dans le cas particulier de l'addition d'entiers à usage unique, nous n'avons pas eu besoin d'écrire d'hypothèse de séparation pour les arguments de `add`. En effet, Why3 suppose par défaut que tous les arguments vivent dans des régions séparées, et vérifie cette séparation au moment de l'appel.

Nous pouvons utiliser ces opérations pour re-définir la plupart des opérateurs. Par exemple, nous pouvons implémenter l'addition de deux entiers de Peano comme suit :

```
val add (p q r: peano) : peano
  requires { 0 ≤ p.v + q.v ≤ r.v }
  ensures  { result.v = p.v + q.v }
= cap (to_int p + to_int q) r
```

Remarquons que nous avons besoin pour effectuer l'addition d'un troisième argu-

ment, donnant une borne supérieure sur le résultat de l'addition. Celui-ci permet de justifier le retour vers les entiers de Peano par le biais de l'opération `cap`. Nous pouvons de manière similaire implémenter les opérateurs de comparaison, ce qui ne demande pas de borne supérieure additionnelle.

Pour obtenir du code exécutable, nous utilisons la procédure d'extraction de code de Why3 vers le langage OCaml. Celle-ci est dirigée par des fichiers de configurations appelés *drivers*, qui affectent des définitions OCaml aux symboles Why3 non définis dans le programme WhyML. Nous fournissons avec la spécification des entiers restreints un driver qui traduit les entiers restreints vers les entiers natifs 63 bits [1] pour la version 64-bits d'OCaml.

Nous utilisons également le driver pour envoyer les opérateurs définis à l'aide de `to_int` et `cap` vers les opérateurs équivalents sur les entiers machine. Nous évitons ainsi le coût de conversion vers des entiers arbitraires tout en gardant le même comportement. Par exemple, la fonction `add` pour les entiers de Peano est traduite vers la fonction OCaml `(fun p q _ → p + q)`. Notons par ailleurs que la traduction ne contient aucune assertion dynamique pour vérifier la sécurité de la fonction. L'argument de borne est ignoré, la vérification déductive du programme WhyML assurant de toute façon que la pré-condition de `add` est vérifiée dans toutes les exécutions possibles.

De la même manière, l'opération `add` pour les entiers à usage unique est extraite vers `(+)`. Remarquons que le drapeau de validité n'apparaît pas dans le code extrait. En effet, ce drapeau ne sert qu'à vérifier la sécurité des appels aux opérateurs sur les entiers à usage unique. Il n'a aucune influence sur le comportement du code extrait, et est donc éliminé au même titre que les valeurs fantômes.

3.3.2 Exemple : *N* reines

Pour démontrer l'utilisation de notre approche, nous reprenons l'exemple du Programme 3.7 en Why3. Nous montrons dans la figure 3.8 une implémentation d'un algorithme de recherche exhaustive pour le problème des N reines, via la fonction récursive `count_bt_queens`. Son argument `solutions` est un compteur entier de Peano, qui est incrémenté à chaque fois qu'une nouvelle solution est trouvée. Nous omettons les annotations de spécification dans le but de rester brefs.

Ce programme utilise deux types d'entiers machine. Le type `int63` est utilisé pour les indices de tableau et les coordonnées des reines. Nous prouvons statiquement l'absence de débordement pour ces valeurs par les méthodes traditionnelles. Par exemple, à la ligne 12, une obligation de preuve est générée pour garantir que `q` peut être incrémenté sans débordement. Cette obligation est facilement vérifiée grâce au test de la boucle.

Nous utilisons également les entiers de Peano pour compter le nombre de solutions, ce qui correspond à l'incrémentation de la ligne 22. Il n'y a pas d'obligations de preuve générées pour cette opération. C'est tant mieux, car nous ne serions pas capable de les vérifier. En effet, nous n'avons *a priori* aucun majorant sur le nombre de solutions, excepté le majorant évident $N!$ qui est bien trop grand. De plus, toute borne obtenue automatiquement sera probablement du même ordre de grandeur.

1. OCaml réserve un bit pour la gestion de la mémoire

```
1   exception Inconsistent
2
3   let check_is_consistent (board: array int63) (pos: int63)
4   = try
5       let q = ref (of_int 0) in
6       while !q < pos do
7         let bq   = board[!q]   in
8         let bpos = board[pos] in
9         if bq        = bpos     then raise Inconsistent;
10        if bq - bpos = pos - !q then raise Inconsistent;
11        if bpos - bq = pos - !q then raise Inconsistent;
12        q := !q + of_int 1
13      done;
14      True
15    with Inconsistent →
16      False
17    end
18
19  let rec count_bt_queens (solutions: ref peano)
20      (board: array int63) (n: int63) (pos: int63)
21  = if eq pos n then
22      solutions := Peano.succ !solutions
23    else
24      let i = ref (of_int 0) in
25      while !i < n do
26        board[pos] ← !i;
27        if check_is_consistent board pos then
28          count_bt_queens solutions board n (pos + of_int 1);
29        i := !i + of_int 1
30      done
31
32  let count_queens (board: array int63) (n: int63) : peano
33  = let solutions = ref (Peano.zero ()) in
34    count_bt_queens solutions board n (of_int 0);
35    !solutions
```

FIGURE 3.8 – Problème des N reines en Why3

Bien que ces deux types d'entiers (`int63` et `peano`) soient traités de manière différentes du point de vue de la preuve, ils sont au final tous deux extraits par Why3 vers le même type OCaml, à savoir le type des entiers machine `int` pour la version 64 bits d'OCaml.

3.3.3 Exemple : hauteurs dans un arbre de type AVL

Comme nous l'avons annoncé, notre approche nous permet de vérifier l'absence de débordement pour les entiers servant à la gestion interne d'une structure de données. Nous démontrons maintenant cette affirmation de manière pratique, en utilisant notre approche pour les hauteurs des sous-arbres d'un arbre de type AVL[3]. Nous reprenons pour cela un développement Why3 que nous avions effectué auparavant sur la vérification automatique de structures de données basées sur les arbres AVL [22]. Dans ce développement, nous avons vérifié plusieurs structures de données, toutes obtenues comme instance d'une structure générale d'arbre de type AVL.

Les arbres AVL sont des arbres binaires équilibrés par le critère suivant. Au niveau de tout nœud, la différence de hauteur entre le sous-arbre droit et le sous-arbre gauche est d'au plus 1. L'équilibrage est garanti par le biais de rotations de la structure d'arbre. Pour pouvoir vérifier rapidement les hauteurs respectives de deux sous-arbres, la hauteur de chaque sous-arbre est typiquement stockée au niveau de chaque nœud. Nous recalculons donc celle-ci quand nécessaire par le maximum des hauteurs des sous-arbres, plus 1. En particulier, ces opérations appartiennent au sous-ensemble des opérations autorisées pour les entiers de Peano.

Nous avons donc repris le développement Why3 sur les arbres AVL, en remplaçant le type utilisé pour les hauteurs, celui des entiers arbitraires, par celui des entiers de Peano. À part le changement de nom des opérations sur les hauteurs, nous n'avons observé aucun impact significatif sur le développement lui-même. En particulier, les obligations de preuve générées par Why3 sont restées pratiquement les mêmes. Le seul changement observé est au niveau des variables représentant les hauteurs, donc l'utilisation a été changé de `h` à `h.v`. Pourtant, cette simple substitution permet d'utiliser des entiers machine pour représenter les hauteurs, et ce sans risque de débordement arithmétique.

Notons qu'il serait beaucoup plus difficile et pénible de vérifier l'absence de débordements arithmétiques de manière traditionnelle. En effet, nous devrions alors ajouter à toutes les opérations sur les structures de données obtenues une pré-condition qui empêche la structure obtenue de grandir au-delà d'une certaine taille. Nous devrions ensuite trouver une méthode au niveau de son utilisation pour garantir que cette taille ne peut pas être atteinte. D'un autre côté, l'emploi des entiers de Peano permet d'éviter la moindre modification du contrat des opérations, et donc tout coût de vérification additionnel pour l'utilisateur de la structure.

Enfin, remarquons que dans notre cas précis nous ne pouvons pas stocker les hauteurs sur des entiers significativement plus petits. Comme les hauteurs sont logarithmiques en la taille de l'arbre, nous nous attendrions pourtant à pouvoir stocker les hauteurs sur de petits entiers. C'est effectivement un argument valable en l'absence de partage, ce qui ne peut pas avoir lieu dans le cadre des arbres binaires de recherche. Cependant, nous vérifions d'autres structures de données à base d'arbres AVL, dont certaines permettent d'exploiter le partage de manière effective. C'est

typiquement le cas des listes à accès aléatoire, dont la concaténation est effectuée en temps logarithmique.

En effet, en utilisant des concaténations répétées pour doubler la taille à chaque étape, nous pouvons construire en temps $O(N \log(N))$ un arbre de type AVL de taille 2^N, et donc de hauteur au moins N. En exploitant certaines opérations internes, nous pourrions même construire un arbre de hauteur exactement N en temps $O(N)$. Nous pouvons donc créer des hauteurs d'une valeur très proche du temps d'exécution. La borne donnée par les entiers de Peano étant alors une borne quasi-optimale pour les hauteurs, nous ne pouvons pas réduire la taille réservée au stockage de celles-ci.

3.4 Limitations et perspectives

Nous nous intéressons maintenant aux limitations et défauts de l'approche présentée, ainsi qu'aux méthodes possibles pour améliorer l'approche, notamment en éliminant ces défauts.

3.4.1 Conversion avec les entiers de taille fixe

Bien que les entiers restreints et les entiers de taille fixe correspondants soient en pratique représentés de la même manière, nous n'avons fourni aucun moyen de passer d'une catégorie à l'autre. Nous ne pouvons bien entendu pas convertir un entier machine arbitraire vers un entier restreint, puisque cela peut briser l'argument de croissance lente. C'est la raison pour laquelle la fonction `cap` demande une borne sur l'entier converti. Cependant, il ne semble a priori y avoir aucune raison pour laquelle nous ne permettons pas de faire l'inverse. Nous affirmons que cela pose plus de problèmes que cela n'en résout.

Premièrement, une telle fonction serait simplement incorrecte dans le cadre particulier de notre implémentation Why3. En effet, les entiers de Peano peuvent être utilisés dans le code fantôme, ce qui permet de créer des entiers de Peano fantômes de taille arbitraire en un temps d'exécution nul (puisque le code fantôme est effacé par l'extraction de code). Cependant, une fonction de conversion vers les entiers machine permettrait de montrer qu'un tel entier est borné, et donc de montrer une contradiction.

De plus, même en l'absence de code fantôme, une telle fonction serait plus problématique que bénéfique. En effet, bien que nous considérions effectivement les étapes d'exécution suffisamment lointaines comme impossible à atteindre, à cause de notre critère de correction relâché, il semblerait étrange que le système valide la correction fonctionnelle, terminaison comprise, de

```
let p = ackermann_with_peano 4 2 (* = 2^65536 − 3 *) in
let n = to_int64 p in
assert { n > max_int64 ≥ n }
```

sans signaler le moindre problème. Enfin, ne pas préciser la taille effective des entiers restreints rend l'implémentation plus modulaire. Nous pouvons ainsi adapter la taille des entiers restreints au temps maximum d'exécution du programme lors de sa compilation.

Notons que dans le cas la conversion d'entiers de n'importe quelle taille fixée vers les entiers de Peano, nous pouvons fournir une fonction partielle en adaptant la fonction `cap`.

3.4.2 Arguments de bornes

Un défaut apparent de notre approche concerne les opérations arithmétiques arbitraires. Pour pouvoir effectuer celles-ci sans risque, nous sommes obligés de passer un argument supplémentaire qui majore le résultat. Cependant, cet argument n'est en pratique pas utilisé par la fonction. Il est donc tentant de chercher d'une manière ou d'une autre à l'éliminer. Dans notre implémentation Why3, la manière la plus simple serait de le rendre fantôme. Par exemple, la fonction `add` pour les entiers de Peano deviendrait :

```
val bad_add (p q: peano) (ghost r: peano) : peano
  requires { 0 ≤ p.v + q.v ≤ r.v }
  ensures { result.v = p.v + q.v }
```

Cependant, cela compromet la sécurité des entiers de Peano. En effet, nous pouvons maintenant construire la borne par une boucle fantôme arbitrairement longue, bien au-delà du nombre d'entiers machine possibles. Cette boucle est exécutée en un temps nul, puisque elle est effacée à l'extraction, mais la variable générée peut alors être utilisée comme borne, brisant la propriété de croissance lente.

Nous avons cependant des moyens d'éliminer cette borne sans perdre la croissance lente. Le premier est d'interdire les appels à la fonction `succ` dans le code fantôme. De cette manière, nous sommes certains que la borne passée en argument est forcément 0 ou la copie fantôme d'un entier de Peano construit ailleurs dans le programme.

La seconde méthode est d'introduire deux nouveaux types presque identiques à `peano` et à `onetime`, mais n'ayant aucun contenu calculatoire sous-jacent. En d'autres termes, les valeurs de ce type se comportent comme des valeurs fantômes, et peuvent être complètement effacées à l'extraction. Nous équipons ces types des mêmes opérations que pour les entiers de Peano et à usage unique, à l'exception de celles pouvant avoir une incidence sur le flot de contrôle du programme, c'est-à-dire `to_int` et les opérateurs de comparaison. Cependant, les opérations manipulant ces valeurs (notamment `succ`) ne sont pas effacées, pour garantir qu'une certaine quantité de temps s'écoule à cet endroit du programme.

Remarquons que dans cette seconde méthode, la variante sans contenu calculatoire des entiers à usage unique joue un rôle proche de celui des « crédits temps » utilisé pour l'analyse de complexité temporelle dans certaines logiques de séparation [5, 18]. Dans ces logiques, les crédits temps représentent la réserve de temps restant au programme pour s'exécuter. Notre approche revient à faire l'inverse : nous utilisons les entiers de Peano comme des « crédits » représentant le temps déjà écoulé. L'opération `succ` revient alors à la génération d'un nouveau crédit temps. Nous pouvons rendre cette analogie complète en ajoutant une fonction `split` pour « couper » en deux parties un entier à usage unique, ce qui permet d'inverser l'addition et respecte l'argument de croissance lente.

```
let split (x: onetime) (p: peano) : (onetime, onetime)
  requires { x.valid }
  requires { 0 ≤ p.P.v ≤ x.v }
  writes   { x.valid }
  ensures  { not x.valid }
  returns  { a, b → a.valid ∧ b.valid
                    ∧ a.v = x.v - b.v ∧ b.v = p.v }
```

Nous avons par ailleurs ajouté cette opération à notre implémentation Why3 des entiers à usage unique. Notons que cette opération peut être ajoutée sans danger à notre formalisation des entiers restreints, car elle préserve le caractère n-borné.

3.4.3 Optimisations

Comme nous l'avons vu dans le cas du code fantôme, l'effacement de l'opération de successeur brise l'argument de croissante lente. En pratique, cela signifie que nous devons également faire attention à ce que les optimisations de code n'effacent pas ces opérations. En effet, notre argument est entièrement basé sur le fait que ces opérations correspondent à une étape d'exécution. L'effacement de ces opérations n'est pas forcément un problème, tant que ces étapes sont reportées sur d'autres opérations. Cependant, si un compilateur réécrit **for** $i = 1$ **to** 2^{32} **do** $s \leftarrow s + 1$ **end for** en $s \leftarrow s + 2^{32}$, notre argument de croissance lente s'effondre.

En pratique, cela n'est pas un problème pour des algorithmes raisonnables. Nous observons que dans les exemples que nous avons étudiés, l'utilisation d'une opération de successeur coïncide avec une allocation ou un branchement du programme, ce qui ne peut pas être optimisé de manière agressive. Pour éliminer le doute, nous pouvons également inclure au code des opérations de successeur une instruction de type « barrière », que le compilateur n'a pas le droit d'éliminer sans au moins émettre un avertissement. Notons que dans le cas où une telle optimisation est possible, il s'agit probablement d'un mauvais usage des entiers de Peano. Le cas idéal serait de disposer d'une instruction « barrière » dont la seule caractéristique serait de ne pas pouvoir être éliminée par le compilateur, ce qui permettrait de détecter les usages invalides des entiers restreints sans pour autant imposer un surcoût prohibitif à l'exécution.

3.4.4 Application à d'autres tailles d'entiers

La solution que nous avons proposée n'est pas directement applicable à d'autres tailles d'entiers, notamment des entiers plus courts. Pourtant, il semble que des arguments similaires s'appliquent. Par exemple, les entiers représentant les rangs d'union-find ou les hauteurs d'un arbre équilibré sans partage ont typiquement une valeur au plus logarithmique par rapport à leur taille, et donc par rapport au temps nécessaire pour les construire. En particulier, ces valeurs peuvent être stockées sur des entiers de taille extrêmement restreinte, typiquement 8 bits.

De manière concrète, les arbres binaires de recherche équilibrés de type AVL [3] sont typiquement implémentés en stockant la hauteur dans chaque nœud. Mais la hauteur d'un arbre AVL avec n nœuds ne dépasse pas $1.44 \log_2(n)$. Avec 8 bits, nous pouvons donc représenter les hauteurs d'arbres AVL ayant jusqu'à 2^{177} nœuds.

Rappelons que l'ordre de grandeur du temps nécessaire avec les processeurs d'aujourd'hui pour effectuer 2^{64} opérations séquentielles tourne autour des centaines d'années. Nous pouvons donc affirmer sans trop de risque que nous n'aurons jamais le temps de construire 2^{177} nœuds d'arbres, et donc que 8 bits suffisent pour les hauteurs d'un arbre AVL. Rappelons également que dans le cas d'un arbre binaire de recherche, le partage de nœuds est impossible, et qu'il faut effectivement tous les construire.

Nous pouvons donc envisager de généraliser l'approche des entiers restreints à ce genre de cas. Pour cela, la méthode la plus simple est d'introduire une nouvelle famille de types entiers, paramétrée par une fonction croissante f du temps d'exécution. Cette fonction correspond à la valeur maximale autorisée pour ces entiers à un temps donné. Enfin, nous autorisons n'importe quelle opération arithmétique standard sur ces entiers, sous réserve du passage d'un entier de Peano additionnel n tel que $f(n)$ majore le résultat. Notons que c'est aussi le cas pour l'opération `succ`.

Nous pouvons alors généraliser la notion d'état n-borné pour prendre en compte ces nouveaux types, en demandant que $f(n)$ majore leur valeur. Par exemple, nous pouvons envisager de modéliser les hauteurs d'un arbre AVL en Why3 par le type suivant :

```
type height = private { v : int }
function max_height (n: int) = ⌊1.44log2(n)⌋
val zero () : height
  ensures { result.v = 0 }
val succ (h: height) (p: peano) : height
  requires { h.v + 1 ≤ max_height p.v }
  ensures  { result.v = h.v + 1 }
val cap (n: int) (p: peano) : height
  requires { n ≤ max_height p.v }
  ensures  { result.v = n }
```

Pour ce qui est de l'extraction de ces nouveaux types, nous devons choisir la taille correspondante en fonction de f et d'un nombre d'opérations T considéré comme ne pouvant jamais être atteint. La taille choisie est alors suffisante pour éviter tout débordement en pratique si tous les entiers jusqu'à $f(T)$ sont représentables. Dans le cas concret des arbres binaires de recherche équilibrés de type AVL, nous pouvons utiliser le type associé à $f(n) = 1.44 \log_2(n)$ pour les hauteurs. Ce type peut être extrait vers des entiers 8 bits tant que nous considérons comme impossible en pratique d'exécuter séquentiellement 2^{177} opérations.

3.4.5 Code parallèle

Le cas du code concurrent ou distribué peut a priori invalider notre argument de croissance lente, à cause des opérations parallèles. En fait, il n'y a qu'une seule opération qui devient invalide dans le cas d'un programme parallèle : l'addition d'entiers à usage unique. En effet, cette opération est la seule qui permette de rassembler des incréments qui n'ont pas été effectués séquentiellement. L'utilisation des entiers de Peano ne pose aucun problème.

Nous pouvons cependant récupérer un argument de croissance lente si le nombre de processus parallèles est borné par un nombre connu P. En effet, la croissance

des entiers restreints est au plus P fois plus rapide que celle d'un programme séquentiel. Si nous avons une borne physique sur le temps durant lequel le programme sera exécuté (par exemple, 100 ans), nous pouvons alors calculer une borne sur le nombre maximum d'opérations qui seront effectuées pendant l'exécution. Si P est assez grand, ce nombre peut effectivement dépasser la borne proposée de 2^{64}, ce qui nous interdit alors de substituer les entiers restreints par des entiers 64 bits. Cependant, nous pouvons appliquer notre méthode pour montrer qu'il n'y a pas de risque pratique de débordement pour des entiers de plus grande taille, par exemple des entiers 128 bits.

3.4.6 Entiers signés

Une dernière limitation apparente de notre approche est qu'elle ne s'applique qu'à des entiers non signés. Il s'agit principalement d'une simplification. Nous pouvons sans problème généraliser au cas signé, en majorant les valeurs absolues. Nous pouvons donc ajouter une opération prédécesseur, la soustraction d'entiers à usage unique, la valeur absolue et l'inversion de signe. Nous avons d'ailleurs implémenté cette variation signée dans Why3 plutôt que la non signée.

Chapitre 4

Jeux pour la preuve de programme

4.1 Motivation

Le but du travail présenté dans ce chapitre est de développer les fondations d'une version générale de la méthode présentée dans la section 2.5. Celle-ci consiste à appliquer la logique de Hoare et le calcul de plus faible pré-condition à du code non structuré pour prouver un programme générateur de code, en l'occurrence un compilateur. Cependant, la méthode présentée précédemment souffre de certains

défauts, que nous cherchons maintenant à pallier. Le but final, que nous développerons en détail dans le chapitre 5, est d'obtenir une bibliothèque de spécification et de preuve pour le code non structuré, basée sur une logique de Hoare générale et les transformateurs de prédicat pour automatiser en partie son application.

Premièrement, nous souhaitons rendre notre méthode de preuve applicable dans un cadre plus général. En effet, le calcul développé dans la section 2.5 est limité à la machine virtuelle de notre langage cible, et ne peut donc pas être réutilisé directement pour un autre langage. Si certains éléments ne peuvent évidemment pas être partagés, notamment les spécifications des instructions de la machine, nous ne voyons pas de raisons pour que des définitions comme la correction d'un triplet de Hoare ou celles des constructeurs encapsulant les règles de raisonnement ne puissent pas être partagées entre divers langages. Nous cherchons donc un cadre général pour ces définitions.

Toujours dans le but de rendre notre méthode plus générale, nous souhaitons une méthode permettant de traiter de manière uniforme différentes interprétations possibles de la logique. Cet objectif nous vient de l'exemple des compilateurs certifiés, qui nous conduit naturellement à deux interprétations distinctes de la logique de Hoare selon le type de résultat voulu. Dans le cadre de notre expérience de vérification d'un petit compilateur, nous cherchions à obtenir un résultat de simulation en avant, c'est-à-dire que tout comportement du programme de source est un comportement du programme généré. Cela nous amène à une interprétation des triplets de Hoare en terme de propriétés d'accessibilité, autrement dit que depuis tout état de la pré-condition, le programme généré admet un comportement qui atteint la post-condition. Il s'agit d'une interprétation de nature *existentielle*.

Cependant, nous aurions également pu vouloir vérifier un énoncé de simulation en arrière pour notre petit compilateur, c'est-à-dire que tout comportement du programme généré est un comportement du programme source. Pour employer notre méthode de preuve, nous aurions alors dû utiliser une autre interprétation des triplets de Hoare, en termes de propriétés de vivacité. Une telle interprétation, de nature *universelle*, signifie que tout comportement du programme généré partant d'un état de la pré-condition atteint inéluctablement la post-condition. Malgré les différences entre ces deux interprétations, les règles de raisonnement changent très peu, voire pas du tout. Nous voulons donc que notre méthode puisse traiter de manière uniforme les deux situations.

Enfin, en plus de généraliser la méthode de preuve vis-à-vis du langage cible, nous souhaitons l'étendre pour pouvoir traiter le cas de programmes qui ne terminent pas. Nous étendons donc la notion de triplet de Hoare pour pouvoir énoncer et établir des propriétés sur l'état du programme après un nombre infini d'étapes. Cela nous permet de traiter des propriétés plus fines que la simple terminaison. Il devient en effet possible d'énoncer des propriétés sur la totalité des effets de bords observables produits au cours de l'exécution, comme les entrées et sorties. Dans le cas des compilateurs, nous pouvons par exemple vérifier que la séquence des caractères imprimés par le programme généré peut être produite par le programme de départ.

Nos objectifs de généralisation nous amènent naturellement à abstraire le langage sous-jacent par un système de transition, qui représente la sémantique opérationnelle à petits pas. Cette abstraction nous permet d'uniformiser le cadre de travail vis-à-vis du langage. Ensuite, nous remarquons que les deux interprétations possibles des triplets de Hoare correspondent respectivement à des interprétations existentielles

et universelles des systèmes de transition. Nous généralisons donc ces systèmes sous la forme de *jeux*, dans lesquels un joueur joue les transitions existentielles et l'autre les transitions universelles. Nous unifions alors les deux interprétations possibles des triplets de Hoare comme l'existence d'une stratégie gagnante du joueur existentiel pour atteindre la post-condition à partir de n'importe quel état de la pré-condition. Cette interprétation des triplets de Hoare satisfait intuitivement des règles d'inférence proches de la logique de Hoare habituelle.

Pour traiter les comportements qui ne terminent pas, nous adjoignons un ordre aux jeux, qui doit être respecté par toutes les transitions. Nous considérons alors qu'un comportement défini par une séquence infinie d'états peut être prolongé par la borne supérieure de cette séquence. Cette modélisation est notamment adaptée pour modéliser une trace d'évènements. En effet, celle-ci ne fait que s'agrandir au cours de l'exécution, et la borne supérieure des traces obtenues durant l'exécution est bien la trace de tous les événements qui se sont produits. Par le biais de cette méthode, nous pouvons donc modéliser correctement l'état obtenu après un temps infini. Nous pouvons également utiliser l'existence de stratégies gagnantes pour représenter des propriétés d'atteinte de tels états, après un temps infini.

Remarquons par ailleurs que dans le cadre uniforme des jeux, les états obtenus par des comportements infinis ne sont pas *a priori* différents des autres. Nous permettons donc que l'évolution continue même après une séquence infinie d'étapes, ce qui nous amène à généraliser à des comportements transfinis. Cette généralisation n'est pas justifiée par nos objectifs initiaux, mais permet d'uniformiser le traitement des différents états du jeu, et permet également l'utilisation d'arguments de preuve transfinis pour certaines règles.

Comme le but est d'obtenir une bibliothèque de spécification et d'aide à la preuve pour la génération de code, nous avons formalisé le développement sur les jeux en Why3. Bien qu'à l'heure où ces lignes sont écrites, nous n'ayons pas encore atteint l'objectif d'une bibliothèque complète, nous avons cependant formalisé la totalité des fondations présentées dans ce chapitre, ainsi que les règles de raisonnement d'une logique à la Hoare pour les jeux. Les preuves reposent sur une utilisation massive des indicateurs de coupures présentés dans la section 2.3.

Le chapitre est organisé de la manière suivante. Dans la section 4.2, nous présentons la notion de jeux que nous utilisons comme cadre de travail, ainsi que des notions de base comme les stratégies et l'interprétation d'un triplet de Hoare. Pour simplifier certains raisonnements subséquents, nous donnons ensuite des alternatives à la description en termes de stratégie dans la section 4.3. Dans la section 4.4, nous démontrons un théorème de simulation pour les jeux, un théorème de transfert des stratégies gagnantes d'un jeu vers un autre. En pratique, ce théorème est notre outil principal pour combiner plusieurs stratégies gagnantes. Nous montrons ensuite dans la section 4.5 comment convertir des systèmes de transition en jeux, et notamment comment convertir des propriétés d'existence de stratégie gagnante sur les jeux obtenus en propriétés d'accessibilité et de vivacité sur le système de transition originel. Enfin, nous présentons le développement Why3 associé dans la section 4.6.

4.2 Jeux et stratégies

Dans cette section, nous introduisons les notions de base sur les jeux utilisés dans ce travail. Ce sont des jeux à deux joueurs, nommés respectivement le *joueur existentiel* et le *joueur universel*. Comme nous ne nous intéressons qu'à l'existence de stratégies gagnantes pour le joueur existentiel, nous avons choisi des définitions asymétriques.

Définition 4.1 (Jeu)**.** Un *jeu* $\mathbb{G} = (G, \preccurlyeq, \Delta)$ est un ensemble partiellement ordonné $(G, \preccurlyeq)$ muni d'un ensemble de transitions $\Delta : G \to \mathcal{P}(\mathcal{P}(G))$ respectant la condition de progression suivante :

$$\forall x \in G, \forall X \in \Delta(x), \forall y \in X, x \preccurlyeq y$$

G et $\preccurlyeq$ sont respectivement appelés le *support* et l'*ordre* du jeu.

L'ensemble de transitions Δ représente la séquence d'un coup du joueur existentiel, suivi par un coup du joueur universel. Le joueur existentiel choisit l'ensemble des successeurs possibles, et le joueur universel choisit parmi eux. Une autre manière de voir est de considérer que G est l'ensemble des états existentiels, et que $\mathcal{P}(G)$ est l'ensemble des états universels. L'ensemble Δ correspond alors aux transitions existentielles, et l'appartenance aux transitions universelles. Nous pouvons également considérer que chaque élément $X \in \Delta(x)$ correspond à une procédure pouvant être appelée à l'état x, X correspondant alors à l'ensemble des états finaux possibles de cette procédure.

Nous avons choisi cette définition car elle permet de simuler l'ajout de transitions existentielles sans pour autant changer de support. Intuitivement, il suffit d'introduire des transitions vers tous les ensembles d'états dont on peut garantir l'accessibilité par le joueur existentiel dans le jeu intermédiaire. Cela peut également être vu comme l'ajout de procédures additionnelles pour le joueur existentiel.

Enfin, nous devons gérer la possibilité qu'une partie ne se termine pas. Pour cela, nous décrivons une partie par l'évolution de son historique, à savoir l'ensemble des états atteints au cours de la partie. Le plus grand élément d'un historique représente alors le dernier état atteint, donc l'état courant du jeu. Quand l'historique devient infini, ce maximum peut ne pas exister. Nous relançons alors le jeu depuis la borne supérieure de l'historique.

Définition 4.2 (Chaîne)**.** Une *chaîne* d'un ensemble partiellement ordonné $(O, \preccurlyeq)$ est un sous-ensemble totalement ordonné. Une chaîne C_1 est *préfixe* d'une chaîne C_2, noté $C_1 \sqsubseteq C_2$, si $C_1 \subseteq C_2$ et si tout élément de $C_2 \backslash C_1$ est un majorant de C_1. On note $C_1 \sqsubset C_2$ lorsque C_1 est un préfixe strict de C_2.

Définition 4.3 (Historique)**.** Un *historique* d'un jeu $(G, \preccurlyeq, \Delta)$ est une chaîne non vide de $(G, \preccurlyeq)$. Un historique H est dit *jouable* s'il possède un maximum, *limite* sinon. Lorsque H est jouable, $\max H$ correspond à l'*état courant* du jeu.

Pour pouvoir définir la notion de partie d'un jeu, nous introduisons la notion de stratégie pour chaque joueur. Les coups possibles pour chaque joueur sont déterminés par l'ensemble des transitions possibles à l'état courant.

Définition 4.4 (Stratégie existentielle)**.** Une *stratégie existentielle* pour un jeu $\mathbb{G}$ donné par $(G, \preccurlyeq, \Delta)$ est une fonction $\sigma_\exists : \{H | H \text{ jouable pour } \mathbb{G}\} \to \mathcal{P}(G)$. Elle fait un *choix valide* en H si $\sigma_\exists(H) \in \Delta(\max H)$.

Définition 4.5 (Stratégie universelle)**.** Une *stratégie universelle* pour $(G, \preccurlyeq, \Delta)$ est une fonction $\sigma_\forall : \mathcal{P}(G) \to G$. Elle fait un *choix valide* en X si $\sigma_\forall(X) \in X$.

Une stratégie existentielle correspond au choix d'une transition dans un état étant donnée l'évolution complète du jeu jusqu'à cet état. Les stratégies universelles correspondent aux choix de transition pour les états universels. Nous avons choisi de ne pas faire dépendre les stratégies universelles de l'historique par simplicité, car la capacité du joueur universel à empêcher l'existence d'une stratégie gagnante pour le joueur existentiel reste inchangée. Ceci sera justifié dans la remarque 4.30.

Remarque 4.6. Le problème reste ouvert de savoir si la dépendance de la stratégie existentielle en l'historique complet est nécessaire, ou s'il est toujours possible de construire une stratégie gagnante ne dépendant que de l'état courant.

Nous pouvons maintenant définir le déroulement d'une partie.

Définition 4.7 (Séquence des coups d'une partie)**.** Soient $(G, \preccurlyeq, \Delta)$ un jeu, $(\sigma_\exists, \sigma_\forall)$ une paire de stratégies, et x_0 un point de départ. Nous définissons une suite (x_α) indexée par les ordinaux et croissante pour $\preccurlyeq$ par récurrence transfinie :

— $x_0 = x_0$,
— $x_{\alpha+1} = \sigma_\forall(\sigma_\exists(\{x_\beta \mid \beta \leq \alpha\}))$ si les choix sont valides, non défini sinon,
— $x_\lambda = \sup_{\alpha<\lambda} x_\alpha$ pour λ limite, non défini si la borne supérieure n'existe pas.

Si un élément x_α n'est pas défini, les éléments indexés par les ordinaux supérieurs à α ne le sont pas non plus. La *séquence des coups* de la partie est alors la suite (x_α) des éléments qui sont bien définis. Nous vérifions qu'elle est bien croissante grâce à la condition de progression des transitions du jeu.

Définition 4.8 (Partie)**.** La *partie* d'un jeu $(G, \preccurlyeq, \Delta)$ associée aux stratégies $(\sigma_\exists, \sigma_\forall)$ et au point de départ $x_0 \in G$ est l'ensemble d'historiques suivant :

$$\mathcal{F} = \{H \in \mathcal{P}(G) \mid \exists \beta \neq 0. H = \{x_\alpha \mid \alpha < \beta\}\}$$

Nous pouvons de manière alternative définir la partie $\mathcal{F}$ comme étant le plus petit ensemble d'historiques clos par les règles suivantes :

(i) $\{x_0\} \in \mathcal{F}$
(ii) Si $H \in \mathcal{F}$ est un historique jouable, et $\sigma_\exists, \sigma_\forall$ effectuent des choix valides en H et $\sigma_\exists(H)$ respectivement, alors $H \cup \{\sigma_\forall(\sigma_\exists(H))\} \in \mathcal{F}$
(iii) Si $\mathcal{H} \subseteq \mathcal{F}$ est une chaîne non vide d'historiques pour l'ordre préfixe, alors $\bigcup \mathcal{H} \in \mathcal{F}$
(iv) Si $H \in \mathcal{F}$ admet une borne supérieure, alors $H \cup \{\sup H\} \in \mathcal{F}$

Nous introduisons cette seconde définition pour montrer que les parties peuvent être définies sans recourir au formalisme ordinal, ce qui correspond au choix de représenter les historiques par des chaînes. Nous aurions également pu choisir de représenter de manière systématique les historiques par des séquences transfinies, mais les chaînes représentent mieux le fait que la partie ne peut pas se dérouler « indéfiniment » (au sens transfini du terme), c'est-à-dire que la durée de la partie

est limitée par le cardinal du support. De plus, le formalisme basé sur les chaînes se prête mieux à la mécanisation des preuves.

Lemme 4.9. *Les deux définitions d'une partie sont équivalentes.*

Démonstration. Tout d'abord, vérifions que tout ensemble $\mathcal{G}$ clos par les règles (i)-(iv) contient bien $\mathcal{F}$. Nous procédons par récurrence transfinie pour montrer que pour tout ordinal $\beta \neq 0$, $\{x_\alpha \mid \alpha < \beta\} \in \mathcal{G}$. Chacun des cas de la récurrence suit directement d'une règle de propagation vérifiée par $\mathcal{G}$.

— Cas $\beta = 1$: règle (i).
— cas $\beta = \gamma + 1$ où γ est limite : règle (iv).
— cas $\beta = \gamma + 1$ où γ est lui-même successeur : règle (ii).
— cas β limite : règle (iii) avec $\mathcal{H} = \{\{x_\alpha \mid \alpha < \gamma\} \mid \gamma < \beta\}$.

Pour l'implication inverse, nous montrons maintenant que $\mathcal{F}$ est bien point fixe de ces quatre règles.

— Règle (i) : $\{x_0\} = \{x_\alpha \mid \alpha < 1\} \in \mathcal{F}$.
— règle (ii) : Considérons $H \in \mathcal{F}$ un historique jouable. Comme $H \in \mathcal{F}$, il existe un ordinal β tel que $H = \{x_\alpha \mid \alpha < \beta\}$. Soit β_0 le plus petit ordinal tel que $x_{\beta_0} = \max H$. Par croissance de (x_α), on peut supposer $\beta = \beta_0 + 1$ car les deux valeurs donnent le même historique H. En particulier, si les choix sont valides alors $H \cup \{\sigma_\forall(\sigma_\exists(H))\} = \{x_\alpha \mid \alpha < \beta + 1\} \in \mathcal{F}$.
— règle (iii) : Soit $\mathcal{H} \subseteq \mathcal{F}$ une chaîne non vide pour l'ordre préfixe. Pour tout $H \in \mathcal{H}$, il existe un ordinal β_H tel que $H = \{x_\alpha \mid \alpha < \beta_H\}$. Alors $\bigcup \mathcal{H} = \{x_\alpha \mid \alpha < \sup_{H \in \mathcal{H}} \beta_H\} \in \mathcal{F}$.
— règle (iv) : Soit $H \in \mathcal{F}$ un historique admettant une borne supérieure. Si H contient sa borne supérieure, alors $H \cup \{\sup H\} = H \in \mathcal{F}$. Sinon, il existe un ordinal λ tel que $H = \{x_\alpha \mid \alpha < \lambda\}$. Comme H n'a pas de maximum, λ est un ordinal limite, et $H \cup \{\sup H\} = \{x_\alpha \mid \alpha < \lambda + 1\} \in \mathcal{F}$.

□

Exemple 4.10. Le jeu $\overline{\mathbb{N}}_3$ est un jeu sur le support $\overline{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$, muni de l'ordre habituel. Son ensemble de transitions est donné par

$$\Delta(n) = \begin{cases} \{\{n + k + j \mid 0 \leq j \leq 3\} \mid 0 \leq k \leq 3\} & \text{si } n \in \mathbb{N} \\ \emptyset & \text{si } n = \infty \end{cases}$$

Ce jeu correspond intuitivement à un compteur, auquel chaque joueur peut tour à tour ajouter un nombre entre 0 et 3. Si le compteur est incrémenté indéfiniment, il devient également infini lors du premier passage à la borne supérieure. Les transitions partant d'un état fini sont représentées par la figure 4.1, où les points blancs représentent les états universels possibles.

Nous pouvons alors encoder les stratégies correspondant à l'incrémentation du compteur par 1 de la manière suivante :

$$\begin{array}{lcll} \sigma_\exists(H) & = & \{x+1, x+2, x+3, x+4\} & \text{avec } x = \max H \\ \sigma_\forall(\{a, a+1, a+2, a+3\}) & = & a+1 & \\ \sigma_\forall(X) & = & 0 & \text{sinon} \end{array}$$

La partie $\mathcal{F}$ de $\overline{\mathbb{N}}_3$ associée à ces deux stratégies et au point de départ 0 est formée par :

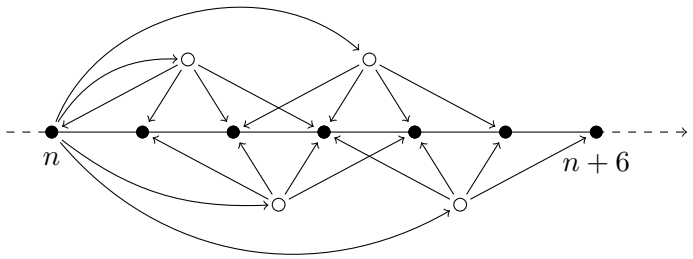

FIGURE 4.1 – Transitions partant d'un état fini n pour le jeu $\overline{\mathbb{N}}_3$

— les historiques $\{2k \mid 0 \leq k < n\}$ pour $n \in \overline{\mathbb{N}}\backslash\{0\}$,
— l'historique $\{2k \mid 0 \leq k\} \cup \{\infty\}$.

Les historiques $\{2k \mid 0 \leq k < n\}$ pour $n \in \mathbb{N}\backslash\{0\}$ sont obtenus par itération des règles (i) et (ii). La règle (iii) donne $\{2k \mid 0 \leq k\}$ comme union des précédents, et la règle (iv) lui ajoute sa borne supérieure ∞.

Nous pouvons maintenant définir la notion de stratégie gagnante.

Définition 4.11 (Historique gagnant)**.** Étant donné un jeu $\mathbb{G}$, des stratégies $(\sigma_\exists, \sigma_\forall)$ et un ensemble d'états cibles Q, un historique H est *gagnant* pour le joueur existentiel si l'une des conditions suivantes est vérifiée :

(i) H est jouable et $\max H \in Q$ (victoire effective) ;
(ii) H est jouable, le choix de $\sigma_\exists$ en H est valide, mais celui de $\sigma_\forall$ en $\sigma_\exists(H)$ ne l'est pas (victoire par forfait du joueur universel) ;
(iii) H n'a pas de borne supérieure (victoire par défaut de complétude)

Définition 4.12 (Stratégie gagnante)**.** Une *stratégie gagnante* pour un point de départ x et un ensemble d'état cibles Q est une stratégie existentielle $\sigma_\exists$ qui assure que quelle que soit la stratégie du joueur universel, pour une partie $\mathcal{F}$ partant de x, $\mathcal{F}$ contient un historique gagnant.

Par extension, une stratégie gagnante pour un ensemble de départ P et un ensemble d'états cibles Q est une stratégie gagnante pour tout élément de P.

Les deux premières conditions de victoire sont naturelles, mais la troisième est inhabituelle. Nous avons choisi de l'introduire pour pouvoir facilement traiter les cas où l'ordre souffre d'un défaut de complétude. Dans ce cas, la partie peut se bloquer parce qu'il n'y a pas d'état limite, ce qui ne semble correspondre à la victoire d'aucun joueur. Comme cette situation ne se produit pas dans les exemples concrets, nous avons décidé arbitrairement que cela correspondait à une victoire du joueur existentiel. Nous pouvons ainsi simplifier les hypothèses de certains théorèmes.

Un autre moyen d'éviter ce problème est d'imposer une contrainte de complétude sur les jeux, qui peut toujours être garantie par un procédé de complétion. Cependant, nous considérons que ce n'est pas toujours une bonne solution car elle nous oblige à traiter les états introduits par la complétion. En pratique, ces états ne sont pas pertinents.

Exemple 4.13. Considérons de nouveau le jeu $\overline{\mathbb{N}}_3$ et les stratégies définies dans l'exemple 4.10. La stratégie existentielle proposée n'est pas gagnante pour le point de départ 0 et l'ensemble cible $\{1\}$, puisqu'aucun des historiques de la partie considérée n'est gagnant. Par contre, on peut montrer que cette stratégie existentielle est gagnante pour l'ensemble cible $\{\infty\}$ car elle force la progression tant que l'état est fini.

De manière plus générale, on peut montrer que le jeu $\overline{\mathbb{N}}_3$ admet une stratégie gagnante pour un point de départ x et un ensemble cible Q si et seulement si Q contient ∞ ou bien Q contient quatre entiers consécutifs supérieurs ou égaux à x.

Remarque 4.14. Le joueur existentiel ne peut effectivement perdre que sous deux conditions. Premièrement, s'il n'a pas encore réussi à atteindre l'ensemble cible. Deuxièmement, s'il choisit une transition invalide, ou une transition qui permet au joueur universel de revenir sur l'état courant. Dans le second cas, le joueur universel peut définitivement bloquer la progression de la partie à l'état actuel, prévenant ainsi l'atteinte de l'ensemble cible. Cela signifie en particulier que les transitions qui permettent de revenir sur leur état source sont inutiles. Nous justifierons cette inutilité dans la section suivante, ainsi que les conditions de défaite susmentionnées pour une stratégie existentielle.

Définition 4.15 (Garantie)**.** Soient $\mathbb{G} = (G, \preccurlyeq, \Delta)$ un jeu et $P, Q \subseteq G$. On dit que P *garantit* Q pour le jeu $\mathbb{G}$, noté $\langle P \hookrightarrow Q \rangle_{\mathbb{G}}$, s'il existe une stratégie gagnante pour l'ensemble de départ P et l'ensemble cible Q.

Remarquons que la notion de garantie ainsi définie est analogue à un triplet de Hoare. Nous identifions ainsi P à une pré-condition, Q à une post-condition, et le jeu au code d'un programme. Dans le cas où le jeu est purement universel, c'est-à-dire que le joueur existentiel ne peut pas effectuer plus d'un choix valide par état, nous retrouvons notamment la notion de correction totale. En effet, la validité de la garantie revient alors à énoncer que toute évolution suivant les transitions du jeu et partant d'un état de P atteindra inéluctablement un état de Q.

Cette définition des garanties n'est pas la seule possible. Nous aurions notamment pu donner une définition qui inverse les quantifications sur la stratégie gagnante et le point de départ. Ces deux définitions sont en réalité équivalentes, comme nous le démontrons dans le lemme suivant.

Lemme 4.16. *La garantie $\langle P \hookrightarrow Q \rangle_{\mathbb{G}}$ est valide si et seulement si pour tout $x \in P$, il existe une stratégie gagnante pour le jeu $\mathbb{G}$, le point de départ x, et l'ensemble cible Q.*

Démonstration. Le sens direct est immédiat. Pour la réciproque, soit $(\sigma_{\exists,x})_{x \in P}$ une famille de stratégies gagnantes pour chaque point de départ.

Posons $\sigma_\exists(H) = \sigma_{\exists,\min H}(H)$ (définie arbitrairement si le minimum n'existe pas). Alors $\sigma_\exists$ est gagnante pour l'ensemble de départ P.

En effet, pour un point de départ x et une stratégie universelle $\sigma_\forall$, considérons les parties associées aux deux stratégies existentielles $\sigma_\exists$ et $\sigma_{\exists,x}$. On montre facilement par induction transfinie que ces deux stratégies génèrent la même séquence de coups, et donc que les deux parties sont identiques. En particulier, comme la partie associée à $\sigma_{\exists,x}$ contient un historique gagnant, celle associée à $\sigma_\exists$ contient le même historique gagnant. □

Nous pouvons alternativement voir ce lemme comme un moyen de partitionner l'ensemble de départ d'une garantie. Par le biais de l'identification aux triplets de Hoare, cela donne un résultat analogue à une règle de raisonnement pour un branchement.

Corollaire 4.17. *La garantie* $\langle \bigcup_{i\in I} P_i \hookrightarrow Q\rangle_{\mathbb{G}}$ *est valide si et seulement si pour tout* $i \in I$*, la garantie* $\langle P_i \hookrightarrow Q\rangle_{\mathbb{G}}$ *est valide.*

Démonstration. Conséquence immédiate du lemme 4.16. □

Notons également que la définition d'une garantie est compatible avec l'interprétation « locale » d'un jeu, au sens qu'elle respecte l'interprétation intuitive des transitions existentielles et universelles en tant que quantificateurs. Nous pouvons alors décomposer naturellement une garantie sur les étapes d'un jeu. Cette propriété se formalise comme suit.

Lemme 4.18. *Soit* $\mathbb{G} = (G, \preccurlyeq, \Delta)$ *un jeu et* $x \in G, Q \subseteq G$*. La garantie* $\langle \{x\} \hookrightarrow Q\rangle_{\mathbb{G}}$ *est valide ssi, ou bien* $x \in Q$*, ou bien il existe* $P \in \Delta(x)$ *tel que la garantie* $\langle P \hookrightarrow Q\rangle_{\mathbb{G}}$ *soit valide.*

Démonstration. Montrons tout d'abord le sens direct. Soit $\sigma_\exists$ une stratégie gagnante témoin de la garantie $\langle \{x\} \hookrightarrow Q\rangle_{\mathbb{G}}$. Supposons également que $x \notin Q$, car si $x \in Q$ on peut conclure. Alors le témoin $P \in \Delta(x)$ voulu est $P = \sigma_\exists(x)$. L'appartenance $P \in \Delta(x)$ est assurée par le fait que le choix de $\sigma_\exists$ en $\{x\}$ doive être valide, sans quoi toute partie associée à $\sigma_\exists$ et au point de départ x se réduirait à $\{\{x\}\}$. Comme cet ensemble ne contient pas d'historique gagnant, ce serait une contradiction.

Montrons maintenant que $\langle P \hookrightarrow Q\rangle_{\mathbb{G}}$ est valide. Nous réduisons d'abord cette garantie à $\langle P\backslash\{x\} \hookrightarrow Q\rangle_{\mathbb{G}}$ via le lemme 4.17, ce qui est toujours possible car $\langle \{x\} \hookrightarrow Q\rangle_{\mathbb{G}}$ est vraie. Cette restriction nous permet d'exhiber une stratégie gagnante $\sigma'_\exists$ définie par :

$$\sigma'_\exists(H) = \begin{cases} \sigma_\exists(\{x\} \cup H) & \text{si } x \text{ est minimal dans } (\{x\} \cup H) \\ \emptyset & \text{sinon} \end{cases}$$

Vérifions que $\sigma'_\exists$ est bien gagnante pour l'ensemble de départ $P\backslash\{x\}$ et l'ensemble cible Q. Soit donc $\sigma'_\forall$ une stratégie universelle et $y \in P\backslash\{x\}$ un point de départ quelconque, et montrons que la partie associée $\mathcal{F}'$ contient un historique gagnant. Posons pour cela $\mathcal{F}$ la partie associée à $\sigma_\exists$, au point de départ x, et à la stratégie universelle $\sigma_\forall$ définie par :

$$\sigma_\forall(X) = \begin{cases} y & \text{si } X = P \\ \sigma'_\forall(X) & \text{sinon} \end{cases}$$

Posons également A l'ensemble des historiques $H \in \mathcal{F}$ tels que

$$\{x, y\} \sqsubseteq H \Rightarrow H\backslash\{x\} \in \mathcal{F}'.$$

Il nous suffit alors de vérifier que A est clos par les règles de propagation (i)-(iv) qui définissent $\mathcal{F}$, ce qui revient en fait à un raisonnement par induction sur les historiques de $\mathcal{F}$. En effet, nous en déduisons que $\mathcal{F} \subseteq A$ par minimalité. En particulier, notons que $\mathcal{F}$ contient un historique gagnant H_0 pour l'ensemble cible Q. Cet

historique H_0 ne peut pas être $\{x\}$ car $x \in Q$ et les choix des stratégies $\sigma_\exists, \sigma_\forall$ sont valides en $\{x\}, \sigma_\exists(\{x\}) = P$. Donc $\{x, y\} \sqsubseteq H_0$, et $H_0 \backslash \{x\} \in \mathcal{F}'$ est un historique gagnant pour Q.

Le fait que A satisfasse les mêmes règles de propagation (iii) et (iv) que $\mathcal{F}$ suit directement des règles correspondantes pour $\mathcal{F}'$, et la règle (i) est immédiate. Pour la règle (ii), il y a une légère subtilité à cause de la différence entre les deux stratégies universelles. Soit donc $H \in A$ un historique, $z = \sigma_\forall(\sigma_\exists(H))$, et vérifions que si $\{x, y\} \sqsubseteq H \cup \{z\}$, alors $(H \cup \{z\}) \backslash \{x\} \in \mathcal{F}'$. Comme $H \in \mathcal{F}$, soit $H = \{x\}$, soit $\{x, y\} \sqsubseteq H$. Dans le premier cas, nous pouvons directement conclure car $z = y$. Dans le cas contraire, nous avons $H \backslash \{x\} \in \mathcal{F}'$, et $\sigma'_\exists(H \backslash \{x\}) = \sigma_\exists(H)$. Si $\sigma_\forall$ et $\sigma'_\forall$ coïncident en $\sigma_\exists(H)$, la règle de propagation (ii) de $\mathcal{F}'$ permet de conclure. Sinon, nous avons forcément $\sigma_\exists(H) = P$ et $z = y \in H$, donc $(H \cup \{z\}) \backslash \{x\} = H \backslash \{x\} \in \mathcal{F}'$.

Pour le sens réciproque, plaçons-nous d'abord dans le cas où $x \in Q$. Alors n'importe quelle stratégie existentielle est gagnante pour le point de départ et l'ensemble cible Q. En effet, l'historique $\{x\}$ appartient à toute partie ayant x pour point de départ, et est un historique gagnant de manière effective.

Plaçons-nous maintenant dans le cas où il existe $P \in \Delta(x)$ tel que $\langle P \hookrightarrow Q \rangle_{\mathbb{G}}$ est valide, et soit $\sigma_\exists$ la stratégie gagnante associée. Nous pouvons tout d'abord éliminer le cas $x \in P$, car alors la stratégie $\sigma_\exists$ est par définition gagnante pour le point de départ x et l'ensemble cible Q. Dans le cas où $x \notin P$, considérons la stratégie $\sigma'_\exists$ définie par :

$$\sigma'_\exists(H) = \begin{cases} P & \text{si } H = \{x\} \\ \sigma_\exists(H \backslash \{x\}) & \text{si } \{x\} \sqsubseteq H \wedge H \neq \{x\} \\ \emptyset & \text{sinon} \end{cases}$$

Soit $\sigma_\forall$ une stratégie universelle quelconque, et montrons que la partie $\mathcal{F}'$ associée à $\sigma'_\exists, \sigma_\forall$ et au point de départ x contient un historique gagnant. Si $\sigma_\forall$ fait un choix invalide en P, $\{x\}$ convient. Sinon, soit $\mathcal{F}$ la partie associée à $\sigma_\exists, \sigma_\forall$ et au point de départ $\sigma_\forall(P)$. Par un raisonnement de clôture/induction similaire à celui effectué pour le sens direct, nous vérifions facilement que pour tout historique $H \in \mathcal{F}$, $\{x\} \cup H \in \mathcal{F}'$. En particulier, $\mathcal{F}$ contient un historique gagnant qui peut être traduit en un historique gagnant de $\mathcal{F}'$. □

Une conséquence immédiate de cette propriété locale est une propriété analogue à la règle de la logique de Hoare pour une instruction nulle.

Corollaire 4.19. *Soit $\mathbb{G} = (G, \preccurlyeq, \Delta)$ un jeu et $P \subseteq Q \subseteq G$ deux ensembles d'états. Alors $\langle P \hookrightarrow Q \rangle_{\mathbb{G}}$ est valide.*

Démonstration. Le lemme 4.16 permet de décomposer $\langle P \hookrightarrow Q \rangle_{\mathbb{G}}$ comme une conjonction de garanties avec des singletons comme ensembles de départ, tandis que le lemme 4.18 montre qu'elles sont toutes vraies. □

Nous pouvons également dériver facilement une propriété des garanties analogue à la règle d'affaiblissement de la logique de Hoare.

Lemme 4.20. *Soit $\mathbb{G} = (G, \preccurlyeq, \Delta)$ un jeu et $P_2 \subseteq P_1 \subseteq G$, $Q_1 \subseteq Q_2 \subseteq G$ quatre ensembles d'états. Alors si la garantie $\langle P_1 \hookrightarrow Q_1 \rangle_{\mathbb{G}}$ est valide, la garantie $\langle P_2 \hookrightarrow Q_2 \rangle_{\mathbb{G}}$ est valide aussi.*

Démonstration. Supposons que $\langle P_1 \hookrightarrow Q_1 \rangle$ soit valide, il existe donc une stratégie gagnante $\sigma_\exists$ pour n'importe quel point de départ $x \in P_1$ et l'ensemble cible Q_1. Nous pouvons montrer que $\sigma_\exists$ est également gagnante pour n'importe quel point de départ $x \in P_2$ et l'ensemble cible Q_2. En effet, comme $x \in P_1$ n'importe quelle partie associée à $\sigma_\exists$ et au point de départ x contient un historique gagnant pour Q_1, qui est immédiatement gagnant pour Q_2 aussi. □

4.3 Jeux sans stratégie

Dans cette section, nous introduisons des notions qui donnent une alternative au raisonnement sur les stratégies. La notion d'historique u-accessible nous permet de donner une autre définition des stratégies gagnantes qui ne fait pas référence aux stratégies universelles, tandis que les invariants de victoire donnent une alternative aux stratégies gagnantes pour définir les garanties. Les notions définies dans cette section sont implicitement relatives à un jeu fixé $\mathbb{G} = (G, \preccurlyeq, \Delta)$.

Définition 4.21 (Historique u-accessible)**.** Étant donné une stratégie existentielle $\sigma_\exists$ et un point de départ x_0, un historique H est dit *u-accessible* (accessible par le joueur universel) pour $\sigma_\exists$ et x_0 s'il existe une stratégie universelle $\sigma_\forall$ pour laquelle cet historique appartient à la partie engendrée par $\sigma_\forall$. Une telle stratégie est appelée un *témoin d'u-accessibilité* pour H.

Une propriété importante des historiques u-accessibles est que l'on peut entièrement caractériser leurs témoins d'u-accessibilité.

Définition 4.22. Soient $\sigma_\exists$ une stratégie existentielle, et H un historique. Nous définissons l'*empreinte des ensembles de choix de la stratégie universelle jusqu'à* H comme l'ensemble

$$\mathcal{C}(\sigma_\exists, H) = \{\sigma_\exists(H_0) | H_0 \sqsubset H, H_0 \text{ jouable } \}$$

Lemme 4.23. *Soient $\sigma_\exists$ une stratégie existentielle, $x_0 \in G$ un point de départ, $\sigma_{\forall,1}, \sigma_{\forall,2}$ deux stratégies universelles et H un historique appartenant à la partie $\mathcal{F}_1$ associée à $\sigma_{\forall,1}$. Alors H appartient à la partie $\mathcal{F}_2$ associée à $\sigma_{\forall,2}$ si et seulement si $\sigma_{\forall,1}$ et $\sigma_{\forall,2}$ coïncident sur $\mathcal{C}(\sigma_\exists, H)$.*

Démonstration. Pour le sens réciproque, considérons l'ensemble d'historiques $A = \{H_0 \mid H_0 \sqsubseteq H \Rightarrow H_0 \in \mathcal{F}_2\}$. On vérifie immédiatement que A est clos par les règles qui définissent $\mathcal{F}_1$, en utilisant l'hypothèse de coïncidence pour la règle (ii). Par minimalité de $\mathcal{F}_1$, $\mathcal{F}_1 \subseteq A$, et en particulier $H \in \mathcal{F}_2$.

Pour le sens direct, nous considérons l'ensemble d'historiques suivant :

$$B = \{H_0 \mid H_0 \sqsubseteq H \Rightarrow \sigma_{\forall,1}, \sigma_{\forall,2} \text{ coïncident sur } \mathcal{C}(\sigma_\exists, H_0)\}$$

De la même façon que pour le sens réciproque, il suffit de vérifier que B est clos par les règles qui définissent $\mathcal{F}_1$:

— règle (i) : immédiat car $\{x_0\}$ n'ayant pas de préfixe strict non vide, $\mathcal{C}(\sigma_\exists, \{x_0\}) = \emptyset$

— règle (iii) : il suffit de vérifier que pour toute chaîne d'historiques $\mathcal{H}$, $\mathcal{C}(\sigma_\exists, \bigcup \mathcal{H}) = \bigcup_{H' \in \mathcal{H}} \mathcal{C}(\sigma_\exists, H')$. Mais comme $\bigcup \mathcal{H}$ est la borne supérieure de $\mathcal{H}$ pour l'ordre préfixe, tout préfixe strict de cette borne supérieure est effectivement un préfixe strict de l'un des éléments de $\mathcal{H}$.

— règle (iv) : comme précédemment, il suffit de vérifier que pour tout historique H_1, tout historique jouable H_0 qui est préfixe strict de $H_2 = H_1 \cup \{\sup H_1\}$ est également préfixe strict de H_1. Mais comme H_0 est préfixe strict, $\max H_0 \prec \sup H_1$. En particulier, il existe $x \in H_1$ tel que $\max H_0 \prec x$, donc H_0 est bien préfixe strict de H_1.

— règle (ii) : nous devons vérifier que pour tout $H_1 \in B$, alors $H_2 \in B$ où $H_2 = H_1 \cup \{\sigma_{\forall,1}(\sigma_\exists(H_1))\}$, en supposant les choix valides pour les deux stratégies. Comme c'est immédiat si H_2 n'est pas préfixe de H, on peut supposer que $H_2 \sqsubseteq H$. De plus, on peut également supposer que H_1 et H_2 sont distincts. Dans ce cas, $\mathcal{C}(\sigma_\exists, H_2) \backslash \mathcal{C}(\sigma_\exists, H_1) \subseteq \{\sigma_\exists(H_1)\}$. Il suffit donc de vérifier que les stratégies universelles coïncident sur $\sigma_\exists(H_1)$.
Remarquons que comme les historiques H_1 et H_2 sont préfixes de $H \in \mathcal{F}_2$, ils sont eux-mêmes des éléments de la partie $\mathcal{F}_2$. En particulier, on en déduit que $\sigma_{\forall,2}$ fait un choix valide en $\sigma_\exists(H_1)$, et que ce choix n'est pas $\max H_1$. Dans le cas contraire la séquence des coups devrait soit s'arrêter à l'état $\max H_1$, soit devenir constante à partir de cet état. Dans tous les cas, H_1 serait le maximum de $\mathcal{F}_2$, en contradiction avec l'existence de H_2. On en déduit donc que l'historique $H_3 = H_1 \cup \{\sigma_{\forall,2}(\sigma_\exists(H_1))\}$ appartient à $\mathcal{F}_2$ et est strictement plus grand que H_1. Comme tous les historiques d'une partie sont comparables pour l'ordre préfixe, H_2 et H_3 doivent être comparables, ce qui n'est possible que si les deux stratégies universelles coïncident en $\sigma_\exists(H_1)$.

□

En particulier, pour un historique u-accessible donné, on peut créer des témoins universels dont la continuation à partir de cet historique est presque arbitraire. La seule restriction est que si la possibilité de revenir à l'état courant existe, il peut être obligatoire de le faire. Cette restriction est sans importance, car comme nous l'avons déjà mentionné dans la remarque 4.14, il s'agit alors du meilleur choix possible pour le joueur universel.

Lemme 4.24. *Soient $\sigma_\exists$ une stratégie, $x_0 \in G$ un point de départ fixés. Si H est un historique jouable et u-accessible pour $x_0, \sigma_\exists$, et si le choix de $\sigma_\exists$ est valide en H, alors pour tout y tel que $(\max H \in \sigma_\exists(H) \Rightarrow y = \max H)$, il existe un témoin d'u-accessibilité $\sigma_\forall$ pour H tel que $\sigma_\forall(\sigma_\exists(H)) = y$.*

Démonstration. Soit $\sigma_{\forall,0}$ un témoin d'u-accessibilité pour H. On définit le témoin voulu par :

$$\sigma_\forall(X) = \begin{cases} y & \text{si } X = \sigma_\exists(H) \\ \sigma_{\forall,0}(X) & \text{sinon} \end{cases}$$

En utilisant le lemme 4.23 il suffit de montrer que $\sigma_{\forall,0}$ et $\sigma_\forall$ coïncident sur $\mathcal{C}(\sigma_\exists, H)$. Par l'absurde, supposons qu'elles ne coïncident pas. Cela revient à dire que $\sigma_{\forall,0}(\sigma_\exists(H)) \neq y$ et qu'il existe un historique jouable $H_0 \sqsubset H$ tel que $\sigma_\exists(H_0) = \sigma_\exists(H)$. En particulier, si $\mathcal{F}$ est la partie associée à $\sigma_{\forall,0}$, $H \in \mathcal{F}$ donc $H_0 \in \mathcal{F}$. Comme H_0 n'est pas maximal dans $\mathcal{F}$, la séquence des coups de la partie

ne s'arrête pas à l'état $\max H_0$, et les choix de $\sigma_\exists$ en H_0 et de $\sigma_{\forall,0}$ en $\sigma_\exists(H_0)$ sont valides. En particulier, l'historique $H_1 = H_0 \cup \{\sigma_{\forall,0}(\sigma_\exists(H_0))\}$ appartient à $\mathcal{F}$, et est comparable avec H. Comme il n'y a pas d'historique strictement entre H_0 et H_1, la seule possibilité est que $H_1 \sqsubseteq H$. Via la condition de progression de $\mathbb{G}$ et $\sigma_{\forall,0}(\sigma_\exists(H_0)) \in H$ on obtient :

$$\max H \preccurlyeq \sigma_{\forall,0}(\sigma_\exists(H)) = \sigma_{\forall,0}(\sigma_\exists(H_0)) \preccurlyeq \max H$$

ce qui est en contradiction avec les hypothèses sur y.

□

Nous pouvons maintenant donner une définition alternative des stratégies gagnantes via la notion d'historique perdant.

Définition 4.25 (Historique perdant)**.** Soient $\sigma_\exists$ une stratégie existentielle et x_0 un point de départ. Un historique u-accessible H pour $\sigma_\exists, x_0$ est *perdant* pour un ensemble cible Q si $H \cap Q = \emptyset$, H est jouable, et soit

— le choix de $\sigma_\exists$ est invalide en H, soit
— $\max H \in \sigma_\exists(H)$

Théorème 4.26. *Une stratégie existentielle $\sigma_\exists$ est gagnante pour un point de départ x_0 et un ensemble cible Q si, et seulement si, il n'existe pas d'historique u-accessible pour $\sigma_\exists, x_0$ et perdant pour Q.*

Démonstration. Pour le sens direct, on suppose par contradiction qu'il existe un historique H qui soit à la fois u-accessible et perdant pour Q. Soit $\sigma_\forall$ un témoin d'u-accessibilité pour H. En utilisant le lemme 4.24, on peut supposer que si le choix de $\sigma_\exists$ est valide en H, alors $\sigma_\forall(\sigma_\exists(H)) = \max H$. Considérons maintenant la partie $\mathcal{F}$ associée à $\sigma_\forall$. Les contraintes que l'on vient d'imposer sur H assurent que soit la séquence des coups de la partie s'arrête, soit elle devient constante. Donc H est le maximum de $\mathcal{F}$. Mais comme $\sigma_\exists$ est gagnante, il existe également un historique $H_0 \in \mathcal{F}$ gagnant. Comme $H_0 \cap Q \subseteq H \cap Q = \emptyset$, on en déduit que soit H_0 n'a pas de borne supérieure, soit le choix de la stratégie universelle en $\sigma_\exists(H_0)$ est invalide. Dans les deux cas, la séquence des coups cesse d'être définie et H_0 est donc le maximum de $\mathcal{F}$. En particulier, $H = H_0$, qui est donc à la fois gagnant et perdant. Les contraintes choisies sur $\sigma_\forall$ garantissent que c'est une contradiction.

Pour le sens réciproque, considérons une stratégie universelle $\sigma_\forall$ et la partie associée $\mathcal{F}$, et montrons que $\mathcal{F}$ contient un historique gagnant. Comme $\mathcal{F}$ est une chaîne pour l'ordre préfixe, l'historique $H = \bigcup \mathcal{F}$ appartient à $\mathcal{F}$ par la règle (iii), et donc est le maximum de la partie. Si H est gagnant, on a l'historique voulu, supposons donc qu'il ne l'est pas. En particulier, H possède une borne supérieure x, et donc $H \cup \{x\} \in \mathcal{F}$. Par maximalité de H, $x \in H$, donc $x = \max H$, donc H est jouable. Donc si $\sigma_\exists$ fait un choix valide en H, $\sigma_\forall$ fait également un choix valide en $\sigma_\exists(H)$ car H n'est pas gagnant. La maximalité de H montre que dans ce cas $\sigma_\forall(\sigma_\exists(H)) = \max H$, et comme les choix sont valides, $\max H \in \sigma_\exists(H)$. En particulier, si $H \cap Q = \emptyset$, alors l'une des deux conditions pour que H soit perdant est vérifiée. Mais ce n'est pas possible car H est u-accessible. Donc $H \cap Q$ admet un habitant x. Soit $H_0 = \{y | y \in H, y \preccurlyeq x\}$. Alors $H_0 \sqsubseteq H$, donc $H_0 \in \mathcal{F}$, et $x = \max H_0$, ce qui revient à dire que H_0 est gagnant de manière effective.

□

La notion d'u-accessibilité est donc particulièrement utile dans le cas où l'on sait qu'une stratégie est gagnante. Par exemple, nous pouvons montrer que si un historique est u-accessible et ne contient aucun état gagnant, alors le choix de la stratégie existentielle induit nécessairement une progression stricte. De plus, nous montrons que l'u-accessibilité peut être propagée entre historiques sans faire explicitement référence aux stratégies universelles, par des règles similaires à celles qui définissent une partie.

Lemme 4.27. *Soit $\sigma_\exists$ une stratégie existentielle et x_0 un point de départ. L'ensemble des historiques u-accessibles pour $\sigma_\exists, x_0$ est clos par les règles suivantes :*

(i) $\{x_0\}$ *est u-accessible ;*

(ii) Si H est un historique u-accessible tel que $\sigma_\exists(H) \in \Delta(\max H)$, et tel que $\max H \notin \sigma_\exists(H)$, alors pour tout $y \in \sigma_\exists(H)$, $H \cup \{y\}$ est u-accessible ;

(iii) Si $\mathcal{H}$ est une chaîne non vide d'historiques u-accessibles pour l'ordre préfixe, alors $\bigcup \mathcal{H}$ est u-accessible ;

(iv) Si H est un historique u-accessible admettant une borne supérieure, alors $H \cup \{\sup H\}$ est u-accessible.

Démonstration. Les règles de propagation (i) et (iv) sont évidentes car elles correspondent exactement à celles qui définissent une partie. De même, la règle (ii) est une conséquence directe du lemme 4.24. En effet, tout témoin d'u-accessibilité pour H peut être modifié en un témoin d'u-accessibilité qui envoie $\sigma_\exists(H)$ sur y, et donc est un témoin d'u-accessibilité pour $H \cup \{y\}$.

Pour le cas de la règle (iii), soit $(\sigma_{\forall,H})_{H\in\mathcal{H}}$ une famille de témoins d'u-accessibilité pour les éléments de $\mathcal{H}$. Alors un témoin d'u-accessibilité pour $\bigcup \mathcal{H}$ est :

$$\sigma_\forall(X) = \begin{cases} \sigma_{\forall,H}(X) & \text{si } H \in \mathcal{H}, X \in \mathcal{C}(\sigma_\exists, H) \\ x_0 & \text{sinon} \end{cases}$$

Premièrement, remarquons que cette définition n'est pas ambiguë. En effet, on montre pour X et deux historiques $H_1, H_2 \in \mathcal{H}$ tels que $X \in \mathcal{C}(\sigma_\exists, H_1) \cap \mathcal{C}(\sigma_\exists, H_2)$, on a $\sigma_{\forall,H_1}(X) = \sigma_{\forall,H_2}(X)$. Pour cela, remarquons que H_1, H_2 sont comparables pour l'ordre préfixe. Par symétrie, on peut supposer que $H_1 \sqsubseteq H_2$. En particulier, $\sigma_{\forall,H_2}$ est également un témoin d'u-accessibilité pour H_1. Comme $X \in \mathcal{C}(\sigma_\exists, H_1)$, le lemme 4.23 donne l'égalité voulue.

Ensuite, le lemme 4.23 montre immédiatement que $\sigma_\forall$ est un témoin d'u-accessibilité pour tout $H \in \mathcal{H}$, car il coïncide avec $\sigma_{\forall,H}$ sur $\mathcal{C}(\sigma_\exists, H)$. En particulier, la règle (iii) définissant les parties permet de conclure que c'est donc un témoin d'u-accessibilité pour $\bigcup \mathcal{H}$. □

En fait, nous pouvons montrer que l'application des quatre règles ci-dessus est suffisante pour construire un historique perdant s'il existe. Nous en déduisons une version plus puissante du théorème 4.26, qui caractérise les stratégies gagnantes sans faire aucune référence aux stratégies universelles.

Définition 4.28. Un historique est dit *fortement u-accessible* s'il appartient au plus petit ensemble clos par les règles de propagation décrites dans le lemme 4.27. En particulier, tout historique fortement u-accessible est u-accessible.

Théorème 4.29. *Une stratégie existentielle $\sigma_\exists$ est gagnante pour un point de départ x_0 et un ensemble cible Q si et seulement s'il n'existe pas d'historique fortement u-accessible pour $\sigma_\exists, x_0$ et perdant pour Q.*

Démonstration. Par le théorème 4.26, il suffit de vérifier que s'il existe un historique u-accessible H perdant pour Q, il existe également un historique fortement u-accessible qui est perdant pour Q. Soit $\sigma_\forall$ un témoin d'u-accessibilité pour H, $\mathcal{F}$ la partie associée, et (x_α) la séquence des coups de la partie. Il existe donc un ordinal β minimal tel que l'historique $H_0 = \{x_\alpha \mid \alpha < \beta\}$ soit perdant. En particulier, H_0 est le plus petit historique perdant dans $\mathcal{F}$. Nous allons démontrer que H_0 est bien fortement u-accessible.

Pour cela, il nous suffit de montrer que l'ensemble d'historiques

$$A = \{H \in \mathcal{F} \mid H \sqsubseteq H_0 \Rightarrow H \text{ fortement u-accessible}\}$$

est clos par les règles qui définissent $\mathcal{F}$. Nous nous intéressons uniquement à la règle (ii) car les autres sont rigoureusement identiques à celles qui propagent l'u-accessibilité forte. Soit donc $H_1 \in A$ tel que les choix associés des stratégies soient valides, et montrons que $H_2 = H_1 \cup \{\sigma_\forall(\sigma_\exists(H_1)\} \in A$. On peut supposer que $H_2 \sqsubseteq H_0$ et que $H_2 \neq H_1$. Dans ce cas, H_1 ne peut pas être perdant par minimalité de H_0, donc $\max H_1 \notin \sigma_\exists(H_1)$. En particulier, la règle de propagation de l'u-accessibilité forte s'applique et $H_2 \in A$. □

Remarque 4.30. Le théorème 4.29 montre que comme annoncé dans la section précédente, la capacité du joueur universel à empêcher une stratégie d'être gagnante ne change pas qu'il ait accès à de la mémoire ou non. En effet, supposons qu'un contre-exemple à la propriété d'être une stratégie gagnante soit donné sous la forme d'une stratégie universelle capable d'accéder à l'historique pour prendre ses décisions, et que l'on modifie la notion de partie de manière à prendre en compte cette capacité. Alors, les deux lemmes ci-dessus permettent de montrer que tous les historiques de la partie associée jusqu'à l'historique perdant minimal sont fortement u-accessibles. En particulier, la stratégie existentielle n'était pas gagnante contre un adversaire sans mémoire non plus.

Remarquons par ailleurs que le théorème 4.29 justifie complètement la remarque 4.14 que nous avions faite dans la section précédente, portant sur les conditions dans lesquelles le joueur existentiel perds ainsi que sur l'inutilité des transitions permettant de rester sur place. En effet, ce théorème énonce précisément les conditions sous lesquelles une stratégie existentielle n'est pas gagnante, et permet de déduire l'inutilité de ces transitions comme conséquence directe du théorème. Nous aurions donc pu proposer une formalisation des jeux qui élimine directement ces transitions. Cependant, ce choix plus large facilite la construction de jeux en pratique.

Corollaire 4.31. *Soit $\mathbb{G}' = (G, \preccurlyeq, \Delta')$ le jeu défini par*

$$\Delta'(x) = \Delta(x) \backslash \{X \subseteq G \mid x \in X\}$$

Alors une stratégie existentielle $\sigma_\exists$ est gagnante pour l'ensemble de départ P et l'ensemble cible Q vis-à-vis du jeu $\mathbb{G}$ si et seulement si elle l'est vis-à-vis du jeu $\mathbb{G}'$.

Démonstration. Notons que les règles de propagation définissant les historiques fortement u-accessibles ne prennent pas en compte les transitions enlevées pour former $\mathbb{G}'$. En particulier, les historiques fortement u-accessible pour $\sigma_\exists$ et un état x vis-à-vis du jeu $\mathbb{G}$ sont les mêmes que les historiques fortement u-accessibles pour $\sigma_\exists, x$ vis-à-vis du jeu $\mathbb{G}'$. Le résultat suit alors directement du théorème 4.29. □

L'ensemble des historiques fortement u-accessibles possède une autre propriété intéressante. Nous pouvons en effet reconstruire entièrement une stratégie gagnante depuis un ensemble d'historiques fortement u-accessibles. Cela signifie qu'un tel ensemble peut être utilisé comme témoin alternatif de l'existence d'une stratégie gagnante. Nous formalisons cette propriété par la notion d'invariant de victoire, ce qui permet ainsi de donner une définition alternative des garanties.

Définition 4.32. Soient P, Q deux ensembles d'états de $\mathbb{G}$. Un *invariant de victoire* pour P, Q est un ensemble d'historiques I tel que :

(i) Pour tout $x \in P$, $\{x\} \in I$
(ii) Pour tout historique jouable $H \in I$, si $H \cap Q = \emptyset$ il existe un ensemble $X \in \Delta(\max H)$ tel que $\max H \notin X$ et pour tout $x \in X$, $H \cup \{x\} \in I$
(iii) Pour toute chaîne non vide d'historique $\mathcal{H} \subseteq I$ pour l'ordre préfixe, $\bigcup \mathcal{H} \in I$
(iv) Pour tout $H \in I$ admettant une borne supérieure, $H \cup \{\sup H\} \in I$

Théorème 4.33. *Soient P, Q deux ensembles d'états de $\mathbb{G}$. Alors P garantit Q dans $\mathbb{G}$ si et seulement s'il existe un invariant de victoire pour P, Q.*

Démonstration. Tout d'abord, remarquons que le sens direct est immédiat. En effet, fixons une stratégie gagnante $\sigma_\exists$ pour l'ensemble de départ P et l'ensemble cible Q. Un invariant de victoire pour P, Q est alors l'ensemble des historiques (fortement) u-accessibles pour $\sigma_\exists$ et n'importe quel point de départ de P.

Montrons maintenant le sens réciproque. Soit I un invariant de victoire pour P, Q. Nous construisons explicitement une stratégie gagante $\sigma_\exists$ pour P, Q de la manière suivante. Étant donné un historique jouable H, $\sigma_\exists$ est défini comme étant un ensemble X qui préserve I au sens de la condition (ii) si un tel X existe, et $\emptyset$ sinon. On vérifie alors immédiatement que I contient tous les historiques fortement u-accessibles associés à $\sigma_\exists$ et n'importe quel point de départ $x_0 \in P$, car il est clos pour les règles de propagation correspondantes. Comme aucun élément de I ne peut être un historique perdant pour $\sigma_\exists$ et Q, $\sigma_\exists$ est bien gagnante. □

Exemple 4.34. Considérons de nouveau le cas du jeu $\overline{\mathbb{N}}_3$. Nous avions énoncé dans l'exemple 4.13 qu'une stratégie gagnante existe pour un point de départ x et un ensemble cible Q si et seulement si Q contient ∞ ou quatre entiers consécutifs supérieurs ou égaux à x. Nous pouvons aisément démontrer le sens réciproque de cette implication à l'aide d'invariants de victoire. En effet, l'ensemble de tous les historiques de $\overline{\mathbb{N}}_3$ est un invariant de victoire pour $\overline{\mathbb{N}}$ et $\{\infty\}$. De même, pour tout $x \in \mathbb{N}$, un invariant de victoire possible pour $\{y \in \mathbb{N} \mid y \leq x\}$ et $\{x, x+1, x+2, x+3\}$ est l'ensemble des historiques de $\overline{\mathbb{N}}_3$ majorés par $x + 3$.

La définition que nous avons donnée pour les invariants de victoire permet d'utiliser des invariants qui contiennent des historiques non pertinents, car ils contiennent dans leur passé des éléments de l'ensemble cible. Nous montrons qu'il est toujours possible de se réduire au cas où il n'y a pas de tels historiques.

Définition 4.35. Un invariant de victoire I pour P, Q est dit *normalisé* si pour tout $H \in I$, tout $x \in H \cap Q$ est maximal dans H.

Lemme 4.36. *Soit I un invariant de victoire pour P, Q deux ensembles d'états de $\mathbb{G}$. Alors l'ensemble J défini par :*

$$H \in J \Leftrightarrow H \in I \wedge \forall x \in H \cap Q.\ x \text{ est maximal dans } H$$

est le plus grand invariant de victoire normalisé pour P, Q qui soit contenu dans I.

Démonstration. Le fait que J soit le plus grand possible est évident car on a uniquement enlevé les historiques qui contredisent la normalisation. Il nous suffit donc de vérifier que J satisfait bien les quatre règles de propagation :

(i) immédiat car H n'a qu'un élément.

(ii) immédiat car on part d'un historique qui ne possède pas d'élément commun avec Q.

(iii) soit $\mathcal{H} \subseteq J$ une chaîne non vide. La propriété étant immédiate si $\bigcup \mathcal{H} \cap Q$ est vide, supposons que cet ensemble contient un élément x. Il existe donc $H_0 \in \mathcal{H}$ tel que $x \in H_0$. Mais pour tout historique $H \in \mathcal{H}$ plus grand que H_0, $x \in H \cap Q$, donc $x = \max H$. En particulier, $H = H_0$, donc $\bigcup \mathcal{H} = H_0 \in J$.

(iv) comme $H \in J$ on en déduit $H \cap Q \subseteq \{\sup H\}$. En particulier,

$$(H \cup \{\sup H\}) \cap Q \subseteq \{\sup H\} = \{\max (H \cup \{\sup H\})\}$$

□

Remarque 4.37. Cette « normalisation » des invariants de victoire n'est pas forcément optimale. Nous pourrions par le même raisonnement imposer que le minimum de tout historique de l'invariant existe et soit dans l'ensemble de départ, ou encore que tout historique de l'invariant soit bien fondé. Cependant, les autres restrictions possibles ne se sont pas avérées utiles. De plus, un invariant de victoire peut ne pas contenir de sous-ensemble minimal qui soit également un invariant de victoire, ce qui limite les possibilités de normalisation.

4.4 Théorème de simulation pour les jeux

Dans cette section, nous démontrons un théorème de transfert des garanties d'un jeu vers un autre sous des condition locales. Ce théorème nous permet notamment de passer par un jeu plus structuré pour démontrer une garantie donnée. En particulier, nous pouvons utiliser ce théorème pour passer par un jeu contenant des informations supplémentaires dans ses états. Un tel enrichissement est analogue à l'ajout de variables auxiliaires ou de code fantôme pour la vérification de programmes.

On dit qu'une relation induit une simulation entre deux jeux si elle permet de transporter les garanties valides de l'un à l'autre.

Définition 4.38 (Simulation)**.** Étant donnés deux jeux $\mathbb{G}_1, \mathbb{G}_2$ et une relation $\mathcal{R}$ entre les domaines respectifs des deux jeux, on dit que $\mathcal{R}$ induit une *simulation* de $\mathbb{G}_1$ par $\mathbb{G}_2$ si elle transporte les garanties de $\mathbb{G}_1$ dans $\mathbb{G}_2$, autrement dit si pour toute paire (P, Q) on a :

$$\langle P \hookrightarrow Q \rangle_{\mathbb{G}_1} \Rightarrow \langle \mathcal{R}[P] \hookrightarrow \mathcal{R}[Q] \rangle_{\mathbb{G}_2}$$

où $\mathcal{R}[X] = \{y \mid \exists x \in X. x\mathcal{R}y\}$ est l'image d'un ensemble par $\mathcal{R}$.

Exemple 4.39. Le résultat énoncé dans le corollaire 4.31 implique directement deux simulations. Plus précisément, soit $\mathbb{G} = (G, \preccurlyeq, \Delta)$ un jeu et $\mathbb{G}' = (G, \preccurlyeq, \Delta')$ le jeu défini par

$$\Delta'(x) = \Delta(x) \backslash \{X \subseteq G \mid x \in X\}$$

Alors la relation d'égalité induit à la fois une simulation de $\mathbb{G}$ par $\mathbb{G}'$ et une simulation de $\mathbb{G}'$ par $\mathbb{G}$.

La définition d'une simulation entre jeux correspond précisément à la notion souhaitée. Notamment, si $\mathcal{R}$ correspond à une projection, une simulation permet de transporter toute garantie d'un jeu enrichi vers une garantie d'un jeu moins structuré. Cependant, il faut d'abord pouvoir prouver qu'une relation induit une simulation. Pour cela, nous utilisons des conditions suffisantes plus simples à vérifier car elles sont structurelles.

Définition 4.40 (Simulation étape par étape)**.** Étant donnée une relation $\mathcal{R}$ entre les supports de deux jeux $\mathbb{G}_1 = (G_1, \preccurlyeq_1, \Delta_1)$ et $\mathbb{G}_2 = (G_2, \preccurlyeq_2, \Delta_2)$, on dit que $\mathcal{R}$ induit une *simulation étape par étape* de $\mathbb{G}_1$ par $\mathbb{G}_2$ si les deux conditions suivantes sont vérifiées :

(i) Pour toute transition $x \in G_1, X \in \Delta_1(x)$ de $\mathbb{G}_1$, $\langle \mathcal{R}[\{x\}] \hookrightarrow \mathcal{R}[X] \rangle_{\mathbb{G}_2}$ est valide

(ii) Étant donnés H un historique de $\mathbb{G}_1$ et f une fonction croissante de $(H, \preccurlyeq_1)$ dans $(G_2, \preccurlyeq_2)$, si pour tout $x \in H$ on a $x\mathcal{R}f(x)$, et si $f[H]$ admet une borne supérieure s_2, alors la garantie

$$\langle \{s_2\} \hookrightarrow \mathcal{R}[\{s_1 \mid s_1 \text{ borne supérieure de } H\}] \rangle_{\mathbb{G}_2}$$

est valide.

Théorème 4.41. *Si $\mathcal{R}$ induit une simulation étape par étape de $\mathbb{G}_1$ par $\mathbb{G}_2$, alors $\mathcal{R}$ induit une simulation de $\mathbb{G}_1$ par $\mathbb{G}_2$.*

Les deux conditions correspondent respectivement au transport des étapes régulières et des étapes limites. La condition de transport des étapes régulières est naturelle, et évidemment une condition nécessaire pour qu'une relation soit une simulation. La condition pour les étapes limites est utilisée pour reconstruire explicitement des invariants de victoire. Elle exprime que lorsque l'on a reflété dans $\mathbb{G}_2$ une quantité limite d'étapes, si la partie dans $\mathbb{G}_2$ peut continuer via une borne supérieure, alors il faut atteindre un état qui est relié correctement à l'état de la partie dans $\mathbb{G}_1$. Cela revient à resynchroniser la relation entre les deux jeux au niveau des limites. Remarquons par ailleurs que le choix de la victoire du joueur existentiel lorsque la complétude est en défaut est utile ici, car ce choix évite de devoir ajouter aux hypothèses l'existence de la borne supérieure dans $\mathbb{G}_2$.

Exemple 4.42. En utilisant le théorème 4.41, nous pouvons démontrer que tout jeu est « équivalent », au sens des garanties, à un jeu où les transitions existentielles et universelles sont parfaitement découplées. Par exemple, considérons de nouveau le jeu $\overline{\mathbb{N}}_3$, et considérons le jeu $\mathbb{G} = (\overline{\mathbb{N}}, \leq, \Delta)$ défini par

— $\Delta(2n) = \{\{2n + 2k + 1\} \mid k \in \{0; 1; 2\}\}$
— $\Delta(2n + 1) = \{\{2n + 2k + 2 \mid k \in \{0; 1; 2; 3\}\}$
— $\Delta(\infty) = \emptyset$

Essentiellement, cette traduction revient à coder les états de $\overline{\mathbb{N}}_3$ par les entiers pairs, et les transitions possibles par les entiers impairs. Considérons maintenant la relation $\mathcal{D}_1$ définie par $n\mathcal{D}_1 m \Leftrightarrow m = 2n$. Alors nous pouvons facilement vérifier que $\mathcal{D}_1$ induit une simulation étape par étape de $\overline{\mathbb{N}}_3$ par $\mathbb{G}$, et donc une simulation. Notons cependant que $\mathcal{D}_1^{-1}$ n'induit pas de simulation de $\mathbb{G}$ par $\overline{\mathbb{N}}_3$, car cela impliquerait qu'il est possible de transporter les transitions depuis les entiers pairs dans $\mathbb{G}$ vers des garanties de $\overline{\mathbb{N}}_3$ dont l'ensemble cible est vide mais pas l'ensemble de départ, ce qui n'est pas possible. Cela ne nous empêche pas de construire une autre relation $\mathcal{D}_2$, qui induit une simulation étape par étape de $\mathbb{G}$ par $\overline{\mathbb{N}}_3$. Nous définissons pour cela $m\mathcal{D}_2 n$ par la propriété que $m \in \{2n; 2n+1; 2n+3; 2n+5\}$. Notons que comme précédemment, $\mathcal{D}_2$ n'induit pas la simulation inverse. Par contre, la paire des simulations induites par $\mathcal{D}_1$ et $\mathcal{D}_2$ induit une bijection entre les garanties valides de $\overline{\mathbb{N}}_3$ et les garanties valides de $\mathbb{G}$ dont les ensembles de départ/cible ne contiennent que des nombres pairs (ou l'infini).

Nous décomposons la preuve du théorème 4.41 en quatre étapes, via la construction de jeux intermédiaires. Tout d'abord, nous assurons que le jeu d'origine contient suffisamment de bornes supérieures en complétant son support. Ensuite, nous démontrons une simulation vers un enrichissement du second jeu. Nous montrons de plus que cet enrichissement admet des invariants de victoire assez contraints. Enfin, nous montrons une simulation qui utilise ces contraintes pour oublier l'enrichissement. Le théorème s'ensuit du fait que la simulation est une notion compositionnelle, ce qui permet de fusionner les différentes simulations en une seule.

À chaque fois que nous démontrerons une simulation, nous montrerons également que cette simulation peut être explicitée comme une traduction des invariants de victoire d'une garantie. En composant les différentes étapes, cela nous donne une correspondance entre une simulation étape par étape et une traduction des invariants de victoire du premier jeu vers le second.

Lemme 4.43 (Compositionnalité)**.** *Soient $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_3$ trois jeux et $\mathcal{R}_{12}, \mathcal{R}_{23}$ deux relations induisant respectivement une simulation de $\mathbb{G}_1$ par $\mathbb{G}_2$ et une simulation de $\mathbb{G}_2$ par $\mathbb{G}_3$. Alors $\mathcal{R}_{12} \circ \mathcal{R}_{23}$ induit une simulation de $\mathbb{G}_1$ par $\mathbb{G}_3$.*

Démonstration. Soient P, Q deux ensembles, alors :

$$\langle P \hookrightarrow Q\rangle_{\mathbb{G}_1} \Rightarrow \langle \mathcal{R}_{12}[P] \hookrightarrow \mathcal{R}_{12}[Q]\rangle_{\mathbb{G}_2} \Rightarrow \langle \mathcal{R}_{23}[\mathcal{R}_{12}[P]] \hookrightarrow \mathcal{R}_{23}[\mathcal{R}_{12}[Q]]\rangle_{\mathbb{G}_3}\,.$$

□

Le reste de cette section est organisé de la manière suivante. Dans la sous-section 4.4.1, nous introduisons les définitions utilisées pour prouver le théorème de simulation, ainsi que les énoncés des théorèmes intermédiaires. Dans les quatre sous-sections qui suivent, nous prouvons ces théorèmes.

4.4.1 Définitions et énoncés

La première étape de la chaîne de composition recherchée est de modifier le support du jeu de départ de façon à ce que toutes les chaînes possèdent une borne

supérieure. Nous réalisons cette étape en décomposant la simulation par l'intermédiaire d'un jeu complété. De cette manière, nous n'aurons pas de problème par la suite pour apparier les comportements des deux jeux dans le cas limites.

Définition 4.44 (Jeu complété)**.** Étant donné un jeu $\mathbb{G} = (G, \preccurlyeq, \Delta)$, le *jeu complété* de $\mathbb{G}$ est le jeu $\overline{\mathbb{G}}$ ayant pour support les historiques de $\mathbb{G}$ ordonnés par ordre préfixe, et pour ensemble de transitions $\overline{\Delta}$ défini par :

$$\overline{\Delta}(H) = \begin{cases} \{\{H \cup \{x\} \mid x \in X\} \mid X \in \Delta(\max H)\} & \text{si } H \text{ jouable} \\ \{\{H \cup \{x\} \mid x \text{ borne supérieure de } H\}\} & \text{si } H \text{ limite} \end{cases}$$

Lemme 4.45. *Le support d'un jeu complété est chaîne-complet, c'est-à-dire que toute chaîne admet une borne supérieure.*

Démonstration. Il est immédiat que pour $\mathcal{H}$ une chaîne d'historiques, $\sup \mathcal{H} = \bigcup \mathcal{H}$. □

Définition 4.46. Soit $\mathbb{G}$ un jeu. Nous définissons la relation $\overline{\mathcal{C}}$ entre les supports de $\mathbb{G}$ et $\overline{\mathbb{G}}$ par $x \overline{\mathcal{C}} H$ si et seulement si x est le maximum de H.

Définition 4.47. Soit $\mathcal{R}$ une relation entre les supports de deux jeux $\mathbb{G}_1, \mathbb{G}_2$. Nous définissons une relation $\overline{\mathcal{R}}$ entre les supports de $\overline{\mathbb{G}}_1$ et de $\mathbb{G}_2$ par $H \overline{\mathcal{R}} y$ si et seulement si H admet une borne supérieure x et pour cette borne supérieure $x \mathcal{R} y$.

Théorème 4.48. *Soit $\mathbb{G}_1, \mathbb{G}_2$ deux jeux. La relation $\overline{\mathcal{C}}$ associée à $\mathbb{G}_1$ induit une simulation de $\mathbb{G}_1$ par $\overline{\mathbb{G}}_1$. De plus, si $\mathcal{R}$ est une relation qui induit une simulation étape par étape de $\mathbb{G}_1$ par $\mathbb{G}_2$, alors la relation associée $\overline{\mathcal{R}}$ induit une simulation étape par étape de $\overline{\mathbb{G}}_1$ par $\mathbb{G}_2$.*

Le théorème 4.48 est démontré dans la sous-section 4.4.2. La seconde étape consiste à décomposer la simulation en deux parties par l'intermédiaire d'un jeu enrichi. Ce jeu enrichi possède une structure analogue à celle du jeu cible, mais contient une « case mémoire supplémentaire » pour représenter l'état du jeu simulé. Cet état permet de réaliser une simulation naturellement.

Définition 4.49. Soient $\mathbb{G} = (G, \preccurlyeq_G, \Delta)$ un jeu, $(O, \preccurlyeq_O)$ un ensemble partiellement ordonné et $\mathcal{R} \subseteq O \times G$ une relation. Le $(O, \preccurlyeq_O, \mathcal{R})$-*enrichissement* de $\mathbb{G}$ est le jeu défini sur le support $(O \times G, \preccurlyeq_O \times \preccurlyeq_G)$ par l'ensemble de transitions :

$$\Delta(x, y) = \{\{(x', y)\} \mid x \preccurlyeq_O x' \wedge x' \mathcal{R} y\} \cup \{\{(x, y') \mid y' \in Y\} \mid Y \in \Delta(y)\}$$

Définition 4.50. Soit $\mathbb{G}_1 = (G_1, \preccurlyeq_1, \Delta_1)$ et $\mathbb{G}_2 = (G_2, \preccurlyeq_2, \Delta_2)$ deux jeux, $\mathcal{R}$ une relation entre leurs supports. Nous définissons la relation $\overrightarrow{\mathcal{R}}$ entre le support de $\mathbb{G}_1$ et le support du $(G_1, \preccurlyeq_1, \mathcal{R})$-enrichissement de $\mathbb{G}_2$ par $x \overrightarrow{\mathcal{R}} (x', y)$ si et seulement si $x = x'$ et $x' \mathcal{R} y$.

Théorème 4.51. *Soient $\mathbb{G}_1 = (G_1, \preccurlyeq_1, \Delta_1)$, $\mathbb{G}_2 = (G_2, \preccurlyeq_2, \Delta_2)$ deux jeux, $\mathcal{R} \subseteq G_1 \times G_2$ une relation induisant une simulation étape par étape de $\mathbb{G}_1$ par $\mathbb{G}_2$, et $\mathbb{G}_p$ le $(G_1, \preccurlyeq_1, \mathcal{R})$-enrichissement de $\mathbb{G}_2$. Alors la relation associée $\overrightarrow{\mathcal{R}}$ induit une simulation de $\mathbb{G}_1$ par $\mathbb{G}_p$.*

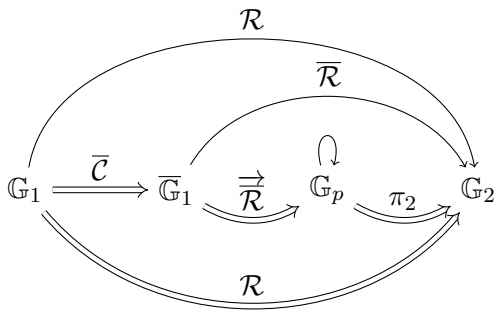

FIGURE 4.2 – Preuve schématique du théorème 4.41

Le théorème 4.51 est démontré dans la sous-section 4.4.3. A partir de ces résultats, il ne reste plus qu'à prouver que la projection vers le jeu cible induit une simulation. Pour cela, nous restreignons d'abord l'ensemble des invariants de victoire possibles à des invariants suffisamment déterministes. Nous garantissons ainsi qu'un historique de l'invariant peut être uniquement reconstruit depuis sa projection.

Définition 4.52 (Relations de projection)**.** Étant donné deux ensembles A_1 et A_2, les relations de projections $\pi_i \subseteq (A_1 \times A_2) \times A_i$ sont les relations définies par $(x_1, x_2)\pi_i y$ si et seulement si $x_i = y$.

Définition 4.53. Soient $\mathbb{G}$ un jeu, $(O, \preccurlyeq_O)$ un ensemble partiellement ordonné et $\mathcal{R}$ une relation. Nous disons qu'un invariant de victoire I du $(O, \preccurlyeq_O, \mathcal{R})$-enrichissement de $\mathbb{G}$ est *réductible* si pour tout historique $H \in I$, l'ensemble

$$\{H' \in I \mid H \sqsubseteq H' \wedge \pi_2[H] = \pi_2[H']\}$$

admet un maximum, et si tout historique H' préfixe de H appartient également à I.

Théorème 4.54. *Pour tout jeu $\mathbb{G}$, $\mathbb{G}_p$ un $(O, \preccurlyeq_O, \mathcal{R})$-enrichissement de $\mathbb{G}$, si $\langle P \hookrightarrow Q\rangle_{\mathbb{G}_p}$ est vérifiée pour deux ensembles P, Q, alors P, Q admet un invariant de victoire réductible.*

Le théorème 4.54 est démontré dans la sous-section 4.4.4. Nous pouvons ensuite vérifier que la projection induit bien une simulation du jeu enrichi par le jeu cible.

Théorème 4.55. *Soit $(O, \preccurlyeq_O)$ un ordre chaîne-complet, $\mathbb{G} = (G, \preccurlyeq, \Delta)$ un jeu, $\mathcal{R} \subseteq O \times G$ une relation, et $\mathbb{G}_p = (G_p, \preccurlyeq_p, \Delta_p)$ le $(O, \preccurlyeq_O, \mathcal{R})$-enrichissement de $\mathbb{G}$. Alors la relation de projection π_2 induit une simulation de $\mathbb{G}_p$ par $\mathbb{G}$.*

Le théorème 4.55 est démontré dans la sous-section 4.4.5. Il est immédiat de vérifier que les quatre théorèmes énoncés ci-dessus et le caractère compositionnel de la simulation démontrent le théorème 4.41. Le schéma de la preuve est résumé dans la figure 4.2. Les flèches simples représentent les simulations étape par étape et les flèches doubles les simulations générales. La boucle représente la restriction aux invariants réductibles.

4.4.2 Décomposition via le complété

Dans cette sous-section, nous démontrons le théorème 4.48.

Lemme 4.56. *Soit $\mathbb{G} = (G, \preccurlyeq, \Delta)$ un jeu. La relation $\overline{\mathcal{C}}$ associée à $\mathbb{G}$ induit une simulation de $\mathbb{G}$ par $\overline{\mathbb{G}}$.*

De plus, cette simulation peut être rendue explicite comme une traduction des invariants de victoire de la manière suivante. Si I est un invariant de victoire normalisé pour $P, Q \subseteq G$, alors nous définissons un invariant de victoire $\overline{I}$ pour $\overline{\mathcal{C}}[P], \overline{\mathcal{C}}[Q]$ de la manière suivante. L'historique $\mathcal{H}$ de $\overline{\mathbb{G}}$ appartient à $\overline{I}$ si et seulement si $\mathcal{H}$ admet comme minimum un historique jouable H_0 de $\mathbb{G}$, tel que :

$$(\bigcup \mathcal{H}) \backslash (H_0 \backslash \{\max H_0\}) \in I$$

Démonstration. Il suffit de vérifier la traduction donnée ci-dessus. Soient P, Q deux ensembles tels que $\langle P \hookrightarrow Q \rangle_{\mathbb{G}}$ soit valide, et I un invariant de victoire pour P, Q. Nous montrons que $\overline{I}$ est bien un invariant de victoire pour $\overline{\mathcal{C}}[P], \overline{\mathcal{C}}[Q]$ en vérifiant chacune des conditions de la définition 4.32.

(i) Pour tout $H \in \overline{\mathcal{C}}[P]$, $\{H\} \in \overline{I}$ est vérifié car H admet un maximum $x \in P$, et la contrainte se réduit en $\{x\} \in I$, qui vient du fait que I soit un invariant de victoire pour P, Q.

(iv) Pour tout $\mathcal{H} \in \overline{I}$, comme nous avons $\sup \mathcal{H} = \bigcup \mathcal{H}$, nous vérifions que $\bigcup \mathcal{H} = \bigcup (\mathcal{H} \cup \{\sup \mathcal{H}\})$. En particulier, la condition d'appartenance à $\overline{I}$ est identique pour $\mathcal{H}$ et $\mathcal{H} \cup \{\sup \mathcal{H}\}$, donc cette règle de propagation est vérifiée.

(iii) Pour toute chaîne non vide $\mathcal{J} \subseteq \overline{I}$, on a

$$\bigcup \Big(\bigcup \mathcal{J} \Big) = \bigcup_{H \in \mathcal{H} \in \mathcal{J}} H = \bigcup_{\mathcal{H} \in \mathcal{J}} \Big(\bigcup \mathcal{H} \Big)$$

En particulier, pour tout $\mathcal{H} \in \mathcal{J}$ on a $(\bigcup \mathcal{H}) \backslash (H_0 \backslash \{x_0\}) \in I$, où $H_0 = \min \mathcal{H}$ et $x_0 = \max H_0$. Comme $\mathcal{J}$ est une chaîne, H_0 ne dépend pas de $\mathcal{H}$, et donc est bien le minimum de $\bigcup \mathcal{J}$. En particulier,

$$\Big(\bigcup \Big(\bigcup \mathcal{J} \Big) \Big) \setminus (H_0 \backslash \{x_0\}) = \bigcup_{\mathcal{H} \in \mathcal{J}} \Big(\Big(\bigcup \mathcal{H} \Big) \setminus (H_0 \backslash \{x_0\}) \Big)$$

Donc comme I satisfait également la règle (iii), $\bigcup \mathcal{J} \in \overline{I}$.

(ii) Soit $\mathcal{H} \in \overline{I}$ un historique jouable tel que $\mathcal{H} \cap \overline{\mathcal{C}}[Q] = \emptyset$, $H_0 = \min \mathcal{H}$, $x_0 = \max H_0$. Il y a deux possibilités selon la nature de $H_1 = \max \mathcal{H}$.

— Soit H_1 est limite. Alors l'unique élément $X_0 \in \overline{\Delta}(H_1)$ vérifie les hypothèses voulues. En effet, pour tout $H_2 = H_1 \cup \{\sup H_1\} \in X_0$, la condition $\mathcal{H} \cup \{H_2\} \in \overline{I}$ se réduit à $H_2 \backslash (H_0 \backslash \{x_0\}) \in I$. Celle-ci est valide car $\sup (H_1 \backslash (H_0 \backslash \{x_0\})) = \sup H_1$, et I satisfait la règle de propagation (iv) pour les invariants de victoire, permettant ainsi d'ajouter la borne supérieure. De plus, on vérifie effectivement que $H_1 \notin X_0$ car H_1 étant limite, il ne peut contenir sa borne supérieure.

— Soit H_1 est jouable. Comme I est normalisé et $H_1 \notin \overline{\mathcal{C}}[Q]$, alors l'historique $H_1' = H_1 \backslash (H_0 \backslash \{x_0\})$ n'a pas d'élément commun avec Q. Comme $H_1' \in I$ par hypothèse, la règle de propagation (ii) appliquée à H_1' nous donne un témoin $X \in \Delta(\max H_1)$ de la propriété associée. Alors un témoin possible pour $\overline{I}$ est $\mathcal{X} = \{H_1 \cup \{x\} \mid x \in X\} \in \overline{\Delta}(H_1)$. En effet,

remarquons tout d'abord que $H_1 \notin \mathcal{X}$, car sinon il existerait $x \in X$ tel que $x \in H_1$. Par la relation de progression, $x = \max H_1 \in X$, ce qui est contradictoire.
Il nous reste donc à vérifier que pour tout $H_2 = H_1 \cup \{x\} \in \mathcal{X}$, on a $\mathcal{H} \cup \{H_2\} \in \overline{I}$. Mais une fois encore cette contrainte est équivalente à $H_2 \backslash (H_0 \backslash \{x_0\}) = H_1' \cup \{x\} \in I$, ce qui se déduit des propriétés connues de X.

□

Lemme 4.57. *Soient $\mathbb{G}_1 = (G_1, \preccurlyeq_1, \Delta_1)$ et $\mathbb{G}_2 = (G_2, \preccurlyeq_2, \Delta_2)$ deux jeux et $\mathcal{R}$ une relation induisant une simulation étape par étape de $\mathbb{G}_1$ par $\mathbb{G}_2$. Alors la relation associée $\overline{\mathcal{R}}$ induit une simulation étape par étape de $\overline{\mathbb{G}_1}$ par $\mathbb{G}_2$.*

Démonstration. Vérifions les deux conditions pour que $\overline{\mathcal{R}}$ soit une simulation étape par étape.

— Étapes régulières : soit H un état de $\overline{\mathbb{G}}_1$, $\mathcal{X} \in \overline{\Delta}(H)$. Notre objectif est de montrer que $\left\langle \overline{\mathcal{R}}[\{H\}] \hookrightarrow \overline{\mathcal{R}}[\mathcal{X}] \right\rangle$. Il y a deux cas possibles en fonction de la nature de H. Si H est un historique limite, alors on a $\overline{\mathcal{R}}[\{H\}] = \mathcal{R}[\{x \mid x = \sup H\}] = \overline{\mathcal{R}}[\mathcal{X}]$ par définition. La garantie est alors immédiate car les ensembles départ et cible coïncident, via la propriété 4.19.
Dans le cas où H est un historique jouable, nous obtenons en déroulant les définitions qu'il existe $X \in \Delta_1(\max H)$ tel que $\overline{\mathcal{R}}[\mathcal{X}] = \mathcal{R}[X]$, et que $\overline{\mathcal{R}}[\{H\}] = \mathcal{R}[\{\max H\}]$. Comme $\mathcal{R}$ est une simulation étape par étape, la garantie voulue est également valide.

— Étapes limites : soit $\mathcal{H}$ un historique de $\overline{\mathbb{G}}_1$, F une fonction croissante de $(\mathcal{H}, \sqsubseteq)$ dans $(G_2, \preccurlyeq_2)$ telle que pour tout $H \in \mathcal{H}$, $H \overline{\mathcal{R}} F(H)$, et s_2 la borne supérieure de $F(\mathcal{H})$ dont nous supposons l'existence. Nous nous ramenons à la simulation étape par étape de $\mathbb{G}_1$ par $\mathbb{G}_2$ par les définitions suivantes :

$$\begin{array}{lcll} H_0 & = & \{\sup H \mid H \in \mathcal{H}\} & \\ f(x) & = & F(H) & \text{où } H = \max\{H \in \mathcal{H} \mid \sup H = x\} \end{array}$$

Notons que H_0 et f sont bien définis : comme F induit une relation par $\overline{\mathcal{R}}$ depuis tout historique de $\mathcal{H}$, ceux-ci admettent tous une borne supérieure. De plus, comme $\mathcal{H}$ est une chaîne le maximum définissant f existe bien pour tout $x \in H_0$. En effet, $\mathcal{H}$ ne peut alors pas contenir plus de deux historiques admettant x comme borne supérieure.
De plus, s_2 est bien la borne supérieure de $f[H_0] \subseteq F[\mathcal{H}]$ car pour tout élément $F(H) \in F[\mathcal{H}]$, on a $F(H) \preccurlyeq_2 f(\sup H) \in f[H_0]$. Nous pouvons donc utiliser le fait que $\mathcal{R}$ est une simulation étape par étape pour transporter l'étape limite représentée par H_0, f. La garantie

$$\langle \{s_2\} \hookrightarrow \mathcal{R}[\{s_1 \mid s_1 \text{ borne supérieure de } H_0\}] \rangle$$

est donc valide. Pour conclure, il nous suffit de vérifier que c'est exactement la garantie correspondant à $\mathcal{H}, F$, ce qui découle immédiatement du fait que pour tout $x \in G_1$,

$$x \text{ majore } H_0 \Leftrightarrow x \text{ majore } \bigcup \mathcal{H}$$

En particulier, ces deux ensembles ont les mêmes bornes supérieures.

□

4.4.3 Simulation par un jeu enrichi

Dans cette sous-section, nous démontrons le théorème 4.51. Il suffit pour cela d'expliciter une traduction d'invariant et de la vérifier.

Lemme 4.58. *Soient* $\mathbb{G}_1 = (G_1, \preccurlyeq_1, \Delta_1)$, $\mathbb{G}_2 = (G_2, \preccurlyeq_2, \Delta_2)$ *deux jeux,* $\mathcal{R} \subseteq G_1 \times G_2$ *une relation induisant une simulation étape par étape de* $\mathbb{G}_1$ *par* $\mathbb{G}_2$*, et* $\mathbb{G}_p = (G_1 \times G_2, \preccurlyeq_1 \times \preccurlyeq_2, \Delta_p)$ *le* $(G_1, \preccurlyeq_1, \mathcal{R})$*-enrichissement de* $\mathbb{G}_2$*. Alors la relation associée* $\overrightarrow{\mathcal{R}}$ *induit une simulation de* $\mathbb{G}_1$ *par* $\mathbb{G}_p$*.*

De plus, cette simulation peut être explicitée comme une traduction des invariants de victoire. Choisissons un invariant de victoire $I_{P',Q'}$ *normalisé pour toute paire* P', Q' *telle que* $\langle P' \hookrightarrow Q' \rangle_{\mathbb{G}_2}$*. Alors pour toute paire* $P, Q \subseteq G_1$ *et* I_1 *un invariant de victoire normalisé pour* (P, Q)*, nous définissons pour* $(\overrightarrow{\mathcal{R}}[P], \overrightarrow{\mathcal{R}}[Q])$ *de la manière suivante. Un historique* H_p *appartient à* I_p *si et seulement si les conditions suivantes sont vérifiées :*

(1) l'historique $H_1 = \pi_1[H_p]$ *appartient à* I_1*.*

(2) pour tout élément $x \in H_1$ *qui n'est pas maximal, il existe* y *minimal tel que* $(x, y) \in H_p$ *et* $x \mathcal{R} y$*.*

(3) si H_1 *est jouable, alors* $\{y \mid (\max H_1, y) \in H_p \wedge (\max H_1) \mathcal{R} y\}$ *admet un minimum ou est vide.*

(4) si H_1 *admet un maximum* $x \notin Q$*, et s'il existe un plus petit* y *tel que* $(x, y) \in H_p$ *et* $x \mathcal{R} y$*, alors* $\{y' \mid (x, y') \in H_p \wedge y \preccurlyeq_2 y'\} \in I_{\mathcal{R}[\{x\}], \mathcal{R}[X]}$ *pour l'ensemble* $X = \{x' \neq x \mid H_1 \cup \{x'\} \in I_1\}$*.*

(5) si H_1 *admet un maximum* x*, mais que le témoin* y *défini ci-dessus n'existe pas, alors il existe* z *minimal tel que* $(x, z) \in H_p$*, et pour ce* z *on a* $\{y' \mid (x, y') \in H_p\} \in I_{\{z\}, \mathcal{R}[\{x\}]}$*. De plus, on a également* $\langle \{z\} \hookrightarrow \mathcal{R}[\{x\}] \rangle_{\mathbb{G}_2}$ *de sorte à ce que l'invariant ci-dessus soit bien défini.*

Démonstration. Comme pour la complétion, il suffit de vérifier la traduction des invariants de victoire. Le principe de la définition de I_p est le suivant : on utilise l'information supplémentaire pour mettre en corrélation l'évolution dans $\mathbb{G}_2$ avec celle dans $\mathbb{G}_1$, ce qui correspond aux conditions (1-2).

La condition (3) assure que l'une des conditions (4-5) est applicable tant que l'état gagnant n'est pas atteint, et les conditions (4-5) nous permettent de retrouver l'état local correspondant à la simulation d'une étape de l'invariant I_1. Soit l'étape en cours de simulation est régulière, soit la relation s'est désynchronisée à la limite et il s'agit alors d'une étape pour synchroniser de nouveau l'état avec la relation.

Remarquons tout d'abord que l'invariant $I_{\mathcal{R}[\{x\}], \mathcal{R}[X]}$ utilisé dans la condition (4) est défini quoi qu'il arrive. En effet, comme $x \notin Q$ et $H_1 \in I_1$, le caractère normalisé de I_1 montre que $H_1 \cap Q = \emptyset$, donc la règle de propagation (ii) s'applique. En particulier, il existe $X_0 \subseteq X$ tel que $X_0 \in \Delta_1(x)$. Comme $\mathcal{R}$ est une simulation étape par étape, $\langle \mathcal{R}[\{x\}] \hookrightarrow \mathcal{R}[X_0] \rangle_{\mathbb{G}_2}$ est vérifiée, et donc en particulier $\langle \mathcal{R}[\{x\}] \hookrightarrow \mathcal{R}[X] \rangle_{\mathbb{G}_2}$ par monotonicité (lemme 4.20).

Montrons donc que I_p vérifie les quatre règles de propagation d'un invariant de victoire.

(i) Pour tout $(x, y) \in \overrightarrow{\mathcal{R}}[P]$, nous vérifions que $\{(x, y)\} \in I_p$ en suivant la définition de I_p. Les conditions (1-3) sont immédiates, et la condition (5) ne s'applique pas car y serait le témoin associé à x. Il nous reste donc à vérifier

la condition (4), qui suit directement de la règle de propagation (i) pour les invariants de victoire.

(ii) Soit $H_p \in I_p$ un historique jouable tel que $H_p \cap \overrightarrow{\mathcal{R}}[Q] = \emptyset$, et $(x,t) = \max H$. Remarquons tout d'abord que l'une des conditions (4-5) est applicable à H_p. En effet, $H_p \cap \overrightarrow{\mathcal{R}}[Q] = \emptyset$, donc il est impossible qu'il existe y tel que $x \in Q$ et $x\mathcal{R}y$ soient vérifiés simultanément. Soit alors $H_2 \in I_{P_2,Q_2}$ l'historique et l'invariant correspondant à la condition (4-5) selon celle qui s'applique (le raisonnement est quasiment identique dans les deux cas). Il y a deux cas de figure possibles.

— Soit $t \notin Q_2$, auquel cas il s'agit principalement de simuler la propagation dans I_{P_2,Q_2}. Soit X un ensemble témoin pour la règle de propagation (ii) de I_{P_2,Q_2} pour l'historique jouable H_2, règle qui est applicable car I_{P_2,Q_2} est normalisé. Alors nous pouvons prendre $X_2 = \{(x,y) \mid y \in X\} \in \Delta_p((x,t))$ comme ensemble témoin pour la règle (ii) associée à I_p. Soit $(x,y) \in X_2$ quelconque, et vérifions que $H'_p = H_p \cup \{(x,y)\} \in I_p$.
Les conditions d'appartenance (1-3) sont directement héritées de celle de H_p. Si la condition qui s'applique pour H_p est (4), alors H'_p satisfait également la condition (4) car l'historique H_2 est remplacé par $H_2 \cup \{y\} \in I_{P_2,Q_2}$. Sinon, la condition qui s'applique pour H_p est (5), et on peut supposer que celle qui s'applique pour H'_p est (4) (c'est-à-dire que $x\mathcal{R}y$), sans quoi nous pouvons montrer que la condition est préservée de manière identique au cas précédent. Dans ce cas, H'_p satisfait bien la condition (4) via la règle de propagation (i) pour l'invariant de victoire correspondant.

— Soit $t \in Q_2$, auquel cas la condition qui s'applique pour H_p ne peut pas être (5) car cela contredit l'inexistence d'une paire liée pour x dans H_p. On est donc dans le cas de (4), avec $Q_2 = \mathcal{R}[X]$ pour X défini comme dans cette condition. Dans ce cas, soit x' un élément de X tel que $x'\mathcal{R}t$. Alors nous pouvons prendre $X_2 = \{(x',t)\} \in \Delta_p((x,t))$ comme ensemble témoin pour la règle (ii) associée à I_p. En effet, vérifions que $H'_p = H_p \cup \{(x',t)\} \in I_p$.
La première condition d'appartenance suit ici de la définition de X, car l'historique H_1 devient $H'_1 = H_1 \cup \{x'\}$ et $x' \in X$. Ensuite, la seconde est facilement étendue de H_p à H'_p car le seul nouvel élément non maximal de H'_1 est x. Celui-ci correspond alors bien à une paire liée dans H'_p car la quatrième condition s'applique pour H_p. La troisième condition est immédiatement vérifiée car l'ensemble qui doit avoir un minimum est un singleton. Enfin, il nous reste à vérifier la quatrième condition pour H'_p dans le cas où $x' \notin Q$. Comme pour la preuve de la règle (i), celle-ci suit immédiatement de la règle de propagation (i) pour les invariants de victoire de $\mathbb{G}_2$.

(iii) Soit $\mathcal{H}_p \subseteq I_p$ une chaîne non vide d'historiques de I_p, et vérifions que $\bigcup \mathcal{H}_p$ appartient à I_p. La condition (1) suit immédiatement de la règle de propagation (iii) pour l'invariant I_1, et la condition (2) vient du fait que tout élément non maximal de $H_1 = \pi_1[\bigcup \mathcal{H}_p]$ est également non maximal pour l'un des historiques de la chaîne $\mathcal{H}_p$. Si H_1 est limite, il n'y a en fait pas d'autres conditions à vérifier et $\bigcup \mathcal{H}_p \in I_p$. Supposons donc que H_1 est jouable. Dans ce cas son maximum x est atteint par au moins un historique de $H \in \bigcup \mathcal{H}_p$.

En particulier, H satisfait la condition (3), et le minimum pour l'ensemble associé est également minimum pour l'ensemble associé à $\bigcup \mathcal{H}_p$ par définition de l'ordre préfixe, donc $\bigcup \mathcal{H}_p$ vérifie également la condition (3).
Il reste à vérifier les conditions (4-5), si l'une d'entre elles s'applique. Pour cela, remarquons tout d'abord que l'on peut remplacer $\mathcal{H}_p$ par la chaîne alternative $\{H \in \mathcal{H}_p \mid x \in \pi_1[H]\}$, de manière à pouvoir supposer que l'historique des premières composantes est H_1 pour tout historique de $\mathcal{H}_p$. En effet, cela ne change pas la borne supérieure car les éléments enlevés sont préfixes des éléments conservés, et au moins un élément est conservé. Une fois ce remplacement effectué, nous avons trois possibilités.

— Soit il existe un historique $H \in \mathcal{H}_p$ pour lequel aucune des conditions (4-5) ne s'applique. Dans ce cas, il existe y tel que $(x, y) \in H$, $x \mathcal{R} y$ et $x \in Q$. En particulier, la situation est la même pour $\bigcup \mathcal{H}_p$ car l'historique des premières composantes est identique, et ces deux conditions ne s'appliquent pas.
— Soit la condition (4) s'applique pour au moins un historique $H \in \mathcal{H}_p$. Comme précédemment, il existe y tel que $(x, y) \in H$, $x \mathcal{R} y$ et $x \notin Q$. En particulier, la condition qui s'applique est la même pour n'importe quel historique de $\mathcal{H}_p$ plus grand que H. On peut donc se réduire au cas où tous les historiques de $\mathcal{H}_p$ sont dans cette situation, et on vérifie que $\bigcup \mathcal{H}_p$ correspond également à la condition (4). Celle-ci suit alors immédiatement de la règle de propagation (iii) pour l'invariant associé à cette condition.
— Soit la condition (5) s'applique à tous les historiques de $\mathcal{H}_p$. Comme précédemment, la condition qui s'applique pour $\bigcup \mathcal{H}_p$ est (5), et est vérifiée via la règle de propagation (iii) pour l'invariant associé.

(iv) Soit $H_p \in I_p$ un historique admettant une borne supérieure, et vérifions que $H'_p = H_p \cup \{\sup H\} \in I_p$. Pour cela, soit $H_1 = \pi_1[H_p]$ l'historique des premières composantes de H_p. La condition d'appartenance (1) pour H'_p suit directement de la règle de propagation (iv) pour I_1. Ensuite, remarquons que tout élément non maximal dans $\pi_1[H'_p]$ est également non maximal dans H_1, d'où la condition (2). La condition (3) est directement héritée de H_p. Il nous reste donc à vérifier les deux dernières conditions quand elles sont applicables.
Supposons dans un premier temps que H_1 est jouable, et donc que l'historique des premières composantes de H'_p est H_1. Si aucune des conditions (4-5) ne s'applique pour H_p, alors c'est également le cas pour H'_p. Dans le cas contraire, nous pouvons faire le même raisonnement que pour la règle de propagation (ii) en remplaçant l'utilisation de la règle (ii) sur les invariants de $\mathbb{G}_2$ par celle de la règle (iv).
Nous pouvons donc supposer que nous sommes dans le cas alternatif où H_1 est limite. Dans ce cas, nous allons appliquer la condition limite d'une simulation étape par étape. Posons pour cela $f(x) = \min\{y \mid (x, y) \in H_p \wedge x \mathcal{R} y\}$, qui d'après la condition (2) est bien défini pour $x \in H_1$. Alors $\sup \pi_2[H_p] = \sup f(H_1)$ car pour tout élément de $y \in \pi_2[H_p]$, il existe un élément de $f(H_1)$ plus grand que y. En effet, posons x tel que $(x, y) \in H_p$. Comme H_1 est limite, il contient un élément x' plus grand que x. En particulier, $y \preccurlyeq_2 f(x')$. car (x, y) et $(x', f(x'))$ sont comparables.
Soit $(s_1, s_2) = \sup H_p = (\sup H_1, \sup f(H_1))$. Nous pouvons appliquer la condition limite d'une simulation étape par étape à $\mathcal{X}$, et en déduire la ga-

rantie $\langle \{s_2\} \hookrightarrow \mathcal{R}[\{s_1\}] \rangle_{\mathbb{G}_2}$. En particulier, si $\neg(s_1 \mathcal{R} s_2)$, la condition (5) s'applique pour H_p, et est vérifiée via la garantie ci-dessus et l'application de la règle de propagation (i) de l'invariant associée. Sinon, la seule condition qui peut s'appliquer pour H_p est (4), auquel cas elle est immédiatement vérifiée par la règle de propagation (i) pour l'invariant correspondant à cette condition.

□

4.4.4 Restriction aux invariants réductibles

Dans cette sous-section, nous montrons le théorème 4.54, qui énonce que tout invariant de victoire d'un enrichissement peut être pris réductible. Pour cela, nous montrons que l'ensemble des historiques fortement u-accessibles d'une stratégie gagnante quelconque convient.

Lemme 4.59. *Pour tout jeu $\mathbb{G}$, et tout $\mathbb{G}_p$ un $(O, \preccurlyeq_O, \mathcal{R})$-enrichissement de $\mathbb{G}$, si $\langle P \hookrightarrow Q \rangle_{\mathbb{G}_p}$ est vérifiée pour deux ensembles P et Q, alors P, Q admet un invariant de victoire réductible.*

Démonstration. Soit $\sigma_\exists$ une stratégie gagnante pour P, Q. D'après le théorème 4.33, l'ensemble des historiques qui sont fortement u-accessibles pour $\sigma_\exists$ et pour un point de départ de P est un invariant de victoire I. Il nous suffit de montrer que I est de plus réductible pour conclure. Remarquons tout d'abord que l'ensemble C défini par :

$$C = \{H \in I \mid \forall H' \sqsubseteq H.\ H' \in I\}$$

est stable par les règles de propagation de l'u-accessibilité forte, ce que l'on peut montrer directement en déroulant les définitions. En particulier, $C = I$, et I est clos par historique préfixe. Il nous reste donc à vérifier la condition d'existence des extensions maximales pour une projection donnée.

Pour cela, soit $H_0 \in I$, et x un point de départ pour lequel H_0 est fortement u-accessible. Considérons maintenant l'ensemble d'historiques $A \subseteq I$ tel qu'un historique $H_1 \in I$ appartienne à A si et seulement si la condition suivante est vérifiée :

$$H_0 \sqsubseteq H_1 \Rightarrow \forall H_2 \in I, \pi_2[H_0] = \pi_2[H_2] \wedge H_0 \sqsubseteq H_2 \Rightarrow H_1 \sqsubseteq H_2 \vee H_2 \sqsubseteq H_1$$

Remarquons qu'il suffit de démontrer que $I \subseteq A$ pour conclure. En effet, cette propriété énonce que les ensembles dont on veut trouver un maximum sont des chaînes de I. Ces chaînes d'historiques admettent alors chacune une borne supérieure, et l'on montre immédiatement que ces bornes supérieures sont en fait des maximums.

Notons que tout historique H_1 fortement u-accessible pour $\sigma_\exists$ et un point de départ $y \in P \backslash \{x\}$ appartient à A, car H_1 et H_0 ne sont pas comparables. En effet, comme H_1 est fortement u-accessible, il est également u-accessible pour le même point de départ, et appartient donc à une partie partant de y. En particulier, $\min H_1 = \{y\}$ et de même $\min H_0 = \{x\}$.

Il suffit donc de vérifier que tous les historiques fortement u-accessibles pour $\sigma_\exists$ et le point de départ x appartiennent à A. Pour cela, nous vérifions que A satisfait les règles de propagation pour les historiques fortement u-accessibles, l'inclusion découlant alors de la minimalité de I.

(i) $\{x\} \in A$ car $\{x\}$ est préfixe de tout historique fortement u-accessible depuis x.

(iii) Soit $\mathcal{H}$ une chaîne d'historiques de A. Montrons que $\bigcup \mathcal{H} \in A$. L'appartenance à I suivant des règles de propagation de I, supposons donc que $H_0 \sqsubseteq \bigcup \mathcal{H}$, et montrons que pour $H_2 \in I$ tel que $\pi_2[H_0] = \pi_2[H_2]$ et $H_0 \sqsubseteq H_2$, $\bigcup \mathcal{H}$ et H_2 sont comparables. Nous distinguons deux cas. Soit H_2 est plus grand que tout historique de $\bigcup \mathcal{H}$, et H_2 est alors plus grand que $\bigcup \mathcal{H}$. Dans le cas inverse, il existe un historique de $\mathcal{H}$ que H_2 ne domine pas. De plus, comme $\bigcup \mathcal{H}$ est plus grand que H_0, il contient un historique plus grand que H_0. Comme $\mathcal{H}$ est une chaîne, on peut trouver un témoin simultané H_3 de ces deux propriétés en prenant le maximum des deux témoins. Mais par appartenance à A, H_3 et H_2 sont comparables. En particulier, $H_2 \sqsubseteq H_3 \sqsubseteq \bigcup \mathcal{H}$.

(iv) Soit $H \in A$ admettant une borne supérieure. Montrons que l'historique $H_1 = H \cup \{\sup H\}$ appartient à A. Pour cela, on peut supposer que $H \neq H_1$, c'est-à-dire que $\sup H \notin H$. Supposons donc que $H_0 \sqsubseteq H_1$. Considérons l'ensemble B défini par

$$\{H_2 \in I \mid \pi_2[H_2] = \pi_2[H_0] \wedge H_0 \sqsubseteq H_2 \Rightarrow H_1 \sqsubseteq H_2 \vee H_2 \sqsubseteq H_1\}$$

Le résultat désiré revient exactement à montrer que $B \subseteq I$. En reprenant la même structure de raisonnement inductif, il suffit de montrer que B est stable par les règles de propagation des historiques fortement u-accessibles partant de x. Le raisonnement pour les règles (i) et (iii) étant quasi identique à ce qui est fait ci-dessus, nous nous intéressons uniquement aux deux règles restantes.

(ii) Soit $H_2 \in B$ un historique jouable tel que $\max H_2 \notin \sigma_\exists(H_2)$, $y \in \sigma_\exists(H_2)$. Pour montrer que $H_3 = H_2 \cup \{y\} \in B$, supposons donc que $\pi_2[H_3] = \pi_2[H_0]$ et $H_0 \sqsubseteq H_3$. Remarquons tout d'abord que si $\pi_2[H_2] \neq \pi_2[H_0]$ ou si H_2 n'est pas plus grand que H_0, alors la seule possibilité est que $H_3 = H_0 \sqsubseteq H_1$, ce qui conclut. Supposons donc que H_2 vérifie ces deux hypothèses comme H_3. Les historiques H_1 et H_2 sont alors comparables par définition de B, et on peut supposer $H_2 \sqsubseteq H_1, H_2 \neq H_1$ car le cas inverse permet de conclure immédiatement. Mais comme $H_2 \neq H_1$, on en déduit que $H_2 \sqsubseteq H$, et comme $\sup H \notin H$, $H_2 \neq H$. En particulier :

$$\pi_2[H_0] = \pi_2[H_2] \subseteq \pi_2[H] \subseteq \pi_2[H_1] = \pi_2[H_0]$$

et de même $H_0 \sqsubseteq H$ donc comme $H \in A$, H et H_3 sont comparables. La seule possibilité compatible avec les contraintes d'ordre préfixe est alors que $H_3 \sqsubseteq H \sqsubseteq H_1$.

(iv) Le raisonnement est rigoureusement identique à celui utilisé pour la règle (ii) en remplaçant y par $\sup H_2$, à l'exception du fait que le cas $H_2 = H$ est maintenant possible. Mais dans ce cas, on a $H_1 = H_3$ donc les deux historiques sont bien comparables.

(ii) Encore une fois, le raisonnement est identique à celui de la règle (iv) en intervertissant le rôle des transitions et des bornes supérieures, à l'exception de l'égalité $H_2 = H$ qui intervient maintenant dans le sous-cas de la règle (ii). Nous nous concentrons donc uniquement sur ce cas. Soient donc $y, z \in \sigma_\exists(H) \in \Delta(\max H)$, $H_1 = H \cup \{y\}$, et $H_3 = H_2 \cup \{z\}$ tels que H_1, H_2, et

H_3 ont tous la même projection que H_0 et sont tous plus grands que H_0 (ce sont des hypothèses obtenues par le raisonnement ci-dessus). On cherche à montrer que H_1 et H_3 sont comparables, ce qui revient à dire que $y = z$. Il suffit pour cela de vérifier que $\sigma_\exists(H)$ ne contient pas plus d'un élément avec la même seconde composante que $\max H$, auquel cas y et z sont égaux à cet élément. Dans le cas où $\sigma_\exists(H)$ est une transition dérivée du jeu d'origine, la première composante est fixée, et donc le seul candidat possible serait $\max H$, ce qui est impossible par hypothèse. Pour les transitions ajoutées par enrichissement, c'est immédiat car les transitions sont des singletons.

□

Remarque 4.60. Nous pouvons partiellement expliciter cette étape par une traduction d'invariants, en ajoutant une étape de conversion d'un invariant arbitraire vers une stratégie, puis en convertissant en invariant de nouveau via les constructions du théorème 4.33. Si on analyse ces constructions en détail, on voit que la traduction est entièrement déterminée par une fonction de choix, laquelle choisit une transition parmi celles possibles pour les historiques jouables de l'invariant. Nous pouvons de plus réduire le domaine de cette fonction de choix en faisant un détour par une extension de $\mathbb{G}_p = (G_p, \preccurlyeq_p, \Delta_p)$, plus précisément le jeu $\mathbb{G}'_p = (G_p, \preccurlyeq_p, \Delta'_p)$ où

$$\Delta'_p(x) = \{X \mid \exists Y.Y \subseteq X \wedge Y \in \Delta_p(x) \wedge \forall z \in X.x \preccurlyeq z\}$$

Il est immédiat que les deux jeux ont les mêmes invariants de victoire. De plus, nous pouvons partiellement caractériser la fonction de choix en prenant l'union des transitions héritées du jeu $\mathbb{G}$ et d'au plus une transition ajoutée par enrichissement. Cela réduit donc la partie non explicite de la traduction à une fonction de choix sur les transitions ajoutées par enrichissement, c'est-à-dire sur les sous-ensembles d'ensembles de la famille

$$(\{x \mid x\mathcal{R}y \wedge z \preccurlyeq x\})_{y\in O, z\in G}$$

En particulier, remarquons que si $\mathcal{R}$ est une relation inversement fonctionnelle, la traduction vers des invariants de victoire réductibles devient complètement explicite, et correspond simplement à la restriction aux historiques accessibles par les règles de propagation (i)-(iv) définissant les invariants de victoire.

4.4.5 Élimination de l'enrichissement

Dans cette sous-section, nous démontrons le théorème 4.55. Nous fixons donc $(O, \preccurlyeq_O)$ un ordre chaîne-complet, $\mathbb{G} = (G, \preccurlyeq, \Delta)$ un jeu, $\mathcal{R} \subseteq O \times G$ une relation et $\mathbb{G}_p = (G_p, \preccurlyeq_p, \Delta_p)$ le $(O, \preccurlyeq_O, \mathcal{R})$-enrichissement de $\mathbb{G}$. Nous montrons la simulation via une traduction des invariants de victoire réductibles. Soit donc I_p un invariant de victoire réductible pour P, Q. Nous allons définir un invariant de victoire pour $\pi_2[P], \pi_2[Q]$ en nous ramenant à l'unique historique de I_p correspondant. Pour garantir cette unicité, nous supposons également fixée une fonction de choix f_P qui à tout $y \in \pi_2[P]$ associe un élément $f_P(y) \in P$ tel que $f_P(y)\pi_2 y$.

Définition 4.61. Nous définissons I un ensemble d'historiques de $\mathbb{G}$ par $H \in I$ si et seulement s'il existe un unique historique $H_p \in I_p$ satisfaisant les conditions suivantes :

(1) $\pi_2[H_p] = H$ et $\min H_p = f_p(\min H)$.
(2) pour tout historique $H_1 \sqsubseteq H_p$ et $x \in G_p$, si $H_1 \cup \{x\} \sqsubseteq H_p$ et $\pi_2[H_1] \neq \pi_2[H_1 \cup \{x\}]$ alors H_1 admet une borne supérieure, et $x, \sup H_1$ ont la même première composante.
(3) pour tout historique jouable $H_0 \sqsubseteq H$, il existe un historique minimal $H_1 \sqsubseteq H_p$ tel que $\pi_2[H_1] = H_0$, et pour cet historique on a

$$\max \{H_2 \in I_p \mid \pi_2[H_2] = \pi_2[H_1] \wedge H_1 \sqsubseteq H_2\} \sqsubseteq H_p$$

Pour montrer que I est bien un invariant de victoire pour P, Q, le principe est de copier les règles de propagation sur les témoins H_p, puis de les étendre de manière maximale en utilisant la réductibilité. Nous montrons tout d'abord des conditions suffisantes pour que l'invariant I soit valide, qui sont des conditions sous lesquelles ce processus d'extension maximale est possible. Comme nous utilisons systématiquement ces conditions pour montrer qu'un historique appartient à I, elles constituent en fait un invariant de victoire alternatif pour P, Q.

Lemme 4.62. *Soit H un historique tel qu'il existe un historique $H_p \in I_p$ minimal satisfaisant les mêmes conditions (1-2) que dans la définition 4.61, ainsi que la restriction suivante de la condition (3) :*

(3') pour tout historique jouable $H_0 \sqsubseteq H$, il existe un historique minimal $H_1 \sqsubseteq H_p$ tel que $\pi_2[H_1] = H_0$, et si $H_0 \neq H$, alors

$$\max \{H_2 \in I_p \mid \pi_2[H_2] = \pi_2[H_1] \wedge H_1 \sqsubseteq H_2\} \sqsubseteq H_p.$$

Alors $H \in I$.

Démonstration. Montrons d'abord le cas où H est limite. Dans ce cas, H_p satisfait sans restriction les conditions (1-3), et il suffit de vérifier l'unicité de H_p. Supposons par l'absurde qu'il existe un témoin $H'_p \neq H_p$. par minimalité de H_p, $H_p \sqsubseteq H'_p$. Soit alors $(x, y) \in H'_p \backslash H_p$. Par la condition (1) associée à H'_p, nous avons $y \in H$. En particulier, comme H est limite il existe $y' \in H$ strictement plus grand que y. Par la condition (1) associée à H_p, il existe x' tel que $(x', y') \in H_p$, ce qui est en contradiction avec $H_p \sqsubseteq H'_p$.

Vérifions maintenant le cas où H est jouable. Posons alors

$$\overline{H}_p = \max\{H_2 \in I_p \mid \pi_2[H_2] = \pi_2[H_p] \wedge H_p \sqsubseteq H_2\}$$

qui existe car I_p est réductible. Par définition, il est immédiat que $\overline{H}_p$ satisfait les conditions (1-2). La condition (3) est immédiate pour les préfixes stricts de H, et le cas de H découle du fait que pour tout $H_1 \sqsubseteq H_p$ tel que $\pi_2[H_1] = H$, on a $H_1 = H_p$. En effet, un tel historique H_1 satisfait alors les conditions (1-2) et (3'), donc par minimalité de H_p, $H_p \sqsubseteq H_1$. En particulier, H_p est le plus petit préfixe de $\overline{H}_p$ tel que $\pi_2[H_p] = H$, ce qui valide la condition (3). Il nous suffit donc de vérifier l'unicité de $\overline{H}_p$.

Soit $H'_p \in I_p$ un témoin quelconque des conditions (1-3) associées à H. Par minimalité de H_p on a $H_p \sqsubseteq H'_p$. Mais comme $\overline{H}_p, H'_p$ satisfont tous les deux la condition (3), l'application respective de ces conditions à H montre que les historiques $\overline{H}_p$ et H'_p sont préfixes l'un de l'autre, et par conséquence sont égaux. □

Une autre propriété cruciale dont nous faisons usage pour montrer le théorème 4.55 est que les témoins correspondant aux historiques respectent les mêmes conditions d'ordre.

Lemme 4.63. *Soit $H_1 \in I$ un historique, $H_{p,1}$ l'unique témoin correspondant. Alors pour tout historique H_2 dont H_1 est préfixe strict, et $H_{p,2} \in I_p$ n'importe quel témoin des conditions (1-3') associées à H_2, $H_{p,1} \sqsubseteq H_{p,2}$.*

Démonstration. Posons $H'_{p,1} = \{(x, y) \in H_{p,2} \mid y \in H_1\}$. Par définition, il est immédiat que $H_{p,1}$ est un témoin des conditions (1-3), et c'est un préfixe de $H_{p,2}$. Par réductibilité de I_p, $H'_{p,1} \in I_p$, donc par unicité de $H_{p,1}$ on a $H_{p,1} = H'_{p,1} \sqsubseteq H_{p,2}$. □

Armés de ces deux lemmes, nous pouvons maintenant vérifier que la traduction des invariants de victoire proposée est correcte.

Lemme 4.64. *I est un invariant de victoire pour $\pi_2[P], \pi_2[Q]$.*

Démonstration. Nous montrons que I vérifie bien les quatre règles de propagation des invariants de victoire.

(i) Pour $x \in \pi_2[P]$, $\{x\} \in I$ car $\{f_P(x)\} \in I_p$ est un témoin minimal des conditions (1-2) et de (3'). Le fait qu'il soit un témoin est direct par définition des conditions, et il est minimal à cause de la condition (1). Le lemme 4.62 permet de conclure.

(iii) Soit $\mathcal{H} \subseteq I$ une chaîne non vide d'historiques. Par définition de I, il existe une unique famille $(T_{p,H})_{H \in \mathcal{H}}$ de témoins pour les conditions (1-3) associées aux historiques respectifs. Le lemme 4.63 montre immédiatement qu'elle est monotone, et donc que l'ensemble $\mathcal{H}_p = \{T_{p,H} \mid H \in \mathcal{H}\}$ forme une chaîne d'historiques de I_p. Par la règle de propagation (iii) pour I_p, $\bigcup \mathcal{H}_p \in I_p$. De plus, il suffit de dérouler les définitions des conditions pour vérifier que $\bigcup \mathcal{H}_p$ est un témoin des conditions (1-3') associées à $\bigcup \mathcal{H}$. Pour montrer que $\bigcup \mathcal{H} \in I$, le lemme 4.62 montre qu'il nous suffit de garantir que c'est un témoin minimal. Soit donc H'_p un autre témoin. Le lemme 4.63 donne directement que pour tout $H \in \mathcal{H}$, $T_{p,H} \sqsubseteq H'_p$. En passant cette inégalité à la borne supérieure, on obtient la minimalité de $\bigcup \mathcal{H}_p$.

(ii) Soit $H \in I$ un historique jouable tel que $H \cap \pi_2[Q] = \emptyset$, et $H_p \in I_p$ l'unique témoin des conditions (1-3) associées à H. Remarquons tout d'abord que H_p est bien jouable. En effet, soit

$$(x, y) = (\sup \{z \mid (z, \max H) \in H_p\}, \max H)$$

la borne supérieure de H_p, bien définie car $(O, \preccurlyeq_O)$ est chaîne-complet. En particulier, $H_p \cup \{(x, y)\} \in I_p$, et il est immédiat que cet historique satisfait les conditions (1-3). Donc $H_p \cup \{(x, y)\} = H_p$ par unicité, et H_p admet comme maximum (x, y).

Comme on vérifie également que $H_p \cap Q = \emptyset$, la règle de propagation (ii) pour I_p montre qu'il existe un ensemble $X \in \Delta_p((x, y))$ tel que $(x, y) \notin X$ et pour tout $(x', y') \in X$, $H_p \cup \{x', y'\} \in I_p$. De plus, X ne peut pas être un singleton $\{x', y\}$ ajouté par l'enrichissement car sinon $H_p \cup \{x', y\}$ serait un témoin pour les conditions (1-3) associées à H. En particulier, l'unicité de H_p entraînerait que $x = x'$, ce qui est contradictoire. Donc il existe $Y \in \Delta(y)$ tel

que $X = \{(x, y') \mid y' \in Y\}$. Comme $(x, y) \notin X$, on en déduit immédiatement que $y \notin Y$. Nous allons montrer que Y est le témoin voulu pour la règle de propagation (ii) de I appliqué à H.
Soit donc $y' \in Y$ quelconque, et montrons que $H' = H \cup \{y'\} \in I$. Pour cela, le lemme 4.62 montre qu'il suffit de trouver un témoin minimal des conditions (1-3') associées, et un candidat naturel est $H'_p = H_p \cup \{(x, y')\}$. C'est bien un élément de I_p grâce aux propriétés de X, et il satisfait immédiatement les conditions (1) et (3'). Nous montrons que H'_p satisfait également la condition (2). En effet, soient $H_1 \sqsubseteq H'_p, z \in G_p$ tels que $H_1 \cup \{z\} \sqsubseteq H'_p$ et $\pi_2[H_1] \neq \pi_2[H_1 \cup \{z\}]$. On peut également supposer que $H_1 \cup \{z\} \not\sqsubseteq H_p$ car sinon la propriété découle immédiatement du fait que H_p vérifie la condition (2). Ce cas de figure n'est possible que si $z = (x, y')$, et donc $H_1 = H_p$. Comme dans ce cas $\sup H_1 = \max H_1 = (x, y)$, $\sup H_1$ et z ont bien la même première composante.
Il nous reste à montrer la minimalité de H'_p. Soit donc $H''_p \in I_p$ un autre témoin des conditions (1-3'). Comme H''_p satisfait la condition (3') associée à H', il existe $H_1 \sqsubseteq H''_p$ minimal tel que $\pi_2[H_1] = H'$. En particulier, il existe un unique x' tel que $(x', y') \in H_1$. L'existence suit de $\pi_2[H_1] = H'$, et l'unicité d'un raisonnement par contradiction. En effet, supposons qu'il existe un tel autre x''. Comme H_1 est une chaîne, x' et x'' sont comparables, et on peut supposer $x' \preccurlyeq_1 x''$ par symétrie. Alors $\pi_2[\{z \in H_1 \mid z \preccurlyeq_p (x', y')\}] = H'$, ce qui contredit la minimalité de H_1.
Considérons maintenant l'historique $H_2 = H_1 \backslash \{(x', y')\}$. Il découle des propriétés de x' que $(x', y') = \max H_1$ et que $\pi_2[H_2] = H$. Or, il est direct que H_2 satisfait les conditions (1-3), donc par unicité $H_p = H_2$. En particulier, $H_p \cup \{(x', y')\} = H_1 \sqsubseteq H''_p$. Comme H''_p satisfait la condition (2), on en déduit que $x' = x$, et donc $H'_p = H_1 \sqsubseteq H''_p$, d'où la minimalité.

(iv) Soit $H \in I$ un historique admettant une borne supérieure $y = \sup H$, et montrons que $H' = H \cup \{y\} \in I$. On peut naturellement supposer que $y \notin H$. Soit donc $H_p \in I_p$ l'unique témoin des conditions (1-3) associées à H. Remarquons tout d'abord que la paire

$$(x, y) = (\sup \{z \mid (z, t) \in H_p\}, y)$$

est bien définie car $(O, \preccurlyeq_O)$ est chaîne-complet, et est la borne supérieure de H_p. En particulier, l'historique $H'_p = H_p \cup \{(x, y)\}$ appartient à I_p, et c'est un candidat naturel pour appliquer le lemme 4.62. Il suffit de dérouler les définitions pour vérifier que H'_p satisfait les conditions (1) et (3') associées à H'. Vérifions maintenant qu'il satisfait bien la condition (2). Soient $H_1 \sqsubseteq H'_p, z \in G_p$ tels que $H_1 \cup \{z\} \sqsubseteq H'_p$ et $\pi_2[H_1] \neq \pi_2[H_1 \cup \{z\}]$. Comme pour le cas de la règle (ii), on peut supposer que $H_1 \cup \{z\} \not\sqsubseteq H_p$ et donc $z = (x, y)$, et $H_1 = H_p$. Comme $\sup H_p = (x, y)$, l'égalité des premières composantes est immédiate. Un raisonnement identique à celui effectué pour la règle (ii) permet de montrer la minimalité de H'_p, et donc $H' \in I$.

□

Remarque 4.65. La condition de chaîne-complétude n'est pas strictement nécessaire pour montrer le théorème 4.55. Il suffit en effet que pour tout historique d'éléments liés entre eux, l'existence d'une borne supérieure dans le support du jeu cible entraîne

l'existence d'une borne supérieure dans le support du jeu d'origine. Autrement dit, il suffit que pour tout historique H du jeu d'origine, et pour toute fonction croissante f de $(H, \preccurlyeq_1)$ dans $(G_2, \preccurlyeq_2)$, si pour tout $x \in H$ on a $x \mathcal{R} f(x)$, alors si $f(H)$ admet une borne supérieure, H en admet également une. Pour certaines relations, notamment la relation d'égalité, nous pouvons alors expliciter une traduction qui ne passe pas par la complétion de l'ordre car les bornes supérieures sont transportées correctement.

La preuve du théorème 4.55 reste alors valide si l'invariant sur le jeu enrichi est un sous-ensemble de celui produit par le lemme 4.58. Plus précisément, il suffit que les historiques appartenant à l'invariant satisfassent la seconde condition définie dans le lemme 4.58, c'est-à-dire qu'il y a suffisamment d'états appariés par la relation. Nous pouvons également renforcer le lemme 4.59 pour montrer que l'invariant réductible obtenu hérite de toute façon cette propriété de la structure du jeu.

4.5 Codage des systèmes de transition en jeux

Dans cette section, nous faisons le lien entre les jeux et la sémantique opérationnelle à petits pas d'un langage. Nous montrons comment convertir la sémantique opérationnelle en jeux, et comment déduire des résultats sur un programme depuis des résultats sur les jeux associés. Dans un premier temps, nous donnons une traduction vers des jeux basés sur les traces. Cette conversion permet d'obtenir facilement des résultats d'équivalence. Ensuite, nous utilisons le théorème de simulation 4.41 pour donner une traduction équivalente mais plus locale, et donc plus pratique à utiliser.

4.5.1 Systèmes de transition

De manière générale, une sémantique à petits pas correspond à la notion abstraite de système de transition. Un exemple typique d'état est alors la paire du code du programme et d'un état mémoire.

Définition 4.66 (Système de transition)**.** Un *système de transition* est un couple $(S, \to)$ où S est un ensemble d'états et $\to \subseteq S \times S$ est une relation de transition.

Nous pouvons alors décrire un comportement du système par une séquence d'états, potentiellement infinie. Pour les comportement infinis, il s'agit en fait d'une représentation identique à celle des modèles de la logique temporelle linéaire [72]. C'est également une représentation adaptée à la conversion en jeux, car quasiment identique aux historiques. En effet, l'ordre préfixe sur les séquences fournit naturellement un ordre compatible avec l'évolution du système. Ce n'est pas une surprise car cet ordre est essentiellement l'ordre préfixe pour les historiques.

Définition 4.67 (Ensemble des traces d'évolution)**.** L'*ensemble des traces d'évolution* d'un système de transition $\mathcal{S} = (S, \to)$ est l'ensemble $T(\mathcal{S})$ des séquences finies ou infinies de transition dans le système, défini par :

$$T(\mathcal{S}) = \{(u_n)_{n \in I} \in S^+ \cup S^\omega \mid \forall n \in I, n > 0 \Rightarrow (u_{n-1} \to u_n)\}$$

où nous utilisons I pour dénoter l'ensemble des indices de la séquence. Dans le cas des séquences qui sont des traces d'évolution, cet ensemble est un intervalle entier potentiellement infini dont le plus petit élément est 0.

Définition 4.68 (Ordre préfixe et extension)**.** Une séquence $(a_n)_{n\in I_a}$ est *préfixe* d'une séquence $(b_n)_{n\in I_b}$, ce que nous notons $(a_n)_{n\in I_a} \leq_p (b_n)_{n\in I_b}$, si $I_a \subseteq I_b$ et les deux séquences coïncident sur I_a. Alternativement, nous pouvons dire que $(b_n)_{n\in I_b}$ est une *extension* de $(a_n)_{n\in I_a}$.

4.5.2 Traduction basée sur les traces

Nous utilisons l'ensemble des traces d'évolution des états d'un système de transition pour définir deux conversions vers les jeux. La première est une traduction existentielle, qui permet de convertir les garanties dans le jeu associé en résultats d'accessibilité dans le système de transition. La seconde est une traduction universelle, qui de manière duale permet de convertir les garanties dans le jeu associé en propriétés de vivacité dans le système de transition, c'est-à-dire qu'un élément d'un ensemble d'états donné sera forcément atteint pour n'importe quelle évolution.

Définition 4.69 (Jeu existentiel des traces)**.** Soit $\mathcal{S} = (S, \to)$ un système de transition. Le *jeu existentiel des traces* associé au système est le jeu $\mathbb{G}^t_{\mathcal{S},\exists}$ dont le support est $(T(\mathcal{S}), \leq_p)$ et dont l'ensemble de transitions $\Delta^t_{\mathcal{S},\exists}$ est défini par :

— $\Delta^t_{\mathcal{S},\exists}((s_n)_{n\in\mathbb{N}}) = \emptyset$ pour une séquence infinie,

— $\Delta^t_{\mathcal{S},\exists}((s_n)_{0\leq n\leq m}) = \{\{(s_0 \dots s_m s')\} \mid s_m \to s'\}$ pour une séquence finie.

Lemme 4.70. *Soit $\mathcal{S} = (S, \to)$ un système de transition, $(u_n)_{n\in I} \in T(\mathcal{S})$ une trace d'évolution de $\mathcal{S}$, et $Q \subseteq T(\mathcal{S})$ un ensemble de traces d'évolution. Alors la garantie $\langle\{(u_n)_{n\in I}\} \hookrightarrow Q\rangle_{\mathbb{G}^t_{\mathcal{S},\exists}}$ est vérifiée si et seulement s'il existe une trace d'évolution dans Q dont $(u_n)_{n\in I}$ soit préfixe.*

Démonstration. Le sens réciproque est immédiat : si il existe une trace d'évolution $(v_n)_{n\in J} \in Q$ dont $(u_n)_{n\in I}$ est préfixe, alors la stratégie consistant à systématiquement choisir une transition cohérente avec $(v_n)_{n\in J}$ est gagnante pour l'ensemble cible Q. De manière plus directe, l'ensemble des historiques de $\mathbb{G}^t_{\mathcal{S},\exists}$ ne contenant que des traces d'évolution préfixes de $(v_n)_{n\in J}$ est un invariant de victoire pour les ensembles $\{(u_n)_{n\in I}\}$ et Q. La vérification des quatre conditions définissant un invariant de victoire est immédiate.

Pour le sens direct, soit $\sigma_\exists$ une stratégie gagnante pour cette garantie, et $\sigma_\forall$ la stratégie universelle qui consiste à choisir l'unique élément d'un singleton. Comme $\sigma_\exists$ est gagnante, il existe un historique gagnant H pour la partie associée. Il s'agit d'une victoire effective, car $\sigma_\forall$ a été choisi de façon à ne jamais pouvoir être en défaut, et toute chaîne non vide de $(T(\mathcal{S}), \leq_p)$ (donc H) admet une borne supérieure. Or, on a $\min H = (u_n)_{n\in I}$ et $\max H \in Q$, donc Q contient bien une trace d'évolution dont $(u_n)_{n\in I}$ est préfixe. □

Le fait que nous exprimions le lemme 4.70 vis-à-vis d'un ensemble de traces plutôt que par un ensemble d'états peut paraître contre-intuitif, mais c'est un passage obligé pour exprimer l'accessibilité de comportements infinis. De plus, cet énoncé permet également de traiter la notion usuelle d'accessibilité. Par exemple, nous pouvons exprimer l'accessibilité d'un état y depuis un autre état x en montrant la propriété $\left\langle\{(x)_{n\in[0;0]}\} \hookrightarrow \{(u_n)_{n\in[0;m]} \in T(\mathcal{S}) \mid u_m = y\}\right\rangle$ dans le jeu existentiel des traces correspondant.

Définition 4.71 (jeu universel des traces)**.** Soit $\mathcal{S} = (S, \to)$ un système de transition. Le *jeu universel des traces* associé au système est le jeu $\mathbb{G}^t_{\mathcal{S},\forall}$ dont le support est $(T(\mathcal{S}), \leq_p)$ et dont l'ensemble de transitions $\Delta^t_{\mathcal{S},\forall}$ est défini par :

— $\Delta^t_{\mathcal{S},\forall}((s_n)_{n\in\mathbb{N}}) = \emptyset$ pour une séquence infinie,

— $\Delta^t_{\mathcal{S},\forall}((s_n)_{0\leq n\leq m}) = \{\{(s_0 \dots s_m s') \mid s_m \to s'\}\}\backslash\{\emptyset\}$ pour une séquence finie.

Nous enlevons l'ensemble vide des transitions possibles pour garantir que le joueur existentiel perde lorsque l'on atteint un état sans successeur dans le système de transition $\mathcal{S}$. Sans cette contrainte, le joueur existentiel aurait la possibilité de fournir un ensemble sans choix valide (car vide) au joueur universel, et donc de gagner de manière immédiate. Le cas d'un ensemble vide de successeurs correspond intuitivement à la notion d'état bloquant dans une sémantique à petits pas, ce qui justifie que nous souhaitions le comportement inverse.

Comme pour le jeu existentiel, nous pouvons convertir les garanties dans le jeu universel en énoncés sur les traces d'évolution du système de transition.

Lemme 4.72. *Soit $\mathcal{S} = (S, \to)$ un système de transition, $(u_n)_{n\in I} \in T(\mathcal{S})$ une trace d'évolution de $\mathcal{S}$, et $Q \subseteq T(\mathcal{S})$ un ensemble de traces d'évolution. Alors la garantie $\langle\{(u_n)_{n\in I}\} \hookrightarrow Q\rangle_{\mathbb{G}^t_{\mathcal{S},\forall}}$ est valide si et seulement si pour toute trace d'évolution $(v_n)_{n\in J}$ qui est une extension de $(u_n)_{n\in I}$, l'une des deux propriétés suivantes est vérifiée :*

(i) Q contient une extension de $(u_n)_{n\in I}$ qui est préfixe de $(v_n)_{n\in J}$.

(ii) $(v_n)_{n\in J}$ admet une extension stricte dans $T(\mathcal{S})$.

Démonstration. Vérifions tout d'abord le sens réciproque. Nous supposons donc que toute trace d'évolution qui étend $(u_n)_{n\in I}$ vérifie l'une des conditions (i) ou (ii). Nous définissons un invariant de victoire $\mathcal{I}$ pour $\{(u_n)_{n\in I}\}$ et Q de la manière suivante. Un historique H de $\mathbb{G}^t_{\mathcal{S},\forall}$ appartient à $\mathcal{I}$ si toutes les traces d'évolution appartenant à H sont des extensions de $(u_n)_{n\in I}$, et si pour toute trace d'évolution $(w_n)_{n\in K} \in H$, H contient toutes les extensions de $(u_n)_{n\in I}$ qui sont préfixes de $(w_n)_{n\in K}$.

Parmi les quatre conditions définissant un invariant de victoire, nous pouvons établir sans difficulté la condition d'initialisation et les deux conditions de passage à la limite. Quand à la condition d'existence d'une transition, elle suit exactement des propriétés assurées sur les extensions de $(u_n)_{n\in I}$. En effet, la condition (i) ne peut être valide pour le maximum d'un historique de I que si cet historique contient un élément de Q. La condition (ii) permet dans l'autre cas de figure de garantir l'existence d'une unique transition, car l'ensemble des extensions strictes de la trace d'évolution n'est pas vide.

Nous montrons maintenant le sens direct. Supposons donc que la garantie $\langle\{(u_n)_{n\in I}\} \hookrightarrow Q\rangle_{\mathbb{G}_{\mathcal{S},\forall}}$ soit vérifiée, via une stratégie gagnante $\sigma_\exists$. Soit $(v_n)_{n\in J}$ une extension quelconque de $(u_n)_{n\in I}$. Supposons de plus que la condition (i) ne soit pas vérifiée pour $(v_n)_{n\in J}$. Nous allons tout d'abord montrer que $(v_n)_{n\in J}$ est maximum d'un historique (fortement) u-accessible pour $\sigma_\exists$. Notons que si $(u_n)_{n\in I}$ est infinie, nous avons $(u_n)_{n\in I} = (v_n)_{n\in J}$, et donc l'historique $\{(u_n)_{n\in I}\}$ convient. Nous pouvons donc nous restreindre au cas où $(u_n)_{n\in I}$ est finie.

Dans le cas où $(u_n)_{n\in I}$ est fini, nous démontrons par récurrence sur leur longueur que tout préfixe fini de $(v_n)_{n\in J}$ qui soit une extension de $(u_n)_{n\in I}$ est maximum d'un

historique (fortement) u-accessible. En effet, il suffit d'appliquer de manière répétée la règle de propagation (ii) des historiques (fortement) u-accessibles. Comme la condition (i) n'est pas vérifiée pour $(v_n)_{n\in J}$, et qu'il est impossible de trouver un historique (fortement) u-accessible pour $\sigma_\exists$ qui soit perdant, $\sigma_\exists$ doit effectuer un choix valide, ce qui permet d'appliquer la règle de propagation. Nous atteignons ensuite $(v_n)_{n\in J}$ par passage à la borne supérieure, c'est-à-dire les règles de propagation (iii) et (iv). Notons que même si ce procédé de complétion fonctionne systématiquement, il n'est nécessaire que si $(v_n)_{n\in J}$ est infinie.

Maintenant, il nous suffit de remarquer que comme $\sigma_\exists$ est gagnante, alors l'historique H dont $(v_n)_{n\in J}$ est maximum ne peut pas être perdant. Comme la condition (i) n'est pas vérifiée, cela signifie que $\sigma_\exists$ effectue un choix valide en H, et donc que la trace $(v_n)_{n\in J}$ peut être prolongée, validant ainsi la condition (ii). □

Le lemme 4.72 permet notamment de convertir les garanties dans le jeu universel en propriété de correction analogue à la définition d'un triplet de Hoare pour un langage non déterministe. En effet, l'énoncé « pour tout élément de P, l'évolution atteindra éventuellement Q » se traduit comme la garantie $\left\langle \{(x)_{n\in[0;0]}\} \hookrightarrow \{(u_n)_{n\in[0;m]} \mid u_m \in Q\} \right\rangle$ dans le jeu universel.

4.5.3 Traduction locale

Bien que fonctionnelle, la traduction basée sur les traces présente l'inconvénient de polluer l'état du jeu avec des informations éphémères, qui ne sont pas pertinentes pour décrire l'état du système, surtout après un nombre infini de transitions. Dans le cadre de la sémantique opérationnelle d'un langage de programmation, un exemple typique est le contenu de la mémoire des états passés. Cette information n'a aucune influence sur la suite de l'évolution car seule la mémoire à l'état présent compte, et ne présente que peu d'intérêt pour caractériser une exécution infinie. Au contraire, une information pertinente dans cette situation serait la séquence des entrées/sorties.

Nous fournissons donc une seconde traduction des systèmes de transitions vers les jeux, plus locales. Nous avons pour cela non seulement besoin du système de transition, mais également des informations qui restent pertinentes lors d'un comportement infini du système. Nous formalisons ces dernières par un ordre partiel ω-complet.

Définition 4.73 (Ordre partiel ω-complet)**.** Un *ordre partiel ω-complet*, ou ω-cpo, est un ensemble partiellement ordonné dont toute séquence croissante $(a_n)_{n\in\mathbb{N}}$ admet une borne supérieure.

Nous donnons maintenant une structure additionnelle aux systèmes de transition. Essentiellement, nous souhaitons que l'état du système de transition soit partitionné en deux morceaux. Le premier est une partie éphémère, représentée par les valeurs d'un ensemble E, et le second est une partie persistante, représentée par les valeurs d'un ω-cpo $(O, \preccurlyeq)$. Dans ces circonstances, l'ensemble des états du système est représenté par $E \times O$, et la seconde composante croît à chaque transition. Comme seule la partie persistante est considérée comme pertinente dans les situations limites, les comportements infinis peuvent être abstraits par la borne supérieure de cette partie persistante. Nous décrivons donc les états limites du système par cette borne supérieure, qui est un élément de O.

Pour représenter ce découpage, nous avons choisi d'utiliser une projection π qui extrait la partie persistante de l'état d'un système de transition. Ce choix de représentation ne nous permet pas d'extraire une représentation de la partie éphémère de l'état, mais nous n'avons aucunement besoin de représenter cette dernière explicitement pour effectuer la traduction locale.

Définition 4.74 (Système de transition ordonné)**.** Un *système de transition ordonné* est un quintuplet $(S, \to, \pi, O, \preccurlyeq)$ tel que :
- $(S, \to)$ est un système de transition,
- $(O, \preccurlyeq)$ est un ω-cpo,
- π est une fonction de S dans O telle que pour tout $x, y \in S$, si $x \to y$ alors $\pi(x) \preccurlyeq \pi(y)$.

Comme nous avons déjà établi que les états limites du système de transition sont décrits par O, nous allons naturellement réutiliser cet ensemble comme une composante du support de la traduction locale du système en jeux. Il nous reste donc à déterminer quel support utiliser pour les états non limites. L'ensemble S semble un bon candidat, mais ne fournit pas d'ordre compatible avec la relation de transition. En effet, la relation déduite de $\preccurlyeq$ par projection n'est pas forcément antisymétrique. Pour obtenir un ordre de progression, nous équipons les états de $\mathcal{S}$ d'un compteur, incrémenté à chaque transition. Nous représentons donc l'ensemble des états non limits par $S \times \mathbb{N}$.

Définition 4.75 (Support de la traduction locale)**.** Le *support de la traduction locale* associé au système de transition ordonné $\mathcal{S} = (S, \to, \pi, O, \preccurlyeq)$ est l'ensemble ordonné $(G_{\mathcal{S}}, \preccurlyeq_{\mathcal{S}})$ défini par :
- $G_{\mathcal{S}} = (S \times \mathbb{N}) \uplus O$
- $(x, n) \preccurlyeq_{\mathcal{S}} (y, m)$ si et seulement si $n \leq m, \pi(x) \preccurlyeq \pi(y)$, et si lorsque $n = m$ on a $x = y$
- $(x, n) \preccurlyeq_{\mathcal{S}} o$ si et seulement si $\pi(x) \preccurlyeq o$
- pour tout $o \in O$, $\neg(o \preccurlyeq_{\mathcal{S}} (x, n))$
- pour tout $o_1, o_2 \in O$, $o_1 \preccurlyeq_{\mathcal{S}} o_2$ si et seulement si $o_1 \preccurlyeq o_2$

Nous donnons maintenant deux traductions vers les jeux, respectivement existentielles et universelles, et des lemmes de traduction des garanties similaires à ceux donnés pour les traductions à base de traces.

Définition 4.76 (Jeu existentiel local)**.** Le *jeu existentiel local* associé au système de transition ordonné $\mathcal{S} = (S, \to, \pi, O, \preccurlyeq)$ est le jeu $\mathbb{G}^l_{\mathcal{S},\exists}$, dont le domaine est le support de la traduction locale $(G_{\mathcal{S}}, \preccurlyeq_{\mathcal{S}})$ et dont l'ensemble de transitions $\Delta^l_{\mathcal{S},\exists}$ est défini par :
- $\Delta^l_{\mathcal{S},\exists}((x, n)) = \{\{(y, n+1)\} \mid x \to y\}$
- $\Delta^l_{\mathcal{S},\exists}(o) = \emptyset$ si $o \in O$

Définition 4.77 (Jeu universel local)**.** Le *jeu universel local* associé au système de transition ordonné $\mathcal{S} = (S, \to, \pi, O, \preccurlyeq)$ est le jeu $\mathbb{G}^l_{\mathcal{S},\forall}$, dont le domaine est le support de la traduction locale $(G_{\mathcal{S}}, \preccurlyeq_{\mathcal{S}})$ et dont l'ensemble de transitions $\Delta_{\mathcal{S},\forall}$ est défini par :
- $\Delta^l_{\mathcal{S},\forall}((x, n)) = \{\{(y, n+1) \mid x \to y\}\} \backslash \{\emptyset\}$

— $\Delta^l_{\mathcal{S},\forall}(o) = \emptyset$ si $o \in O$

Pour obtenir les lemmes de traduction des garanties, nous allons nous réduire au cas des jeux obtenus par la traduction à base de traces. Pour cela, nous utilisons des simulations pour traduire les garanties dans un sens et dans l'autre. Nous introduisons une famille de relations qui induit ces simulations. Cette famille de relations est paramétrée par un entier $d \in \mathbb{Z}$, qui impose un écart constant entre l'indice maximal d'une trace d'évolution et le compteur de la traduction locale. Nous sommes obligé de contraindre cet écart pour synchroniser les comportements limites des jeux obtenus par les deux traductions. Sans cet écart constant, une partie qui se déroule pendant une infinité d'étapes pourrait être simulée par une partie qui reste au même état.

Définition 4.78 (Relation d'écart entre trace et support local)**.** Soit d un entier relatif et $\mathcal{S} = (S, \to, \pi, O, \preccurlyeq)$ un système de transition ordonné. Nous définissons la relation d'écart d entre les traces d'évolutions $T(\mathcal{S})$ et le support de la traduction local $G_{\mathcal{S}}$ comme la relation $\mathcal{D}_d$ définie par :

$$\begin{array}{ll} (u_m)_{m\in I}\mathcal{D}_d(x,n) & \text{si et seulement si } d \leq n,\ I = [0; n-d] \text{ et } u_{n-d} = x \\ (u_m)_{m\in I}\mathcal{D}_d o & \text{si et seulement si } I = \mathbb{N} \text{ et } \sup_{n\in\mathbb{N}} \pi(u_n) = o \end{array}$$

Lemme 4.79. *Soient $\mathcal{S} = (S, \to, \pi, O, \preccurlyeq)$ un système de transition ordonné et d un entier relatif. Alors pour tout $q \in \{\exists, \forall\}$, la relation $\mathcal{D}_d$ induit une simulation de $\mathbb{G}^t_{\mathcal{S},q}$ par $\mathbb{G}^l_{\mathcal{S},q}$, et la relation $\mathcal{D}_d^{-1}$ induit une simulation dans le sens inverse.*

Démonstration. Par le théorème 4.41, il suffit de démontrer qu'il s'agit dans les quatre cas d'une simulation étape par étape. Pour établir les conditions régulières, il suffit de dérouler les définitions des transitions et relations impliquées. Nous nous intéressons donc aux conditions limites. Pour traiter les quatre cas d'un seul coup, nous généralisons au cas d'une chaîne $\mathcal{X}$ non vide sur $T(\mathcal{S}) \times G_{\mathcal{S}}$ pour l'ordre produit $\leq_p \times \preccurlyeq_{\mathcal{S}}$. Nous avons alors seulement besoin de vérifier que si $\mathcal{X} \subseteq \mathcal{D}_d$, alors la borne supérieure de $\mathcal{X}$ existe et ses deux composantes sont également liées par $\mathcal{D}_d$. En effet, les conditions limites des quatre cas considérés correspondent tous à des instances particulières de telles chaînes, et l'existence de la borne supérieure liée par $\mathcal{D}_d$ rend les garanties à vérifier immédiates par le lemme 4.19.

Soit donc $\mathcal{X}$ ayant les propriétés ci-dessus. Nous pouvons alors construire explicitement une borne supérieure $(\overline{u}_n)_{n\in\overline{I}}$ pour $\pi_1[\mathcal{X}]$, où $\overline{I}$ est l'union des supports des traces de $\pi_1[\mathcal{X}]$ et $\overline{u}_n$ est définie comme la valeur éventuelle (et commune) d'une trace à l'indice n. Si cette trace est un maximum pour $\pi_1[\mathcal{X}]$, alors comme $\mathcal{D}_d$ est fonctionnelle cette trace ne peut être appariée qu'à un seul élément, et $\mathcal{X}$ admet donc un maximum. Ses éléments sont alors liés par $\mathcal{D}_d$ par hypothèse, car ce maximum appartient à $\mathcal{X}$.

Dans le cas où $\pi_1[\mathcal{X}]$ n'admet pas de maximum, remarquons tout d'abord que $\overline{I} = \mathbb{N}$, sans quoi la borne supérieure serait nécessairement atteinte. En particulier, toutes les traces de $\pi_1[\mathcal{X}]$ sont finies, et admettent des longueurs arbitrairement grandes. Nous en déduisons que les éléments de $\pi_2[\mathcal{X}]$ forment une sous-suite infinie de $(\overline{u}_n, n+d)_{n\geq \min(-d,0)}$. Par croissance de $(\pi(\overline{u}_n))_{n\in\mathbb{N}}$, il est alors immédiat de vérifier que la borne supérieure de $\pi_2[\mathcal{X}]$ existe et est $\sup_{n\in\mathbb{N}} \pi(\overline{u}_n)$. En particulier, la borne supérieure de $\mathcal{X}$ existe bien, et ses composantes sont bien liées par $\mathcal{D}_d$. □

Armés de ces simulations, nous pouvons maintenant établir des résultats de traduction des garanties pour les traductions locales des systèmes de transition.

Lemme 4.80. *Soient $\mathcal{S} = (S, \to, \pi, O, \preccurlyeq)$ un système de transition ordonné, $P, Q \subseteq S$ et $Q_\infty \subseteq O$. Alors la garantie $\left\langle \pi_2^{-1}[P] \hookrightarrow \pi_2^{-1}[Q] \cup Q_\infty \right\rangle_{\mathbb{G}^l_{\mathcal{S},\exists}}$ est valide si et seulement si pour tout $x \in P$, il existe une trace d'évolution $(u_n)_{n\in I}$ dont le premier élément est x et pour laquelle l'une des conditions suivantes est vérifiée :*

— *la trace est finie, et son dernier élément appartient à Q*

— *la trace est infinie, et $\sup_{n\in\mathbb{N}} \pi(u_n) \in Q_\infty$.*

Autrement dit, la garantie ci-dessus est valide si et seulement si pour tout élément de P, ou bien il est possible d'accéder à un élément de Q, ou bien il est possible d'accéder « en temps infini » à un élément de Q_∞.

Démonstration. Commençons par le sens direct, et supposons donc que la garantie

$$\left\langle \pi_2^{-1}[P] \hookrightarrow \pi_2^{-1}[Q] \cup Q_\infty \right\rangle_{\mathbb{G}^l_{\mathcal{S},\exists}}$$

soit valide. Par simulation, la garantie

$$\left\langle \mathcal{D}_0^{-1}[\pi_2^{-1}[P]] \hookrightarrow \mathcal{D}_0^{-1}[\pi_2^{-1}[Q] \cup Q_\infty] \right\rangle_{\mathbb{G}^t_{\mathcal{S},\exists}}$$

est également valide. En particulier, soit $x \in P$. Comme $(x)_{n\in[0;0]} \in \mathcal{D}_0^{-1}[\pi_2^{-1}[P]]$, par le lemme 4.70 il existe une trace d'évolution de $\mathcal{D}_0^{-1}[\pi_2^{-1}[Q] \cup Q_\infty]$ dont x est le premier élément. En déroulant la définition de $\mathcal{D}_0$, nous obtenons que cette trace d'évolution a exactement les propriétés voulues.

Pour le sens réciproque, supposons que pour tout $x \in P$, il existe une trace d'évolution avec les propriétés voulues. Pour montrer la garantie voulue, nous utilisons tout d'abord le lemme 4.16 pour réduire le point de départ à un singleton $\{(x, d)\}$. En utilisant le lemme 4.70, il est alors immédiat que la garantie

$$\left\langle \{(x)_{n\in[0;0]}\} \hookrightarrow \mathcal{D}_d^{-1}[\pi_2^{-1}[Q] \cup Q_\infty] \right\rangle_{\mathbb{G}^t_{\mathcal{S},\exists}}$$

est valide. Par simulation la garantie

$$\left\langle \mathcal{D}_d[\{(x)_{n\in[0;0]}\}] \hookrightarrow (\mathcal{D}_d \circ \mathcal{D}_d^{-1})[\pi_2^{-1}[Q] \cup Q_\infty] \right\rangle_{\mathbb{G}^l_{\mathcal{S},\exists}}$$

est également valide. Mais comme $\mathcal{D}_d$ est une relation fonctionnelle, nous pouvons simplifier cette garantie en

$$\langle \{(x, d)\} \hookrightarrow A \rangle_{\mathbb{G}^l_{\mathcal{S},\exists}}$$

avec A un sous-ensemble de $\pi_2^{-1}[Q] \cup Q_\infty$. Nous appliquons alors le lemme 4.20 pour affaiblir et ainsi obtenir la garantie voulue. □

Remarque 4.81. Contrairement au résultat obtenu pour la traduction à base de trace, la traduction donnée par le lemme 4.80 ne permet pas de traduire des garanties arbitraires du jeu existentiel local, mais seulement celles données dans un format qui ignore le compteur. De plus, ce format interdit les états limites dans la précondition de la garantie. Nous avons fait ces restrictions car il s'agit du cas utile

de traduction des garanties, correspondant précisément à la notion d'accessibilité. Nous pourrions néanmoins étendre le raisonnement effectué ci-dessus pour traduire n'importe quelle garantie du jeu existentiel local en terme de propriété existentielle sur les traces d'évolution, mais l'énoncé du lemme doit alors mettre en corrélation les différences possibles entre les compteurs d'arrivée et de départ avec la longueur des traces d'évolution autorisées. Nous trouvons que cela complexifie le lemme pour peu de résultats. Quant à l'extension aux états limites dans la pré-condition, elle se réduit à une simple inclusion de ces derniers dans la post-condition et a donc peu d'intérêt, d'autant plus qu'il ne s'agit pratiquement jamais des états de départ.

Lemme 4.82. *Soient $\mathcal{S} = (S, \to, \pi, O, \preccurlyeq)$ un système de transition ordonné, $P, Q \subseteq S$ et $Q_\infty \subseteq O$. Alors la garantie $\left\langle \pi_2^{-1}[P] \hookrightarrow \pi_2^{-1}[Q] \cup Q_\infty \right\rangle_{\mathbb{G}^l_{\mathcal{S},\forall}}$ est valide si et seulement si pour toute trace d'évolution $(u_n)_{n\in I}$ dont le point de départ appartient à P, au moins une des conditions suivantes est vérifiée :*

- *$(u_n)_{n\in I}$ contient un élément de Q*
- *$(u_n)_{n\in I}$ est infinie, et $\sup_{n\in\mathbb{N}} \pi(u_n) \in Q_\infty$*
- *$(u_n)_{n\in I}$ admet une extension stricte.*

Autrement dit, la garantie ci-dessus est valide si et seulement si toute évolution partant de P atteindra forcément un état de Q en temps fini ou un état de Q_∞ en temps infini.

Démonstration. Établissons tout d'abord le sens direct. En procédant comme pour la traduction existentielle, nous obtenons par simulation que la garantie

$$\left\langle \mathcal{D}_0^{-1}[\pi_2^{-1}[P]] \hookrightarrow \mathcal{D}_0^{-1}[\pi_2^{-1}[Q] \cup Q_\infty] \right\rangle_{\mathbb{G}^t_{\mathcal{S},\forall}}$$

est valide. En particulier, considérons $x \in P$, et la trace particulière $(x)_{n\in[0;0]}$ de $\mathcal{D}_0^{-1}[\pi_2^{-1}[P]]$. Le lemme 4.72 montre alors que toute trace admettant x comme premier élément admet soit un préfixe dans $\pi_2^{-1}[Q] \cup Q_\infty$, ce qui implique l'une des deux premières conditions possibles, soit un prolongement, ce qui est la troisième.

Vérifions maintenant le sens réciproque. Supposons donc que toute trace d'évolution partant d'un élément de P satisfasse l'une des trois conditions ci-dessus. Pour montrer la garantie finale, nous réduisons tout d'abord l'ensemble de départ à un singleton $\{(x, d)\}$. Nous pouvons alors utiliser le lemme 4.72 pour établir la garantie

$$\left\langle \{(x)_{n\in[0;0]}\} \hookrightarrow \mathcal{D}_d^{-1}[\pi_2^{-1}[Q] \cup Q_\infty] \right\rangle_{\mathbb{G}^t_{\mathcal{S},\forall}}.$$

En effet, toute trace d'évolution admettant $x \in P$ comme point de départ admet soit un élément appartenant à Q, et donc en coupant au niveau de cet élément un préfixe dans la post-condition souhaitée, soit est infinie et est lié par $\mathcal{D}_d$ à un élément de Q_∞, et un de ses préfixes (elle-même) est lié à la post-condition souhaitée, soit admet un prolongement. Comme pour la traduction existentielle, il suffit alors d'appliquer la simulation induite par $\mathcal{D}_d$ pour revenir au jeu $\mathbb{G}^l_{\mathcal{S},\forall}$ et obtenir la garantie finale. □

Exemple 4.83. Nous considérons un système de transition sur les paires (p, w) formées d'un compteur positif et d'un mot dans l'alphabet $\{(,)\}$. Autrement dit, w est une suite finie de parenthèses. Étant donné un état du système, les transitions partant de cet état sont les transitions qui ajoutent une parenthèse à la fin du mot,

et où le compteur est incrémenté si la parenthèse est ouvrante, décrémenté sinon. La seconde transition n'existe évidemment pas si le compteur vaut 0. Nous pouvons naturellement étendre la structure de système de transition en une structure de système de transition ordonné $\mathcal{S}_{()}$ en considérant l'ordre préfixe sur les mots, et en complétant par des mots infinis. Nous pouvons donc également parler des mots atteints à la limite en considérant les bornes supérieures des traces d'évolutions.

Bien entendu, si l'on part de l'état $(0, \epsilon)$, il est possible d'atteindre n'importe quel mot bien parenthésé, au sens d'un mot dont toutes les parenthèses fermantes correspondent à une parenthèse ouvrante. De même, toutes les parenthèses fermantes d'un mot atteint, y compris en passant à la limite, correspondront bien à une parenthèse ouvrante.

Nous pouvons formaliser ce résultat de la manière suivante. Posons W le plus grand ensemble de mot finis ou infinis (plus grand point fixe) correspondant à la grammaire suivante :

$$W ::= \epsilon \mid (W)W \mid (W$$

qui est une variante du langage de Dyck pour des mots infinis et des parenthèses potentiellement laissées ouvertes. Nous interprétons la concaténation de mot par $w_1 w_2 = w_1$ si w_1 est infini. Alors les mots atteignables par le système depuis l'état $(0, \epsilon)$ (potentiellement comme cas limites) sont précisément ceux de W. Nous n'allons pas montrer cette propriété ici, mais nous allons montrer que nous pouvons utiliser les conversions présentées précédemment pour transformer cette propriété en un ensemble de garanties équivalent. Nous pouvons alors appliquer toutes les méthodes développées pour la preuve de garanties.

Premièrement, la propriété d'accessibilité de tout mot peut se formuler comme l'existence, pour tout $w \in W$, d'une trace d'évolution partant de $(0, \epsilon)$ et atteignant un état (potentiellement limite) dont le mot correspondant est W. En utilisant le lemme 4.80, nous pouvons réinterpréter cette propriété comme :

$$\forall w \in W.\ \langle \{((0,\epsilon),t) \mid t \in \mathbb{N}\} \hookrightarrow \{((n,w),t) \mid n,t \in \mathbb{N}\} \cup \{w\} \rangle_{\mathbb{G}^l_{\mathcal{S}_{()},\exists}}$$

Nous pouvons également utiliser le lemme 4.82 pour réinterpréter la propriété inverse en termes de garanties, mais cela demande quelques contorsions. Le problème vient du fait que tous les mots atteignables par le système depuis $(0, \epsilon)$ soient dans W n'est pas une propriété de vivacité, mais une propriété de sûreté. Remarquons toutefois que toute trace d'évolution du système contient au plus un mot d'une longueur finie donnée l. De même, les mots de longueur infinie ne sont produits que comme borne supérieure d'une trace d'évolution infinie. Nous pouvons donc exprimer cette propriété de sûreté par la famille de propriétés de vivacité suivante, paramétrée par $l \in \overline{\mathbb{N}}$: en partant de $(0, \epsilon)$, toute évolution atteindra forcément un état dont le mot appartient à l'ensemble W_l des mots de W de longueur l.

En effet, tout mot w pouvant être produit par le système peut être atteint par une trace d'évolution. Cette dernière peut être étendue en une trace infinie par ajout de parenthèses ouvrantes. Mais alors, la propriété de vivacité correspondant à $|w|$ montre que cette trace infinie doit contenir un mot de $W_{|w|}$, qui est forcément w. En termes de garanties, nous obtenons alors la propriété :

$$\forall l \in \overline{\mathbb{N}}.\ \langle \{((0,\epsilon),t) \mid t \in \mathbb{N}\} \hookrightarrow \{((n,w),t) \mid n,t \in \mathbb{N}, w \in W_l\} \cup W_l \rangle_{\mathbb{G}^l_{\mathcal{S}_{()},\forall}}$$

4.6 Développement Why3

Dans cette section, nous présentons une formalisation des jeux en Why3. Celle-ci reprend toutes les notions présentées dans ce chapitre. Nous fournissons pour chaque propriété démontrée un équivalent vérifié en Why3. En fait, nous avons effectué ce développement en parallèle de la formalisation présentée dans les sections précédentes, ce qui nous a forcé à faire attention aux détails dans les démonstrations. Nous avons dû reprendre certains énoncés et preuves à cause de détails bloquants découverts en tentant de vérifier la formalisation en Why3. Par exemple, c'est de cette manière que nous nous sommes rapidement rendu compte de la nécessité de dépendre de l'historique pour les stratégies existentielles. Nous avons également dû reprendre l'énoncé et la démonstration du théorème de simulation plusieurs fois avant d'obtenir un résultat correct.

Le développement peut être trouvé à l'url http://toccata.lri.fr/gallery/hoare_logic_and_games.en.html.

4.6.1 Structure du développement

Notre formalisation est structurée en un certain nombre de théories, elles-mêmes regroupées au sein de divers fichiers source Why3. Ces fichiers sont regroupés en trois catégories : des théories de support, l'axiome du choix, et les fichiers correspondant au développement sur les jeux proprement dit.

Les théories de support fournissent des notions utilisées partout dans notre développement sur les jeux, comme celle d'ensemble ordonné et de chaîne. Nous avons mis l'axiome du choix à part à cause de sa nature axiomatique. Il s'agit en effet de la seule propriété non démontrée du développement. Rappelons que nous avons effectivement eu besoin de cet axiome dans les sections précédentes, notamment pour choisir une famille de stratégies gagnantes lors de la preuve du lemme 4.16, ou pour construire une stratégie gagnante correspondant à un invariant de victoire (théorème 4.33). Nous énonçons cet axiome dans le fichier `choice.mlw`, par l'ajout d'une fonction de choix polymorphe dans la logique.

```
function choice ('a → bool) : 'a
axiom choice_def : ∀p,x:'a. p x → p (choice p)
```

Enfin, les fichiers correspondant au développement sur les jeux sont en correspondance parfaite avec les sections précédentes. Chacun des fichiers définit les notions de la section correspondante, et fournit la preuve des propriétés énoncées dans ladite section. En effet, le fichier `game.mlw` définit les notions de jeux et de stratégies de la section 4.2, et l'équivalence avec les notions d'u-accessibilité et d'invariant de victoire de la section 4.3 est établie dans le fichier `game_no_strat.mlw`. De même, le théorème de simulation (cf. section 4.4) pour les jeux est démontré dans le fichier `game_simulation.mlw`, et les lemmes de mise en correspondance avec les systèmes de transition (cf. section 4.5) sont établis dans le fichier `transition.mlw`.

4.6.2 Méthodes de preuve

Pour effectuer la preuve des divers énoncés, nous utilisons massivement les indicateurs de coupures présentés dans la section 2.3 du chapitre 2. Nous utilisons

ensuite une combinaison de démonstrateurs automatiques pour démontrer les buts obtenus, précisément Alt-Ergo [13], Z3 [37], E [77] et CVC4 [10]. Notons au passage que le simple fait que nous ayons réussi à effectuer ces preuves, notamment celle du théorème de simulation entre jeux (cf. section 4.5), démontre les gains apportés par les indicateurs de coupures pour les preuves. Sans ces indicateurs, nous aurions difficilement pu envisager d'effectuer une telle preuve directement en Why3.

Cependant, les indicateurs ne suffisent pas à garantir la modularité pour un développement de cette ampleur. En effet, nous pouvons les utiliser pour cacher les étapes intermédiaires d'une preuve, mais pas les notions utilisées pour effectuer celle-ci. Par exemple, supposons que nous soyons dans la configuration suivante, où nous établissons le lemme L en utilisant les lemmes établis dans la théorie `B`, et un prédicat additionnel `p` correspondant à une notion intermédiaire pour la preuve :

```
module A
  use import B
  predicate p (...) = ...
  lemma L : ...
end
```

Dans cette situation, les lemmes se trouvant dans `B` et le prédicat `p` seront visibles dans toute théorie qui se réfère directement ou indirectement à la théorie `A`. En particulier, si nous importons la théorie `A` pour fournir aux démonstrateurs automatiques le lemme `L`, nous ajoutons également au contexte tous les lemmes de `B`, ainsi que la définition de `p`. Nous avons observé que cette accumulation d'éléments parasites dans le contexte à tendance à saturer les démonstrateurs automatiques, au point que ceux-ci deviennent parfois incapables d'établir des trivialités.

Pour éviter ce problème, nous avons choisi d'adopter la méthodologie suivante. Lorsque nous souhaitons établir une théorie type « banque de lemmes », nous commençons par écrire une théorie axiomatique `A` où ces lemmes sont posés comme des axiomes. Nous écrivons ensuite une théorie `A_proof`, où nous effectuons les démonstrations de ces mêmes énoncés avec tous les outils nécessaires. Enfin, nous lions les deux théories ensemble en effectuant un « clonage » de la théorie `A` à la fin de la théorie `A_proof`, en remplaçant tous les axiomes par des buts. Cette dernière opération revient à générer des obligations de preuve correspondant exactement à la validité des énoncés de la théorie `A`. Nous pouvons ainsi établir la validité de ces propriétés sans leur attacher les éléments utilisés pour leur preuve.

```
module A "W:non_conservative_extension:N" (* => A_proof *)
  axiom L_1 : ...
  axiom L_2 : ...
end
module A_proof
  ...
  ...
  clone A with goal L_1, goal L_2
end
```

Ce schéma a néanmoins le défaut de n'être pas validé par l'outil Why3. En particulier, aucune vérification n'est effectuée par le système pour vérifier que la

théorie `A_proof` n'utilise pas `A` par ailleurs, ni pour vérifier que les axiomes sont tous remplacés par des buts au niveau du clonage. Nous avons dû effectuer ces vérification par nous-mêmes, et toute personne voulant être absolument certaine de la validité du développement devrait les refaire.

Pour rendre cette opération plus facile, nous avons marqué les théories concernées par l'étiquette `"W:non_conservative_extension:N"` et un commentaire indiquant le nom de la théorie où le clonage est effectué, qui se trouve dans le même fichier. Pour vérifier l'absence d'une telle dépendance incorrecte, nous pouvons alors simplement vérifier le critère plus strict qu'aucune théorie n'utilise la théorie axiomatique `A` entre la définition de `A` et la théorie de justification `A_proof`. En ce qui concerne l'exhaustivité du remplacement des axiomes, nous pouvons simplement vérifier par comptage car Why3 interdit de redéfinir deux fois le même symbole lors d'un clonage.

Notons par ailleurs que l'étiquette `"W:non_conservative_extension:N"` sert normalement à supprimer un avertissement du système, qui signale les axiomes n'utilisant que des symboles définis. Si un tel axiome est a priori une erreur lors de l'écriture d'une théorie axiomatique, il s'agit du cas typique dans notre schéma d'utilisation.

4.6.3 Théories de support

Les théories de support sont réparties dans trois fichiers. Nous fournissons tout d'abord dans le fichier `base.mlw` un nombre conséquent de définitions relatives aux objets d'ordre supérieur, qui sont utilisées à divers endroits du développement. Ces définitions sont assez simples, et pourraient à terme être intégrées à la bibliothèque standard de Why3. Nous donnons par exemple une définition des ensembles et relations quelconques en termes de fonctions vers des booléens. Nous donnons également des opérations simples sur ces objets, comme par exemple l'ensemble vide, l'image d'une fonction/relation, ou encore l'union d'un ensemble d'ensembles. Cette liste non-exhaustive de définitions est donnée dans la figure 4.3.

Comme les jeux sont basés sur les ensembles ordonnés et les chaînes, nous avons également besoin de théories de support pour ces notions. Celles-ci sont définies dans le fichier `order.mlw`. Nous définissons des notions utilisées partout dans le développement, comme celle de borne supérieure ou d'ordre préfixe sur les chaînes. En plus de ces définitions, nous fournissons au sein de ces théories un certain nombre de propriétés sur les chaînes. Nous montrons en particulier que le caractère de chaîne est préservé par l'ajout d'un majorant. Nous montrons également comment construire une borne supérieure pour l'ordre préfixe des chaînes. Ces propriétés sont démontrées à l'aide de la méthode proposée dans la sous-section 4.6.2.

Une fois de plus, cette liste de propriétés (donnée dans la figure 4.4) ne représente pas l'intégralité du contenu du fichier `order.mlw`. Par exemple, nous y démontrons aussi un certains nombre de propriétés sur l'ordre produit, ou encore l'unicité de la borne supérieure.

Enfin, nous fournissons dans le fichier `transfinite.mlw` des théories relatives au processus de génération transfinis. Ces processus permettent notamment de modéliser la notion de partie pour un jeu, ou d'historique fortement u-accessible. Comme nous ne disposons pas de théories des ordinaux en Why3, nous avons utilisé la notion de chaîne pour représenter ces séquences (l'ordre étant de toute façon fourni avec les jeux) et de plus petit point fixe pour représenter la récurrence transfinie.

```
(* Ensemble *)
type set 'a    = 'a → bool

(* Relation *)
type rel 'a 'b = 'a → 'b → bool

(* Endorelation *)
type erel 'a   = rel 'a 'a

(* ∅ *)
constant none : set 'a = λ_. false

(* s ∪ {x} *)
function add (s:set 'a) (x:'a) : set 'a =
  λy. s y ∨ x = y

(* y ∈ f(s) *)
predicate image (f:'a → 'b) (s:set 'a) (y:'b) =
  ∃x. s x ∧ f x = y

(* y ∈ r[s] *)
predicate related (r:rel 'a 'b) (s:set 'a) (y:'b) =
  ∃x. s x ∧ r x y

(* y ∈ ⋃c *)
predicate bigunion (c:set (set 'a)) (y:'a) =
  ∃s. c s ∧ s y

(* Relation transitive *)
predicate transitive (r:erel 'a) =
  ∀x y z. r x y ∧ r y z → r x z

(* Relation d'ordre *)
predicate order (o:erel 'a) =
  reflexive o ∧ antisymmetric o ∧ transitive o
```

FIGURE 4.3 – Éléments extraits des théories de support du fichier `base.mlw`

```
(* Majorant d'un ensemble *)
predicate upper_bound (o:erel 'a) (s:set 'a) (x:'a) =
  ∀y. s y → o x y

(* Borne supérieure *)
predicate supremum (o:erel 'a) (s:set 'a) (x:'a) =
  minimum o (upper_bound o s) x

(* Chaînes pour un ordre donné *)
predicate chain (o:erel 'a) (c:set 'a) =
  ∀x y. s x ∧ s y → o x y ∨ o y x

(* c1 ⊑ c2 *)
predicate subchain (o:erel 'a) (c1 c2:set 'a) =
  subset c1 c2 ∧ ∀x y. c1 x ∧ c2 y ∧ not c1 y → o x y

(* Fonction monotone sur un ensemble *)
predicate monotone_on (s:'a → bool)
                      (o1:erel 'a) (f:'a → 'b) (o2:erel 'b) =
  ∀x y. s x ∧ s y ∧ o1 x y → o2 (f x) (f y)

(* Ajouter un majorant à une chaîne crée une chaîne plus grande *)
lemma add_subchain : ∀o ch,x:'a.
  upper_bound o ch x → subchain o ch (add ch x)

lemma add_chain : ∀o ch,x:'a.
  reflexive o ∧ chain o ch ∧ upper_bound o ch x → chain o (add ch x)

(* La borne supérieure d'une chaîne pour l'ordre préfixe
   est sa réunion *)
lemma subchain_completion : ∀o:erel 'a,ch:set (set 'a).
  chain (subchain o) ch → supremum o ch (bigunion ch)

lemma chain_subchain_completion : ∀o:erel 'a,ch:set (set 'a).
  chain (subchain o) ch ∧ (∀x. ch x → chain o x) →
    chain o (bigunion ch)
```

FIGURE 4.4 – Éléments extraits des théories de support du fichier `order.mlw`

Nous définissons un processus de génération transfini de la manière suivante. Étant donnée un ensemble ordonné $(O, \preccurlyeq)$, une relation $\mathcal{S}$, dite de succession, entre les chaînes de O et les majorants de ces chaînes, et une chaîne initiale H_0, nous définissons le processus de génération associé comme le plus petit ensemble de chaînes satisfaisant les règles de propagation suivantes :

— H_0 appartient au processus de génération,
— Pour toute chaîne H du processus de génération, alors pour tout x tel que $H\mathcal{S}x$, la chaîne $H \cup \{x\}$ appartient au processus de génération,
— Pour toute chaîne non vide $\mathcal{H}$ pour l'ordre préfixe $\sqsubseteq$ qui soit sous-ensemble du processus de génération, la borne supérieure $\bigcup\mathcal{H}$ de $\mathcal{H}$ appartient au processus de génération.

Nous pouvons remarquer qu'il s'agit bien d'une généralisation des notions de parties (cf. définition 4.8) et d'historique fortement u-accessibles (cf. définition 4.28). Nous formalisons précisément les processus de génération transfinis en Why3 par la propriété inductive `tr_ext` suivante, dont les différents cas correspondent respectivement aux règles de propagation ci-dessus.

```
inductive tr_ext (erel 'a) (rel (set 'a) 'a) (_ _:set 'a) =
  | Ext_base : ∀o succ,chb:set 'a. tr_ext o succ chb chb
  | Ext_succ : ∀o succ chb ch,x:'a.
    tr_ext o succ chb ch ∧ succ ch x → tr_ext o succ chb (add ch x)
  | Ext_sup : ∀o succ chb,chh:set (set 'a).
    chain (subchain o) chh ∧ inhabited chh ∧
    (∀ch. chh ch → tr_ext o succ chb ch) →
    tr_ext o succ chb (bigunion chh)
```

Les arguments de `tr_ext` correspondent respectivement à l'ordre $\preccurlyeq$, à la relation de succession $\mathcal{S}$, à la chaîne initiale H_0 et à la chaîne générée par le processus. Cette définition n'a un sens que si le premier argument est un ordre, et si le second argument est une relation de succession pour cet ordre. Nous formalisons la notion de relation de succession pour un ordre par le prédicat suivant :

```
predicate tr_succ (o:erel 'a) (succ:rel (set 'a) 'a) =
  ∀h x. succ h x → upper_bound o h x
```

En plus d'effectuer ces définitions, nous démontrons un certains nombre de propriétés pour ces processus, que nous fournissons comme lemmes. Nous donnons certains de ces lemmes dans la figure 4.5. Nous démontrons notamment certaines propriétés simples mais utiles, comme par exemple le fait que tout ensemble généré par le processus est effectivement une chaîne plus grande que la chaîne initiale. Nous démontrons également des propriétés plus puissantes, comme un critère permettant de transporter la génération d'une chaîne entre deux processus pour deux relations de succession distinctes. De fait, ce critère contient la quasi-totalité du raisonnement derrière le lemme 4.23.

Enfin, nous définissons également la notion de processus de génération transfini déterministe, en restreignant la seconde règle de propagation au cas où la relation de succession ne lie pas la chaîne à plus d'un majorant. Ce type de processus correspond à celui des parties d'un jeu, et possède des propriétés supplémentaires que nous établissons, comme le fait que toutes les chaînes générées sont comparables pour

```
(* Les objets générés sont des chaînes plus grandes *)
lemma tr_ext_preserve_chain : ∀o succ,h1 h2:set 'a.
  order o ∧ tr_succ o succ ∧ chain o h1 ∧ tr_ext o succ h1 h2 →
  chain o h2

lemma tr_ext_compare : ∀o succ,h1 h2:set 'a.
  order o ∧ tr_succ o succ ∧ tr_ext o succ h1 h2 → subchain o h1 h2

(* Les chaînes intermédiaires sont toutes générées. *)
lemma tr_ext_decompose : ∀o succ,h1 h2 h3:set 'a.
  order o ∧ tr_succ o succ ∧ tr_ext o succ h1 h3 →
  subchain o h1 h2 ∧ subchain o h2 h3 →
    tr_ext o succ h1 h2 ∧ tr_ext o succ h2 h3

(* Si la relation de successeur ne permet pas d'atteindre
   un nouvel élément, la chaîne alors générée est maximale. *)
lemma tr_ext_stop_progress : ∀o succ,h1 h2:set 'a.
  order o ∧ tr_succ o succ →
  tr_ext o succ h1 h2 ∧ subset (succ h2) h2 →
  maximal (subchain o) (tr_ext o succ h1) h2

(* Critère de transport de la propriété de génération
   d'une relation à une autre. *)
predicate transport_criterion (o:erel 'a) (succ:rel (set 'a) 'a)
  (h1 h2:set 'a) =
  ∀h x.
    subchain o h1 h ∧ subchain o (add h x) h2 ∧
    upper_bound o h x ∧ not h x → succ h x

lemma transport : ∀o succ1 succ2,h1 h2:set 'a.
  transitive o ∧ tr_succ o succ1 ∧ tr_ext o succ1 h1 h2 ∧
  transport_criterion o succ2 h1 h2 →
  tr_ext o succ2 h1 h2

lemma transport_criterion : ∀o succ,h1 h2:set 'a.
  order o ∧ tr_succ o succ ∧ tr_ext o succ h1 h2 →
  transport_criterion o succ h1 h2
```

FIGURE 4.5 – Propriétés extraites des théories de support du fichier `transfinite.mlw`

l'ordre préfixe. Nous établissons également des lemmes de transfert entre les deux types de processus.

Pour démontrer toutes ces propriétés sur les processus de génération transfinis, nous devons presque systématiquement effectuer un raisonnement de type inductif. Nous utilisons pour cela la transformation `induction_pr` de l'outil Why3, que nous avons présentée dans la section 2.2.

4.6.4 Jeux dans l'outil Why3

Nous définissons les notions de jeux et de stratégies dans le fichier `game.mlw`, qui correspondent précisément à la section 4.2. Nous y définissons les jeux par un type enregistrement `game 'a`, dont les champs correspondent à l'ordre du jeu et à son ensemble de transitions. Le domaine du jeu est donné par le paramètre de type `'a`. Nous munissons le type enregistrement `game 'a` d'un invariant correspondant à la condition de progression du jeu, ce qui assure que toute valeur de ce type satisfait effectivement cette condition. Notons que comme les types de Why3 ne peuvent pas être vides, l'outil génère une obligation d'habitation. Nous la vérifions en fournissant un exemple explicite de jeu via `by`.

```
type game 'a = {
  progress : erel 'a;
  transition : 'a → set (set 'a)
} invariant { order progress }
invariant { ∀x s y. transition x s ∧ s y → progress x y }
by { progress = (=); transition = λ_ _. false }
```

Nous définissons naturellement les stratégies existentielles et universelles par des fonctions :

```
(* stratégie existentielle: historique → transition *)
type angel 'a = set 'a → set 'a
(* stratégie universelle: transition → élément suivant *)
type demon 'a = set 'a → 'a
```

Nous définissons ensuite les parties comme une instance des processus de génération transfinis fournis par les théories de support, en utilisant la relation de succession suivante :

```
predicate step (g:game 'a) (ang:angel 'a)
               (dmn:demon 'a) (h:set 'a) (n:'a) =
  let x = choice (supremum g.progress h) in
  let a = ang h in
  let d = dmn a in
  supremum g.progress h x ∧
  if h x then g.transition x a ∧ a d ∧ n = d else n = x
```

Celle-ci lie une chaîne à sa borne supérieure si elle n'est pas déjà atteinte, et au choix effectué par la cascade des deux stratégies s'il est valide. Nous pouvons alors facilement vérifier qu'en considérant le processus de génération transfini donné en Why3 par `tr_ext g.progress (step g ang dmn) ((=) x)`, associé à l'ordre du jeu, à la relation de succession ci-dessus, et à l'ensemble de départ $\{x\}$, nous retrou-

vons exactement la définition 4.8 d'une partie en termes de plus petit ensemble clos par des règles de propagation.

Nous pouvons alors définir sans difficulté les autres notions de la section 4.2, ainsi qu'énoncer les mêmes lemmes. Nous donnons ces éléments dans la figure 4.6. Le seul lemme manquant est le corollaire 4.17 de partitionnement de la pré-condition d'une garantie, qui découle immédiatement du lemme 4.16. Nous avons estimé qu'il n'était pas nécessaire d'énoncer ce corollaire dans notre développement, car en pratique nous utilisons directement le lemme dont il découle.

De manière analogue, nous définissons dans les fichiers `game_no_strat.mlw` et `game_simulation.mlw` les notions correspondant aux sections 4.3, et 4.4. Nous établissons également les propriétés correspondantes. Ces définitions, propriétés et preuves ne présentent pas spécialement de particularités par rapport à leurs équivalents écrits dans cette thèse. Par exemple, nous définissons la notion d'invariant de victoire par le prédicat suivant :

```
predicate next_hist (inv:set (set 'a)) (h:set 'a) (y:'a) =
  inv (add h y)

predicate victory_invariant (g:game 'a) (p q:set 'a)
                            (inv:set (set 'a)) =
  let o = g.progress in
  (∀x. p x → inv ((=) x)) ∧
  (∀h x. inv h ∧ not inhabited (inter h q) ∧ maximum o h x →
    ∃a. g.transition x a ∧ not a x
              ∧ subset a (next_hist inv h)) ∧
  (∀ch. subset ch inv ∧ chain (subchain o) ch ∧ inhabited ch →
    inv (bigunion ch)) ∧
  (∀h x. inv h ∧ supremum o h x → inv (add h x))
```

ce qui correspond exactement à la définition 4.32. De même, la partie la plus volumineuse correspond sans surprise à la preuve du théorème de simulation.

De la même manière, les fichiers `transition.mlw` et `ordered_transition.mlw` définissent les deux encodages des systèmes de transition en jeu donnés dans la section 4.5.

4.6.5 Volume du développement

Dans sa version actuelle, notre développement représente presque 6000 lignes de Why3 et 4306 obligations de preuve envoyées aux démonstrateurs automatiques. Cette dernière mesure représente assez bien la partie manuelle du travail de preuve, puisque presque toutes les obligations de preuves sont induites par un indicateur de coupure, ajouté pour représenter une étape de preuve (cf. section 2.3). Nous donnons la répartition entre les différents fichiers dans le tableau de la figure 4.7. Notons que ce tableau référence des fichiers dont nous n'avons pas encore parlé, `subgame.mlw` et `game_logic.mlw`, qui correspondent à la formalisation en Why3 de la première section du chapitre suivant. Nous détaillerons le contenu de ces fichiers à la fin de celle-ci.

Étant donnée la taille conséquente de la formalisation des jeux dans cette thèse, nous ne sommes pas outre mesure surpris de la taille du développement Why3 asso-

```
(* Historique gagnant (définition 4.11) *)
predicate win_at (g:game 'a) (win:set 'a)
                 (ang:angel 'a) (dmn:demon 'a) (h:set 'a) =
  let o = g.progress in let x = sup o h in
  not supremum o h x ∨ (maximum g.progress h x ∧
  (win x ∨ let a = ang h in g.transition x a ∧ not a (dmn a)))

predicate win_against (g:game 'a) (x:'a) (win:set 'a)
                      (ang:angel 'a) (dmn:demon 'a) =
  ∃h. tr_ext g.progress (step g ang dmn) ((=) x) h ∧
    win_at g win ang dmn h
(* Stratégie gagnante (définition 4.12) *)
predicate winning_strat
  (g:game 'a) (x:'a) (win:set 'a) (ang:angel 'a) =
  ∀dmn. win_against g x win ang dmn
predicate uniform_winning_strat
  (g:game 'a) (strt win:set 'a) (ang:angel 'a) =
  ∀x. strt x → winning_strat g x win ang

predicate have_winning_strat (g:game 'a) (x:'a) (win:set 'a) =
  ∃ang. winning_strat g x win ang
(* Garantie (définition 4.15) *)
predicate have_uniform_winning_strat
  (g:game 'a) (start win:set 'a) =
  ∃ang. uniform_winning_strat g start win ang
(* lemme 4.16 *)
lemma have_uniform_winning_strat_quantifier_inversion :
  ∀g:game 'a,start win.
    (have_uniform_winning_strat g start win ↔
      (∀x. start x → have_winning_strat g x win))
(* lemme 4.18 *)
lemma have_winning_strat_local_criterion : ∀g,x:'a,win.
  (have_winning_strat g x win ↔
   win x ∨ ∃s. g.transition x s
               ∧ have_uniform_winning_strat g s win)
(* corollaire 4.19 *)
lemma uniform_winning_strat_subset :
  ∀g:game 'a,start win ang.
   subset start win → uniform_winning_strat g start win ang
(* lemme 4.20 *)
axiom uniform_winning_strat_mono :
  ∀g:game 'a,start1 start2 win1 win2 ang.
    subset start2 start1 ∧ subset win1 win2 →
    uniform_winning_strat g start1 win1 ang →
    uniform_winning_strat g start2 win2 ang
```

FIGURE 4.6 – Notions et propriétés correspondant à la section 4.2

fichier	lignes de déclarations	obligations de preuves déchargées par les démonstrateurs automatiques
`base.mlw`	186	50
`choice.mlw`	11	3
`order.mlw`	400	251
`transfinite.mlw`	369	199
`game.mlw`	279	181
`game_no_strat.mlw`	396	231
`game_simulation.mlw`	2259	1737
`transition.mlw`	394	266
`ordered_transition.mlw`	382	252
`subgame.mlw`	23	15
`game_logic.mlw`	1250	1121
total	5949	4306

FIGURE 4.7 – Volume de la formalisation des jeux en Why3

cié. Celle-ci ne nous semble pas excessive vis-à-vis des notions mises en jeu. Notons également que cette formalisation démontre la possibilité d'effectuer un développement logique conséquent directement dans l'outil Why3, et d'une certaine manière la possibilité (certes limitée) d'utiliser cet outil comme assistant de preuve.

Chapitre 5

Une logique de programme pour les jeux

5.1 Objectifs

Nous avons développé dans le chapitre précédent la notion de jeu et de garantie, correspondant intuitivement à une interprétation générale de la logique de Hoare. Nous allons maintenant rendre cette intuition formelle en développant des outils de preuve pour les garanties, inspirés directement de la logique de Hoare et du calcul de plus faible pré-condition.

Remarquons que nous ne pouvons pas nous contenter des règles classiques de la logique de Hoare. En effet, rappelons que l'une de nos motivations (cf. section 4.1)

pour introduire le cadre des jeux est de pouvoir établir des propriétés sur les comportements qui ne terminent pas. Nous pouvons énoncer de telles propriétés dans le cadre des jeux, mais c'est complètement inutile si nos règles d'inférence ne permettent pas de les démontrer. Nous devons donc adapter les règles d'inférence pour prendre en compte ces possibilités.

Cette adaptation concerne surtout la règle d'itération, laquelle correspond à une structure de boucle, puisqu'elle peut introduire de nouveaux comportements infinis. Nous l'adaptons en ajoutant l'équivalent en terme de garanties d'un code gestionnaire de divergence, qui doit permettre de terminer ou de reprendre l'exécution de la boucle à partir d'une « situation limite », autrement dit, d'un historique limite de l'itération. Pour limiter les cas limites possibles, nous équipons l'itération d'un ordre de progression sur ces paramètres, qui doit croître à chaque itération et en cas de reprise dans le gestionnaire de divergence. Si l'ordre miroir de l'ordre de progression est bien fondé, il n'y a aucun cas limite possible, et nous retrouvons exactement la règle d'itération de la logique de Hoare pour la correction totale.

Enfin, nous avons en pratique besoin d'ajouter une règle d'inférence supplémentaire, une règle de récurrence correspondant à une structure récursive. Comme nos autres motivations, cette nécessité vient tout d'abord de l'exemple des compilateurs. Nous avons en effet besoin d'une façon de prouver la compilation d'un programme récursif, et l'utilisation d'une règle de récurrence est a priori la manière la plus simple possible. Comme une structure récursive peut être mal fondée, et donc générer de nouvelles situations limites, nous l'adaptons de la même manière que la règle d'itération. Nous ajoutons l'équivalent en terme de garanties d'un gestionnaire de « débordement de pile infini », qui doit choisir l'un des appels récursifs sur la pile et le « refermer » en choisissant son résultat.

Bien que nous puissions en théorie nous passer complètement de règles supplémentaires, en pratique la règle de récurrence simplifie considérablement les choses. Nous pourrions simuler la structure récursive par une structure itérative, et c'est d'ailleurs de cette manière que nous établissons la règle de récurrence dans le cas général, mais pour cela nous devons typiquement expliciter une structure de pile auxiliaire et son évolution au sein d'une structure de boucle. La preuve de plusieurs programmes de nature dérécursifiée [1] [2] via l'outil Why3, notamment l'algorithme de Schorr-Waite [76], a achevé de nous convaincre de la pénibilité d'une telle approche comparé à une preuve en style récursif.

Pour établir la règle de récurrence, nous utilisons comme annoncé une conversion générale de la structure récursive vers une structure itérative. Nous avons pour cela besoin de récupérer les différents fragments de la structure récursive uniquement à partir des garanties correspondantes. Nous créons pour cela des jeux étendus par des transitions supplémentaires, qui correspondent à la validation des hypothèses récursives. Le « code » d'un niveau d'appel récursif correspond alors intuitivement à la stratégie gagnante dans le jeu étendu correspondant. Notons par ailleurs que ces jeux étendus ont la propriété intéressante de n'être en général ni purement existentiels, ni purement universels, et justifient donc l'emploi de la construction hybride que sont les jeux plutôt qu'une simple alternative entre le comportement existentiel et le comportement universel d'un système de transition. De plus, un effet secondaire de

1. http://toccata.lri.fr/gallery/verifythis_2016_tree_traversal.en.html
2. http://toccata.lri.fr/gallery/schorr_waite.en.html

ce type de construction est de permettre l'établissement d'une règle supplémentaire correspondant à la prise d'une continuation.

Le chapitre est organisé de la manière suivante. Dans la section 5.2, nous introduisons l'équivalent d'une logique de Hoare sur les jeux, basée sur les garanties plutôt que sur des triplets. Nous donnons ensuite dans la section 5.3 un système de preuve alternatif pour les garanties, analogue du calcul de plus faible pré-condition. Ce système est basé sur la combinaison de *transformateur de post-conditions* pour automatiser l'application de certaines règles d'inférence. Enfin, nous donnons dans la section 5.4 quelques exemples simples d'application du système de preuve.

5.2 Une logique de Hoare pour les jeux

La notion de garantie 4.15 que nous avons définie dans la section 4.2 correspond intuitivement à un triplet de Hoare : l'ensemble de départ correspond à la pré-condition, l'ensemble cible à la post-condition, et d'une certaine manière le jeu correspond au code du programme. Nous justifions maintenant cette correspondance en développant une logique pour les garanties, dont les règles sont analogues à celles de la logique de Hoare.

5.2.1 Logique et règles d'inférence

Définition 5.1 (Contexte et énoncés)**.** Un *contexte* Γ sur le domaine G est un ensemble (potentiellement infini) de paires de sous-ensembles de G. Un *énoncé* sur le domaine G est la donnée d'un contexte Γ sur G, d'un ordre $\preccurlyeq$ sur G et d'une paire P, Q de sous-ensembles de G.

Définition 5.2 (Satisfaction d'un contexte)**.** Un contexte Γ sur le domaine G est *satisfait* par un jeu $\mathbb{G} = (G, \preccurlyeq, \Delta)$ si pour toute paire $(P, Q) \in \Gamma$, la garantie $\langle P \hookrightarrow Q \rangle_{\mathbb{G}}$ est valide.

Définition 5.3 (Validité d'un énoncé)**.** Un énoncé sur le domaine G donné par Γ, $\preccurlyeq$ et les ensembles P, Q est dit *valide* si, pour tout jeu $\mathbb{G}$ admettant $(G, \preccurlyeq)$ comme support et satisfaisant Γ, la garantie $\langle P \hookrightarrow Q \rangle_{\mathbb{G}}$ est valide.

Définition 5.4 (Notation pour les contextes et énoncés)**.** Comme le contexte représente un ensemble d'hypothèses, nous utilisons les notations standard Γ, Γ' pour représenter l'union de deux contextes, $\forall a \in A.\ \Gamma_a$ pour représenter l'union d'une famille de contextes et $\langle P \hookrightarrow Q \rangle$ pour représenter un contexte singleton. De même, les énoncés jouent le rôle d'un séquent. Nous utilisons donc la notation $\Gamma \vdash_{\preccurlyeq} \langle P \hookrightarrow Q \rangle$ pour l'énoncé formé de Γ, de $\preccurlyeq$, et de P, Q.

Nous employons le contexte Γ pour représenter les briques de base à partir desquelles la garantie finale est établie. Si nous voulions représenter des triplets de Hoare pour un langage de programmation donné, nous prendrions naturellement un contexte correspondant à la sémantique étape par étape du langage. Plus précisément, nous utiliserions un contexte contenant une hypothèse par règle de réduction de cette sémantique.

Nous pouvons également voir le contexte Γ d'une autre manière. Les garanties du contexte peuvent en effet être considérées comme des procédures déjà vérifiées.

En particulier, nous pouvons utiliser le contexte pour introduire des hypothèses récursives.

Nous allons maintenant donner des règles d'inférence pour les énoncés. Cependant, comme nous prenons en compte les comportements infinis, nous devons caractériser les situations limites à prendre en compte dans les règles générant de tels comportements.

Définition 5.5 (Ensemble des situations limites)**.** Étant donnés deux ensembles ordonnés $(G, \preccurlyeq)$, $(O, \leq)$ et une famille $(X_o)_{o \in O}$ de sous-ensembles de G indexés par O, nous définissons l'ensemble des *situations limites* pour le support $(G, \preccurlyeq)$ sous les contraintes $(O, \leq), X$ comme étant l'ensemble des paires (H, f) telles que :

— H est une chaîne non vide de $(O, \leq)$, qui n'admet pas de maximum
— f est une fonction croissante de $(H, \leq)$ dans $(G, \preccurlyeq)$ telle que pour tout $o \in H$, $f(o) \in X_o$
— $f(H)$ admet une borne supérieure.

Nous notons l'ensemble des situations limites par $\texttt{limit}_{\preccurlyeq}(\leq, X)$.

Nous donnons la liste des règles d'inférence dans la figure 5.1. Ces règles correspondent presque toutes à des règles de la logique de Hoare habituelle, à ceci près que nous n'utilisons pas la structure du programme pour guider l'application des règles :

— la règle d'identité correspond à la règle d'une instruction vide.
— la règle d'axiome correspond à l'application d'une spécification déjà connue.
— la règle de remplacement est une règle purement logique servant à remplacer un contexte par un autre. Elle permet notamment d'affaiblir le contexte et/ou d'y ajouter des garanties qui en sont déductibles.
— les règles de composition et de conséquence sont exactement les règles habituelles.
— la règle de partition est l'analogue des règles de nature conditionnelle. En fait, il s'agit précisément de l'équivalent de la construction alternative du langage à commandes gardées introduit par Dijkstra [40], à ceci près que la partition n'est pas donnée par la structure du programme.
— la règle d'itération est une extension directe de la règle d'itération de la logique de Hoare, en ajoutant la contrainte qu'il est également possible de repartir d'une situation limite. Dans ce dernier cas, le paramètre d'itération doit être choisi comme un majorant des valeurs prises par ce paramètre pour les états qui ont mené à cette situation. En l'absence de conditions de terminaison explicitement fournies par un programme, nous représentons l'alternative entre la terminaison de la boucle et la préservation de l'invariant par une disjonction au niveau de la post-condition.
— la règle de récurrence correspond à une règle de récurrence standard pour une logique de Hoare sur un langage avec procédures, à l'exception près de la contrainte limite. Celle-ci exprime le fait qu'à partir d'une situation limite, qui correspond intuitivement à un « débordement de pile infini », il est possible de terminer l'exécution de l'un des appels récursifs ayant donné lieu au débordement.
— la règle de continuation n'a pas d'analogue dans la logique de Hoare standard, mais correspond naturellement à un mécanisme d'échappement, analogue à

$$\textsc{Axiome}\ \frac{}{\Gamma, \langle P \hookrightarrow Q\rangle \vdash_{\preccurlyeq} \langle P \hookrightarrow Q\rangle} \qquad \textsc{Identité}\ \frac{P \subseteq Q}{\Gamma \vdash_{\preccurlyeq} \langle P \hookrightarrow Q\rangle}$$

$$\textsc{Partition}\ \frac{\forall i \in I.\ \Gamma \vdash_{\preccurlyeq} \langle P_i \hookrightarrow Q\rangle}{\Gamma \vdash_{\preccurlyeq} \langle \bigcup_{i\in I} P_i \hookrightarrow Q\rangle}$$

$$\textsc{Conséquence}\ \frac{P_2 \subseteq P_1 \qquad \Gamma \vdash_{\preccurlyeq} \langle P_1 \hookrightarrow Q_1\rangle \qquad Q_1 \subseteq Q_2}{\Gamma \vdash_{\preccurlyeq} \langle P_2 \hookrightarrow Q_2\rangle}$$

$$\textsc{Remplacement}\ \frac{\Gamma \vdash_{\preccurlyeq} \langle P_0 \hookrightarrow Q_0\rangle \qquad \forall \langle P \hookrightarrow Q\rangle \in \Gamma.\ \Gamma' \vdash_{\preccurlyeq} \langle P \hookrightarrow Q\rangle}{\Gamma' \vdash_{\preccurlyeq} \langle P_0 \hookrightarrow Q_0\rangle}$$

$$\textsc{Progression}\ \frac{\Gamma \vdash_{\preccurlyeq} \langle \{x\} \hookrightarrow Q\rangle}{\Gamma \vdash_{\preccurlyeq} \langle \{x\} \hookrightarrow \{y \in Q \mid x \preccurlyeq y\}\rangle}$$

$$\textsc{Composition}\ \frac{\Gamma \vdash_{\preccurlyeq} \langle P \hookrightarrow Q\rangle \qquad \Gamma \vdash_{\preccurlyeq} \langle Q \hookrightarrow R\rangle}{\Gamma \vdash_{\preccurlyeq} \langle P \hookrightarrow R\rangle}$$

$$\textsc{Itération}\ \frac{\begin{array}{c}(O, \leq) \text{ ensemble ordonné}, \quad o_1 \in O \\ \forall o \in O.\ \Gamma \vdash_{\preccurlyeq} \langle I_o \hookrightarrow Q \cup \bigcup_{o'>o} I_{o'}\rangle \\ \forall (H, f) \in \mathtt{limit}_{\preccurlyeq}(\leq, I).\ \Gamma \vdash_{\preccurlyeq} \left\langle \{\sup f(H)\} \hookrightarrow Q \cup \bigcup_{o\geq H} I_o \right\rangle\end{array}}{\Gamma \vdash_{\preccurlyeq} \langle I_{o_1} \hookrightarrow Q\rangle}$$

$$\textsc{Continuation}\ \frac{\Gamma, \langle Q \hookrightarrow \emptyset\rangle \vdash_{\preccurlyeq} \langle P \hookrightarrow Q\rangle}{\Gamma \vdash_{\preccurlyeq} \langle P \hookrightarrow Q\rangle}$$

$$\textsc{Récurrence}\ \frac{\begin{array}{c}(O, \leq) \text{ ensemble ordonné}, \quad o_1 \in O \\ \forall o \in O.\ \Gamma, (\forall o' > o.\ \langle P_{o'} \hookrightarrow Q_{o'}\rangle) \vdash_{\preccurlyeq} \langle P_o \hookrightarrow Q_o\rangle \\ \forall (H, f) \in \mathtt{limit}_{\preccurlyeq}(\leq, P). \\ \Gamma, (\forall o \geq H.\ \langle P_o \hookrightarrow Q_o\rangle) \vdash_{\preccurlyeq} \langle \{\sup f(H)\} \hookrightarrow \bigcup_{o\in H} Q_o\rangle\end{array}}{\Gamma \vdash_{\preccurlyeq} \langle P_{o_1} \hookrightarrow Q_{o_1}\rangle}$$

FIGURE 5.1 – Règles d'inférence

celui fourni par `call/cc` [21], l'opérateur `escape` de Reynolds [75] ou l'opérateur J de Landin [55]. Cet opérateur permet de capturer son contexte d'exécution, c'est-à-dire l'état de contrôle courant du programme, et de le passer en paramètre à une procédure. Le contexte est capturé sous la forme d'une continuation, qui permet de relancer l'exécution du programme directement sous ce contexte.

— la règle de progression exprime le fait qu'une garantie respecte nécessairement l'ordre du jeu sous-jacent. Nous pouvons voir cette règle comme l'exploitation des informations déductibles d'un invariant historique [59, 61].

Notons que d'un point de vue purement logique, cet ensemble de règles est inutilement large. Comme nous le montrerons dans le théorème 5.37, les règles d'axiome, d'identité, de progression, de conséquence et d'itération forment à elles seules un ensemble de règles complet pour les énoncés, au sens que n'importe quel énoncé valide est dérivable en utilisant ces seules règles. Les autres règles ne sont donc pas nécessaires *stricto sensu*, mais présentent un intérêt pratique car elles simplifient les dérivations. En particulier, cela nous évite de refaire les constructions sous-jacentes à la preuve de ces règles, ce qui est un avantage plus que conséquent dans le cas de la règle de récurrence.

5.2.2 Preuve des règles d'inférence (hors récurrence)

Nous démontrons maintenant la validité des règles énoncées, à l'exception dans un premier temps de la règle de récurrence, car la preuve de celle-ci ne suit pas directement des outils déjà introduits. Nous décomposons cette preuve en un certain nombre de lemmes, en montrant les règles au fur et à mesure.

Lemme 5.6. *Les règles d'axiome et de remplacement sont valides.*

Démonstration. Immédiat par définition de la validité des énoncés. □

Lemme 5.7. *Les règles d'identité, de partition et de conséquence sont valides.*

Démonstration. Immédiat par les lemmes respectifs 4.19, 4.17 et 4.20. □

Lemme 5.8. *La règle de progression est valide.*

Démonstration. Considérons Γ, x, Q tels que $\Gamma \vdash_{\preccurlyeq} \langle \{x\} \hookrightarrow Q \rangle$ soit valide. Soit $\mathbb{G}$ un jeu satisfaisant Γ. Immédiatement, $\langle \{x\} \hookrightarrow Q \rangle_{\mathbb{G}}$ est vérifiée, donc il existe une stratégie gagnante $\sigma_\exists$ pour le point de départ x et l'ensemble cible Q. En particulier, cette stratégie est également gagnante pour l'ensemble cible $Q_x = \{y \in Q \mid x \preccurlyeq y\}$. En effet, soit $\sigma_\forall$ une stratégie universelle, et $\mathcal{F}$ la partie associée. Il existe donc un historique gagnant $H \in \mathcal{F}$. S'il est gagnant pour l'une des conditions (ii) ou (iii), il est aussi gagnant dans le cas de Q_x. Sinon, il est gagnant car $\max H \in Q$. Mais comme $H \in \mathcal{F}$, on a immédiatement que $\min H = x$. En particulier, $x \preccurlyeq \max H$ donc $\max H \in Q_x$. □

Pour vérifier certaines des règles, notamment la règle de continuation, nous avons besoin de remarquer un point important. Étant donné un contexte et un jeu donné, nous pouvons explicitement étendre ce jeu de manière minimale pour satisfaire ce contexte.

Définition 5.9 (Sous-jeu)**.** Un jeu $\mathbb{G}_1$ est dit *sous-jeu* d'un jeu $\mathbb{G}_2$ s'ils ont le même support, et si la relation d'égalité induit une simulation de $\mathbb{G}_1$ par $\mathbb{G}_2$. Cette dernière notion revient à dire que toute garantie vraie dans $\mathbb{G}_1$ est également vraie dans $\mathbb{G}_2$.

Définition 5.10 (Extension minimale pour un contexte donné)**.** Étant donné un jeu $\mathbb{G} = (G, \preccurlyeq, \Delta)$ et Γ un contexte pour le domaine G, nous définissons l'*extension minimale* de $\mathbb{G}$ pour Γ comme le jeu $\mathbb{G}_\Gamma$ suivant :

— le domaine de $\mathbb{G}_\Gamma$ est $(G, \preccurlyeq)$;

— l'ensemble des transitions de $\mathbb{G}_\Gamma$ est donné par

$$\Delta_\Gamma(x) = \Delta(x) \cup \{\{y \in Q \mid x \preccurlyeq y\} \mid x \in P, \langle P \hookrightarrow Q\rangle \in \Gamma\}$$

Lemme 5.11. *Soient $\mathbb{G} = (G, \preccurlyeq, \Delta)$ un jeu, et Γ un contexte pour le domaine G. Vis-a-vis du préordre sous-jeu, l'extension minimale $\mathbb{G}_\Gamma$ de $\mathbb{G}$ pour Γ est le plus petit jeu plus grand que $\mathbb{G}$ qui satisfait Γ.*

Démonstration. En utilisant les lemmes 4.17, 4.20 et 4.18, il est immédiat que $\mathbb{G}_\Gamma$ satisfait Γ. Il est également immédiat que la relation d'égalité induit une simulation étape par étape de $\mathbb{G}$ par $\mathbb{G}_\Gamma$, et donc d'après le théorème 4.41, $\mathbb{G}$ est sous-jeu de $\mathbb{G}_\Gamma$.

Montrons maintenant que tout jeu $\mathbb{G}_2$ dont $\mathbb{G}$ est sous-jeu et qui satisfait Γ admet $\mathbb{G}_\Gamma$ comme sous-jeu. Nous allons pour cela montrer que la relation d'égalité induit une simulation étape par étape de $\mathbb{G}_\Gamma$ par $\mathbb{G}_2$, et donc par le théorème 4.41 une simulation.

Remarquons que dans le cas de la relation d'égalité, la condition limite est systématiquement vérifiée d'après le lemme 4.19. Il nous reste donc à vérifier la condition pour les étapes régulières. Soit donc $x \in G, X \in \Delta_\Gamma(x)$ une transition de $\mathbb{G}_\Gamma$, et vérifions que $\langle\{x\} \hookrightarrow X\rangle_{\mathbb{G}_2}$. Dans le cas où $X \in \Delta(x)$, il suffit de vérifier que $\langle\{x\} \hookrightarrow X\rangle_{\mathbb{G}}$ est vérifiée car $\mathbb{G}$ est sous-jeu de $\mathbb{G}_2$. Or, les lemmes 4.18 et 4.19 permettent de conclure immédiatement que c'est le cas.

Il ne nous reste donc plus que le cas où il existe $\langle P \hookrightarrow Q\rangle \in \Gamma$ tel que $x \in P$ et $X = \{y \in Q \mid x \preccurlyeq y\}$. Mais d'après les règles d'axiome, de progression et de conséquence que nous avons déjà vérifiées,

$$\Gamma \vdash_\preccurlyeq \langle\{x\} \hookrightarrow \{y \in Q \mid x \preccurlyeq y\}\rangle$$

En particulier, comme $\mathbb{G}_2$ satisfait toutes les garanties de Γ, il satisfait la garantie voulue. □

Nous en déduisons immédiatement qu'un contexte est satisfait par un plus petit jeu. Par ailleurs, remarquons que comme la relation sous-jeu est un préordre mais pas un ordre, ce jeu n'est pas forcément unique. Par exemple, la construction donnée dans l'exemple 4.39 permet de construire au moins un jeu équivalent dès que le support du jeu n'est pas vide.

Définition 5.12. Le *jeu vide* sur le support $(G, \preccurlyeq)$ est le jeu défini par l'ensemble de transitions $\Delta_\emptyset(x) = \emptyset$.

Corollaire 5.13. *Soit $(G, \preccurlyeq)$ un ensemble ordonné et Γ un contexte sur le domaine G. Vis-a-vis du préordre sous-jeu, l'extension minimale du jeu vide de support $(G, \preccurlyeq)$ pour Γ est le plus petit jeu qui satisfait Γ.*

Démonstration. Immédiat car à support fixé, le jeu vide est un plus petit élément du préordre sous-jeu. □

Cette construction nous permet notamment de démontrer la règle de continuation, qui est très pratique pour vérifier la validité des dernières règles.

Lemme 5.14. *La règle de continuation est valide.*

Démonstration. Considérons Γ, P, Q tels que $\Gamma, \langle Q \hookrightarrow \emptyset\rangle \vdash_{\preccurlyeq} \langle P \hookrightarrow Q\rangle$. Soit $\mathbb{G}$ un jeu qui satisfait Γ, et $\mathbb{G}_2$ le plus petit jeu qui satisfait $\langle Q \hookrightarrow \emptyset\rangle$ et dont $\mathbb{G}$ est sous-jeu, construit via la méthode du lemme 5.11. Le jeu $\mathbb{G}_2$ vérifie donc toutes les garanties de $\Gamma, \langle Q \hookrightarrow \emptyset\rangle$, et donc la garantie $\langle P \hookrightarrow Q\rangle$. En particulier, d'après le théorème 4.33 il existe un invariant de victoire I pour P, Q dans le jeu $\mathbb{G}'$. Nous pouvons vérifier que I est également un invariant de victoire pour P, Q dans le jeu $\mathbb{G}$. En effet, les conditions (i), (iii) et (iv) ne dépendent pas de l'ensemble de transitions et sont donc immédiatement préservées. La condition (ii) est vérifiée pour $\mathbb{G}$ aussi car pour tout historique jouable $H \in I$ tel que $H \cap Q = \emptyset$, tous les choix de transitions possibles dans $\mathbb{G}_2$ sont hérités de $\mathbb{G}$, car $\max H \notin Q$. □

Une première application de la règle de continuation est de simplifier la preuve de la règle d'itération, en éliminant la post-condition du cœur du raisonnement. Nous récupérerons la règle admettant une post-condition par la suite.

Lemme 5.15. *La règle d'itération est valide dans le cas particulier où $Q = \emptyset$.*

Démonstration. Soient $(O, \leq)$ un ensemble ordonné, $(I_o)_{o\in O}$ une famille de sous-ensembles de G, et Γ un contexte tels que les prémisses de la règle d'itération soient vérifiées. Remarquons que I joue le rôle d'un invariant de boucle, et $(O, \leq)$ de variables auxiliaires utilisées pour la vérification. Montrons que pour $o_1 \in O$, $\Gamma \vdash_{\preccurlyeq} \langle I_{o_1} \hookrightarrow \emptyset\rangle$.

La structure de la règle d'itération ressemble beaucoup à celle du théorème de simulation, et c'est effectivement le résultat que nous allons appliquer pour dériver le résultat. Il semble que le jeu source doive tout simplement correspondre à l'ordre $(O, \leq)$, mais cela ne suffit pas pour plusieurs raisons, notamment l'absence a priori de bornes supérieures pour les chaînes. Nous définissons donc un jeu source $\mathbb{G}_O = (G_O, \preccurlyeq_O, \Delta_O)$ de la manière suivante :

- Le support du jeu est l'ensemble constitué des paires (H, f) où H est une chaîne non vide de $(O, \leq)$, et f une fonction croissante de $(H, \leq)$ dans $(G, \preccurlyeq)$ telle que pour tout $x \in H$, $f(x) \in I_x$. Notons qu'il s'agit d'une certaine classe de fonctions partielles de O dans G. Nous les ordonnons par la variante suivante de l'ordre sur les fonctions partielles : $(H_1, f_1) \preccurlyeq_O (H_2, f_2)$ si et seulement si $H_1 \sqsubseteq H_2$ et si f_1, f_2 coïncident sur l'ensemble H_1.
- L'ensemble de transitions est donné par $\Delta_O(x) = \{\{y \mid x \prec_O y\}\}$.

Comme attendu, les transitions du jeu ne comportent aucune structure particulière. Par contre, la structure du domaine correspond à celle choisie pour les situations limites, permettant ainsi d'enregistrer toutes les étapes de la boucle. Remarquons de plus que toutes les garanties sont valides dans $\mathbb{G}_O$, car l'ensemble de tous les historiques de $\mathbb{G}_O$ constitue un invariant de victoire pour n'importe quelle garantie. En effet, les règles de propagation (i),(iii) et (iv) sont immédiates, et la règle de

propagation (ii) est valide car il existe une transition sans boucle depuis n'importe quel élément.

Dans le but d'établir une simulation, nous définissons également une relation $\mathcal{R}$ entre G_O et G de la manière suivante : $(H, f)\mathcal{R}x$ si et seulement si x est la borne supérieure de $f(H)$. Maintenant, il nous suffit de montrer que pour tout jeu $\mathbb{G}$ satisfaisant Γ, $\mathcal{R}$ induit une simulation de $\mathbb{G}_O$ par $\mathbb{G}$. En effet, d'après le lemme 4.16, il suffit pour démontrer $\langle I_{o_1} \hookrightarrow \emptyset\rangle_{\mathbb{G}}$de vérifier que pour tout $x \in I_{o_1}$, la garantie $\langle\{x\} \hookrightarrow \emptyset\rangle_{\mathbb{G}}$ est vérifiée. Or, cette garantie est l'image par $\mathcal{R}$ de la garantie $\langle\{(\{o_1\}, (\_ \to x))\} \hookrightarrow \emptyset\rangle_{\mathbb{G}_O}$.

Montrons maintenant la simulation. Soit $\mathbb{G}$ un jeu qui satisfait Γ. Par le théorème 4.41, il suffit de vérifier que $\mathcal{R}$ induit une simulation étape par étape de $\mathbb{G}_O$ par $\mathbb{G}$. Commençons par la simulation des étapes régulières. Soient $(H, f) \in G_O$ et $X \in \Delta_O((H, f))$, et vérifions que $\langle\mathcal{R}[(H, f)] \hookrightarrow \mathcal{R}[X]\rangle_{\mathbb{G}}$. Comme $\mathbb{G}$ satisfait le contexte Γ, il suffit naturellement de vérifier l'énoncé qui dit que Γ implique cette garantie. Par la règle de partition, il suffit de vérifier que pour tout x tel que $(H, f)\mathcal{R}x$, l'énoncé $\Gamma \vdash_{\preccurlyeq} \langle\{x\} \hookrightarrow \mathcal{R}[X]\rangle$ est valide.

Il y a deux possibilités. Premièrement, H admet un maximum o. Dans ce cas, $x = f(o) \in I_o$, et donc $\Gamma \vdash_{\preccurlyeq} \langle I_o \hookrightarrow \bigcup_{o'>o} I_{o'}\rangle$. La garantie voulue suit alors immédiatement de la règle de conséquence, et du fait que $\bigcup_{o'>o} I_{o'} \subseteq \mathcal{R}[X]$. En effet, pour tout o' et $y \in I_{o'}$, l'extension de (H, f) définie par :

$$\left(H \cup o', \left(o \to \begin{cases} f(o) & \text{si } o \neq o' \\ y & \text{sinon} \end{cases}\right)\right)$$

est bien un élément de X lié à y par $\mathcal{R}$.

Seconde possibilité, H n'admet pas de maximum. Par définition, (H, f) est alors une situation limite, et donc l'énoncé $\Gamma \vdash_{\preccurlyeq} \left\langle\{x\} \hookrightarrow \bigcup_{o' \geq H} I_{o'}\right\rangle$ est vérifié. Par la même extension que ci-dessus, nous en déduisons l'énoncé voulu.

Enfin, vérifions la simulation des étapes limites. Soient $\mathcal{H}$ un historique de $\mathbb{G}_O$, F une fonction croissante de $\mathcal{H}$ dans $(G, \preccurlyeq)$ telle que pour tout $H \in \mathcal{H}$, $H\mathcal{R}F(H)$, et supposons de plus que $F(\mathcal{H})$ admette une borne supérieure x. Remarquons tout d'abord que $\mathcal{H}$ admet une borne supérieure, définie par :

$$(\overline{H}, \overline{f}) = \left(\bigcup \pi_1[\mathcal{H}], (o \to f(o) \text{ pour un } (H, f) \in \mathcal{H} \text{ tel que } o \in H)\right)$$

Cette définition n'est pas ambigüe car $\mathcal{H}$ est une chaîne, et donc si o appartient au domaine commun de deux éléments de $\mathcal{H}$, les deux fonctions associées à ces éléments renvoient la même valeur pour o. De même, vérifier qu'il s'agit bien de la borne supérieure voulue ne pose aucune difficulté. La garantie que nous devons démontrer est donc $\left\langle\{x\} \hookrightarrow \mathcal{R}[\{\overline{H}, \overline{f}\}]\right\rangle_{\mathbb{G}}$. Il suffit pour cela de remarquer que $(\overline{H}, \overline{f})\mathcal{R}\{x\}$, et donc que le lemme 4.19 s'applique. En effet, remarquons que :

$$\sup \overline{f}(\overline{H}) = \sup_{(H,f)\in\mathcal{H}} \sup f(H) = \sup_{(H,f)\in\mathcal{H}} F((H, f))$$

Cette identité peut être vérifiée par un raisonnement standard, en démontrant que les deux ensembles ont les mêmes majorants. □

Lemme 5.16. *La règle de composition est valide.*

Démonstration. Soient Γ, P, Q, R tels que $\Gamma \vdash_{\preccurlyeq} \langle P \hookrightarrow Q\rangle$ et $\Gamma \vdash_{\preccurlyeq} \langle Q \hookrightarrow R\rangle$, et vérifions que $\Gamma \vdash_{\preccurlyeq} \langle P \hookrightarrow R\rangle$. Par les règles de continuation et de conséquence, il suffit de vérifier que $\Gamma, \langle R \hookrightarrow \emptyset\rangle \vdash_{\preccurlyeq} \langle P \hookrightarrow \emptyset\rangle$. Nous allons pour cela utiliser la règle d'itération pour la post-condition vide, avec la structure suivante :

— L'ensemble ordonné $(\{0;1;2;3\}, \leq)$

— L'invariant $(I_n)_{0\leq n\leq 3}$ défini par $I_0 = P$, $I_1 = Q$, $I_2 = R$, $I_3 = \emptyset$.

Comme l'ensemble ordonné choisi est fini, il n'y a pas de situation limite à traiter. Les hypothèses de propagation pour les valeurs $0, 1$ correspondent alors aux hypothèses modulo l'utilisation de la règle de remplacement pour affaiblir le contexte, tandis que le cas de la valeur 2 correspond à la règle d'axiome et celui de la valeur 3 à la règle d'identité. □

Lemme 5.17. *La règle d'itération est valide.*

Démonstration. Nous allons utiliser les règles de continuation et de composition pour nous ramener au cas où la post-condition est vide. Soient $(O, \leq)$ un ensemble ordonné, $(I_o)_{o\in O}$ un invariant de boucle et Γ, Q tels que les prémisses de la règle d'itération sont vérifiées. Soit $o_1 \in O$. Par les règles de continuation et de conséquence, il suffit de vérifier que $\Gamma' \vdash_{\preccurlyeq} \langle I_{o_1} \hookrightarrow \emptyset\rangle$ pour $\Gamma' = \Gamma, \langle Q \hookrightarrow \emptyset\rangle$. Ce dernier énoncé suit alors de la règle d'itération pour les mêmes ordre et invariant, mais pour le contexte Γ' et une post-condition vide. Les prémisses nécessaires découlent directement de celles déjà connues. En effet, pour tous A, B tels que $\Gamma \vdash_{\preccurlyeq} \langle A \hookrightarrow Q \cup B\rangle$:

$$\dfrac{\dfrac{\Gamma \vdash_{\preccurlyeq} \langle A \hookrightarrow Q \cup B\rangle}{\Gamma' \vdash_{\preccurlyeq} \langle A \hookrightarrow Q \cup B\rangle} \quad \dfrac{\dfrac{\overline{\Gamma' \vdash_{\preccurlyeq} \langle Q \hookrightarrow \emptyset\rangle}}{\Gamma' \vdash_{\preccurlyeq} \langle Q \hookrightarrow B\rangle} \quad \overline{\Gamma' \vdash_{\preccurlyeq} \langle B \hookrightarrow B\rangle}}{\Gamma' \vdash_{\preccurlyeq} \langle Q \cup B \hookrightarrow B\rangle}}{\Gamma' \vdash_{\preccurlyeq} \langle A \hookrightarrow B\rangle}$$

Cette dérivation se réinterprète en termes de programmes comme l'injection d'un appel à la continuation dans toutes les hypothèses. □

5.2.3 Preuve de la règle de récurrence

Pour démontrer la règle de récurrence, nous allons procéder à une version générale de la dérécursification. Autrement dit, nous allons mettre les hypothèses sous une forme où la règle d'itération est applicable. Sans surprise, nous utilisons pour cela un état auxiliaire analogue à une pile d'appels, à ceci près que celle-ci peut devenir de taille infinie au cours de « l'exécution ».

Dans cette sous-section, nous supposons fixés le support $(G, \preccurlyeq)$ du jeu, $(O, \leq)$ un ensemble ordonné, $(P_o, Q_o)_{o\in O}$ une famille de paires pré/post-condition, et Γ un contexte pour le domaine G, tels que les prémisses de la règle de récurrence associée à $(O, \leq, (P_o, Q_o)_{o\in O})$ soient vérifiées. Comme pour la règle d'itération, nous utilisons la règle de continuation pour simplifier le cœur de la preuve. Nous supposons dans un premier temps que la post-condition de l'énoncé que l'on cherche à prouver est vide. Plus précisément, nous fixons le point de départ $o_1 \in O$ de la récurrence, et nous restreignons l'énoncé à prouver $\Gamma \vdash_{\preccurlyeq} \langle P_{o_1} \hookrightarrow Q_{o_1}\rangle$ au cas particulier où $Q_{o_1} = \emptyset$.

Tout d'abord, nous allons caractériser Γ pour un jeu équivalent $\mathbb{G}$.

Lemme 5.18. *Soit $\mathbb{G}$ l'extension minimale du jeu vide pour Γ. Alors pour toute transition $X \in \Delta(x)$ de $\mathbb{G}$, $\Gamma \vdash_{\preccurlyeq} \langle \{x\} \hookrightarrow X \rangle$.*

Démonstration. Le jeu $\mathbb{G}$ est le jeu minimal sur le domaine $(G, \preccurlyeq)$ qui satisfait le contexte Γ. Comme $\mathbb{G}$ satisfait les garanties associées à ses transitions, c'est aussi le cas de n'importe quel jeu dont $\mathbb{G}$ est sous-jeu, et donc de tout jeu satisfaisant Γ. □

Nous fixons ainsi le jeu pour avoir une base sur laquelle procéder à des extensions. En effet, nous remarquons que si une garantie $\langle P_o \hookrightarrow Q_o \rangle$ n'est a priori pas vérifiée dans $\mathbb{G}$, elle l'est néanmoins dans l'extension minimale de $\mathbb{G}$ qui satisfait les hypothèses de récurrence. En particulier, nous pouvons démontrer cette garantie de la manière suivante. Nous suivons l'invariant de victoire dans l'extension de $\mathbb{G}$, et si une transition qui n'existe pas dans $\mathbb{G}$ est nécessaire, nous procédons récursivement au même raisonnement pour simuler cette étape.

Ce procédé nous conduit à l'introduction prévisible de l'équivalent d'une pile d'appels pour savoir quelles étapes sont en cours de simulation, où les différents appels sont a priori indexés par O. Cependant, la possibilité d'un débordement de pile infini existe, et nous conduit à l'introduction d'un appel limite qui n'est pas indexé par O, mais plutôt par une situation limite. Nous utilisons donc une version complétée de $(O, \leq)$ pour indexer les appels récursifs.

Définition 5.19 (Identifiants des appels récursifs)**.** Nous définissons l'ensemble des *identifiants des appels récursifs* comme étant l'ensemble ordonné $(\overline{O}, \sqsubseteq)$ constitué des chaînes non vides de $(O, \leq)$ ordonnées par l'ordre préfixe. Autrement dit, $(\overline{O}, \sqsubseteq)$ est l'ensemble ordonné des « historiques » de $(O, \leq)$.

Pour éviter de devoir gérer les cascades de retours, ainsi que pour gérer uniformément le cas des appels limites et des appels récursifs normaux, nous introduisons une post-condition commune qui autorise un retour direct vers n'importe quel autre appel de la pile.

Définition 5.20 (Post-conditions des appels récursifs)**.** La post-condition de l'appel récursif identifié par $K \in \overline{O}$ est l'ensemble $\overline{Q}_K = \bigcup_{o \in K} Q_o$.

Nous associons alors un jeu particulier à chaque niveau, qui correspond à l'extension de $\mathbb{G}$ dans laquelle nous savons que la garantie du niveau associé est valide.

Définition 5.21 (Jeu associé à un appel récursif)**.** Le jeu $\mathbb{G}_K$ associé à l'appel récursif $K \in \overline{O}$ est l'extension minimale de $\mathbb{G}$ pour le contexte

$$\forall o \in O \text{ majorant strict de } K.\ \langle P_o \hookrightarrow Q_o \rangle .$$

Lemme 5.22. *Pour tout $K \in \overline{O}$,*

- *si K admet un maximum o, alors la garantie $\langle P_o \hookrightarrow Q_K \rangle_{\mathbb{G}_K}$ est vraie.*
- *sinon, pour toute fonction f telle que K, f soit une situation limite, la garantie $\langle \{\sup f(K)\} \hookrightarrow Q_K \rangle_{\mathbb{G}_K}$ est vraie.*

Démonstration. Dans le cas où K admet un maximum o, $\mathbb{G}_K$ vérifie par définition toutes les garanties de $\Gamma, \forall o' > o.\ \langle P_{o'} \hookrightarrow Q_{o'} \rangle$. En particulier, la garantie $\langle P_o \hookrightarrow Q_o \rangle_{\mathbb{G}_K}$ est vraie d'après les prémisses de la règle de récurrence, et par conséquence $\langle P_o \hookrightarrow Q_K \rangle_{\mathbb{G}_K}$ l'est aussi.

Dans le cas où K n'a pas de maximum, tout majorant de K est majorant strict de K, et il suffit donc d'appliquer l'autre prémisse. □

Nous définissons maintenant la structure de pile d'appels que nous employons.

Définition 5.23 (Pile d'appels)**.** Une *pile d'appels* est une paire $\pi = (\mathcal{K}, f)$ d'une chaîne de $\overline{O}$ et d'une fonction de $\mathcal{K}$ vers des triplets $f(K) = (x, H, \mathcal{I})$ formés d'un élément de G, d'un historique de $\mathbb{G}_K$ et d'un invariant de victoire dans $\mathbb{G}_K$. Ces trois éléments sont appelés respectivement le point de départ, l'historique et l'invariant de victoire associés à K.

Une pile d'appels *bien formée* est une pile d'appels vérifiant de plus les propriétés suivantes :

(i) La chaîne $\mathcal{K}$ est non vide et bien fondée.

(ii) La chaîne $\mathcal{K}$ est close par préfixe. Autrement dit, pour tout $K \in \mathcal{K}$, si $K' \sqsubseteq K$ alors $K' \in \mathcal{K}$.

(iii) Pour tout $K_1, K_2 \in \mathcal{K}$ distincts, si $K_1 \sqsubseteq K_2$ alors le point de départ de K_2 majore l'historique de K_1.

(iv) Pour tout $K \in \mathcal{K}$, si $f(K) = (x, H, \mathcal{I})$ alors $\mathcal{I}$ est un invariant de victoire dans $\mathbb{G}_K$ pour $\{x\}, \overline{Q}_K$, et $H \in \mathcal{I}$. De plus, x est le minimum de H, et $H \cap \overline{Q}_K = \emptyset$.

(v) Pour tout $K \in \mathcal{K}$, pour tout $y \in \overline{Q}_K$ qui majore l'historique de K, il existe $K_y \in \mathcal{K}$ préfixe strict de K tel que ajouter y à l'historique de K_y crée un historique appartenant encore à l'invariant de K_y.

(vi) Pour tout $K \in \mathcal{K}$, si K admet un maximum o, alors le point de départ de K appartient à P_o.

Nous notons Π l'ensemble des piles d'appels bien formées.

Les contraintes de bonne formation servent à garantir diverses propriétés pour maintenir correctement l'évolution de la boucle simulant la récurrence. Les propriétés (iv) et (v) sont des propriétés habituelles d'une pile d'appels. En effet, la propriété (iv) exprime le fait que chaque niveau représente une exécution correcte et non terminée, tandis que la propriété (v) correspond au chaînage des retours. La propriété (i) garantit entre autre qu'il existe toujours un niveau minimal vers lequel retourner, coupant court les potentielles cascades de retours. Les trois autres propriétés servent principalement à reconstruire une situation limite correcte pour la récurrence à partir d'un débordement de pile infini. Les contraintes (vi) et (ii) permettent de construire la situation, tandis que la contrainte (iii) garantit la croissance.

Comme Π est l'ensemble que nous allons utiliser pour appliquer la règle d'itération, nous devons également l'équiper d'un ordre correspondant à l'évolution d'une pile d'appels. Il est naturel d'utiliser un ordre lexicographique, mais nous avons également besoin de garantir que la reprise de l'exécution d'un appel inférieur se produit après l'exécution des appels récursifs au-dessus. Nous utilisons donc une variante de l'ordre lexicographique qui prend en compte ces contraintes supplémentaires.

Définition 5.24 (Ordre sur les piles d'appels)**.** Nous définissons une relation $\preccurlyeq_\Pi$ sur les piles d'appels bien formées par $(\mathcal{K}_1, f_1) \preccurlyeq_\Pi (\mathcal{K}_2, f_2)$ si et seulement si l'une des deux conditions suivantes est vérifiée :

— $\mathcal{K}_1 \sqsubseteq \mathcal{K}_2$, et f_1, f_2 coïncident sur $\mathcal{K}_1$

— il existe $\mathcal{K}$ préfixe de $\mathcal{K}_1$ et $\mathcal{K}_2$, admettant un maximum $K = \max \mathcal{K}$, tel que

- f_1 et f_2 coïncident sur $\mathcal{K}\backslash\{K\}$,
- f_1 et f_2 donnent les mêmes points de départ et invariants pour K,
- si H_1 et H_2 sont les historiques renvoyés par f_1 et f_2 en K, alors H_1 est un préfixe strict de H_2,
- pour tout $x \in H_2\backslash H_1$, x est un majorant de tout historique renvoyé par f_1 pour un élément de K_1.

Dans le cas où la première condition est vérifiée, nous disons que la première pile est préfixe de la seconde.

Lemme 5.25. *La relation $\preccurlyeq_\Pi$ est un ordre sur Π.*

Démonstration. La réflexivité de $\preccurlyeq_\Pi$ suit directement de la première condition. Pour l'antisymétrie, soient $(\mathcal{K}_1, f_1)$ et $(\mathcal{K}_2, f_2)$ deux piles d'appels bien formées qui sont liées par $\preccurlyeq_\Pi$ ainsi que par sa relation inverse. Dans le cas où les deux cas sont induits par la première condition définissant $\preccurlyeq_\Pi$, le résultat suit directement de l'antisymétrie de $\sqsubseteq$ et de la coïncidence de f_1 avec f_2. Dans le cas contraire, l'une des deux relations est induite par la seconde condition. Notons que cela implique que les historiques associés par f_1 et f_2 à un certain $K \in \mathcal{K}_1 \cap \mathcal{K}_2$ soient distincts. En particulier, il est impossible qu'une des deux piles soit préfixe de l'autre, ce qui implique que les deux relations sont induites par la seconde condition. Dans ce cas, notons que n'importe quel préfixe $\mathcal{K}$ de $\mathcal{K}_1$ et $\mathcal{K}_2$ qui montre une de ces relation est le plus petit préfixe commun pour lequel f_1 et f_2 ne coïncident pas. En particulier, il est uniquement déterminé. De par les conditions, $f_1(\max \mathcal{K})$ et $f_2(\max \mathcal{K})$ renvoient alors des historiques qui sont préfixes stricts l'un de l'autre, ce qui est impossible.

Nous n'avons plus qu'à montrer la transitivité de $\preccurlyeq_\Pi$, ce qui demande une grande disjonction de cas. Soient donc $\pi_a = (\mathcal{K}_a, f_a)$, $\pi_b = (\mathcal{K}_b, f_b)$ et $\pi_c = (\mathcal{K}_c, f_c)$ trois piles d'appels bien formées telles que $\pi_a \preccurlyeq_\Pi \pi_b \preccurlyeq_\Pi \pi_c$ et montrons que $\pi_a \preccurlyeq_\Pi \pi_c$.

- Si π_a est préfixe de π_b qui est préfixe de π_c, alors π_a est également préfixe de π_c.
- Si π_a n'est pas préfixe de π_b et π_b est préfixe de π_c, alors le préfixe $\mathcal{K}$ qui montre que $\pi_a \preccurlyeq_\Pi \pi_b$ montre immédiatement la seconde condition pour $\pi_a \preccurlyeq_\Pi \pi_c$.
- Si π_a est préfixe de π_b mais que π_b n'est pas préfixe de π_c, alors posons $\mathcal{K}$ le préfixe qui montre que $\pi_b \preccurlyeq_\Pi \pi_c$.
  - Si $\mathcal{K}$ est préfixe de π_a, alors $\mathcal{K}$ montre immédiatement la validité de la seconde condition pour la relation $\pi_a \preccurlyeq \pi_c$.
  - Sinon, π_a est un préfixe de π_c.
- Si π_a n'est pas préfixe de π_b et π_b n'est pas non plus préfixe de π_c, alors posons $\mathcal{K}_1$ et $\mathcal{K}_2$ les préfixes qui montrent respectivement que $\pi_a \preccurlyeq_\Pi \pi_b$ et $\pi_b \preccurlyeq_\Pi \pi_c$. Alors notons que $\mathcal{K}_1$ et $\mathcal{K}_2$ sont comparables car préfixes communs de $\mathcal{K}$. En particulier, le minimum de $\mathcal{K}_1$ et $\mathcal{K}_2$ permet de valider la seconde condition pour la relation $\pi_a \preccurlyeq \pi_c$. La seule condition non triviale est celle de majoration des historiques renvoyés par f_a.
  - Si $\mathcal{K}_1$ est un préfixe strict de $\mathcal{K}_2$, c'est immédiat.
  - Si $\mathcal{K}_1 = \mathcal{K}_2$, soit H_a, H_b, et H_c les historiques renvoyés respectivement par f_a, f_b et f_c en $\max \mathcal{K}_1$. Alors nous devons montrer que les historiques renvoyés par f_a sont majorés par les éléments de $H_c\backslash H_a$. Mais nous savons déjà que ces historiques sont majorés par les éléments de $H_b\backslash H_a$. Comme

$H_a \sqsubseteq H_b \sqsubseteq H_c$, les éléments de $H_c \backslash H_a$ sont soit des éléments de $H_b \backslash H_a$, soit des majorants de $H_b \backslash H_a$. Mais comme $H_b \backslash H_a \neq \emptyset$, les majorants de $H_b \backslash H_a$ sont aussi des majorants des historiques renvoyés par f_a.

— Si $\mathcal{K}_2$ est un préfixe strict de $\mathcal{K}_1$, soit H_a et $H_{b,1}$ les historiques renvoyés respectivement par f_a et f_b en $\max \mathcal{K}_1$, et de même $H_{b,2}$ et H_c les historiques renvoyés respectivement par f_b et f_c en $\max \mathcal{K}_2$. Alors nous devons montrer que les historiques renvoyés par f_a sont majorés par les éléments de $H_c \backslash H_{b,2}$. Mais non savons déjà que ces historiques sont majorés par les éléments de $H_{b,1} \backslash H_a \neq \emptyset$. Or, $H_{b,1}$ est également majoré par les éléments de $H_c \backslash H_{b,2}$, ce qui conclut.

□

Enfin, nous définissons l'invariant de boucle que nous utilisons pour la règle d'itération.

Définition 5.26 (Invariant de dérécursification)**.** L'*invariant de dérécursification* est la famille $(I_\pi)_{\pi \in \Pi}$ définie par $x \in I_{(\mathcal{K},f)}$ si et seulement si $\mathcal{K}$ admet un maximum K, et x est le maximum de l'historique associé à K.

Nous dérivons enfin l'énoncé $\Gamma \vdash_{\preccurlyeq} \langle P_{o_1} \hookrightarrow \emptyset \rangle$ en appliquant la règle d'itération à l'ensemble ordonné $(\Pi, \preccurlyeq_\Pi)$, à l'invariant de boucle $(I_\pi)_{\pi \in \Pi}$, et à une post-condition vide. Vérifions tout d'abord que nous obtenons bien le résultat voulu.

Lemme 5.27. *Supposons que les prémisses de l'instance choisie de la règle d'itération soient vérifiées. Alors* $\Gamma \vdash_{\preccurlyeq} \langle P_{o_1} \hookrightarrow \emptyset \rangle$.

Démonstration. Par la règle de partition, il suffit de vérifier $\Gamma \vdash_{\preccurlyeq} \langle \{x\} \hookrightarrow \emptyset \rangle$ pour tout $x \in P_{o_1}$. Posons pour cela $\mathcal{I}$ un invariant de victoire de $\mathbb{G}_{\{o_1\}}$ pour $\{x\}, \emptyset$, qui existe d'après le lemme 5.22. Alors la pile d'appels

$$\pi = (\{o_1\}, (o \to (x, \{x\}, \mathcal{I})))$$

est bien formée, et $I_\pi = \{x\}$. La garantie voulue suit directement de la règle d'itération. □

Il nous reste maintenant à vérifier que les prémisses de la règle d'itération sont vérifiées. Pour cela, nous employons quelques lemmes auxiliaires visant à effectuer des constructions standards au niveau de la pile, comme l'appel ou le retour. Le premier d'entre eux permet de prolonger arbitrairement l'historique d'un niveau donné sous certaines contraintes.

Lemme 5.28. *Soient* $\pi = (\mathcal{K}, f)$ *une pile d'appels bien formée,* $K_0 \in \mathcal{K}$ *et* $(x, H, \mathcal{I}) = f(K_0)$*. Alors pour tout* $H_2 \in \mathcal{I}$ *dont* H *est préfixe strict, si* $H_2 \cap \overline{Q}_K = \emptyset$*, et si tout élément de* $H_2 \backslash H$ *est un majorant de tous les historiques de* π*, alors la pile d'appels* $\pi' = (\mathcal{K}', f')$ *définie par :*

$$\begin{array}{lcll} \mathcal{K}' & = & \{K \in \mathcal{K} \mid K \sqsubseteq K_0\} & \\ f'(K_0) & = & (x, H_2, \mathcal{I}) & \\ f'(K) & = & f(K) & K \neq K_0 \end{array}$$

est bien formée, et $\pi \prec_\Pi \pi'$.

Démonstration. Il suffit de dérouler les définitions pour vérifier que π' est bien formée. La contrainte d'ordre suit immédiatement par la seconde condition pour le témoin $\mathcal{K}'$, dont le maximum est K_0. □

Nous montrons maintenant un lemme qui permet d'obtenir directement la préservation de l'invariant dans le cas où l'état courant permet d'effectuer un retour.

Lemme 5.29. *Soient $\pi = (\mathcal{K}, f)$ une pile d'appels bien formée et x un majorant de tous les historiques de π. Si $x \in \overline{Q}_K$ pour un élément $K \in \mathcal{K}$, alors il existe une pile d'appels bien formée π' strictement plus grande que π pour l'ordre $\preccurlyeq_\Pi$ telle que $x \in I_{\pi'}$.*

Démonstration. Comme $\mathcal{K}$ est bien fondée, il existe $K_0 \in \mathcal{K}$ minimal tel que $x \in \overline{Q}_{K_0}$. De plus, les retours sont correctement chaînés dans π, donc il existe K_x préfixe strict de K_0 tel que pour $(y, H, \mathcal{I}) = f(K_x)$, $H \cup \{x\} \in \mathcal{I}$. Comme K_1 est strictement plus petit que K_0, $x \notin \overline{Q}_{K_1}$. De plus, $x \notin H$. Dans le cas contraire, il serait plus petit que le point de départ de K_0. Comme on a supposé qu'il devait aussi être plus grand, nous obtenons que l'historique associé à K_0 contient $x \in \overline{Q}_{K_0}$, ce qui est contradictoire. Nous pouvons alors appliquer le lemme 5.28 à $(K_1, H \cup \{x\})$ pour conclure. □

Notons qu'en combinant les deux lemmes précédents, nous pouvons naturellement obtenir la préservation de l'invariant dans le cas de l'ajout correct d'un élément suffisamment grand à n'importe quel niveau.

Lemme 5.30. *Soient $\pi = (\mathcal{K}, f)$ une pile d'appels bien formée, $K \in \mathcal{K}$ et x un majorant de tous les historiques de π. Si $f(K) = (y, H, \mathcal{I})$ et $H \cup \{x\} \in \mathcal{I}$, alors il existe une pile d'appels bien formée π' strictement plus grande que π pour l'ordre $\preccurlyeq_\Pi$ telle que $x \in I_{\pi'}$.*

Démonstration. Il suffit d'appliquer le lemme 5.29 ou le lemme 5.28 selon que $x \in \overline{Q}_K$ ou non. □

Enfin, nous énonçons un dernier lemme permettant d'ajouter un nouvel élément à la pile d'appels.

Lemme 5.31. *Soient $\pi = (\mathcal{K}, f)$ une pile d'appels bien formée, $K_0 \notin \mathcal{K}$ un identifiant dont tous les préfixes stricts appartiennent à $\mathcal{K}$, $x \notin \overline{Q}_{K_0}$ un majorant de tous les historiques de π et $\mathcal{I}$ un invariant de victoire pour $\{x\}, \overline{Q}_{K_0}$ dans $\mathbb{G}_{K_0}$. Supposons de plus que :*

- *si K_0 admet un maximum o, alors $x \in P_o$.*
- *tous les nouveaux retours introduits par K sont correctement chaînés dans $\mathcal{K}$. Autrement dit, pour tout $y \in \overline{Q}_{K_0}$ tel que $x \preccurlyeq y$ et n'appartenant à aucun des $(\overline{Q}_K)_{K \in \mathcal{K}}$, il existe $K_y \in \mathcal{K}$ tel que ajouter y à l'historique de K_y crée un historique appartenant encore à l'invariant de K_y.*

Alors il existe une pile d'appels bien formée π', strictement plus grande que π pour l'ordre $\preccurlyeq_\Pi$, telle que $x \in I_{\pi'}$.

Démonstration. Nous définissons la pile d'appels $\pi' = (\mathcal{K}', f')$ voulue par :

$$\begin{array}{lcll} \mathcal{K}' & = & \mathcal{K} \cup \{K_0\} & \\ f'(K_0) & = & (x, \{x\}, \mathcal{I}) & \\ f'(K) & = & f(K) & K \in \mathcal{K} \end{array}$$

Comme π est préfixe de π', $\pi \preccurlyeq \pi'$. De plus, il suffit de dérouler les définitions pour vérifier que les conditions imposées sur les entrées permettent de transférer les propriétés (ii-vi) et (iv) de bonne formation d'une pile. La contrainte (i) de bonne fondation est immédiate puisque nous ajoutons un unique élément maximal.

Le chaînage correct des retours est un peu moins direct dans le cas de K_0, puisqu'il faut trouver des retours pour tous les éléments y qui appartiennent à une post-condition $\overline{Q}_K$ pour un identifiant $K \in \mathcal{K}$. Nous récupérons pour cela les retours chaînés depuis K, ce qui conclut la preuve de bonne formation de π'. Le fait que $x \in I_{\pi'}$ est immédiat. □

Nous avons maintenant suffisamment d'éléments pour démontrer la prémisse régulière de la règle d'itération.

Lemme 5.32. *La prémisse régulière de l'instance choisie de la règle d'itération est valide. Autrement dit,*

$$\Gamma \vdash_{\preccurlyeq} \left\langle I_\pi \hookrightarrow \bigcup_{\pi' \succ_\Pi \pi} I_{\pi'} \right\rangle$$

Démonstration. Si I_π est vide, l'énoncé est immédiat par la règle d'identité. Supposons maintenant que I_π contient un élément x, nécessairement unique. Soit K l'identifiant maximal de π, et $\mathcal{I}$ l'invariant de victoire associé. Nous allons tout simplement suivre la structure de $\mathcal{I}$. Par la règle de propagation (ii), celui-ci nous fournit une transition X depuis le maximum x de l'historique de π. Il y a maintenant deux possibilités.

Premièrement, cette transition existe dans $\mathbb{G}$. Dans ce cas, le lemme 5.18 montre que la garantie équivalente se déduit de Γ. En appliquant ensuite la règle de conséquence, il suffit de vérifier que X est un sous-ensemble de la post-condition voulue. C'est une conséquence immédiate du lemme 5.30 appliqué à tous les éléments de X.

Deuxième possibilité, cette transition n'existe pas dans $\mathbb{G}$, seulement dans $\mathbb{G}_K$. Alors il existe $o \in O$ majorant strict de K tel que $X = \{y \in Q_o \mid x \preccurlyeq y\}$ et $x \in P_o$. L'idée est alors d'ajouter un niveau d'appels récursif $K \cup \{o\}$ pour simuler cette transition, en appliquant le lemme 5.31. L'invariant de victoire voulu existe bien car la garantie $\langle P_o \hookrightarrow Q_o \rangle_{\mathbb{G}_{K \cup \{o\}}}$ est valide, et les propriétés fournies par la règle de propagation (ii) de $\mathcal{I}$ pour X garantissent le chaînage des nouveaux retours. Le lemme est donc bien applicable, et permet de conclure. □

Pour la prémisse concernant les situations limites, nous classifions ces situations en deux catégories : soit il y a un niveau minimal qui change d'historique indéfiniment, soit la situation correspond à un débordement infini. Pour obtenir cette dichotomie, nous utilisons la notion de préfixe éventuel.

Définition 5.33 (Préfixe éventuel)**.** Soit $\mathcal{J} \subseteq \Pi$ une chaîne non vide de piles d'appels. Une pile d'appels π_0 est un *préfixe éventuel* de $\mathcal{J}$ s'il existe $\pi_1 \in \mathcal{J}$ telle que pour tout $\pi_2 \in J$, si $\pi_1 \preccurlyeq_\Pi \pi_2$ alors π_0 est préfixe de π_2. Autrement dit, il existe un rang dans $\mathcal{J}$ à partir duquel π_0 est préfixe de n'importe quelle pile d'appels.

Lemme 5.34. *Soit $\mathcal{J} \subseteq \Pi$ une chaîne non vide de piles d'appels, $\mathcal{A}$ l'ensemble de ses préfixes éventuels. Alors $\mathcal{A}$ est une chaîne de piles d'appels pour l'ordre préfixe, et $\mathcal{A}$ admet pour cet ordre une borne supérieure définie par :*

$$(\{K \mid K \in \mathcal{K}, (\mathcal{K}, f) \in \mathcal{A}\}, (K \to f(K) \text{ pour un } (\mathcal{K}, f) \in \mathcal{A} \text{ tel que } K \in \mathcal{K}))$$

Démonstration. La borne supérieure étant rigoureusement identique à celle introduite pour démontrer le cœur de la règle d'itération (lemme 5.15), il nous suffit de vérifier que c'est une chaîne. Soient donc π_1, π_2 deux préfixes éventuels. Il existe donc une pile π dans $\mathcal{J}$ au-delà de laquelle π_1 et π_2 sont préfixes de n'importe quelle pile de $\mathcal{J}$. En particulier, π_1 et π_2 sont préfixes d'une pile commune, et sont donc comparables. □

Nous pouvons enfin vérifier le rétablissement de l'invariant de dérécursification à partir d'une situation limite.

Lemme 5.35. *La prémisse limite de l'instance choisie de la règle d'itération est valide. Autrement dit, étant donnée une situation limite $(\mathcal{J}, F)$ et $x = \sup F(\mathcal{J})$,*

$$\Gamma \vdash_{\preccurlyeq} \left\langle \{x\} \hookrightarrow \bigcup_{\pi' \succcurlyeq_\Pi \mathcal{J}} I_{\pi'} \right\rangle$$

Démonstration. Nous allons systématiquement rétablir l'invariant sans effectuer de véritable étape, c'est-à-dire en appliquant la règle d'identité. Il nous suffit donc de vérifier qu'il existe une pile d'appels bien formée π' reliée à x, et majorant $\mathcal{J}$. Pour cela, nous allons faire une disjonction de cas à partir des préfixes éventuels de $\mathcal{J}$. Soit $\overline{\pi} = (\overline{\mathcal{K}}, \overline{f})$ la borne supérieure des préfixes éventuels de $\mathcal{J}$ pour l'ordre préfixe. Deux cas de figure se présentent.

— Si $\overline{\pi}$ est un préfixe éventuel de $\mathcal{J}$, toute pile de $\mathcal{J}$ à partir d'une certaine pile π_0 doit admettre $\overline{\pi}$ comme préfixe strict. Le caractère strict est garanti pas l'absence de maximum dans $\mathcal{J}$: dans le cas où $\overline{\pi}$ appartient à $\mathcal{J}$, nous pourrions prendre π_0 au-delà de $\overline{\pi}$. Nous pouvons donc de manière équivalente nous réduire au cas où toutes les piles de $\mathcal{J}$ satisfont cette propriété, en éliminant les piles plus petites que π_0. En particulier, les piles de $\mathcal{J}$ admettent toutes par bonne fondation un élément minimal K_0 n'appartenant pas à $\overline{\mathcal{K}}$. En utilisant le fait que tous les éléments de $\mathcal{J}$ sont comparables, et en déroulant la définition de $\preccurlyeq_\Pi$, il apparaît que cet élément est toujours le même, et est toujours associé au même point de départ et au même invariant. De plus, comme ajouter un niveau associé à K_0 à $\overline{\pi}$ ne peut pas donner un préfixe éventuel, l'historique associé à K_0 ne peut pas atteindre de maximum le long de la chaîne $\mathcal{J}$.
Posons alors $\mathcal{H}$ l'ensemble des historiques associés à K_0 pour les éléments de $\mathcal{J}$. Par la règle de propagation (iii) des invariants de victoire, $\bigcup \mathcal{H}$ appartient à l'invariant commun associé à K_0. De plus, les contraintes de progression strictes de $\preccurlyeq_\Pi$ montrent que chaque changement de l'historique associé à K_0 ajoute des éléments plus grands que tout ce qui appartient aux piles précédentes. Comme ces changements ne s'arrêtent pas, nous en déduisons que x est également la borne supérieure de $\mathcal{H}$. En particulier, par la règle de propagation (iv), $\bigcup \mathcal{H} \cup \{x\}$ appartient à l'invariant commun associé à

K_0 et admet x comme maximum. L'application du lemme 5.28 au niveau K_0 et à l'historique $\bigcup \mathcal{H}$ crée alors la même pile d'appels bien formée quelle que soit la pile de départ utilisée dans $\mathcal{J}$, et celle-ci majore donc $\mathcal{J}$. Nous obtenons ensuite une pile d'appels bien formée encore plus grande et liée à x par l'application du lemme 5.30 pour simuler la règle de propagation (iv).

— Si $\overline{\pi}$ n'est pas un préfixe éventuel, alors remarquons tout d'abord que $\overline{\pi}$ est non vide. En déroulant les définitions, il est immédiat de vérifier que $\overline{\pi}$ est une pile bien formée. De plus, c'est un majorant de $\mathcal{J}$. En effet, considérons $\pi \in \mathcal{J}$. Comme $\overline{\pi}$ n'est pas un préfixe éventuel, il existe un préfixe π_0 de $\overline{\pi}$ et une pile $\pi_1 \in \mathcal{J}$ plus grande que π dont π_0 n'est pas préfixe. Mais comme π_0 est un préfixe éventuel, il existe aussi une pile $\pi_2 \in \mathcal{J}$ encore plus grande que π_1 dont π_0 est préfixe. La contrainte $\pi_1 \preccurlyeq_\Pi \pi_2$ est donc induite par un préfixe commun $\mathcal{K}$ de leurs chaînes d'identifiants, lequel est nécessairement plus petit que la chaîne de π_0. En particulier, ce témoin montre également que $\pi_1 \preccurlyeq \overline{\pi}$, et par transitivité que $\pi \preccurlyeq \overline{\pi}$.
La pile $\overline{\pi}$ représente alors parfaitement la structure d'un débordement infini. En particulier, elle n'a pas de niveau maximum (sinon elle serait un préfixe éventuel), et n'est donc pas reliée à x. Nous devons donc introduire un niveau de pile supplémentaire relié à x, et pour cela il est naturel d'utiliser la prémisse limite de la règle de récurrence.
Posons $\overline{K} = \bigcup \overline{\mathcal{K}}$, et F' associant à $k \in \overline{K}$ le point de départ associé par $\overline{f}$ à $\{k' \in \overline{K} \mid k' \leq k\}$. Comme les piles sont closes par préfixe, cet ensemble apparaît bien dans $\overline{K}$. Alors (K, F') représente une situation limite pour la règle de récurrence, et grâce à la contrainte de progression stricte de $\preccurlyeq_\Pi$, nous pouvons déduire du fait que $\overline{\pi}$ majore J que x est bien la borne supérieure de $F'(K)$. En particulier, $\mathbb{G}_K$ vérifie les hypothèses récursives du cas limite de la règle de récurrence, et admet donc un invariant de victoire pour $\{x\}, \overline{Q}_K$. Dans le cas où $x \notin \overline{Q}_K$, le lemme 5.31 nous permet de créer un niveau de pile supplémentaire et de conclure car il n'y a pas de nouveau retour à chaîner.
Enfin, dans le cas où $x \in \overline{Q}_K$, nous avons également $x \in \overline{Q}_{K_0}$ pour un certain $K_0 \in \overline{\mathcal{K}}$. En particulier, le lemme 5.29 nous permet de simuler un retour et de conclure.

□

Le raisonnement effectué jusqu'ici démontre la validité de la règle de récurrence dans le cas particulier où la post-condition finale est vide. Nous montrons maintenant que nous pouvons en déduire la règle de récurrence générale par le biais d'autres règles, notamment la règle de continuation.

Lemme 5.36. *La règle de récurrence est valide.*

Démonstration. Nous nous ramenons au cas précédent en posant le contexte $\Gamma' = \Gamma, \langle Q_{o_1} \hookrightarrow \emptyset \rangle$ et la famille de post-condition Q' définie par $Q'_o = Q_o$ si $o \neq o_1$, $Q'_{o_1} = \emptyset$. Par les règles de continuation et de conséquence, il suffit effectivement de vérifier $\Gamma' \vdash_\preccurlyeq \langle P_{o_1} \hookrightarrow Q'_{o_1} \rangle$. Nous obtenons les bonnes prémisses de la règle de récurrence restreinte en employant une combinaison des autres règles :

— Pour établir une prémisse dont la post-condition finale est basée sur Q' plutôt que sur Q, nous l'établissons d'abord pour Q, et utilisons la règle de

composition couplée aux règles de partition et d'axiome pour éliminer toute occurrence de Q_{o_1}. Cela revient à injecter un appel à la continuation finale.
— Pour réduire les hypothèses récursives à des hypothèses basées sur Q' plutôt que sur Q, nous prenons les hypothèses basées sur Q', et par la règle de remplacement couplée à la règle de conséquence, nous remplaçons les occurrences de $Q'_{o_1} = \emptyset$ par Q_{o_1}. Cela revient à affaiblir les hypothèses.

□

5.2.4 Complétude du système de raisonnement

Maintenant que nous avons établi la validité des règles de raisonnement, nous établissons que ces règles de raisonnement sont complètes, c'est-à-dire que n'importe quel énoncé peut être établi en appliquant ces règles. En fait, nous établissons la complétude pour un sous-ensemble restreint de ces règles, où la seule règle non triviale est celle d'itération. De plus, nous n'avons besoin d'appliquer celle-ci qu'une seule fois et en dernier. De ce fait, le théorème de complétude est assez proche du résultat énonçant que tout programme peut être réécrit sous la forme d'une unique boucle.

Théorème 5.37. *Les règles données pour les énoncés sont complètes. Plus précisément, en utilisant uniquement les règles d'axiome, d'identité, de progression, de conséquence et d'itération, il est possible de dériver n'importe quel énoncé valide. De plus, nous pouvons garantir que la règle d'itération n'est utilisée qu'une seule fois, et en tout dernier.*

Démonstration. Soit $\Gamma \vdash_{\preccurlyeq} \langle P \hookrightarrow Q\rangle$ un énoncé valide. Remarquons tout d'abord que d'après le lemme 5.13, l'extension minimale $\mathbb{G} = (G, \preccurlyeq, \Delta)$ du jeu vide pour le support $(G, \preccurlyeq)$ qui satisfait Γ est le plus petit jeu à satisfaire Γ (pour l'ordre sous-jeu). En particulier, comme l'énoncé est valide, la garantie $\langle P \hookrightarrow Q\rangle_{\mathbb{G}}$ est valide. D'après le lemme 4.36, $\mathbb{G}$ admet un invariant de victoire normalisé J pour P, Q.

Nous allons construire une dérivation de $\Gamma \vdash_{\preccurlyeq} \langle P \hookrightarrow Q\rangle$ via les cinq règles autorisées en utilisant la règle d'itération avec un invariant de boucle basé sur J. Plus précisément, nous utilisons la structure suivante :
— l'ensemble ordonné $(J \cup \{\emptyset\}, \sqsubseteq)$, qui adjoint à J un élément minimal $\emptyset$ pour l'ordre préfixe.
— l'invariant de boucle $(I_H)_{H \in J \cup \{\emptyset\}}$ défini par $I_\emptyset = P$, et pour $H \in J$, $x \in I_H$ si et seulement si x est le maximum de H.

Si les prémisses de la règle d'itération sont valides pour cette structure et la post-condition Q, nous obtenons immédiatement l'énoncé voulu en partant de l'ensemble vide. Il nous reste donc à vérifier lesdites prémisses. Commençons par le cas régulier. Étant donné $H \in J$, nous devons dériver que :

$$G \vdash_{\preccurlyeq} \left\langle I_H \hookrightarrow Q \cup \bigcup_{H' \in J \cup \{\emptyset\}, H \sqsubset H'} I_{H'} \right\rangle$$

Commençons par les cas particuliers. Tout d'abord, si I_H est vide, la règle d'identité conclut. Ensuite, si H est vide, nous pouvons également appliquer la règle d'identité car par la règle (i) de propagation des invariants de victoire, $P \subseteq \bigcup_{x \in P} I_{\{x\}}$, ensemble qui est lui-même un sous-ensemble de la post-condition voulue. Enfin, si

$H \cap Q \neq \emptyset$, par normalisation de J, le maximum de H existe et appartient à Q, donc la règle d'identité permet encore de conclure.

Nous pouvons maintenant supposer que nous sommes dans le cas où $H \cap Q = \emptyset$ et où il existe $x \in I_H$, et donc que H est jouable. La règle de propagation (ii) des invariants de victoire nous fournit un ensemble $X \in \Delta(x)$ tel que $x \notin X$, et tel que pour tout $y \in X$ on ait $H \cup \{y\} \in J$. En particulier, par application de la règle de conséquence il suffit de vérifier que $\Gamma \vdash_{\preccurlyeq} \langle \{x\} \hookrightarrow X \rangle$ est dérivable car pour tout $y \in X$, on a $H \sqsubset H \cup \{y\}$ et $y \in I_{H \cup \{y\}}$. Mais comme nous avons construit $\mathbb{G}$ comme extension minimale du jeu vide, il existe $\langle P \hookrightarrow Q \rangle \in \Gamma$ tel que $x \in P$ et $X = \{y \mid x \preccurlyeq y\}$. Une application directe des règles d'axiome, de conséquence et de progression permet alors de dériver l'énoncé voulu.

Vérifions maintenant la prémisse correspondant aux situations limites. Soit $(\mathcal{H}, f)$ une situation limite. Par la règle de propagation (iii) des invariants de victoire, $\bigcup \mathcal{H} \in J$. Il est par ailleurs direct de vérifier que $\sup f(\mathcal{H}) = \sup \bigcup \mathcal{H}$ par un raisonnement classique d'interversion des bornes supérieures. En particulier, par la règle (iv) de propagation des invariants de victoire, l'historique $H_0 = \bigcup \mathcal{H} \cup \sup f(\mathcal{H})$ est un élément de J. Comme H_0 admet de plus $\sup f(\mathcal{H})$ comme maximum, et que H_0 est un majorant de $\mathcal{H}$, la règle d'identité permet de conclure que :

$$\Gamma \vdash_{\preccurlyeq} \left\langle \{\sup f(\mathcal{H})\} \hookrightarrow Q \cup \bigcup_{H \in J \text{ tel que } \forall H' \in \mathcal{H}.\ H' \sqsubseteq H} I_H \right\rangle$$

□

5.2.5 Formalisation en Why3

Comme annoncé à la fin de la section 4.6, la logique présentée est formalisée en Why3, et les règles présentées sont démontrées correctes. La formalisation Why3 et la formalisation présentée dans cette section diffèrent toutefois sur un point notable, à savoir la formalisation des énoncés et des contextes. Nous avons formalisé ceux-ci en Why3 par le biais d'un plongement léger d'une logique intuitionniste où les garanties sont des formules, avec une interprétation de Kripke. Nous définissons alors simplement les contextes comme des formules de cette logique, et les énoncés comme des paires de formules. Nous avons initialement fait le choix d'une logique intuitionniste en visant l'objectif d'une logique d'ordre supérieur, et en particulier d'un principe de récurrence d'ordre supérieur, mais cet objectif n'a pas été atteint.

Plus précisément, nous définissons le plongement léger de la logique intuitionniste via le modèle de Kripke constitué des jeux sur un domaine, ordonné par le préordre sous-jeu. Dans ce cadre, nous définissons donc le plongement léger d'une formule par une fonction croissante des jeux vers les booléens. Nous définissons alors l'interprétation des garanties, en tant que formules de la logique, par les garanties sur les jeux.

```
type fmla 'a = { interp : game 'a → bool }
  invariant { ∀g1 g2. interp g1 ∧ subgame g1 g2 → interp g2 }
  by { interp = λ_. false } (* témoin d'habitation *)
let ghost function enforce (p q:set 'a) : fmla 'a =
  { interp = λg. have_uniform_winning_strat g p q }
```

Ce choix change peu la démonstration des diverses règles, car la principale propriété que nous exploitons sur un contexte arbitraire est précisément la croissance. La règle de récurrence est le seul cas impacté car nous exploitons une propriété additionnelle qui est ici perdue, à savoir l'existence d'un plus petit jeu satisfaisant un contexte Γ de forme arbitraire. Cela vient du fait que les contextes ne sont pas nécessairement des conjonctions de garanties. Pour la règle de récurrence, nous nous réduisons donc au cas où le minimum existe. Nous remarquons que pour tout jeu $\mathbb{G}$ satisfaisant Γ, le raisonnement prévu s'applique pour le contexte constitué des garanties valides dans $\mathbb{G}$, dont $\mathbb{G}$ est un minimum évident. En particulier, la garantie que l'on cherche à démontrer pour $\mathbb{G}$ est bien valide.

Par contre, nous perdons la propriété de complétude de la logique, à cause de la présence potentielle de formules d'implication dans le contexte. En effet, l'interprétation choisie permet de transformer les implications en disjonction d'une collection de garanties. Il s'agit d'une certaine manière du phénomène que nous exploitons pour démontrer la règle de récurrence. Mais sous cette interprétation, on peut également déduire d'une implication de garantie $\langle P \hookrightarrow Q\rangle \Rightarrow \langle R \hookrightarrow \emptyset\rangle$ la formule

$$\langle R \hookrightarrow \emptyset\rangle \vee (\langle R \hookrightarrow Q\rangle \wedge \langle Q \hookrightarrow \emptyset\rangle),$$

ce que les règles d'inférence proposées dans cette section ne permettent pas de démontrer. Nous estimons cependant que la propriété de complétude n'est pas un élément critique de notre formalisation mécanisée, car son absence ne fait aucunement barrage à l'utilisation du système logique.

5.3 Transformateurs de post-conditions

Comme nous l'avions déjà remarqué pour la vérification du compilateur de Imp (cf. section 2.5.7), l'utilisation directe d'une logique à la Hoare est loin d'être une solution idéale pour vérifier un programme. En effet, cela nous impose d'utiliser explicitement la règle de conséquence un grand nombre de fois. Notre système de règles d'inférence pour les garanties conserve évidemment cette propriété peu pratique. Nous proposons donc un système de déduction pragmatique pour l'établissement de garanties, basé comme auparavant sur la combinaison de *transformateurs de prédicats en arrière*. Ce système suit l'esprit de celui que nous avons utilisé pour établir la correction du petit compilateur, et permet d'effectuer une partie du travail de preuve par un simple calcul.

Pour pouvoir écrire facilement des transformateurs en pratique, nous proposons également une notation suivant la syntaxe d'un programme annoté pour le système de combinaison. De ce fait, le fait de donner la structure d'une combinaison de transformateurs peut être assimilée à l'explicitation d'une simulation entre un programme structuré et le comportement du jeu sous-jacent. Cela permet également de voir le transformateur résultant comme la plus faible pré-condition du « programme » ainsi écrit, et l'énoncé à établir comme le contrat voulu pour ce programme.

Comme les règles de construction présentées ne dépendent pas du support du jeu, nous supposons dans cette section que le support du jeu $(G, \preccurlyeq)$ est fixé.

5.3.1 Énoncés étiquetés et transformateurs

Rappelons tout d'abord le principe d'un système de preuve basé sur les transformateurs. Les objets que l'on cherche à obtenir sont des groupes de garanties, organisées sous une forme analogue à des contrats de fonction. Pour vérifier un nouveau « contrat », nous procédons de la manière suivante. Tout d'abord, nous convertissons les « contrats » déjà connus sous la forme d'objets fonctionnels appelés *transformateurs de post-conditions*. Nous composons ensuite ces transformateurs à l'aide d'un certain nombre de règles de combinaison, qui doivent correspondre au squelette de preuve attendu avec des garanties. Nous obtenons ainsi un transformateur final, que nous « appliquons » au contrat que nous cherchons à établir. L'application du transformateur au contrat voulu est une propriété, que l'on peut calculer explicitement en déroulant les définitions des transformateurs. Il s'agit typiquement d'une formule de structure conjonctive, avec des implications et quantifications externes pour partager certaines prémisses. Si ladite propriété est vraie, alors le contrat est établi.

Nous devons faire quelques ajustements vis-à-vis du système proposé pour la preuve du petit compilateur. La première différence majeure est la gestion d'un contexte, qui doit être organisé sous une forme exploitable. Celui-ci contient notamment toutes les hypothèses qui définissent le jeu voulu, mais peut également être étendu pour accommoder des hypothèses récursives ou des continuations pour faciliter la preuve. Nous pouvons alors voir le contexte comme une réserve de fonctions déjà vérifiées, et l'utilisation d'une hypothèse comme un appel de fonction. Nous avons donc choisi de représenter les hypothèses sous une forme analogue à celle d'un contrat de procédure, plus pratique à exploiter qu'un ensemble de garanties non structuré.

Définition 5.38 (Contrat)**.** Un *contrat* est un quadruplet

$$(A, B, (P_a)_{a\in A}, (Q_{a,x,b})_{(a,x,b)\in A\times G\times B})$$

formé d'un ensemble de *paramètres auxiliaires* A, d'un ensemble de *valeurs de retour auxiliaires* B, d'une pré-condition $(P_a)_{a\in A}$ paramétrée par les paramètres auxiliaires, et d'une post-condition $(Q_{a,x,b})_{(a,x,b)\in A\times G\times B}$ paramétrée par les valeurs d'entrée (a, x) et par les valeurs de retour auxiliaires.

Le contexte correspondant à un contrat est alors donné par

$$\forall a \in A, x \in P_a.\ \left\langle \{x\} \hookrightarrow \bigcup_{b\in B} Q_{a,x,b} \right\rangle$$

Les paramètres et valeurs de retour auxiliaires d'un contrat jouent un rôle analogue à celui des valeurs fantômes dans les outils de vérification. Nous avions déjà observé la nécessité de paramètres auxiliaires pour vérifier le petit compilateur (cf. section 2.5.8), et réutilisons donc naturellement cet ingrédient. L'ajout de valeurs de retour auxiliaires est une extension naturelle de cette approche. Remarquons que considérer ces valeurs n'est pas une nécessité logique, puisque nous pouvons toujours les représenter par un quantificateur existentiel dans la post-condition. Cependant, cela nous permet d'éliminer la plupart des quantifications existentielles des conditions de vérification finales, et donc de faciliter l'exploitation de ces valeurs.

Nous organisons maintenant le contexte en nommant les différentes hypothèses, de manière à pouvoir identifier les hypothèses et donc y faire référence.

Définition 5.39 (Contexte étiqueté)**.** Supposons fixé un ensemble infini de noms $\mathcal{N}$. Un *contexte étiqueté* est une fonction à support fini de $\mathcal{N}$ dans les contrats. Le contexte non étiqueté correspondant à un contexte étiqueté Γ est alors l'union des contextes correspondant à chacun des contrats de l'image de Γ.

Comme ce n'est pas ambigu, nous utilisons directement la notation Γ pour représenter le contexte non étiqueté correspondant à un contexte étiqueté Γ dans la partie gauche d'un énoncé $\Gamma \vdash_{\preccurlyeq} \langle P \hookrightarrow Q \rangle$.

Les énoncés que nous cherchons à démontrer peuvent être vus comme des contrats de procédures, tout comme les hypothèses. Nous organisons donc aussi les énoncés sous une telle forme.

Définition 5.40 (Énoncé étiqueté)**.** Un *énoncé étiqueté* est une paire

$$\left(\begin{array}{l} (\Gamma_{a,x})_{(a,x)\in A\times G}, \\ (A, B, (P_a)_{a\in A}, (Q_{a,x,b})_{(a,x,b)\in A\times G\times B}) \end{array} \right)$$

formée d'une famille de contextes étiquetés $(\Gamma_{a,x})_{(a,x)\in A\times G}$ paramétrée par les valeurs d'entrée d'un contrat, ainsi que du contrat correspondant. Un énoncé étiqueté est dit *valide* si pour toute paire $(a, x) \in A \times P_a$, l'énoncé (non étiqueté) suivant est valide :

$$\Gamma_{a,x} \vdash_{\preccurlyeq} \left\langle \{x\} \hookrightarrow \bigcup_{b\in B} Q_{a,x,b} \right\rangle$$

Dans le cas particulier où les éléments du contrat ne dépendent pas des paramètres, nous retrouvons la notion d'énoncé précédemment définie :

Lemme 5.41. *Supposons que les ensembles A, B sont non vides. Alors l'énoncé étiqueté*

$$((\Gamma)_{(a,x)\in A\times G}, (A, B, (P)_{a\in A}, (Q)_{(a,x,b)\in A\times G\times B}))$$

est valide si et seulement si l'énoncé (non étiqueté) $\Gamma \vdash_{\preccurlyeq} \langle P \hookrightarrow Q \rangle$ est valide.

Démonstration. Conséquence immédiate des règles de partition et de conséquence. □

Exemple 5.42. Considérons le jeu $\overline{\mathbb{N}}_\times$ suivant sur l'ensemble $\overline{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$ muni de l'ordre habituel : chacun leur tour, les joueurs peuvent multiplier l'état par un entier non nul. L'ensemble des transitions Δ correspondant à ce comportement est donné par :

$$\Delta(n) = \{\{k \times l \times n \mid l \in \mathbb{N}\backslash\{0\}\} \mid k \in \mathbb{N}\backslash\{0\}\}$$

Dans ce jeu, pour n'importe quel $N \in \mathbb{N}$ et n'importe quel point de départ entier, le joueur existentiel peut garantir l'atteinte d'un état entier divisible par les N premiers entiers. Entre autres stratégies, ce joueur peut prendre la transition correspondant à la multiplication par $N!$, ou effectuer N multiplications successives par les premiers entiers. Nous pouvons représenter ce résultat par l'énoncé étiqueté formé

— du contexte étiqueté

$$[\texttt{mult} : (\mathbb{N}\backslash\{0\}, \mathbb{N}\backslash\{0\}, (\mathbb{N})_{k\in\mathbb{N}\backslash\{0\}}, (\{k \times l \times n_0\})_{(k,n_0,l)\in(\mathbb{N}\backslash\{0\})\times\overline{\mathbb{N}}\times(\mathbb{N}\backslash\{0\})})]$$

ici indépendant des paramètres du contrat. Celui-ci représente un contexte formé d'une hypothèse nommée $\texttt{mult}$, associé à un contrat représentant la structure d'une transition. Le paramètre d'entrée correspond à la valeur choisie par le joueur existentiel, et le paramètre de sortie à la valeur choisie par le joueur universel.

— du contrat

$$(\mathbb{N}, \{0\}, (\mathbb{N})_{N\in\mathbb{N}}, (\{n_1 \in \mathbb{N} \mid \forall k \in \mathbb{N}.1 \leq k \leq N \Rightarrow k|n_1\})_{(N,n_0,0)\in\mathbb{N}\times\overline{\mathbb{N}}\times\{0\}})$$

correspondant au résultat d'accessibilité voulu. Le paramètre d'entrée correspond à l'entier l choisi, et il n'y a pas besoin de valeur de retour pour caractériser le résultat.

Le fait que le contexte étiqueté choisi soit vérifié par $\overline{\mathbb{N}}_\times$ est une conséquence directe du lemme 4.16. Par un argument de simulation, nous pouvons également démontrer que tout jeu vérifiant ce contexte admet $\overline{\mathbb{N}}_\times$ comme sous-jeu, ce qui signifie que ce contexte étiqueté représente parfaitement $\overline{\mathbb{N}}_\times$. Cet énoncé étiqueté reflète donc bien la garantie voulue.

Nous pourrions bien entendu donner d'autres énoncés étiquetés équivalents pour cette propriété. Par exemple, nous avons la possibilité d'éliminer la valeur de retour du contrat associé à $\texttt{mult}$, en cachant cette valeur dans une propriété de divisibilité.

Nous introduisons maintenant une notion de transformateur de post-conditions correspondant aux énoncés étiquetés. Ces transformateurs servent à calculer une pré-condition correcte à partir d'une post-condition.

Définition 5.43 (Transformateur de post-conditions)**.** Un *transformateur de post-conditions* de type A, B est une fonction $W^{A,B}$ telle que :

— l'ensemble d'entrée de $W^{A,B}$ est l'ensemble des triplets $(\Gamma, (Q_b)_{b\in B}, a)$ formé d'un contexte étiqueté Γ, d'une famille $(Q_b)_{b\in B}$ de sous-ensembles de G, et d'un élément a de A.

— l'ensemble de sortie de $W^{A,B}$ est $\mathcal{P}(G)$.

— pour toute valeur $(\Gamma, (Q_b)_{b\in B}, a)$ dans le domaine $W^{A,B}$, l'énoncé (non étiqueté)

$$\Gamma \vdash_{\preccurlyeq} \left\langle W^{A,B}(\Gamma, (Q_b)_{b\in B}, a) \hookrightarrow \bigcup_{b\in B} Q_b \right\rangle$$

est valide.

Notre choix d'ajouter le contexte aux paramètres du transformateur de post-conditions plutôt que de l'ajouter à son type peut de prime abord paraître surprenant, puisque le contexte sert principalement à référencer les hypothèses possibles. Il s'agit cependant d'une nécessité pour traiter correctement certaines constructions, notamment la prise d'une continuation. En effet, dans le cas de cette dernière nous devons ajouter au contexte un élément qui dépend de la post-condition passé en paramètre.

Nous montrons maintenant comment établir la validité d'un énoncé étiqueté à partir d'un transformateur de post-conditions, et réciproquement comment dériver

un transformateur d'un énoncé étiqueté pré-établi. La méthode de preuve basée sur les transformateurs est alors la suivante. Premièrement, nous dérivons des transformateurs à partir d'énoncés étiquetés connus. Ensuite, nous construisons un nouveau transformateur en combinant les transformateurs existants par des règles de combinaison. Enfin, nous appliquons le transformateur ainsi obtenu à l'énoncé étiqueté que l'on cherche à prouver, ce qui génère une condition. Il suffit ensuite de vérifier ladite condition pour prouver que l'énoncé étiqueté est correct.

La condition obtenue peut être mise sous une forme purement logique en déroulant mécaniquement les définitions des transformateurs, qui disparaissent alors complètement. La contrainte à laquelle ceux-ci cèdent la place suit une structure similaire à celle des plus faibles pré-conditions et hérite la structure conjonctive de celles-ci, sauf en cas d'utilisation du transformateur de meilleur choix. À cause de cette similarité avec les plus faibles pré-conditions, nous appelons *condition de vérification* la propriété résultant de l'application d'un transformateur de post-conditions à un énoncé étiqueté.

Lemme 5.44. *Soient* $((\Gamma_{a,x}), (A, B, (P_a), (Q_{a,x,b})))$ *un énoncé étiqueté, et* $W^{A,B}$ *un transformateur de post-conditions de type* A, B*. Si*

$$\forall a \in A, x \in P_a.\ x \in W^{A,B}(\Gamma_{a,x}, (Q_{a,x,b})_{b\in B}, a)$$

*alors l'énoncé étiqueté est valide. La propriété ci-dessus est appelé l'*application *du transformateur* $W^{A,B}$ *à l'énoncé étiqueté donné.*

Démonstration. Application immédiate de la règle de conséquence. □

Comme nous voulons également être capables de réutiliser des énoncés déjà démontrés pour la preuve d'énoncés subséquents, il nous faut un mécanisme permettant de repasser des énoncés étiquetés aux transformateurs. Nous introduisons donc un mécanisme de conversion des énoncés étiquetés valides en transformateurs. Le transformateur résultant de la conversion décrit en fait les conséquences immédiates de l'énoncé. Cela nous amène à comparer les contextes, et donc à des conditions d'inclusion de contexte.

Définition 5.45. Un contexte étiqueté Γ_1 est dit sous-contexte d'un contexte Γ_2, noté $\Gamma_1 \subseteq \Gamma_2$, si pour tout nom H dans le domaine de Γ_1, alors

(i) le nom H est dans le domaine de Γ_2.

(ii) les contrats $\Gamma_1(H)$ et $\Gamma_2(H)$ ont les mêmes paramètres auxiliaires.

(iii) en posant $(A, B, (P_a), (Q_{a,x,b}))$ et $(A, B, (P'_a), (Q'_{a,x,b}))$ les contrats associés à H dans Γ_1 et Γ_2 respectivement, l'inclusion

$$P_a \subseteq \{x \in P'_a \mid \forall b \in B.\ Q'_{a,x,b} \subseteq Q_{a,x,b}\}$$

est vérifiée.

Lemme 5.46. *Si* $\Gamma_1 \subseteq \Gamma_2$*, alors pour tout énoncé valide* $\Gamma_1 \vdash_{\preccurlyeq} \langle P \hookrightarrow Q\rangle$*, l'énoncé* $\Gamma_2 \vdash_{\preccurlyeq} \langle P \hookrightarrow Q\rangle$ *est également valide.*

Démonstration. Il s'agit d'une instance directe de la règle de remplacement. Soit donc $\langle P \hookrightarrow Q\rangle$ une garantie qui appartient au contexte non étiqueté associé à Γ_1, et montrons qu'elle est déductible de Γ_2. Par appartenance à Γ_1, il existe un

contrat $(A, B, (P_a), (Q_{a,x,b})$ associé à un nom $\texttt{H}$ dans Γ_1 ainsi que deux éléments $(a, x) \in A \times P_a$ tels que $P = \{x\}$ et $Q = \bigcup_{b \in B} Q_{a,x,b}$. Comme $\Gamma_1 \subseteq \Gamma_2$, nous en déduisons que $\texttt{H}$ appartient aussi au domaine de Γ_2, et donc que $\Gamma_2(\texttt{H})$ peut s'écrire sous la forme $(A, B, (P'_a), (Q'_{a,x,b}))$. En particulier, $x \in P'_a$, et donc Γ_2 contient la garantie $\left\langle \{x\} \hookrightarrow \bigcup_{b \in B} Q'_{a,x,b} \right\rangle$. Le résultat voulu suit alors de l'application de la règle d'axiome puis de celle de conséquence. □

Lemme 5.47. *Soit* $((\Gamma_{a,x}), (A, B, (P_a), (Q_{a,x,b})))$ *un énoncé étiqueté valide. Alors la fonction* $W^{A,B}$ *définie par*

$$W^{A,B}(\Gamma, (Q'_b)_{b \in B}, a) = \{x \in P_a \mid \Gamma_{a,x} \subseteq \Gamma \wedge \forall b \in B.\ Q_{a,x,b} \subseteq Q'_b\}$$

est un transformateur de post-conditions de type A, B*, appelé la* conversion *de l'énoncé étiqueté.*

Démonstration. Soit $\Gamma, (Q'_b)_{b \in B}, a$ une valeur d'entrée quelconque de $W^{A,B}$. En utilisant la règle de partition pour distribuer sur tout l'ensemble, il suffit de vérifier que pour tout $x \in W^{A,B}(\Gamma, (Q'_b)_{b \in B}, a)$ l'énoncé $\Gamma \vdash_{\preccurlyeq} \langle \{x\} \hookrightarrow Q'_b \rangle$ est valide. Mais par validité de l'énoncé étiqueté, l'énoncé $\Gamma_{a,x} \vdash_{\preccurlyeq} \langle \{x\} \hookrightarrow Q_{a,x,b} \rangle$ est alors valide. En appliquant le lemme 5.46 puis la règle de conséquence, nous obtenons l'énoncé voulu. □

5.3.2 Dénomination des paramètres et premières notations

Pour pouvoir disposer de valeurs auxiliaires dans les transformateurs, les contrats, et les énoncés, nous avons introduit deux ensembles, correspondant respectivement à des paramètres auxiliaires et aux valeurs de retours auxiliaires. Ces deux ensembles sont typiquement des ensembles produits, dont chaque composante représente l'ensemble des valeurs possibles pour un paramètre. Cependant, cette structure n'est pas représentée par les énoncés étiquetés et les transformateurs. Dans le but d'obtenir un système de combinaison simple à utiliser, nous avons besoin d'introduire une couche additionnelle correspondant à la gestion de ces paramètres. En particulier, nous avons besoin de pouvoir différencier facilement ces paramètres, ce qui nous conduit à les nommer.

Nous représentons donc la décomposition des ensembles de valeurs auxiliaires en paramètres individuels par le biais d'association de noms.

Définition 5.48 (Association de noms)**.** Une *association de noms* est une fonction f à support fini de l'ensemble des noms vers des ensembles. L'ensemble des valeurs auxiliaires associé est l'ensemble produit des ensembles images de la fonction. Intuitivement, une association de noms correspond au typage des paramètres d'un contrat/transformateur.

Nous avons de même besoin de pouvoir différencier les paramètres au niveau des hypothèses fournies par les contextes étiquetés. En d'autre termes, nous avons besoin de la signature du contexte.

Définition 5.49 (Signature)**.** La *signature* d'un contexte est une fonction à domaine fini de l'ensemble des noms d'hypothèses vers des couples d'association de noms.

Nous généralisons donc directement les notions de contexte/énoncé étiqueté et de transformateur de post-conditions en leur ajoutant cette structure nommée.

Définition 5.50 (Contrat nommé)**.** Un *contrat nommé* est un contrat muni de deux associations de noms, telles que les ensembles de valeurs auxiliaires associés à ces deux associations soient respectivement les ensembles de paramètres auxiliaires et de valeurs de retour auxiliaires du contrat.

Définition 5.51 (Contexte nommé)**.** Un *contexte nommé* est un contexte étiqueté muni d'une signature, tel que
- le support du contexte et de la signature coïncident,
- pour toute hypothèse `H` appartenant au support du contexte, le contrat et les associations de noms correspondant à `H` forment une structure de contrat nommé.

Définition 5.52 (Énoncé nommé)**.** Un *énoncé nommé* est un énoncé étiqueté dont les éléments sous-jacents, c'est-à-dire un contexte étiqueté et un contrat sous-jacent, sont tous deux muni d'une structure nommée.

Définition 5.53 (Transformateur nommé)**.** Un *transformateur nommé* est un transformateur de post-conditions muni de la même structure additionnelle qu'un énoncé nommé, c'est-à-dire d'une signature et de deux associations de noms. Les deux associations de noms doivent correspondre aux deux ensembles de valeurs auxiliaires du contrat, et le transformateur ne peut être appliqué correctement qu'aux contextes étiquetés qui correspondent à la signature.

Nous pouvons directement donner des notations pragmatiques pour les contrats et énoncés nommés. Pour représenter un contrat nommé sur un jeu dont le support est fixé et connu, nous réutilisons la même notation que pour les garanties, en remplaçant les ensembles par des formules logiques. Ces formules définissent ces ensembles par compréhension. Nous introduisons les paramètres auxiliaires par des quantificateurs universels portant sur la garantie ainsi écrite, et liant sur les deux formules. Les valeurs de retour auxiliaires, s'il y en a, sont introduites par une liste de variables précédant la formule correspondant à la post-condition. Nous obtenons ainsi la notation suivante :

$$\forall (i \in A_i)_{i\in I}.\ \langle (P) \hookrightarrow ((j \in B_j)_{j\in J} :\ Q)\rangle$$

où $(A_i)_{i\in I}$, $(B_j)_{j\in J}$ sont les associations de noms du contrat nommé, et P, Q les formules définissant la pré- et la post-condition. Dans ces formules, nous réservons le nom de paramètre now pour représenter l'état actuel du jeu, et le nom de paramètre old pour représenter l'état initial du jeu dans la post-condition.

Exemple 5.54. Considérons de nouveau l'exemple du jeu $\overline{\mathbb{N}}_\times$, ainsi que l'énoncé étiqueté donné dans l'exemple 5.42. Nous pouvons réécrire le contrat correspondant à l'hypothèse `mult` via le contrat nommé suivant :

$$\forall k \in \mathbb{N}.\ \langle (k \neq 0 \wedge \mathsf{now} \neq \infty) \hookrightarrow (l \in \mathbb{N} :\ l \neq 0 \wedge \mathsf{now} = k \times l \times \mathsf{old})\rangle$$

Le contrat sous-jacent est alors le contrat formé par :
- l'ensemble de paramètres auxiliaires $\mathbb{N}$,

— l'ensemble de valeurs de retour auxiliaires $\mathbb{N}$,
— la pré-condition paramétrée $(\{n \in \overline{\mathbb{N}} \mid k \neq 0 \wedge n \neq \infty\})_{k\in\mathbb{N}}$,
— la post-condition paramétrée $(\{m \in \overline{\mathbb{N}} \mid l \neq 0 \wedge m = k \times l \times n\})_{(k,n,l)\in\mathbb{N}\times\overline{\mathbb{N}}\times\mathbb{N}}$

Pour représenter les énoncés nommés, nous utilisons une barre d'inférence. Nous représentons le contexte nommé au-dessus de la barre d'inférence, par une liste donnant pour chaque nom d'hypothèse du contexte les deux associations de noms correspondantes. Nous plaçons le contrat nommé sous-jacent à l'énoncé nommé en dessous de la barre d'inférence.

Exemple 5.55. Considérons une fois encore l'énoncé étiqueté donné dans l'exemple 5.42. Nous obtenons alors la notation suivante pour sa version nommée :

$$\frac{\texttt{mult} : \forall k \in \mathbb{N}.\ \langle (k \neq 0 \wedge \mathsf{now} \neq \infty) \hookrightarrow (l \in \mathbb{N} :\ l \neq 0 \wedge \mathsf{now} = k \times l \times \mathsf{old}) \rangle}{\forall n \in \mathbb{N} : \langle (\mathsf{now} \neq \infty) \hookrightarrow (\mathsf{now} \neq \infty\ \wedge\ \forall k.\ 1 \leq k \leq n \Rightarrow k | \mathsf{now}) \rangle}$$

Remarque 5.56. Cette dernière notation ne s'applique qu'à des énoncés nommés où le contexte ne dépend pas des paramètres auxiliaires du contrat final, ni de son état d'entrée. Dans le cas où une telle dépendance existe, comme dans certains énoncés nommés donnés dans le chapitre 6, nous préférons donner explicitement les composants de ces énoncés nommés en précisant les dépendances.

Pour ce qui est des transformateurs nommés, nous donnerons les notations de construction au fur et à mesure des règles de combinaison correspondantes. Ces notations seront supposées écrites dans un contexte où l'on connaît une signature et une association de noms correspondant aux paramètres auxiliaires, qui sont utilisées pour définir le transformateur sous-jacent. Cela revient plus ou moins à une légère inférence de types. Pour déterminer ces informations au niveau d'une combinaison complète, nous écrivons au-dessus de la combinaison l'interface du transformateur, c'est-à-dire la signature et les associations de noms correspondantes. Nous pouvons alternativement écrire l'énoncé nommé que l'on cherche à établir, car celui-ci fournit la totalité de ces informations.

Définition 5.57 (Notation pour l'interface d'un transformateur)**.** Soit W un transformateur nommé, $(A_i)_{i\in I}$ et $(B_j)_{j\in J}$ les associations de noms correspondant respectivement aux paramètres et retours auxiliaires, et

$$(H_k \to ((A'_{k,i})_{i\in I_k}), ((B'_{k,j})_{j\in J_k})_{k\in K}$$

l'association d'hypothèses associée à W. Nous notons l'*interface* d'un transformateur, c'est-à-dire le triplet formé des deux associations de noms et de l'association d'hypothèses, par

$$\left[\begin{array}{l} \left(H_k : \left[\ (i \in A'_{k,i})_{i\in I_k}\ \right] \Rightarrow \left[\ (j \in B'_{k,j})_{j\in J_k}\ \right]\right)_{k\in K} \\ (i \in A_i)_{i\in I} \end{array}\right] \Rightarrow \left[\ (j \in B_j)_{j\in J}\ \right]$$

Exemple 5.58. Considérons l'énoncé nommé proposé dans l'exemple 5.55. Nous notons l'interface d'un transformateur visant à démontrer cet énoncé par :

$$\left[\begin{array}{l} \texttt{mult} : \left[\ k \in \mathbb{N}\ \right] \Rightarrow \left[\ l \in \mathbb{N}\ \right] \\ n \in \mathbb{N} \end{array}\right] \Rightarrow []$$

5.3.3 Règles de combinaison pour les transformateurs

Nous donnons maintenant des règles de combinaison pour les transformateurs de post-conditions, permettant de construire de nouveaux transformateurs. Ces règles donnent des équivalents à la totalité des règles d'inférence présentées dans la section 5.2, à l'exception de la règle de conséquence. Cette dernière est en effet appliquée automatiquement si nécessaire par la combinaison de transformateurs, et lors de la conversion entre transformateurs de post-conditions et énoncés étiquetés. En parallèle de la définition de ces règles de combinaison, nous donnons des notations pour l'écriture pratique de transformateurs. Pour effectuer correctement la liaison dans ces notations, nous utilisons la structure de transformateur nommé.

Dans un premier temps, nous ne présentons que les règles de combinaison n'impliquant pas de situations limites. Celles-ci correspondent aux règles de calcul habituelles du calcul de plus faible pré-condition, tandis que les règles d'itération et de récurrence sont munies de contraintes supplémentaires pour la gestion des situations limites.

Création de paramètres auxiliaires (règle d'identité)

Nous présentons tout d'abord une règle de création de transformateur correspondant à l'application d'une fonction sur les paramètres auxiliaires. Comme cette règle correspond intuitivement à une étape vide pour le jeu sous-jacent, il s'agit d'un analogue de la règle d'identité.

Lemme 5.59. *Soient A, B deux ensembles quelconques, et f une fonction de $A \times G$ dans B. Alors la fonction $W_f^{A,B}$ définie par*

$$W_f^{A,B}(\Gamma, (Q_b)_{b\in B}, a) = \{x \in G \mid x \in Q_{f(a,x)}\}$$

est un transformateur de post-conditions de type A, B, appelé le transformateur fonctionnel *associé à f.*

Démonstration. Nous pouvons utiliser la règle de partition pour distribuer complètement les énoncés voulus sur les ensembles singleton $\{x\}$ pour $x \in Q_{f(a,x)}$. Le résultat devient alors une application directe de la règle d'identité. □

Bien que cette règle semble extrêmement simple et peu utile, nous l'utilisons en réalité très souvent pour lier de nouveaux paramètres.

Définition 5.60 (Notation pour la liaison fonctionnelle de nouveaux paramètres)**.** Supposons données une signature $\mathcal{S}$, ainsi qu'une association de noms $(A_i)_{i\in I}$ pour les paramètres auxiliaires. Soient $(j \in B_j \leftarrow e_j)_{j\in J}$ une collection de noms distincts J, associés à des ensembles $(B_j)_{j\in J}$ et à des expressions $(e_j)_{j\in J}$, dont les variables libres appartiennent à $I \uplus \{\mathsf{now}\}$. Alors nous définissons le transformateur de liaison fonctionnelle des nouveaux paramètres J par les ensembles $(B_k)_{k\in\{x\}}$ et les expressions $(e_j)_{j\in J}$, noté

$$\mathsf{let}\ (j \in B_j \leftarrow e_j)_{j\in J}$$

comme le transformateur nommé défini par la signature $\mathcal{S}$, les associations de noms $(A_i)_{i\in I}$ et $(B_j)_{j\in J}$, et par le transformateur de post-conditions fonctionnel défini par

la fonction

$$\begin{array}{rccl} f: & (\prod_{i\in I} A_i) \times G & \to & \prod_{j\in J} B_j \\ & ((v_i)_{i\in I}, x) & \to & (e_j[(i \leftarrow v_i)_{i\in I}, \mathsf{now} \leftarrow x])_{j\in J} \end{array}$$

Nous autorisons également l'omission de l'ensemble B_j pour les variables qui appartiennent déjà à I. Dans ce cas, nous considérons que la variable conserve le même domaine, c'est-à-dire que $B_j = A_j$. Nous autorisons également l'omission de l'ensemble dans le cas où le nom a été globalement associé à un ensemble. Ces omissions sont également possibles pour toutes les formes d'introduction de paramètres. Enfin, si J est vide, nous utilisons la notation skip à la place de let .

Notons que nous pouvons généraliser la liaison fonctionnelle à la liaison relationnelle de nouveaux paramètres, correspondant au choix arbitraire de ces nouveaux paramètres parmi un ensemble de possibilités.

Lemme 5.61. *Soient A, B deux ensembles quelconques, et $\mathcal{R}$ une relation entre $A \times G$ et B. Alors la fonction $W_{\mathcal{R}}^{A,B}$ définie par*

$$W_{\mathcal{R}}^{A,B}(\Gamma, (Q_b)_{b\in B}, a) = \{x \in G \mid \mathcal{R}\{(a, x)\} \neq \emptyset \wedge \forall b.(a, x)\mathcal{R}b \Rightarrow x \in Q_b\}$$

est un transformateur de post-conditions de type A, B, appelé le transformateur relationnel *associé à $\mathcal{R}$.*

Démonstration. Nous effectuons la même preuve que pour la construction fonctionnelle : application de la règle de partition pour distribuer les énoncés sur des singletons, puis application de la règle d'identité. □

Définition 5.62 (Notation pour la liaison relationnelle de nouveaux paramètres)**.** Le transformateur de liaison relationnelle des nouveaux paramètres J par les ensembles $(B_j)_{j\in J}$ et l'expression booléenne $\mathcal{B}$ (dont les variables libres appartiennent à $(I \cup J) \uplus \{\mathsf{now}\}$), noté

$$\mathsf{choose}\ (j \in B_j)_{j\in J}\ \mathsf{such\ that}\ \mathcal{B}$$

est défini de la même manière que le transformateur de liaison fonctionnelle, en remplaçant le transformateur de post-conditions fonctionnel par le transformateur de post-conditions relationnel défini par la relation $\mathcal{R}$ suivante entre $(\prod_{i\in I} A_i) \times G$ et $\prod_{j\in J} B_j$:

$$((v_i)_{i\in I}, x)\mathcal{R}(w_j)_{j\in J} \Leftrightarrow \mathcal{B}[(j \leftarrow w_j)_{j\in J}][(i \leftarrow v_i)_{i\in I}, \mathsf{now} \leftarrow x].$$

L'ordre choisi entre les substitutions revient à dire que les variables nouvellement introduites par J cachent celles de I en cas de conflit. Comme pour le transformateur relationnel, nous autorisons l'omission d'un ensemble B_j si un nom a été globalement associé à un ensemble donné.

Définition 5.63 (Notation pour un transformateur d'impossibilité)**.** Dans le cas où la famille de variables liées est vide et l'expression booléenne est définie comme fausse, nous utilisons la notation absurd pour la version nommée du transformateur relationnel.

Liaison séquentielle (règle de composition)

Nous donnons maintenant une règle de combinaison analogue de la règle de composition, correspondant essentiellement à la règle de plus faible pré-condition pour une construction séquentielle.

Lemme 5.64. *Soit $W^{A,B}$ un transformateur de post-conditions de type A, B, et $W^{A\times B,C}$ un transformateur de post-conditions de type $A \times B, C$. Alors la fonction $W^{A,C}_{Seq}$ définie par*

$$W^{A,C}_{Seq}(\Gamma, (Q_c)_{c\in C}, a) = W^{A,B}(\Gamma, (W^{A\times B,C}(\Gamma, (Q_c)_{c\in C}, (a, b)))_{b\in B}, a)$$

est un transformateur de post-conditions de type A, C, appelé le transformateur de liaison séquentielle *de $W^{A,B}$ et de $W^{A\times B,C}$.*

Démonstration. Pour tout Γ contexte étiqueté et $a \in A$ donnés, nous pouvons, par le biais de la règle de partition, fusionner les différents énoncés valides provenant de $W^{A\times B,C}$ et de Γ, a en

$$\Gamma \vdash_{\preccurlyeq} \left\langle \bigcup_{b\in B} W^{A\times B,C}(\Gamma, (Q_c)_{c\in C}, (a, b)) \hookrightarrow \bigcup_{c\in C} Q_c \right\rangle.$$

Il suffit alors d'appliquer la règle de composition pour conclure. □

Le fait que les paramètres auxiliaires du second transformateur soient représentés par $A \times B$ revient à dire que les valeurs de retour auxiliaires du premier transformateur sont introduites comme de nouveaux paramètres auxiliaires pour le second transformateur. Cela revient en fait précisément à la plus faible pré-condition d'une construction de la forme `let ... = ... in ...` dans un langage de type ML.

Nous ne pouvons cependant pas transformer cette construction directement en transformateur nommé. En effet, l'ensemble $A \times B = (\prod_{i\in I} A_i) \times (\prod_{j\in J} B_j)$ ne peut pas être représenté par une association de noms si $I \cap J \neq \emptyset$, c'est-à-dire en cas de conflit de nom entre les valeurs de retours et les paramètres. Dans ce cas, nous souhaitons que les valeurs de retours cachent les anciens paramètres, ce qui simule une mise à jour. Il faut donc une construction additionnelle pour ajuster les paramètres d'entrée du second transformateur, ce que nous obtenons par le biais d'une construction purement compositionnelle.

Lemme 5.65. *Soit $W^{A,B}$ un transformateur de post-conditions de type A, B, et $W^{B,C}$ un transformateur de post-conditions de types B, C. Alors la fonction $W^{A,C}_{Comp}$ définie par*

$$W^{A,C}_{Comp}(\Gamma, (Q_c)_{c\in C}, a) = W^{A,B}(\Gamma, (W^{B,C}(\Gamma, (Q_c)_{c\in C}, b))_{b\in B}, a)$$

est un transformateur de post-conditions de type A, C, appelé la composition *de $W^{A,B}$ et $W^{B,C}$.*

Démonstration. Nous effectuons la même preuve que pour le lemme 5.64, à ceci près que a n'est pas passé en argument à $W^{B,C}$. □

Définition 5.66 (Notation pour la liaison séquentielle de transformateurs nommés)**.** Supposons données une signature $\mathcal{S}$ et une association de noms $(A_i)_{i\in I}$ pour les paramètres auxiliaires. Soient W_1 un transformateur nommé pour $\mathcal{S}$, $(A_i)_{i\in I}$, et l'association de noms $(B_j)_{j\in J}$ pour les valeurs de retour auxiliaires de W_1. Posons $(\hat{A}_l)_{l\in I\cup J}$ l'association de noms définie par

$$\hat{A}_l = \begin{cases} B_l & \text{si } l \in J \\ A_l & \text{sinon} \end{cases} .$$

Soit également W_2 un transformateur nommé pour $\mathcal{S}$, $(\hat{A}_l)_{l\in I\cup J}$, et l'association de noms $(C_k)_{k\in K}$ pour les valeurs de retour auxiliaires de W_2. Alors nous définissons la *liaison séquentielle* de W_1 et W_2, notée par

$$\begin{array}{l} W_1 \\ W_2 \end{array}$$

de la manière suivante. La structure nommée de la liaison séquentielle de W_1 et W_2 est donnée respectivement par $\mathcal{S}$, $(A_i)_{i\in I}$, et $(C_k)_{k\in K}$. Pour définir le transformateur de post-conditions sous-jacent, nous définissons tout d'abord la fonction d'ajustement des paramètres f par

$$\begin{array}{lll} f: & ((\prod_{i\in I} A_i) \times (\prod_{j\in J} B_j)) \times G & \to \quad \prod_{l\in I\cup J} \hat{A}_l \\ & (((v_i)_{i\in I}, (w_j)_{j\in J}), x) & \to \quad \left(l \in I\cup J \to \begin{cases} w_l & \text{si } l \in J \\ v_l & \text{sinon} \end{cases} \right) \end{array}$$

Alors le transformateur de post-conditions sous-jacent à la liaison séquentielle de W_1 et W_2 est défini comme la liaison séquentielle du transformateur de post-conditions $W^{\prod_{i\in I} A_i, \prod_{j\in J} B_j}$ sous-jacent à W_1, et de la composition du transformateur fonctionnel associé à f avec le transformateur de post-conditions $W^{\prod_{l\in I\cup J} \hat{A}_l, \prod_{k\in K} C_k}$ sous-jacent à W_2.

Pour donner un sens à la liaison séquentielle de plus de deux transformateurs, nous fixons la convention que la liaison séquentielle est associative vers le bas. De cette manière, les variables introduites par un transformateur sont liées dans tous les transformateurs qui suivent, ce qui est le comportement voulu dans la majorité des cas. Nous introduisons également une notation pour lier localement des variables :

Définition 5.67 (Notation pour l'instanciation d'un transformateur)**.** Nous utilisons la notation

$$W[(x_i \in A_i \leftarrow e_i)_{i\in I}]$$

comme un raccourci pour le transformateur nommé

$$\begin{array}{l} \mathsf{let}\ (x_i \in A_i \leftarrow e_i)_{i\in I} \\ W \end{array}$$

En particulier, les noms introduits par le transformateur de liaison fonctionnelle ne sont visibles que dans W.

Appel d'hypothèse (règle d'axiome)

Lemme 5.68. *Soit A, B deux ensembles et H un nom d'hypothèse. Alors la fonction $W_H^{A,B}$ dont la valeur en $(\Gamma, (Q_b)_{b\in B}, a)$ est définie comme*

$$\begin{cases} \{x \in P_a \mid \forall b \in B.\ Q'_{a,x,b} \subseteq Q_b\} & \text{si } \Gamma(H) = (A, B, (P_{a'})_{a'\in A}, (Q'_{a',x,b})_{(a',x,b)\in A\times G\times B}) \\ \emptyset & \text{sinon} \end{cases}$$

est un transformateur de post-conditions de type A, B, appelé le transformateur d'appel *de H.*

Démonstration. la preuve est quasiment identique à celle du lemme 5.47, à ceci près que le contexte utilisé est celui passé en paramètre, et donc que la condition d'inclusion de contexte est triviale. En utilisant la règle de partition pour distribuer sur l'ensemble, nous réduisons la preuve à la démonstration de l'énoncé $\Gamma \vdash_{\preccurlyeq} \langle\{x\} \hookrightarrow \bigcup_{b\in B} Q_b\rangle$ pour tout $x \in W_v^{A,B}(\Gamma, (Q_b)_{b\in B}, a)$. Comme alors $\Gamma(H) = (A, B, (P_a), (Q'_{a,x,b}))$, $x \in P_a$, et $\bigcup_{b\in B} Q'_{a,x,b} \subseteq \bigcup_{b\in B} Q_b$, nous obtenons immédiatement le résultat en employant la règle d'axiome puis celle de conséquence. □

Définition 5.69 (Notation pour le transformateur d'appel)**.** Supposons données une signature $\mathcal{S}$ et une association de noms $(A_i)_{i\in I}$ correspondant à des paramètres auxiliaires. Soit H un nom d'hypothèse, associé par la signature $\mathcal{S}$ aux associations de noms respectives $(A'_i)_{i\in I'}$ et $(B_j)_{j\in J}$. Si $I' \subseteq I$, et si pour tout $i \in I'$, $A_i \subseteq A'_i$, alors nous étendons la notion de transformateur d'appel de H au transformateur nommé, que nous notons directement par H au sein d'une combinaison, comme le transformateur dont la structure nommée est donnée par respectivement $\mathcal{S}$, $(A_i)_{i\in I}$ et $(B_j)_{j\in J}$, et dont le transformateur de post-conditions sous-jacent est la composition d'un transformateur fonctionnel correspondant à la sélection des paramètres d'appel et du transformateur d'appel de H. La fonction de sélection de paramètres est donnée par

$$\begin{array}{ccc} \prod_{i\in I} A_i & \to & \prod_{i\in I'} A'_i \\ (v_i)_{i\in I} & \to & (v_i)_{i\in I'} \end{array}$$

Exemple 5.70. Nous disposons maintenant de suffisamment de règles de combinaison pour pouvoir démontrer l'énoncé proposé dans l'exemple 5.55, en utilisant la multiplication par une factorielle. Nous résumons la preuve en donnant le transformateur pour l'énoncé, c'est-à-dire par

$$\frac{\texttt{mult} : \forall k \in \mathbb{N}.\ \langle (k \neq 0 \wedge \mathsf{now} \neq \infty) \hookrightarrow (l \in \mathbb{N} :\ l \neq 0 \wedge \mathsf{now} = k \times l \times \mathsf{old})\rangle}{\forall n \in \mathbb{N} : \langle (\mathsf{now} \neq \infty) \hookrightarrow (\mathsf{now} \neq \infty\ \wedge\ \forall k.\ 1 \leq k \leq n \Rightarrow k|\mathsf{now})\rangle}$$
$$\texttt{mult}[k \in \mathbb{N} \leftarrow n!]$$
$$\mathsf{skip}$$

La construction skip finale sert uniquement à ignorer le résultat auxiliaire l de l'appel à `mult`, ce qui est nécessaire pour des raisons de « typage » car l'énoncé que l'on cherche à démontrer n'a pas de valeur de retour auxiliaire. Comme l'énoncé est déjà fixé, nous pourrions également nous contenter de donner le transformateur par

$$\left[\begin{array}{l} \texttt{mult} : \left[\ k \in \mathbb{N}\ \right] \Rightarrow \left[\ l \in \mathbb{N}\ \right] \\ n \in \mathbb{N} \end{array}\right] \Rightarrow []$$
$$\texttt{mult}[k \in \mathbb{N} \leftarrow n!]$$
$$\mathsf{skip}$$

Γ	contexte provenant de l'énoncé étiqueté
Q	post-condition provenant de l'énoncé étiqueté
$W_{k\in\mathbb{N}\leftarrow n!}$	transformateur fonctionnel pour $k \leftarrow n!$
$W_{\texttt{mult}}$	transformateur d'appel pour `mult`
$W_{f,0}$	transformateur fonctionnel pour sélectionner les paramètres de l'appel à `mult`
$W_{c,1}$	composition de $W_{f,0}$ et de $W_{\texttt{mult}}$
$W_{f,2}$	transformateur fonctionnel pour ajuster les paramètres entre $W_{k\in\mathbb{N}\leftarrow n!}$ et $W_{c,1}$
$W_{c,3}$	composition de $W_{f,2}$ et de $W_{c,1}$
$W_{\texttt{mult}[k\in\mathbb{N}\leftarrow n!]}$	liaison séquentielle de $W_{k\in\mathbb{N}\leftarrow n!}$ et de $W_{c,3}$
W_{skip}	transformateur fonctionnel vide
$W_{f,4}$	transformateur fonctionnel pour ajuster les paramètres entre $W_{\texttt{mult}[k\in\mathbb{N}\leftarrow n!]}$ et W_{skip}
$W_{c,5}$	composition de $W_{f,4}$ et de W_{skip}
W_{all}	transformateur complet, liaison séquentielle de $W_{\texttt{mult}[k\in\mathbb{N}\leftarrow n!]}$ et de $W_{c,5}$

$$
\begin{array}{ll}
 & \forall n \in \mathbb{N}, x \in \{x \in \overline{\mathbb{N}} \mid x \neq \infty\}.\ x \in W_{all}(\Gamma, (Q)_{b\in\{0\}}, n) \\
\Leftrightarrow & \forall n \in \mathbb{N}, x \in \overline{\mathbb{N}}.\ x \neq \infty \Rightarrow x \in W_{\texttt{mult}[k\in\mathbb{N}\leftarrow n!]}(\Gamma, (W_{c,5}(\Gamma, (Q)_{b\in\{0\}}, (n,l)))_{l\in\mathbb{N}}, n) \\
\Leftrightarrow & \left(\begin{array}{l} \forall n \in \mathbb{N}, x \in \overline{\mathbb{N}}.\ x \neq \infty \Rightarrow \\ \quad x \in W_{k\in\mathbb{N}\leftarrow n!}(\Gamma, (W_{c,3}(\Gamma, (W_{c,5}(\Gamma, (Q)_{b\in\{0\}}, (n,l)))_{l\in\mathbb{N}}, (n,k)))_{k\in\mathbb{N}}, n) \end{array}\right) \\
\Leftrightarrow & \left(\begin{array}{l} \forall n \in \mathbb{N}, x \in \overline{\mathbb{N}}.\ x \neq \infty \Rightarrow \\ \quad x \in W_{c,3}(\Gamma, (W_{f,4}(\Gamma, (W_{\mathsf{skip}}(\Gamma, (Q)_{b\in\{0\}}, (n,l)))_{(n,l)\in\mathbb{N}^2}, (n,l)))_{l\in\mathbb{N}}, (n,n!)) \end{array}\right) \\
\Leftrightarrow & \left(\begin{array}{l} \forall n \in \mathbb{N}, x \in \overline{\mathbb{N}}.\ x \neq \infty \Rightarrow \\ \quad x \in W_{f,2}(\Gamma, (W_{c,1}(\Gamma, (W_{f,4}(\Gamma, (Q)_{(n,l)\in\mathbb{N}^2}, (n,l)))_{l\in\mathbb{N}}, (n,k)))_{(n,k)\in\mathbb{N}^2}, (n,n!)) \end{array}\right) \\
\Leftrightarrow & \left(\begin{array}{l} \forall n \in \mathbb{N}, x \in \overline{\mathbb{N}}.\ x \neq \infty \Rightarrow \\ \quad x \in W_{c,1}(\Gamma, (Q)_{l\in\mathbb{N}}, (n,n!)) \end{array}\right) \\
\Leftrightarrow & \left(\begin{array}{l} \forall n \in \mathbb{N}, x \in \overline{\mathbb{N}}.\ x \neq \infty \Rightarrow \\ \quad x \in W_{f,0}(\Gamma, (W_{\texttt{mult}}(\Gamma, (Q)_{l\in\mathbb{N}}, k))_{k\in\mathbb{N}}, (n,n!)) \end{array}\right) \\
\Leftrightarrow & \left(\begin{array}{l} \forall n \in \mathbb{N}, x \in \overline{\mathbb{N}}.\ x \neq \infty \Rightarrow \\ \quad x \in W_{\texttt{mult}}(\Gamma, (Q)_{l\in\mathbb{N}}, n!) \end{array}\right) \\
\Leftrightarrow & \left(\begin{array}{l} \forall n \in \mathbb{N}, x \in \overline{\mathbb{N}}.\ x \neq \infty \Rightarrow \\ \quad n! \neq 0 \wedge x \neq \infty \wedge \forall l \in \mathbb{N}.\ \{y \in \overline{\mathbb{N}} \mid l \neq 0 \wedge y = n! \times l \times x\} \subseteq Q \end{array}\right) \\
\Leftrightarrow & \left(\begin{array}{l} \forall n \in \mathbb{N}, x \in \overline{\mathbb{N}}.\ x \neq \infty \Rightarrow \\ \quad n! \neq 0 \wedge x \neq \infty \wedge \forall l \in \mathbb{N}, y \in \overline{\mathbb{N}}.\ l \neq 0 \wedge y = n! \times l \times x \Rightarrow \\ \qquad \forall k.\ 1 \leq k \leq n \Rightarrow k|y \end{array}\right)
\end{array}
$$

FIGURE 5.2 – Exemple de calcul de condition de vérification

Pour effectuer la preuve, nous procédons de la manière suivante. Nous commençons par construire le transformateur de post-conditions représenté par cette notation, puis nous l'appliquons à l'énoncé final. Nous obtenons alors la condition de vérification en déroulant mécaniquement les définitions des transformateurs de post-conditions au niveau de l'application. Nous montrons dans la figure 5.2 le calcul détaillé de la condition de vérification pour cet exemple. Notons que dans ce cas précis, la condition de vérification découle immédiatement des propriétés de la factorielle.

Branchement, meilleur choix et généralisation (règle de partition)

Nous pouvons donner un analogue extrêmement simple de la règle de partition comme règle de combinaison de transformateurs.

Lemme 5.71. *Soit $(W_i^{A,B})_{i\in I}$ une famille de transformateurs de post-conditions de type A, B. Alors la fonction $W_{alt}^{A,B}$ définie par*

$$W_{alt}^{A,B}(\Gamma, (Q_b)_{b\in B}, a) = \bigcup_{i\in I} W_i^{A,B}(\Gamma, (Q_b)_{b\in B}, a)$$

est un transformateur de post-conditions de type A, B, appelé le transformateur de meilleur choix *associé à la famille $(W_i^{A,B})_{i\in I}$.*

Démonstration. Application directe de la règle de partition. □

Le transformateur de meilleur choix possède cependant le défaut d'introduire une disjonction, ou un quantificateur existentiel, au-dessus de conditions générées par l'application de transformateurs. Ce transformateur n'a d'ailleurs pas d'équivalent dans le calcul de plus faible pré-condition standard. Nous pouvons toutefois relier cette construction à l'opérateur ambigu `amb` de McCarthy [66], qui étant donnés deux expressions renvoie le résultat de n'importe laquelle qui soit définie.

Pour pallier au problème de la disjonction, nous donnons un autre analogue de la règle de partition basée sur un découpage explicite, correspondant au calcul de plus faible pré-condition pour une commande gardée [40]. Cela permet notamment de simuler les constructions conditionnelles.

Lemme 5.72. *Soit $(W_i^{A,B})_{i\in I}$ une famille de transformateurs de post-conditions de type A, B, et $(P_{i,a})_{(i,a)\in I\times A}$ une famille de sous-ensembles de G. Alors la fonction $W_{galt}^{A,B}$ définie par*

$$W_{galt}^{A,B}(\Gamma, (Q_b)_{b\in B}, a) = \{x \in \bigcup_{i\in I} P_{i,a} \mid \forall i \in I.\ x \in P_{i,a} \to x \in W_i^{A,B}(\Gamma, (Q_b)_{b\in B}, a)\}$$

est un transformateur de post-conditions de type A, B, appelé le transformateur de branchement *selon les $(P_{i,a})_{(i,a)\in I\times A}$ vers les $(W_i^{A,B})_{i\in I}$.*

Démonstration. Nous utilisons la règle de partition pour découper l'énoncé selon les $P_{i,a}$, puis la règle de conséquence pour réduire à un énoncé déjà connu. □

Définition 5.73 (Notation pour un transformateur de branchement)**.** Soient $(W_i)_{1\leq i\leq n}$ une famille non vide de transformateurs nommés possédant tous la même structure nommée (même signature, mêmes associations de noms), et $(B_i)_{1\leq i\leq n}$ une famille d'expressions booléennes, dont les variables libres sont soit des noms de paramètres auxiliaires, soit now. Nous utilisons alors la notation

$$\begin{array}{l}\mathsf{switch}\\ \mathsf{case}\ B_1 :\\ \quad W_1\\ \quad \vdots\\ \mathsf{case}\ B_n :\\ \quad W_n\\ \mathsf{end\text{-}switch}\end{array}$$

pour représenter le transformateur de branchement selon les sous-ensembles caractérisés par la compréhension des expressions $(B_i)_{1\leq i\leq n}$, vers les transformateurs $(W_i)_{1\leq i\leq n}$. La structure nommée du transformateur de branchement est la structure commune aux $(W_i)_{1\leq i\leq n}$. Nous autorisons l'omission de B_n pour le dernier cas, alors noté

$$\begin{array}{l}\mathsf{default} :\\ \quad W_n\end{array}$$

et équivalent à la définition de B_n comme la conjonction des négations des autres $(B_i)_{1\leq <n}$. Nous autorisons également la notation

$$\begin{array}{l}\mathsf{case\ exist}\ (x_i)_{0\leq i<n}\ \mathsf{such\ that}\ B :\\ \quad W\end{array}$$

comme un raccourci pour

$$\begin{array}{l}\mathsf{case}\ \exists (x_i)_{0\leq i<n}.\ B :\\ \quad \mathsf{choose}\ (x_i)_{0\leq i<n}\ \mathsf{such\ that}\ B\\ \quad W\end{array}$$

Remarquons que la condition existentielle introduite par le transformateur relationnel est exactement donnée par la validité de la condition de garde, et peut donc être simplifiée. Dans ce type de cas, nous éliminerons systématiquement cette contrainte existentielle lors du calcul de la condition de vérification.

Remarquons que dans le cas d'un branchement à un seul cas, cette dernière règle de combinaison représente l'insertion d'une assertion intermédiaire. Nous pouvons également l'utiliser pour dériver une troisième variante de la règle de partition, où la partition est représentée par les paramètres auxiliaires. Ce dernier cas représente intuitivement la transformation d'une quantification universelle externe par l'ajout d'un paramètre auxiliaire.

Lemme 5.74. *Soit $(W_i^{A,B})_{i\in I}$ une famille de transformateurs de post-conditions de type A, B. Alors la fonction $W_{join}^{A\times I,B}$ définie par*

$$W_{join}^{A\times I,B}(\Gamma, (Q_b)_{b\in B}, (a,i)) = W_i^{A,B}(\Gamma, (Q_b)_{b\in B}, a)$$

est un transformateur de post-condition de type $A\times I, B$.

Démonstration. Immédiat par définition. Nous pouvons aussi dériver ce transformateur en utilisant les règles de combinaison déjà connues. Pour cela, nous changeons tout d'abord les paramètres auxiliaires des $W_i^{A,B}$ en $A \times I$ en composant l'entrée avec une fonction qui oublie la composante I, puis nous partitionnons sur la valeur de cette composante additionnelle. □

Progression

Nous donnons ensuite un analogue de la règle de progression. Cette règle de combinaison établit la progression entre l'entrée et la sortie du transformateur. Notons que si l'on souhaite établir toutes les progressions possibles sans utiliser directement cette règle, nous pouvons simplement adapter les règles de combinaison pour appliquer la progression de manière systématique.

Lemme 5.75. *Soit $W^{A,B}$ un transformateur de post-conditions de type A, B. Alors la fonction $W^{A,B}_{prog}$ définie par $x \in W^{A,B}_{prog}(\Gamma, (Q_b)_{b\in B}, a)$ si et seulement si*

$$x \in W^{A,B}(\Gamma, (\{y \in G \mid x \preccurlyeq y \to y \in Q_b\})_{b\in B}, a)$$

est un transformateur de post-conditions de type A, B, appelé le transformateur de progression *pour $W^{A,B}$.*

Démonstration. L'inclusion

$$\{y \in \bigcup_{b\in B} \{z \in G \mid x \preccurlyeq z \to z \in Q_b\} \mid x \preccurlyeq y\} \subseteq \bigcup_{b\in B} Q_b$$

montre qu'il s'agit d'une application directe des règles de progression et de conséquence. □

Définition 5.76 (Transformateur de progression)**.** Soit W un transformateur nommé. Nous utilisons la notation

progression
W
end-progression

pour représenter la version nommée du transformateur de progression associé à W, qui est obtenu en ajoutant la structure nommée de W au transformateur de progression obtenu pour le transformateur de post-conditions correspondant.

Introduction d'hypothèse (règle de remplacement)

Lemme 5.77. *Supposons donnés $W^{A,B}$ un transformateur de post-conditions de type A, B, $W^{A\times A',B'}$ un transformateur de post-conditions de type $A \times A', B'$, et*

$$(A', B', (P_{a,a'})_{(a,a')\in A\times A'}, (Q_{a,a',x,b'})_{(a,a',x,b')\in A\times A'\times G\times B'})$$

un contrat dont les pré et post-conditions sont paramétrées par A. Soit également $\mathtt{H}$ un nom d'hypothèse, et E la fonction définie par

$$E(\Gamma, a)(\mathtt{H}') = \begin{cases} (A', B', (P_{a,a'})_{a'\in A'}, (Q_{a,a',x,b'})_{(a',x,b')\in A'\times G\times B'} & \text{si } \mathtt{H}' = \mathtt{H} \\ \Gamma(\mathtt{H}') & \text{sinon} \end{cases}$$

correspondant à l'association du nom H *à ce contrat, potentiellement en écrasant un nom existant. Alors la fonction* $W^{A,B}_{intro}$ *définie par* $x \in W^{A,B}_{intro}(\Gamma, (Q_b)_{b\in B}, a)$ *si et seulement si les deux conditions suivantes sont vérifiées :*

(i) $\forall a' \in A', y \in P_{a,a'}.\ y \in W^{A\times A',B'}(\Gamma, (Q_{a,a',y,b'})_{b'\in B'}, (a, a'))$

(ii) $x \in W^{A,B}(E(\Gamma, a), (Q_b)_{b\in B}, a)$

est un transformateur de post-conditions de type A, B*, appelé* transformateur d'introduction *de l'hypothèse de nom* H *associée au contrat paramétré.*

Démonstration. Si la condition (i) n'est pas vérifiée, alors le résultat de $W^{A,B}_{intro}(\Gamma, (Q_b)_{b\in B}, a)$ est vide, auquel cas la règle d'identité permet de conclure. Sinon, le lemme 5.44 montre que l'énoncé étiqueté

$$((\Gamma), (A', B', (P_{a,a'})_{a'\in A'}, (Q_{a,a',x,b'})_{(a',x,b')\in A'\times G\times B'}))$$

est valide. En particulier, la règle de remplacement permet alors de remplacer $E(\Gamma, a)$ par Γ et de transporter le résultat obtenu pour $W^{A,B}$. □

Définition 5.78 (Notation pour un transformateur d'introduction)**.** Supposons données une signature $\mathcal{S}$ et une association de noms $(A_i)_{i\in I}$ pour les paramètres auxiliaires. Soit H un nom d'hypothèse, $\mathcal{C}$ un contrat nommé, et $(A'_j)_{j\in J}$ l'association de noms correspondant aux paramètres auxiliaires de $\mathcal{C}$. Soit $W_{\mathcal{C}}$ un transformateur nommé admettant la structure nommée donnée par $\mathcal{S}$, $(A_i)_{i\in I}$ étendue par l'association de noms correspondant aux paramètres auxiliaires de $\mathcal{C}$, et l'association de noms correspondant aux valeurs de retours auxiliaires de $\mathcal{C}$. Soit également W un transformateur nommé dont la signature est donnée par l'ajout (ou le remplacement) dans $\mathcal{S}$ de l'hypothèse H par les association de noms de $\mathcal{C}$, et dont l'association de noms pour les paramètres auxiliaires est $(A_i)_{i\in I}$. Nous utilisons alors la notation

intro $H((j \in A'_j)_{j\in J})$: $\mathcal{C}$
by
  $W_{\mathcal{C}}$
in
  W
end-intro

pour représenter la version nommée du transformateur d'introduction, dont la signature est $\mathcal{S}$, et dont les associations de noms sont identiques à celles de W. Comme dans le cas du transformateur de liaison séquentielle, la construction donnée en termes de transformateurs de post-conditions n'est pas directement applicable à cause de la structure des paramètres auxiliaires de $W_{\mathcal{C}}$. Nous utilisons donc la même méthode pour ajuster ceux-ci, c'est-à-dire de composer avec une fonction d'ajustement, et les mêmes règles de priorité : les nouveaux paramètres introduits par $\mathcal{C}$ cachent les autres.

Comme la notation explicite complètement les paramètres introduits dans $W_{\mathcal{C}}$, nous autorisons l'omission des quantificateurs sur les paramètres auxiliaires au sein de $\mathcal{C}$.

Continuation

Enfin, nous donnons une règle de combinaison analogue à la prise de continuation. Celle-ci peut être vue comme un calcul de plus faible pré-condition pour la

construction `call/cc`, ou pour un bloc de rattrapage d'exception. L'hypothèse additionnelle correspond alors respectivement au contrat de la continuation ou à celui de la construction lançant l'exception rattrapée.

Lemme 5.79. *Soit $W^{A,B}$ un transformateur de post-conditions de type A, B. Soit également `K` un nom d'hypothèse, E une fonction telle que $E(\Gamma, (Q_b)_{b\in B})$ renvoie Γ étendu par l'association du nom `K` vers le contrat $(B, \{0\}, (Q_b)_{b\in B}, (\emptyset))$ (l'ancien contrat associé à `K` dans Γ est écrasé s'il existe). Alors la fonction $W_{\texttt{K}}^{A,B}$ définie par*

$$W_{\texttt{K}}^{A,B}(\Gamma, (Q_b)_{b\in B}, a) = W^{A,B}(E(\Gamma, (Q_b)_{b\in B}), (Q_b)_{b\in B}, a)$$

est un transformateur de post-conditions de type A, B, appelé transformateur d'introduction de continuation *de nom `K` dans $W^{A,B}$.*

Démonstration. Supposons fixées les entrées $\Gamma, (Q_b)_{b\in B}, a$ de $W_{\texttt{K}}^{A,B}$. Nous savons donc que l'énoncé

$$E(\Gamma, (Q_b)_{b\in B}) \vdash_{\preccurlyeq} \left\langle W_{\texttt{K}}^{A,B}(\Gamma, (Q_b)_{b\in B}, a) \hookrightarrow \bigcup_{b\in B} Q_b \right\rangle$$

est valide. Remarquons que les garanties ajoutées à Γ par le nouveau contrat sont toutes des conséquences directes de $\langle \bigcup_{b\in B} Q_b \hookrightarrow \emptyset \rangle$. Nous pouvons donc appliquer la règle de remplacement pour remplacer l'ajout de ce nouveau contrat par l'ajout de la garantie $\langle \bigcup_{b\in B} Q_b \hookrightarrow \emptyset \rangle$ à Γ. Nous éliminons ensuite cette hypothèse par le biais de la règle de continuation, et obtenons ainsi l'énoncé

$$\Gamma \vdash_{\preccurlyeq} \left\langle W_{\texttt{K}}^{A,B}(\Gamma, (Q_b)_{b\in B}, a) \hookrightarrow \bigcup_{b\in B} Q_b \right\rangle$$

□

Définition 5.80 (Notation pour un transformateur d'introduction de continuation)**.** Supposons données une signature $\mathcal{S}$, ainsi que deux associations de noms $(A_i)_{i\in I}$ et $(B_j)_{j\in J}$. Soit `K` un nom d'hypothèse et W un transformateur nommé, dont la signature est $\mathcal{S}$ étendu en `K` par les associations de noms respectives $(B_j)_{j\in J}$ pour les paramètres et l'association vide pour les valeurs de retour. Supposons également que les associations de noms correspondant aux paramètres et aux valeurs de retour de W sont respectivement $(A_i)_{i\in I}$ et $(B_j)_{j\in J}$. Nous notons par

```
continuation K((j ∈ B_j)_{j∈J})
    W
end-continuation
```

la version nommée du transformateur d'introduction de continuation K dans W, obtenu directement à partir de la construction au niveau des transformateurs de post-conditions en ajoutant au résultat la structure nommée donnée par $\mathcal{S}$, $(A_i)_{i\in I}$ et $(B_j)_{j\in J}$.

5.3.4 Règles de combinaison impliquant des limites

Nous voulons donner deux règles de combinaison impliquant des situations limites, correspondant respectivement à l'itération et à la récurrence. Cependant, nous remarquons que la description des situations limites telle que donnée dans les règles d'inférence n'est pas toujours pratique. En effet, ces règles nous obligent à mettre la totalité des paramètres dans un ensemble ordonné, alors que seul un fragment est à priori nécessaire. De plus, cela nous oblige à dupliquer l'état du jeu si celui-ci intervient dans l'ordre de progression utilisé. Nous choisissons donc de changer légèrement le format des situations limites de manière à alléger ces contraintes. Ce format est bien entendu représentable par le biais des situations limites précédemment définies, mais évite de polluer les conditions par un codage systématique.

Définition 5.81 (Ensemble des situations limites étendues)**.** Étant donné un ensemble de paramètres auxiliaires A, une fonction de progression F de $A \times G$ dans un ensemble ordonné $(O, \leq)$, et une famille $(X_a)_{a \in A}$ de sous-ensembles de G indexés par A, l'ensemble des *situations limites étendues* pour la progression F et les contraintes $(X_a)_{a \in A}$ est défini comme étant l'ensemble des triplets (H, f_A, f_G) tels que

(i) H est une chaîne non vide de $(O, \leq)$, qui n'admet pas de maximum
(ii) f_A est une fonction de O dans A
(iii) f_G est une fonction croissante de $(H, \leq)$ dans $(G, \preccurlyeq)$
(iv) pour tout $o \in H$, $o = F(f_A(o), f_G(o))$ et $f_G(o) \in X_{f_A(o)}$
(v) $f_G(H)$ admet une borne supérieure.

Nous notons l'ensemble des situations limites étendues par $\texttt{elimit}_{\preccurlyeq}(F, \leq, X)$.

Nous pouvons alors représenter les situations limites étendues par des situations limites de la manière suivante :

Définition 5.82 (Situations limites associées à des situations limites étendues)**.** Soient $(A, F, (O, \leq), (X_a)_{a \in A})$ des éléments permettant de définir des situations limites étendues. Nous associons à ces éléments l'ensemble des situations limites $\texttt{limit}_{\preccurlyeq}(\leq_{(F,\leq)}, \hat{X})$, où $\leq_{(F,\leq)}$ et $(\hat{X}_{a,x})_{(a,x) \in A \times G}$ sont respectivement un ordre sur $A \times G$ et une famille de sous-ensembles de G, définis par

$$\hat{X}_{a,x} = \begin{cases} \{x\} & \text{si } x \in X_a \\ \emptyset & \text{sinon} \end{cases}$$

et $(a, x) \leq_{(F,\leq)} (b, y)$ si et seulement si les deux conditions suivantes sont vérifiées :

(i) $F(a, x) \leq F(b, y)$
(ii) $F(b, y) \leq F(a, x) \to a = b \wedge x = y$

La situation limite associée à une situation limite étendue (H, f_A, f_G) est alors définie comme $\{(f_A(o), f_G(o)) \mid o \in H\}, f_G \circ F$.

Il suffit de dérouler les définitions pour vérifier que l'objet construit est bien une situation limite. Nous formalisons maintenant l'équivalence entre les deux définitions.

Lemme 5.83. *Soient $(A, F, (O, \leq), (X_a)_{a \in A})$ des éléments permettant de définir des situations limites étendues, et $\leq_{(F,\leq)}, (\hat{X}_{(a,x) \in A \times G})$ les éléments définissant les situations limites associées. Alors pour tout (H, f_A, f_G) élément de $\texttt{elimit}_{\preccurlyeq}(F, \leq, X)$,*

la situation limite (H_0, f_0) *associée vérifie* $F(H_0) = H$*, et donc* $\sup f_G(H) = \sup f_0(H_0)$*. De plus, tout élément* (H_0, f_0) *appartenant à* $\texttt{limit}_{\preccurlyeq}(\leq_{(F,\leq)}, \hat{X})$ *est associée à une unique situation limite étendue.*

Démonstration. La propriété $F(H_0) = H$ suit directement de la condition (iv) définissant les situations limites étendues, et alors $f_G(H) = f_0(H_0)$, d'où l'égalité des bornes supérieures.

Montrons maintenant que tout $(H_0, f_0) \in \texttt{limit}_{\preccurlyeq}(\leq_{(F,\leq)}, \hat{X})$ est bien associée à une unique situation limite étendue. Cela revient à reconstruire les (H, f_A, f_G) correspondants. Comme nous l'avons déjà vérifié, H doit être défini comme $F(H_0)$. Il faut maintenant reconstruire f_A et f_G. La définition de $\leq_{(F,\leq)}$ garantit qu'il existe un unique couple $(a, x) \in H_0$ tel que $o = F(a, x)$. En particulier, nous devons avoir $f_A(o) = a$ et $f_G(o) = x$. Nous définissons donc f_A et f_G comme renvoyant les composantes de cet unique témoin. Il est alors immédiat que H_0 est bien l'image de H par f_A, f_G. Par définition de la famille $\hat{X}$ et d'une situation limite, nous obtenons de plus que pour tout $(a, x) \in H_0$, $f_0(a, x) = x$. En particulier, $f_0 = f_G \circ F$ est bien vérifiée. Enfin, il suffit de dérouler les définitions pour vérifier que (H, f_A, f_G) est bien une situation limite étendue, qui est donc associée à (H_0, f_0). □

Nous pouvons maintenant donner une règle de combinaison analogue à l'itération. Cette règle est proche du calcul de plus faible pré-condition pour une boucle équipée d'un invariant, à la différence près que la contrainte potentielle de bonne fondation est remplacée par une contrainte sur les situations limites. Bien entendu, cette condition est triviale dans le cas où l'ordre de progression inverse est bien fondé.

Notons par ailleurs que nous avons choisi de forcer la post-condition de toute itération à être vide, pour éviter d'avoir à représenter au niveau des paramètres la disjonction entre les deux cas de sortie possibles. Cette simplification n'est pas un problème en pratique car nous pouvons facilement exploiter le transformateur de continuation pour créer la possibilité de sortir de l'itération.

Lemme 5.84. *Supposons donnés*

- *un invariant de boucle* $(I_a)_{a\in A}$
- *une fonction de progression* F *de* $A \times G$ *dans un ensemble ordonné* $(O, \leq)$
- *un transformateur de post-conditions* $W^{A,A}_{step}$ *de type* A, A*, représentant intuitivement le code d'une étape de l'itération*
- *un transformateur de post-conditions* $W^{\texttt{elimit}_{\preccurlyeq}(F,\leq,I),A}_{lim}$ *dont le type est donné par* $\texttt{elimit}_{\preccurlyeq}(F, \leq, I), A$*, représentant intuitivement le code de gestion des situations limites de l'itération.*

Pour représenter la post-condition d'une étape de boucle, nous définissons une famille $(J_{H,a})_{(H,a)\in\mathcal{P}(O)\times A}$ *de sous-ensembles de* G *par*

$$J_{H,a} = \{y \in I_a \mid F(a, y) \text{ majorant strict de } H\}$$

Alors la fonction $W^{A,\{0\}}_{iter}$ *définie par* $x_0 \in W^{A,\{0\}}_{iter}(\Gamma, (Q_0), a_0)$ *si et seulement si les trois conditions suivantes sont vérifiées :*

(i) $x_0 \in I_{a_0}$

(ii) Pour tout $a \in A$*, et pour tout* $x \in I_a$*, alors*

$$x \in W^{A,A}_{step}(\Gamma, (J_{\{F(a,x)\},a'})_{a'\in A}, a)$$

(iii) Pour toute situation limite étendue $(H, f_A, f_G) \in \texttt{elimit}_{\preccurlyeq}(F, \leq, I)$,

$$\sup f_G(H) \in W_{lim}^{\texttt{elimit}_{\preccurlyeq}(F,\leq,I),A}(\Gamma, (J_{H,a})_{a\in A}, (H, f_A, f_G)))$$

est un transformateur de post-conditions de type $A, \{0\}$*, appelé un* transformateur d'itération.

Démonstration. Par la règle de partition, il suffit de montrer que pour tout contexte étiqueté Γ, toute post-condition Q_0, et toutes valeurs d'entrée a_0, x_0 telles que $x_0 \in W_{iter}(\Gamma, (Q_0), a_0)$, l'énoncé donné par $\Gamma \vdash_{\preccurlyeq} \langle\{x_0\} \hookrightarrow Q_0\rangle$ est vérifié. Nous allons pour cela appliquer la règle d'itération, en employant l'ordre et l'invariant de boucle $\leq_{(F,\leq)}, (\hat{I}_{a,x})_{(a,x)\in A\times G}$ définissant les situations limites associées à $A, F, (O, \leq), I$. Nous utilisons l'ensemble vide comme post-condition. Comme la condition (i) nous donne $\hat{I}_{a_0,x_0} = \{x_0\}$, nous obtenons le résultat voulu en prenant (a_0, x_0) comme point de départ, par le biais de la règle de conséquence.

Vérifions maintenant les prémisses de la règle d'itération, en commençant par le cas régulier. Soient donc a, x quelconques et montrons la validité de l'énoncé

$$\Gamma \vdash_{\preccurlyeq} \left\langle \hat{I}_{a,x} \hookrightarrow \bigcup_{(a',y)>_{(F,\leq)}(a,x)} \hat{I}_{a',y} \right\rangle.$$

Nous pouvons supposer que $\hat{I}_{a,x} = \{x\}$ car sinon cet ensemble est vide, et l'énoncé suit alors directement de la règle d'identité. Dans ce cas, $x \in I_a$, donc par la condition (ii) et la règle de conséquence l'énoncé

$$\Gamma \vdash_{\preccurlyeq} \left\langle \{x\} \hookrightarrow \bigcup_{a'\in A} J_{\{F(a,x)\},a'} \right\rangle$$

est valide. Mais pour tout $a' \in A$, et $y \in J_{\{F(a,x)\},a'}$, nous avons $y \in \hat{I}_{a,y}$ et $(a', y) >_{(F,\leq)} (a, x)$ par définition. En particulier, il nous suffit d'appliquer la règle de conséquence pour obtenir l'énoncé voulu.

Enfin, vérifions le cas limite. Soit donc (H_0, f_0) une situation limite quelconque, et vérifions que l'énoncé

$$\Gamma \vdash_{\preccurlyeq} \left\langle \sup f_0(H_0) \hookrightarrow \bigcup_{(a,x)\geq_{(F,\leq)}H_0} \hat{I}_{a,x} \right\rangle$$

est bien valide. Le lemme 5.83 nous permet de réduire le problème à l'unique situation limite étendue (H, f_A, f_G) correspondante. Par la condition (iii), l'énoncé

$$\Gamma \vdash_{\preccurlyeq} \left\langle \{\sup f_G(H)\} \hookrightarrow \bigcup_{a\in A} J_{H,a} \right\rangle$$

est valide. Comme la pré-condition coïncide avec celle voulue, il nous suffit de vérifier l'inclusion des post-conditions. Pour cela, remarquons que pour tout $a \in A$ et $x \in J_{H,a}$, nous avons bien $x \in \hat{I}_{a,x}$. De plus, $F(a, x)$ est un majorant strict de H, et donc (a, x) est également un majorant strict de H_0 pour $\leq_{(F,\leq)}$, ce qui conclut. □

Remarquons que la règle de combinaison associée à l'itération telle qu'énoncée ci-dessus nous oblige à faire évoluer l'intégralité des paramètres pendant l'itération. Cependant, il est souvent désirable de fixer certains paramètres, de manière à garantir que ceux-ci restent constants durant l'itération. Nous pouvons obtenir une variante de la règle de combinaison pour l'itération permettant une telle partition des paramètres en employant le lemme 5.74, ce qui nous permet faire passer les paramètres constants à l'extérieur des transformateurs de post-conditions.

Définition 5.85 (Notation pour un transformateur d'itération)**.** Soient W_{step} et W_{lim} deux transformateurs nommés, $(O, \leq)$ un ordre, I une expression booléenne, V une expression à valeur dans O, $(x_i \in A_i)_{0\leq i<n}$ une liste ordonnée de noms distincts associés à des ensembles, et $H, f_G, (f_i)_{0\leq i<n}$ une liste ordonnée de noms distincts. Nous utilisons la notation

$$
\begin{array}{l}
\mathsf{iter}((x_i \in A_i)_{0\leq i<n}) \\
\mathsf{invariant}\{I\} \\
\mathsf{variant}\{V\} \\
\mathsf{progress}\{\leq\} \\
\quad W_{step} \\
\mathsf{diverges}[H, f_G, (f_i)_{0\leq i<n}] \\
\quad W_{lim} \\
\mathsf{end\text{-}iter}
\end{array}
$$

pour représenter la version nommée d'un transformateur d'itération. La correspondance se fait de la manière suivante :

- la famille $(x_i \in A_i)_{0\leq i<n}$ correspond aux paramètres auxiliaires de l'itération, liés dans les expressions et le transformateur W_{step} ;
- les valeurs de retours du transformateur d'itération sont donnés par une famille vide ;
- l'expression V correspond à la fonction de progression à valeur dans l'ensemble ordonné $(O, \leq)$;
- W_{step} correspond au transformateur d'étapes ;
- le triplet formé de $H \in \mathcal{P}(G)$, de $f_G \in G^H$, et de la fonction produit des $(f_i)_{0\leq i<n}$ (définie sur H à valeur dans $\prod_{0\leq i<n} A_i$) représente une situation limite, c'est-à-dire aux paramètres du transformateur de gestion des situations limites ;
- $H, f_G, (f_i)_{0\leq i<n}$ sont liés dans W_{lim} ;
- W_{lim} correspond au transformateur de gestion des situations limites.

Comme nous l'avons remarqué ci-dessus, nous pouvons gérer la présence de paramètres « globaux » vis-à-vis du transformateur d'itération par le biais de la construction du lemme 5.74. Cela nous force toutefois à ajuster les paramètres auxiliaires des transformateurs, comme dans le cas du second transformateur d'une liaison séquentielle, ce que nous faisons exactement de la même manière. La structure nommée du transformateur d'itération est également obtenue par les mêmes méthodes que pour les autres constructions, et est déterminée par les liaisons que nous avons énoncées ci-dessus.

Nous donnons maintenant une règle de combinaison correspondant à la récurrence. Cette règle correspond à la vérification d'une procédure récursive équipée d'un

contrat et suivie immédiatement de son appel, avec une fois de plus une contrainte de gestion des situations limites différente de la bonne fondation. Le fait que la procédure récursive soit directement appelée et non introduite dans le contexte vient de la structure de la règle de récurrence. Nous pouvons si nécessaire simuler la déclaration de procédure récursive par combinaison avec le transformateur d'introduction.

La preuve de la correction de la règle de combinaison récursive est très proche de la preuve effectuée pour l'itération, puisqu'il s'agit principalement de convertir le format des situations limites.

Lemme 5.86. *Supposons donnés*

- $(A, B, (P_a), (Q_{a,x,b}))$ *un contrat*
- *une fonction de progression* F *de* $A \times G$ *dans un ensemble ordonné* $(O, \leq)$
- *un transformateur de post-conditions* $W^{A,B}_{loc}$ *de type* A, B*, représentant intuitivement le code d'une fonction récursive*
- *un transformateur de post-conditions* $W^{\texttt{elimit}_{\preccurlyeq}(F,\leq,P),O\times B}_{lim}$ *dont le type est donné par* $\texttt{elimit}_{\preccurlyeq}(F, \leq, P), O \times B$*, représentant intuitivement le code de gestion d'un débordement de pile infini*
- *une fonction* $E(\Gamma, H)$ *associant à un contexte étiqueté* Γ *et un sous-ensemble de* O *le contexte* Γ *étendu par l'association d'un nom fixé* `R` *au contrat*

$$(A, B, (\{x \in P_a \mid F(a, x) \text{ majorant strict de } H\})_{a\in A}, (Q_{a,x,b}))$$

(une fois de plus, cela ne pose pas de problème si un ancien contrat est écrasé). Alors la fonction $W^{A,B}_{rec}$ *définie par* $x_0 \in W^{A,B}_{rec}(\Gamma, (Q'_b)_{b\in B}, a_0)$ *si et seulement si les trois conditions suivantes sont vérifiées*

(i) $x_0 \in P_{a_0}$ *et* $\forall b \in B.\ Q_{a_0,x_0,b} \subseteq Q'_b$

(ii) Pour tout $a \in A$ *et* $x \in P_a$*,*

$$x \in W^{A,B}_{loc}(E(\Gamma, \{F(a, x)\}), (Q_{a,x,b})_{b\in B}, a)$$

(iii) Pour toute situation limite étendue $(H, f_A, f_G) \in \texttt{elimit}_{\preccurlyeq}(F, \leq, P)$*,*

$$\sup f_G(H) \in W_{lim}(E(\Gamma, H), (\overline{Q}_{o,b})_{(o,b)\in O\times B}, (H, f_A, f_G))$$

où $\overline{Q}_{o,b}$ *est définie par*

$$\overline{Q}_{o,b} = \begin{cases} Q_{f_A(o),f_G(o),b} & \text{si} \quad o \in H \\ \emptyset & \text{si} \quad o \notin H \end{cases}$$

est un transformateur de post-conditions de type A, B*, appelé un* transformateur de récurrence*.*

Démonstration. Posons $(\leq_{F,\leq}, (\hat{P}_{a,x})_{(a,x)\in A\times G})$ l'ordre et la famille de sous-ensemble définissant les situations limites associées à $A, F, (O, \leq), P$. Nous allons obtenir le résultat en employant la règle de récurrence pour l'ordre $\leq_{(F,\leq)}$ et la pré-condition $\hat{P}_{a,x}$, ainsi que la post-condition $\hat{Q}_{a,x} = \bigcup_{b\in B} Q_{a,x,b}$.

Par la règle de partition, il suffit de vérifier que pour tout contexte étiqueté Γ, toute famille de post-conditions $(Q'_b)_{b\in B}$, et toutes valeurs (a_0, x_0) telles que $x_0 \in W^{A,B}_{rec}(\Gamma, (Q'_b)_{b\in B}, a_0)$, l'énoncé donné par $\Gamma \vdash_{\preccurlyeq} \langle\{x_0\} \hookrightarrow \bigcup_{b\in B} Q'_b\rangle$ est valide. En exploitant la condition (i), la règle de conséquence permet de se réduire à $\Gamma \vdash_{\preccurlyeq}$

$\left\langle \hat{P}_{a_0,x_0} \hookrightarrow \hat{Q}_{a_0,x_0} \right\rangle$, ce qui est la conclusion de l'instance proposée de la règle de récurrence pour la valeur (a_0, x_0).

Vérifions maintenant les prémisses de la règle de récurrence, en commençant par le cas régulier. Soient donc a, x quelconques et montrons la validité de l'énoncé

$$\Gamma, \left(\forall (a', y) >_{(F,\leq)} (a, x).\ \left\langle \hat{P}_{a,y} \hookrightarrow \hat{Q}_{a,y} \right\rangle\right) \vdash_{\preccurlyeq} \left\langle \hat{P}_{a,x} \hookrightarrow \hat{Q}_{a,y} \right\rangle$$

Nous pouvons supposer que $\hat{P}_{a,x} = \{x\}$, sans quoi cet ensemble est vide et la règle d'identité permet de conclure. Dans ce cas, $x \in P_a$, et par la condition (ii) combinée à la règle de conséquence l'énoncé

$$E(\Gamma, \{F(a, x)\}) \vdash_{\preccurlyeq} \left\langle \{x\} \hookrightarrow \hat{Q}_{a,x} \right\rangle$$

est valide. Comme la garantie finale est déjà celle voulue, il suffit d'appliquer la règle de remplacement pour transformer le contexte. Or, il suffit de dérouler la définition de l'hypothèse ajoutée par E et celle de $\leq_{(F,\leq)}$ pour vérifier que les garanties correspondantes appartiennent bien au contexte final voulu, et sont donc déductible par la règle d'axiome.

Il ne nous reste donc plus qu'à vérifier le cas limite. Soit donc H_0, f_0 une situation limite quelconque, et vérifions que l'énoncé

$$\Gamma, \left(\forall (a, x) \geq_{(F,\leq)} H_0.\ \left\langle \hat{P}_{a,x} \hookrightarrow \hat{Q}_{a,x} \right\rangle\right) \vdash_{\preccurlyeq} \left\langle \{\sup f_0(H_0)\} \hookrightarrow \bigcup_{(a,x)\in H_0} \hat{Q}_{a,x} \right\rangle$$

est valide. Une fois de plus, le lemme 5.83 nous permet de réduire le problème à l'unique situation limite étendue (H, f_A, f_G) correspondante. Par la condition (iii), l'énoncé

$$E(\Gamma, H) \vdash_{\preccurlyeq} \left\langle \{\sup f_G(H)\} \hookrightarrow \bigcup_{(o,b)\in O\times B} \overline{Q}_{o,b} \right\rangle$$

est alors valide. À cause des propriétés liant les deux types de situations limites, la garantie obtenue coïncide avec celle voulue. Il suffit donc une fois encore d'appliquer la règle de remplacement pour transformer le contexte. Le même raisonnement que celui effectué pour le cas régulier s'applique, et permet de conclure. □

Définition 5.87 (Notation pour un transformateur de récurrence)**.** Soit W_{loc} et W_{lim} deux transformateurs nommés, $(O, \leq)$ un ordre, $\mathcal{C}$ un contrat nommé, V une expression à valeur dans O, $\mathtt{F}$ un nom d'hypothèse, $(x_i \in A_i)_{0\leq i<n}$ une liste ordonnée de noms distincts, et $H, f_G, (f_i)_{0\leq i<n}$ une liste ordonnée de noms distincts. Nous utilisons la notation

call-rec $\mathtt{F}((x_i \in A_i)_{0\leq i<n})$: $\mathcal{C}$
variant$\{V\}$
progress$\{\leq\}$
  W_{loc}
diverges$(H, f_G, (f_i)_{0\leq i<n} \mid clos)$
  W_{lim}
end-call-rec

pour représenter le *transformateur de récurrence* de la procédure récursive définie par W_{loc}, de nom $\mathtt{F}$, avec paramètres d'appel $(x_i \in A_i)_{0\leq i<n}$, contrat $\mathcal{C}$, progression de V dans $(O, \leq)$, et W_{lim} pour la gestion des situations limites données par

$H, f_G, (f_i)_{0 \leq i < n}$. Le nom *clos* est fourni pour représenter la valeur de retour auxiliaire supplémentaire demandée par une situation limite, c'est-à-dire l'élément de H correspondant à la fonction que l'on a choisi de terminer. Ce nom doit être distinct des noms utilisés pour les valeurs de retour de $\mathcal{C}$. Nous avons également la contrainte que l'association de noms correspondant au paramètres de $\mathcal{C}$ doit coïncider avec l'association des x_i aux A_i.

Pour obtenir la version nommée du transformateur de récurrence, nous utilisons exactement les mêmes manipulations de paramètres auxiliaires que pour le cas de l'itération.

Comme pour le cas du transformateur d'introduction, nous autorisons l'omission des quantificateurs sur les paramètres auxiliaires au sein de $\mathcal{C}$ puisque celle-ci est explicite dans la notation.

5.4 Exemples d'application du système de preuve

Nous donnons maintenant quelques exemples d'utilisation effective du système pour obtenir des résultats sur un système de transition, ou sur un programme.

5.4.1 Jeu de multiplication

Nous reprenons ici l'exemple du jeu $\overline{\mathbb{N}}_\times$, et de l'énoncé proposé dans l'exemple 5.42, à savoir qu'il est toujours possible de garantir l'atteinte d'un état entier divisible par les n premiers entiers. Nous avions formalisé cet énoncé par l'énoncé nommé suivant :

$$\frac{\texttt{mult} : \forall k \in \mathbb{N}.\ \langle (k \neq 0 \wedge \mathsf{now} \neq \infty) \hookrightarrow (l \in \mathbb{N} :\ l \neq 0 \wedge \mathsf{now} = k \times l \times \mathsf{old}) \rangle}{\forall n \in \mathbb{N} : \langle (\mathsf{now} \neq \infty) \hookrightarrow (\mathsf{now} \neq \infty\ \wedge\ \forall k.\ 1 \leq k \leq n \Rightarrow k | \mathsf{now}) \rangle}$$

Nous avons démontré cet énoncé à l'aide des transformateurs, à partir de la factorielle. Cependant, nous pouvons également nous passer de cette factorielle, en utilisant un transformateur qui simule son calcul et démontre en même temps les propriétés que l'on souhaite.

La première manière d'obtenir ce résultat est essentiellement d'utiliser un transformateur correspondant au programme itératif d'une factorielle, en se servant de l'état du jeu comme accumulateur. Nous donnons un exemple de combinaison de transformateurs possible dans la figure 5.3. Hormis les variables introduites pour les situations limites, nous n'utilisons que des entiers comme variables, ce qui nous permet d'omettre les annotations d'appartenance des variables à $\mathbb{N}$. Nous utilisons comme invariant le fait que l'état soit divisible par tous les entiers par lesquels la multiplication a été effectuée. Nous définissons comme ordre de progression l'ordre inverse sur les entiers naturels, ce qui garantit l'absence de situation limite. Pour pouvoir sortir de la structure itérative en cours de route, nous exploitons le transformateur de continuation.

Nous pouvons alors vérifier la validité de l'énoncé voulu par application du transformateur, puis en démontrant la condition de vérification obtenue. Nous donnons la formule obtenue dans la figure 5.4. Remarquons qu'en distribuant les implications et quantificateurs universels, cette condition de vérification peut être décomposée en une conjonction de contraintes. Chaque contrainte obtenue correspond à l'établissement de la pré-condition d'un appel, d'un invariant de boucle, de la post-condition

$$\frac{\texttt{mult} : \forall k \in \mathbb{N}.\ \langle (k \neq 0 \wedge \mathsf{now} \neq \infty) \hookrightarrow (l \in \mathbb{N} :\ l \neq 0 \wedge \mathsf{now} = k \times l \times \mathsf{old}) \rangle}{\forall n \in \mathbb{N} : \langle (\mathsf{now} \neq \infty) \hookrightarrow (\mathsf{now} \neq \infty\ \wedge\ \forall k.\ 1 \leq k \leq n \Rightarrow k | \mathsf{now}) \rangle}$$

```
continuation K()
    let m ← n
    iter(m)
    invariant{0 ≤ m ≤ n ∧ now ≠ ∞ ∧ ∀k. m < k ≤ n ⇒ k|now}
    variant{m}
    progress{≥}
        switch
        case m = 0 :
            K
            choose m such that ⊥
        case m > 0 :
            mult[k ← m]
            let m ← m − 1
        end-switch
    diverges[H, f_G, f_m]
        choose m such that ⊥
    end-iter
end-continuation
```

FIGURE 5.3 – Exemple de transformateur itératif

finale ou de l'exhaustivité d'un test. Pour établir ces contraintes, nous disposons des informations déjà établies sur l'état au niveau de ce point, en particulier la pré-condition initale, les post-conditions des appels précédents, ou encore la validité des tests englobants. Dans ce cas précis, les contraintes obtenues sont faciles à vérifier.

Notons également que dans la formule de la figure 5.4, nous n'avons pas simplifié les formules contenant $\bot$ pour préserver la forme de la condition de vérification, en particulier les éléments introduits après l'appel à la continuation et le rétablissement de l'invariant dans la gestion des situations limites. Par la suite, nous effectuerons ces simplifications dans un souci de concision.

Comme notre système de preuve admet une règle de récurrence, nous pouvons également effectuer la démonstration de cet énoncé à l'aide d'une combinaison récursive de transformateurs. Dans ce cas, nous utilisons la combinaison donnée dans la figure 5.5, qui correspond essentiellement au code d'une implémentation récursive de la factorielle. Vis-à-vis de la structure itérative, l'invariant est simplement remplacé par un contrat. Ce changement rend les contraintes générées légèrement différentes, mais tout aussi faciles à vérifier.

5.4.2 Système de production de mots parenthésés

Nous considérons de nouveau l'exemple 4.83, portant sur un système de transition générant des mots bien parenthésés. Pour rappel, nous avions présenté le système de transition ordonné $\mathcal{S}_{()}$ dont les états sont formés d'un compteur et d'une séquence de parenthèses. Le compteur représente le nombre de parenthèses laissées

$$
\begin{array}{l}
\forall n \in \mathbb{N}, \mathsf{now} \in \overline{\mathbb{N}}.\ \mathsf{now} \neq \infty \Rightarrow \\
\left(\begin{array}{l}
(0 \leq n \leq n \ \wedge\ \mathsf{now} \neq \infty \ \wedge\ \forall k, n < k \leq n \Rightarrow k|\mathsf{now}) \\
\wedge \left(\begin{array}{l}
\forall m \in \mathbb{N}, \mathsf{now}_2 \in \overline{\mathbb{N}}. \\
(0 \leq m \leq n \ \wedge\ \mathsf{now}_2 \neq \infty \ \wedge\ \forall k.\ m < k \leq n \Rightarrow k|\mathsf{now}_2) \Rightarrow \\
\left(\begin{array}{l}
(m = 0 \vee m > 0) \\
\wedge \left(\begin{array}{l}
m = 0 \Rightarrow \\
\mathsf{now}_2 \neq \infty \\
\wedge\ (\forall k.\ 1 \leq k \leq n \Rightarrow k|\mathsf{now}_2) \\
\wedge \left(\begin{array}{l}
\forall \mathsf{now}_3.\ \bot \Rightarrow \\
(\exists m' \in \mathbb{N}.\ \bot) \\
\wedge \left(\begin{array}{l}
\forall m' \in \mathbb{N}.\ \bot \Rightarrow \\
0 \leq m' \leq n \wedge \mathsf{now}_3 \neq \infty \\
\wedge\ (\forall k.\ m' < k \leq n \Rightarrow k|\mathsf{now}_3) \\
\wedge\ m > m'
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
\wedge \left(\begin{array}{l}
m > 0 \Rightarrow \\
(m \neq 0 \ \wedge\ \mathsf{now}_2 \neq \infty) \\
\wedge \left(\begin{array}{l}
\forall l \in \mathbb{N}, \mathsf{now}_3 \in \overline{\mathbb{N}}. \\
(l \neq 0 \wedge \mathsf{now}_3 = k \times l \times \mathsf{now}_2) \Rightarrow \\
0 \leq m - 1 \leq n \wedge \mathsf{now}_3 \neq \infty \\
\wedge\ (\forall k.\ m - 1 < k \leq n \Rightarrow k|\mathsf{now}_3) \\
\wedge\ m > m - 1
\end{array} \right)
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
\wedge \left(\begin{array}{l}
\forall (H, f_m, f_G) \in \mathtt{elimit}_{\geq}(((m, \_) \Rightarrow m), \geq, (I_m)_{m \in \mathbb{N}}). \\
f_G(H) \text{ a une borne supérieure } \mathsf{now}_2 \Rightarrow \\
(\exists m' \in \mathbb{N}.\ \bot) \\
\wedge \left(\begin{array}{l}
\forall m' \in \mathbb{N}.\ \bot \Rightarrow \\
0 \leq m' \leq n \wedge \mathsf{now}_2 \neq \infty \\
\wedge\ (\forall k.\ m' < k \leq n \Rightarrow k|\mathsf{now}_2) \\
\wedge\ (\forall x \in H.\ f_m(x) > m')
\end{array} \right) \\
\text{où } I_m = \left\{ \mathsf{now}_3 \in \overline{\mathbb{N}} \;\middle|\; \begin{array}{l} 0 \leq m \leq n \wedge \mathsf{now}_3 \neq \infty \\ \wedge\ \forall k.\ m < k \leq n \Rightarrow k|\mathsf{now}_3 \end{array} \right\}
\end{array} \right)
\end{array} \right)
\end{array}
$$

FIGURE 5.4 – Condition de vérification associée au transformateur itératif

$$\frac{\texttt{mult} : \forall k \in \mathbb{N}.\ \langle (k \neq 0 \wedge \mathsf{now} \neq \infty) \hookrightarrow (l \in \mathbb{N} :\ l \neq 0 \wedge \mathsf{now} = k \times l \times \mathsf{old}) \rangle}{\forall n \in \mathbb{N} : \langle (\mathsf{now} \neq \infty) \hookrightarrow (\mathsf{now} \neq \infty\ \wedge\ \forall k.\ 1 \leq k \leq n \Rightarrow k|\mathsf{now}) \rangle}$$

call-rec $F(n)$: $\langle (0 \leq n \wedge \mathsf{now} \neq \infty) \hookrightarrow (\mathsf{now} \neq \infty \wedge \forall k.\ 1 \leq k \leq n \Rightarrow k|\mathsf{now}) \rangle$
variant$\{n\}$
progress$\{\geq\}$
  switch
  case $n = 0$:
    skip
  default :
    $F[n \leftarrow n-1]$
    $\texttt{mult}[k \leftarrow n]$
    skip
  end-switch
diverges$(H, f_G, f_n \mid clos)$
  absurd
end-call-rec

Résultat de l'application :

$$\begin{array}{l}
\forall n \in \mathbb{N}, \mathsf{now} \in \overline{\mathbb{N}}.\ \mathsf{now} \neq \infty \Rightarrow \\
\left(\begin{array}{l}
0 \leq n\ \wedge\ \mathsf{now} \neq \infty\ \wedge \\
\left(\begin{array}{l} \forall \mathsf{now}_2 \in \overline{\mathbb{N}}.\ \mathsf{now}_2 \neq \infty\ \wedge\ (\forall k.\ 1 \leq k \leq n \Rightarrow k|\mathsf{now}_2) \Rightarrow \\ \mathsf{now}_2 \neq \infty\ \wedge\ \forall k.\ 1 \leq k \leq n \Rightarrow k|\mathsf{now}_2 \end{array} \right) \\
\wedge \left(\begin{array}{l}
\forall n \in \mathbb{N}, \mathsf{now}_2 \in \overline{\mathbb{N}}.\ (0 \leq n\ \wedge\ \mathsf{now}_2 \neq \infty) \Rightarrow \\
(n = 0 \vee n \neq 0) \\
\wedge\ (n = 0 \Rightarrow \mathsf{now}_2 \neq \infty\ \wedge\ \forall k.\ 1 \leq k \leq 0 \Rightarrow k|\mathsf{now}_2) \\
\wedge \left(\begin{array}{l} n \neq 0 \Rightarrow 0 \leq (n-1)\ \wedge \mathsf{now}_2 \neq \infty\ \wedge\ n > n-1 \\
\wedge \left(\begin{array}{l} \forall \mathsf{now}_3 \in \overline{\mathbb{N}}. \\ \left(\begin{array}{l} \mathsf{now}_3 \neq \infty \\ \wedge\ \forall k.\ 1 \leq k \leq n-1 \Rightarrow k|\mathsf{now}_3 \end{array} \right) \Rightarrow \\ n \neq 0\ \wedge \mathsf{now}_3 \neq \infty \\ \wedge\ \forall l \in \mathbb{N}, \mathsf{now}_4 \in \overline{\mathbb{N}}. \\ l \neq 0\ \wedge\ \mathsf{now}_4 = n \times l \times \mathsf{now}_3 \Rightarrow \\ \mathsf{now}_4 \neq \infty\ \wedge\ \forall k.\ 1 \leq k \leq n \Rightarrow k|\mathsf{now}_4 \end{array} \right) \end{array} \right) \end{array} \right) \\
\wedge \left(\begin{array}{l} \forall (H, f_n, f_G) \in \texttt{elimit}_{\geq}(((n, _) \Rightarrow n), \geq, (P_n)_{n \in \mathbb{N}}). \\ f_G(H) \text{ a une borne supérieure } \mathsf{now}_2 \Rightarrow \bot \\ \text{où } P_n = \{\mathsf{now}_3 \in \overline{\mathbb{N}} \mid 0 \leq n \wedge \mathsf{now}_3 \neq \infty\} \end{array} \right)
\end{array} \right)
\end{array}$$

FIGURE 5.5 – Exemple de transformateur récursif et condition de vérification associée

$$\texttt{print}(a \in \{(,)\}) : \left\langle \begin{pmatrix} \mathsf{now} \text{ s'écrit } ((n,w),t) \\ \text{et } n = 0 \Rightarrow a = \text{'('} \end{pmatrix} \hookrightarrow \begin{pmatrix} \mathsf{now} = ((n+v, wa), t+1) \\ \text{si } \mathsf{old} \text{ s'écrit } ((n,w),t) \\ \text{et } v = 1 \text{ si } a = \text{'('}, -1 \text{ sinon} \end{pmatrix} \right\rangle$$

$$\forall w \in W : \left\langle (\mathsf{now} \text{ s'écrit } ((0,\epsilon),t)) \hookrightarrow \begin{pmatrix} \mathsf{now} \text{ de la forme } ((n,w),t) \\ \text{ou bien } \mathsf{now} = w \end{pmatrix} \right\rangle$$

$$\texttt{scan} : \left\langle (\mathsf{now} \text{ s'écrit } ((n,w),t)) \hookrightarrow \begin{pmatrix} a \in \{(,)\} : & \mathsf{now} = ((n+v, wa), t+1) \\ & \text{si } \mathsf{old} \text{ s'écrit } ((n,w),t) \\ & \text{et } v = 1 \text{ si } a = \text{'('}, -1 \text{ sinon} \\ & \text{et } n = 0 \Rightarrow a = \text{'('} \end{pmatrix} \right\rangle$$

$$\forall l \in \overline{\mathbb{N}} : \left\langle (\mathsf{now} \text{ s'écrit } ((0,\epsilon),t)) \hookrightarrow \begin{pmatrix} \mathsf{now} \in W_l \\ \text{ou bien } \mathsf{now} \text{ s'écrit } ((n,w),t) \\ \text{avec } w \in W_l \end{pmatrix} \right\rangle$$

FIGURE 5.6 – Énoncés nommés pour la correction du système producteur de parenthèses

ouvertes dans la séquence. Nous avions affirmé que les mots atteignables par le système, ou qui sont borne supérieure d'une trace d'évolution infinie, étaient précisément les mots appartenant à un ensemble W, défini comme le plus grand ensemble de mots finis ou infinis correspondant à la grammaire :

$$W ::= \epsilon \mid (W)W \mid (W$$

Nous allons maintenant démontrer ce résultat. Pour cela, nous avions déjà démontré qu'il suffit de démontrer les garanties :

$$\forall w \in W.\ \langle \{((0,\epsilon),t) \mid t \in \mathbb{N}\} \hookrightarrow \{((n,w),t) \mid n,t \in \mathbb{N}\} \cup \{w\} \rangle_{\mathbb{G}^l_{\mathcal{S}_{()},\exists}}$$

et

$$\forall l \in \overline{\mathbb{N}}.\ \langle \{((0,\epsilon),t) \mid t \in \mathbb{N}\} \hookrightarrow \{((n,w),t) \mid w \in W_l\} \cup W_l \rangle_{\mathbb{G}^l_{\mathcal{S}_{()},\forall}}$$

où W_l est l'ensemble des mots de longueur l (potentiellement infinie).

Nous allons utiliser les transformateurs définis dans la section 5.3. Pour cela, nous devons préalablement réduire la preuve de ces deux garanties vers la preuve d'énoncés. Par définition de la validité d'un énoncé, il nous suffit de trouver des contextes $\Gamma_{\mathcal{S}_{()},\exists}$ et $\Gamma_{\mathcal{S}_{()},\forall}$ qui permettent de déduire ces garanties, et qui sont satisfaits par les deux traductions locales respectives de $\mathcal{S}_{()}$. Pour cela, nous prenons comme contextes les garanties correspondant à la traduction des transitions de $\mathcal{S}_{()}$. Nous obtenons ainsi les énoncés nommés donnés dans la figure 5.6.

Remarquons que les contrats donnés dans les contextes $\Gamma_{\mathcal{S}_{()},\exists}$ et $\Gamma_{\mathcal{S}_{()},\forall}$ possèdent presque les mêmes pré- et post-conditions. La différence vient principalement du rôle de la variable a, qui joue dans un cas le rôle de paramètre et dans l'autre le rôle de valeur de retour. Cette différence correspond à l'interprétation du non-déterminisme, qui est encodé par la valeur de a.

Lemme 5.88. *Les garanties correspondant aux deux contextes choisis sont toutes satisfaites par les jeux correspondants aux traductions locales respectives du système de transition ordonné* $\mathcal{S}_{()}$.

Démonstration. Application directe des lemmes 4.80 et 4.82 qui permettent de traduire les garanties sur ces jeux en des propriétés sur des traces. Dans le cas existentiel, il suffit de construire une trace à deux éléments correspondant à la transition du système pour montrer l'accessibilité. Dans le cas universel, il suffit de faire une distinction de cas sur la trace. Soit la trace ne contient qu'un élément, et la pré-condition permet de la prolonger, soit elle en contient au moins deux, et par définition des transitions le second appartient à la post-condition. □

Remarque 5.89. Le schéma de conversion proposé pour transformer une traduction locale de $\mathcal{S}_{()}$ en un contexte se généralise sans difficulté à n'importe quel système de transition ordonné. Nous pouvons remplacer la traduction existentielle par une hypothèse correspondant à la simulation d'une transition passée en paramètre. De même, nous pouvons remplacer la traduction universelle par une hypothèse dont la pré-condition est qu'au moins une transition soit possible depuis l'état courant, et dont la post-condition est qu'une transition a été effectuée.

Nous allons maintenant établir ces deux énoncés. Nous commençons par la version existentielle du résultat. Nous allons simplement appliquer le transformateur nommé donné dans la figure 5.7. Ce transformateur, de nature récursive, consiste simplement à dérouler la structure arborescente de l'élément que l'on souhaite obtenir. Cette structure est essentiellement contenu dans la condition d'appartenance à W. En fait, le transformateur correspond tout simplement à une routine d'impression pour les mots de W depuis leur description sous forme d'arbre de syntaxe. Cette structure est implicitement contenue dans la condition d'appartenance à W. Nous simulons l'analyse d'un nœud en testant lequel des trois cas de la grammaire correspond au mot courant.

Nous donnons également la condition de vérification dans la figure 5.8, après déroulement des définitions des transformateurs. Pour des raisons de lisibilité, nous avons simplifié les conditions de forme de l'état, éliminé les quantificateurs inutiles et éliminé les contraintes redondantes générées par la composition d'un branchement et de choix validés par le test de la branche courante. Comme la plupart des contraintes obtenues sont complètement évidentes, nous nous focalisons sur le seul cas non trivial, à savoir la gestion des situations limites (H, f_G, f_w). Notons qu'il s'agit également du seul cas à ne pas apparaître dans le calcul de plus faible pré-condition traditionnel.

Étant donnée une situation limite, nous déduisons de sa structure que les mots des états associées forment une famille de préfixes du mot w_0 que l'on cherche à obtenir, de longueur non bornée. En particulier, la borne supérieure w_∞ de $f_G(H)$ est nécessairement un état limite, car le mot associé ne peut pas être fini. Mais comme tous les préfixes finis de w_0 sont préfixes de n'importe quel mot plus long dans $f_G(H)$, et donc également préfixe de w_∞, nous obtenons que w_0 est un préfixe infini de w_∞. Cela revient à dire que $w_\infty = w_0$.

Pour vérifier l'énoncé de nature universelle, nous allons naturellement nous inspirer de l'opération duale de l'impression, à savoir l'analyse syntaxique. Plus précisément, nous allons nous inspirer de la procédure récursive consistant à lire un

$$\left[\begin{array}{l} \texttt{print} : \left[\ a \in \{(,)\}\ \right] \Rightarrow [] \\ w \in W \end{array}\right] \Rightarrow []$$

let $w_0 \leftarrow w$
continuation $K()$
call-rec $F(w \in W)$: $\left\langle \left(\begin{array}{l}\text{now s'écrit } ((n, w_c), t) \\ \text{et } w_c w \text{ préfixe de } w_0\end{array}\right) \hookrightarrow \left(\begin{array}{l}\text{now s'écrit } ((n, w_c w), t') \\ \text{si old s'écrit } ((n, w_c), t)\end{array}\right)\right\rangle$
variant$\{|w_c| \text{ si now s'écrit } ((n, w_c), t)\}$
progress$\{\leq\}$
switch
case exist $w_1 \in W, w_2 \in W$ such that $w = \text{'('}w_1\text{')'}w_2$:
$\texttt{print}[a \leftarrow \text{'('}]$
$F[w \in W \leftarrow w_1]$
$\texttt{print}[a \leftarrow \text{')'}]$
$F[w \in W \leftarrow w_2]$
case exist $w_1 \in W$ such that $w = \text{'('}w_1$:
$\texttt{print}[a \leftarrow \text{'('}]$
$F[w \in W \leftarrow w_1]$
default :
skip
end-switch
diverges$(H, f_G, f_w \mid clos)$
K
absurd
end-call-rec
end-continuation

FIGURE 5.7 – Transformateur pour l'accessibilité des mots bien parenthésés

mot bien parenthésé suivi d'une parenthèse fermante excédentaire, ou bien un mot parenthésé suivi d'une fin de fichier. Ce marqueur de fin de fichier se trouve ici après le l-ième caractère lu, où l est la longueur donnée en paramètre. Nous donnons le squelette du transformateur nommé correspondant dans la figure 5.9, où nous avons noté le contrat de la procédure récursive par $\mathcal{C}$ et l'ordre de progression par $\preccurlyeq_w$.

Pour le contrat, nous donnons des pré- et post-conditions correspondant au comportement annoncé de l'analyseur syntaxique récursif, à savoir la lecture correcte jusqu'à la fin du fichier, ou la lecture jusqu'à atteindre une parenthèse excédentaire.

Définition 5.90. Pour $l \in \overline{\mathbb{N}}$, nous définissons le contrat nommé $\mathcal{C}_l$ de la manière suivante :

(A) aucun paramètre auxiliaire
(B) les valeurs de retour auxiliaires $b \in \{\bot, \top\}$ et $w \in W$. Le booléen indique si l'on a atteint ou non la fin de la lecture, et le mot indique ce qui a été reconnu par la procédure.
(P) la pré-condition constituée des états de la forme $((n, w_c), t)$ tels que $|w_c| \leq l$.
(Q) la post-condition constituée des états de la forme $((n', w'_c), t')$ tels que, si

$$
\left(\begin{array}{l}
\forall w_0 \in W. \\
\epsilon w_0 \text{ préfixe de } w_0 \\
\wedge w_0 = \epsilon w_0 \\
\wedge \left(\begin{array}{l}
\forall w \in W, n, w_c.\ w_c w \text{ préfixe de } w_0 \Rightarrow \\
(\exists w_1, w_2 \in W.\ w = \text{'('}w_1\text{')'}w_2) \vee (\exists w_1.\ w = \text{'('}w_1) \vee (w = \epsilon) \\
\wedge \left(\begin{array}{l}
\forall w_1, w_2 \in W.\ w = \text{'('}w_1\text{')'}w_2 \Rightarrow \\
(n = 0 \Rightarrow \text{'('} = \text{'('}) \\
\wedge\ w_c\text{'('} \text{ préfixe de } w_0 \wedge\ |w_c| < |w_c\text{'('}| \\
\wedge\ (n+1 = 0 \Rightarrow \text{')'} = \text{'('}) \\
\wedge\ w_c\text{'('}w_1\text{')'} \text{ préfixe de } w_0 \wedge\ |w_c| < |w_c\text{'('}w_1\text{')'}| \\
\wedge\ w_c w = w_c\text{'('}w_1\text{')'}w_2
\end{array}\right) \\
\wedge \left(\begin{array}{l}
\forall w_1 \in W.\ w = \text{'('}w_1 \Rightarrow \\
(n = 0 \Rightarrow \text{'('} = \text{'('}) \\
\wedge\ w_c\text{'('} \text{ préfixe de } w_0 \wedge\ |w_c| < |w_c\text{'('}| \\
\wedge\ w_c w = w_c\text{'('}w_1
\end{array}\right) \\
\wedge \left(\ w = \epsilon \Rightarrow w_c w = w_c \epsilon\ \right)
\end{array}\right) \\
\wedge \left(\begin{array}{l}
\forall (H, f_G, f_w) \in \texttt{elimit}(((n, \_) \Rightarrow n), \leq, (P)_{l \in \mathbb{N}}). \\
f_G(H) \text{ a une borne supérieure } \mathsf{now} \Rightarrow \\
\mathsf{now} = w \vee \exists n, t.\ \mathsf{now} = ((n, w), t) \\
\text{où } P_l = \{((n, w_c), t) \mid w_c w \text{ préfixe de } w_0 \wedge |w_c| = l\}
\end{array}\right)
\end{array}\right)
$$

FIGURE 5.8 – Condition de vérification pour l'accessibilité des mots bien parenthésés

$b = \top$, alors $|w'_c| = l$ et $w'_c = w_c w$, sinon dans le cas où $b = \bot$, alors $n' = n-1$ et $w'_c = w_c w\text{')'}$, avec $|w'_c| \leq l$.

Pour l'ordre de progression, nous devons faire un choix très précis. En effet, nous devons être capable de reconstruire une dérivation de l'appartenance à W pour le mot limite. Comme celle-ci est infinie, nous avons a priori besoin d'effectuer un raisonnement coinductif. Nous choisissons donc un ordre qui explicite la règle de dérivation inversée entre chaque appel.

Définition 5.91. Nous définissons l'ordre de progression $\preccurlyeq_w$ entre deux mots comme la clôture réflexive transitive de la relation suivante, dite de liaison en une étape. Nous disons que w_1 et w_2 sont *liés en une étape* si soit $w_2 = w_1\text{'('}$, soit il existe $w \in W$ tel que $w_2 = w_1\text{'('}w\text{')'}$.

Nous obtenons alors la condition de vérification donnée dans la figure 5.10 par application du transformateur, puis quelques simplifications mineures pour éliminer les conditions de forme de l'état ainsi que des tautologies. La condition de vérification est assez imposante, même sous forme simplifiée, mais la majorité des contraintes sont complètement évidentes. Nous nous focalisons donc une fois de plus sur la seule contrainte non triviale, à savoir la gestion des situation limites.

Étant donné une situation limite donnée par (H, f_G), remarquons tout d'abord que l'ordre $\preccurlyeq_w$ impose une croissance stricte de la longueur du mot, et donc que la borne supérieure de $f_G(H)$ est forcément un mot infini w_∞. La validité de la pré-condition sur l'historique montre alors que la longueur est également infinie. Il nous suffit donc de vérifier que $w_\infty \in W$.

$$\left[\begin{array}{l} \texttt{scan} : [] \Rightarrow \left[\ a \in \{(,)\}\ \right] \\ l \in \overline{\mathbb{N}} \end{array}\right] \Rightarrow []$$

continuation $K()$
  call-rec $F() : \mathcal{C}_l$
  variant$\{w_c$ où now s'écrit $((n, w_c), t)\}$
  progress$\{\preccurlyeq_w\}$
    switch
    case $|w_c| = l$ où now s'écrit $((n, w_c), t)$:
      let $b \in \{\bot, \top\} \leftarrow \top, w \in W \leftarrow \epsilon$
    default :
      `scan`
      switch
      case $a =$ ')' :
        let $b \in \{\bot, \top\} \leftarrow \bot, w \in W \leftarrow \epsilon$
      case $a =$ '(' :
        `F`
        switch
        case $b = \top$:
          let $b \leftarrow b, w \leftarrow$ '('w
        case $b = \bot$:
          let $w_0 \in W \leftarrow w$
          `F`
          let $b \leftarrow b, w \leftarrow$ '('w_0')'w
        end-switch
      end-switch
    end-switch
  diverges$(H, f_G \mid clos)$
    `K`
    choose $b \in \{\bot, \top\}, w \in W$ such that $\bot$
  end-call-rec
end-continuation

FIGURE 5.9 – Transformateur pour vérifier que tout mot atteint est correctement parenthésé

$$
\left(\begin{array}{l}
\forall l \in \overline{\mathbb{N}}.\ |\epsilon| \leq l \wedge \left(\begin{array}{l} \forall n \in \mathbb{N}, b \in \{\bot, \top\}, w \in W, w_c. \\ \left(\begin{array}{l} (b = \top \Rightarrow |w_c| = l \wedge w_c = \epsilon w) \\ \wedge\ (b = \bot \Rightarrow |w_c| \leq l \wedge n = 0 - 1 \wedge w_c = \epsilon w')') \end{array}\right) \Rightarrow \\ w_c \in W \wedge |w_c| = l \end{array}\right) \\
\wedge \left(\begin{array}{l} \forall n \in \mathbb{N}, w_1.\ |w_1| \leq l \Rightarrow (|w_1| = l \vee |w_1| \neq l) \\ \wedge\ (|w_1| = l \Rightarrow |w_1| = l \wedge w_1 = w_1\epsilon) \\ \wedge \left(\begin{array}{l} |w_1| \neq l \Rightarrow \forall a \in \{(,)\}, m \in \mathbb{N}, w_2. \\ \left(\begin{array}{l} w_2 = w_1 a \\ \wedge\ m = n + (\text{si } a = \text{'(' alors 1 sinon } (-1)) \\ \wedge\ (n = 0 \Rightarrow a = \text{'('}) \end{array}\right) \Rightarrow \\ (a = \text{'('} \vee a = \text{')'}) \\ \wedge\ (a = \text{')'} \Rightarrow m = n - 1 \wedge |w_c| \leq l \wedge \exists w \in W. w_2 = w_1 w')') \\ \wedge \left(\begin{array}{l} a = \text{'('} \Rightarrow \\ |w_2| \leq l \wedge w_1 \preccurlyeq_w w_2 \\ \wedge \left(\begin{array}{l} \forall b \in \{\bot, \top\}, o \in \mathbb{N}, w_0 \in W, w_3. \\ \left(\begin{array}{l} \left(\ b = \top \Rightarrow |w_3| = l \wedge w_3 = w_2 w\ \right) \\ \wedge \left(\begin{array}{l} b = \bot \Rightarrow \\ |w_3| \leq l \wedge o = m - 1 \\ \wedge\ w_3 = w_2 w')' \end{array}\right) \end{array}\right) \Rightarrow \\ (b = \top \vee b = \bot) \\ \wedge \left(\ b = \top \Rightarrow |w_3| = l \wedge w_3 = w_1\text{'('}w\ \right) \\ \wedge \left(\begin{array}{l} b = \bot \Rightarrow \\ |w_3| \leq l \wedge w_3 \preccurlyeq_w w_1 \\ \wedge \left(\begin{array}{l} \forall c \in \{\bot, \top\}, p \in \mathbb{N}, w \in W, w_4. \\ \left(\begin{array}{l} \left(\begin{array}{l} c = \top \Rightarrow \\ |w_4| = l \\ \wedge\ w_4 = w_3 w \end{array}\right) \\ \wedge \left(\begin{array}{l} c = \bot \Rightarrow \\ |w_4| \leq l \\ \wedge\ p = o - 1 \\ \wedge\ w_4 = w_3 w')' \end{array}\right) \end{array}\right) \Rightarrow \\ \left(\begin{array}{l} c = \top \Rightarrow \\ |w_4| = l \\ \wedge\ w_4 = w_1\text{'('}w_0\text{')'}w \end{array}\right) \\ \wedge \left(\begin{array}{l} c = \bot \Rightarrow \\ |w_4| \leq l \\ \wedge\ p = n - 1 \\ \wedge\ w_4 = \text{'('}w_0\text{')'}w')' \end{array}\right) \end{array}\right) \end{array}\right) \end{array}\right) \end{array}\right) \\
\wedge \left(\begin{array}{l} \forall (H, f_G, f_0) \in \texttt{elimit}(((0, ((n, w), t)) \Rightarrow w), \preccurlyeq_w, P). \\ f_G(H) \text{ a une borne supérieure } \mathsf{now} \Rightarrow \\ \mathsf{now} \in W_l \vee (\exists n, w, t. \mathsf{now} = ((n, w), t) \wedge w \in W_l) \\ \text{où } P_w = \{((n, w), t) \mid n, t \in \mathbb{N}, |w| \leq l\} \end{array}\right)
\end{array}\right)
$$

FIGURE 5.10 – Condition de vérification pour montrer que tout mot atteint est correctement parenthésé

Pour cela, nous remarquons que nous pouvons déduire de la validité d'une relation d'ordre stricte une inversion des règles de définition de W pour un suffixe de w_∞. Plus précisément, nous avons la propriété suivante.

Lemme 5.92. *Pour tout w préfixe de w_∞, posons $s(w)$ le suffixe de w_∞ tel que $ws(w) = w_\infty$. Alors pour tous w_1, w_2 préfixes distincts de w_∞ tels que $w_1 \preccurlyeq_w w_2$, il existe w_3 préfixe de w_∞ tel que $w_3 \preccurlyeq_w w_2$ et tel que soit $s(w_1) = \text{'('}s(w_3)$, soit il existe $w \in W$ tel que $s(w_1) = \text{'('}s(w_3)\text{')'}W$.*

Démonstration. Premièrement, remarquons que l'ordre $\preccurlyeq_w$ est un sous-ensemble de l'ordre préfixe sur les mots. En effet, deux mots liés en une étape sont en relation préfixe. Comme la relation préfixe est déjà réflexive et transitive, par minimalité de la clôture transitive, l'ordre $\preccurlyeq_w$ est bien un sous-ensemble de l'ordre préfixe. En particulier, cela justifie le caractère antisymétrique de $\preccurlyeq_w$.

Considérons maintenant w_1, w_2 deux préfixes distincts de w_∞ tels que $w_1 \preccurlyeq_w w_2$. Par définition de la clôture réflexive transitive, il existe donc une séquence de liaisons en une étape permettant de passer de w_1 à w_2. Comme $w_1 \neq w_2$, celle-ci est non vide, et soit alors w_3 le mot obtenu après la première étape depuis w_1. Nous avons donc $w_3 \preccurlyeq_w w_2$ par définition. En particulier, w_3 est un préfixe de w_2, et donc de w_∞. De plus, w_1 et w_3 sont liés en une étape. Dans le cas $w_3 = w_1$'(', nous obtenons directement $s(w_1) = \text{'('}s(w_3)$, et dans l'autre cas, nous obtenons précisément l'autre inversion possible. □

Ce lemme d'inversion nous permet d'effectuer un raisonnement par coinduction pour l'ensemble des mots de la forme $s(w)$, où w est un mot inférieur par $\preccurlyeq_w$ à un élément de H. Comme H n'admet pas de maximum, nous pouvons systématiquement appliquer le lemme d'inversion en prenant un élément supérieur. L'union A des mots de la forme choisie et de W est donc contenue dans l'ensemble des éléments obtenus par l'application de ces règles de propagation à A. Comme W est le plus grand ensemble à satisfaire ces propriétés (autrement dit par coinduction), $A = W$.

Cependant, nous sommes incapables de conclure. En effet, nous n'avons aucun moyen de montrer que w_∞ est bien de la forme choisie, autrement dit que ϵ soit inférieur par $\preccurlyeq_w$ à un élément de H. Nous ne pouvons donc pas démontrer la condition de vérification telle quelle. Par contre, nous pouvons modifier légèrement le transformateur en ajoutant $\epsilon \preccurlyeq_w w_c$ à la pré-condition récursive. Nous récupérons alors l'information manquante depuis la pré-condition de n'importe quel habitant de H.

Nous devons maintenant vérifier que l'ajout de cette nouvelle pré-condition ne pose pas de problème. Nous l'établissons immédiatement au début de la récurrence puisque dans ce cas $w_c = \epsilon$, et nous la propageons facilement par transitivité pour la pré-condition de chaque appel récursif. En effet, si w_c et w'_c sont les mots correspondant respectivement à la valeur du mot au début d'un appel récursif englobant, et w'_c à la valeur du mot au début d'un appel récursif immédiatement en dessous, alors $\epsilon \preccurlyeq_w w_c$ est garanti par la pré-condition de l'appel englobant, et $w_c \preccurlyeq_w w'_c$ doit de toute manière être établi pour pouvoir effectuer l'appel au niveau de w'_c.

Remarque 5.93. La construction que nous venons juste d'effectuer se généralise directement à n'importe quelle transformateur récursif ou itératif, et permet d'imposer que l'habitant garanti d'une situation limite corresponde en fait à l'élément initial de l'itération/la construction récursive. Il suffit en effet d'ajouter à l'invariant/à la

pré-condition du contrat récursif la condition que l'élément courant soit plus grand que l'élément initial. Nous pouvons alors construire la situation limite en question. De plus, nous pouvons reconstruire les contraintes habituelles pour les autres cas. Dans le cas de la récurrence, nous pouvons éliminer cet ajout de l'hypothèse de récurrence via le transformateur d'introduction, car cette nouvelle contrainte est déductible de la contrainte de progression de l'appel récursif. Nous pouvons effectuer le même type de construction au niveau des transformateurs itératifs pour éliminer la nouvelle contrainte de l'invariant.

Cette remarque suggère donc de raffiner certains transformateurs a posteriori pour obtenir des transformateurs plus puissants, notamment celui que nous venons de proposer. Pour cela, nous les construisons tout d'abord en termes des transformateurs déjà définis, puis nous éliminons de leurs définitions les contraintes vérifiables dans le cas général. Nous pourrions de même ajouter des constructions spécialisées pour les cas bien fondés ou dénombrables pour l'itération et la récurrence.

5.4.3 Preuve de programmes de nature dérécursifiée

Comme le système de preuve que nous avons obtenu n'est pas dirigé par la syntaxe, nous pouvons utiliser les transformateurs pour créer une structure de contrôle arbitraire au-dessus d'un programme à vérifier. Cela revient en pratique à établir une simulation entre deux programmes, en utilisant la structure donnée au transformateur pour la vérification plutôt que celle du programme de départ. Nous pouvons également voir cette méthode comme une généralisation de la méthode des paramètres auxiliaires/fantômes, consistant à équiper un programme avec un flot de contrôle supplémentaire dédié à la vérification. Nous remarquons que c'est particulièrement utile pour vérifier des programmes itératifs qui simulent une procédure récursive.

En effet, notre système de preuve nous permet de plaquer la structure récursive d'un transformateur sur le programme itératif, et ensuite de prouver le programme au travers de la structure récursive. Nous pourrions bien entendu effectuer la preuve du programme itératif en respectant sa structure de contrôle, mais pour cela nous devons typiquement écrire un invariant sur une pile auxiliaire, laquelle est en général très proche de l'état du programme récursif. Cette méthode revient en fait à suivre le cheminement inverse, à savoir prouver la version récursive du programme au travers d'un prisme itératif. Cependant, la gestion de la pile auxiliaire peut poser des difficultés supplémentaires pour la preuve, et l'invariant sur la pile n'est pas toujours évident à écrire. Trouver une quantité simple qui décroisse strictement peut également s'avérer impossible. Par exemple, les algorithmes de type Schorr-Waite [76] posent ce genre de problèmes.

Pour démontrer cette méthode de preuve, nous nous intéressons à un cas très simple, une version itérative de la fonction 91 de McCarthy [63]. La fonction 91 de McCarthy est la fonction f_{91} de $\mathbb{N}$ dans $\mathbb{N}$ définie par

$$f_{91}(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ f_{91}(f_{91}(n + 11)) & \text{sinon} \end{cases}$$

qui est bien définie et vaut en fait 91 pour tout argument inférieur ou égal à 101, d'où son nom. Par exemple, si nous déroulons la définition de f_{91} en 84 :

$$\begin{aligned} f_{91}(84) &= f_{91}(f_{91}(95)) = f_{91}(f_{91}(f_{91}(106))) = f_{91}(f_{91}(96)) = f_{91}(f_{91}(f_{91}(107))) \\ &= f_{91}(f_{91}(97)) = \ldots = f_{91}(f_{91}(101)) \\ &= f_{91}(91) = \ldots = f_{91}(101) \\ &= 91 \end{aligned}$$

Nous observons donc que toute étape du processus de déroulage possède la forme d'une application itérée de f_{91}, c'est-à-dire

$$f_{91}^{(e)}(n) = \underbrace{f_{91}(\ldots f_{91}}_{e \text{ fois}}(n))$$

pour des entiers e et n. Nous pouvons donc écrire un programme itératif pour calculer f_{91} en utilisant e et n comme variables, comme le programme 5.1. Cela revient en fait à représenter la pile d'appel par le compteur e.

Programme 5.1 Algorithme itératif calculant f_{91}

```
1:  function F91(n)
2:      e ← 1
3:      while e ≠ 0 do
4:          if n > 100 then
5:              n ← n − 10
6:              e ← e − 1
7:          else
8:              n ← n + 11
9:              e ← e + 1
10:         end if
11:     end while
12:     return n
13: end function
```

Partant du programme 5.1, l'objectif est de démontrer que cet algorithme calcule bien la fonction 91 de McCarthy. Dans ce cas précis, la structure correspondant à la pile (un compteur) est suffisamment simple pour pouvoir effectuer la preuve directement. En effet, l'invariant sur la « pile » est donné par $f_{91}(n_0) = f_{91}^{(e)}(n)$ et permet de conclure à la correction partielle du programme. Cependant, nous pouvons faire une preuve alternative qui évite toute mention de la structure de la pile. Tout d'abord, nous convertissons (une fois pour toutes) la sémantique à petit pas du langage utilisé par le programme en système de transition ordonné, puis en jeu, ce qui donne une équivalence entre les triplets de Hoare et les énoncés pour un contexte fixé. Ensuite, nous prouvons ce programme itératif par le biais du transformateur récursif donné en figure 5.11.

Dans le contrat de la structure récursive du transformateur, le prédicat `start-while` sert à synchroniser l'état du jeu avec le flot du contrôle du programme, en indiquant que le contrôle se situe au début de la boucle. De même, $W_{prelude}, W_{corps}, W_{postlude}$ correspondent respectivement à des transformateurs naturels pour l'exécution de la procédure jusqu'à l'entrée de la boucle, d'une étape de

$\left[\ \text{(contexte associé au langage)}\ \right] \Rightarrow []$
$W_{prelude}$
call-rec $F()$: $\left\langle \begin{pmatrix} \texttt{start-while}(\mathsf{now}) \\ \wedge\ e > 0 \end{pmatrix} \hookrightarrow \begin{pmatrix} \texttt{start-while}(\mathsf{now}) \\ \wedge\ \mathsf{now}[n] = f_{91}(\mathsf{old}[n]) \\ \wedge\ \mathsf{now}[e] = \mathsf{old}[e] - 1 \end{pmatrix} \right\rangle$
variant$\{(101 - \mathsf{now}[n])^+\}$
progress$\{\geq\}$
    let $n_0 \in \mathbb{N} \leftarrow \mathsf{now}[n]$
    W_{corps}
    switch
    case $n_0 \leq 100$:
        F
        F
    default :
        skip
    end-switch
diverges$(H, f_G \mid clos)$
    absurd
end-call-rec
$W_{postlude}$

FIGURE 5.11 – Transformateur récursif pour le programme itératif calculant f_{91}

la boucle, et de la sortie de la boucle jusqu'à la fin de la procédure. Ces transformateurs sont obtenus en suivant la structure des parties correspondantes du programme itératif. Nous utilisons également la notation $x[n]$ pour récupérer la valeur d'une variable n depuis un état x du programme itératif.

Nous pouvons alors établir la condition de vérification du programme sans difficulté. Pour établir la condition de progression, autrement dit la terminaison du programme, nous avons besoin de propriétés supplémentaires sur f_{91}. Nous pouvons en fait nous contenter de montrer ces propriétés simultanément en ajoutant la condition

$$\mathsf{now}[n] = \begin{cases} 91 & \text{si } \mathsf{old}[n] \leq 100 \\ \mathsf{old}[n] - 10 & \text{sinon} \end{cases}$$

dans la post-condition de la structure récursive.

Même si nous n'obtenons pas vraiment de gains dans ce cas précis, remarquons que nous avons effectivement réussi à éliminer complètement l'invariant correspondant à la structure de la pile. Nous avons également réduit les conditions de vérification à des notions purement arithmétiques, plus simples que l'itération de fonction. Enfin, nous avons obtenu très facilement la terminaison. À première vue, il ne semble pas forcément évident que la quantité décroissante choisie pour la récurrence puisse être facilement adaptée à la structure itérative.

Remarquons que nous pouvons vérifier les gains potentiels de cette méthodologie par le biais de l'outil Why3 sans pour autant utiliser le développement sur

les jeux. En effet, nous pouvons utiliser le langage de Why3 lui-même pour simuler les combinaisons de transformateurs. Nous simulons alors les transformateurs de base, c'est-à-dire $W_{prelude}$, W_{corps}, et $W_{postlude}$ dans le cas précédent, par l'exécution de routines qui isolent les fragments du programme en question. Dans le cas d'un programme consistant en une unique boucle, nous pouvons par exemple simuler l'exécution d'un tour de boucle par une unique fonction avec effets, qui lance une exception pour représenter la sortie de la boucle. Dans le cas du programme 5.1 cela correspond à la fonction suivante :

```
let corps () : unit
= if not (!e > 0) then raise Stop;
  if !n > 100 then begin
    n := !n - 10;
    e := !e - 1
  end else begin
    n := !n + 11;
    e := !e + 1
  end
```

où les deux variables mutables n et e sont simulées par des références.

Nous pouvons alors obtenir la condition de vérification du transformateur récursif proposé en demandant à Why3 de calculer la plus faible pré-condition du programme suivant :

```
let f91_impl (n:int) : int
  ensures { result = f91 n }
= let e = ref 1 in
  let n = ref n in
  let corps () : unit = ... in
  let rec f () : unit
    requires { !e > 0 }
    variant { 101 - !n }
    ensures { !e = !(old e) - 1 ∧ !n = f91 !(old n) }
    raises { Stop → false }
  = let n0 = !n in corps (); if n0 ≤ 100 then (f (); f ()) in
  try f (); corps (); absurd with Stop → !n end
```

En pratique, nous devons soit ajouter un contrat à la routine `corps` pour pouvoir obtenir le résultat, soit remplacer `corps` par sa définition à chaque utilisation. Nous avons choisi d'écrire un contrat donnant la relation d'entrées/sorties de cette procédure, ce qui donne un effet quasi identique au déroulage systématique de la définition. La condition de vérification calculée par Why3 est alors suffisamment simple pour être démontrée quasi instantanément par les démonstrateurs SMT.

Remarquons qu'en pratique, cette technique revient à une transformation du flot de contrôle sous une forme équivalente, mais plus pratique pour la vérification. En effet, le comportement d'une exécution correcte du fragment `f(); corps(); absurd` est parfaitement équivalent à celui de l'exécution d'une séquence infinie d'appel à `corps`, c'est-à-dire précisément à une boucle. Cela justifie a posteriori la méthode de preuve que nous utilisons. L'outil Why3 ne nous permet cependant pas d'éliminer la structure récursive additionnelle, car nous n'avons aucun moyen de vérifier que

ce flot de contrôle supplémentaire est inutile pour l'exécution.

L'exemple de la fonction de McCarthy est ici trop simple pour constituer un réel gain. Que ce soit pour le programme itératif ou sa transformation, les conditions de vérification sont déchargées sans aide extérieure par les démonstrateurs automatiques. Nous avons donc essayé d'appliquer exactement le même type d'approche à un cas plus compliqué, à savoir l'algorithme de Schorr-Waite. Il s'agit d'un algorithme itératif de marquage des sommets accessibles dans un graphe mémoire, pouvant par exemple être utilisé au sein d'un glaneur de cellules. Le principe de l'algorithme est le même que celui de la recherche en profondeur, à ceci près que les pointeurs des sommets traversés sont temporairement renversés pour coder la pile d'appel en place. Il s'agit donc bien d'un programme de nature dérécursifiée, et comme il s'agit d'un programme constitué d'une unique boucle, nous pouvons appliquer rigoureusement la même technique que pour la fonction 91 de McCarthy.

Pour effectuer cette expérience, nous nous sommes initialement basé sur le code d'une version déjà vérifiée en Why3 de l'algorithme de Schorr-Waite[3]. Il nous a fallu moins d'une journée pour vérifier le programme dans le cadre de notre approche alternative, tout en généralisant le programme au cas d'un nombre arbitraire de pointeurs par sommet. Grâce à la transformation du flot de contrôle que nous effectuons pour la vérification, nous avons pu réduire l'effort de preuve à la vérification d'une recherche en profondeur récursive, ce qui ne présente pas de difficulté particulière. Il y a donc un gain clair dû à l'approche choisie, ce que nous pouvons également observer en comparant avec une preuve effectuée de manière directe. La version sur laquelle nous nous sommes basés, vérifiée directement, consiste en 69 lignes de code et a nécessité 153 lignes de spécification pour la vérification. Notre variante[4] consiste en 76 lignes de code, et a nécessité 135 lignes de spécification, dont 37 sont en théorie inutile. Ces dernières correspondent en effet à la relation d'entrée/sorties du corps de la boucle — le contrat de `corps`, calculé à la main — et pourraient être éliminées en remplaçant chaque appel par la définition du corps de la boucle.

Nous avons également appliqué cette approche de preuve à un autre algorithme de marquage, l'algorithme de traversée en place d'arbre binaire[5] proposé comme second problème pour l'édition 2016 de la compétition VerifyThis. Nous n'avons cependant pas de preuve directe pouvant servir de point de comparaison pertinent. Nous n'avons trouvé qu'une seule preuve alternative de ce programme, effectuée à l'aide de l'outil VeriFast [51]. La comparaison semble nettement en faveur de la preuve effectuée via l'outil VeriFast, deux fois plus courte que notre preuve utilisant Why3, mais est considérablement biaisée par la différence de traitement de la mémoire. L'outil VeriFast propose un modèle mémoire natif, traité via la logique de séparation, tandis que nous avons dû effectuer un encodage en Why3 et démontrer des lemmes ad-hoc. De plus, nous ne démontrons pas les mêmes propriétés puisque notre preuve traite le cas potentiel du partage de sous-arbres, ce qui aurait demandé des efforts supplémentaires dans le cadre de la logique de séparation.

Nous disposons également d'une preuve directe du programme de traversée en place d'arbre binaire en Why3, mais cette dernière preuve est dérivée mécanique-

3. http://toccata.lri.fr/gallery/schorr_waite.en.html
4. http://toccata.lri.fr/gallery/schorr_waite_via_recursion.en.html
5. http://toccata.lri.fr/gallery/verifythis_2016_tree_traversal.en.html

ment de la preuve en style récursif. En effet, nous avons originellement introduit cette transformation en style récursif pour résoudre ce problème de preuve de programme, précisément pour éviter de manipuler une pile. Nous avons dérivé la preuve directe après coup. Il n'est donc pas surprenant que dans ce cas, la comparaison soit nettement en faveur de la version récursive.

Chapitre 6

Preuve d'un schéma de compilation à l'aide des jeux

Nous nous intéressons maintenant à une mise en pratique de la méthode de preuve proposée dans le chapitre 5. Comme dans la section 2.5, nous étudions la compilation d'un langage structuré vers un langage non structuré type assembleur, et cherchons à prouver sa correction. Dans le but de démontrer les gains obtenus par notre formalisme, nous ajoutons au langage source un certain nombre de mécanismes de contrôle que nous aurions eu des difficultés à traiter auparavant, comme les fonctions (mutuellement) récursives et des pointeurs de fonction. Nous ajoutons également aux deux langages des mécanismes d'entrée/sortie pour démontrer la capacité de notre approche à gérer le non-déterminisme, ainsi qu'à établir des propriétés non triviales dans le cas de comportements qui ne terminent pas.

Afin de montrer la capacité de notre système à gérer aussi bien le caractère existentiel que le caractère universel, nous prouvons pour notre schéma de compilation un résultat de simulation en avant et un résultat de simulation en arrière. Nous démontrons ces deux résultats en les reformulant tout d'abord dans les jeux via les outils de la section 4.5, puis nous les vérifions à l'aide des outils de preuve de la section 5.3.

Nous affirmons que, comme dans l'exemple plus simple de la section 2.5, l'effort de preuve est quasi intégralement contenu dans la formulation des garanties correspondant à la correction du schéma de compilation, ainsi que dans les choix de combinaisons effectuées pour établir ces garanties. Le reste de la preuve correspond à un déroulement mécanique de ces combinaisons, et à des vérifications de propriétés suffisamment simples pour être *a priori* presque toutes à la portée d'un démonstrateur automatique. Néanmoins, cette dernière affirmation est encore en attente d'une confirmation expérimentale à l'heure de l'écriture de cette thèse. Nous avons l'intention d'effectuer celle-ci dès que le développement Why3 correspondant aux chapitres 4 et 5 sera terminé.

Ce chapitre est organisé de la manière suivante. Dans la section 6.1, nous présentons le schéma de compilation que nous allons vérifier. Nous énonçons ensuite dans la section 6.2 les deux résultats de correction que nous souhaitons obtenir. Enfin, nous démontrons ces deux résultats dans les sections 6.4 et 6.3.

6.1 Compilateur

6.1.1 Langage source

Nous définissons comme langage source le langage impératif $\mathcal{L}_{\mathtt{s}}$ non typé, présenté dans la figure 6.1. Un programme $\mathcal{L}_{\mathtt{s}}$ est composé d'un ensemble fini de déclarations de fonction, contenant la définition d'une fonction principale `main` utilisée comme point d'entrée. Le corps d'une fonction est défini par une expression. Pour simplifier la présentation du langage, nous n'avons pas fait de distinction entre expressions et instructions. Nous avons également choisi de représenter les opérateurs mathématiques tels que les opérateurs arithmétiques et booléens par le biais d'un ensemble fini de fonctions prédéfinies. L'exécution d'un tel programme $\mathcal{L}_{\mathtt{s}}$ a pour effet observable d'écrire une séquence d'entiers dans un canal de sortie, au moyen de la fonction prédéfinie `print`. Le langage fournit également une fonction non-déterministe `scan` pour simuler la lecture d'un entier sur un canal d'entrée.

En plus des constructions types d'un langage Imp avec fonctions, le langage $\mathcal{L}_{\mathtt{s}}$ offre notamment des pointeurs de fonctions inspirés du langage C, ainsi qu'un mécanisme de retour généralisant les opérateurs de contrôle usuels `break`, `continue` et `return`. Ce dernier fonctionne de la manière suivante. La déclaration d'une fonction ainsi que l'ouverture d'un bloc introduit une étiquette. Celle-ci peut être utilisée pour sortir du bloc/de la fonction à un point interne arbitraire.

Nous donnons la sémantique du langage $\mathcal{L}_{\mathtt{s}}$ sous la forme d'une sémantique à petit pas, par le biais d'une machine abstraite de type CEK donnée sous la forme introduite par Felleisen et Friedman [41]. L'état de la machine est donc un triplet (e, E, k) composé d'une expression, d'un environnement et d'un code de continuation. Nous présentons la structure des valeurs et des codes de continuation dans la figure 6.2. Les valeurs peuvent être de trois natures possibles : il s'agit soit d'une

$$
\begin{array}{lll}
e & ::= x & \text{(variable)} \\
& \mid \&f \mid e(e,\ldots,e) & \text{(adressage/appel de fonction)} \\
& \mid () & \text{(expression vide)} \\
& \mid x \leftarrow e & \text{(affectation)} \\
& \mid e; e & \text{(séquence)} \\
& \mid \texttt{if } e \texttt{ then } e \texttt{ else } e & \text{(conditionnelle)} \\
& \mid \texttt{while } e \texttt{ do } e & \text{(boucle)} \\
& \mid [L]\{\ x_1,\ldots,x_n : e\ \} & \text{(bloc)} \\
& \mid \texttt{return}[L]\ e & \text{(sortie arbitraire)} \\
\\
fd & ::= f[L](x_1,\ldots,x_n) = e & \text{(déclaration de fonction)}
\end{array}
$$

FIGURE 6.1 – Langage $\mathcal{L}_{\mathtt{s}}$

$$
\begin{array}{l}
v ::= \ldots \mid () \mid \&f \\
\\
k ::= \top \\
\quad \mid k(e,\ldots,e) \mid \&f(v,\ldots,v,k,e,\ldots,e) \\
\quad \mid x \leftarrow k \\
\quad \mid k; e \\
\quad \mid \texttt{if } k \texttt{ then } e \texttt{ else } e \\
\quad \mid [L]\{\ x_1,\ldots,x_n : k\ \} \\
\quad \mid \texttt{return}[L]k
\end{array}
$$

FIGURE 6.2 – Valeurs et codes de continuation pour la machine CEK

constante mathématique pouvant être manipulée par le biais d'opérateurs prédéfinis (notamment les entiers et les booléens), soit d'un pointeur de fonction, ou encore d'une valeur supplémentaire inexploitable `()`. Cette dernière est utilisée pour décrire le résultat des instructions. L'environnement de la machine abstraite est donné par une paire $E = (\Sigma, \Pi)$ composée de :

— une fonction Σ qui à tout nom de variable associe une suite finie de valeurs (potentiellement vide). Cette suite donne les valeurs associées aux instances successives d'une même variable.

— une séquence finie Π correspondant à la trace des évènements d'entrée/sortie du programme. Les évènements d'entrée sont représentés sous la forme $I(n)$ et les évènements de sortie sous la forme $O(n)$, où n est un entier.

Nous définissons alors la sémantique d'un programme $\mathcal{L}_{\mathtt{s}}$ donné par un ensemble F de déclarations de fonctions comme l'ensemble des exécutions de la machine abstraite à partir des états de la forme $(\&\texttt{main()}, ((\_ \rightarrow s_0), s_0), \top)$, où s_0 dénote une suite vide. Les états d'arrêt valides de la machine sont les états de la forme $(v, (\Sigma_1, \Pi_1), \top)$, où v est une valeur. Nous appelons *état bloquant* tout autre état pour lequel la machine abstraite ne permet pas de réduction.

Nous donnons la relation de transition $\longrightarrow_F$ de la machine abstraite associée au langage $\mathcal{L}_{\mathtt{s}}$ en figure 6.3. Cette relation est paramétrée par l'ensemble de déclarations de fonctions du programme. Pour simplifier la présentation, nous utilisons les

notations suivantes pour la mise à jour de l'environnement :

Définition 6.1 (Ajout de variables)**.** Pour Σ une association des variables, $(x_i)_{0\leq i<n}$ une suite de variables distinctes, et $(v_i)_{0\leq i<n}$ une séquence de valeurs de même taille, nous définissons l'*ajout* à Σ des variables $(x_i)_{0\leq i<n}$ initialisées par $(v_i)_{0\leq i<n}$ comme la fonction $\Sigma' = \Sigma[+(x_i \leftarrow v_i)_{0\leq i<n}]$ définie par

$$\Sigma'(x) = \begin{cases} \Sigma(x) \text{ auquel on ajoute un dernier élément } v_i & \text{si } x = x_i \\ \Sigma(x) & \text{sinon} \end{cases}$$

Nous étendons cette notation aux environnements par

$$(\Sigma, \Pi)[+(x_i \leftarrow v_i)_{0\leq i<n}] = (\Sigma[+(x_i \leftarrow v_i)_{0\leq i<n}], \Pi)$$

.

Définition 6.2 (Retrait de variables)**.** Pour Σ une association des variables et $(x_i)_{0\leq i<n}$ une suite de variables distinctes telles que $\Sigma(x_i)$ soit non vide pour tout i, nous définissons le *retrait* des variables $(x_i)_{0\leq i<n}$ de Σ comme la fonction $\Sigma' = \Sigma[-(x_i)_{0\leq i<n}]$ définie par

$$\Sigma'(x) = \begin{cases} \Sigma(x) \text{ auquel on enlève le dernier élément} & \text{si } x = x_i \\ \Sigma(x) & \text{sinon} \end{cases}$$

Nous étendons cette notation aux environnements par

$$(\Sigma, \Pi)[-(x_i)_{0\leq i<n}] = (\Sigma[-(x_i)_{0\leq i<n}], \Pi)$$

Définition 6.3 (Mise à jour d'une variable)**.** Pour Σ une association des variables, x une variable telle que $\Sigma(x)$ soit non vide, et v une valeur, nous définissons la *mise à jour* de x par v dans Σ comme la fonction $\Sigma' = \Sigma[x \leftarrow v]$ définie par

$$\Sigma'(y) = \begin{cases} \Sigma(y) \text{ où le dernier élément est remplacé par } v & \text{si } y = x \\ \Sigma(y) & \text{sinon} \end{cases}$$

Nous étendons cette notation aux environnements par

$$(\Sigma, \Pi)[x \leftarrow v] = (\Sigma[x \leftarrow v], \Pi)$$

Définition 6.4 (Émission d'un évènement entrée/sortie)**.** Pour une trace d'entrée/-sortie Π, nous définissons l'*émission* d'un évènement d'entrée/sortie x comme la séquence $\Pi +\!\!+ x$ définie par Π auquel on ajoute x comme dernier élément. Nous étendons cette notation aux environnements par

$$(\Sigma, \Pi) +\!\!+ x = (\Sigma, \Pi +\!\!+ x)$$

Remarque 6.5. Dans le but de simplifier les règles de réduction des paramètres pour les appels de fonction, nous ajoutons un paramètre additionnel sans signification () aux fonctions lors de la mise sous forme de code de continuation. Ce paramètre sert de marqueur pour la fin de l'évaluation des paramètres, et permet d'éviter de faire des règles particulières pour l'appel d'une fonction sans paramètres. De cette manière, nous évitons également de donner un rôle particulier au dernier paramètre significatif dans les règles de réduction.

$$\frac{E = (\Sigma, \Pi) \qquad \Sigma(x) = (u_i)_{0 \leq i \leq m} \qquad u_m \neq ()}{(x, E, k) \longrightarrow_F (u_m, E, k)}$$

$$\frac{}{(e((e_i)_{0 \leq i < n}), E, k) \longrightarrow_F (e, E, k((e_i)_{0 \leq i < n}))}$$

$$\frac{e_n = ()}{(\&f, E, k((e_i)_{0 \leq i < n})) \longrightarrow_F (e_0, E, \&f(k, (e_i)_{0 < i \leq n}))}$$

$$\frac{v_m \neq () \qquad n \neq 0}{(v_m, E, \&f((v_i)_{0 \leq i < m}, k, (e_i)_{0 \leq i < n})) \longrightarrow_F (e_0, E, \&f((v_i)_{0 \leq i \leq m}, k, (e_i)_{0 < i < n}))}$$

$$\frac{(f[L]((x_i)_{0 \leq i < n}) = e) \in F}{((), E, \&f((v_i)_{0 \leq i < n}, k)) \longrightarrow_F (e, E[+(x_i \leftarrow v_i)_{0 \leq i < n}], [L]\{(x_i)_{0 \leq i < n} : k\})}$$

$$\frac{f \text{ prédéfinie par l'opérateur } \mathcal{F} \qquad (v_i)_{0 \leq i < n} \in \mathtt{dom}(\mathcal{F})}{((), E, \&f((v_i)_{0 \leq i < n}, k) \longrightarrow_F (\mathcal{F}((v_i)_{0 \leq i < n}), E, k)}$$

$$\frac{v \in \mathbb{Z}}{((), E, \&\mathtt{print}(v, k)) \longrightarrow_F ((), E +\!\!+ O(v), k)} \qquad \frac{v \in \mathbb{Z}}{((), E, \&\mathtt{scan}(k)) \longrightarrow_F (v, E +\!\!+ I(v), k)}$$

$$\frac{}{(x \leftarrow e, E, k) \longrightarrow_F (e, E, x \leftarrow k)} \qquad \frac{v \neq ()}{(v, E, x \leftarrow k) \longrightarrow_F ((), E[x \leftarrow v], k)}$$

$$\frac{}{(e_1; e_2, E, k) \longrightarrow_F (e_1, E, k; e_2)} \qquad \frac{}{(v, E, k; e_2) \longrightarrow_F (e_2, E, k)}$$

$$\frac{}{(\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3, E, k) \longrightarrow_F (e_1, E, \mathtt{if}\ k\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3)}$$

$$\frac{v \in \{\top, \bot\} \qquad i \in \{1; 2\} \qquad (i = 1 \Leftrightarrow v = \top)}{(v, E, \mathtt{if}\ k\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2) \longrightarrow_F (e_i, E, k)}$$

$$\frac{}{(\mathtt{while}\ e_1\ \mathtt{do}\ e_2, E, k) \longrightarrow_F (\mathtt{if}\ e_1\ \mathtt{then}\ e_2;\ \mathtt{while}\ e_1\ \mathtt{do}\ e_2\ \mathtt{else}\ (), E, k)}$$

$$\frac{}{([L]\{\ (x_i)_{0 \leq i < n} : e\ \}, E, k) \longrightarrow_F (e, E[+(x_i \leftarrow ())_{0 \leq i < n})], [L]\{\ (x_i)_{0 \leq i < n} : k\ \})}$$

$$\frac{}{(v, E, [L]\{\ (x_i)_{0 \leq i < n} : k\ \}) \longrightarrow_F (v, E[-(x_i)_{0 \leq i < n}], k)}$$

$$\frac{}{(\mathtt{return}[L]\ e, E, k) \longrightarrow_F (e, E, \mathtt{return}[L]\ k)}$$

$$\frac{k_2 \text{ directement construit sur } k_1 \qquad k_2 \text{ pas un bloc étiqueté par } L}{(v, E, \mathtt{return}[L]\ k_2) \longrightarrow_F (v, E, \mathtt{return}[L]\ k_1)}$$

$$\frac{}{(v, E, \mathtt{return}[L]\ [L]\{\ (x_i)_{0 \leq i < n} : k\ \}) \longrightarrow_F (v, E, [L]\{\ (x_i)_{0 \leq i < n} : k\ \})}$$

FIGURE 6.3 – Sémantique à petits pas de $\mathcal{L}_{\mathtt{s}}$

`instr`	::=	`...`	
	|	`nop`	(sans effet)
	|	`pop` n	($n \in \mathbb{N}$, dépile n valeurs)
	|	`aload L`	(empile l'étiquette `L`)
	|	`load` n	($n \in \mathbb{N}$, empile la n-ième valeur)
	|	`store` n	($n \in \mathbb{N}$, dépile une valeur v, affecte la n-ième valeur à v)
	|	`jmp`	(dépile l'étiquette `L` au sommet de la pile, déplace la tête de lecture à `L`)
	|	`jnz L`	(dépile le représentant d'un booléen en sommet de pile, si représente $\top$, déplace la tête de lecture à `L`)
	|	`in`	(lecture d'une entrée)
	|	`out`	(émission d'une sortie)
	|	`exit`	(arrête la machine)

FIGURE 6.4 – Instructions du langage $\mathcal{L}_\mathtt{t}$

Définition 6.6 (Programme $\mathcal{L}_\mathtt{s}$ bien formé)**.** Un programme $\mathcal{L}_\mathtt{s}$ est *bien formé* s'il respecte les conditions suivantes :

(liaison des variables) tout accès/affectation à une variable x se fait à l'intérieur d'un bloc ou d'une déclaration de fonction qui déclare x.

(liaison des étiquettes) de même, tout retour à une étiquette L se fait à l'intérieur d'un bloc ou d'une déclaration de fonction qui déclare L.

(liaison des adresses de fonctions) tout adressage de fonction correspond soit à une fonction déclarée dans le programme, soit à un opérateur prédéfini, soit à l'une des deux fonctions `print` et `scan`.

(unicité des fonctions 1) le programme ne contient pas deux déclarations de fonction sous le même nom.

(unicité des fonctions 2) le programme ne déclare pas de fonction ayant pour nom le nom de l'un des opérateurs prédéfinis, `print` ou `scan`.

(existence d'un point d'entrée) le programme déclare une fonction principale sans argument appelée `main`.

Par la suite, nous ne considérons que des programmes $\mathcal{L}_\mathtt{s}$ bien formés.

6.1.2 Langage cible

Nous choisissons comme langage cible $\mathcal{L}_\mathtt{t}$ un langage bas niveau proche d'un assembleur. Les programmes du langage $\mathcal{L}_\mathtt{t}$ sont définis comme des séquences d'instructions pour une machine virtuelle. Chaque instruction de la séquence est potentiellement porteuse d'une étiquette identifiant l'instruction. Ces étiquettes doivent être distinctes pour chaque instruction.

L'état de la machine virtuelle est donné par un triplet $(\mathtt{pc}, S, \Pi)$, composé de :

— une tête de lecture `pc` pointant sur une instruction du code. La tête de lecture est modélisée par un entier naturel correspondant à la position dans la séquence d'instructions correspondant au code.

— une pile de valeurs S.
— un canal d'entrée/sortie Π suivant la même structure que celui du langage source.

À chaque étape, la machine procède en exécutant l'instruction qui se trouve au niveau de la tête de lecture. À moins que l'instruction ne change explicitement l'emplacement de la tête de lecture, la machine avance ensuite la tête de lecture d'un cran, et recommence.

Comme nous n'avons pas précisé la nature exacte des valeurs du langage source, nous allons également laisser la nature des valeurs de la pile en grande partie indéterminée. Nous allons cependant ajouter certaines contraintes pour que les valeurs du langage cible puissent représenter facilement les valeurs du langage source. Les valeurs du langage source doivent être associées à un ensemble de valeurs du langage cible, qui sont les représentants valides de la valeur source. Nous imposons également que l'ensemble des étiquettes soit inclus dans l'ensemble des valeurs, pour pouvoir représenter les adresses de fonctions.

Remarquons qu'il est tout à fait possible qu'une même valeur du langage $\mathcal{L}_\mathtt{t}$ représente plusieurs valeurs du langage $\mathcal{L}_\mathtt{s}$, notamment si les étiquettes sont des entiers. Ce n'est pas un problème tant qu'aucune opération correcte du langage source ne permet de différencier ces deux valeurs.

Nous donnons le fragment du jeu d'instruction que nous allons utiliser pour effectuer la compilation dans la figure 6.4. Les positions référencées par les instructions `load` et `store` correspondent à la position relative au sommet de la pile, avant exécution de l'instruction pour `load` et après que la valeur à stocker ait été dépilée pour `store`. La position 0 correspond au sommet, la position 1 à la valeur juste en dessous, et ainsi de suite. Les instructions de saut déplacent la tête de lecture à l'instruction marquée par l'étiquette correspondante. Nous donnons la correspondance précise entre les instructions et la relation de transition de la machine virtuelle $\longrightarrow_{(I_l)_{0\leq l<n}}$ dans la figure 6.5. Cette relation est – comme pour le langage source – paramétrée par le code du programme, donné par une séquence d'instructions étiquetées $(I_l)_{0\leq l<n}$.

A priori, notre machine virtuelle possède plus d'instructions que le fragment mentionné. Il est notamment nécessaire d'avoir des opérations permettant de compiler les fonctions prédéfinies (notamment des instructions arithmétiques). Comme nous n'allons pas nous intéresser en détail à la compilation des opérations prédéfinies, nous n'allons pas non plus préciser le jeu d'instructions correspondant à celles-ci.

Nous utilisons une notation type assembleur pour représenter les programmes du langage $\mathcal{L}_\mathtt{t}$: une liste d'instructions verticalement consécutives, où les étiquettes associées aux instructions sont marquées sur la gauche lorsqu'il y en a. Par exemple, nous pouvons écrire un programme qui émet de manière répétée la valeur initialement en sommet de pile de la manière suivante :

```
L:  load 0
    out
    aload L
    jmp
```

Un autre exemple possible est le fragment suivant, qui correspond à l'échange des deux valeurs au sommet de la pile :

$$\frac{I_{\mathtt{pc}} = \mathtt{nop}}{(\mathtt{pc}, S, \Pi) \longrightarrow_{(I_l)_{0 \leq l < N}} (\mathtt{pc} + 1, S, \Pi)}$$

$$\frac{I_{\mathtt{pc}} = \mathtt{pop}\ \ n}{(\mathtt{pc}, (S; (v_i)_{0 \leq i < n}), \Pi) \longrightarrow_{(I_l)_{0 \leq l < N}} (\mathtt{pc} + 1, S, \Pi)}$$

$$\frac{I_{\mathtt{pc}} = \mathtt{aload}\ \mathtt{L}}{(\mathtt{pc}, S, \Pi) \longrightarrow_{(I_l)_{0 \leq l < N}} (\mathtt{pc} + 1, (S; \mathtt{L}), \Pi)}$$

$$\frac{I_{\mathtt{pc}} = \mathtt{load}\ \ n}{(\mathtt{pc}, (S; (v_i)_{0 \leq i \leq n}), \Pi) \longrightarrow_{(I_l)_{0 \leq l < N}} (\mathtt{pc} + 1, (S; (v_i)_{0 \leq i \leq n}; v_0), \Pi)}$$

$$\frac{I_{\mathtt{pc}} = \mathtt{store}\ \ n}{(\mathtt{pc}, (S; (v_i)_{0 \leq i \leq n}; v), \Pi) \longrightarrow_{(I_l)_{0 \leq l < N}} (\mathtt{pc} + 1, (S; v; (v_i)_{0 < i \leq n}), \Pi)}$$

$$\frac{I_{\mathtt{pc}} = \mathtt{jmp} \qquad I_{\mathtt{pc}_L} \text{ étiquetée par } \mathtt{L}}{(\mathtt{pc}, (S; \mathtt{L}), \Pi) \longrightarrow_{(I_l)_{0 \leq l < N}} (\mathtt{pc}_L, S, \Pi)}$$

$$\frac{I_{\mathtt{pc}} = \mathtt{jnz}\ \mathtt{L} \qquad v \text{ représente } \top \qquad I_{\mathtt{pc}_L} \text{ étiquetée par } \mathtt{L}}{(\mathtt{pc}, (S; v), \Pi) \longrightarrow_{(I_l)_{0 \leq l < N}} (\mathtt{pc}_L, S, \Pi)}$$

$$\frac{I_{\mathtt{pc}} = \mathtt{jnz}\ \mathtt{L} \qquad v \text{ représente } \bot}{(\mathtt{pc}, (S; v), \Pi) \longrightarrow_{(I_l)_{0 \leq l < N}} (\mathtt{pc} + 1, S, \Pi)}$$

$$\frac{I_{\mathtt{pc}} = \mathtt{in} \qquad v \text{ représente } m \in \mathbb{Z}}{(\mathtt{pc}, S, \Pi) \longrightarrow_{(I_l)_{0 \leq l < N}} (\mathtt{pc} + 1, (S; v), \Pi +\!\!+ I(m))}$$

$$\frac{I_{\mathtt{pc}} = \mathtt{out} \qquad v \text{ représente } m \in \mathbb{Z}}{(\mathtt{pc}, (S; v), \Pi \longrightarrow_{(I_l)_{0 \leq l < N}} (\mathtt{pc} + 1, S, \Pi +\!\!+ O(m))}$$

FIGURE 6.5 – Sémantique à petit pas du langage cible

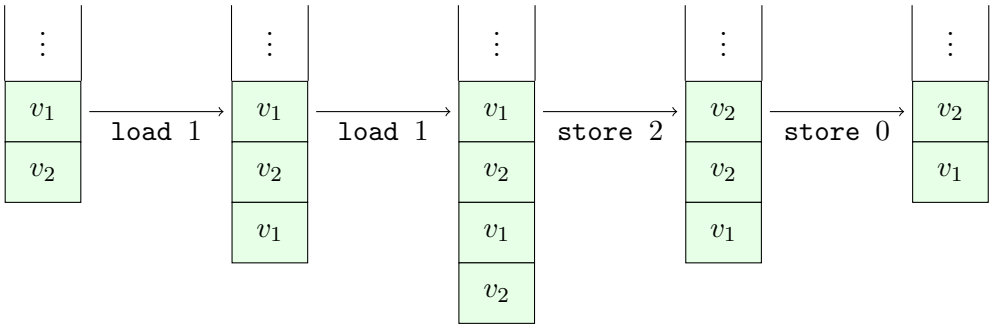

FIGURE 6.6 – Échange de valeurs en sommet de pile

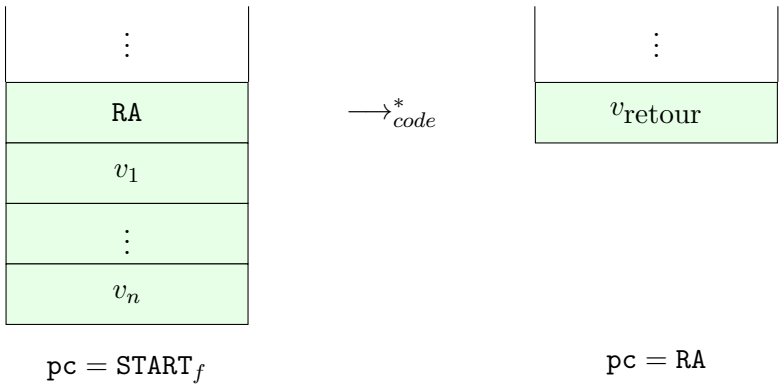

FIGURE 6.7 – Protocole d'appel

```
load 1
load 1
store 2
store 0
```

Par la suite, nous abrégerons ce fragment par la pseudo-instruction `swap`. Cette séquence d'instructions fonctionne de la manière suivante. Les deux premières instructions `load` dupliquent les deux valeurs en sommet de la pile, puis les deux suivantes stockent ces valeurs à la position finales.

6.1.3 Schéma de compilation

Le schéma de compilation que nous allons vérifier compile chaque fonction indépendamment. Nous compilons les fonctions de manière traditionnelle. Nous associons chaque nom de fonction à une étiquette, correspondant au point d'entrée de sa traduction. Une fonction est alors correctement traduite si lorsque l'on démarre du point d'entrée associé, si la pile contient au sommet les paramètres de la fonction dans l'ordre inverse suivis d'une adresse dite de retour, alors l'exécution à partir de ce point d'entrée atteint l'adresse de retour, et la pile alors obtenue correspond à la pile d'entrée où on a remplacé les paramètres et l'adresse de retour par la va-

```
START_PRINT : out
              load 0
              jmp
START_SCAN :  in
              swap
              jmp
```

FIGURE 6.8 – Code compilé pour `print` et `scan`

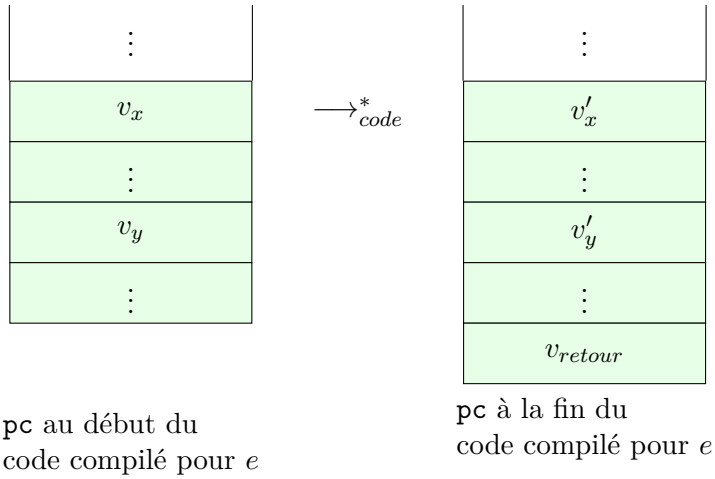

FIGURE 6.9 – Fonctionnement normal du code compilant une expression e, où les seules variables modifiées sont x et y

leur de retour de la fonction. Le protocole d'appel décrit ci-dessus est illustré par la figure 6.7.

La représentation des valeurs du langage source correspond principalement aux ensembles de représentants décrits dans la section 6.1.2. Nous raffinons ce prédicat de la manière suivante. Tout d'abord, nous ajoutons le fait que la valeur inexploitable () peut être représentée par n'importe quelle valeur du langage cible. De cette manière, nous pouvons traiter la valeur inexploitable de manière uniforme vis-à-vis des autres valeurs. De plus, nous ajoutons le fait qu'une adresse de fonction $\&f$ est correctement représentée par une valeur du langage cible si cette valeur est l'étiquette associée à f. Cette représentation est donc dépendante de l'association définie au tout début de la compilation. De plus, elle garantit que seules les fonctions prédéfinies ou définies par le programme sont représentables.

Nous traitons tout d'abord les opérateurs prédéfinis ainsi que les fonctions `print` et `scan`. Comme le comportement de ces fonctions est connu, il suffit de leur affecter à chacune un code dédié, ce qui est immédiat si les opérateurs prédéfinis sont proches des instructions de la machine virtuelle. Nous montrons par exemple le code généré pour les fonctions `print` et `scan` dans la figure 6.8. Les (pseudo-)instructions présentes avant le saut final servent à mettre la pile dans une forme correspondant au protocole d'appel.

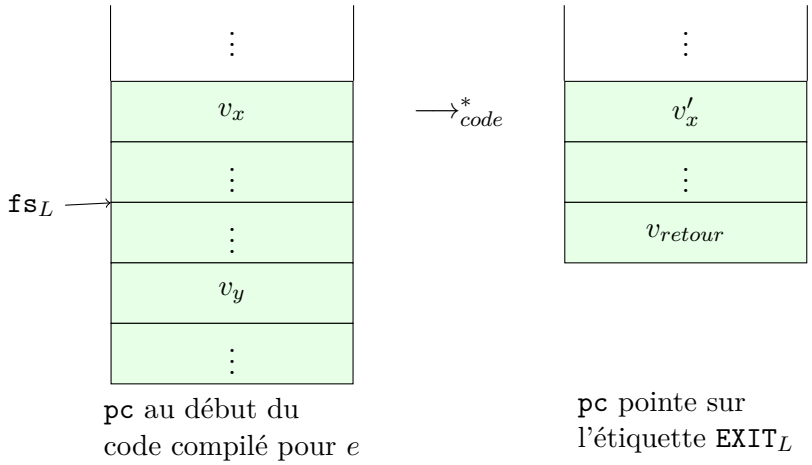

FIGURE 6.10 – Fonctionnement du code compilant une expression e en cas de retour vers L

Nous compilons ensuite les expressions de manière récursive, en associant à chaque expression un code construit à partir des codes obtenus pour les sous-expressions. Une expression compilée correctement fonctionne de la manière suivante. Si l'on part du début du code d'une expression, alors exécuter ce code doit être équivalent à effectuer les effets de l'expression et à empiler le résultat de l'expression sur la pile, ou à sauter à la fin d'un bloc englobant en cas de retour. Dans le second cas, il faut restaurer la taille de la pile à celle du début du bloc avant d'empiler le résultat. Le comportement attendu de la pile dans chaque cas de figure est décrit respectivement dans les figures 6.9 et 6.10.

Cependant, pour compiler correctement une expression nous avons besoin de certaines informations sur le contexte dans lequel son code va être exécuté. Nous devons notamment être en mesure de retrouver l'endroit où les variables locales sont stockées. Celles-ci sont allouées à des endroits fixes du tableau d'activation de l'appel couramment exécuté, c'est-à-dire la partie de la pile réservée pour l'exécution de l'appel en question. Nous devons également pouvoir effectuer correctement un retour. Nous effectuons ceux-ci en restaurant directement la taille du tableau d'activation au niveau d'avant le bloc/l'appel associé(e) avant de sauter directement à la fin de celui-ci.

Nous devons donc faire dépendre la traduction d'une expression de plusieurs paramètres :

— L'association de chaque nom de fonction à une étiquette, dans le but de traduire la prise d'adresse d'une fonction. Comme ce paramètre est constant au cours de la compilation d'un programme donné, nous allons en général omettre ce paramètre.
— La taille du tableau d'activation lorsque le point d'entrée de l'expression est atteint. Nous notons cette valeur `fs`.
— Pour chaque variable définie dans le contexte de l'expression, la distance entre le début de la partie réservée à la fonction courante et la cellule associée à ladite variable. Nous notons cette association σ_v.

— Pour chaque étiquette définie dans le contexte de l'expression, la taille du tableau d'activation avant l'exécution du bloc qui l'a engendré, ainsi que l'étiquette du point de sortie de ce bloc. Nous notons cette association σ_l.

Nous pouvons alors donner le schéma de compilation complet d'un programme. Nous commençons par donner le schéma de compilation d'une expression e, règle par règle. Nous notons le résultat de la compilation d'une expression par $[\![e]\!]_{\texttt{fs},\sigma_v,\sigma_l}$. Nous donnons ensuite le schéma de compilation d'une fonction, puis enfin d'un programme complet.

À chaque fois que le schéma de compilation introduit une étiquette, nous supposons qu'il s'agit d'un nom frais.

Cas des variables Il suffit d'utiliser l'instruction `load` pour récupérer la valeur de la variable. La localisation relative est obtenue comme différence des localisations absolues.

$$[\![x]\!]_{\texttt{fs},\sigma_v,\sigma_l} = \texttt{load } (\texttt{fs} - \sigma_v(x))$$

Cas de l'expression vide Il faut simplement ajouter un élément sur la pile, n'importe lequel. Comme nous pouvons garantir qu'une expression ne sera jamais exécutée dans une pile vide, nous allons simplement dupliquer le sommet de la pile.

$$[\![()]\!]_{\texttt{fs},\sigma_v,\sigma_l} = \texttt{load } 0$$

Cas de l'affectation Le cas de l'affectation est similaire à celui d'une variable, à ceci près qu'il faut faire attention à ce que l'usage d'autres instructions auparavant augmente la taille de la pile. Il faut également s'assurer de laisser une valeur (n'importe laquelle, ici celle renvoyée par l'expression sous-jacente) sur la pile.

$$[\![x \leftarrow e]\!]_{\texttt{fs},\sigma_v,\sigma_l} = \begin{cases} [\![e]\!]_{\texttt{fs},\sigma_v,\sigma_l} \\ \texttt{load } 0 \\ \texttt{store } (\texttt{fs} + 1 - \sigma_v(x)) \end{cases}$$

Cas de la prise d'adresse d'une fonction Pour compiler une prise d'adresse $\&f$, il suffit de charger l'étiquette $\texttt{START}_f$ associée à f au sommet de la pile.

$$[\![\&f]\!]_{\texttt{fs},\sigma_v,\sigma_l} = \texttt{aload START}_f$$

Cas d'un appel de fonction Pour compiler un appel de fonction, nous commençons par évaluer les expressions qui le composent, dans l'ordre. Nous devons faire attention à ajuster la taille du tableau d'activation qui croît au fur à et mesure. Ensuite, nous récupérons l'étiquette correspondant à la fonction appelée au sommet de la pile, puis nous plaçons à sa précédente position l'adresse de retour attendue, de manière à structurer la pile comme attendu au début de la fonction appelée. Enfin, nous ajoutons une instruction de saut pour effectuer l'appel proprement dit, puis l'emplacement du retour.

$$[\![e_0((e_i)_{0<i\leq n})]\!]_{\mathtt{fs},\sigma_v,\sigma_l} = \left\{ \begin{array}{lll} & [\![e_0]\!]_{\mathtt{fs},\sigma_v,\sigma_l} & \\ & \vdots & \\ & [\![e_i]\!]_{\mathtt{fs}+i,\sigma_v,\sigma_l} & \\ & \vdots & \\ & [\![e_n]\!]_{\mathtt{fs}+n,\sigma_v,\sigma_l} & \\ & \mathtt{load}\ n & \text{(Charge la valeur de } e_0\text{)} \\ & \mathtt{aload\ RA} & \text{(Adresse de retour)} \\ & \mathtt{store}\ (n+1) & (\mathtt{RA} \text{ à la position originelle de } e_0) \\ & \mathtt{jmp} & \text{(saut à la valeur de } e_0\text{)} \\ \mathtt{RA:} & \mathtt{nop} & \text{(point de retour)} \end{array} \right.$$

Cas d'un retour Pour compiler un retour, nous commençons par stocker le résultat de l'expression à l'endroit prévu pour le retour du bloc englobant. Nous restaurons ensuite la taille du tableau d'activation, puis nous sautons directement à la fin du bloc. Les informations nécessaires $(\mathtt{fs}_L, \mathtt{EXIT}_L)$ sont obtenues directement à partir de σ_l. Remarquons que dans le cas particulier où $\mathtt{fs}_L$ est égal à la taille courante ($\mathtt{fs}$) du tableau d'activation, nous n'avons pas besoin d'effectuer ces manipulations avant le saut. Nous ne générons donc pas ces instructions si $\mathtt{fs} = \mathtt{fs}_L$ (celles-ci ne fonctionneraient d'ailleurs pas dans ce cas, puisque $0 - 1$ n'est pas un entier naturel). Comme nous réutiliserons ce même schéma plus tard, nous résumons le processus de coupe de la pile sous la valeur de retour par la pseudo-instruction $\mathtt{cut}\ n$, correspondant à l'élimination des n valeurs juste en dessous du sommet de la pile.

$$[\![\mathtt{return}[L]\ e]\!]_{\mathtt{fs},\sigma_v,\sigma_l} = \left\{ \begin{array}{ll} [\![e]\!]_{\mathtt{fs},\sigma_v,\sigma_l} & \\ \mathtt{cut}\ (\mathtt{fs} - \mathtt{fs}_L) & \text{où } (\mathtt{fs}_L, \mathtt{EXIT}_L) = \sigma_l(L) \\ \mathtt{aload\ EXIT}_L & \\ \mathtt{jmp} & \end{array} \right.$$

$$\mathtt{cut}\ n = \left\{ \begin{array}{ll} \epsilon & \text{si } n = 0 \\ \mathtt{store}\ (n-1) & \\ \mathtt{pop}\ (n-1) & \text{sinon} \end{array} \right.$$

Cas d'un bloc Pour compiler un bloc, nous commençons par allouer de l'espace pour les variables locales, initialisé par des valeurs quelconques. Nous exécutons ensuite l'expression interne du bloc. Pour cela, nous devons modifier les paramètres de la compilation pour prendre en compte l'agrandissement du tableau d'activation, les positions des nouvelles variables dans ce tableau, et les informations de sortie associées à l'étiquette du bloc. Une fois que cette expression est terminée, nous faisons une passe de nettoyage similaire à celle du retour. Finalement, nous ajoutons une étiquette tout à la fin pour récupérer les retours.

$$[\![[L]\{\ (x_i)_{0\leq i<n} : e\ \}]\!]_{\mathtt{fs},\sigma_v,\sigma_l} = \left\{ \begin{array}{ll} (n \text{ fois}) & \mathtt{load\ 0} \\ & [\![e]\!]_{\mathtt{fs}+n,\sigma_v[(x_i \leftarrow \mathtt{fs}+i)_{0\leq i<n}],\sigma_l[L \leftarrow (\mathtt{fs},\mathtt{EXIT})]} \\ & \mathtt{cut}\ n \\ \mathtt{EXIT:} & \mathtt{nop} \end{array} \right.$$

Cas de la mise en séquence Le schéma de compilation est direct, nous devons juste faire attention à éliminer le retour de e_1.

$$[\![e_1; e_2]\!]_{\mathtt{fs},\sigma_v,\sigma_l} = \begin{cases} [\![e_1]\!]_{\mathtt{fs},\sigma_v,\sigma_l} \\ \mathtt{pop}\ 1 \\ [\![e_2]\!]_{\mathtt{fs},\sigma_v,\sigma_l} \end{cases}$$

Cas de l'expression conditionnelle Le schéma de compilation d'une expression conditionnelle est un schéma standard en trois parties reliées par des instructions de saut.

$$[\![\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3]\!]_{\mathtt{fs},\sigma_v,\sigma_l} = \begin{cases} & [\![e_1]\!]_{\mathtt{fs},\sigma_v,\sigma_l} \\ & \mathtt{jnz\ THEN} \\ & [\![e_3]\!]_{\mathtt{fs},\sigma_v,\sigma_l} \\ & \mathtt{aload\ EXIT} \\ & \mathtt{jmp} \\ \mathtt{THEN:} & \mathtt{nop} \\ & [\![e_2]\!]_{\mathtt{fs},\sigma_v,\sigma_l} \\ \mathtt{EXIT:} & \mathtt{nop} \end{cases}$$

Cas d'une boucle Pour compiler la construction de boucle, nous utilisons un schéma qui inverse l'ordre du test et du corps de boucle, de manière à n'utiliser qu'une instruction de saut par itération. L'exécution d'une itération s'effectue donc par la séquence « corps de boucle, test, saut conditionnel », où le saut final revient au début du corps de boucle si le test est valide. Comme nous devons cependant commencer par le test, nous ajoutons au préalable une séquence d'instruction pour sauter directement au point intermédiaire entre le corps de la boucle et le test.

$$[\![\mathtt{while}\ e_1\ \mathtt{do}\ e_2]\!]_{\mathtt{fs},\sigma_v,\sigma_l} = \begin{cases} & \mathtt{aload\ TEST} \\ & \mathtt{jmp} \\ \mathtt{BODY:} & \mathtt{nop} \\ & [\![e_2]\!]_{\mathtt{fs},\sigma_v,\sigma_l} \\ & \mathtt{pop}\ 1 \\ \mathtt{TEST:} & \mathtt{nop} \\ & [\![e_1]\!]_{\mathtt{fs},\sigma_v,\sigma_l} \\ & \mathtt{jnz\ BODY} \end{cases}$$

Cas d'une déclaration de fonction Nous compilons une déclaration de fonction $f[L]((x_i)_{0\leq i<n}) = e$ de la manière suivante. Tout d'abord, nous récupérons l'étiquette $\mathtt{START}_f$ associée à f, et l'utilisons pour marquer le début du code associé à f. Nous générons ensuite le code associé au corps de la fonction, en initialisant correctement les paramètres de la compilation. À l'exception de la génération de l'espace pour les variables (puisque celui-ci est déjà alloué), ce code suit la même structure que celui d'un bloc. À l'issue de celui-ci, le tableau d'activation de la fonction ne contient que l'adresse de retour et la valeur de retour. L'adresse de retour est bien conservée car nous initialisons la taille du tableau d'activation à 1. Nous ajoutons quelques instructions pour échanger ces deux dernières valeurs, puis une instruction de saut pour revenir au site d'appel.

$$
\begin{array}{rl}
\texttt{START}_f\texttt{:} & \texttt{nop} \\
& [\![e]\!]_{n+1,[(x_i \to i+1)_{0 \leq i < n}],[L \to (1,\texttt{EXIT})]} \\
& \texttt{cut } n \\
\texttt{EXIT:} & \texttt{swap} \\
& \texttt{jmp}
\end{array}
$$

Compilation d'un programme complet Pour compiler un programme complet, nous commençons par créer l'association entre les fonctions (pré)définies par le programme et les étiquettes. Nous devons garantir que toute étiquette identifie au plus une fonction, ce que nous pouvons obtenir en générant un nom frais pour chaque fonction. Nous générons ensuite le code associé à chacune de ces fonctions. Le code d'un programme complet est obtenu de la manière suivante. Nous récupérons l'étiquette `MAIN` associée au point d'entrée du programme, puis nous générons le code correspondant à un appel à ce point d'entrée suivi de l'arrêt du programme. Enfin, nous ajoutons à la suite de ce fragment les codes obtenus pour toutes les fonctions (pré)définies par le programme, dans n'importe quel ordre.

$$
\begin{array}{rl}
 & \texttt{aload EXIT} \\
 & \texttt{aload MAIN} \\
 & \texttt{jmp} \\
\texttt{EXIT:} & \texttt{exit} \\
\text{(pour chaque fonction } f\text{)} & \text{(code associé à } f\text{)}
\end{array}
$$

6.2 Énoncés de correction

Pour prouver la correction du compilateur, nous devons d'abord préciser quel type de résultat nous souhaitons obtenir. Intuitivement, la compilation d'un programme est correcte si le programme généré se comporte de la même manière que le programme source. En présence de non-déterminisme, nous pouvons donner deux sens différents à cette propriété intuitive. La première interprétation est la *simulation en avant*, autrement dit que tout comportement du programme source est un comportement possible du programme compilé. La seconde interprétation, la *simulation en arrière*, exprime la propriété inverse, c'est-à-dire que tout comportement du programme compilé est un comportement possible du programme source.

Nous allons ici démontrer les deux propriétés pour notre schéma de compilation, sous réserve de leur établissement pour les opérations prédéfinies. Plus précisément, nous allons vérifier les deux théorèmes suivants.

Définition 6.7 (Exécution infinie)**.** Pour un état X de la machine associée, nous disons qu'un programme (du langage source ou cible) admet une *exécution infinie* $X \longrightarrow^{\infty} \Pi_\infty$ à partir de X s'il existe une séquence infinie $(X_i)_{i \in \mathbb{N}}$ telle que

(i) $X_0 = X$

(ii) $\forall i \in \mathbb{N}, X_i \longrightarrow X_{i+1}$

(iii) Π_∞ est la borne supérieure pour l'ordre d'extension de tous les canaux entrée/sortie présents dans les X_i. Autrement dit, Π_∞ correspond à la totalité des évènements d'entrée/sortie qui se sont produits au long de cette exécution non terminante.

Théorème 6.8 (Résultat de simulation en avant)**.** *Pour tout programme source bien formé F, si le résultat de la compilation est le programme cible c, alors les deux conditions suivantes sont vérifiées :*

(i) Pour toute exécution finie

$$(\&\texttt{main}(), (\Sigma_0, \Pi_0), \top) \longrightarrow_F^* (v, (\Sigma_1, \Pi_1), \top)$$

du programme source, pour toute pile S_0, la machine virtuelle admet une exécution finie

$$(0, S_0, \Pi_0) \longrightarrow_c^* (\texttt{pc}, (S_0; w), \Pi_1)$$

pour le programme c, où `pc` *désigne l'instruction* `exit` *dans c et où la valeur w représente v.*

(ii) Pour toute exécution infinie

$$(\&\texttt{main}(), (\Sigma_0, \Pi_0), \top) \longrightarrow_F^\infty \Pi_\infty$$

du programme source, pour toute pile S_0 la machine virtuelle admet une exécution infinie

$$(0, S_0, \Pi_0) \longrightarrow_c^\infty \Pi_\infty$$

pour le programme c.

Théorème 6.9 (Résultat de simulation en arrière)**.** *Pour tout programme source bien formé F, pour tout environnement (Σ_0, Π_0) tel que l'exécution de la machine abstraite à partir de $(\&\texttt{main}(), (\Sigma_0, \Pi_0), \top)$ ne puisse pas atteindre d'état bloquant, si le résultat de la compilation est le programme cible c, alors les deux conditions suivantes sont vérifiées :*

(i) Pour toute exécution finie

$$(0, S_0, \Pi_0) \longrightarrow_c^* (\texttt{pc}, S_1, \Pi_1)$$

de la machine virtuelle pour le code c telle que l'exécution ne puisse plus progresser, alors `pc` *désigne une instruction* `exit` *dans c, la pile S_1 s'écrit sous la forme $S_0; w$, et le programme source admet une exécution finie*

$$(\&\texttt{main}(), (\Sigma_0, \Pi_0), \top) \longrightarrow_F^* (v, (\Sigma_0, \Pi_1), \top)$$

telle que la valeur w représente v.

(ii) Pour toute exécution infinie

$$(0, S_0, \Pi_0) \longrightarrow_c^\infty \Pi_\infty$$

de la machine virtuelle pour le code c, alors le programme source admet une exécution infinie

$$(\&\texttt{main}(), (\Sigma_0, \Pi_0), \top) \longrightarrow_F^\infty \Pi_\infty$$

Remarquons que pour le résultat de simulation en arrière, nous imposons la contrainte supplémentaire que le programme source n'admette pas d'exécution incorrecte. Ce type de restriction est nécessaire car a priori le code compilé ne s'arrête pas (ou pas correctement) en cas de comportement incorrect du code source (comme par exemple un comportement indéfini dans les langages type `C`). Nous pourrions

également autoriser la machine virtuelle à atteindre une situation en correspondance avec un état bloquant de l'exécution source, mais ce type de résultat peut être incompatible avec un schéma de compilation optimisant. En effet, certaines optimisations effectuées par les compilateurs reposent sur des informations déduites de l'absence d'états bloquants dans l'exécution du programme source. Bien que nous n'effectuions pas ce type d'optimisation ici, nous ne souhaitons pas pour autant les interdire.

Pour obtenir ces deux résultats, nous allons exploiter les deux encodages locaux des systèmes de transition donnés dans la section 4.5.3. Ces encodages permettent de transformer les résultats de simulation voulus en un ensemble de garanties sur deux jeux distincts. Nous transformons ensuite cet ensemble de garanties en un énoncé étiqueté, que nous établissons en utilisant le système de combinaison de transformateurs donné dans la section 5.3. Nous montrons récursivement sur la syntaxe des programmes sources que chaque règle de compilation établit un énoncé étiqueté correspondant à l'objet compilé, et récupérons au final les garanties voulues.

L'approche que nous proposons s'applique selon un schéma identique pour les deux résultats. Nous commençons par montrer le fonctionnement de cette approche pour démontrer le résultat de simulation en arrière.

6.3 Preuve de la simulation en arrière

6.3.1 Énoncé équivalent en termes de garanties

La conversion de la sémantique du langage cible en un système de transition ordonné est immédiate, puisqu'il s'agit d'une sémantique à petit pas et que la partie ordonnée correspond au canal d'entrée/sortie.

Définition 6.10 (Système de transition ordonné associé au langage cible)**.** Soit c le code d'un programme cible. Le *système de transition ordonné associé* à c est donné par :

- l'ensemble des états est celui des états de la machine virtuelle.
- la relation de transition est celle de la machine virtuelle.
- l'ensemble ordonné correspondant est celui des séquences potentiellement infinies d'évènements d'entrée/sortie, muni de l'ordre d'extension.
- la projection d'un état de la machine virtuelle vers un élément de l'ensemble ordonné consiste à isoler le canal d'entrée/sortie.

Nous convertissons ensuite le système de transition ordonné obtenu en jeu. Comme le résultat de simulation en arrière correspond à une propriété universelle sur les exécutions du langage cible, nous utilisons la traduction du système de transition correspondant à ce type de propriété.

Définition 6.11 (Jeu universel associé au langage cible)**.** Soit c le code d'un programme cible. Le *jeu universel associé* à c est le jeu $\mathbb{G}_{c,\forall}$ défini comme le jeu universel local (cf. définition 4.77) associé au système de transition ordonné associé à c.

Nous pouvons alors appliquer le lemme 4.82, correspondant à la traduction des garanties dans le jeu universel local, pour reformuler le résultat de simulation en arrière comme la validité de certaines garanties. L'ensemble des garanties ainsi obtenu correspond à la structure d'un contrat. Ce contrat exprime le fait que l'exécution du

programme compilé produit une trace d'exécution du programme source, dont les états initiaux et finaux correspondent à ceux de l'exécution du programme compilé.

Définition 6.12 (Contrat pour la simulation en arrière)**.** Soit F un programme source et c un programme cible. Le *contrat nommé correspondant à la simulation en arrière de F par c*, noté $\overleftarrow{\mathcal{C}}(F, c)$, est le contrat nommé défini vis-à-vis du support de $\mathbb{G}_{c,\forall}$ par :

(A) l'unique paramètre auxiliaire est une association de variables Σ_0.

(B) l'unique valeur de retour auxiliaire est une valeur source v_R.

(P) la pré-condition sur l'état initial est que cet état soit de la forme $((0, S_0, \Pi_0), t)$, et que l'exécution du programme source à partir de l'état

$$(\&\texttt{main}(), (\Sigma_0, \Pi_0), \top)$$

ne puisse pas atteindre un état bloquant.

(Q) la post-condition sur l'état final exprime que s'il s'agit d'un état limite correspondant au canal d'entrée/sortie Π_∞, alors

$$(\&\texttt{main}(), (\Sigma_0, \Pi_0), \top) \longrightarrow_F^\infty \Pi_\infty$$

et que s'il s'agit d'un état non limite, alors il est de la forme $((\texttt{pc}, (S_0; w_R), \Pi_1), t)$, où S_0 correspond à la pile de l'état initial, et

(i) w_R représente la valeur source v_R.

(ii) `pc` désigne l'instruction `exit` au sein de c.

(iii) $(\&\texttt{main}(), \Sigma_0, \Pi_0), \top) \longrightarrow_F^* (v, (\Sigma_0, \Pi_1), \top)$

Lemme 6.13. *L'énoncé du théorème 6.9 est équivalent à ce que pour tout programme source bien formé F, si le résultat de la compilation est le programme cible c, alors toutes les garanties associées au contrat nommé $\overleftarrow{\mathcal{C}}(F, c)$ sont valides dans $\mathbb{G}_{c,\forall}$.*

Démonstration. Application directe du lemme 4.82. □

Comme l'ensemble de garanties correspondant au résultat de simulation en arrière est décrit par un contrat nommé, nous pouvons facilement convertir la validité de ces garanties en un énoncé nommé. Il nous suffit pour cela de compléter le contrat par un contexte étiqueté $\Gamma_{c,\forall}$ équivalent au jeu $\mathbb{G}_{c,\forall}$, équipé d'associations de noms pour les valeurs auxiliaires. Comme nous avons seulement besoin de déduire les garanties depuis des énoncés nommés valides, nous ne vérifierons que le sens utile (et facile) de cette équivalence, c'est-à-dire que le jeu $\mathbb{G}_{c,\forall}$ satisfait le contexte $\Gamma_{c,\forall}$. Bien entendu, ledit contexte nommé devra néanmoins être suffisamment précis par rapport au comportement du langage cible pour pouvoir effectuer la preuve, et sera donc sans doute équivalent.

Pour prouver la correction de notre schéma de compilation, la seule partie où la définition précise du langage cible intervient est au niveau de l'exécution d'une instruction. Nous utilisons donc les instructions pour structurer $\Gamma_{c,\forall}$. Plus précisément, nous définissons ce contexte comme l'association de chaque instruction à la spécification de son comportement dans le jeu sous-jacent. Les instructions du langage correspondent donc à des noms d'hypothèses.

$$\forall n \in \mathbb{N}. \left\langle \begin{pmatrix} \mathsf{now}\text{ s'écrit} \\ ((\mathtt{pc}, S, \Pi), t) \\ \text{avec } |S| \geq n \\ \wedge\ I_{\mathtt{pc}} = \mathtt{load}\ \ n \end{pmatrix} \hookrightarrow \begin{pmatrix} \mathsf{now} = ((\mathtt{pc}+1, (S; v_0), \Pi), t+1) \\ \text{si } \mathsf{old}\text{ s'écrit } ((\mathtt{pc}, S, \Pi), t) \\ \text{et si } S \text{ s'écrit } S'; (v_i)_{0 \leq i \leq n} \end{pmatrix} \right\rangle$$

FIGURE 6.11 – Contrat pour `load` vis-à-vis du programme $c = (I_l)_{0 \leq l < N}$

$$\left\langle \begin{pmatrix} \mathsf{now}\text{ s'écrit} \\ ((\mathtt{pc}, S, \Pi), t) \\ \text{avec } I_{\mathtt{pc}} = \mathtt{in} \end{pmatrix} \hookrightarrow \begin{pmatrix} v_R : & \mathsf{now} = ((\mathtt{pc}+1, (S; v_R), \Pi +\!\!+ I(n)), t+1) \\ & \text{si } \mathsf{old}\text{ s'écrit } ((\mathtt{pc}, S, \Pi), t) \text{ et } v_R \text{ représente } n \end{pmatrix} \right\rangle$$

FIGURE 6.12 – Contrat pour `in` vis-à-vis du programme $c = (I_l)_{0 \leq l < N}$

Les contrats suivent immédiatement de la définition de la relation de transition de la machine virtuelle. Nous utilisons comme pré-condition les conditions sous lesquelles l'instruction peut être exécutée, et comme post-condition l'ensemble des états qui peuvent être atteints par exécution de l'instruction à partir de l'état initial donné. Pour les instructions déterministes, les paramètres auxiliaires des contrats sont donnés par les paramètres des instructions, et nous n'utilisons pas de valeurs de retour auxiliaires. Nous donnons un tel exemple de contrat nommé dans la figure 6.11.

Le cas de l'instruction `in` est particulier, à cause de son comportement non-déterministe. Nous donnons son contrat explicitement dans la figure 6.12, en utilisant la valeur de retour auxiliaire pour lier l'entier obtenu.

Remarque 6.14. Comme le code et la tête de lecture permettent de reconstruire l'instruction à exécuter, nous pourrions très bien donner un unique contrat correspondant à un pas d'exécution de la machine virtuelle, et en éliminer tout paramètre auxiliaire. Cependant, cela génère des branchements de grande taille dans les pré- et post-conditions, qui polluent les conditions de vérification exploitant ce contrat. Cela peut rendre les choses plus difficiles pour une preuve mécanisée, en particulier pour une preuve automatique. Nous estimons que cette réduction à un unique contrat n'en vaut pas la peine puisque préciser l'instruction quand on utilise ce contrat ne nous demande aucun effort particulier, et possède le mérite de clarifier les transformateurs.

Au sein des interfaces des transformateurs utilisés, nous utilisons la notation $\mathcal{I}_{\mathtt{asm},\forall}$ pour représenter l'association d'hypothèses correspondant aux contrats de $\Gamma_{c,\forall}$.

Lemme 6.15. *Toutes les garanties du contexte $\Gamma_{c,\forall}$ ainsi obtenues sont satisfaites par $\mathbb{G}_{c,\forall}$.*

Démonstration. Il s'agit d'une conséquence directe du lemme 4.82. En effet, considérons une trace d'évolution dont le point de départ appartient à la pré-condition

associée. Si la trace d'évolution ne possède qu'un élément, la relation de transition de la machine virtuelle montre que cette trace admet un prolongement. Dans le cas où cette trace possède un deuxième élément, la relation de transition de la machine virtuelle montre que ce deuxième élément appartient à la post-condition. □

Nous obtenons alors l'équivalence attendue.

Définition 6.16. Soit F un programme source et c un programme cible. L'*énoncé nommé correspondant à la simulation en arrière de F par c*, noté $\overleftarrow{\mathcal{E}}(F, c)$, est l'énoncé nommé défini vis-à-vis du support de $\mathbb{G}_{c,\forall}$ par :

— le contexte $\Gamma_{c,\forall}$ équipé de l'association d'hypothèse $\mathcal{I}_{\mathtt{asm},\forall}$,

— le contrat $\overleftarrow{\mathcal{C}}(F, c)$.

Lemme 6.17. *L'énoncé du théorème 6.9 est équivalent à ce que pour tout programme source bien formé, si le résultat de la compilation est le programme c, alors l'énoncé nommé $\overleftarrow{\mathcal{E}}(F, c)$ est valide.*

Démonstration. Conséquence immédiate des lemmes 6.13 et 6.15. □

6.3.2 Réduction à la compilation correcte des fonctions

Nous avons maintenant réduit la preuve de la correction du schéma de compilation à la preuve de la validité d'un énoncé nommé dépendant du code source et du code compilé. Il nous reste maintenant à effectuer ladite preuve. Pour cela, remarquons que le résultat final doit être une conséquence quasi immédiate de la compilation correcte des fonctions. En effet, le code final correspond à un appel à la fonction `main`. Si l'on obtient pour cet appel un énoncé nommé proche de l'énoncé nommé final, alors nous pouvons passer par les transformateurs pour le combiner avec les quatre instructions encadrant l'appel. Cette combinaison correspond à l'application à plusieurs reprises du transformateur de liaison séquentielle. Les conditions de vérification correspondant au passage à l'énoncé nommé final seront a priori triviales.

Nous commençons donc par donner le contrat type que nous souhaitons établir pour un appel de fonction. Celui-ci se déduit du protocole d'appel et du résultat voulu.

Définition 6.18 (Contrat associé à une fonction source)**.** Soit f une fonction (pré)définie par F, $\mathtt{START}_f$ l'étiquette associée à f, et $(c_i)_{i\in I}$ une famille de programmes $\mathcal{L}_\mathtt{t}$. Le *contrat nommé associé* à f et à la famille $(c_i)_{i\in I}$ est le contrat $\mathcal{C}_{fun}(F, f, (c_i)_{i\in I})$ défini par :

(A) les paramètres auxiliaires $(c_{glob}, \mathtt{pc}_R, (v_i)_{0\leq i<n}, \Sigma_0, k)$ constitués d'un programme c_{glob} du langage $\mathcal{L}_\mathtt{t}$ dit code global, d'une tête de lecture $\mathtt{pc}_R$, de n valeurs sources $(v_i)_{0\leq i<n}$, d'une association de variables Σ_0, et d'un code de continuation k correspondant à des parties de l'état de la machine abstraite du langage source.

(B) la valeur de retour auxiliaire v_R correspondant à une valeur source.

(P) une pré-condition constituée de l'ensemble des états de $\mathbb{G}_{c_{glob},\forall}$ de la forme

$$((\mathtt{pc}_f, (S_0; \mathtt{RA}; (w_i)_{0\leq i<n}), \Pi_0), t)$$

tels que

(i) l'exécution du programme source F à partir de l'état

$$((), (\Sigma_0, \Pi_0), \&f((v_i)_{0\leq i<n}, k))$$

ne peut pas atteindre d'état bloquant.
(ii) pour tout $i \in I$, c_i est une sous-séquence de c_{glob}.
(iii) $\texttt{pc}_f$ correspond à la position de $\texttt{START}_f$ dans c_{glob}.
(iv) $\texttt{pc}_R$ correspond à la position de `RA` dans c_{glob}.
(v) les valeurs cible $(w_i)_{0\leq i<n}$ représentent correctement les valeurs source $(v_i)_{0\leq i<n}$.

(Q) une post-condition donnée par la réunion de l'ensemble des canaux d'entrée/sortie Π_∞ tels que

$$((), (\Sigma_0, \Pi_0), \&f((v_i)_{0\leq i<n}, k)) \longrightarrow_F^\infty \Pi_\infty$$

et de l'ensemble des états de $\mathbb{G}_{c_{glob},\forall}$ de la forme

$$((\texttt{pc}_R, (S_0; w), \Pi_1), t')$$

où S_0 correspond au fragment du même nom de l'état d'entrée, $\texttt{pc}_R$ au paramètre auxiliaire de même nom, et tels que
(i) $((), (\Sigma_0, \Pi_0), \&f((v_i)_{0\leq i<n}, k)) \longrightarrow_F^* (v_R, (\Sigma_0, \Pi_1), k)$
(ii) la valeur cible w représente correctement la valeur source v_R.

La famille de codes $(c_i)_{i\in I}$ que nous avons ajoutée en paramètre de la définition correspond intuitivement aux sections du code définissant f. À cause du caractère mutuellement récursif des fonctions de $\mathcal{L}_\texttt{s}$, une fonction compilée peut effectivement voir sa définition complète (comprenant les fonctions pouvant être appelées) être éclatée en plusieurs morceaux. En pratique, cela nous oblige à énoncer le contrat d'une fonction à partir des codes générés pour toutes les fonctions du programme.

Remarque 6.19. Le paramètre c_{glob} représente le code final produit par la compilation. Il semblerait donc que nous puissions transformer ce paramètre en une constante, correspondant à ce code final. Cependant, c'est incompatible avec la méthodologie de preuve mécanisée présentée dans la section 2.5, puisque l'on doit démontrer ces énoncés dans un contexte où ce code final n'a pas encore été construit. Nous quantifions donc sur toutes les possibilités.

Nous vérifions maintenant l'affirmation selon laquelle la correction du programme compilé complet se déduit directement de celles des fonctions compilés.

Définition 6.20 (Compilation correcte d'une fonction)**.** Soit F le code du programme source, f une fonction (pré)définie dans F, et (c_g) la famille des codes compilés pour les fonctions (pré)définies dans F. Nous disons que f est compilée de manière *correcte* si l'énoncé nommé $\mathcal{E}_{glob}(F, f, (c_g))$ composé
— du contrat nommé $\mathcal{C}_{fun}(F, f, (c_g))$
— du contexte $\Gamma_{c_{glob},\forall}$, dépendant du paramètre auxiliaire c_{glob} dit « code global » du contrat nommé $\mathcal{C}_{fun}(F, f, (c_g))$
est valide.

Lemme 6.21. *Soit F un programme source, compilé vers le programme c, ainsi que (c_f) la famille des codes générés séparément pour chaque fonction (pré)définie par F. Supposons que la fonction* `main` *soit compilée de manière correcte. Alors le résultat de simulation en arrière est vérifié.*

$$
\begin{array}{l}
\forall \Sigma_0, \mathsf{now}_0.\ P_{\overleftarrow{\mathcal{C}}}(\Sigma_0, \mathsf{now}_0) \Rightarrow \\
\quad P_{\mathit{aload}}(\mathtt{EXIT}, \mathsf{now}_0) \\
\quad \wedge \left(\begin{array}{l}
\forall \mathsf{now}_1.\ Q_{\mathit{aload}}(\mathtt{EXIT}, \mathsf{now}_0, \mathsf{now}_1) \Rightarrow \\
P_{\mathit{aload}}(\mathtt{MAIN}, \mathsf{now}_1) \\
\quad \wedge \left(\begin{array}{l}
\forall \mathsf{now}_2.\ Q_{\mathit{aload}}(\mathtt{MAIN}, \mathsf{now}_1, \mathsf{now}_2) \Rightarrow \\
P_{\mathit{jmp}}(\mathsf{now}_2) \\
\quad \wedge \left(\begin{array}{l}
\forall \mathsf{now}_3.\ Q_{\mathit{jmp}}(\mathsf{now}_2, \mathsf{now}_3) \Rightarrow \\
P_{\mathtt{main},(c_f)}(c, \mathtt{pc}_{\mathtt{EXIT}}, (x)_{x\in\emptyset}, \Sigma_0, \top, \mathsf{now}_3) \\
\wedge\ \Gamma_{c,\forall} \subseteq \Gamma_{c,\forall} \\
\quad \wedge \left(\begin{array}{l}
\forall v_R, \mathsf{now}_4. \\
\quad \wedge \left(\begin{array}{l}
Q_{\mathtt{main}}(c, \mathtt{pc}_{\mathtt{EXIT}}, (x)_{x\in\emptyset}, \Sigma_0, \top, \mathsf{now}_3, v_R, \mathsf{now}_4) \Rightarrow \\
Q_{\overleftarrow{\mathcal{C}}}(\Sigma_0, \mathsf{now}_0, v_R, \mathsf{now}_4)
\end{array} \right)
\end{array} \right)
\end{array} \right)
\end{array} \right)
\end{array} \right)
\end{array}
$$

FIGURE 6.13 – Structure de la condition de vérification pour passer de la validité du contrat associé à la fonction principale au résultat de correction final

Démonstration. Par le lemme 6.17, il suffit de démontrer la validité de l'énoncé nommé $\overleftarrow{\mathcal{E}}(F, c)$. Nous allons donc le déduire de celle de l'énoncé $\mathcal{E}_{glob}(F, \mathtt{main}, (c_f))$. Essentiellement, nous appliquons le transformateur

$$
\begin{array}{l}
\left[\begin{array}{l} \mathcal{I}_{\mathtt{asm},\forall} \\ \Sigma_0 \end{array} \right] \Rightarrow \left[\begin{array}{l} v_R \end{array} \right] \\
\mathit{aload}[\mathtt{L} \leftarrow \mathtt{EXIT}] \\
\mathit{aload}[\mathtt{L} \leftarrow \mathtt{MAIN}] \\
\mathit{jmp} \\
W_{\mathcal{E}_{glob}(F,\mathtt{main},(c_f))}[c_{glob} \leftarrow c, \mathtt{pc}_R \leftarrow \mathtt{pc}_{\mathtt{EXIT}}, (v_i)_{0\le i<0} \leftarrow (x)_{x\in\emptyset}, k \leftarrow \top]
\end{array}
$$

à $\overleftarrow{\mathcal{E}}(F, c)$. Dans ce transformateur, nous avons utilisé la notation $\mathtt{pc}_{\mathtt{EXIT}}$ pour la position de l'étiquette `EXIT` dans c, que nous pouvons obtenir exactement depuis le code produit, et $W_{\mathcal{E}_{glob}(F,\mathtt{main},(c_f))}$ pour le transformateur nommé correspondant à la conversion de $\mathcal{E}_{glob}(F, \mathtt{main}, (c_f))$.

Il suffit alors de démontrer la condition de vérification générée par l'application de ce transformateur, ce qui est quasi immédiat une fois cette application complètement déroulée. Cette condition de vérification possède la structure donnée dans la figure 6.13, en utilisant les P_i pour dénoter les pré-conditions et les Q_i pour les post-conditions. Il s'agit essentiellement de vérifier que la post-condition de $\overleftarrow{\mathcal{C}}(F, c)$, ainsi que toute pré-condition d'une instruction/de $\mathcal{C}_{fun}(F, c, (c_f))$, est déductible de la pré-condition de $\overleftarrow{\mathcal{C}}(F, c)$ et des post-conditions des instructions/de $\mathcal{C}_{fun}(F, c, (c_f))$ qui précèdent.

La majorité de ces conditions sont immédiates. Il y a essentiellement deux parties de la condition qui ressortent du lot : l'établissement de la pré-condition de l'appel à `main` ($P_{\mathtt{main}}$ dans la figure), et celui de la post-condition de $\overleftarrow{\mathcal{C}}(F, c)$ ($Q_{\overleftarrow{\mathcal{C}}}$). Pour la pré-condition de l'appel à `main`, la condition d'inclusion des (c_f) dans c suit de la structure du code, tandis que la structure voulue de l'état machine est garantie par les post-conditions des trois premières instructions exécutées. Enfin, le reste de

cette pré-condition coïncide avec la pré-condition de $\overleftarrow{\mathcal{C}}(F, c)$, et la post-condition à établir coïncide aussi avec la post-condition de l'appel à `main`. □

Comme nous l'avons précédemment remarqué, le contrat que nous devons montrer pour chaque fonction dépend de l'intégralité du code généré pour le programme. Cette dépendance globale semble être incompatible avec le raisonnement local (règle de compilation par règle de compilation) que nous avions proposé au début. Cependant, comme cette dépendance est causée par la récursivité mutuelle des fonctions de $\mathcal{L}_{\mathtt{s}}$, nous avons une manière simple de nous réduire à un raisonnement local. Nous allons employer la règle de récurrence de notre système pour expliciter la source des appels récursifs, et ainsi généraliser par rapport au code correspondant aux fonctions appelées récursivement.

Nous utilisons comme ordre de la progression de la récurrence un ordre exprimant la progression stricte de l'état source simulé ainsi que de l'état de la machine virtuelle de $\mathcal{L}_{\mathtt{t}}$. De cette manière, nous garantissons que toute situation limite de la récurrence correspond à la fois à un comportement infini des programmes sources et compilés, lesquels sont mis en correspondance par les pré-conditions des appels récursifs. Notons que nous sommes forcés d'exprimer le caractère strict pour les deux éléments, plutôt que d'utiliser un ordre produit. Si nous utilisons un simple ordre produit, nous ne pourrions pas garantir que les deux exécutions divergent simultanément.

Définition 6.22 (Ordre de progression de la récurrence)**.** Soit F le code du programme source, (X, t) et (Y, t') deux paires formées d'un état de la machine abstraite de $\mathcal{L}_{\mathtt{s}}$ et d'un compteur t. Nous disons que (X, t) est strictement inférieur à (Y, t') si

$$X \longrightarrow_F^+ Y \ \wedge \ t < t'.$$

Nous disons également qu'une telle paire est inférieure (largement) à une autre, noté $(X, t) \preccurlyeq_p (Y, t')$ si elle est égale ou strictement inférieure. Nous appelons l'ordre obtenu *l'ordre de progression de la récurrence.*

Définition 6.23 (Fonction de progression de la récurrence)**.** Nous définissons la *fonction de progression de la récurrence* comme la fonction qui aux paramètres auxiliaires précédemment nommés $(c_{glob}, \mathtt{pc}_R, (v_i)_{0\leq i<n}, \Sigma_0, k)$ d'un contrat de fonction $\mathcal{C}_{fun}(F, f, (c_i)_{i\in I})$ et à son état initial $((\mathtt{pc}, S, \Pi_0), t)$ associe la paire

$$(((), (\Sigma_0, \Pi_0), \&f((v_i)_{0\leq i<n}, k)), t)$$

.

Définition 6.24 (Contrat récursif)**.** Soit F un programme, c_{glob} un programme de $\mathcal{L}_{\mathtt{t}}$, et D un ensemble de paires formées d'un état de la machine abstraite de $\mathcal{L}_{\mathtt{s}}$ et d'un compteur t. Nous définissons le *schéma général d'un contrat de fonction récursif* $\mathcal{C}_{rec}(F, c_{glob}, D)$ comme le contrat suivant, obtenu en modifiant les contrats de fonction :

(A) les paramètres auxiliaires $(g, \mathtt{pc}_R, (v_i)_{0\leq i<n}, \Sigma_0, k)$, où g est un nom de fonction et les autres paramètres sont les mêmes que ceux d'un contrat de fonction.

(B) la valeur de retour auxiliaire v_R, qui est la même que celle d'un contrat de fonction.

(P) une pré-condition constituée de l'ensemble des états de $\mathbb{G}_{c_{glob},\forall}$ de la forme

$$((\mathtt{pc}_g, (S_0; \mathtt{RA}; (w_i)_{0\leq i<n}), \Pi_0), t)$$

tels que
(i) l'exécution du programme source F à partir de l'état

$$((), (\Sigma_0, \Pi_0), \&g((v_i)_{0\leq i<n}, k))$$

ne peut pas atteindre d'état bloquant.
(ii) $(((), (\Sigma_0, \Pi_0), \&g((v_i)_{0\leq i<n}, k)), t) \in D$
(iii) $\mathtt{pc}_g$ correspond à la position de l'étiquette $\mathtt{START}_g$ associée à g dans c_{glob}.
(iv) $\mathtt{pc}_R$ correspond à la position de RA dans c_{glob}.
(v) les valeurs cibles $(w_i)_{0\leq i<n}$ représentent correctement les valeurs sources $(v_i)_{0\leq i<n}$.

(Q) une post-condition donnée par la réunion de l'ensemble des canaux d'entrée/sortie Π_∞ tels que

$$((), (\Sigma_0, \Pi_0), \&g((v_i)_{0\leq i<n}, k)) \longrightarrow_F^\infty \Pi_\infty$$

et de l'ensemble des états de $\mathbb{G}_{c_{glob},\forall}$ de la forme

$$((\mathtt{pc}_R, (S_0; w), \Pi_1), t')$$

tels que S_0 correspond au fragment du même nom de l'état d'entrée, et tels que
(i) $((), (\Sigma_0, \Pi_0), \&g((v_i)_{0\leq i<n}, k)) \longrightarrow_F^* (v_R, (\Sigma_0, \Pi_1), k)$
(ii) la valeur cible w représente correctement la valeur source v_R.

Autrement dit, le contrat récursif est obtenu à partir du contrat de fonction en :
— transformant le paramètre auxiliaire c_{glob} en paramètre externe du contrat ;
— transformant le nom de la fonction en paramètre auxiliaire (interne) ;
— remplaçant la contrainte d'inclusion (dans (P), condition (ii)) d'une famille de code dans c_{glob} par une contrainte d'appartenance de la fonction de progression de la récurrence à un ensemble donné. Cet contrainte correspond typiquement à une contrainte de progression.

Définition 6.25 (Compilation localement correcte d'une fonction)**.** Soit F le code du programme source, f une fonction (pré)définie dans F, et c_f le code produit pour f. Nous disons que f est compilée en c_f de manière *localement correcte* si l'énoncé nommé $\mathcal{E}_{loc}(F, f, c_f)$ composé
— du contrat nommé $\mathcal{C}_{fun}(F, f, (c_g)_{g\in\{f\}})$
— du contexte $\Gamma_{c_{glob},\forall}$ (dépendant du paramètre auxiliaire c_{glob} dit « code global »), étendu par l'association du nom d'hypothèse `all-fun` au contrat

$$\mathcal{C}_{rec}(F, c_{glob}, \{x \mid x \succ_p (((), (\Sigma_0, \Pi_0), \&f((v_i)_{0\leq i<n}, k)), t)\})$$

dépendant des paramètres auxiliaires $(c_{glob}, \mathtt{pc}_R, (v_i)_{0\leq i<n}, \Sigma_0, k)$ du contrat nommé associé à f et à c_f, ainsi que du canal d'entrée/sortie Π_0 de l'état de départ de ce dernier contrat et du compteur t provenant du même état
est valide.

Nous montrons maintenant que le résultat de simulation en arrière pour notre schéma de compilation se déduit de la validité d'une famille d'énoncés nommés indexée par les noms de fonctions, où chaque énoncé dépend uniquement des codes sources et compilés pour la fonction en question.

Lemme 6.26. *Soit F le code du programme source, (c_f) les codes générés pour les fonctions (pré)définies par le programme (ou prédéfinie). Supposons que tout f soit compilée en c_f de manière localement correcte. Alors le résultat de simulation en arrière est valide.*

Démonstration. Il suffit de vérifier les hypothèses du lemme 6.21, et donc d'établir l'énoncé nommé $\mathcal{E}_{glob}(F, \texttt{main}, (c_f))$. Pour cela, nous allons appliquer le transformateur nommé de nature récursive suivant :

$$\left[\begin{array}{l} \mathcal{I}_{\texttt{asm},\forall} \\ c_{glob}, \texttt{pc}_R, (v_i)_{0\leq i<n}, \Sigma_0, k \end{array}\right] \Rightarrow \left[\ v_R\ \right]$$
$$\mathsf{let}\ g \leftarrow \texttt{main}$$
$$\mathsf{call\text{-}rec}\ \texttt{all-fun}(g, \texttt{pc}_R, (v_i)_{0\leq i<n}, \Sigma_0, k) :\ \mathcal{C}_{rec}(F, c_{glob}, A)$$
$$\mathsf{variant}\{V\}$$
$$\mathsf{progress}\{\preccurlyeq_p\}$$
$$\quad W_{all}$$
$$\mathsf{diverges}(H, g_G, g_{fn}, g_{RA}, g_v, g_\Sigma, g_k \mid clos)$$
$$\quad \mathsf{choose}\ clos \in H, v_R\ \mathsf{such\ that}\ \top$$
$$\quad \mathsf{let}\ clos \leftarrow clos, v_R \leftarrow ()$$
$$\mathsf{end\text{-}call\text{-}rec}$$

où

- A est l'ensemble de toutes les paires formées d'un état de $\mathcal{L}_{\texttt{s}}$ et d'un entier, ce qui nous permet d'énoncer le contrat final obtenu par récurrence sans contrainte de progression ;
- V désigne l'application de la fonction de progression ;
- W_{all} désigne un transformateur nommé regroupant les conversions des énoncés de correction locale de chaque fonction.

Pour obtenir le transformateur W_{all}, nous utilisons la construction du lemme 5.74 pour transformer le nom de fonction g en paramètre auxiliaire. Nous utilisons pour cela la famille (W_g) des conversions des énoncés $(\mathcal{E}_{loc}(F, g, c_g))$ pour les fonctions g (pré)définies dans F. Cela revient en fait à regrouper ces énoncés en un unique énoncé dont le contexte est le contexte commun de ces énoncés, et donc le contrat est $\mathcal{C}_{fun}(F, c_{glob}, (c_g))$, puis à convertir cet énoncé en transformateur.

Il ne reste alors qu'à vérifier les conditions générées par l'application du transformateur récursif ainsi obtenu à l'énoncé nommé voulu. Les conditions correspondant à l'application de W_{all} au contrat $\mathcal{C}_{rec}(F, c_{glob}, A)$ sont immédiates puisque les contrats sont quasiment identiques, aux pré-conditions d'inclusions des c_f dans c_{glob} près. Cependant, ces dernières suivent immédiatement de la pré-condition de l'énoncé de correction globale que l'on cherche à vérifier. De même, la post-condition de $\mathcal{C}_{fun}(F, c_{glob}, (c_f))$ suit directement de la post-condition de $\mathcal{C}_{rec}(F, c_{glob}, A)$. Le seul cas non trivial est celui des conditions associées aux situations limites de la récurrence. En fait, il se trouve que dans le cas d'une situation limite nous sommes en mesure d'établir directement la totalité des post-conditions des appels récursifs

constituant la situation limite, et donc en particulier à l'un d'entre eux. Nous nous sommes donc contentés d'utiliser le transformateur de choix pour obtenir l'un de ces éléments, qui existe bien car une situation limite correspond à une chaîne non vide.

Posons alors H_0 l'ensemble des éléments de H au moins aussi grands que l'élément *clos* choisi. Alors par définition de l'ordre de progression $g_G(H_0)$ doit être une suite infinie strictement croissante d'états du système de transition ordonné associé à la machine virtuelle, dont la borne supérieure est de la forme Π_∞. Remarquons également que d'après la nature de l'ordre de progression, nous pouvons mettre H_0 en correspondance avec une séquence infinie d'états de la machine abstraites

$$X_0 \longrightarrow_F^+ X_1 \longrightarrow_F^+ X_2 \longrightarrow_F^+ \dots X_i \longrightarrow_F^+ X_{i+1} \dots$$

dont les canaux d'entrée/sortie sont les mêmes que ceux de la séquence d'état de la machine virtuelle, à cause de la pré-condition des appels récursifs constituant la situation limite, et où X_0 correspond à l'élément *clos* choisi. Nous en déduisons que la borne supérieure des canaux des (X_i) est la même que celle des états de la machine virtuelle, c'est-à-dire, et en particulier :

$$X_0 \longrightarrow_F^\infty \Pi_\infty$$

ce qui est la post-condition voulue pour les états limites. □

6.3.3 Énoncé de compilation correcte pour les expressions

Nous avons maintenant réduit le résultat de simulation en arrière à un ensemble de propriétés locales sur le code compilé, décrites par des énoncés nommés. Il ne nous reste plus qu'à établir ces énoncés, ce que nous allons faire par un raisonnement local. Pour cela, nous allons associer un énoncé nommé à chaque expression compilée du code source, et établir lesdits énoncés par induction sur la syntaxe de l'expression. Ceux-ci correspondent au protocole d'exécution attendu pour une expression.

Remarquons que les expressions ne s'évaluent pas toutes jusqu'au bout, mais peuvent être interrompues par des retours ou encore ne pas terminer. Cela introduit une disjonction de l'espace d'arrivée. Nous avons plusieurs manières de représenter cette disjonction. Soit nous utilisons un espace de type somme disjointe pour les valeurs de retours auxiliaires, qui représente les différents cas de figure, soit nous utilisons des hypothèses de type continuation pour représenter ces post-conditions alternatives. Pour éviter d'appliquer systématiquement une partition des valeurs de retours, nous avons choisi d'utiliser la méthode à base de continuations. En particulier, comme ces post-conditions alternatives sont déplacées dans le contexte, cette disjonction de cas est absente de la post-condition du contrat nommé associé à une expression, qui ne représente que le cas d'exécution normal de l'expression.

Définition 6.27 (Contrat associé à une expression compilée)**.** Soit F un programme du langage $\mathcal{L}_\mathtt{s}$. Soient $e, (\mathtt{fs}, \sigma_v, \sigma_l)$ une expression e équipée des paramètres de sa compilation, et $c = [\![e]\!]_{\mathtt{fs},\sigma_v,\sigma_l}$. Le *contrat associé à l'expression e munie de* $(\mathtt{fs}, \sigma_v, \sigma_l)$ est le contrat nommé $\mathcal{C}_{expr}(F, e, (\mathtt{fs}, \sigma_v, \sigma_l), c)$ défini par :

(A) les paramètres auxiliaires (c_{glob}, Σ_0, k) constitués d'un programme c_{glob} du langage $\mathcal{L}_\mathtt{t}$ (le « code global »), d'une association de variables Σ_0, et d'un code de continuation k correspondant à des parties de l'état de la machine abstraite du langage source.

(B) les valeurs de retour auxiliaires (v_R, Σ_1) correspondant à une valeur source et à une association de variables Σ_1

(P) une pré-condition constituée de l'ensemble des états de $\mathbb{G}_{c_{glob},\forall}$ de la forme

$$((\mathtt{pc}, S_0, \Pi_0), t)$$

tels que

(i) l'exécution du programme source à partir de l'état $(e, (\Sigma_0, \Pi_0), k)$ n'atteint pas d'état bloquant.

(ii) c est une sous-séquence de c_{glob}, commençant à la position $\mathtt{pc}$.

(iii) pour tout x appartenant au domaine de σ_v, la $(\mathtt{fs} - \sigma_v(x))$-ième valeur depuis le sommet de la pile S_0 (en partant de 0 pour le sommet) contient une valeur représentant correctement la dernière valeur de la séquence $\Sigma_0(x)$.

(iv) S_0 contient au moins $\mathtt{fs}$ valeurs.

(v) pour tout L appartenant au domaine de σ_l, l'étiquette $\mathtt{EXIT}_L$ associée à L existe dans c_{glob}.

(Q) une post-condition paramétrée par les paramètres auxiliaires, l'état d'entrée de la forme ci-dessus, et les retours auxiliaires, donnée par l'ensemble des états de $\mathbb{G}_{c_{glob},\forall}$ de la forme

$$((\mathtt{pc}', (S_1; w), \Pi_1), t')$$

tels que

(i) $(e, (\Sigma_0, \Pi_0), k) \longrightarrow_F^* (v_R, (\Sigma_1, \Pi_1), k)$

(ii) la différence $\mathtt{pc}' - \mathtt{pc}$ correspond au nombre d'instructions dans c.

(iii) S_1 fait la même taille que la pile initiale S_0, et leurs valeurs coïncident à l'exception potentielle, pour tout x appartenant au domaine de σ_v, de la $(\mathtt{fs} - \sigma_v(x))$-ième valeur (partant de 0) depuis le sommet de la pile.

(iv) pour tout x appartenant au domaine de σ_v, la $(\mathtt{fs} - \sigma_v(x))$-ième valeur depuis le sommet de la pile S_1 représente correctement la dernière valeur de la séquence $\Sigma_1(x)$.

(v) la valeur cible w représente correctement la valeur source v_R.

(vi) Σ_1 est une mise à jour de Σ_0 pour les variables appartenant au domaine de σ_v. Autrement dit, les valeurs pour les autres variables ainsi que les valeurs précédentes des variables du domaine de σ_v sont les mêmes dans Σ_1 et Σ_0.

Définition 6.28 (Hypothèses additionnelles associées à une expression compilée)**.** Soit F un programme du langage $\mathcal{L}_{\mathtt{s}}$. Les hypothèses additionnelles associées à l'expression e munie de $(\mathtt{fs}, \sigma_v, \sigma_l)$ sont les hypothèses suivantes, dépendantes des mêmes paramètres auxiliaires (c_{glob}, Σ_0, k) que le contrat nommé associé à e ainsi que de son état initial $((\mathtt{pc}, S_0, \Pi_0), t)$:

(`all-fun`) L'hypothèse nommée `all-fun`, associée au contrat

$$\mathcal{C}_{rec}(F, c_{glob}, \{(X, t) \mid (e, (\Sigma_0, \Pi_0), k) \longrightarrow_F^* X\}).$$

Remarquons que vis-à-vis des énoncés de correction locale pour les fonctions, nous avons éliminé la contrainte de progression du compteur. En effet, celle-ci est déductible de l'ordre de progression du jeu. Nous justifions cette conséquence logique au niveau de la preuve de la compilation correcte des déclarations de fonctions.

(`divg`) L'hypothèse nommée `divg` associée au contrat $\mathcal{C}_{div}$ de type continuation (dont la post-condition est vide), correspondant aux comportements divergents, sans paramètre ni valeur de retour auxiliaire et dont la pré-condition est l'ensemble des états Π_∞ tels que

$$(e, (\Sigma_0, \Pi_0), k) \longrightarrow_F^\infty \Pi_\infty$$

(`ret`) L'hypothèse nommée `ret`, associée au contrat $\mathcal{C}_{ret}$ de type continuation (dont la post-condition est vide), correspondant au cas d'un retour au sein de l'expression compilée. Les paramètres auxiliaires de ce contrat sont l'étiquette source L correspondant au retour, la valeur renvoyée v_R et l'association de variable finale Σ_1. La pré-condition de $\mathcal{C}_{ret}$, proche de la post-condition du contrat nommé associé à e, est l'ensemble des états $((\mathtt{pc}_L, (S_1; w), \Pi_1), t')$ tels que

(i) $(e, (\Sigma_0, \Pi_0), k) \longrightarrow_F^* (v_R, (\Sigma_1, \Pi_1), \mathtt{return}[L]\ k)$

(ii) L est associé à $(\mathtt{fs}_L, \mathtt{EXIT}_L)$ par σ_l, et $\mathtt{pc}_L$ désigne l'emplacement de $\mathtt{EXIT}_L$ dans c_{glob}.

(iii) S_0 contient $\mathtt{fs} - \mathtt{fs}_L$ valeurs supplémentaires par rapport à S_1, et l'extension de S_1 par les $\mathtt{fs} - \mathtt{fs}_L$ valeurs supplémentaires de S_0 coïncide avec S_0 à l'exception potentielle, pour tout x appartenant au domaine de σ_v, de la $(\mathtt{fs} - \sigma_v(x))$-ième valeur (partant de 0) depuis le sommet de la pile.

(iv) pour tout x appartenant au domaine de σ_v, la $(\mathtt{fs} - \sigma_v(x))$-ième valeur depuis le sommet de la pile S_1 représente correctement la dernière valeur de la séquence $\Sigma_1(x)$.

(v) la valeur cible w représente correctement la valeur v_R.

(vi) Σ_1 est une mise à jour de Σ_0 pour les variables appartenant au domaine de σ_v.

Définition 6.29 (Compilation correcte d'une expression)**.** Soit F un programme du langage $\mathcal{L}_\mathtt{s}$. Nous disons qu'une expression e munie de $(\mathtt{fs}, \sigma_v, \sigma_l)$ est *correctement compilée* en $c = [\![e]\!]_{\mathtt{fs},\sigma_v,\sigma_l}$ si l'énoncé nommé $\mathcal{E}_{expr}(F, e, (\mathtt{fs}, \sigma_v, \sigma_l), c)$ composé

— du contrat nommé associé à l'expression e munie de $(\mathtt{fs}, \sigma_v, \sigma_l)$

— du contexte étiqueté $\Gamma_{c_{glob},\forall}$ (dépendant du paramètre auxiliaire c_{glob} dit « code global ») étendu par les hypothèses additionnelles associées à e munie de $(\mathtt{fs}, \sigma_v, \sigma_l)$, avec les associations de noms présentées pour chaque hypothèse

est valide.

6.3.4 Compilation correcte des expressions

Nous pouvons alors énoncer le théorème de correction de compilation des expressions.

Lemme 6.30. *Soit F un programme du langage $\mathcal{L}_\mathtt{s}$. Alors toute expression e munie d'un triplet $(\mathtt{fs}, \sigma_v, \sigma_l)$ tel que :*

(i) $\mathtt{fs} \in \mathbb{N}^*$

(ii) pour tout x dans le domaine de σ_v, $0 < \sigma_v(x) \leq \mathtt{fs}$

(iii) pour tout L associé à $(\mathtt{fs}_L, \mathtt{EXIT}_L)$ par σ_l, $0 \leq \mathtt{fs}_L \leq \mathtt{fs}$

est correctement compilée.

Démonstration. Nous procédons par induction sur la syntaxe de l'expression. Chaque cas de l'induction correspond alors à une règle de compilation. Le principe de la preuve d'une règle est toujours le même. Premièrement, nous convertissons les énoncés nommés correspondant aux expressions bien compilées par hypothèse d'induction en transformateurs nommés. Nous notons le transformateur nommé obtenu pour une sous-expression e et pour les paramètres de compilation associés à e dans le schéma de compilation par W_e.

Ensuite, nous combinons ces transformateurs en employant les règles correspondant au sens voulu du programme compilé. Enfin, nous appliquons le transformateur nommé résultant à l'énoncé nommé $\mathcal{E}_{expr}(F, e, (\texttt{fs}, \sigma_v, \sigma_l), c)$ correspondant à l'expression courante. Nous déroulons mécaniquement les définitions des transformateurs pour obtenir une condition de vérification. Bien que les conditions de vérification obtenues soient des conjonctions de nombreuses conditions, la très large majorité de ces dernières sont immédiates à vérifier.

Cas des variables

$$[\![x]\!]_{\texttt{fs},\sigma_v,\sigma_l} = \texttt{load } (\texttt{fs} - \sigma_v(x))$$

transformateur nommé associé :

$$\begin{array}{l} \left[\begin{array}{l} \mathcal{I}_{\texttt{asm},\forall}, \texttt{all-fun}, \texttt{divg}, \texttt{ret} \\ c_{glob}, \Sigma_0, k \end{array}\right] \Rightarrow \left[\begin{array}{l} v_R, \Sigma_1 \end{array}\right] \\ \texttt{load}[n \leftarrow (\texttt{fs} - \sigma_v(x))] \\ \mathsf{let}\ v_R \leftarrow (\text{dernier élément de } \Sigma_0(x)), \Sigma_1 \leftarrow \Sigma_0 \end{array}$$

La condition de vérification obtenue se réduit principalement à l'établissement de la post-condition finale. La pré-condition du contrat de l'expression garantit que la valeur stockée à la position accédée représente bien la valeur de variable à laquelle on cherche à accéder, et la contrainte sur la valeur de $\sigma_v(x)$ assure que cette position existe. Nous pouvons donc établir que la valeur obtenue au sommet de la pile est bien associée à v_R, comme attendu par la post-condition du contrat d'une expression.

Comme la pré-condition assure qu'un état bloquant ne peut pas être atteint, la valeur obtenue existe bien et ne peut pas être (). La chaîne de transitions à établir est donc directement obtenue par définition de la règle d'évaluation d'une variable du langage source. Enfin, les autres parties de la post-condition finale suivent directement soit de la pré-condition du contrat, comme la structure de la pile vis-à-vis de Σ_1, soit de la définition de Σ_1, soit de la spécification de `load`, comme le fait que la tête de lecture avance exactement d'un cran.

Cas de l'expression vide

$$[\![()]\!]_{\texttt{fs},\sigma_v,\sigma_l} = \texttt{load } 0$$

transformateur nommé associé :

$$\begin{array}{l} \left[\begin{array}{l} \mathcal{I}_{\texttt{asm},\forall}, \texttt{all-fun}, \texttt{divg}, \texttt{ret} \\ c_{glob}, \Sigma_0, k \end{array}\right] \Rightarrow \left[\begin{array}{l} v_R, \Sigma_1 \end{array}\right] \\ \texttt{load}[n \leftarrow 0] \\ \mathsf{let}\ v_R \leftarrow (), \Sigma_1 \leftarrow \Sigma_0 \end{array}$$

La contrainte $\texttt{fs} > 0$ assure que la pile est non vide, ce qui nous permet d'utiliser `load` 0. Le reste de la condition de vérification se traite exactement de la même manière que pour le cas des variables.

Cas de la prise d'adresse d'une fonction

$$[\![\&f]\!]_{\texttt{fs},\sigma_v,\sigma_l} = \texttt{aload START}_f$$

transformateur nommé associé :

$$\begin{array}{l} \left[\begin{array}{l} \mathcal{I}_{\texttt{asm},\forall}, \textit{all-fun}, \textit{divg}, \textit{ret} \\ c_{glob}, \Sigma_0, k \end{array}\right] \Rightarrow \left[\begin{array}{l} v_R, \Sigma_1 \end{array}\right] \\ \textit{aload}[\texttt{L} \leftarrow \texttt{START}_f] \\ \text{let } v_R \leftarrow \&f, \Sigma_1 \leftarrow \Sigma_0 \end{array}$$

La preuve est quasi identique au cas de l'expression vide, la corrélation entre l'étiquette $\texttt{START}_f$ et la valeur $\&f$ étant immédiate par définition.

Cas de la mise en séquence

$$[\![e_1; e_2]\!]_{\texttt{fs},\sigma_v,\sigma_l} = \left\{\begin{array}{l} [\![e_1]\!]_{\texttt{fs},\sigma_v,\sigma_l} \\ \texttt{pop } 1 \\ [\![e_2]\!]_{\texttt{fs},\sigma_v,\sigma_l} \end{array}\right.$$

transformateur nommé associé :

$$\begin{array}{l} \left[\begin{array}{l} \mathcal{I}_{\texttt{asm},\forall}, \textit{all-fun}, \textit{divg}, \textit{ret} \\ c_{glob}, \Sigma_0, k \end{array}\right] \Rightarrow \left[\begin{array}{l} v_R, \Sigma_1 \end{array}\right] \\ W_{e_1}[k \leftarrow (k; e_2)] \\ \textit{pop}[n \leftarrow 1] \\ W_{e_2}[\Sigma_0 \leftarrow \Sigma_1] \end{array}$$

Ce cas est le premier où nous utilisons des transformateurs obtenus par conversion d'énoncés nommés préalablement établis, en l'occurence W_{e_1} et W_{e_2}. Nous devons donc traiter de nouveaux types de conditions par rapport aux cas précédents :

— L'inclusion des hypothèses additionnelles associées à e_1 ou e_2 dans les hypothèses additionnelles associées à e. Comme les post-conditions coïncident, ainsi que la partie purement structurelle des pré-conditions, il suffit de vérifier l'inclusion de la partie des pré-conditions correspondant à l'existence d'une séquence de transitions dans la machine abstraite source depuis l'état d'origine. Il s'agit des pré-conditions données dans la définition 6.28, plus précisément la condition (i) dans le cas de *ret* et la condition (P).(ii) originaire de la définition 6.24 dans le cas de *all-fun*. En déroulant les définitions, cela revient à dire que si la chaîne de transitions correspondant à la pré-condition d'une hypothèse associée à e_1 (resp. e_2) existe, alors la chaîne de transitions correspondant à la pré-condition de l'hypothèse associée à e existe également. Dans le cas de e_1, il suffit pour cela de prolonger le début de la chaîne par la transition

$$(e_1; e_2, (\Sigma_0, \Pi_0), k) \longrightarrow_F (e_1, (\Sigma_0, \Pi_0), k; e_2)$$

correspondant à la première étape d'une séquence. Pour l'inclusion des deux hypothèses de nom *ret* correspondant à la création d'un retour, il faut également prolonger la chaîne de transitions de l'autre côté de manière à faire passer le retour au-dessus de la séquence :

$$(v, (\Sigma_1, \Pi_1), \texttt{return}[L]\ (k; e_2)) \longrightarrow_F (v, (\Sigma_1, \Pi_1), \texttt{return}[L]\ k)$$

Dans le cas de e_2, il faut utiliser la chaîne de transitions

$$(e_1, (\Sigma_0, \Pi_0), k; e_2) \longrightarrow_F^* (v_R, (\Sigma_1, \Pi_1), k; e_2)$$

obtenue via la post-condition du contrat de e_1 pour pouvoir prolonger le début de la chaîne de transitions jusqu'à e.

— Les contraintes structurelles découlent directement de celles établies par les codes compilés pour e_1 et e_2.

— Pour établir la chaîne de transitions finale correspondant à la post-condition, il faut combiner les chaînes de transitions générées respectivement pour e_1 et e_2 par transitivité.

Ce raisonnement d'inclusion des hypothèses, ainsi que celui de transitivité pour générer la chaîne de transitions finale, reviennent pratiquement à l'identique pour chaque construction.

Cas de l'affectation

$$[\![x \leftarrow e]\!]_{\texttt{fs},\sigma_v,\sigma_l} = \begin{cases} [\![e]\!]_{\texttt{fs},\sigma_v,\sigma_l} \\ \texttt{load}\ 0 \\ \texttt{store}\ (\texttt{fs} + 1 - \sigma_v(x)) \end{cases}$$

transformateur nommé associé :

$$\begin{array}{l} \left[\begin{array}{l} \mathcal{I}_{\texttt{asm},\forall}, \textit{all-fun}, \textit{divg}, \textit{ret} \\ c_{glob}, \Sigma_0, k \end{array}\right] \Rightarrow \left[\begin{array}{l} v_R, \Sigma_1 \end{array}\right] \\ W_e[k \leftarrow (x \leftarrow k)] \\ \textit{load}[n \leftarrow 0] \\ \textit{store}[n \leftarrow (\texttt{fs} + 1 - \sigma_v(x))] \\ \mathsf{let}\ \Sigma_1 \leftarrow \Sigma_0[x \leftarrow v_R], v_R \leftarrow () \end{array}$$

La preuve de la condition de vérification suit l'esprit de ce que nous avons fait pour la séquence, notamment pour l'inclusion des hypothèses additionnelles. Comme la valeur source () peut être représentée par n'importe quoi, la valeur dupliquée par l'instruction `load 0` représente bien la valeur source obtenue. De même, la valeur finalement associée à Σ_1 se retrouve bien dans l'emplacement voulu de la pile. Enfin, comme pour le cas des variables nous pouvons garantir que la valeur source stockée n'est pas () car aucun état bloquant n'est accessible, et pouvons donc construire la chaîne de transitions demandée par la post-condition.

Cas de l'expression conditionnelle

$$[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]_{\texttt{fs},\sigma_v,\sigma_l} = \left\{ \begin{array}{ll} & [\![e_1]\!]_{\texttt{fs},\sigma_v,\sigma_l} \\ & \texttt{jnz THEN} \\ & [\![e_3]\!]_{\texttt{fs},\sigma_v,\sigma_l} \\ & \texttt{aload EXIT} \\ & \texttt{jmp} \\ \texttt{THEN:} & \texttt{nop} \\ & [\![e_2]\!]_{\texttt{fs},\sigma_v,\sigma_l} \\ \texttt{EXIT:} & \texttt{nop} \end{array} \right.$$

transformateur nommé associé :

$$\left[\begin{array}{l} \mathcal{I}_{\texttt{asm},\forall}, \textit{all-fun}, \textit{divg}, \textit{ret} \\ c_{glob}, \Sigma_0, k \end{array} \right] \Rightarrow \left[\begin{array}{l} v_R, \Sigma_1 \end{array} \right]$$

$W_{e_1}[k \leftarrow \texttt{if } k \texttt{ then } e_2 \texttt{ else } e_3]$
jnz[L ← THEN]
switch
case $v_R = \top$:
  $W_{e_2}[\Sigma_0 \leftarrow \Sigma_1]$
case $v_R = \bot$:
  $W_{e_3}[\Sigma_0 \leftarrow \Sigma_1]$
  aload[L ← EXIT]
  jmp
  let $v_R \leftarrow v_R, \Sigma_1 \leftarrow \Sigma_1$
end-switch
nop
let $v_R \leftarrow v_R, \Sigma_1 \leftarrow \Sigma_1$

Comme l'expression conditionnelle introduit un branchement dans l'exécution du code, nous simulons ce branchement au niveau de la combinaison de transformateurs choisie. Cela génère au final deux parties dans la condition de vérification, correspondant aux deux chemins d'exécution. En plus des conditions d'une forme déjà traitée, nous avons une contrainte d'exhaustivité du branchement, qui indique que la valeur au sommet de la pile doit représenter un booléen après l'exécution du code correspondant à e_1. Mais si ce n'était pas le cas, nous pourrions construire une chaîne de transitions de la machine abstraite du langage source qui atteint un état bloquant depuis l'état associé à e, ce qui est supposé impossible dans la pré-condition associée à e.

Cas d'un retour

$$[\![\texttt{return}[L_0]\ e]\!]_{\texttt{fs},\sigma_v,\sigma_l} = \left\{ \begin{array}{l} [\![e]\!]_{\texttt{fs},\sigma_v,\sigma_l} \\ \texttt{cut } (\texttt{fs} - \texttt{fs}_{L_0}) \\ \texttt{aload EXIT}_{L_0} \\ \texttt{jmp} \end{array} \right.$$

transformateur nommé associé :

$$\left[\begin{array}{l} \mathcal{I}_{\mathtt{asm},\forall}, \mathit{all\text{-}fun}, \mathit{divg}, \mathit{ret} \\ c_{glob}, \Sigma_0, k \end{array}\right] \Rightarrow \left[\ v_R, \Sigma_1\ \right]$$
$W_e[k \leftarrow \mathtt{return}[L_0]\ k]$
$\mathit{cut}[n \leftarrow (\mathtt{fs} - \mathtt{fs}_{L_0})]$
$\mathit{aload}[\mathtt{L} \leftarrow \mathtt{EXIT}_{L_0}]$
jmp
$\mathit{ret}[L \leftarrow L_0]$
choose v_R, Σ_1 such that $\bot$

Comme `cut` est une pseudo-instruction, il n'y a pas d'hypothèse correspondante. Nous avons deux manières d'obtenir le transformateur correspondant. La première possibilité est d'établir une fois pour toutes un contrat nommé correspondant à la spécification de `cut`, par le biais du transformateur nommé suivant :

$$\left[\begin{array}{l} \mathcal{I}_{\mathtt{asm},\forall} \\ n \in \mathbb{N} \end{array}\right] \Rightarrow []$$
switch
case $n = 0$:
    skip
default :
    $\mathit{store}[n \leftarrow n - 1]$
    $\mathit{pop}[n \leftarrow n - 1]$
end-switch

La seconde possibilité consiste à utiliser directement ce transformateur, ce qui possède l'avantage de la simplicité, mais la disjonction de cas duplique les chemins d'exécution, et augmente donc la taille de la condition de vérification.

La preuve de la condition de vérification est très similaire aux preuves précédentes. Le fait que nous utilisions l'hypothèse de continuation `ret` au lieu d'atteindre la post-condition voulue pour l'expression ne change pas grand chose, puisque la pré-condition de `ret` est quasiment identique à ladite post-condition. Les seules différences sont le code de continuation, qui devient $\mathtt{return}[L]\ k$ plutôt que k, et que la pile doit être en partie coupée plutôt que préservée.

Cas d'un bloc

$$[\![[L_0]\{\ (x_i)_{0\le i<m} : e\ \}]\!]_{\mathtt{fs},\sigma_v,\sigma_l} = \left\{\begin{array}{ll} (m \text{ fois}) & \mathtt{load\ 0} \\ & [\![e]\!]_{\mathtt{fs}+m,\sigma_v[(x_i \leftarrow \mathtt{fs}+i)_{0\le i<m}],\sigma_l[L_0 \leftarrow (\mathtt{fs},\mathtt{EXIT})]} \\ & \mathtt{cut}\ m \\ \mathtt{EXIT:} & \mathtt{nop} \end{array}\right.$$

transformateur nommé associé :

$$\left[\begin{array}{l}\mathcal{I}_{\mathtt{asm},\forall}, \mathtt{all\text{-}fun}, \mathtt{divg}, \mathtt{ret}\\ c_{glob}, \Sigma_0, k\end{array}\right] \Rightarrow \left[\ v_R, \Sigma_1\ \right]$$

$W_{m\times\mathtt{load}\ 0}$
continuation $K(v_R, \Sigma_1)$
  intro $\mathtt{ret}(L, v_R, \Sigma_1)$: $\mathcal{C}_{ret}$
  by
    switch
    case $L = L_0$:
      K
    default :
      $\mathtt{ret}$
    end-switch
  in
    $W_e[\Sigma_0 \leftarrow \Sigma_0[+(x_i \leftarrow ())_{0\le i<m}], k \leftarrow [L_0]\{\ (x_i)_{0\le i<m} : k\ \}]$
    $\mathtt{cut}[n \leftarrow m]$
    let $v_R \leftarrow v_R, \Sigma_1 \leftarrow \Sigma_1$
  end-intro
end-continuation
$\mathtt{nop}$
let $v_R \leftarrow v_R, \Sigma_1 \leftarrow \Sigma_1$

où $\mathcal{C}_{ret}$ désigne le contrat associé à $\mathtt{ret}$ pour l'expression e associée à $\mathtt{fs}{+}m$, $\sigma_v[(x_i \leftarrow \mathtt{fs}+i)_{0\le i<m}]$, et $\sigma_l[L_0 \leftarrow (\mathtt{fs}, \mathtt{EXIT})]$, et $W_{m\times\mathtt{load}\ 0}$ est défini par la conversion d'un contrat pour la répétition à m reprises de l'instruction `load 0`. Ce dernier contrat indique que le sommet de la pile est dupliqué m fois, et est immédiatement établi par induction sur m. Nous pourrions également établir ce contrat directement par le biais d'un transformateur d'itération, ou définir directement $W_{m\times\mathtt{load}\ 0}$ par un tel transformateur.

Le cas d'un bloc introduit de nouveaux types de conditions à cause du changement d'environnement de compilation. Par exemple, nous devons vérifier la contrainte structurelle que le bloc ne change pas de régions de la pile autres que celles désignées par la valeur initiale de σ_v, malgré les modifications additionnelles permises par l'exécution du code interne du bloc. Nous n'avions pas ce problème précédemment car les régions permises pour la modification coïncidaient pour les piles associées à e et à ses sous-expressions. Nous obtenons le même résultat pour le cas du bloc en remarquant que les nouvelles régions qui peuvent être modifiées disparaissent de la pile à la fin du bloc.

Enfin, nous devons également nous occuper des effets de la modification de σ_l. Cela change suffisamment la nature de l'hypothèse additionnelle de retour pour que nous devions la redéfinir dans le cas où l'étiquette source est L_0. Nous devons donc vérifier qu'il y a bien inclusion de l'hypothèse de retour, ce qui demande un raisonnement spécifique dans le cas où l'étiquette source est L_0. La contrainte correspondante se réduit facilement à l'extension de la chaîne de transitions qui va de e vers un retour associé à L_0 en une chaîne qui va du bloc vers une valeur. Cette extension est immédiate par définition des transitions de la machine abstraite associée au langage source.

Cas d'un appel de fonction

$$[\![e_0((e_i)_{0<i\leq m})]\!]_{\texttt{fs},\sigma_v,\sigma_l} = \begin{cases} & [\![e_0]\!]_{\texttt{fs},\sigma_v,\sigma_l} \\ & \vdots \\ & [\![e_i]\!]_{\texttt{fs}+i,\sigma_v,\sigma_l} \\ & \vdots \\ & [\![e_m]\!]_{\texttt{fs}+m,\sigma_v,\sigma_l} \\ & \texttt{load } m \\ & \texttt{aload RA} \\ & \texttt{store } (m+1) \\ & \texttt{jmp} \\ \texttt{RA:} & \texttt{nop} \end{cases}$$

transformateur nommé associé :

$$\left[\begin{array}{l} \mathcal{I}_{\texttt{asm},\forall}, \textit{all-fun}, \textit{divg}, \textit{ret} \\ c_{glob}, \Sigma_0, k \end{array}\right] \Rightarrow \left[\begin{array}{l} v_R, \Sigma_1 \end{array}\right]$$

$W_{(e_i)_{0\leq i\leq m}}$
load$[n \leftarrow m]$
aload$[\texttt{L} \leftarrow \texttt{RA}]$
store$[n \leftarrow m+1]$
jmp
choose g such that $v_0 = \&g$
all-fun$[\texttt{pc}_R \leftarrow \texttt{pc}_{\texttt{RA}}]$
switch
case now est un état limite :
    divg
default :
    skip
end-switch
nop
let $v_R \leftarrow v_R, \Sigma_1 \leftarrow \Sigma_1$

Pour tout $l \in [0; m]$, le transformateur nommé $W_{(e_i)_{0\leq i\leq l}}$ est obtenu par conversion de l'énoncé nommé correspondant au contrat de l'évaluation des $l+1$ premières expressions de l'appel. Ce contrat est le même que celui associé à l'expression d'appel, aux différences suivantes :

— La valeur de retour v_R est remplacée par la valeur de retour $(v_i)_{0\leq i\leq l}$, correspondant à une séquence de valeurs source
— La pile est étendue par $l+1$ valeurs au lieu d'une seule, correspondant respectivement aux valeurs des $(v_i)_{0\leq i\leq l}$
— La tête de lecture finale se trouve à la fin du code correspondant aux $(e_i)_{0\leq i\leq l}$ plutôt qu'à la fin du code de l'expression d'appel.
— L'état final de la chaîne de transitions générée est remplacé par

$$(e_{l+1}, (\Sigma_1, \Pi_1), v_0((v_j)_{0<j\leq l}, k, (e_j)_{l+1<j\leq m+1})$$

où l'on pose $e_{m+1} = ()$ pour simplifier les notations, et v_0 doit être une adresse de fonction pour que cet état source soit bien formé.

L'établissement de ce contrat est immédiat par induction sur l, le cas d'hérédité correspondant à une simple liaison séquentielle. Pour le cas de base, où nous devons montrer que v_0 est une adresse de fonction, nous exploitons la partie de la pré-condition qui assure la réduction non bloquante de l'appel de fonction. Comme pour le cas du bloc, nous pouvons alternativement employer une structure itérative pour effectuer l'induction directement par le biais des transformateurs.

Nous démontrons la condition de vérification restante de manière directe. La post-condition du contrat obtenu pour l'évaluation des $m + 1$ expressions et des instructions qui suivent garantit la mise en forme de la pile sous la forme demandée par le protocole d'appel. L'appel lui-même est réalisé par l'utilisation de l'hypothèse récursive `all-fun`, dont la pré-condition correspond au protocole d'appel plus la condition de progression récursive. Cette dernière correspond précisément à la chaîne de transitions produite par la post-condition du contrat correspondant à l'évaluation des $m + 1$ premières expressions.

Enfin, comme la post-condition de l'appel est similaire à celle d'une sous-expression, et de plus garantit que l'emplacement de la tête de lecture est au niveau de l'étiquette `RA`, nous pouvons établir la post-condition finale de l'expression comme précédemment. Notons que la disjonction de cas juste avant la fin du transformateur permet précisément d'éliminer le cas des états limites par le biais de l'hypothèse de continuation correspondant à la divergence. Nous devons effectuer cette disjonction à cause de la différence de gestion des états limites dans les contrats récursifs et dans les énoncés associés aux expressions : dans un cas, ces états sont traités par une disjonction dans la post-condition, tandis que dans l'autre, nous utilisons une hypothèse de continuation pour représenter cette disjonction.

Cas d'une boucle Pour le cas de la boucle, nous utilisons sans surprise un transformateur d'itération, combiné au transformateur de continuation pour sortir de la boucle. Le transformateur est donné dans la figure 6.14. La structure de ce transformateur est proche de celle de la récurrence faite pour les fonctions, notamment à cause de l'ordre de progression choisi.

L'invariant I correspond à la propriété suivante :

(i) L'existence d'une chaîne de transitions

$$(\texttt{while}\ e_1\ \texttt{do}\ e_2, (\Sigma_0, \Pi_0), k) \longrightarrow_F^* (\texttt{while}\ e_1\ \texttt{do}\ e_2, (\Sigma_1, \Pi_1), k)$$

où Π_0 et Π_1 correspondent respectivement aux canaux d'entrée/sortie pour l'état initial et l'état courant de la boucle.

(ii) La présence de la tête de lecture au niveau de l'étiquette `TEST`.

(iii) Les mêmes conditions liant $(S_0, \Sigma_0, S_1, \Sigma_1, \texttt{fs}, \sigma_v)$ que dans la post-condition d'une expression compilée, où S_0 et S_1 représentent respectivement la pile initiale et la pile courante. Autrement dit, l'absence de mises à jours de la pile autres que pour les variables présentes dans σ_v, la représentation correcte des associations de Σ_1 dans S_1, et le fait que Σ_1 soit une mise à jour de Σ_0.

La majorité des conditions obtenues sont alors similaires à celles correspondant à la compilation d'un test, et se prouvent de la même façon. La seule nouvelle contrainte correspond à la gestion des situations limites. Mais comme l'ordre de progression est le même, les situations limites suivent une structure similaire à celle de la récurrence. Nous pouvons donc obtenir une chaîne de transitions infinie depuis

$$[\![\texttt{while } e_1 \texttt{ do } e_2]\!]_{\texttt{fs},\sigma_v,\sigma_l} = \begin{cases} & \texttt{aload TEST} \\ & \texttt{jmp} \\ \texttt{BODY:} & \texttt{nop} \\ & [\![e_2]\!]_{\texttt{fs},\sigma_v,\sigma_l} \\ & \texttt{pop } 1 \\ \texttt{TEST:} & \texttt{nop} \\ & [\![e_1]\!]_{\texttt{fs},\sigma_v,\sigma_l} \\ & \texttt{jnz BODY} \end{cases}$$

transformateur nommé associé :

$$\left[\begin{array}{l} \mathcal{I}_{\texttt{asm},\forall}, \textit{all-fun}, \textit{divg}, \textit{ret} \\ c_{glob}, \Sigma_0, k \end{array}\right] \Rightarrow \left[\ v_R, \Sigma_1\ \right]$$

aload[L ← TEST]
jmp
continuation *K*(Σ_1)
  let $\Sigma_1 \leftarrow \Sigma_0$
  iter(Σ_1)
  invariant{I}
  variant{((while e_1 do e_2, (Σ_1, Π), k), t) si now = ((pc, S, Π), t)}
  progress{$\preccurlyeq_p$}
    nop
    $W_{e_1}[\Sigma_0 \leftarrow \Sigma_1, k \leftarrow \texttt{if } k \texttt{ then } e_2; \texttt{while } e_1 \texttt{ do } e_2 \texttt{ else } ()]$
    jnz[L ← BODY]
    switch
    case $v_R = \top$:
      nop
      $W_{e_2}[\Sigma_0 \leftarrow \Sigma_1, k \leftarrow k; \texttt{while } e_1 \texttt{ do } e_2]$
      pop[$n \leftarrow 1$]
      let $\Sigma_1 \leftarrow \Sigma_1$
    case $v_R = \bot$:
      K
      choose Σ_1 such that $\bot$
    end-switch
  diverges[H, f_G, f_{Σ_1}]
    divg
    choose Σ_1 such that $\bot$
  end-iter
end-continuation
let $v_R \leftarrow (), \Sigma_1 \leftarrow \Sigma_1$

FIGURE 6.14 – Code compilé et transformateur pour une boucle

n'importe quel état source provenant de la situation limite vers le canal d'entrée/-sortie courant. Mais comme l'invariant assure que cet état source est atteignable par une chaîne de transitions depuis l'état correspondant au début de la boucle, nous pouvons prolonger le début de la chaîne de transitions infinie jusqu'à l'état source, ce qui établit la pré-condition de l'hypothèse de divergence `divg`.

□

6.3.5 Des expressions aux fonctions

Pour finir la preuve, il nous suffit maintenant d'établir la correction locale de la compilation des fonctions.

Lemme 6.31. *Les fonctions sont compilées de manière localement correcte. En particulier, le résultat de simulation en arrière énoncé par le théorème 6.9 est vérifié.*

Démonstration. Pour les fonctions prédéfinies, cela se réduit à un simple problème de preuve de programme, pour lequel nous pouvons exploiter nos transformateurs. Dans le cas où la machine virtuelle admet une instruction proche de l'opérateur prédéfini, cela revient alors à une simple mise en séquence, comme pour le cas des fonctions `print` et `scan`.

Pour les fonctions définies par le programme, la correction locale de la compilation se déduit de la compilation correcte des expressions. Nous utilisons une combinaison de transformateurs nommés pour passer de l'une à l'autre. Nous rappelons la compilation d'une déclaration de fonction dans la figure 6.15, et donnons en dessous le transformateur nommé correspondant au passage de la correction de la compilation de l'expression sous-jacente à la correction locale de la compilation de la fonction.

Ce transformateur est nettement plus imposant que le code compilé correspondant car nous devons établir les hypothèses additionnelles de l'expression sous-jacente. La structure est proche de celle d'un bloc pour le retour, mais comme il n'y a qu'une seule étiquette possible nous n'avons pas besoin de faire une disjonction de cas sur la continuation créée. De même, nous pouvons créer l'hypothèse associée aux comportements ne terminant pas en prenant une continuation plus permissive sur la fin du transformateur ainsi créé. Enfin, nous devons également adapter l'hypothèse récursive. En effet, l'hypothèse récursive fournie impose une contrainte de croissance stricte sur le compteur t du jeu sous-jacent, tandis que cette contrainte disparaît pour les expressions.

Comme l'exécution du code compilé pour la fonction commence par une instruction `nop`, il est immédiat de réduire la contrainte à une croissance large du compteur. Pour éliminer complètement cette contrainte, nous exploitons le transformateur de progression, qui élargit la post-condition permise au cas où l'état final n'est pas plus grand que l'état initial. Nous récupérons ensuite cette post-condition au sein d'une continuation, ce qui nous permet d'éliminer le cas où la contrainte désirée n'est pas satisfaite. Les conditions obtenues par déroulement de l'application du transformateur se traitent alors de manière similaire à celles obtenues pour le cas des expressions, car les pré- et post-conditions du contrat d'une expression sont de même nature que les pré- et post-conditions du contrat d'une fonction.

□

code source :

$$f[L_0]((x_i)_{0\leq i<m}) = e$$

code compilé :

$$\begin{array}{rl} \texttt{START}_f\texttt{:} & \texttt{nop} \\ & [\![e]\!]_{m+1,[(x_i\to i+1)_{0\leq i<m}],[L_0\to(1,\texttt{EXIT})]} \\ & \texttt{cut } m \\ \texttt{EXIT:} & \texttt{swap} \\ & \texttt{jmp} \end{array}$$

transformateur nommé associé :

$$\left[\begin{array}{l} \mathcal{I}_{\texttt{asm},\forall}, \textit{all-fun} \\ c_{glob}, \texttt{pc}_R, (v_i)_{0\leq i<m}, \Sigma_0, k \end{array}\right] \Rightarrow \left[\ v_R\ \right]$$

nop
continuation *divg*()
  progression
    let $t \leftarrow (t$ si now s'écrit $(s,t))$
    continuation *ret*(L, v_R, Σ_1)
      let $\Pi_0 \leftarrow$ le canal d'entrée/sortie de now
      let $D \leftarrow \{(X,t) \mid (e,(\Sigma_0,\Pi_0),k) \longrightarrow_F^* X\}$
      intro *all-fun*$(g, \texttt{pc}_R, (v_i)_{0\leq i<m_g}, \Sigma_0, k)$: $\mathcal{C}_{rec}(F, c_{glob}, D)$
      by
        switch
        case $(t' < t$ si now s'écrit $(s,t'))$:
          ret
          choose v_R such that $\bot$
        default :
          all-fun
        end-switch
      in
        $W_e[\Sigma_0 \leftarrow \Sigma_0[+(x_i \leftarrow v_i)_{0\leq i<m}], k \leftarrow [L_0]\{(x_i)_{0\leq i<m} : k\}]$
        cut$[n \leftarrow m]$
        let $v_R \leftarrow v_R, \Sigma_1 \leftarrow \Sigma_1$
      end-intro
      let $v_R \leftarrow v_R, \Sigma_1 \leftarrow \Sigma_1, L \leftarrow L_0$
    end-continuation
  end-progression
  swap
  jmp
  let $v_R \leftarrow v_R$
end-continuation

FIGURE 6.15 – Code compilé et transformateur pour une déclaration de fonction

6.4 Preuve de la simulation en avant

Pour démontrer le résultat de simulation en avant, nous allons utiliser les mêmes outils que pour la simulation en arrière, c'est-à-dire les contrats et transformateurs nommés. La preuve suit en fait exactement le même schéma, en employant l'autre conversion des systèmes de transition en jeu et des contrats différents. Cette section suit donc la même structure que la section 6.3. Pour rendre la similitude complètement explicite, nous redéfinissons les notions jouant le même rôle que dans la preuve de simulation en arrière sous le même nom.

6.4.1 Énoncé équivalent en termes de garanties

Nous utilisons la même conversion en système de transition ordonné, telle que définie par la définition 6.10. Cependant, comme la simulation en avant correspond cette fois à un énoncé d'existence d'une trace d'exécution plutôt que d'un énoncé universel, nous utilisons l'autre type de traduction en jeu.

Définition 6.32 (Jeu existentiel associé au langage cible)**.** Soit c le code d'un programme cible. Le *jeu existentiel associé* à c est le jeu $\mathbb{G}_{c,\exists}$ défini comme le jeu existentiel local (cf. définition 4.76) associé au système de transition ordonné associé à c.

Comme pour la simulation en arrière, nous pouvons appliquer le lemme 4.80 de traduction des garanties correspondant au jeu ainsi construit pour reformuler le résultat voulu en termes de la validité d'un ensemble de garanties, regroupées sous la structure d'un contrat. Ce contrat exprime le fait que l'exécution du programme compilé consomme une trace d'exécution du programme source, dont les états initiaux et finaux correspondent à ceux de l'exécution du programme compilé. Notons qu'il s'agit précisément du contrat inverse de celui correspondant à la simulation en arrière, lequel correspond à la production d'une trace d'exécution du programme source.

Définition 6.33 (Contrat pour la simulation en avant)**.** Soit F un programme source et c un programme cible. Le *contrat nommé correspondant à la simulation en avant de F par c*, noté $\overrightarrow{\mathcal{C}}(F, c)$, est le contrat nommé défini vis-à-vis du support de $\mathbb{G}_{c,\exists}$ par :

(A) l'unique paramètre auxiliaire, correspondant à la trace d'exécution du programme source, est une séquence finie ou infinie d'états de la machine abstraite du langage source $\mathcal{T} = (s_n)$, dont deux états consécutifs sont liés par $\longrightarrow_F$, et dont le dernier état, s'il existe, est un état d'arrêt valide. Le premier état de la trace est indexé par 0.

(B) aucune valeur de retour auxiliaire

(P) la pré-condition sur l'état initial est qu'il soit de la forme $((0, S_0, \Pi_0), t)$, et que le premier élément de $\mathcal{T}$ soit de la forme

$$(\&\texttt{main}(), (\Sigma_0, \Pi_0), \top)$$

(Q) la post-condition sur l'état final exprime que s'il s'agit d'un état limite correspondant au canal d'entrée/sortie Π_∞, alors $\mathcal{T}$ est infinie et la borne supérieure des canaux d'entrée/sortie associés est Π_∞, et que s'il s'agit d'un

état non limite, alors il est de la forme $((\texttt{pc}, (S_0; w_R), \Pi_1), t)$ (où S_0 correspond à la pile de l'état initial) et
(i) `pc` désigne l'instruction `exit` au sein de c
(ii) $\mathcal{T}$ est finie
(iii) w_R représente la valeur source v_R correspondant au dernier état $(v_R, (\Sigma_1, \Pi_1), \top)$ de $\mathcal{T}$, qui est nécessairement de cette forme par définition de $\mathcal{T}$.

Lemme 6.34. *L'énoncé du théorème 6.8 est équivalent à ce que pour tout programme source bien formé F, si le résultat de la compilation est le programme cible c, alors toutes les garanties associées au contrat nommé $\overrightarrow{\mathcal{C}}(F, c)$ sont valides dans le jeux $\mathbb{G}_{c,\exists}$.*

Démonstration. Il s'agit d'une application directe du lemme 4.80. Pour obtenir la correspondance avec les deux conditions du lemme, nous effectuons une disjonction sur le caractère limite ou non de $\mathcal{T}$, qui est alors respectivement un témoin pour les propriétés d'atteinte infinie et finie. □

Comme la trace d'exécution à simuler sera un paramètre de presque tous nos contrats pour la preuve de simulation en avant, nous réutiliserons le paramètre $\mathcal{T}$ dans les contrats subséquents sans préciser à chaque fois de quoi il s'agit. Pour éviter de devoir construire des suffixes de cette trace, nous utiliserons ce paramètre comme une constante globale, et noterons la position de l'état à simuler dans la trace par le paramètre $t_{\texttt{src}}$, appelé curseur de simulation. De cette manière, la progression dans la trace à simuler correspond à la mise à jour d'un compteur. De plus, cela permet de garantir la cohérence d'une exécution infinie vis-à-vis de la trace initiale plus facilement qu'en manipulant de multiples traces d'exécution. Comme pour $\mathcal{T}$, le paramètre $t_{\texttt{src}}$ désignera toujours ce compteur dans les contrats que nous définissons par la suite.

Nous continuons ensuite de suivre le même schéma que pour la simulation en arrière, en transformant le résultat de simulation en avant par la validité d'un énoncé nommé. Pour cela, nous complétons le contrat par un contexte étiqueté $\Gamma_{c,\exists}$, qui est la version existentielle de $\Gamma_{c,\forall}$. Nous employons exactement les mêmes contrats pour les instructions déterministes.

Pour le cas particulier de l'instruction non déterministe `in`, nous donnons le contrat explicitement dans la figure 6.16. Remarquons que la valeur lue joue le rôle d'un paramètre dans ce contrat, tandis que dans la version universelle nous avions utilisé une valeur de retour pour la valeur lue. Ce changement reflète la nature existentielle du jeu. Au lieu que la valeur lue ne soit arbitrairement choisie par le non-déterminisme, nous pouvons choisir la valeur qui sera lue.

Au sein des interfaces des transformateurs utilisés, nous utilisons la notation $\mathcal{I}_{\texttt{asm},\exists}$ pour représenter l'association d'hypothèses correspondant aux contrats de $\Gamma_{c,\exists}$. À cause du cas de l'instruction non-déterministe `in` dont le contrat n'a pas la même signature, cette association n'est pas parfaitement identique à $\mathcal{I}_{\texttt{asm},\forall}$.

Lemme 6.35. *Toutes les garanties du contexte $\Gamma_{c,\exists}$ ainsi obtenues sont satisfaites par $\mathbb{G}_{c,\exists}$.*

Démonstration. Il s'agit d'une conséquence directe du lemme 4.80. En effet, toutes les propriétés d'accessibilité correspondante peuvent être réalisées par une trace

$$\forall v_R, n. \left\langle \begin{pmatrix} v_R \text{ représente } n \\ \text{et } \mathsf{now} \text{ s'écrit} \\ ((\mathtt{pc}, S, \Pi), t) \\ \text{avec } I_{\mathtt{pc}} = \mathtt{in} \end{pmatrix} \hookrightarrow \begin{pmatrix} \mathsf{now} = ((\mathtt{pc}+1, (S; v_R), \Pi +\!\!+ I(n)), t+1) \\ \text{si } \mathsf{old} \text{ s'écrit } ((\mathtt{pc}, S, \Pi), t) \end{pmatrix} \right\rangle$$

FIGURE 6.16 – Contrat pour `in`

d'évolution à deux états, où le deuxième état est donné exactement par une transition de la machine virtuelle. □

Définition 6.36. Soit F un programme source et c un programme cible. L'*énoncé nommé correspondant à la simulation en avant de F par c*, noté $\overrightarrow{\mathcal{E}}(F, c)$, est l'énoncé nommé défini vis-à-vis du support de $\mathbb{G}_{c,\exists}$ par :

— le contexte $\Gamma_{c,\exists}$ équipé de l'association d'hypothèse $\mathcal{I}_{\mathtt{asm},\exists}$,
— le contrat $\overrightarrow{\mathcal{C}}(F, c)$.

Lemme 6.37. *L'énoncé du théorème 6.8 est équivalent à ce que pour tout programme $\mathcal{L}_{\mathtt{s}}$ bien formé, si le résultat de la compilation est le programme c, alors l'énoncé nommé $\overrightarrow{\mathcal{E}}(F, c)$ est valide.*

Démonstration. Conséquence immédiate des lemmes 6.34 et 6.35. □

6.4.2 Réduction à la compilation correcte des fonctions

Comme pour la preuve de la simulation en arrière, nous réduisons maintenant la preuve du résultat de simulation en avant à la correction de la compilation des fonctions. Nous donnons donc tout d'abord le schéma de contrat attendu pour la compilation d'une fonction source. Comme le protocole d'appel est inchangé, la partie de ce contrat correspondant à la structure de la pile et au positionnement de la tête de lecture est inchangée par rapport au cas de la simulation en arrière. Les conditions de représentation liant l'état du langage cible à l'état du programme source sont également inchangées, bien que les états sources soient obtenus de manière différente, à partir de la trace $\mathcal{T}$. Nous marquons en gras les parties du contrat qui ont changé vis-a-vis de la définition 6.18.

Définition 6.38 (Contrat associé à une fonction source)**.** Soit f une fonction (pré)définie dans F, $\mathtt{START}_f$ l'étiquette associée à f, et $(c_i)_{i\in I}$ une famille de programmes $\mathcal{L}_{\mathtt{t}}$. Le *contrat nommé associé* à f et à la famille $(c_i)_{i\in I}$ est le contrat $\mathcal{C}_{fun}(F, f, (c_i)_{i\in I})$ défini par

(A) les paramètres auxiliaires $c_{glob}, \mathtt{pc}_R, \mathcal{T}, t_{\mathtt{src}}$, où c_{glob} est un programme du langage $\mathcal{L}_{\mathtt{t}}$ dit code global, et $\mathtt{pc}_R$ est une tête de lecture.

(B) la valeur de retour auxiliaire $t'_{\mathtt{src}}$ correspondant à la mise à jour du curseur de simulation.

(P) la pré-condition constituée de l'ensemble des états de $\mathbb{G}_{c_{glob},\exists}$ de la forme

$$((\mathtt{pc}_f, (S_0; \mathtt{RA}; (w_i)_{0\le i<n}), \Pi_0), t)$$

tels que

(i) l'état de $\mathcal{T}$ indexé par $t_{\texttt{src}}$ est de la forme

$$((), (\Sigma_0, \Pi_0), \&f((v_i)_{0\leq i<n}, k))$$

où le canal d'entrée/sortie Π_0 est identique à celui de la machine virtuelle.

(ii) pour tout $i \in I$, c_i est une sous-séquence de c_{glob}.

(iii) $\texttt{pc}_f$ correspond à la position de $\texttt{START}_f$ dans c_{glob}.

(iv) $\texttt{pc}_R$ correspond à la position de `RA` dans c_{glob}.

(v) les valeurs cibles $(w_i)_{0\leq i<n}$ représentent correctement les valeurs sources $(v_i)_{0\leq i<n}$ introduites ci-dessus.

(Q) la post-condition exprimant que soit l'état final est l'état limite correspondant au canal d'entrée/sortie limite de $\mathcal{T}$ (qui n'existe que si $\mathcal{T}$ est infinie), soit l'état final est de la forme

$$((\texttt{pc}_R, (S_0; w), \Pi_1), t')$$

où S_0 est la pile de l'état d'entrée, et de plus

(i) $t'_{\texttt{src}} \geq t_{\texttt{src}}$, et l'état de $\mathcal{T}$ indexé par la valeur de retour auxiliaire $t'_{\texttt{src}}$ est de la forme

$$(v_R, (\Sigma_0, \Pi_1), k)$$

où Π_1 est le canal d'entrée/sortie de l'état final, et k est le code de continuation correspondant à la suite de l'appel dans l'état source initial.

(ii) la valeur cible w représente correctement la valeur source v_R définie ci-dessus.

Définition 6.39 (Compilation correcte d'une fonction)**.** Soit F le code du programme source, f une fonction (pré)définie dans F, et (c_g) la famille des code compilés pour les fonctions (pré)définies dans F. Nous disons que f est compilée de manière *correcte* si l'énoncé nommé $\mathcal{E}_{glob}(F, f, (c_g))$ composé

— du contrat nommé $\mathcal{C}_{fun}(F, f, (c_g))$

— du contexte $\Gamma_{c_{glob},\exists}$, dépendant du paramètre auxiliaire c_{glob} dit « code global » du contrat nommé $\mathcal{C}_{fun}(F, f, (c_g))$

est valide.

Lemme 6.40. *Soit F un programme source, compilé vers le programme c, ainsi que (c_f) la famille des codes générés séparément pour chaque fonction (pré)définie dans F. Supposons que* `main` *soit compilée de manière correcte. Alors le résultat de simulation en avant est vérifié.*

Démonstration. Comme pour le cas de la simulation en arrière, le lemme 6.37 montre qu'il suffit de démontrer la validité de $\overrightarrow{\mathcal{E}}(F, c)$. Il suffit pour cela d'employer le transformateur suivant, très proche de celui employé pour la simulation en arrière (cf. lemme 6.21) :

$$\left[\begin{array}{l} \mathcal{I}_{\texttt{asm},\exists} \\ \mathcal{T} \end{array}\right] \Rightarrow []$$
$$\texttt{aload}[\texttt{L} \leftarrow \texttt{EXIT}]$$
$$\texttt{aload}[\texttt{L} \leftarrow \texttt{MAIN}]$$
$$\textit{jmp}$$
$$W_{\mathcal{E}_{glob}(F,\texttt{main},(c_f))}[c_{glob} \leftarrow c, \texttt{pc}_R \leftarrow \texttt{pc}_{\texttt{EXIT}}, t_{\texttt{src}} \leftarrow 1]$$
$$\texttt{skip}$$

où $\mathtt{pc}_{\mathtt{EXIT}}$ correspond à la position (connue) de l'étiquette `EXIT` dans c, et $W_{\mathcal{E}_{glob}(F,\mathtt{main},(c_f))}$ au transformateur nommé correspondant à la conversion de $\mathcal{E}_{glob}(F, \mathtt{main}, (c_f))$, l'énoncé nommé supposé valide. Les contraintes structurelles sont les mêmes que pour la simulation en arrière, et se vérifient donc de manière identique. La pré-condition de forme de l'état machine source pour l'appel à `main` est garantie par l'avancement du compteur en seconde position, ce qui par définition de la sémantique du langage source revient à transformer l'appel à `main` en code de continuation. Enfin, la post-condition à établir (celle de $\overrightarrow{\mathcal{C}}(F, c)$) coïncide avec celle de l'appel à `main` puisque l'état source renvoyé par `main` (s'il existe) est forcément un état d'arrêt valide. □

Nous établissons ensuite une structure récursive pour éliminer le caractère global de la correction des fonctions. Nous utilisons pour cela une structure récursive différente par rapport au cas de la simulation en arrière. En cas de divergence, nous devions garantir l'établissement d'une progression infinie dans le programme source, tandis que dans notre cas actuel nous devons simplement garantir une progression infinie dans la trace d'entrée.

Définition 6.41 (Ordre de progression de la récurrence)**.** Soit $(t_{\mathtt{src}}, t)$ et $(t'_{\mathtt{src}}, t')$ deux paires d'entiers naturels, formés d'un curseur de simulation et d'un compteur de temps du système de transition ordonné. Nous disons que $(t_{\mathtt{src}}, t)$ est strictement inférieure à $(t'_{\mathtt{src}}, t')$ si les composantes de la première paire sont toutes deux strictement inférieures à leur composante respective dans la seconde paire. Nous disons également qu'une telle paire est inférieure (largement) à une autre, noté $(t_{\mathtt{src}}, t) \preccurlyeq_p (t'_{\mathtt{src}}, t')$ si elle est égale ou strictement inférieure. Nous appelons l'ordre obtenu *l'ordre de progression de la récurrence.*

Définition 6.42 (Fonction de progression de la récurrence)**.** Nous définissons la *fonction de progression de la récurrence* comme la fonction qui aux paramètres auxiliaires précédemment nommés $c_{glob}, \mathtt{pc}_R, \mathcal{T}, t_{\mathtt{src}}$ d'un contrat de fonction $\mathcal{C}_{fun}(F, f, (c_i)_{i\in I})$ et à son état initial $((\mathtt{pc}, S, \Pi_0), t)$ associe la paire $(t_{\mathtt{src}}, t)$.

Définition 6.43 (Contrat récursif)**.** Soit F un programme source, c_{glob} un programme de cible, et D un ensemble de paires d'entiers. Nous définissons le *schéma général d'un contrat de fonction récursif* $\mathcal{C}_{rec}(F, c_{glob}, D)$ comme le contrat suivant, dérivé du contrat d'une fonction en transformant le paramètre c_{glob} en paramètre externe du contrat, en transformant le nom de la fonction en paramètre auxiliaire (interne), et en remplaçant la contrainte d'inclusion d'une famille de codes dans c_{glob} par une contrainte de progression :

- **(A)** les paramètres auxiliaires $g, \mathtt{pc}_R$ constitués d'un nom de fonction g, d'une tête de lecture $\mathtt{pc}_R$, et des paramètres auxiliaires implicites $\mathcal{T}, t_{\mathtt{src}}$.
- **(B)** la valeur de retour auxiliaire $t'_{\mathtt{src}}$ correspondant à la mise à jour du curseur de simulation.
- (P) la pré-condition constituée de l'ensemble des états de $\mathbb{G}_{c_{glob},\exists}$ de la forme

$$((\mathtt{pc}_g, (S_0; \mathtt{RA}; (w_i)_{0\le i<n}), \Pi_0), t)$$

tels que

(i) l'état de $\mathcal{T}$ indexé par $t_{\mathtt{src}}$ est de la forme

$$((), (\Sigma_0, \Pi_0), \&g((v_i)_{0\leq i<n}, k))$$

où le canal d'entrée/sortie Π_0 est identique à celui de la machine virtuelle.

(ii) $(t_{\mathtt{src}}, t) \in D$

(iii) $\mathtt{pc}_g$ correspond à la position de l'étiquette $\mathtt{START}_g$ associée à g dans c_{glob}.

(iv) $\mathtt{pc}_R$ correspond à la position de `RA` dans c_{glob}.

(v) les valeurs cibles $(w_i)_{0\leq i<n}$ représentent correctement les valeurs sources $(v_i)_{0\leq i<n}$ introduites ci-dessus.

(Q) la post-condition exprimant que soit l'état final est l'état limite correspondant au canal d'entrée/sortie limite de $\mathcal{T}$ (qui n'existe que si $\mathcal{T}$ est infinie), soit l'état final est de la forme

$$((\mathtt{pc}_R, (S_0; w), \Pi_1), t')$$

où S_0 est la pile de l'état d'entrée, et de plus

(i) $t'_{\mathtt{src}} \geq t_{\mathtt{src}}$, et l'état de $\mathcal{T}$ indexé par la valeur de retour auxiliaire $t'_{\mathtt{src}}$ est de la forme

$$(v_R, (\Sigma_0, \Pi_1), k)$$

où Π_1 est le canal d'entrée/sortie de l'état final, et k est le code de continuation correspondant à la suite de l'appel dans l'état source initial.

(ii) la valeur cible w représente correctement la valeur source v_R définie ci-dessus.

Définition 6.44 (Compilation localement correcte d'une fonction)**.** Soit F le code du programme source, f une fonction (pré)définie dans F, et c_f le code produit pour f. Nous disons que f est compilée en c_f de manière *localement correcte* si l'énoncé nommé $\mathcal{E}_{loc}(F, f, c_f)$ composé

— du contrat nommé $\mathcal{C}_{fun}(F, f, (c_g)_{g\in\{f\}}$

— du contexte $\Gamma_{c_{glob},\exists}$ dépendant du paramètre auxiliaire c_{glob} dit « code global », étendu par l'association du nom d'hypothèse `all-fun` au contrat

$$\mathcal{C}_{rec}(F, c_{glob}, \{x \mid x \succ_p (t_{\mathtt{src}}, t)\})$$

dépendant des mêmes valeurs que celles accessible dans la pré-condition du contrat nommé associé à f

est valide.

Nous obtenons un résultat de réduction à la correction locale de la compilation.

Lemme 6.45. *Soit F le code du programme source, (c_f) les codes générés pour les fonctions (pré)définies dans F. Supposons que toute fonction f soit compilée en c_f de manière localement correcte. Alors le résultat de simulation en avant est valide.*

Démonstration. Grâce au lemme 6.40, il suffit d'établir l'énoncé nommé $\mathcal{E}_{glob}(F, \mathtt{main}, (c_f))$. Comme prévu, nous appliquons un transformateur nommé de

nature récursive

$$\left[\begin{array}{l}\mathcal{I}_{\mathtt{asm},\exists}\\ c_{glob}, \mathtt{pc}_R, \mathcal{T}, t_{\mathtt{src}}\end{array}\right] \Rightarrow \left[\ t'_{\mathtt{src}}\ \right]$$
$$\mathsf{let}\ g \leftarrow \mathtt{main}$$
$$\mathsf{call\text{-}rec}\ \mathtt{all\text{-}fun}(g, \mathtt{pc}_R, t_{\mathtt{src}}) :\ \mathcal{C}_{rec}(F, c_{glob}, \mathbb{N}^2)$$
$$\mathsf{variant}\{V\}$$
$$\mathsf{progress}\{\preccurlyeq_p\}$$
$$\quad W_{all}$$
$$\mathsf{diverges}(H, g_G, g_{fn}, g_{RA}, g_{t_{\mathtt{src}}} \mid clos)$$
$$\quad \mathsf{choose}\ clos \in H\ \mathsf{such\ that}\ \top$$
$$\quad \mathsf{let}\ clos \leftarrow clos, t'_{\mathtt{src}} \leftarrow 0$$
$$\mathsf{end\text{-}call\text{-}rec}$$

où

— V désigne l'application de la fonction de progression.

— W_{all} désigne un transformateur nommé regroupant les conversions des énoncés de correction locale de chaque fonction.

Nous construisons le transformateur W_{all} par regroupement, exactement comme nous l'avions fait pour la preuve de la simulation en arrière (cf. lemme 6.26). De la même manière, la condition de vérification se réduit au cas des situations limites de la récurrence.

Posons alors H_0 l'ensemble des éléments de H au moins aussi grand que l'élément $clos$ choisi. Alors par définition de l'ordre de progression et par monotonicité de g_G, $g_G(H_0)$ doit être une suite infinie strictement croissante d'états du système de transition ordonné associé à la machine virtuelle, dont la borne supérieure est de la forme Π_∞. De plus, $g_{t_{\mathtt{src}}}(H_0)$ correspond à une suite infinie strictement croissante de curseurs de simulation. À cause de la pré-condition des appels récursifs, les états source de $\mathcal{T}$ correspondant à ces curseurs ont le même canal d'entrée/sortie que l'élément de $g_G(H_0)$ correspondant. En particulier, Π_∞ est bien la borne supérieure des canaux d'entrée/sortie de $\mathcal{T}$, ce qui est la post-condition voulue pour un état limite. □

6.4.3 Énoncé de compilation correcte pour les expressions

Nous suivons toujours le fil conducteur donné par la preuve de la simulation en arrière. L'étape suivante est donc de donner les énoncés nommés correspondant au contrats des expressions compilées. Comme pour les fonctions, le protocole d'évaluation des expressions est inchangé. La partie du contrat des expressions correspondant à la structure de la pile et au positionnement de la tête de lecture est inchangée. La partie liant ces informations à l'état du programme source est également inchangée, bien que ces états sources ne soient pas obtenus de la même manière. Nous marquons en gras les parties des contrats modifiées vis-à-vis des définitions 6.27 et 6.28.

Définition 6.46 (Contrat associé à une expression compilée)**.** Soit F un programme du langage $\mathcal{L}_{\mathtt{s}}$. Soient $e, (\mathtt{fs}, \sigma_v, \sigma_l)$ une expression e équipée des paramètres de sa compilation, et $c = [\![e]\!]_{\mathtt{fs},\sigma_v,\sigma_l}$. Le *contrat associé à l'expression e munie de* $(\mathtt{fs}, \sigma_v, \sigma_l)$ est le contrat nommé $\mathcal{C}_{expr}(F, e, (\mathtt{fs}, \sigma_v, \sigma_l), c)$ défini par :

(A) le paramètre auxiliaire c_{glob} (un programme du langage $\mathcal{L}_{\mathtt{t}}$, le « code global »), et les paramètres $\mathcal{T}, t_{\mathtt{src}}$

(B) la valeur de retour auxiliaire $t'_{\mathtt{src}}$ correspondant à la mise à jour du curseur de simulation

(P) une pré-condition constituée de l'ensemble des états de $\mathbb{G}_{c_{glob},\forall}$ de la forme

$$((\mathtt{pc}, S_0, \Pi_0), t)$$

tels que

(i) l'état de $\mathcal{T}$ indexé par $t_{\mathtt{src}}$ est de la forme

$$(e, (\Sigma_0, \Pi_0), k)$$

(ii) c est une sous-séquence de c_{glob}, commençant à la position $\mathtt{pc}$.

(iii) pour tout x appartenant au domaine de σ_v, la $(\mathtt{fs} - \sigma_v(x))$-ième valeur depuis le sommet de la pile S_0 (en partant de 0 pour le sommet) contient une valeur représentant correctement la dernière valeur de la séquence $\Sigma_0(x)$.

(iv) S_0 contient au moins $\mathtt{fs}$ valeurs.

(v) pour tout L appartenant au domaine de σ_l, l'étiquette $\mathtt{EXIT}_L$ associée à L existe dans c_{glob}.

(Q) une post-condition paramétrée par les paramètres auxiliaires, l'état d'entrée de la forme ci-dessus, et les retours auxiliaires, donnée par l'ensemble des états de $\mathbb{G}_{c_{glob},\forall}$ de la forme

$$((\mathtt{pc}', (S_1; w), \Pi_1), t')$$

tels que

(i) $t'_{\mathtt{src}} \geq t_{\mathtt{src}}$, et l'état de $\mathcal{T}$ indexé par la valeur de retour $t'_{\mathtt{src}}$ est de la forme

$$(v_R, (\Sigma_1, \Pi_1), k)$$

où k est le code de continuation correspondant à l'état de $\mathcal{T}$ indexé par $t_{\mathtt{src}}$.

(ii) la différence $\mathtt{pc}' - \mathtt{pc}$ correspond au nombre d'instructions dans c.

(iii) S_1 fait la même taille que la pile initiale S_0, et leurs valeurs coïncident à l'exception potentielle, pour tout x appartenant au domaine de σ_v, de la $(\mathtt{fs} - \sigma_v(x))$-ième valeur (partant de 0) depuis le sommet de la pile.

(iv) pour tout x appartenant au domaine de σ_v, la $(\mathtt{fs} - \sigma_v(x))$-ième valeur depuis le sommet de la pile S_1 représente correctement la dernière valeur de la séquence $\Sigma_1(x)$.

(v) la valeur cible w représente correctement la valeur source v_R.

(vi) Σ_1 est une mise à jour de Σ_0 pour les variables appartenant au domaine de σ_v, où Σ_0 est l'association de variables correspondant à l'état source construit pour l'état initial. Autrement dit, les valeurs pour les autres variables ainsi que les valeurs précédentes des variables du domaine de σ_v sont les mêmes dans Σ_1 et Σ_0.

Définition 6.47 (Hypothèses additionnelles associées à une expression compilée)**.** Soit F un programme du langage $\mathcal{L}_{\mathtt{s}}$. Les hypothèses additionnelles associées à l'expression e munie de $(\mathtt{fs}, \sigma_v, \sigma_l)$ sont les hypothèses suivantes, qui dépendent des mêmes paramètres auxiliaires que le contrat nommé associé à e ainsi que de son état initial $((\mathtt{pc}, S_0, \Pi_0), t)$:

(`all-fun`) L'hypothèse nommée ***`all-fun`***, associée au contrat

$$\mathcal{C}_{rec}(F, c_{glob}, \{(t_0, t) \mid t_0 \geq t_{\mathtt{src}}\})$$

(`divg`) L'hypothèse nommée ***`divg`*** associée à un contrat $\mathcal{C}_{div}$ de type continuation (dont la post-condition est vide), correspondant aux comportements divergents. Ce contrat n'a ni paramètre auxiliaire ni valeur de retour auxiliaire, et sa pré-condition est soit l'ensemble vide si $\mathcal{T}$ est fini, soit l'ensemble singleton contenant la borne supérieure des canaux d'entrée/sortie des états de $\mathcal{T}$ sinon.

(`ret`) L'hypothèse nommée ***`ret`*** associée au contrat $\mathcal{C}_{ret}$ de type continuation (dont la post-condition est vide), correspondant au cas d'un retour au sein de l'expression compilée. Les paramètres auxiliaires de ce contrat sont donnés par l'étiquette source L correspondant au retour, et le curseur de simulation $t'_{\mathtt{src}}$ correspondant à l'état du programme source au niveau du retour. La pré-condition de ***`ret`*** est constituée de l'ensemble des états $((\mathtt{pc}_L, (S_1; w), \Pi_1), t')$ tels que

(i) $t'_{\mathtt{src}} \geq t_{\mathtt{src}}$, et l'état de $\mathcal{T}$ indexé par $t'_{\mathtt{src}}$ est de la forme

$$(v_R, (\Sigma_1, \Pi_1), \mathtt{return}[L]\ k)$$

où k est le code de continuation correspondant à l'état de $\mathcal{T}$ indexé par $t_{\mathtt{src}}$.

(ii) L est associé à $(\mathtt{fs}_L, \mathtt{EXIT}_L)$ par σ_l, et $\mathtt{pc}_L$ désigne l'emplacement de $\mathtt{EXIT}_L$ dans c_{glob}.

(iii) S_0 contient $\mathtt{fs} - \mathtt{fs}_L$ valeurs supplémentaires par rapport à S_1, et l'extension de S_1 par les $\mathtt{fs} - \mathtt{fs}_L$ valeurs supplémentaires de S_0 coïncide avec S_0 à l'exception potentielle, pour tout x appartenant au domaine de σ_v, de la $(\mathtt{fs} - \sigma_v(x))$-ième valeur (partant de 0) depuis le sommet de la pile.

(iv) pour tout x appartenant au domaine de σ_v, la $(\mathtt{fs} - \sigma_v(x))$-ième valeur depuis le sommet de la pile S_1 représente correctement la dernière valeur de la séquence $\Sigma_1(x)$.

(v) la valeur cible w représente correctement la valeur v_R.

(vi) Σ_1 est une mise à jour de Σ_0 pour les variables appartenant au domaine de σ_v.

La définition de la compilation correcte d'une expression est identique la définition donnée pour la simulation en arrière (cf. definition 6.29), à l'exception du contexte $\Gamma_{c_{glob},\forall}$ qui est remplacé par son pendant existentiel.

Définition 6.48 (Compilation correcte d'une expression)**.** Soit F un programme du langage $\mathcal{L}_{\mathtt{s}}$. Nous disons qu'une expression e munie de $(\mathtt{fs}, \sigma_v, \sigma_l)$ est *correctement compilée* en $c = [\![e]\!]_{\mathtt{fs},\sigma_v,\sigma_l}$ si l'énoncé nommé $\mathcal{E}_{expr}(F, e, (\mathtt{fs}, \sigma_v, \sigma_l), c)$ composé

— du contrat nommé associé à l'expression e munie de $(\mathtt{fs}, \sigma_v, \sigma_l)$

— du contexte étiqueté $\Gamma_{c_{glob},\exists}$ (dépendant du paramètre auxiliaire c_{glob} dit « code global ») étendu par les hypothèses additionnelles associées à e munie de $(\mathtt{fs}, \sigma_v, \sigma_l)$, avec les associations de noms présentées pour chaque hypothèse

est valide.

6.4.4 Compilation correcte des expressions

Nous passons maintenant à la preuve de correction de la compilation des expressions.

Lemme 6.49. *Soit F un programme du langage $\mathcal{L}_{\mathtt{s}}$. Alors toute expression e munie d'un triplet $(\mathtt{fs}, \sigma_v, \sigma_l)$ tel que*
(i) $\mathtt{fs} \in \mathbb{N}^*$
(ii) pour tout x dans le domaine de σ_v, $0 < \sigma_v(x) \leq \mathtt{fs}$
(iii) pour tout L associé à $(\mathtt{fs}_L, \mathtt{EXIT}_L)$ par σ_l, $0 \leq \mathtt{fs}_L \leq \mathtt{fs}$
est correctement compilée.

Démonstration. Nous procédons de la même manière que pour la preuve du lemme 6.30 pour la simulation en arrière, par induction sur la syntaxe de l'expression. Nous employons également des combinaisons de transformateur très proches de celles utilisées pour ce lemme. Les parties de la condition de vérification correspondant à la structure de l'état de la machine virtuelle s'établissent de manière quasiment identique.

Nous rappelons que nous utilisons la notation W_e pour représenter le transformateur nommé correspondant à la conversion de l'énoncé nommé obtenu inductivement pour une sous-expression e, et pour les paramètres de compilation correspondant à cette sous-expression dans le schéma de compilation.

Cas des variables

$$[\![x]\!]_{\mathtt{fs},\sigma_v,\sigma_l} = \mathtt{load}\ \ (\mathtt{fs} - \sigma_v(x))$$

transformateur nommé associé :

$$\begin{array}{l} \left[\begin{array}{l} \mathcal{I}_{\mathtt{asm},\exists}, \mathtt{all\text{-}fun}, \mathit{divg}, \mathtt{ret} \\ c_{glob}, \mathcal{T}, t_{\mathtt{src}} \end{array}\right] \Rightarrow \left[\ t'_{\mathtt{src}}\ \right] \\ \mathtt{load}[n \leftarrow (\mathtt{fs} - \sigma_v(x))] \\ \mathsf{let}\ t'_{\mathtt{src}} \leftarrow t_{\mathtt{src}} + 1 \end{array}$$

Comme la pré-condition impose que l'état de départ corresponde à l'expression x, nous pouvons immédiatement obtenir la post-condition puisqu'il n'y a qu'une seule transition possible. De plus, l'état indexé par $t_{\mathtt{src}} + 1$ existe bien dans $\mathcal{T}$ car sinon celui indexé par $t_{\mathtt{src}}$ serait le dernier, et devrait être un état d'arrêt valide, ce qui n'est pas le cas. Les autres conditions sont purement structurelles, et se traitent comme pour le cas de la simulation en arrière.

Cas de l'expression vide

$$[\![()]\!]_{\mathtt{fs},\sigma_v,\sigma_l} = \mathtt{load}\ \ 0$$

transformateur nommé associé :

$$\begin{array}{l} \left[\begin{array}{l} \mathcal{I}_{\mathtt{asm},\exists}, \mathtt{all\text{-}fun}, \mathit{divg}, \mathtt{ret} \\ c_{glob}, \mathcal{T}, t_{\mathtt{src}} \end{array}\right] \Rightarrow \left[\ t'_{\mathtt{src}}\ \right] \\ \mathtt{load}[n \leftarrow 0] \\ \mathsf{let}\ t'_{\mathtt{src}} \leftarrow t_{\mathtt{src}} + 1 \end{array}$$

Cas de la prise d'adresse d'une fonction

$$[\![\&f]\!]_{\mathtt{fs},\sigma_v,\sigma_l} = \mathtt{aload\ START}_f$$

transformateur nommé associé :

$$\begin{array}{l}\left[\begin{array}{l}\mathcal{I}_{\mathtt{asm},\exists}, \mathtt{all\text{-}fun}, \mathtt{divg}, \mathtt{ret}\\ c_{glob}, \mathcal{T}, t_{\mathtt{src}}\end{array}\right] \Rightarrow \left[\begin{array}{l} t'_{\mathtt{src}}\end{array}\right]\\ \mathtt{aload}[\mathtt{L} \leftarrow \mathtt{START}_f]\\ \mathsf{let}\ t'_{\mathtt{src}} \leftarrow t_{\mathtt{src}} + 1\end{array}$$

Les preuves de ces deux cas ne présentent pas de différence notable vis-à-vis de la preuves déjà effectuée pour le cas des variable.

Cas de la mise en séquence

$$[\![e_1; e_2]\!]_{\mathtt{fs},\sigma_v,\sigma_l} = \left\{\begin{array}{l}[\![e_1]\!]_{\mathtt{fs},\sigma_v,\sigma_l}\\ \mathtt{pop\ 1}\\ [\![e_2]\!]_{\mathtt{fs},\sigma_v,\sigma_l}\end{array}\right.$$

transformateur nommé associé :

$$\begin{array}{l}\left[\begin{array}{l}\mathcal{I}_{\mathtt{asm},\exists}, \mathtt{all\text{-}fun}, \mathtt{divg}, \mathtt{ret}\\ c_{glob}, \mathcal{T}, t_{\mathtt{src}}\end{array}\right] \Rightarrow \left[\begin{array}{l} t'_{\mathtt{src}}\end{array}\right]\\ W_{e_1}^{+1}[t_{\mathtt{src}} \leftarrow t_{\mathtt{src}} + 1]\\ \mathtt{pop}[n \leftarrow 1]\\ W_{e_2}[t_{\mathtt{src}} \leftarrow t'_{\mathtt{src}} + 1]\end{array}$$

où

$$W_{e_1}^{+1} = \left\{\begin{array}{l}\mathsf{intro}\ \mathtt{ret}(t'_{\mathtt{src}}) :\ \mathcal{C}_{\mathtt{ret}}\\ \mathsf{by}\\ \quad \mathtt{ret}[t'_{\mathtt{src}} \leftarrow t'_{\mathtt{src}} + 1]\\ \mathsf{in}\\ \quad W_{e_1}\\ \mathsf{end\text{-}intro}\end{array}\right.$$

Le cas de la séquence introduit la première différence majeure vis-à-vis de la preuve que nous avons effectuée pour la simulation en arrière, à cause des contraintes d'inclusion pour les hypothèses additionnelles lors de l'utilisation des transformateurs associées aux sous-expressions. L'inclusion d'hypothèses se passe de manière similaire pour les hypothèses de nom `all-fun`, en utilisant la garantie de progression du curseur de simulation à la place d'une chaîne de transitions. Nous obtenons également l'inclusion pour les hypothèses de nom `divg` de manière immédiate puisque l'hypothèse est ici indépendante de l'expression à laquelle elle est associée. Par contre, si nous réutilisons la même structure de transformateur, c'est-à-dire si nous avions défini $W_{e_1}^{+1}$ comme W_{e_1}, alors l'inclusion entre les hypothèses de nom `ret` serait fausse pour la première expression e_1.

En effet, le problème vient de la forme de l'état attendu en position $t'_{\mathtt{src}}$ dans $\mathcal{T}$. L'hypothèse de continuation de nom `ret` associée à e_1 demande un état de la forme

$$(v_R, (\Sigma_1, \Pi_1), \mathtt{return}[L]\ (k; e_2))$$

où k est le code de continuation correspondant à l'état source associé à l'état initial du contrat de e, tandis que l'hypothèse de nom ***ret*** associée à e (ou à e_2) demande un état de la forme

$$(v_R, (\Sigma_1, \Pi_1), \texttt{return}[L]\ k)$$

qui est différente. Par contre, la définition de la sémantique du langage source indique que si l'on a un état sous la première forme, alors l'état suivant sera sous la seconde, ce qui revient à dire que le retour s'échappe de la séquence. Nous insérons donc un décalage du curseur de simulation au niveau de l'hypothèse ***ret*** via la construction $W_{e_1}^{+1}$, où $\mathcal{C}_{\texttt{ret}}$ est le contrat de l'hypothèse de retour associée à e_1 et aux $\texttt{fs}, \sigma_v, \sigma_l$ correspondants. Une fois cette construction appliquée, le problème de la contrainte d'inclusion disparaît. La partie de la contrainte correspondant à la progression du curseur de simulation se traite comme pour l'hypothèse récursive. Comme cette construction revient un certain nombre de fois par la suite, nous réutiliserons la notation W_e^{+1} pour l'application de cette construction à W_e.

Notons par ailleurs que la raison pour laquelle ce problème n'apparaissait pas dans le cas de la simulation en arrière est que nous avions choisi de générer la trace d'exécution du programme source sous une forme implicite, via un prédicat. Si nous avions choisi une génération explicite, sous la forme d'une variable auxiliaire répertoriant la portion de l'exécution du programme source déjà simulée, nous aurions dû faire une manipulation similaire.

Enfin, l'établissement des autres conditions ne présente pas de difficulté particulière. Les incrémentations du curseur de simulation correspondent aux règles d'évaluation spécifiques à la séquence, et permettent de garantir la bonne forme de l'état source au début du code compilé pour une sous-expression.

Cas de l'affectation

$$[\![x \leftarrow e]\!]_{\texttt{fs},\sigma_v,\sigma_l} = \begin{cases} [\![e]\!]_{\texttt{fs},\sigma_v,\sigma_l} \\ \texttt{load}\ 0 \\ \texttt{store}\ (\texttt{fs} + 1 - \sigma_v(x)) \end{cases}$$

transformateur nommé associé :

$$\begin{array}{l} \left[\begin{array}{l} \mathcal{I}_{\texttt{asm},\exists}, \textit{all-fun}, \textit{divg}, \textit{ret} \\ c_{glob}, \mathcal{T}, t_{\texttt{src}} \end{array}\right] \Rightarrow \left[\ t'_{\texttt{src}}\ \right] \\ W_e^{+1}[t_{\texttt{src}} \leftarrow t_{\texttt{src}} + 1] \\ \textit{load}[n \leftarrow 0] \\ \textit{store}[n \leftarrow (\texttt{fs} + 1 - \sigma_v(x))] \\ \mathsf{let}\ t'_{\texttt{src}} \leftarrow t'_{\texttt{src}} + 1 \end{array}$$

La preuve de la condition de vérification ne présente pas de cas fondamentalement différent de ce que nous avons fait jusqu'à présent.

Cas de l'expression conditionnelle

$$[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]_{\texttt{fs},\sigma_v,\sigma_l} = \begin{cases} & [\![e_1]\!]_{\texttt{fs},\sigma_v,\sigma_l} \\ & \texttt{jnz THEN} \\ & [\![e_3]\!]_{\texttt{fs},\sigma_v,\sigma_l} \\ & \texttt{aload EXIT} \\ & \texttt{jmp} \\ \texttt{THEN:} & \texttt{nop} \\ & [\![e_2]\!]_{\texttt{fs},\sigma_v,\sigma_l} \\ \texttt{EXIT:} & \texttt{nop} \end{cases}$$

transformateur nommé associé :

$$\left[\begin{array}{l} \mathcal{I}_{\texttt{asm},\exists}, \texttt{all-fun}, \texttt{divg}, \texttt{ret} \\ c_{glob}, \mathcal{T}, t_{\texttt{src}} \end{array}\right] \Rightarrow \left[\ t'_{\texttt{src}}\ \right]$$

$W_{e_1}^{+1}[t_{\texttt{src}} \leftarrow t_{\texttt{src}} + 1]$
$\texttt{jnz}[\texttt{L} \leftarrow \texttt{THEN}]$
switch
case $v_R = \top$:
  $W_{e_2}[t_{\texttt{src}} \leftarrow t'_{\texttt{src}} + 1]$
case $v_R = \bot$:
  $W_{e_3}[t_{\texttt{src}} \leftarrow t'_{\texttt{src}} + 1]$
  $\texttt{aload}[\texttt{L} \leftarrow \texttt{EXIT}]$
  $\texttt{jmp}$
  let $t'_{\texttt{src}} \leftarrow t'_{\texttt{src}}$
end-switch
$\texttt{nop}$
let $t'_{\texttt{src}} \leftarrow t'_{\texttt{src}}$

Cas d'un retour

$$[\![\texttt{return}[L_0]\ e]\!]_{\texttt{fs},\sigma_v,\sigma_l} = \begin{cases} & [\![e]\!]_{\texttt{fs},\sigma_v,\sigma_l} \\ & \texttt{cut } (\texttt{fs} - \texttt{fs}_{L_0}) \\ & \texttt{aload EXIT}_{L_0} \\ & \texttt{jmp} \end{cases}$$

transformateur nommé associé :

$$\left[\begin{array}{l} \mathcal{I}_{\texttt{asm},\exists}, \texttt{all-fun}, \texttt{divg}, \texttt{ret} \\ c_{glob}, \mathcal{T}, t_{\texttt{src}} \end{array}\right] \Rightarrow \left[\ t'_{\texttt{src}}\ \right]$$

$W_e^{+1}[t_{\texttt{src}} \leftarrow t_{\texttt{src}} + 1]$
$\texttt{cut}[n \leftarrow (\texttt{fs} - \texttt{fs}_{L_0})]$
$\texttt{aload}[\texttt{L} \leftarrow \texttt{EXIT}_{L_0}]$
$\texttt{jmp}$
$\texttt{ret}[L \leftarrow L_0]$
choose $t'_{\texttt{src}}$ such that $\bot$

Pour la pseudo-instruction `cut`, nous reprenons les mêmes méthodes que pour la simulation en arrière. Les instructions sous-jacentes étant déterministes, le comportement est identique.

Cas d'un bloc

$$[\![[L_0]\{\ (x_i)_{0\le i<m} : e\ \}]\!]_{\mathtt{fs},\sigma_v,\sigma_l} = \begin{cases} (m \text{ fois}) & \mathtt{load\ 0} \\ & [\![e]\!]_{\mathtt{fs}+m,\sigma_v[(x_i\leftarrow \mathtt{fs}+i)_{0\le i<m}],\sigma_l[L_0\leftarrow(\mathtt{fs},\mathtt{EXIT})]} \\ & \mathtt{cut}\ m \\ \mathtt{EXIT:} & \mathtt{nop} \end{cases}$$

transformateur nommé associé :

$$\left[\begin{array}{l} \mathcal{I}_{\mathtt{asm},\exists}, \mathtt{all\text{-}fun}, \mathit{divg}, \mathit{ret} \\ c_{glob}, \mathcal{T}, t_{\mathtt{src}} \end{array}\right] \Rightarrow \left[\ t'_{\mathtt{src}}\ \right]$$

$W_{m\times \mathtt{load\ 0}}$
continuation $K(t'_{\mathtt{src}})$
  intro $\mathit{ret}(L, t'_{\mathtt{src}})$: $\mathcal{C}_{ret}$
  by
    let $t'_{\mathtt{src}} \leftarrow t'_{\mathtt{src}} + 1$
    switch
    case $L = L_0$:
      K
    default :
      ret
    end-switch
  in
    $W_e[t_{\mathtt{src}} \leftarrow t_{\mathtt{src}} + 1]$
    $\mathit{cut}[n \leftarrow m]$
    let $t'_{\mathtt{src}} \leftarrow t'_{\mathtt{src}}$
  end-intro
end-continuation
nop
let $t'_{\mathtt{src}} \leftarrow t'_{\mathtt{src}} + 1$

où $\mathcal{C}_{\mathit{ret}}$ désigne le contrat associé à ret pour l'expression e associée à $\mathtt{fs} + m$, $\sigma_v[(x_i \leftarrow \mathtt{fs} + i)_{0\le i<m}]$, et $\sigma_l[L_0 \leftarrow (\mathtt{fs}, \mathtt{EXIT})]$. Comme pour le cas de la simulation en arrière, $W_{m\times \mathtt{load\ 0}}$ est défini comme la conversion d'un contrat pour la répétition à m reprises de l'instruction `load` 0. Nous pouvons en fait réutiliser le même contrat car il s'agit d'une exécution déterministe.

Comme pour le cas de la simulation en arrière, le cas du bloc ajoute de nouveaux types de conditions à cause de la modification des environnements de compilation. Nous pouvons établir les contraintes structurelles sur l'état de la machine virtuelle de la même manière que pour le cas de la simulation en arrière, et le changement du contrat associé à ret génère des conditions similaires à celles que nous avons déjà traitées pour le décalage du curseur à son niveau. Nous établissons donc ces contraintes de la même manière.

Cas d'un appel de fonction

$$[\![e_0((e_i)_{0<i\leq m})]\!]_{\mathtt{fs},\sigma_v,\sigma_l} = \begin{cases} & [\![e_0]\!]_{\mathtt{fs},\sigma_v,\sigma_l} \\ & \vdots \\ & [\![e_i]\!]_{\mathtt{fs}+i,\sigma_v,\sigma_l} \\ & \vdots \\ & [\![e_n]\!]_{\mathtt{fs}+m,\sigma_v,\sigma_l} \\ & \mathtt{load}\ m \\ & \mathtt{aload\ RA} \\ & \mathtt{store}\ (m+1) \\ & \mathtt{jmp} \\ \mathtt{RA:} & \mathtt{nop} \end{cases}$$

transformateur nommé associé :

$$\left[\begin{array}{l} \mathcal{I}_{\mathtt{asm},\exists}, \mathtt{all\text{-}fun}, \mathtt{divg}, \mathtt{ret} \\ c_{glob}, \mathcal{T}, t_{\mathtt{src}} \end{array}\right] \Rightarrow \left[\ t'_{\mathtt{src}}\ \right]$$

$W_{(e_i)_{0\leq i\leq m}}[t_{\mathtt{src}} \leftarrow t_{\mathtt{src}} + 1]$
$\mathtt{load}[n \leftarrow m]$
$\mathtt{aload}[\mathtt{L} \leftarrow \mathtt{RA}]$
$\mathtt{store}[n \leftarrow m+1]$
$\mathtt{jmp}$
choose g such that $v_0 = \&g$
$\mathtt{all\text{-}fun}[\mathtt{pc}_R \leftarrow \mathtt{pc}_{RA}]$
switch
case now est un état limite :
    $\mathtt{divg}$
default :
    skip
end-switch
$\mathtt{nop}$
let $t'_{\mathtt{src}} \leftarrow t'_{\mathtt{src}}$

Pour tout $l \in [0; m]$, nous construisons le transformateur nommé $W_{(e_i)_{0\leq i\leq l}}$ d'une manière similaire à sa version pour la simulation en arrière, en ajustant le contrat pour la simulation en avant. Le contrat établi par induction puis converti est donc le même que celui associé à l'expression d'appel, aux différences suivantes :

— l'état de $\mathcal{T}$ indexé par $t'_{\mathtt{src}}$ est de la forme

$$(e_{l+1}, (\Sigma_1, \Pi_1), v_0((v_j)_{0<j\leq l}, k, (e_j)_{l+1<j\leq m+1}))$$

où $e_{m+1} = ()$ pour simplifier les notations, et où v_0 doit être une adresse de fonction pour que cet état source soit bien formé.

— la pile est étendue par $l+1$ valeurs au lieu d'une seule, correspondant respectivement aux valeurs des $(v_i)_{0\leq i\leq l}$.

— la tête de lecture finale se trouve à la fin du code correspondant aux $(e_i)_{0\leq i\leq l}$ plutôt qu'à la fin du code de l'expression d'appel.

La condition de vérification est alors du même type que celles vérifiées jusqu'à présent, et nous l'établissons de la même manière.

$$[\![\texttt{while } e_1 \texttt{ do } e_2]\!]_{\texttt{fs},\sigma_v,\sigma_l} = \begin{cases} & \texttt{aload TEST} \\ & \texttt{jmp} \\ \texttt{BODY:} & \texttt{nop} \\ & [\![e_2]\!]_{\texttt{fs},\sigma_v,\sigma_l} \\ & \texttt{pop } 1 \\ \texttt{TEST:} & \texttt{nop} \\ & [\![e_1]\!]_{\texttt{fs},\sigma_v,\sigma_l} \\ & \texttt{jnz BODY} \end{cases}$$

transformateur nommé associé :

$$\left[\begin{array}{l} \mathcal{I}_{\texttt{asm},\exists}, \texttt{all-fun}, \texttt{divg}, \texttt{ret} \\ c_{glob}, \mathcal{T}, t_{\texttt{src}} \end{array}\right] \Rightarrow \left[\ t'_{\texttt{src}}\ \right]$$

$\texttt{aload}[\mathsf{L} \leftarrow \texttt{TEST}]$
$\texttt{jmp}$
continuation $K(t'_{\texttt{src}})$
  let $t'_{\texttt{src}} \leftarrow t_{\texttt{src}}$
  iter$(t'_{\texttt{src}})$
  invariant$\{I\}$
  variant$\{(t'_{\texttt{src}}, t)$ si now $= ((\mathrm{pc}, S, \Pi), t)\}$
  progress$\{\preccurlyeq_p\}$
    $\texttt{nop}$
    $W_{e_1}^{+1}[t_{\texttt{src}} \leftarrow t'_{\texttt{src}} + 2]$
    $\texttt{jnz}[\mathsf{L} \leftarrow \texttt{BODY}]$
    switch
    case $v_R = \top$:
      $\texttt{nop}$
      $W_{e_2}^{+1}[t_{\texttt{src}} \leftarrow t'_{\texttt{src}} + 2]$
      $\texttt{pop}[n \leftarrow 1]$
      let $t'_{\texttt{src}} \leftarrow t'_{\texttt{src}} + 1$
    case $v_R = \bot$:
      $K[t'_{\texttt{src}} \leftarrow t'_{\texttt{src}} + 1]$
      choose $t'_{\texttt{src}}$ such that $\bot$
    end-switch
  diverges$[H, f_G, f_{t'_{\texttt{src}}}]$
    $\texttt{divg}$
    choose $t'_{\texttt{src}}$ such that $\bot$
  end-iter
  choose $t'_{\texttt{src}}$ such that $\bot$
end-continuation

FIGURE 6.17 – Code compilé et transformateur pour une boucle

Cas d'une boucle Pour le cas de la boucle, nous utilisons le transformateur de nature itérative donné dans la figure 6.17. L'invariant I de la construction itérative est défini par les conditions suivantes :

(i) $t'_{\texttt{src}} \geq t_{\texttt{src}}$, et l'état de $\mathcal{T}$ d'index $t'_{\texttt{src}}$ est de la forme

$$(\texttt{while}\ e_1\ \texttt{do}\ e_2, (\Sigma_1, \Pi_1), k)$$

où k est le code de continuation donné pour l'état de $\mathcal{T}$ d'index $t_{\texttt{src}}$, qui à cause de la pré-condition du contrat de l'expression de boucle est forcément de la forme

$$(\texttt{while}\ e_1\ \texttt{do}\ e_2, (\Sigma_0, \Pi_0), k)$$

(ii) La présence de la tête de lecture au niveau de l'étiquette `TEST`.

(iii) Les mêmes conditions liant $(S_0, \Sigma_0, S_1, \Sigma_1, \texttt{fs}, \sigma_v)$ que dans la post-condition d'une expression compilée, où S_0, S_1 représentent respectivement la pile initiale et la pile courante. Autrement dit, l'absence de mise à jour de la pile à d'autres emplacements que ceux désignés pour les variables, la représentation correcte des variables de Σ_1 dans S_1, et le fait que Σ_1 soit une mise à jour de Σ_0.

La majorité des conditions obtenues se prouvent comme pour les constructions précédentes. Les incréments par 2 du curseur de simulation plutôt que par 1 correspondent simplement au déroulage des constructions auxquelles la boucle est réduite pas à pas, et n'introduisent pas de difficultés.

La seule contrainte d'un type différent est l'établissement de la pré-condition de *`divg`* dans le cas de gestion des états limites. Cependant, comme l'ordre de progression choisi est le même que celui de la récurrence et que l'invariant est de même nature que les pré-conditions des contrats de fonctions, les états limites suivent la même structure que ceux de la récurrence. En particulier, nous pouvons reprendre le même raisonnement pour démontrer que $\mathcal{T}$ est infinie et que la borne supérieure des canaux d'entrée/sortie associée est bien l'état courant du jeu.

□

6.4.5 Des expressions aux fonctions

Comme pour la simulation en arrière, il ne nous reste plus qu'à établir la correction locale de la compilation des fonctions pour terminer la preuve.

Lemme 6.50. *Les fonctions sont compilées de manière localement correctes. En particulier, le résultat de simulation en avant énoncé par le théorème 6.8 est vérifié.*

Démonstration. Pour les fonctions prédéfinies, la preuve de correction se réduit comme pour la simulation en arrière à un problème de preuve de programme, pour lequel nous pouvons exploiter nos transformateurs. L'établissement des contrats souhaités est extrêmement simple dans le cas où la machine virtuelle admet une instruction proche de l'opérateur prédéfini.

Pour les fonctions définies par le programme, la correction locale de la compilation se déduit de la compilation correcte des expressions. Comme pour la preuve de la simulation en arrière, nous utilisons un transformateur nommé pour passer de l'une à l'autre, donné dans la figure 6.18.

code source :

$$f[L_0]((x_i)_{0\leq i<m}) = e$$

code compilé :

```
START_f :  nop
           [[e]]_{m+1,[(x_i→i+1)_{0≤i<m}],[L_0→(1,EXIT)]}
           cut m
  EXIT :   swap
           jmp
```

transformateur nommé associé :

$$\left[\begin{array}{l}\mathcal{I}_{\mathtt{asm},\exists},\ \mathtt{all\text{-}fun}\\ c_{glob}, \mathtt{pc}_R, \mathcal{T}, t_{\mathtt{src}}\end{array}\right] \Rightarrow \left[\ t'_{\mathtt{src}}\ \right]$$

nop
continuation *divg*()
  progression
    let $t \leftarrow (t$ si now $= (s,t))$
    continuation *ret*$(L, t'_{\mathtt{src}})$
      intro *all-fun*$(g, \mathtt{pc}_R, t_{\mathtt{src}})$: $\mathcal{C}_{rec}(F, c_{glob}, \{(t_0,t) \mid t_0 \geq t_{\mathtt{src}}\})$
      by
        switch
        case $(t' < t$ si now $= (s,t'))$:
          ret
          choose $t'_{\mathtt{src}}$ such that $\bot$
        default :
          all-fun
        end-switch
      in
        $W_e^{+1}[t_{\mathtt{src}} \leftarrow t_{\mathtt{src}} + 1]$
        cut$[n \leftarrow m]$
        let $t'_{\mathtt{src}} \leftarrow t'_{\mathtt{src}}$
      end-intro
      let $t'_{\mathtt{src}} \leftarrow t'_{\mathtt{src}}, L \leftarrow L_0$
    end-continuation
    let $t'_{\mathtt{src}} \leftarrow t'_{\mathtt{src}} + 1$
  end-progression
  swap
  jmp
  let $t'_{\mathtt{src}} \leftarrow t'_{\mathtt{src}}$
end-continuation

FIGURE 6.18 – Code compilé et transformateur pour une déclaration de fonction

Comme pour le cas de la simulation arrière, ce transformateur crée et met en forme les hypothèses additionnelles correspondant à la compilation d'une expression. Nous utilisons par ailleurs la même méthode pour éliminer la contrainte de progression du compteur temporel des hypothèses récursives. Nous démontrons alors la condition de vérification de façon identique aux précédentes.

□

Chapitre 7

Conclusion

7.1 Résumé des contributions

Nous avons présenté plusieurs méthodes pour réduire l'effort nécessaire pour spécifier des comportements attendus complexes, ainsi que pour démontrer qu'un programme respecte ces spécifications. Dans le chapitre 2, nous avons présenté plusieurs fonctionnalités de Why3 que nous avons introduites pour réduire la quantité de preuve manuelle, notamment les indicateurs de coupure et la transformation de calcul. Dans le chapitre 3, nous nous sommes intéressés au problème de la preuve d'absence de débordements arithmétiques. Nous avons proposé une méthodologie qui rend la preuve de cette absence triviale pour une classe importante d'utilisation des entiers, par le biais d'un argument temporel. Enfin, dans les chapitres 4 et 5, nous avons développé un système de preuve de programmes générique inspiré de la logique de Hoare et du calcul de plus faible pré-condition, dans le but d'obtenir une bibliothèque de spécification et de preuve pour les programmes générateurs de code. Ce système de preuve permet de prendre en compte de manière fine les comportements d'un programme qui peut ne pas terminer. De plus, ce système de preuve n'est pas dirigé par la syntaxe, ce qui permet son application à des programmes non structurés. Nous avons montré dans le chapitre 6 comment appliquer ce système à la preuve d'un schéma de compilation conséquent.

Nous avons mené ce travail au travers de multiples études de cas. Nous listons ici les plus significatives.

— Une étude sur la vérification de programmes utilisant des types de données avec lieurs en Why3, laquelle a donné lieu à la vérification d'un petit démonstrateur automatique basé sur la méthode des tableaux [28]. Cet exemple nous a convaincu qu'il était possible d'obtenir une modélisation

raisonnable des structures avec lieurs en Why3, mais que des efforts restaient à faire pour vérifier efficacement des programmes concrets qui manipulent ces structures. En effet, ces preuves nous ont demandé d'appliquer explicitement des lemmes de commutation un grand nombre de fois. Les sources Why3 correspondant à cet exemple peuvent être trouvées à l'adresse http://toccata.lri.fr/gallery/prover.en.html.

— La vérification de plusieurs structures de données basées sur les arbres AVL (http://toccata.lri.fr/gallery/avl.fr.html). Ces structures de données sont obtenues par instanciation d'une structure générique d'arbre équilibré, ce qui nous a permis de rendre la spécification plus modulaire. Cet exemple, avec celui du démonstrateur qui utilise des indices de de Bruijn [36], fait partie des sources d'inspiration pour la méthode proposée dans le chapitre 3.

— La preuve d'un petit compilateur présentée en section 2.5 (http://toccata.lri.fr/gallery/double_wp.fr.html), effectuée avec Léon Gondelman. Ce travail nous a conduit à introduire la transformation d'induction présentée dans la section 2.2, et à raffiner la transformation de calcul présentée dans la section 2.4. Ce travail nous a également mené au formalisme des jeux pour la vérification (cf. chapitre 4 et 5), de manière à pouvoir appliquer la méthode dans un cadre plus ambitieux (cf. chapitre 6). Nous avons également réutilisé la méthode sous-jacente pour vérifier un petit compilateur d'expressions qui minimise le nombre de registres utilisés (http://toccata.lri.fr/gallery/register_allocation.fr.html).

— La vérification d'un algorithme parallèle de calcul de plus grand commun diviseur (*PGCD*) (http://toccata.lri.fr/gallery/verifythis_2015_parallel_gcd.en.html), effectuée avec Léon Gondelman. La vérification de cet algorithme a été posée comme second problème de la compétition VerifyThis 2015 [1]. Nous avons effectué cette preuve en encodant la structure de l'algorithme comme une machine à états. Le point le plus intéressant est que nous avons vérifié la terminaison sous une hypothèse d'équité. Celle-ci est formalisée en exprimant que l'exécution ininterrompue d'un fil d'exécution doit faire décroître un compteur auxiliaire, qui est modifié de manière arbitraire lors d'un changement de contexte.

— La preuve de l'algorithme de multiplication de matrices de Strassen, présentée dans la section 2.6 (http://toccata.lri.fr/gallery/verifythis_2016_matrix_multiplication.en.html), effectuée avec Léon Gondelman et Mário Pereira. Cet exemple, proposé comme premier problème de la compétition VerifyThis 2016 [2], nous a amené à effectuer une preuve par réflexion grâce à la transformation de calcul de Why3.

— La preuve d'un algorithme de traversée d'arbre binaire (http://toccata.lri.fr/gallery/verifythis_2016_tree_traversal.fr.html), correspondant au second problème de la compétition VerifyThis 2016. Nous avons effectué cette preuve par le biais d'une transformation du programme en style récursif, ce qui simplifie considérablement la preuve. Cette preuve nous a convaincu de la supériorité pratique d'une règle de preuve récursive vis-à-

1. http://etaps2015.verifythis.org/
2. http://etaps2016.verifythis.org/

vis d'une règle de preuve itérative, d'autant plus que nous avons pu réutiliser la même méthodologie pour vérifier aisément l'algorithme de Schorr-Waite (http://toccata.lri.fr/gallery/schorr_waite_via_recursion.en.html). Cela nous a également fourni de nouveaux exemples d'applications possibles du formalisme des jeux pour la vérification de programmes (cf. section 5.4.3).

— La preuve d'un algorithme de résolution de chemins dans un système de fichiers [19], effectuée avec Claude Marché et Ran Chen. Nous avons contribué à cette étude en proposant un argument simple pour la complétude de l'algorithme, basé sur l'étude des tailles potentielles des témoins de résolution. Le code associé à cet exemple est disponible à l'adresse http://toccata.lri.fr/gallery/path_resolution.fr.html.

— Le développement effectué pour mécaniser la formalisation des jeux (cf. section 4.6). Ce développement montre qu'il est possible d'utiliser Why3 comme un assistant de preuve, et repose de façon critique sur l'utilisation des indicateurs de coupure que nous avons ajoutés à Why3 (cf. section 2.3). Le code associé est disponible à l'adresse http://toccata.lri.fr/gallery/hoare_logic_and_games.en.html.

7.2 Travaux connexes autour des jeux

Notre formalisation à base de jeux laisse naturellement penser à un lien avec la sémantique des jeux [50], d'autant plus qu'il s'agit d'un outil utilisé pour la compilation vers des circuits matériels, via la *géométrie de la synthèse* [47]. Toutefois, l'usage des jeux est différent. Dans la sémantique des jeux, les jeux (appelés « arènes » dans ce domaine) sont utilisés pour représenter les types des objets, tandis que les stratégies gagnantes représentent les programmes de ces types. Dans notre cas, les jeux servent à représenter la sémantique à petits pas des programmes, et la notion de stratégie gagnante correspond aux témoins des propriétés que l'on cherche à démontrer (e.g. accessibilité/vivacité). Dans le cadre de la géométrie de la synthèse, cette différence est visible dans le fait que la sémantique des jeux est directement utilisée pour dériver le schéma de compilation vers des circuits, tandis que dans notre cas les jeux sont principalement un outil de preuve.

La modélisation des programmes par des jeux n'est pas nouvelle. Elle sert en général à représenter des programmes contenant deux types de non-déterminisme, respectivement appelés angélique (existentiel) et démoniaque (universel). Dans ce cadre, la sémantique de plus faible pré-condition est un outil standard car directement adaptable au cas des jeux [6]. Une logique de Hoare a par ailleurs été proposée [62] pour la preuve de programme contenant les deux types de non-déterminisme. Cette logique est basée sur la notion d'*implication de Hoare*, qui est similaire à nos « énoncés ». Cependant, les systèmes de preuves mentionnés restent dirigés par la syntaxe, tandis que notre formalisme permet l'application libre des règles de preuve. De plus, ces systèmes ne sont applicables qu'à la correction partielle ou à la correction totale, tandis que nous avons la possibilité de démontrer des propriétés plus fines sur le comportement des programmes qui ne terminent pas. Notons par ailleurs que ces travaux ne considèrent que des stratégies sans mémoire, qui suffisent dans le cas de la correction partielle ou totale.

7.3 Perspectives

Dans un futur immédiat, la suite de ce travail est l'estimation pratique des gains obtenus par l'usage des jeux et du système de preuve proposé dans le chapitre 5 pour les garanties. Pour cela, nous avons l'intention de mécaniser la preuve présentée dans le chapitre 6.

Dans un avenir plus lointain, nous pouvons envisager un certain nombre de pistes à suivre. Nous avons mentionné dans le chapitre 3 plusieurs extensions possibles de notre travail sur la preuve d'absence de débordements arithmétiques, comme l'application à d'autres schémas de croissance des entiers. Nous pourrions donc mener des expériences pour vérifier que la méthode est applicable sans difficultés majeures à ces schémas. Nous pourrions également mener une étude théorique visant à formaliser la connexion que nous avons remarquée entre les entiers restreints et la notion de « crédit temps ».

Une autre piste de recherche intéressante serait la vérification de programmes conçus pour être performants, aspect auquel nous nous sommes assez peu intéressé dans cette thèse. La vérification de tels programmes peut être assez difficile, potentiellement à cause de l'utilisation de structure de données bas niveau, d'optimisation de cas particuliers, ou encore d'un flot de contrôle modifié. Notre travail sur les entiers restreints va dans ce sens, en facilitant les preuves d'absence de débordement arithmétique, mais ne couvre qu'une fraction des difficultés que peut poser ce genre de vérification. Comme un programme performant est généralement conçu en optimisant un programme de haut niveau, les méthodes à base de raffinement et de transformation de programmes nous semblent être les pistes les plus prometteuses, car ces méthodes permettent de réduire la vérification à celle d'un programme de plus haut niveau, plus abstrait et mieux structuré. Notre étude sur la preuve de programmes de nature dérécursifiée (cf. section 5.4.3) constitue un argument dans ce sens, puisque nous obtenons une réduction notable de l'effort de vérification en remplaçant un programme itératif par un programme récursif, dont le flot de contrôle est plus adapté à la vérification.

Une autre possibilité est d'étendre le travail sur les jeux. D'un point de vue théorique, nous avons laissé ouvert le problème de l'existence systématique de stratégies gagnantes sans mémoire (cf. remarque 4.6). En effet, si celles-ci suffisent dans le cadre de la correction partielle ou totale, nous n'avons pas réussi à déterminer si c'est toujours le cas dans le cadre de propriétés plus fine sur les exécutions qui ne terminent pas.

D'un point de vue plus pratique, nous souhaiterions améliorer la logique proposée pour les garanties, présentée dans la section 5.2. Cela peut se faire de plusieurs manières. Comme cette logique est basée sur la logique de Hoare, nous pouvons raisonnablement envisager de réutiliser ce travail pour obtenir une variante de la logique de séparation dans le cadre des jeux, par exemple en ajoutant une structure d'algèbre de séparation [16]. Nous pourrions également étudier les possibilités d'extension de notre logique de garanties à l'ordre supérieur. Les tentatives effectuées à cet égard durant cette thèse ont jusqu'ici été infructueuses, principalement à cause de la règle de récurrence. L'extension naturelle de cette règle à l'ordre supérieur permettrait en effet d'utiliser des hypothèses récursives pour établir la pré-condition d'un « appel récursif », ce qui en termes de programmes revient à cacher un appel récursif dans une clôture puis à passer cette clôture en paramètre d'un autre

appel récursif. Cela complexifie significativement l'analyse du comportement d'une structure récursive, surtout au niveau des situations limites.

Enfin, une dernière possibilité est de continuer à améliorer l'outil Why3. Dans le chapitre 2, nous avons observé que l'usage contrôlé d'une transformation de calcul pouvait simplifier considérablement certaines preuves. En fait, notre utilisation de la transformation de calcul revient à configurer la génération de conditions de vérification pour réduire ces conditions à une forme plus adéquate. D'une certaine manière nous pouvons également voir cette méthode comme une technique pour fournir automatiquement une partie de la preuve, et donc comme un complément aux indicateurs de coupure. Cependant, le processus est très ad-hoc, et implique un effort conséquent de la part de l'utilisateur pour être appliqué en pratique. Il serait intéressant d'essayer de réduire ce coût en intégrant la configuration de la génération de condition de vérification plus profondément dans le langage de spécification de Why3. En d'autre termes, nous aimerions étendre le langage de spécification de Why3 de manière à pouvoir traiter mécaniquement des propriétés plus complexes, et à pouvoir paramétrer finement l'automatisation du processus de preuve. Nous considérons qu'une telle extension du langage de spécification est un passage obligé pour pouvoir vérifier efficacement des outils complexes, comme par exemple Why3 lui-même.

Remarquons que le sujet des extensions possibles du langage de spécification recoupe en partie le travail effectué sur les jeux. En effet, nous dupliquons une grande partie de ce qui est déjà fourni par Why3, à savoir la logique de Hoare et le calcul de plus faible pré-condition. Nous aimerions pouvoir éviter cette duplication, en réutilisant autant que possible ce qui est déjà fourni par l'outil. Pour cela, il faudrait au minimum embarquer directement dans le langage de spécification une notion générique de triplets de Hoare, ce qui serait une extension conséquente.

Bibliographie

[1] Jean-Raymond Abrial. *The B-Book, assigning programs to meaning.* Cambridge University Press, 1996.

[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin : an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6) :447–466, 2010.

[3] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics–Doklady*, 3(5) :1259–1263, September 1962.

[4] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

[5] Robert Atkey. Amortised Resource Analysis with Separation Logic. *Logical Methods in Computer Science*, Volume 7, Issue 2, June 2011.

[6] Ralph-Johan Back and Joakim von Wright. *Refinement calculus - a systematic introduction.* Undergraduate texts in computer science. Springer, 1999.

[7] Ralph-Johan J. Back. *On the correctness of refinement in program development.* PhD thesis, Department of Computer Science, University of Helsinki, 1978.

[8] Mike Barnett, Robert DeLine, Bart Jacobs, Bor-Yuh Evan Chang, and K. Rustan M. Leino. Boogie : A Modular Reusable Verifier for Object-Oriented Programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects : 4th International Symposium*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, 2005.

[9] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System : An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[10] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development.* Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[12] Joshua Bloch. Nearly all binary searches and mergesorts are broken, 2006. http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html.

[13] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover, 2008. http://alt-ergo.lri.fr/.

[14] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3 : Shepherd your herd of provers. In *Boogie*, pages 53–64, August 2011. https://hal.inria.fr/hal-00790310.

[15] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6) :709–727, 2015. See also http://toccata.lri.fr/gallery/fm2012comp.en.html.

[16] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Proceedings of the 22Nd Annual IEEE Symposium on Logic in Computer Science*, LICS '07, pages 366–378, Washington, DC, USA, 2007. IEEE Computer Society.

[17] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP)*, pages 418–430, Tokyo, Japan, September 2011. ACM.

[18] Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *Journal of Automated Reasoning*, September 2017.

[19] Ran Chen, Martin Clochard, and Claude Marché. A formally proved, complete algorithm for path resolution with symbolic links. *Journal of Formalized Reasoning*, 10(1) :51–66, 2017.

[20] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, April 1986.

[21] W Clinger, D P Friedman, and M Wand. Algebraic methods in semantics. chapter A Scheme for a Higher-level Semantic Algebra, pages 237–250. Cambridge University Press, New York, NY, USA, 1986.

[22] Martin Clochard. Automatically verified implementation of data structures based on AVL trees. In Giannakopoulou and Kroening [48], pages 167–180.

[23] Martin Clochard. Preuves taillées en biseau. In Sylvie Boldo and Julien Signoles, editors, *Vingt-huitièmes Journées Francophones des Langages Applicatifs*, Gourette, France, January 2017.

[24] Martin Clochard, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Formalizing semantics with an automatic program verifier. In Giannakopoulou and Kroening [48], pages 37–51.

[25] Martin Clochard, Jean-Christophe Filliâtre, and Andrei Paskevich. How to avoid proving the absence of integer overflows. In Arie Gurfinkel and Sanjit A. Seshia, editors, *7th Working Conference on Verified Software : Theories, Tools and Experiments (VSTTE)*, volume 9593 of *Lecture Notes in Computer Science*, pages 94–109, San Francisco, California, USA, July 2015. Springer.

[26] Martin Clochard and Léon Gondelman. Double WP : vers une preuve automatique d'un compilateur. In *Vingt-sixièmes Journées Francophones des Langages Applicatifs*, Val d'Ajol, France, January 2015.

[27] Martin Clochard, Léon Gondelman, and Mário Pereira. The Matrix reproved. *Journal of Automated Reasoning*, 2018.

[28] Martin Clochard, Claude Marché, and Andrei Paskevich. Verified programs with binders. In *Programming Languages meets Program Verification (PLPV)*. ACM Press, 2014.

[29] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC : A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes in Computer Science*. Springer, 2009.

[30] David R. Cok. OpenJML : Software verification for Java 7 using JML, OpenJDK, and Eclipse. In Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, editors, *Proceedings 1st Workshop on Formal Integrated Development Environment*, volume 149 of *EPTCS*, pages 79–92, 2014.

[31] David R. Cok and Joseph Kiniry. ESC/Java2 : Uniting ESC/Java and JML. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.

[32] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1) :70–90, 1978.

[33] Patrick Cousot. Methods and logics for proving programs. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 841–993. North-Holland, 1990.

[34] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL '79 : Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, New York, NY, USA, 1979. ACM.

[35] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C : A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, number 7504 in Lecture Notes in Computer Science, pages 233–247. Springer, 2012.

[36] Nikolas G. de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proc. of the Koninklijke Nederlands Akademie*, 75(5) :380–392, 1972.

[37] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[38] Maxime Dénès, Anders Mörtberg, and Vincent Siles. A refinement-based approach to computational algebra in Coq. In Lennart Beringer and Amy Felty, editors, *ITP - 3rd International Conference on Interactive Theorem Proving - 2012*, volume 7406 of *Lecture Notes In Computer Science*, pages 83–98, Princeton, United States, 2012. Springer, Springer.

[39] Edsger W. Dijkstra. Notes on structured programming. In O. Dahl, E. Dijkstra, and C. Hoare, editors, *Structured programming*. Academic Press, 1971.

[40] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18 :453–457, August 1975.

[41] Matthias Felleisen and Daniel P. Friedman. Control Operators, the SECD-machine, and the λ-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, pages 193–219, August 1986.

[42] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. In Armin Biere and Roderick Bloem, editors, *26th International Conference on Computer Aided Verification*, volume 8859 of *Lecture Notes in Computer Science*, pages 1–16, Vienna, Austria, July 2014. Springer.

[43] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[44] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In *ESOP* [43], pages 125–128.

[45] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[46] G. Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift*, (39) :176–210, 405–431, 1934.

[47] Dan R. Ghica. Geometry of synthesis : A structured approach to vlsi design. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 363–375, New York, NY, USA, 2007. ACM.

[48] Dimitra Giannakopoulou and Daniel Kroening, editors. volume 8471 of *Lecture Notes in Computer Science*, Vienna, Austria, July 2014. Springer.

[49] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580 and 583, October 1969.

[50] J.M.E. Hyland and C.-H.L. Ong. On full abstraction for pcf. *Inf. Comput.*, 163(2) :285–408, December 2000.

[51] Bart Jacobs. Partial Solutions to Verifythis 2016 Challenges 2 and 3 with VeriFast. In *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs*, FTfJP'16, pages 7 :1–7 :6, New York, NY, USA, 2016. ACM.

[52] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast : A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[53] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris : Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 637–650, New York, NY, USA, 2015. ACM.

[54] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML : A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191, New York, NY, USA, 2014. ACM.

[55] P. J. Landin. Correspondence between algol 60 and church's lambda-notation : Part i. *Commun. ACM*, 8(2) :89–101, February 1965.

[56] K. Rustan M. Leino. Dafny : An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[57] K. Rustan M. Leino. Automating induction with an SMT solver. In *Proc. 13th Int. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*, Philadelphia, PA, 2012.

[58] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer, 2009.

[59] K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2007.

[60] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4) :363–446, 2009.

[61] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6) :1811–1841, November 1994.

[62] Konstantinos Mamouras. Synthesis of strategies using the hoare logic of angelic and demonic nondeterminism. *Logical Methods in Computer Science*, 12(3), 2016.

[63] Z. Manna and J. McCarthy. Properties of programs and partial function logic. In *Machine Intelligence*, volume 5, 1970.

[64] Claude Marché. Jessie : an intermediate language for Java and C verification. In *Programming Languages meets Program Verification (PLPV)*, pages 1–2, Freiburg, Germany, 2007. ACM Press.

[65] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2) :89–106, 2004. http://krakatoa.lri.fr.

[66] John McCarthy. A basis for a mathematical theory of computation, preliminary report. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '61 (Western), pages 225–238, New York, NY, USA, 1961. ACM.

[67] John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015.

[68] P. Müller, M. Schwerhoff, and A. J. Summers. Viper : A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[69] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot : Reasoning with the awkward squad. In *Proceedings of ICFP'08*, 2008.

[70] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[71] Francisco Palomo-Lozano, Inmaculada Medina-Bulo, and J.A. Alonso-Jiménez. Certification of matrix multiplication algorithms : Strassen's algorithm in ACL2. In *Supplemental Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, pages 283–298, Edinburgh (Scotland), 2001.

[72] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[73] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[74] J. C. Reynolds. Separation logic : a logic for shared mutable data structures. In *17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.

[75] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM.

[76] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10 :501–506, 1967.

[77] Stephan Schulz. System description : E 0.81. In David A. Basin and Michaël Rusinowitch, editors, *Second International Joint Conference on Automated Reasoning*, volume 3097 of *Lecture Notes in Computer Science*, pages 223–228. Springer, 2004.

[78] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, January 2016.

[79] René Thiemann and Akihisa Yamada. Matrices, Jordan normal forms, and spectral radius theory. *Archive of Formal Proofs*, 2015, 2015.

[80] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 97–108, Nice, France, January 2007.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Méthodes et outils pour la spécification et la preuve de propriétés difficiles de programmes séquentiels
Mots clés : Vérification déductive/Calcul symbolique/Preuve formelle
Résumé : Cette thèse se positionne dans le domaine de la vérification déductive de programmes, qui consiste à transformer une propriété à vérifier sur un programme en un énoncé logique, pour ensuite démontrer cet énoncé. La vérification effective d'un programme peut poser de nombreuses difficultés pratiques. En fait, les concepts mis en jeu derrière le programme peuvent suffire à faire obstacle à la vérification. En effet, certains programmes peuvent être assez courts et n'utiliser que des constructions simples, et pourtant s'avérer très difficiles à vérifier. Cela nous amène à la question suivante : dans le contexte d'un environnement de vérification déductive de programmes basé sur les démonstrateurs automatiques, quelles méthodes appliquer pour réduire l'effort nécessaire à la fois pour spécifier des comportements attendus complexes, ainsi que pour démontrer qu'un programme respecte ces comportements attendus ? Pour mener notre étude, nous nous sommes placés dans le cadre de l'environnement de vérification déductive de programmes Why3. La vérification de programmes en Why3 est basée sur la génération de conditions de vérification, et l'usage de démonstrateurs externes pour les prouver, que ces démonstrateurs soient automatiques ou interactifs. Nous avons développé plusieurs méthodes, certaines générales et d'autres spécifiques à des classes de programmes, pour réduire l'effort manuel. Nos contributions sont les suivantes. Tout d'abord, nous ajoutons des fonctionnalités à Why3 pour assister le processus de vérification, notamment un mécanisme léger de preuve déclarative basé sur la notion d'indicateurs de coupures. Ensuite, nous présentons une méthode de vérification d'absence de débordement arithmétique pour une classe d'utilisation des entiers difficile à traiter par les méthodes standards. Enfin, nous nous intéressons au développement d'une bibliothèque générique pour la spécification et la preuve de programmes générateurs de code.

Title : Methods and tools for specification and proof of difficult properties of sequential programs
Keywords : Deductive verification/Symbolic computations/Formal proof
Abstract : This thesis is set in the domain of deductive verification of programs, which consists of transforming a property to be verified about a program into a logical statement, and then proving this statement. Effective verification of a program can pose many practical difficulties. In fact, the concepts behind the program may be sufficient to impede verification. Indeed, some programs can be quite short and use only simple constructions, and yet prove very difficult to verify. This leads us to the following question : in the context of a deductive program verification environment based on automatic provers, what methods can be applied to reduce the effort required both to specify complex behaviors, as well as to prove that a program respects these expected behaviors ? To carry out our study, we placed ourselves in the context of the deductive verification environment of programs Why3. The verification of programs in Why3 is based on the generation of verification conditions, and the use of external provers to prove them, whether these provers are automatic or interactive. We have developed several methods, some general and others specific to some program classes, to reduce manual effort. Our contributions are as follows. First, we add features to Why3 to assist the verification process, including a lightweight declarative proof mechanism based on the notion of cut indicators. Then we present a method for checking the absence of arithmetic overflow, for use cases which are difficult to process by standard methods. Finally, we are interested in the development of a generic library for the specification and proof of code generating programs.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France