THÈSE de DOCTORAT
du
Conservatoire National des Arts et Métiers

École Doctorale d'Informatique, Télécommunication et Électronique

pour obtenir le grade de

DOCTEUR es SCIENCES

Spécialité

INFORMATIQUE

présentée par

Matthieu CARLIER

Test automatique de propriétés dans un atelier de développement de logiciels sûrs

Soutenue le 7 octobre 2009,

Devant le jury composé de :

Ioannis Parissis	Rapporteur
Burkhart Wolff	Rapporteur
Thérèse Hardin	Examinatrice
Pascale Le Gall	Examinatrice
Arnaud Gotlieb	Examinateur
Catherine Dubois	Directrice de Thèse

Remerciements

Tout d'abord, je tiens à remercier Catherine Dubois pour avoir accepté de diriger ma thèse et m'avoir ainsi permis d'arriver jusqu'à l'écriture de ce manuscrit. Catherine possède les qualités humaines et pédagogiques qui font d'elle une bonne directrice de thèse. Je remercie aussi Burkhart Wolff et Ioannis Parissis d'avoir accepté de rapporter cette thèse et de leurs remarques qui ont permis d'améliorer le manuscrit. Je remercie également Thérèse Hardin et Pascale Le Gall d'avoir accepté de faire partie du jury ainsi qu'Arnaud Gotlieb qui m'a de plus apporté son aide et son soutien dans le développement de mes travaux sur la programmation par contraintes.

Mes travaux ont été initiés grâce au projet FoCal. Je remercie les différents acteurs de FoCal pour leur aide dans le développement de ce projet. Virgile Prévosto pour la première version *focc* du compilateur, François Pessaux pour la deuxième version *focalize* et plus généralement tous les contributeurs de FoCal avec qui j'ai interagi à des degrés divers.

J'ai fait partie pendant ces années de l'équipe CPR du laboratoire CÉDRIC. Je remercie tous les membres de CPR qui m'ont accueilli chaleureusement et avec qui j'ai eu des discussions variées. Je pense bien sûr à ceux que j'ai croisés réguliérement comme Sandrine Blazy, Pierre Courtieu, David Delahaye, Julien Forest, Olivier Pons, Renaud Rioboo et Xavier Urbain. Mais je n'oublie pas Véronique Viguié-Donzeau-Gouge qui s'est éloignée de l'équipe au moment de mon arrivée et que j'ai peu connue. J'aurais aimé qu'elle fasse partie du Jury mais elle n'a malheureusement pas pu se libérer.

J'ai passé mes années de thèse au sein de l'ENSIIE. Durant cette période j'ai pu côtoyer différents enseignants-chercheurs avec lesquels j'ai partagé de nombreuses discussions durant les instants de détente comme les déjeuners et les pauses café : Gérard Berthelot, Guillaume Bouyer Nicolas Brunel, Louis Gacogne, Yacine Ghamri-Doudane, Brigitte Grau, Mireille Jouve, Vathana Ly Vath, David Roussel, Françoise Vuillermet, Corine Waroquiers. Je tiens à remercier particulièrement Anne-Laure Ligozat. Son arrivée à l'école a clairement marqué un avant et un après dans ma thèse.

Je remercie les membres de l'école qui m'ont aidé à résoudre les problèmes au quotidien. Olivier Hubert, Jean-Luc Kors et Gaël Thomas qui m'ont dépanné lorsque j'avais des soucis avec les machines et le réseau de l'école. Éric Lejeune pour les problèmes d'emploi du temps. Anne-Marie Pavageau qui connait les rouages de l'administration.

J'ai croisé ces dernières années plusieurs doctorants qui, pour la plupart, ont soutenu leur thèse. Parmi eux, je me rappelle particulièrement de Louis Mandel qui, par l'intermédiaire de mon stage de 2^{e} année de Magistère, m'a fait découvrir la programmation réactive. Pierre Rousseau qui a effectué sa thèse dans un cadre difficile et reste un modèle pour moi. Hakim Belhaouari avec qui j'ai fait connaissance en Master et qui a toujours été présent pendant ces années et m'a remonté le moral plus d'une fois. Enfin Zaynah Dargaye que j'ai d'abord rencontrée pendant mon année de Master mais que j'ai réellement découverte pendant sa dernière année de thèse et avec qui j'ai partagé plusieurs discussions qui m'ont beaucoup apporté. Ces rencontres m'ont marqué et j'ai une pensée pour ces personnes.

Pendant mes années de thèse, j'ai partagé mon bureau avec des personnes qui m'ont soutenu

sans forcément s'en rendre compte. Je remercie Frédéric Gervais qui m'a accueilli durant les premières années. Sa présence et les discussions que j'ai eu avec lui ont été précieuses pour moi. Je remercie Benoît Robillard et William Bartlett qui m'ont rejoint par la suite et qui ont participé à une bonne ambiance. Je souhaite la bienvenue au nouvel arrivé, Pierre-Nicolas Tollitte, et lui souhaite de réussir dans son projet de thèse.

Enfin, je désire également remercier les différents élèves de l'école avec qui j'ai lié amitié : aussi bien les stagiaires qui ont permis pendant les mois qui suivent la fin des cours que l'école ne soit pas entièrement vide et triste, que les élèves que j'ai côtoyés tout au long de l'année.

Sommaire

Introduction

L'informatique est désormais partout : dans les avions, dans les voitures, dans les métros, dans les habitations, dans les téléphones portables, dans les cartables, dans les télévisions, dans les loisirs etc. Une multitude d'applications logicielles contrôle, surveille, gère, assiste, régule, planifie, répartit etc. Certains systèmes informatisés, dits critiques, ont des enjeux importants en termes financiers, économiques, environnementaux et humains, par exemple les systèmes de commande dans les transports ou les systèmes de surveillance de centrales nucléaires.

En fait plus les dommages susceptibles d'être causés par une panne logicielle ou matérielles sont élevés, plus les exigences dans le développement de ces systèmes et applications sont élevés. Nous nous intéresserons ici qu'à l'aspect logiciel et par conséquent aux méthodes, techniques et outils permettant de fournir des logiciels de qualité, avec un fort niveau de confiance. Pour des systèmes critiques ou embarqués, on peut être amené à faire valider le logiciel, du point de la sûreté de fonctionnement ou de la sécurité, par une source extérieure. On parle alors de certification de systèmes.

Certification

La certification par un organisme extérieur consiste à faire vérifier qu'un logiciel est conforme à des normes de qualité.

La norme DO-178B [RTC92] est une norme de qualité internationale qui est imposée à tout logiciel qui est utilisé dans le cadre de l'aéronautique. Celle-ci définit 5 niveaux de criticité, de A à E. Le niveau A, maximal, est imposé aux logiciels pour lesquels un dysfonctionnement provoque un crash de l'appareil en vol. Le niveau E, quant à lui, n'impose aucune contrainte sur le logiciel et chaque niveau cumule les contraintes de développement imposées aux niveaux de moindre exigence.

De même la norme des critères communs [Com06] définit des niveaux d'assurance de l'évaluation (EAL) pour la sécurité des systèmes. Faire certifier un logiciel avec les critères communs ou la DO-178B requiert de fournir à l'organisme de certification tous les documents qui montrent que l'ensemble des exigences de la méthode de certification a été appliqué correctement. De manière générale, pour atteindre de hauts niveaux de certification, les méthodes formelles sont requises : il est alors nécessaire de spécifier les propriétés attendues et/ou définir un modèle formel du système, de fournir des preuves formelles mécanisées, de mettre en oeuvre des techniques de test, ou de concevoir des analyses statiques permettant de garantir les propriétés requises. Il faut ajouter que pour chacune des normes de certification (comme la DO-178B ou les critères communs), tous les niveaux d'exigence imposent que les logiciels soient testés méthodologiquement. Le test est requis même lorsque le niveau de sécurité impose que le logiciel soit vérifié formellement avec des méthodes de preuve. Ainsi, si on souhaite obtenir un logiciel avec un haut

niveau de qualité, il faut assurer que le logiciel ait été testé.

Pour vérifier un logiciel il a y donc deux principales méthodes : la preuve et le test.

Preuve

La preuve permet de montrer que le programme implante effectivement les exigences de sa spécification. Prouver un programme nécessite de disposer d'une spécification formelle des fonctionnalités du programme, de la sémantique formelle du langage de programmation dans lequel est écrit le programme, et des outils de preuve de formules logiques. Par exemple, la preuve de programmes impératifs, utilise en général l'approche de la logique de Hoare : les spécifications prennent alors la forme de pré et post condition, des obligations de preuves sont générées et doivent être démontrées à l'aide d'un outil de démonstration automatique ou d'un assistant à la preuve interactive, le cas échéant.

Le développement d'un programme avec un outil de preuve passe généralement par des cycles qui alternent la preuve du programme et la correction de l'implantation. La phase de preuve est généralement fastidieuse, il est admis que pour une ligne d'implantation il faut écrire entre 2 et 4 lignes de preuve. Prouver un programme prend généralement plus de temps que de le développer.

Test logiciel

Le test est la seconde procédure de validation d'un logiciel. Le test est une validation partielle, elle n'assure pas que le logiciel est exempt d'erreur. La philosophie principale du test est que le but du test est de trouver les erreurs et non de montrer l'absence d'erreur dans un logiciel. En effet, utiliser le test pour montrer l'absence d'erreurs dans un logiciel impliquerait de vérifier tous ses comportements, ce qui n'est généralement pas possible. Contrairement aux assistants à la preuve, le test logiciel est très largement utilisé dans l'industrie et constitue le principal moyen de validation de logiciel. Le test constitue d'ailleurs le premier pôle de dépense dans le développement d'un logiciel industriel et est un moyen d'augmenter la fiabilité des logiciels malgré l'accroissement de leur complexité.

Le test et la preuve ne sont pas des techniques de validation contradictoires. Ces deux approches sont complémentaires. Prouver un programme revient à vérifier que le programme se comporte correctement par rapport à un modèle. Ceci suppose que le modèle utilisé décrit correctement l'environnement dans lequel la programme est exécuté. Le lien entre le modèle et l'environnement réel n'est pas établi par la preuve. Dans cette optique le test logiciel est aussi le moyen de valider les hypothèses faites en modélisant ou spécifiant le fonctionnement attendu du programme.

FoCal

Le propos de ce manuscrit est centré sur l'atelier de développement FoCal et son utilisation dans le cadre du développement d'applications de confiance, certifiables à de hauts niveaux de certification. FoCal permet de construire incrémentalement des composants de bibliothèque et prouver leur correction. Un composant, appelé espèce, peut contenir des énoncés de propriétés

spécifiant le composant, des définitions de ses opérations et les preuves que ces opérations satisfont leurs spécifications.

Les espèces sont construites à l'aide de l'héritage multiple, de la liaison retardée. La paramétrisation des composants par les spécifications d'autres composants permet de prendre en compte la compositionnalité. Ces différents traits, qui confèrent au langage un aspect orienté objet, donnent une souplesse dans le développement des applications. Les composants utilisés par la paramétrisation doivent au préalable être soumis à un mécanisme d'encapsulation des composants, ce qui accroît la sûreté des développements et ajoute une connotation modulaire au langage.

Les propriétés qui décrivent le comportement attendu des applications sont des formules du premier ordre. FoCal intègre un prouveur automatique (au premier ordre), Zenon développé par D. Doligez, permettant de décharger automatiquement les preuves qui sont fournies au moyen d'un langage déclaratif.

Le langage permettant d'implanter les composants est un langage fonctionnel, proche du noyau fonctionnel de OCaml. Ce langage, à la sémantique bien définie, aide à la production de code lisible, structuré sous la forme d'un ensemble de fonctions.

Le compilateur FoCal est la composante centrale du développement d'un logiciel. Tout développement, quel que soit son stade d'avancée, peut être soumis au compilateur qui, après une phase de typage et d'analyse de dépendances gérant la liaison retardée, produira du code exécutable (en OCaml) avec les définitions déjà fournies et traduira l'ensemble du développement en un fichier vérifié par Coq. Ainsi, non seulement les preuves de propriétés démontrées par Zenon (éventuellement directement en Coq par des experts) mais aussi l'architecture globale du développement sont formellement vérifiées.

Nous souhaitons donc ajouter à FoCal la possibilité de faire du test logiciels. Ce test doit se faire automatiquement, c'est-à-dire que la sélection et la soumission des jeux de test ne nécessitent pas d'interaction avec l'utilisateur. En plus des raisons propres au développement de logiciel avec un haut niveau de confiance, FoCal permet de développer des logiciels sans en prouver nécessairement la correction. Prouver qu'un logiciel respecte bien sa spécification peut prendre beaucoup de temps. On peut « remplacer » la preuve de correction d'un programme FoCal par un test de celui-ci, la spécification ne servant plus pour prouver mais pour tester.

Prouver un programme dans son intégralité est un travail qui est long et qui demande d'alterner phases de preuve et de mise au point de l'implantation. Il est rare qu'un programme soit correct dès la première écriture. Il est fréquent que la phase de preuve mette en exergue un cas logique qui révèle une erreur dans le programme. L'utilisation du test avant la phase de preuve peut permettre de détecter au plus tôt les comportements non conformes et ainsi de limiter les aller-retours entre la preuve et l'implantation.

Comme nous l'avons vu, les normes de certification demandent en plus des preuves de correction que le logiciel soit testé. Si l'on souhaite que des logiciels développés dans un environnement comme FoCal soient certifiés par des organismes indépendants, il faut fournir des résultats de test. De plus, le test logiciel fait partie intégrante des principaux cycles de développement de logiciels, il paraît ainsi naturel d'intégrer des outils de test dans les assistants à la preuve.

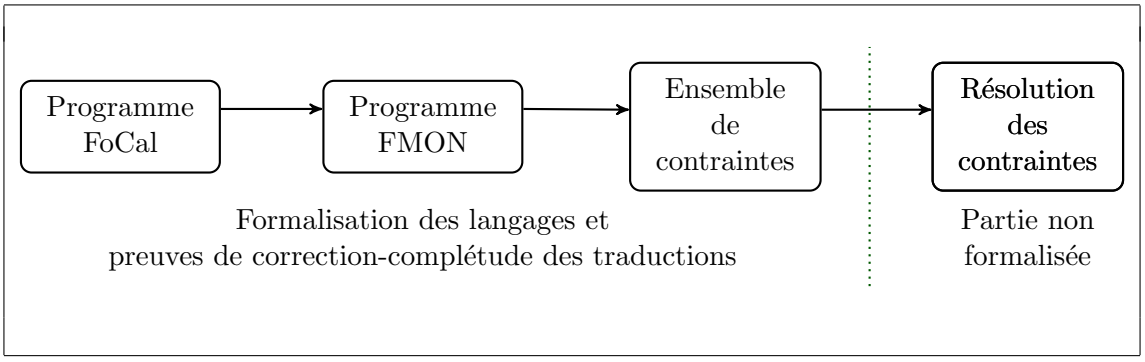

FIGURE 1 – Résumé de ce qui est formalisé

Contributions

Ce manuscrit présente un travail qui regroupe les contributions suivantes :

- *l'intégration d'un outil de test dans un atelier de développement de logiciel certifiable.* En l'occurrence, nous avons intégré notre outil dans l'environnement FoCal. Un développeur FoCal peut ainsi à partir de son programme FoCal et de la spécification qui le compose lancer des procédures de test pour vérifier si oui ou non le programme respecte bien sa spécification. FoCal est un atelier qui permet de prouver que le programme vérifie bien sa spécification, le test peut se faire avant que cette preuve soit réalisée. Dans ce cas, la spécification nous sert de base pour produire nos jeux de test ;

- *la mise en œuvre de test de propriétés.* Il s'agit de partir d'une propriété FoCal et de l'implantation des fonctions spécifiées par la propriété pour générer les jeux de test. La propriété est découpée en un ensemble de propriétés dites élémentaires. Ces dernières correspondent à chacun des comportements de la propriété initiale. Nous testons les propriétés élémentaires séparément et distinguons sur chacune d'elle deux parties : la précondition et la conclusion. Intuitivement, la précondition définit la classe des jeux de test que nous recherchons et la conclusion est un prédicat qui doit être satisfait par l'ensemble des jeux de test. Plus précisément, un jeu de test est une valuation des variables quantifiées qui valide la précondition et la conclusion nous sert à déterminer si l'implantation passe le jeu de test ou non ;

- *l'utilisation de la programmation par contraintes pour synthétiser les jeux de test.* Nous avons plus exactement deux stratégies pour la recherche de jeux de test. La première stratégie consiste à générer des valuations des variables quantifiées jusqu'à l'obtention d'une valuation qui valide la précondition (et qui donne donc un jeu de test). Cette approche n'étant pas efficace dès que peu de valuations satisfont la précondition, nous avons développé la stratégie de recherche se basant sur la programmation par contraintes. Dans cette dernière, nous traduisons la précondition de la propriété en un ensemble de contraintes. L'ensemble des jeux de test recherchés correspond alors à l'ensemble des solutions des contraintes. Cette traduction s'effectue en convertissant la portion du programme FoCal spécifié par la propriété en un ensemble de contraintes. Cette conversion nous donne l'environnement de contrainte. La conversion de la précondition en contraintes donne alors le système à résoudre pour obtenir les jeux de test ;

– *la formalisation et la correction de l'approche à contrainte.* Ceci est résumé dans la figure 1. Nous montrons que la traduction du programme FoCal résulte en un système de contraintes dont chaque solution correspond à une évaluation possible du programme d'origine et inversement, chaque évaluation du programme correspond à une solution particulière du système de contrainte résultant. Nous déléguons la résolution des contraintes à un solveur de contrainte dédié et vérifions *a posteriori* si une solution trouvée est correcte ;

– *la détermination et l'implantation de contraintes particulières pour modéliser les structures propres à FoCal, la conditionnelle et le filtrage par motif ainsi que le domaine des variables qui s'y rapportent les valeurs algébriques.* Pour convertir le programme FoCal en contraintes, il faut faire correspondre chaque construction du langage à une ou plusieurs contraintes. Nous pouvons convertir directement le filtrage par motif et la conditionnelle en des contraintes classiques mais nous sommes guidés par un souci d'efficacité de temps de résolution. Nous avons donc défini un cadre pour créer des contraintes particulières, les *méta-contraintes.*

L'intégralité des travaux que nous venons de détaillé ont été intégrées à FoCal. Nous avons appelé l'outil de test ainsi obtenue FoCalTest. Pour résumer, FoCalTest est capable de tester une classe des propriétés du premier ordre automatiquement. Il génère des jeux de test en utilisant la programmation par contraintes, réalise leur soumission et génère un rapport de test.

Objectif et plan de thèse

Le but de ce manuscrit est de présenter une technique de test dans un atelier de développement de logiciels certifiables. Au premier abord, il parait contradictoire d'intégrer du test dans un environnement dont le but est de s'en passer. Lorsqu'un programme est prouvé correct vis-à-vis de la spécification, il est *a priori* inutile de faire du test. La preuve nous indique que le programme se comporte correctement pour toutes les valeurs d'entrée. Nous pensons que le test et la preuve ne sont pas deux techniques de validation de logiciel ambivalentes. D'une part le programmeur n'a pas l'obligation de prouver la correction des programmes qu'il implante. Dans ce cas, pouvoir tester le programme est un plus car il permet de faire de la détection d'erreur facilement. D'autre part, on peut souhaiter tester un programme avant de rentrer dans la phase de preuve. Dans ce cas, les erreurs détectées par le test, puis corrigées sont autant d'aller-retours en moins entre la preuve et l'implantation.

La thèse est organisée comme suit :

La partie I introduit les concepts utilisés dans le reste du manuscrit. Ainsi dans le chapitre 1 nous ferons un rapide aperçu des différentes techniques de test mises au point. Cette introduction nous permet d'embrayer dans le chapitre 2 sur les différents travaux et efforts qui ont été effectués pour automatiser le test. Nous présentons des outils de test de la littérature. Nous ne faisons pas une présentation exhaustive de l'état actuel des choses mais plutôt un panorama des outils qui présentent des particularités communes avec notre approche. Dans la deuxième moitié de cette partie, nous présentons tour-à-tour les deux principaux paradigmes sur lesquelles notre approche est fondée. Nous faisons d'abord une présentation informelle du langage FoCal au chapitre 3. Cette présentation introduit au fur et à mesure toutes les notions importantes de FoCal à partir d'un exemple détaillé, l'exemple des treillis. Finalement cette partie se termine par le chapitre 4 avec une présentation générale du problème de satisfaction de contraintes. Nous en profitons pour donner une définition des *méta-contraintes.*

La partie II présente en détail nos contributions, nous commençons par définir le cadre du test

de propriété avec le découpage de la propriété sous test en propriétés élémentaires, la procédure de test des propriétés élémentaires, dans le chapitre 5. Nous effectuons ensuite une présentation plus formelle du langage FoCal au chapitre 6. Cette présentation se veut plus approfondie que la présentation informelle de la partie I. En particulier, nous donnons la sémantique du langage FoCal. Après cette présentation formelle, nous détaillons notre procédure pour générer des jeux de test avec l'utilisation de la programmation par contraintes dans le chapitre 7. Nous donnons les règles de traduction des programmes FoCal en un ensemble de contraintes. Cette traduction s'effectue en deux passes. La première traduit les programmes FoCal en un programme du langage FMON, langage intermédiaire monadique qui n'intègre pas les traits objets de FoCal. La deuxième passe traduit le programme FMON en un ensemble de contraintes. Pour chacune de ces deux passes nous prouvons la correction et la complétude de la traduction. Enfin, nous convertissons la précondition à l'aide du formalisme développé pour convertir le programme.

La partie III fait une présentation de l'implantation de notre approche. Nous commençons par présenter au chapitre 8 la sémantique effective des *méta-contraintes* et donnons des indications pour montrer l'équivalence de cette sémantique avec celle développée dans la partie II. Nous présentons ensuite l'outil FoCalTest développé à l'issue des travaux présentés dans la partie II. Nous présentons l'architecture de FoCalTest et précisons les choix d'implantation. Nous présentons aussi la procédure utilisée pour poser le harnais de test. Nous poursuivons ensuite sur une étude expérimentale. Nous évaluons les performances de FoCalTest en termes de temps de génération de jeux de test sur 6 exemples. Nous comparons les temps de recherche sur différentes sémantiques pour les contraintes spécifiques aux conditionnelles et filtrages par motif. Nous comparons aussi le temps de génération de jeux de test entre la stratégie de recherche aléatoire et par résolution de contraintes. Finalement, nous faisons un récapitulatif du travail et des contributions présentées dans ce manuscrit et nous décrivons les perspectives que ce soit en terme d'améliorations de l'outil FoCalTest ou bien en terme de nouveaux horizons.

Première partie

État de l'art et contexte

Chapitre 1

Le test logiciel

Nous introduisons dans ce chapitre le test logiciel. Notre but est de donner au lecteur un aperçu des techniques de test utilisées dans le cadre du test logiciel. Nous ne nous intéressons ici qu'au test dynamique, qui requiert l'exécution du programme. Nous ne nous intéressons pas en particulier au test statique qui relève de l'analyse statique de code. Nous avons fait le choix de ne présenter que le test dynamique car notre méthodologie de test relève de cette classe de test.

1.1 Définition du test logiciel

Le test logiciel est une activité à part entière du développement logiciel. Les principaux modèles de cycle de développement de logiciels imposent des phases de test. Par exemple, le modèle du cycle en V présenté dans la figure 1.1 fait correspondre à chaque phase de conception une phase de test.

Il existe de nombreuses définitions du test logiciel. Néanmoins, toutes les définitions s'accordent à dire que le test est un processus de validation qui a pour but de révéler des erreurs dans un logiciel. Avant même qu'une théorie du test logiciel voie le jour, Dijkstra rappelle que « Tester un programme peut démontrer la présence de fautes, jamais leur absence » [Dij72]. Cette remarque est importante car effectivement, en dehors du test exhaustif, le test ne peut pas montrer l'absence d'erreur. Pour cela il faut exécuter tous les comportements du logiciel, ce qui n'est pas possible en pratique car le nombre de comportements est en général trop grand.

Nous retenons ici, la définition du test logiciel orientée test dynamique : *tester un programme signifie donner en entrée des valeurs et recevoir après exécution de celui-ci la (ou les) sortie pour les comparer avec la sortie attendue* [IEE98]. La figure 1.2 résume le cycle de génération, exécution et soumission qui est effectué pour tester un logiciel.

Dans la définition que nous venons de donner, tester implique de connaître les sorties que doit nous donner le programme pour les valeurs d'entrée examinées. Le problème de la détermination des sorties d'un programme est appelé *problème de l'oracle*. L'oracle est généralement donné sous la forme d'une spécification qui indique le comportement que doit avoir le programme. Il peut être une autre implantation du programme ou le testeur lui-même. L'oracle peut ne pas donner les valeurs exactes attendues du programme mais être un prédicat qui doit être satisfait par les sorties réelles.

Avant de tester un logiciel, on fixe un objectif à atteindre. Ce critère va nous permettre de

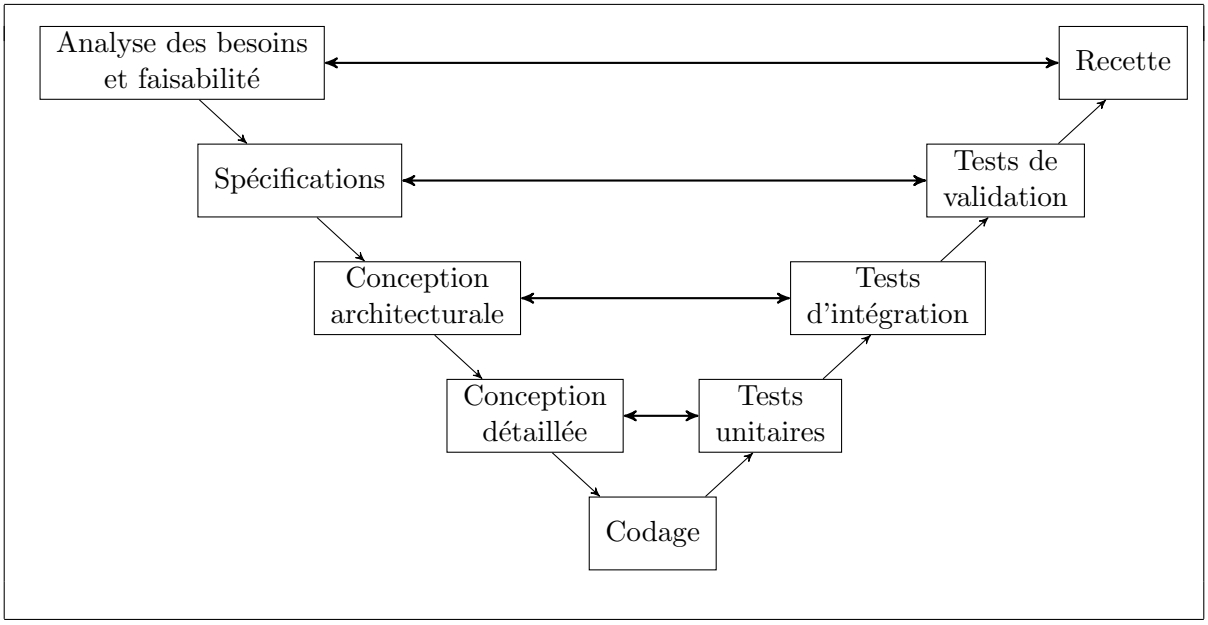

FIGURE 1.1 – Cycle de vie en V d'un logiciel

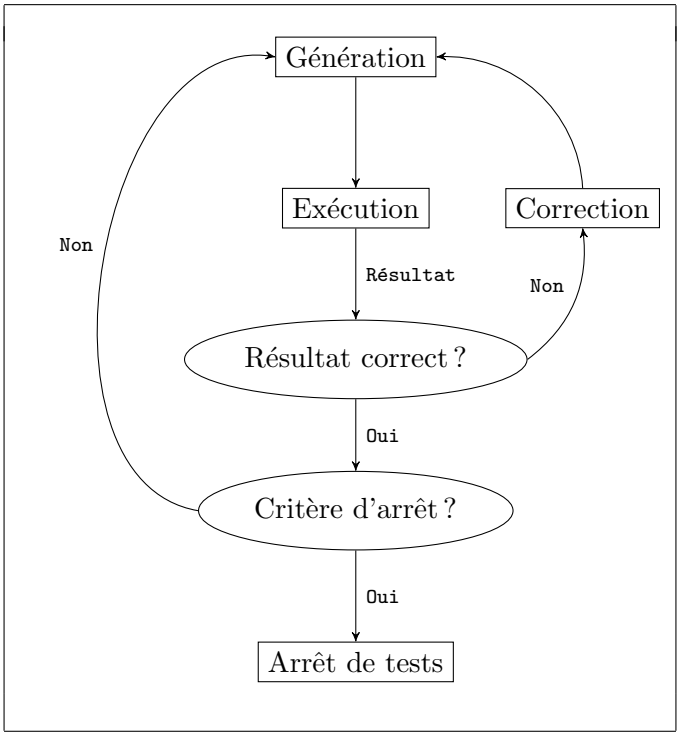

FIGURE 1.2 – Test d'un logiciel

décider de l'arrêt du test, c'est pour cela que les critères de test sont parfois appelés *critère d'arrêt*.

Pour résumer, tester un programme revient à déterminer des valeurs d'entrée et des valeurs de sortie, les *données de test*. Un ensemble de données de test (valeurs d'entrée et les sorties correspondantes) est un *jeu de test*. Tester un programme revient à lui soumettre un ensemble de *jeux de test* dans le but d'atteindre un certain *critère de test*.

1.2 Couverture de test

Avant de commencer à tester un logiciel, il faut établir les objectifs qu'on souhaite atteindre lors de l'exécution du programme. Faire du test sans avoir d'objectif précis n'a pas de sens car, d'une part, cela ne nous donne pas l'indication de quand arrêter de tester le logiciel et, d'autre part, nous ne pouvons pas dire ce qui a été testé.

Pour répondre à ces deux exigences, il a été défini des critères de couverture. Les critères de couverture s'appuient soit sur la spécification du programme, soit sur le programme lui-même. Chaque critère de couverture présente des avantages et inconvénients et permet d'identifier certains types d'erreur.

Nous présentons les principes qui permettent de définir des critères de test. Nous les classons en deux catégories les *critères structurels* et les *critères fonctionnels*.

1.2.1 Test structurel

Les *techniques de test structurel* constituent la classe des techniques de test qui reposent sur une analyse de la structure du programme testé. Elles définissent des objectifs à atteindre en terme de couverture de la structure interne du programme testé.

La plupart des techniques de test structurel définissent leurs objectifs en se fondant sur le graphe de flot de contrôle du programme testé. Il permet de représenter un programme sous une forme non linéaire. Il laisse apparaître les chemins d'exécution définis par les différentes structures de contrôle (les structures *conditionnelles*, *itératives* et *séquentielles* principalement).

Les nœuds d'un graphe de flot de contrôle représentent des blocs d'instructions séquentielles. Les arcs du graphe symbolisent la possibilité de transfert de l'exécution d'un nœud à un autre. Lorsque plusieurs arcs sortent d'un même nœud, les arcs sont annotés par la condition de transfert pour un programme impératif. Les arcs sortants d'un même nœud portent des conditions mutuellement exclusives (déterminisme) et doivent couvrir tous les cas logiques possibles.

Exemple 1.2.1. Soit le programme C présenté en figure 1.3 qui prend en entrée deux entiers e_1 et e_2 et retourne un entier qui donne le signe du produit de e_1 et e_2 (-1 si le produit est négatif et 1 si le produit est positif) ou 0 si l'un des deux est nul. Le graphe de flot de contrôle du programme est présenté sur la droite de la figure.

On remarque que dans le graphe de flot de contrôle, on a mis en évidence les chemins de programme dans lesquels les conditionnelles ne sont pas vérifiées par des flèches en pointillées. Par exemple, l'arc $b \to e$ montre le chemin qui correspond à la négation de `e1` < 0.

```
int signe(int e1, int e2)
{
  int s_e1 = 1;
  int s_e2 = -1;
  int res = -1;

  if(e1 == 0 || e2 == 0)
    res = 0;
  else
  {
    if(e1 < 0)
      s_e1 = -1;
    if(e2 > 0)
      s_e2 = 1;
    if(s_e1 == s_e2)
      res = 1;
  }
  return res;
}
```

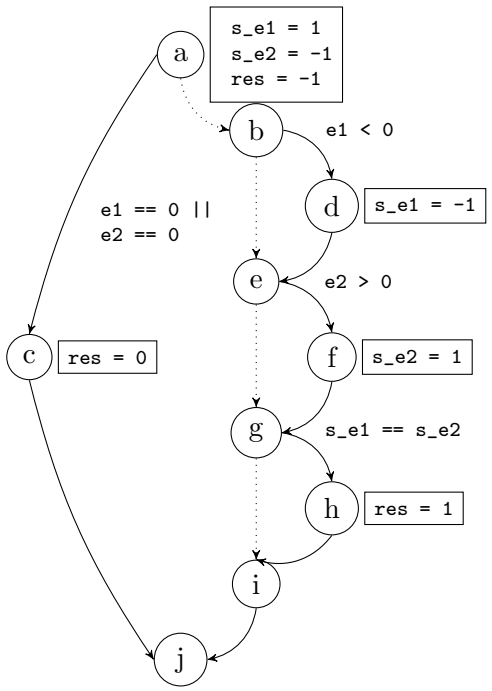

FIGURE 1.3 – Calcul le signe du produit de deux entiers

Critère sur le flot de contrôle

Les critères de couverture fondés sur le flot de contrôle s'intéressent directement à la structure du graphe de flot de contrôle du programme testé. Ces critères portent sur les arcs, les nœuds et aussi les chemins dans le graphe.

Parmi les critères les plus classiques nous en détaillons trois :
- la *couverture des instructions*. Il s'agit d'exécuter au moins une fois chaque instruction du programme. Au niveau du graphe de contrôle cela consiste à passer au moins une fois sur chaque nœud ;
- la *couverture des décisions*. Pour chaque décision (les prédicats des conditionnelles) du programme il faut évaluer la décision une fois à *vrai* et une fois à *faux*. Dans le graphe de contrôle cela revient à passer au moins une fois dans chaque arc ;
- la *couverture des chemins exécutables*. Elle réclame de passer au moins une fois par tous les chemins du graphe de flot de contrôle qui caractérisent une exécution possible du programme.

Exemple 1.2.2. Pour le programme qui calcule le signe du produit de deux entiers présenté à l'exemple 1.2.1, ces trois critères se couvrent de la manière suivante :

On peut obtenir la *couverture des instructions* avec trois jeux de test. Un jeu de test avec un entier nul, un avec les deux entiers non nuls et de même signe et enfin un avec e_1 positif et e_2 négatif. Par exemple : $e_1 = 0$, $e_2 = 1$ (a, c), $e_1 = -2$, $e_2 = 18$ (a, d et f) et $e_1 = 10$, $e_2 = 10$ (a, f et h).

La couverture des décisions peut être obtenue avec trois jeux de test. On peut prendre par exemple les entrées suivantes :

$$e_1 = -2, e_2 = 18 \ (b \to d,\ e \to f \text{ et } g \to i)$$
$$e_1 = 10, e_2 = 10 \ (b \to e,\ e \to f \text{ et } g \to h)$$
$$e_1 = 0, e_2 = 25 \ (a \to j)$$

La couverture des chemins exécutables demande cinq jeux de test au moins. On peut prendre par exemple :

$$e_1 = 0, e_2 = 8 \ (a \to c \to j)$$
$$e_1 = 4, e_2 = 9 \ (a \to b \to e \to f \to g \to h \to i \to j)$$
$$e_1 = -7, e_2 = 4 \ (a \to b \to d \to e \to f \to g \to i \to j)$$
$$e_1 = 13, e_2 = -3 \ (a \to b \to e \to g \to i \to j)$$
$$e_1 = -6, e_2 = -7 \ (a \to b \to d \to e \to g \to h \to i \to j)$$

D'autres critères fondés sur les décisions existent. Ils s'appuient sur la structure des décisions. Dans ce cadre, on appelle *condition* une expression booléenne atomique sans connecteur.

Le *critère des conditions* demande des jeux de test pour lesquels chaque condition dans les décisions est une fois à *vrai* et une fois à *faux*. Le nombre de jeux de test est alors le double du nombre de conditions dans la décision.

Le *critère des conditions multiples* réclame que la table de vérité des conditions soit couverte. Si une décision contient n conditions, il faut alors 2^n jeux de test.

Le *critère de décisions/conditions modifié* (MC/DC) est plus subtil. Il réclame de montrer pour chaque condition l'*indépendance d'impact*. Cette dernière exige d'exhiber pour chaque condition C deux jeux de test dans lesquels seule la valeur de vérité de C varie et où la valeur de vérité de la décision change entre les deux jeux de test.

Exemple 1.2.3. Soit la décision $A \ || \ (B \ \&\& \ C)$ avec les conditions A, B et C. Couvrir le critère MC/DC nécessite 6 jeux de test. Voici les requis sur les valeurs de vérité des conditions (par exemple, la ligne 1 indique que l'on cherche un jeu de test qui rende A vraie, B fausses etc) :

	A	B	C	Résultat	Indépendance sur
1	V	F	F	V	A
2	F			F	
3	V	V	V	V	B
4		F		F	
5	V	V	V	V	C
6			F	F	

Critères sur les flots de données

Dans ce critère, on se concentre sur les flots d'informations qui circulent entre les nœuds du graphe de flot contrôle. On effectue une analyse sur les définitions et utilisations des variables dans le programme. On annote dans le graphe de flot de contrôle chaque nœud et chaque arc avec la liste des variables qui y sont utilisées et définies. Il s'agit dans ce cas de couvrir l'ensemble des chemins qui relient une définition de variable à une utilisation de la valeur définie.

Test mutationnel

Le test mutationnel est une manière originale de tester un programme. Il a été introduit par DeMillo et Lipton [DLS78]. Le principe du test mutationnel est de modifier le programme initial en modifiant une instruction. Un programme modifié de la sorte est appelé *mutant*.

L'ensemble des modifications autorisées est prédéterminé, ces modifications sont définies à l'aide d'*opérateurs mutationnels*. Le but du test mutationnel est de déterminer la qualité d'un ensemble de jeux de test en lui attribuant un *score mutationnel*. Le principe est de soumettre l'ensemble de jeux de test sur chacun des mutants. Un mutant est dit tué lorsqu'il se comporte différemment du programme initial. Le calcul du *ratio nombre de mutants tués/nombre de mutants total tuables*[1] donne le score mutationnel.

L'efficacité de cette méthode repose sur le bon choix des opérateurs de mutation et sur leur adéquation avec les erreurs réelles dans le programme testé. Elle est utilisée principalement pour évaluer des méthodes de sélection de jeux de test.

1.2.2 Test fonctionnel

Les *techniques de test fonctionnel* sont fondées sur l'analyse de la spécification du programme testé. Dans cette classe de test, on n'utilise pas le code source du programme, seul le code exécutable correspondant est utilisé. Le test fonctionnel est aussi appelé test *boîte noire*. Cette métaphore provient du fait que le programme est testé au travers de ses interfaces d'entrée et de sortie et qu'on ne connaît rien de son contenu. Les jeux de test sont sélectionnés à partir d'une analyse de la spécification. Cette spécification peut prendre différentes formes. Elle peut tout aussi bien être rédigée en langue naturelle ou à l'aide d'un langage formel.

Les objectifs de test sont déterminés à partir d'une analyse partitionnelle des domaines d'entrée et de sortie du programme. Chaque domaine d'entrée est découpé en une partition qui doit être justifiée par la spécification. Chaque partition doit représenter un comportement spécifique du programme.

Une fois que le partitionnement est trouvé, on distingue deux manières de l'exploiter : le test nominal et le test aux limites.

Dans le test au fonctionnement *nominal*, on choisit dans chaque partition une valeur d'entrée quelconque et on détermine, toujours à partir de la spécification, la valeur de sortie correspondante. Cela permet de tester au moins une fois chacun des comportements repérés.

Dans le cas du *test aux limites*, on détermine des jeux de test aux alentours des bornes de chaque partition. Ce choix de test s'explique par le fait que les bornes des partitions sont des valeurs qui sont forts susceptibles de provoquer des comportements incorrects du programme.

Exemple 1.2.4. Reprenons le programme qui calcule le signe du produit de deux entiers. Une valeur d'entrée peut être nulle, positive strictement ou négative strictement. Ainsi une partition pertinente pour notre problème est constituée des ensembles $\mathbb{Z}^-, \{0\}, \mathbb{Z}^+$. En mode nominal, on retiendra 9 jeux de test de manière à couvrir tous les cas possibles. Pour du test aux limites, on peut considérer les valeurs autour de 0 ainsi que, les valeurs maximales et minimales de la représentation machine des entiers.

1.3 Génération des jeux de test

La partie la plus délicate dans le test est la génération des jeux de test. Après avoir déterminé à l'aide de critères de couverture l'ensemble des buts à atteindre, il faut rechercher des jeux de test qui permettent d'obtenir la couverture. Cette sélection peut être libre si on a un oracle complet (qui décide la valeur de sortie pour toutes valeurs d'entrée) du programme sous test

1. Un mutant est dit tuable s'il n'est pas équivalent au programme sans injection d'erreur.

ou dans le cas contraire être contrainte par le manque d'information de l'oracle. Nous exposons dans ce qui suit les problèmes que peuvent poser un oracle partiel.

1.3.1 Approche aléatoire

Le test aléatoire [Mye04] consiste à sélectionner les jeux de test en utilisant une répartition uniforme sur les domaines d'entrée du programme. On teste ainsi au hasard les comportements du programme sous test. Cette approche est facile à mettre en œuvre mais comporte des défauts. En donnant en entrée des valeurs aléatoires, on n'est pas assuré de tester à chaque fois un comportement différent. On teste surtout les comportements les plus probables Tester plusieurs fois le même comportement n'est pas gênant en soi. Le problème est plutôt posé pour les comportements qui sont activés sur des conditions des variables d'entrée restreignantes. Le test aléatoire peut pas garantir que ces comportements sont activés. En augmentant considérablement le nombre de jeux de test, on augmente la probabilité de sensibiliser les comportements les moins probables mais on n'est pas assuré de tous les obtenir.

1.3.2 Approche statistique

Le test statistique [TF89] est une amélioration de la sélection aléatoire des jeux de test. Celui-ci consiste à modifier la répartition de la fonction de génération aléatoire afin d'augmenter la probabilité de sensibiliser les comportements qui sont très peu probables en présence d'un tirage aléatoire uniforme sur le domaine des variables d'entrée. Dans l'idéal, on cherche à obtenir une fonction de répartition des variables d'entrée telle que chaque comportement du programme a une probabilité uniforme d'être sélectionné.

Exemple 1.3.1. Soit l'instruction suivante :

```
if( x < 50)
{ // Comportement 1
  ...
}
else
{ // Comportement 2
  ...
}
```

Si les portions de programme des deux branches de la conditionnelle ne contiennent qu'un seul chemin, avec une approche statistique, on cherche à trouver des valeurs aléatoires pour x de manière à avoir la probabilité $\frac{1}{2}$ de sensibiliser le comportement 1 et la probabilité $\frac{1}{2}$ de sensibiliser le comportement 2. Il faut alors utiliser un générateur qui génère des valeurs de x inférieures à 50 avec 1 chance sur 2. De manière générale, si le comportement 1 contient n chemins et si le comportement 2 en contient m, on va chercher à avoir des valeurs aléatoires de x qui sensibilisent le comportement 1 avec une probabilité $\frac{n}{n+m}$ ainsi que le comportement 2 avec une probabilité $\frac{m}{n+m}$.

De nombreux travaux concernent l'automatisation de la méthode statistique comme ; par exemple [TFW93] et [DGG04].

1.4 Synthèse

Nous avons défini dans ce chapitre les principales notions nécessaires pour appréhender le test logiciel. Cette présentation n'est bien sûr pas complète mais donne un aperçu suffisant pour le reste de ce manuscrit.

Dans le chapitre suivant, nous faisons un état de l'art des outils de test et détaillons les principaux travaux qui ont eu pour but d'automatiser le test, plus précisément la phase de génération des jeux de test.

Chapitre 2

Travaux et outils sur la génération de jeux de test

De nombreux travaux de recherche ont été menés sur le test logiciel, aussi bien afin de donner des fondements mathématiques que pour automatiser la procédure de test. De nombreux outils ont été développés. Le but de ce chapitre n'est pas de faire un recensement exhaustif de ces outils mais plutôt de donner une idée générale des moyens mis en œuvre dans la communauté du test logiciel pour intégrer la composante test dans les processus de développement. D'autre part, nous nous intéressons principalement aux approches et outils proches de nos préoccupations. Dans ce chapitre nous présentons les différents travaux en matière de génération de jeux de test en nous axant sur la manière de les obtenir. Dans la première partie nous présentons des travaux qui se servent du programme testé afin d'obtenir des jeux de test qui respectent un critère de couverture sur le code (test structurel). Dans la seconde partie nous présentons des travaux qui se fondent sur les spécifications du programme sous test ou un modèle de celles-ci (test fonctionnel).

2.1 Test à partir de programmes

Ici est présenté un ensemble de travaux qui cherchent à réaliser du test de programmes sans utilisation d'une spécification particulière. Chacun des travaux suppose l'existence d'un oracle qui établit les valeurs de sortie à partir des valeurs d'entrée. L'objectif de ces travaux est d'assurer une certaine couverture structurelle du programme testé. Nous présentons dans cette partie deux techniques assez récentes, à savoir l'exécution concolique et l'utilisation de la résolution par contraintes.

2.1.1 Obtention de jeux de test par exécution concolique

L'approche classique du test structurel est d'effectuer une évaluation symbolique du programme [How77, GRT07]. À chaque variable d'entrée du programme est attribuée une valeur symbolique et l'évaluation du programme sur ces symboles conduit pour chaque chemin du programme à un prédicat qui caractérise les conditions nécessaires pour activer le chemin. Reste alors à déterminer des valeurs concrètes pour les variables d'entrée qui satisfont le prédicat de chemin pour pouvoir activer ce chemin.

L'exécution concolique est une méthode qui mélange exécution et évaluation symbolique. L'idée sous-jacente à l'exécution concolique est de se servir de l'exécution (concrète) pour, par exemple, forcer des variables symboliques à leur valeur réelle. Cette approche offre l'avantage

d'assurer que seuls des chemins exécutables sont empruntés, à la différence de l'approche classique évoquée dans le paragraphe précédent.

L'exécution concolique a été utilisée principalement pour définir des méthodes de test unitaire (test d'une seule fonction ou méthode) visant à satisfaire des critères de couverture structurelle. Le principe est de commencer par exécuter le programme sur une valeur d'entrée obtenue arbitrairement. L'exécution concolique nous permet d'obtenir le prédicat correspondant au chemin sensibilisé. Ce prédicat de chemin est ensuite modifié pour obtenir un ensemble de contraintes à résoudre qui caractérise un (ou plusieurs) chemin qui n'a pas encore été sensibilisé. Ainsi, cette technique permet d'atteindre la couverture sur un critère « tous les chemins exécutables » sans avoir à calculer au préalable le graphe de flot de contrôle de la fonction testée. Une instrumentation du code permet de reconstruire au fur et à mesure de l'exécution concolique le graphe de flot de contrôle.

La méthode par exécution concolique a été utilisée dans plusieurs outils de test structurel. Par exemple, PathCrawler [MMW04] permet de tester automatiquement des fonctions C avec les critères « tous les chemins », Pex [TdH08] est l'outil de test intégré à .NET et permet à partir d'un point de programme de générer des jeux de test qui couvrent les chemins passant par ce point. Java PathFinder [VPK04] est un outil de test pour Java qui intègre, en plus de l'exécution concolique, des techniques de vérification par modèle. Il est capable de faire du test de programmes *multithreadés*. PathCrawler a besoin que l'utilisateur lui donne un oracle pour déterminer si les jeux de test passent. Les deux autres outils utilisent les mécanismes d'exception pour déterminer automatiquement si un jeu de test ne passe pas. De plus, le testeur a la possibilité de regarder le rapport de test généré pour vérifier l'exactitude des sorties du programme.

Chacun de ces outils utilise des solveurs de contraintes pour obtenir les jeux de test à partir du prédicat de chemin récolté. Pathcrawler utilise Colibri, le solveur de contraintes développé au CEA. Java PathFinder utilise une version modifiée du solveur de contraintes Korat [BKM02] dédié aux programmes JAVA, Pex utilise le solveur de contraintes Z3 [dMB08]. Toutefois, les contraintes ne forment pas l'aspect central de la méthode de test utilisée. Nous présentons dans ce qui suit un travail qui repose intégralement sur la notion de contraintes

2.1.2 Obtention de jeux de test par traduction du programme en contraintes

La recherche de jeux de test pour atteindre des couvertures sur les critères structurels se prêtent bien à la résolution de contraintes. Ici est présentée une méthode de test qui traduit intégralement et systématiquement le programme sous test en un problème de résolution de contraintes. Nous axons la présentation sur l'outil Inka [GBR00] dont les fondements ont guidé et inspirés en partie les travaux décrits dans cette thèse.

Le prototype Inka [GBR00] permet de faire du test structurel de programmes C et C^{++}. Inka peut être comparé à l'atelier récent Euclide [Got09] qui en reprend les principes de base. La méthode utilisée par Inka fait appel à la programmation par contraintes et offre la possibilité d'utiliser les critères de couvertures usuels. À partir d'un critère de couverture structurelle donné, l'outil Inka génère les jeux de test en passant par quatre étapes : mettre en forme SSA [CFR$^+$91] la fonction testée (celle-ci permet de désambiguïser les utilisations/définitions de variables) ; traduire le programme résultant en un programme CLP(FD), langage de programmation logique à contraintes sur les domaines finis ; calculer un ensemble de points de programme à atteindre pour satisfaire le critère voulu, puis, pour chacun de ces points calculer les contraintes qui

permettent au flot de contrôle de passer par celui-ci ; résoudre les contraintes obtenues à l'étape précédente. Ces étapes nous donnent ainsi les jeux de test.

Cette approche présente une originalité dans la manière dont sont traitées les structures de contrôle conditionnelles et itératives. Pour ces dernières, des contraintes spéciales CLP(FD) ont été définies [GBR98, BGM+02, DGD07]. Cette approche rejoint la nôtre étant donné que nous nous sommes inspirés des contraintes spéciales pour définir nos propres contraintes modélisant les structures particulières à FoCal.

2.2 Test à partir de modèles

Nous présentons dans cette section quelques outils qui permettent de générer des jeux de test à partir d'une spécification formelle ou d'un modèle formel, voire semi-formel.

2.2.1 À partir de spécifications B et Z

L'environnement BZ-Testing-Tools (BZ-TT) [LP01, BLLP04] (qui se nomme maintenant Leirios) est un environnement de test qui automatise la phase de recherche de jeux de test à partir de spécifications B ou Z. Il est inspiré des travaux de Dick et Faivre [DF93] qui se fondaient sur des spécifications VDM. Le programme testé n'est pas nécessairement celui obtenu par raffinement de la spécification utilisée pour tester, BZ-TT permet de tester des programmes qui ont été développés directement dans un langage tiers.

BZ-TT inclut la possibilité de faire du test aux limites et d'utiliser des critères de couverture structurelle. Il s'agit d'une méthode de test qui se trouve à mi-chemin entre le test structurel et le test fonctionnel car les critères de test sont fondés sur la structure de la spécification et non sur celle de l'implantation.

Pour BZ-TT, un jeu de test est une suite d'appels d'opérations. Il vise à tester une opération lorsque la machine est dans un état donné. À partir d'une opération de la machine abstraite B, BZ-TT commence par déterminer l'ensemble des buts à atteindre pour tester l'opération. Il s'agit d'une phase qui consiste à déplier la définition de l'opération et à y appliquer des transformations [LPU04]. Cela donne un ensemble d'états dans lesquels l'opération doit être testée. Toutes ces étapes sont effectuées automatiquement, l'outil utilise un solveur de contraintes ensemblistes (CLPS) [ABC+02] pour générer les jeux de test.

Pour le test aux limites, BZ-TT propose de maximiser et minimiser les valeurs des variables d'états du système apparaissant dans la précondition de sorte que l'invariant de la machine et cette même précondition soient tous deux vérifiés. À partir des états calculés, BZ-TT est capable de générer les jeux de test adéquats.

BZ-TT rejoint FoCalTest dans le sens où la phase de génération de jeux de test est assurée par la résolution d'un ensemble de contraintes. De plus, BZ-TT n'est pas dépendant de l'environnement B puisqu'il utilise sa propre représentation des spécifications et pourrait être adapté à d'autres types de spécification que B ou Z. D'ailleurs, plus récemment, les concepts définis dans BZ-TT ont été repris dans un outil de test pour Java utilisant JML nommé JML-TT [BDL06b]. En ce point BZ-TT est supérieur à FoCalTest qui est lié à l'atelier FoCal.

2.2.2 À partir de contrats

Nous présentons dans cette section une autre forme de test, à savoir le test à partir de contrats. Un contrat est la donnée d'une précondition, d'une postcondition et d'un invariant pour une fonctionnalité d'un programme (généralement une fonction ou une méthode dans le cadre orienté objet). Ceux-ci sont généralement exprimés sous la forme de prédicats logiques. Dans la suite les outils que nous présentons utilisent des spécifications JML, langage qui permet de spécifier des contrats pour des programmes Java.

De nombreux outils utilisant JML ont été développés. On peut citer dans le domaine de la preuve de programme l'outil Krakatoa [MPMU04] qui prend en entrée des spécifications JML et utilise l'assistant à la preuve Coq pour prouver que le programme est conforme aux spécifications ou l'outil de développement formel KeY [HJL+06] qui interprète les spécifications JML pour effectuer des preuves.

JUnit, JMLUnit, JET

JUnit est une bibliothèque qui permet d'effectuer du test unitaire de programmes Java. Les unités testées sont des objets. JUnit fournit un environnement de travail pour effectuer du test de programme Java. Le testeur écrit dans une classe Java qui contient toutes les directives de tests : la classe testée, les méthodes dans la classe qu'il souhaite tester, les jeux de test qu'il soumet aux méthodes ainsi que des assertions qui font office d'oracle.

JMLUnit est un outil qui génère des fichiers de test JUnit à partir d'une description JML. Il est capable d'automatiser la phase de sélection de jeux de test et de générer un oracle à partir des spécifications JML [CL02]. L'utilisateur donne pour chaque type de données manipulé par la spécification un ensemble de valeurs possibles et JMLUnit utilise ensuite ces valeurs pour générer un oracle. JMLUnit vérifie à l'exécution les violations de préconditions et invariants

JET [CRM07, CCCL08] est un outil de test automatique de classes Java qui utilise les annotations JML, analogue à JMLUnit. Son but est de tester toutes les méthodes d'une classe. Chaque méthode est testée séparément. Il génère des jeux de test aléatoirement et est capable d'exporter les jeux de test dans une classe JUnit. Il peut également utiliser des contraintes sur le domaine fini des entiers pour trouver des jeux de test. Ainsi, comme FoCalTest, JET utilise des contraintes pour générer des jeux de test en analysant les préconditions. Cependant, JET se limite à la génération de contraintes sur les types de données entières. Pour les autres types de données, JET génère des valeurs aléatoires. Jet a donc une utilisation des contraintes plus limitée que FoCalTest qui intègre des contraintes sur les données de types concrets.

Jartege

Java Random Test Generator (Jartege) [Ori05] est un autre outil de test pour Java qui utilise les spécifications JML. Les spécifications JML servent aussi bien à générer les jeux de test qu'à produire un oracle.

Un jeu de test de Jartege est une suite d'appels de méthodes, déterminée aléatoirement. La spécification JML est utilisée pour déterminer les paramètres des appels de méthode de sorte que les préconditions de chaque méthode soient vérifiées. Les paramètres sont déterminés à l'aide d'une analyse de la précondition. Ainsi, Jartege extrait les intervalles de valeurs possibles pour les paramètres entiers.

Jartege utilise les invariants et postconditions des spécifications JML pour calculer le verdict du test. Il est capable de générer un rapport de test qui répertorie les jeux de test qui ont échoués.

2.2.3 À partir de spécifications algébriques

Des travaux ont été effectués dans le but de créer une théorie du test logiciel à partir d'une spécification fondée de manière générique sur des algèbres [Mar91] [GA95]. Ces travaux supposent que les programmes sont spécifiés de manière structurée et hiérarchisée par l'intermédiaire de modules qui contiennent un ensemble d'opérations signées et d'axiomes, ces derniers étant sous la forme de clauses de Horn [BGM90].

La méthodologie dégagée par ces travaux propose de tester chaque axiome de la spécification séparément. Ce test est guidé par des hypothèses, incluses dans la théorie, qui ont pour but de mettre en œuvre les hypothèses utilisées par le testeur mais qui ne sont en réalité pas explicitées. La définition de ces hypothèses définit un cadre formel du test logiciel et permet aussi de fournir des critères de couverture à atteindre [Mar98].

Par exemple, l'*hypothèse d'uniformité* précise que pour une propriété donnée et une variable de la propriété, si la propriété est vérifiée pour une valeur possible de la variable alors la propriété est vérifiée pour toutes les valeurs ; toujours pour une propriété et une variable données, l'hypothèse de régularité de niveau k (k entier) indique que si la propriété est vérifiée pour toutes les valeurs de la variable de taille inférieure ou égale à k alors la propriété est vérifiée pour toute valeur de k. Ces hypothèses ne doivent bien entendu pas être utilisées n'importe quand, par exemple il est nécessaire de décomposer le domaine des variables en un ensemble de classes d'équivalence avant d'appliquer l'hypothèse d'uniformité pour ne faire qu'un test par classe.

Ainsi, pour un ensemble de variables d'une propriété donnée, on peut appliquer au choix l'une des hypothèses de la théorie. Ces idées ont donné lieu à l'outil de test Loft. Les hypothèses d'uniformité/régularité sont appliquées après le dépliage des définitions équationnelles des opérations dans les propriétés testées [BGM90].

En ce qui concerne le test à partir de spécification algébriques, nous pouvons également citer les travaux d'Hamlet et Antoy [AH00] sur le test d'implantation de types de données abstraits donc on possède une spécification algébrique. Des règles de réécriture permettent de transformer la spécification en oracle. Les jeux de test sont déterminés aléatoirement. Gaudel et Le Gall présentent dans [GG08] un panorama assez complet de la problématique du test à partir de spécifications algébriques ainsi que quelques outils et méthodes.

2.2.4 À partir de propriétés

Les travaux présentés dans cette partie ont tous en commun l'idée de tester une propriété vis à vis d'une implantation. C'est par exemple une propriété qui relie plusieurs exécutions du même programme ou une propriété qui relie les résultats de plusieurs fonctions du programme sus test.

Test métamorphique

Le test métamorphique [CCY98] considère une fonction d'un programme ainsi que les relations entre les entrées de la fonction et ses sorties. Le test métamorphique consiste à vérifier si

l'implantation de la fonction vérifie bien les relations. Cette méthode a été utilisée pour obtenir et soumettre des jeux de test automatiquement en utilisant la programmation par contrainte [GB03].

Plus formellement, soit f la fonction sous test et soit $\{I_1, \ldots, I_n\}_{n>1}$ des valeurs d'entrées de f. Soit r une relation sur $\{I_1, \ldots, I_n\}$ et r_f une relation sur les sorties de f $(f(I_1), \ldots, f(I_n))$. Une propriété métamorphique sur f est une propriété de la forme :

$$r(I_1, \ldots, I_n) \Longrightarrow r_f(f(I_1), \ldots, f(I_n))$$

Elle signifie que si les entrées I_1, ..., I_n vérifient la relation r alors les sorties respectives de chacune d'elles vérifient la relation r_f. Il faut rappeler qu'on se place ici au niveau spécification. Une telle propriété peut être obtenue à partir de la spécification de f par exemple. Toute implantation de f doit satisfaire la propriété métamorphique.

Exemple 2.2.1. On considère la fonction qui calcule le PGCD de deux entiers. On peut utiliser la relation métamorphique suivante qui correspond à la propriété de commutativité du PGCD :

$$\begin{cases} I_1 = (u, v) \\ I_2 = (v, u) \end{cases} \Longrightarrow \gcd(I_1) = \gcd(I_2)$$

Maintenant, si on considère une implantation p potentiellement incorrecte de la fonction f, la propriété devient la relation métamorphique suivante :

$$r(I_1, \ldots, I_n) \Longrightarrow r_f(p(I_1), \ldots, p(I_n))$$

Du point de vue du test, un jeu de test est un ensemble de valeurs d'entrées I_1, ..., I_n qui vérifient la relation r. Le résultat du test est alors donné par la vérification de la relation sur les sorties retournées par f.

Exemple 2.2.2. Reprenons l'exemple du PGCD. Considérons une implantation gcd de la fonction sur les entiers naturels. En reprenant la propriété métamorphique de l'exemple 2.2.1, on peut considérer les jeux de test CT1 et CT2 suivants :

	I_1	I_2
CT1	$(18, 64)$	$(64, 18)$
CT2	$(120, 4590)$	$(4590, 120)$

Le test métamorphique s'approche beaucoup de notre méthodologie de test pour FoCal car elle se fonde sur une propriété qui expose une relation entre les entrées et les sorties du programme.

QuickCheck

QuickCheck est un outil de test fondé sur le test de propriétés [CH00, CH02]. Il est inclus, en tant que bibliothèque, dans le langage de programmation Haskell et forme, à notre connaissance, le premier outil qui exploite des propriétés pour faire du test pour des programmes écrits dans un langage fonctionnel.

L'originalité de QuickCheck provient du fait qu'il ajoute à Haskell la possibilité d'écrire des propriétés. Le langage de spécification offre la possibilité d'utiliser des combinateurs, ce qui le rend flexible et extensible. Les propriétés considérées par QuickCheck correspondent à

un sous-ensemble des propriétés du premier ordre dans lesquelles les variables sont quantifiées universellement en tête. Les propriétés portent directement sur des fonctions Haskell.

Pour QuickCheck, un jeu de test est une valuation des variables quantifiées dans la propriété. Les valeurs des variables sont générées aléatoirement. Lorsqu'une propriété est sous la forme d'une implication, les jeux de test considérés sont les valuations des variables qui satisfont la prémisse de l'implication. Dans le cas où la propriété testée contraint fortement les données (par exemple la propriété requiert une liste triée) l'approche pâtit du fait que les données sont générées aléatoirement. Pour remédier quelque peu à ce problème, QuickCheck intègre la possibilité pour l'utilisateur de définir et utiliser ses propres générateurs de valeurs pour un type de données (par exemple un générateur de listes triées).

De nombreux outils s'inspirant de QuickCheck ont été développés par la suite. Ces outils sont fondés sur le test de propriétés et apportent chacun leur originalité. Par exemple, il y a QCheck pour SML ou Gast pour Clean [KAaRP03, KATP02]. Dans ce dernier, trois stratégies d'énumération des variables sont proposées pour tester la propriété : une énumération exhaustive pour les types finis, une génération aléatoire ou bien une énumération qui va des plus petites valeurs vers les plus grandes.

L'idée d'utiliser du test de propriétés a été reprise dans le cadre des assistants à la preuve. L'idée principale est de tester une propriété avant de rentrer dans la phase de preuve. Cela permet de rejeter les propriétés fausses ou de corriger des erreurs d'implantation rapidement. Ainsi, Nipkow et Berghofer [BN04] ont développé un outil de test s'inspirant de QuickCheck dans Isabelle [NPW02]. Celui-ci utilise la définition des types de données pour obtenir des générateurs aléatoires. Dybjer, Qiao et Takeyama [DHT03] ont créé un outil de test dans l'environnement de preuve d'AGDA. L'outil de test est accessible directement au milieu d'une preuve, les lemmes intermédiaires peuvent ainsi être testés dès leur conception. Owre [SO06] quant à lui propose un outil de test pour le système PVS [ORS92]. Notre approche s'insère directement dans la lignée de QuickCheck et de ces outils de preuve qui incorporent des techniques de test.

HOL-TestGen

HOL-TestGen [BW] est un outil de test unitaire à partir de spécifications formelles, intégré à l'assistant de preuve Isabelle/HOL. HOL-TestGen permet d'écrire des spécifications de test en HOL, formalisme de la logique d'ordre supérieur. Cette spécification de test décrit les propriétés à tester du programme sous test, elle s'accompagne d'une théorie de test qui précise les types de données et les prédicats utilisés dans la spécification de test (sous la forme $precondition \Rightarrow postcondition$). L'outil est capable de partitionner l'espace des entrées et produit automatiquement les cas de test qui seront ensuite instanciés en données concrètes de test. Il fournit également les scripts pour lancer les jeux de test. Le programme sous test peut, quant à lui, être écrit dans un langage tiers (par exemple SML, C).

Il s'agit ici d'un outil qui se distingue des outils précédemment cités car il utilise les mécanismes de preuve de son hôte pour générer les tests et fait le lien entre test et preuve de la spécification de test. En effet pour les obtenir, l'outil HOL-TestGen décompose la propriété à tester en un théorème de test contenant explicitement les hypothèses de test utilisées (régularité, uniformité essentiellement).

Plus précisément, à partir de la spécification de test, HOL-TestGen [BW07] applique une procédure de dépliage des prédicats utilisés dans la propriété ainsi que des procédures de mise sous forme normale (mise en clauses de Horn, simplification de la spécification, suppression des

redondances, ...). À l'issue de cette phase, HOL-TestGen trouve un ensemble de propriétés appelées cas de test (CT_i). Les cas de test sont ensuite réunis en un théorème appelé *théorème de test* qui prend la forme suivante :

$$[\![\mathrm{CT}_1; \ldots; \mathrm{CT}_n; \mathrm{THYP}\ H_1; \ldots; \mathrm{THYP}\ H_m]\!] \Longrightarrow \mathrm{TS}$$

avec THYP $H_1; \ldots;$ THYP H_m désignant ici les hypothèses de test.

Cette propriété énonce que si les cas de test sont exécutés par le programme sous test avec succès et si les hypothèses de test sont satisfaites (elles deviennent des obligations de preuve) alors le programme sous test vérifie la spécification de test.

Les données de test sont produites par une phase de sélection qui s'apparente à un problème de satisfaction de contraintes. Cette phase utilise la génération aléatoire, les procédures de décisions d'Isabelle et différentes autres manipulations syntaxiques.

Du point de vue externe, FoCalTest et HOL-TestGen se comportent de manière similaire, en effet ils permettent tout deux de tester une propriété dans un cadre donné. Pour le premier le contexte est un ensemble de composants Focal comprenant du code et des spécifications, pour le deuxième le contexte est donné par une théorie de test dans le langage HOL et un programme sous test indépendant du reste. Les deux outils partagent certains traits en particulier la transformation de la propriété à tester. Cependant HOL-TestGen est capable de tester des propriétés qui contiennent des traits d'ordre supérieur contrairement à FoCalTest confiné à la logique du premier ordre.

2.3 Test à partir d'automates

Cette section présente des outils qui se basent sur un modèle du programme testé pour générer les jeux de test et plus particulièrement aux travaux qui utilisent une représentation sous la forme d'un automate de la spécification du programme.

2.3.1 Lutess

Lutess [dBOP+98, dBORZ99] est un outil de test pour le langage synchrone Lustre. Le test de langage synchrone présente plusieurs difficultés qui ne sont pas présentes dans les autres langages de programmation. D'une part, un programme synchrone n'est pas représenté par un graphe de flot de contrôle, mais pas un diagramme de flot de contrôle, ceci rend caduc les différentes notions de couverture usuellement utilisées pour effectuer du test structurel. Cette première difficulté est résolue par la définition de critères de couverture adaptés aux diagrammes et intégrés dans l'outil de mesure de couverture Lustructu [LP05]. La deuxième difficulté omniprésente dans le test de programme synchrone provient du fait que par définition, un programme synchrone est un programme qui interagit avec un environnement dont le comportement n'est pas contenu dans le programme.

L'outil Lutess étend le langage Lustre et permet de décrire, sous la forme d'un nœud Lustre spécial, les pré-requis du test. Il s'agit de la donnée d'une description de l'environnement du programme (cette description contient un ensemble de propriétés qui doivent être respectées par l'environnement), un oracle (une propriété de sûreté sur les valeurs d'entrée/sortie, Lutess cherche à éprouver cette propriété) et d'un ensemble de connaissances sur le programme testé, i.e. des propriétés sur les entrées et sorties qui n'affectent pas la sûreté du logiciel.

À partir de la description d'environnement, Lutess crée un générateur aléatoire d'environnement qui respecte les propriétés invariantes de la description. Le processus de test en lui-même est itératif. Chaque itération correspond à un instant logique. L'environnement prend les sorties du programme de l'instant précédent pour générer les valeurs d'entrée de l'instant courant. Au premier instant, l'environnement décide des valeurs d'entrée sans contrainte sur le passé. Ces aller-retour sont observés par un l'oracle qui génère à tous les instants un verdict.

Dans sa première version, [dBOP+98] ne prenait en compte que les variables booléennes. Depuis, l'outil a été étendu aux valeurs entières grâce à l'utilisation d'un solveur CLP(FD) [dBORZ99].

Dans la lignée des outils de test de programme Lustre, il existe l'outil de test GATel qui permet de générer automatiquement des jeux de test pour du test structurel et du test fonctionnel [Mar91].

2.3.2 Autofocus

Autofocus [SH99] est un outil de spécification graphique pour les logiciels embarqués. La structure d'un système est représentée par un diagramme de composants qui communiquent par l'intermédiaire de canaux. La dynamique d'un système est spécifiée par un ensemble d'automates à état-transition. Un automate spécifie le comportement d'un composant du système et chaque transition est annotée par des préconditions et postconditions. Autofocus prend en compte les aspects temporels et est donc adapté pour la spécification de programmes synchrones ou réactifs. Les types de données d'Autofocus sont similaires à ceux que l'on trouve dans des langages fonctionnels tels que FoCal, c'est-à-dire, les types concrets.

Autofocus incorpore un outil de test à partir de spécifications qui génère automatiquement des traces ou des séquences de test. Une trace est une suite d'appels de fonctions, elle a pour but d'atteindre un état donné.

Pour sélectionner automatiquement les traces, les automates du modèle sont traduits vers des clauses Prolog. L'énumération des valeurs ainsi que la définition d'opérateurs de base sur les types concrets sont réalisées à l'aide de contraintes CHR [Frü95]. Ce dernier point diffère de FoCalTest. Nous avons choisi de créer notre propre » énumérateur » de valeurs valable pour tous les types concrets.

2.3.3 ParTeG

ParTeG [WS08] est un outil de test qui permet de faire du test de programme à partir d'un modèle UML. Il prend en entrée une machine à états UML, un diagramme de classes ainsi que des spécifications OCL et est capable de générer des jeux de test aux limites.

Pour déterminer les jeux de tests, ParTeG analyse les chemins de la machine à états. Dans un premier temps, il ne détermine pas les paramètres des appels de méthodes rencontrées et laisse des valeurs abstraites. Lorsque ParTeG rencontre une garde, il analyse un à un les nœuds de la machine jusqu'à rencontrer une méthode dont la postcondition influe sur la garde. Cela permet d'obtenir une corrélation entre les gardes et les postconditions sous la forme de contraintes. La résolution de ces contraintes en instanciant les paramètres par des valeurs aux bornes donnent des jeux de test concrets.

D'autres travaux ont porté sur l'utilisation de modèles UML pour générer des jeux de test [PJJ+07, OA99, BGL+07]. Récemment, une étude comparative a permis de constater que les jeux de test obtenus à partir de diagrammes d'activité et de statecharts couvrent des classes d'erreurs différentes [KOAB08]. Ceci suggère que les différents schémas d'UML jouent un rôle complémentaire pour la génération de jeux de test. Outre autour d'UML, il existe de nombreux outils qui exploitent des machines à états finis ou des systèmes de transitions étiquetées que nous ne détaillons pas ici. On peut citer entre autres ASmL test Tool [BGS+03], SpecExplorer [VCG+08], TGV [JM99] et Agatha [BaFG+03].

2.4 Synthèse

Nous avons donné ici un bref aperçu de différents travaux sur le test logiciel. Nous avons pu constater qu'il existe deux principales approches pour la génération de jeux de test, l'une exploite le programme sous test, l'autre un modèle. On peut aussi remarquer que les approches et outils présentés ont recours à un processus de génération de données pseudo-aléatoire et par résolution de contraintes, par exemple l'outil de test de programme Java JET, l'outil Inka et l'outil de test intégré à Autofocus utilisent des contraintes pour obtenir des jeux de test, comme de nombreux autres outils que nous avons présentés. La résolution de contraintes est en fait utilisée avec des niveaux différents, par exemple elle peut être un outil pour générer des valeurs d'entrée qui activent un chemin donné, elle peut aussi être le formalisme de modélisation du programme sous test comme c'est le cas pour Inka. Les liens entre résolution de contraintes et test sont tels que le terme « Constraint-based testing » commence à être utilisé [BBD+09].

La méthodologie de test présentée dans cette thèse est initialement inspirée de l'outil de test QuickCheck. Nous avons toutefois dévié de QuickCheck en plusieurs points :

- les propriétés que nous testons sont réécrites en un ensemble de propriétés élémentaires. Intuitivement, ces propriétés élémentaires correspondent à chaque comportement spécifié dans la propriété d'origine. Cette transformation est analogue à la mise en forme normale des prédicats avant/après de l'outil de test BZ-TT ou des transformations de HOL-TestGen.

- nous intégrons notre méthodologie de test au sein d'un atelier de développement de logiciel certifiable. Dans un certain sens, cela nous rapproche des outils de test intégrés dans Isabelle, AGDA ou PVS puisque le but du test peut être de vérifier une propriété avant d'essayer de la prouver.

- tout comme QuickCheck, nous avons aussi une stratégie de recherche aléatoire. Mais nous nous sommes inspirés des travaux de Gotlieb sur l'outil Inka (puis Euclide) pour développer notre procédure de recherche de jeux de test par contraintes. Pour cela, nous traduisons une partie du programme testé en contraintes, puis nous traduisons la précondition de la propriété (le prédicat qui spécifie ce que doit être un jeu de test). C'est l'approche choisie par l'outil Inka. Comme pour Inka, nous avons défini des contraintes spécifiques pour modéliser les structures de notre langage de départ.

Chapitre 3

Présentation de FoCal

Le projet FoCal (`http://focal.inria.fr`) [DHVG04, ACD+08] est issu de l'initiative de Thérèse Hardin et de Renaud Rioboo. À l'origine, le but du projet était de développer une bibliothèque certifiée de calcul formel. Ici le mot certifié prend le sens donné par les assistants à la preuve, c'est-à-dire, les fonctions de la bibliothèque sont prouvées correctes par rapport à un ensemble de propriété spécifiées. Le projet avait alors pour nom *FOC* pour *Formal OCaml Coq*, qui met l'accent sur les différents outils utilisés : le langage de programmation OCaml implantant la bibliothèque et l'assistant à la preuve Coq prouvant la correction de cette dernière [Bou00b, BHR00, BHH+99].

Cette bibliothèque se présente sous la forme d'un ensemble de composants représentant les différentes structures mathématiques utilisées dans le monde du calcul formel (*monoïde*, *anneau*, *treillis*, ...). Une notion de raffinement a été définie pour pouvoir développer la bibliothèque incrémentalement. Par exemple, la structure d'anneau peut être définie comme une extension de la structure des monoïdes.

Par la suite, les différentes abstractions et outils créés pour développer la bibliothèque de calcul formel se sont avérés suffisamment originaux pour constituer les bases d'un langage de programmation [Fec01, Bou00a, Pre05]. Ainsi, le projet *FOC* « a revu ses ambitions à la hausse » et se propose maintenant de fournir un atelier de développement de logiciels certifiés.

Les programmes développables à l'aide de cet atelier ne sont pas nécessairement tournés vers le calcul formel. Le projet a aussi profité de ce changement de stratégie pour changer de nom et devenir le projet FoCal. Cet atelier fournit son propre langage de programmation, le langage FoCal. La bibliothèque de calcul formel développée fait maintenant partie intégrante de la bibliothèque standard fournie par le langage.

Autour de FoCal se trouve un certain nombre d'outils. Parmi eux, on peut citer le prouveur automatique Zenon[BDD07], le générateur automatique de documentation FocDoc ou bien des outils pour visualiser les dépendances entre les composants d'un programme FoCal.

L'atelier FoCal a été, en partie, développé dans le cadre de l'Action Concertée Incitative (ACI) sécurité et informatique. Le but de ce projet était de concevoir un atelier de construction modulaire et logiquement fondée de systèmes logiciels pouvant répondre à des exigences élevées de certification.

Depuis la fin de ce projet, FoCal est développé et maintenu dans le cadre du projet ANR *Security and Safety UndeR Focal* (SSURF) qui a pour objectif d'approfondir le sujet et d'étudier et de caractériser les fonctionnalités qu'un atelier, comme FoCal, doit offrir pour assurer des

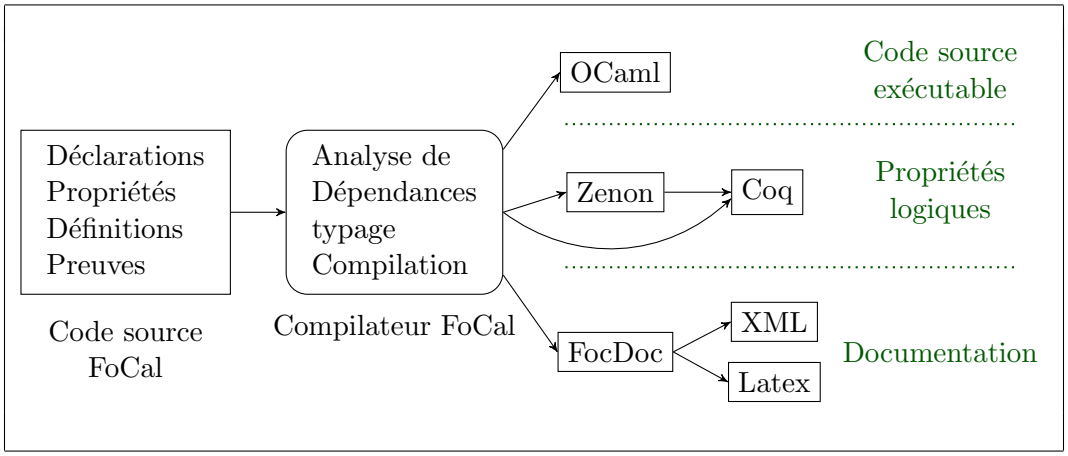

FIGURE 3.1 – Schéma de compilation d'un programme FoCal

propriétés de sécurité et de sûreté de fonctionnement.

FoCal a été utilisé avec succès dans la spécification et l'implantation de politiques de sécurité telles que Bell et LaPadula [JM06] ou la muraille de Chine ainsi que pour la formalisation et la validation d'une politique de sécurité dans les aéroports [DÉVDG06]. Pour ce dernier, le travail a consisté à trouver une formalisation définissant une politique de sécurité dans les aéroports [DÉVDG06] à partir du document papier (et informelle) la décrivant.

Nous présentons dans ce chapitre le langage FoCal informellement en déroulant un exemple. Pour plus de clarté, les notions relatives à FoCal apparaîtront les unes après les autres. Nous présenterons formellement un sous-ensemble du langage au chapitre 6. Ce dernier correspond aux parties du langage sur lesquelles nous nous fondons dans notre approche d'intégration de techniques de test dans FoCal.

3.1 Fonctionnement et philosophie

L'environnement FoCal fournit un langage de programmation pour spécifier, programmer et prouver des unités de bibliothèque. Il permet donc à la fois d'écrire des programmes, de spécifier et de prouver des propriétés. La figure 3.1 présente le schéma de compilation d'un programme FoCal. Le compilateur FoCal commence par faire des vérifications sur les unités de bibliothèque du programme. Il effectue une analyse de dépendance entre les unités, afin de vérifier si leur interaction est cohérente, et il réalise une phase de typage. Les unités de bibliothèque sont ensuite traduites par le compilateur FoCal selon différents formats : vers du code source OCaml pour aboutir à des programmes exécutables, vers un format appelé FocDoc générant des fichiers XML de documentation et vers du code source Coq à des fins de vérification formelle. L'environnement FoCal propose un langage d'expression de propriétés qui peuvent être vérifiées automatiquement et/ou prouvées interactivementToutes les preuves sont vérifiées par Coq. En outre, le langage de FoCal permet un développement modulaire par passage progressif de la spécification à l'implantation grâce aux traits objets dont il dispose (héritage, redéfinition, instanciation, ...).

Un des fondamentaux de FoCal est l'abstraction de la représentation des données. Le principe de cette approche est de voir une unité de bibliothèque comme une boîte noire qui manipule des

entités. Un type de données, comme les polynômes, peut être implanté de plusieurs manières différentes. L'abstraction du type permet de rendre interchangeables différentes implantations d'un même type de données par rapport à une interface.

3.2 Présentation du langage

3.2.1 Les espèces et les collections

À partir d'un exemple que nous déroulons de la phase de spécification à la phase d'implantation, nous présentons ici informellement les différentes facettes du langage FoCal.

La notion principale introduite par FoCal est la notion d'espèce. Il s'agit d'une forme préliminaire des unités de bibliothèque décrites plus haut. Une espèce ne fait pas l'abstraction de la représentation des données. Une espèce définit essentiellement un ensemble de valeurs et une structure algébrique sur ces valeurs. La structure algébrique sous-jacente à l'espèce est précisée par un ensemble d'opérations appelées *fonctions*, un ensemble de *propriétés* et le type des entités. Le terme *méthodes* fait référence aux fonctions et propriétés. Le type des entités est appelé « *type support* de l'espèce ». Ce type peut être abstrait (ce qui permet de rester général pour définir des structures algébriques qui ne dépendent pas de la représentation des valeurs) ou bien être défini. Comme déjà dit, les fonctions sont les opérations qui permettent de manipuler des valeurs de la structure. Elles peuvent être présentes dans l'espèce sous la forme d'une simple signature ou bien être définies. Les propriétés portent sur les opérations et permettent de préciser la structure algébrique que définit l'espèce. Lorsque l'une des opérations sur laquelle une propriété porte n'est pas définie, la propriété peut ne pas être prouvable immédiatement, la preuve de celle-ci est alors retardée.

Un programme peut comporter des propriétés qui sont juste énoncées. De telles propriétés peuvent être prouvées par la suite. Cette particularité permet d'obtenir plus de flexibilité dans le processus de développement d'un logiciel. Il est par exemple possible de découper le développement en trois phases, une phase de spécification dans laquelle les propriétés sont énoncées et admises, une phase d'implantation où les opérations sont pourvues d'une définition et enfin une phase de preuve dans laquelle l'ensemble des propriétés est prouvées. Quand l'ensemble des propriétés est prouvées, le logiciel est *certifié*. Notons qu'il est également possible d'admettre une propriété par FoCal.

Notre exemple commence par l'introduction d'une espèce qui définit un demi-treillis inférieur :

```
species demi_treillis_inf =
  rep;

  sig equal : self → self → bool;

  sig inf : self → self → self;

  property inf_commutes :
    all x y in self, equal(inf(x, y), inf(y, x));

  property inf_is_associative :
    all x y z in self, equal(inf(inf(x, y), z), inf(x, inf(y, z)));

  property inf_is_idempotent :
    all x in self, equal(inf(x, x), x);
```

```
end
```

Dans cette espèce, nous commençons par déclarer l'existence d'un type support (**rep**;). Le type support reste abstrait car l'espèce spécifie ce qu'est un demi-treillis inférieur de manière indépendante de la représentation des éléments.

Nous déclarons ensuite la fonction `equal` qui détermine si deux éléments du treillis sont égaux. Nous remarquons la présence du mot-clef **self** dans la définition du type de `equal`. Ce mot-clef désigne le type des éléments de l'espèce. Le type **self** est important dans FoCal. Les éléments de ce type sont des valeurs de l'espèce en cours de spécification. **self** permet de cacher la représentation en mémoire des valeurs de l'espèce. Par souci de simplicité, nous avons volontairement omis les propriétés usuelles attendues pour une relation d'égalité, à savoir la *réflexivité*, la *transitivité* et la *symétrie*.

Nous introduisons ensuite la loi de composition des demi-treillis inférieurs. La fonction `inf` uniquement déclarée prend en argument deux valeurs du demi-treillis inférieur et retourne la borne inférieure de ces deux éléments.

Les trois propriétés qui suivent forment la spécification de l'espèce. `inf_commutes` spécifie que `inf` est une opération commutative. `inf_is_associative` indique que `inf` est associative. Enfin la propriété `inf_is_idempotent` exprime que `inf` est une fonction idempotente. Toute implantation de `inf` devra satisfaire ces trois propriétés.

Cette espèce est une déclaration très générale des demis-treillis inférieurs. Elle ne dépend pas de la représentation des éléments du treillis. Grâce à cette espèce, on va pouvoir, par raffinement, obtenir une implantation de n'importe quel demi-treillis inférieur. Par exemple, on peut définir un type énuméré pour représenter les éléments du treillis et définir ensuite une fonction `inf` qui fixe la forme du demi-treillis.

On déclare ensuite l'espèce des demis-treillis supérieurs. Cette déclaration est identique à celle des demi-treillis inférieurs (`demi_treillis_inf`) exceptée la loi de composition qui est appelée `sup`.

```
species demi_treillis_sup =
  rep;

  sig equal : self → self → bool;

  sig sup : self → self → self;

  property sup_commutes :
    all x y in self, equal(sup(x, y), sup(y, x));

  property sup_is_associative :
    all x y z in self, equal(sup(sup(x, y), z), sup(x, sup(y, z)));

  property sup_is_idempotent :
    all x in self, equal(sup(x, x), x);

end
```

Nous pouvons définir ensuite l'espèce des treillis complets. On définit un treillis complet comme étant un demi-treillis inférieur et un demi-treillis supérieur vérifiant des propriétés supplémentaires. Pour ne pas alourdir l'exemple, nous n'avons pas montré les propriétés des ordres spécifiés. Nous obtenons donc notre espèce en utilisant le mécanisme d'héritage de FoCal :

```
species treillis inherits demi_treillis_inf, demi_treillis_sup =

  let order_inf(x, y) = equal(x, inf(x, y));

  let order_sup(x, y) = equal(x, sup(x, y));

  property inf_absorb =
    all x y in self, equal(sup(x, inf(x, y)), x);

  property sup_absorb =
    all x y in self, equal(inf(x, sup(x, y)), x);

  property inf_distrib_sup =
    all x y z in self, equal(sup(inf(x,y),z), inf(sup(x,z), sup(y,z)));

  property sup_distrib_inf =
    all x y z in self, equal(inf(sub(x,y),z), sup(inf(x,z), inf(y,z)));

end
```

La structure des treillis complets est définie par héritage à partir des deux espèces précédentes `demi_treillis_inf` et `demi_treillis_sup`. Cela signifie que l'espèce `treillis` contient l'ensemble des méthodes des espèces parentes. Les déclarations de `sup` et de `inf` sont « recopiées » dans `treillis`. Il en est de même pour les propriétés qui leur sont associées.

Les deux espèces demi-treillis contiennent toutes les deux la déclaration de la fonction `equal` ainsi que des propriétés qui s'y rapportent. Si la fonction `equal` avait été définie dans les deux espèces parentes, FoCal aurait gardé la définition de l'espèce la plus à droite dans la liste des espèces héritées. Dans ce cas, `treillis` aurait contenu la définition de `equal` de `demi_treillis_sup`. Ici, la fonction `equal` est uniquement déclarée, FoCal ne fait que vérifier que les deux types dans les déclarations sont compatibles. Deux déclarations sont compatibles si leur type est unifiable, le type de `equal` dans `treillis` est le résultat de l'unification des deux types. FoCal fait aussi une vérification de compatibilité des propriétés, elles doivent avoir le même énoncé dans les deux espèces parentes. La surcharge est interdite dans FoCal, on ne peut pas redéfinir/hériter une fonction en changeant son type.

Dans l'espèce `treillis`, nous avons aussi ajouté les propriétés usuelles des treillis, à savoir la distributivité entre les deux lois de compositions `inf` et `sup` et les propriétés d'absorption.

Nous ajoutons aussi la définition des deux ordres `order_inf` et `order_sup` induits par respectivement `inf` et `sup`.

Nous pouvons donner la définition de `order_inf` et `order_sup` bien que les fonctions `sup` et `inf` ne soient connues uniquement que par leur signature grâce au mécanisme de *liaison retardée* de FoCal. Les définitions des fonctions `sup` et `inf` seront données dans un raffinement ultérieur de `treillis`. Ces deux liaisons retardées rendent les fonctions `order_sup` et `order_inf` non exécutables tant que `sup` et `inf` ne sont pas définies.

Maintenant que nous avons défini une espèce abstraite représentant un treillis complet, nous pouvons, par raffinement, préciser le type des données sur lesquelles portent les treillis. Toujours en utilisant l'héritage, nous construisons une nouvelle espèce à partir de `treillis` et donnons la définition du type support. Ceci est fait en associant un type au mot-clef **rep**. Ainsi, le mot-clef **self**, que nous avons introduit tout à l'heure, fait désormais référence au type défini par **rep**.

```
species treillis_entiers inherits treillis =
  rep = int;
```

```
  let inf(x, y) = #int_min(x, y);
  let sup(x, y) = #int_max(x, y);
  let equal(x, y) = #int_eq(x, y);

  let from_int(x in int) in self = x;
  [...]
  theorem exemple :
        all x y z in self, equal(inf(sup(x, x), sup(x, y)), x)
  proof:
    <1>1 assume x y z in self
       prove equal(inf(sup(x, x), sup(x, y)), sup(x, inf(x, y)))
       by sup_distrib_inf, equal_symmetric
    <1>2 assume x y z in self
       prove equal(sup(x, inf(x, y)), x)
       by inf_absorb
    <1>3 qed by <1>1, <1>2, equal_transitive;
  [...]
end
```

Dans cette espèce, le type support est défini et vaut **int**. La forme du treillis est ensuite donnée par la définition de `inf`, `sup` et `equal`. Les définitions de fonction ont une syntaxe proche de OCaml. Dans l'exemple, `inf` est la fonction prédéfinie sur les entiers `int_min`. Le signe # est là pour spécifier que la fonction est prédéfinie dans FoCal. La fonction `sup` a pour définition la fonction prédéfinie `int_max` et la fonction `equal` est l'égalité des entiers.

Pour la fonction `from_int` nous avons utilisé des annotations de type introduites par le mot-clef **in**. FoCal est pourvu d'une inférence de types. Toutefois, nous sommes obligés dans ce contexte de donner le type de l'argument de la fonction et de la valeur retournée. Le type de `from_int` est forcé à **int** $\rightarrow$ **self**, cela est fait grâce au mot-clef **in**.

Cette fonction est l'identité et doit permettre de faire la traduction entre le type **int** et **self**. Avec une seule des deux annotations, le type inféré pour `from_int` serait soit **int** $\rightarrow$ **int** soit **self** $\rightarrow$ **self**. C'est pour cela que l'annotation est obligatoire. Bien sûr, la compatibilité entre **int** et **self** est ici exploitée.

Une fois `inf` et `sup` définis, on donne une preuve de chacune des propriétés de l'espèce. Nous montrons ici une propriété qui est prouvée à l'aide de `sup_distrib_inf`, `inf_absorb` et de la symétrie de l'égalité. Cette preuve est traitée par le prouveur automatique Zenon [BDD07]. Une preuve Zenon est organisée comme un ensemble d'indications qui permet de reconstruire la preuve. Nous n'expliquons pas la preuve montrée dans l'exemple puisqu'il dans la suite nous n'utiliserons pas de telles preuve. Cette preuve est montrée dans le but de donner une idée de la forme des preuves Zenon.

Nous constatons que toutes les fonctions propriétés de `treillis_entiers` sont définies et démontrées. De plus le type support est défini. Une telle espèce est appelée *espèce complète*. Il faut préciser qu'une espèce, même complète, est un composant qui n'est pas exécutable. Pour obtenir le type défini par `treillis_entiers`, il faut créer une collection. La collection permet d'obtenir la structure algébrique sous-jacente à l'espèce `treillis_entiers` :

```
collection z_treillis implements treillis_entiers;;
```

La collection `z_treillis` est un nouveau type de données utilisable dans la suite du programme.

Le mot-clef **implements** signifie qu'on veut définir la collection `z_treillis` qui a la même interface que l'espèce `treillis_entiers`. L'interface d'une collection est la donnée de l'ensemble des noms de fonctions qu'elle contient ainsi que leur signature. Les interfaces sont utilisées

dans le contexte de paramétrisation pour vérifier que les paramètres effectifs respectent bien les pré-requis spécifiés par le paramètre formel. Ceci est détaillé dans la section 3.2.2.

La collection `z_treillis` reprend la signature de chaque fonction issue de `treillis_entiers` en prenant soin de remplacer toutes les occurrences de **self** dans leur type par le nouveau type défini : `z_treillis`. Ce mécanisme est l'abstraction du type support. Par exemple, dans la collection `z_treillis`, la fonction `inf` a pour type `z_treillis` → `z_treillis` → `z_treillis`.

Plus précisément, `z_treillis` est un type abstrait de données, ce qui signifie que l'utilisateur du type n'a pas accès à la représentation des éléments du type. L'unique moyen de créer, de manipuler et de faire du calcul sur des éléments d'un type abstrait de données est d'utiliser les fonctions fournies.

Les collections permettent d'assurer des propriétés sur les valeurs possibles du type. Par exemple, on peut assurer que les éléments d'une collection qu'il est possible de créer respectent un invariant de représentation si on prouve que chaque fonction de l'espèce retourne de tels éléments.

Nous pouvons maintenant utiliser la collection `z_treillis` comme ceci :

```
z_treillis!inf(z_treillis!from_int(18), z_treillis!from_int(5));;
```

La notation `z_treillis!inf` permet de préciser qu'on veut utiliser la fonction `inf` de la collection `treillis`. Il faut remarquer qu'on ne peut pas donner directement la valeur 18 à la fonction `inf`. L'abstraction du type support a rendu les types `z_treillis` et **int** incompatibles. Ainsi, l'utilisation suivante de la collection `z_treillis` n'est pas correcte :

```
z_treillis!inf(18, 5);;
```

Avant de passer à la suite, nous devons ajouter que FoCal offre à l'utilisateur la possibilité de créer des définitions de propriété dans le même style que les définitions de méthode.

```
species modulo_10 =
  ...
  letprop normalise(x) = x < 10 and x >= 0;
  ...
end
```

Dans l'exemple précédent, on donne la définition `normalise` qui indique que son paramètre est un entier qui doit être dans l'intervalle $[\![0, 9]\!]$. Cette définition est comparable aux macros du langage C. `normalise` n'est ni une fonction de l'espèce ni une propriété. Elle n'est pas utilisable à l'intérieur d'une fonction. En revanche, elle peut être utilisée au sein d'une propriété et est remplacée par sa définition. Les constructions **letprop** peuvent servir par exemple pour définir un invariant de représentation des éléments du type support. Dans le cas de l'espèce `modulo_10`, de telles propriétés peuvent avoir la forme **all** `x y` **in self**, `normalise(x)` → `normalise(y)` → ....

3.2.2 Paramétrisation des espèces

FoCal permet de paramétrer les espèces par des collections. Nous pouvons grâce au paramétrage définir, par exemple, le produit direct de deux treillis à partir de l'espèce `treillis` de la section précédente :

```
species treillis_produit (t1 is treillis, t2 is treillis)
                                          inherits treillis =
  rep = t1 * t2;

  let inf(x, y) = let (x_g, x_d) = x in
                  let (y_g, y_d) = y in
```

```
                    #crp(t1!inf(x_g, y_g),
                         t2!inf(x_d, y_d));

  let sup(x, y) = ...
  [...]
end
```

Le produit de deux treillis est encore un treillis. Par conséquent, nous définissons l'espèce `treillis_produit` par héritage de l'espèce `treillis` et qui contient donc l'ensemble des signatures et définitions relatives aux treillis.

Elle prend en paramètre `t1` et `t2` qui sont deux collections dont l'interface doit être compatible avec celle des treillis. Intuitivement, cela signifie que les paramètres effectifs de l'espèce `treillis_produit` doivent contenir au minimum un ensemble de méthodes ayant les mêmes noms et les mêmes signatures (en remplaçant les occurrences de `treillis` par `t1` respectivement `t2` dans les types) de `treillis`. Il n'est pas exclu que les paramètres effectifs possèdent des méthodes supplémentaires.

Bien que ces deux collections doivent être compatibles avec une même interface, elles ne sont pas interchangeables pour autant. Il ne faut pas oublier que pendant la phase d'abstraction du type support, les occurrences de **self** dans les signatures des méthodes sont remplacées par le nom de la collection, ici respectivement `t1` et `t2`. `t1` et `t2` sont donc des types différents. Ces paramètres ont pour caractéristique d'être définis uniquement par une interface de collection. Ils sont appelés *paramètres collections*.

Nous définissons ensuite le type support de l'espèce par le produit de `t1` et `t2`, soit le type des couples dont la première composante est une valeur de `t1` et la deuxième composante une valeur de `t2`. Le constructeur de couple est l'opérateur `#crp`.

Une fois le type support défini, nous pouvons définir les deux lois de composition du treillis produit en fonction des lois de `t1` et `t2`. Les propriétés portant sur les lois sont ensuite prouvées, elles ne sont pas explicitées ici.

À partir de l'espèce `treillis_prod`, nous pouvons définir le treillis des couples d'entiers, par la collection `zz_treillis_prod`, en instanciant les deux paramètres par la collection `z_treillis`. Nous le faisons car leur interface est compatible avec l'interface requise dans la définition donnée dans `treillis_prod`.

```
collection zz_treillis_prod implements
                              treillis_produit(z_treillis, z_treillis);;
```

Le langage FoCal définit une autre sorte de paramètre, *les paramètres entités*. Contrairement aux paramètres collections qui sont des types abstraits à l'intérieur d'une espèce, les paramètres entités sont des valeurs de collections qui ont une signification particulière.

```
species treillis_entiers_min_max(t is treillis_entiers, min in t,
                                 max in t) inherits treillis =
  rep = t;

  let min in self = min;
  let max in self = max;
  let normalise(x) = if  t!order_inf(x, min) then
                       min
                     else if t!order_sup(x, max) then
                       max
                     else
                       x;
  let inf(x, y) = normalise(t!inf(x, y));
```

```
  [...]
end
```

Dans cet exemple, nous définissons l'espèce des treillis bornés sur les entiers, c'est-à-dire, une espèce dans laquelle la borne inférieure (resp. la borne supérieure) est représentée par la valeur entière donnée en paramètre. Les paramètres `min` et `max` représentent les deux bornes et sont des valeurs de la collection `t`.

Nous pouvons maintenant définir la collection des treillis entiers à 16 éléments comme suit :

```
collection z_5_treillis implements
                    treillis_entiers_min_max(z_treillis,
                                             z_treillis!from_int(7),
                                             z_treillis!from_int(22));
```

3.3 Synthèse

Dans ce chapitre nous avons fait une présentation informelle du langage FoCal. Le but était de donner au lecteur une idée assez générale des traits de FoCal, en particulier des mécanismes d'abstraction, d'héritage et de paramétrisation. Nous avons illustré sur l'exemple des treillis l'ensemble de ces fonctionnalités. On trouvera dans le chapitre 6 une présentation plus formelle de ce dernier.

Le mécanisme d'abstraction du type support est une des notions les plus importantes de FoCal. Il offre les garanties qui permettent de rendre les collections modulaires et de substituer une collection à une autre.

Dans le chapitre suivant, nous introduisons les notions de base qui forment la programmation par contraintes.

Chapitre 4

Programmation par contraintes

Ce chapitre sert de brève introduction à la programmation par contraintes. Nous y définissons la problématique de la satisfaction de contraintes ainsi que les mécanismes mis en œuvre pour résoudre ces problèmes. Nous présentons ensuite les méta-contraintes qui seront utilisées par la suite pour faire la traduction des programmes FoCal en contraintes.

4.1 Problèmes de satisfaction de contraintes

Le problème de satisfaction de contraintes (ou CSP) est un modèle générique pour représenter des problèmes qui s'expriment sous la forme d'un ensemble fini de contraintes à satisfaire. Une solution d'un problème modélisé par un CSP est une solution du CSP.

Les objets de base d'un CSP sont les variables. Chaque variable du système est associée à un domaine qui définit un ensemble de valeurs que peut prendre la variable. Les contraintes d'un CSP sont concrétisées par un ensemble de relations sur les variables. Intuitivement, une solution d'un CSP affecte à chaque variable une valeur et est telle que les contraintes du système sont vérifiées.

Définition 4.1.1 Problème de satisfaction de contraintes**.**
Soit un ensemble de relations $\mathcal{R}$. Un problème de satisfaction de contraintes est un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ pour lequel on a :

- *$\mathcal{X}$ est un ensemble de variables. Il s'agit de l'ensemble des variables du problème. On note $X_i \in \mathcal{X}$ la i^evariable de $\mathcal{X}$;*
- *$\mathcal{D}$ est une fonction qui associe à chaque variable $X_i \in \mathcal{X}$ son domaine de définition, noté $\mathcal{D}(X_i)$. Dans la suite, on note $\mathcal{D}_i = \mathcal{D}(X_i)$ le domaine associé à la variable X_i par $\mathcal{D}$;*
- *$\mathcal{C}$ est l'ensemble des contraintes du problème de satisfaction de contraintes. Ces contraintes portent sur un sous-ensemble des variables de $\mathcal{X}$. Une contrainte C_i est définie par $C_i \equiv R_i(X_{i_1}, \ldots, X_{i_n})$ où R_i est une relation de $\mathcal{R}$, n est l'arité de R_i et les variables X_{i_k} sont des variables de $\mathcal{X}$.*

Exemple 4.1.2. Le célèbre problème des dames consiste à placer sur un échiquier, de la forme d'un carré de taille $n * n$ pour un entier n fixé, n dames (qui peuvent se déplacer selon les règles dictées par les échecs) de sorte qu'aucune dame ne soit en prise avec aucune autre dame de l'échiquier. Une solution de ce problème est alors un placement des dames de sorte qu'il y ait une seule dame posée dans chaque ligne, chaque colonne et chaque diagonale de l'échiquier.

Nous pouvons modéliser le problème des dames d'un échiquier de taille 4 avec un CSP

de plusieurs manières différentes. Nous présentons une version dans laquelle chaque variable représente une colonne et prend pour valeur la ligne dans laquelle la dame de cette colonne se place.

$$\begin{array}{l} \mathcal{X} = \{X_1, X_2, X_3, X_4\} \\ \mathcal{D}_1 = \mathcal{D}_2 = \mathcal{D}_3 = \{1, 2, 3, 4\} \\ \mathcal{C} = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7\} \text{ avec} \\ \quad C_1 \equiv \mathit{AllDiff}(X_1, X_2, X_3, X_4) \\ \quad C_2 \equiv |X_1 - X_2| \neq 1 \qquad C_3 \equiv |X_1 - X_3| \neq 2 \\ \quad C_4 \equiv |X_1 - X_4| \neq 3 \qquad C_5 \equiv |X_2 - X_3| \neq 1 \\ \quad C_6 \equiv |X_2 - X_4| \neq 2 \qquad C_7 \equiv |X_3 - X_4| \neq 1 \end{array}$$

AllDiff est la contrainte qui impose que toutes les variables prennent des valeurs différentes. Les autres contraintes expriment que chaque dame n'est pas sur une même diagonale que les trois autres.

4.1.1 Solutions d'un problème de satisfaction de contraintes

Comme déjà dit, le but d'un problème de satisfaction de contraintes est d'exprimer un problème sous la forme d'un ensemble de contraintes à satisfaire. Nous devons avant toute chose donner une définition formelle à chacune des notions nécessaires pour définir une solution d'un CSP.

Définition 4.1.3 Affectation**.**
Pour un CSP donné, une affectation *est une fonction $\mathcal{A}$ qui associe à certaines variables du CSP une unique valeur de son domaine. On définit les affectations par la syntaxe suivante :*

$$\begin{array}{rcl} \mathcal{A} & ::= & \emptyset \\ & | & \mathcal{A}, (X, v) \end{array}$$

On note $\mathcal{A}(X) = v$ si on a $(X, v) \in \mathcal{A}$.

On raffine la définition d'une affectation en fonction de l'ensemble des variables qu'elle définit. On obtient alors les définitions suivantes.

Définition 4.1.4 Affectation partielle/totale**.**
Pour un CSP donné, une affectation est partielle *si elle définit la valeur d'un sous-ensemble strict des variables du système. Une affectation qui définit les valeurs de l'ensemble des variables du système est dite* totale*.*

Nous donnons maintenant les définitions utilisées pour définir une solution d'un CSP.

Définition 4.1.5 Violation de contrainte**.**
Soit un CSP. Une affectation $\mathcal{A}$ viole *la contrainte C_k si toutes les variables de C_k sont définies par $\mathcal{A}$ et si la relation de C_k n'est pas vérifiée.*

Exemple 4.1.6. Reprenons l'exemple 4.1.2 du problème des dames sur un échiquier de taille 3. On définit les affectations :

– $\mathcal{A}_1 = (X_1, 1), (X_3, 3)$

– $\mathcal{A}_2 = (X_1, 3), (X_2, 1), (X_3, 2), (X_4, 4)$

$\mathcal{A}_1$ est une affectation *partielle* car la variable X_2 n'y est pas définie. Au contraire, $\mathcal{A}_2$ est une affectation *totale*. L'affectation $\mathcal{A}_1$ viole la contrainte C_3 car on a $|\mathcal{A}_2(X_1) - \mathcal{A}_2(X_3)| = 2$. Pour des raisons similaires, $\mathcal{A}_2$ viole la contrainte C_5 mais ne viole ni la contrainte C_2 ni la contrainte C_4.

Définition 4.1.7 Affectation consistante**.**
Une affectation $\mathcal{A}$ est consistante *par rapport à un CSP si elle ne viole aucune contrainte. À l'inverse, une affectation $\mathcal{A}$ est* inconsistante *si elle viole au moins une contrainte.*

Nous pouvons maintenant donner la définition d'une solution d'un CSP.

Définition 4.1.8 Solution d'un CSP**.**
Une affectation est une solution *d'un CSP si elle est* totale *et* consistante*.*

Exemple 4.1.9. Pour l'exemple 4.1.2, $\mathcal{A} = (X_1, 2), (X_2, 4), (X_3, 1), (X_4, 3)$ est une solution du système.

4.1.2 Résolution d'un CSP

Une méthode de recherche de solutions d'un CSP consiste à énumérer, pour chaque variable, l'ensemble des valeurs de son domaine et à vérifier la consistance de chaque affectation obtenue. Cette approche par énumération exhaustive et systématique n'est pas judicieuse dans le sens où un très grand nombre d'affectations inconsistantes est créé avant de trouver une solution. Avec des domaines très larges, cette énumération devient infaisable en pratique.

Pour accélérer les méthodes de recherche de solutions d'un CSP, nous devons raffiner la notion de consistance et permettre de détecter l'absence de solution à partir d'une détermination partielle des variables.

Consistance partielle

Nous présentons ici les principaux tests de consistance partielle utilisés dans la résolution d'un CSP.

Définition 4.1.10 Inconsistance d'un CSP**.**
Un CSP est inconsistant *si et seulement si l'une de ses variables a un domaine vide.*

Nous voudrions utiliser la notion d'inconsistance pour détecter la présence ou l'absence de solutions d'un CSP. L'inconsistance en tant que telle n'est pas suffisante pour cela car en règle générale lorsqu'on définit un CSP on donne aux variables du CSP des domaines qui ne sont pas vides. En d'autres termes les CSP que nous définissons ne sont généralement pas inconsistants mais peuvent pourtant ne pas admettre de solution.

Théorème 4.1.11. *Un CSP* inconsistant *n'admet pas de solution.*

Démonstration. Dans un CSP inconsistant, le domaine d'au moins une variable est vide. Il n'y a donc pas d'affectation totale possible et pas de solution. □

Dans la résolution d'un CSP, l'action la plus utilisée est la transformation d'un CSP en un autre CSP qui admet les mêmes solutions. Dans cette optique, la dernière définition devient une condition suffisante pour détecter l'absence de solution d'un CSP après l'application d'une suite de transformations. Les transformations d'un CSP se fondent sur la définition d'extension de CSP.

Définition 4.1.12 Extension de CSP**.**
Soient deux CSPs σ et σ'. σ' est une extension *de σ si et seulement si les ensembles des contraintes de σ et σ' portent sur les mêmes variables, sont identiques et que pour toute variable X_i des deux CSPs on a $\mathcal{D}_i(\sigma') \subseteq \mathcal{D}_i(\sigma)$.*

À cette définition, nous adjoignons les définitions de consistances entre une contrainte et un CSP.

Définition 4.1.13 Consistance de domaine**.**
Une contrainte C est domaine-consistante *par rapport à un CSP $\sigma = (\{X_1, \ldots, X_n\}, \mathcal{D}, \mathcal{C})$ si pour chaque variable X_i et chaque valeur $v_i \in \mathcal{D}_i$, il existe un (n-1)-uplet de valeurs (v_1, …, v_{i-1}, v_{i+1}, …, v_n), avec $v_k \in \mathcal{D}_k$ pour $k \in [\![1, n]\!]$ tel que C est vérifiée.*
Par extension, un CSP σ est domaine-consistant *si chaque contrainte C de σ est* domaine-consistante*.*

Exemple 4.1.14. Sur l'exemple des dames 4.1.2. Si on considère les domaines suivants :

$$\mathcal{D}_1 = \{1, 4\}$$
$$\mathcal{D}_2 = \{1, 3\}$$
$$\mathcal{D}_3 = \{1, 4\}$$
$$\mathcal{D}_4 = \{1, 2, 3, 4\}$$

On constate que la contrainte C_6 ($|X_2 - X_4| \neq 2$) est domaine-consistante car pour les valeurs 1 et 3 de X_2 nous trouvons respectivement les valeurs 2 et 3 pour X_4 qui vérifient C_6 et pour les quatre valeurs 1, 2, 3 et 4 de X_4 nous pouvons respectivement donner 1 1 3 et 3 pour X_2 qui vérifie la contrainte.
De plus, le CSP est domaine-consistant car toutes les contraintes sont domaines consistantes.

La consistance de domaine est un test partiel de consistance car il n'assure pas que le CSP contient une solution. Il s'agit d'un test purement local qui n'examine qu'une contrainte à la fois. Il vérifie, pour chaque contrainte prise à part, si toutes les valeurs du domaine des variables sur lesquelles elle porte, permettent de satisfaire la contrainte.

Une deuxième forme de consistance est possible dans le cas où les variables portent sur des valeurs entières. Il s'agit dans ce cas de ne pas considérer le domaine exact de chaque variable, mais une approximation de ce domaine par un intervalle. La consistance d'intervalle est un test de consistance pour lequel on considère le plus petit intervalle qui englobe le domaine des variables. Pour un domaine $\mathcal{D}$, on définit le domaine $\mathcal{D}^*$ par $\mathcal{D}^* = \{x \mid \min(\mathcal{D}) \leq x \leq \max(\mathcal{D})\}$.

Définition 4.1.15 Consistance d'intervalle**.**
Une contrainte C est intervalle-consistante *par rapport à un CSP σ, si pour chaque variable X_i et pour chaque valeur $v_i \in \{\min(\mathcal{D}_i), \max(\mathcal{D}_i)\}$, il existe des valeurs $v_k \in \mathcal{D}_k^*$ pour $k \neq i$ telles que le (n-1)-uplet $(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)$ vérifie les contraintes C.*

Exemple 4.1.16. Si on reprend l'exemple 4.1.14, on constate que le CSP est intervalle consistant.

La définition de la consistance d'intervalle peut sembler inutile par rapport à la consistance de domaine car elle considère un sur-ensemble du domaine des variables. Elle demande donc d'évaluer la contrainte sur plus de valeurs que nécessaire. Toutefois, cette approximation des domaines entiers permet d'utiliser directement l'arithmétique des intervalles pour vérifier la consistance sur un ensemble de valeurs.

Filtrage

Les filtrages sont les extensions naturelles des tests de consistance. Il s'agit de supprimer du domaine des variables les valeurs qui ne font pas partie de la solution du système de contraintes. À l'instar des tests de consistance, les algorithmes de filtrage sont partiels. Ils ne suppriment pas l'ensemble des valeurs ne faisant pas partie des solutions mais seulement un sous-ensemble. L'ensemble des valeurs supprimées est déduit des tests de consistance.

Définition 4.1.17 Filtrage par domaine-consistance**.**
Soit un CSP σ. Le filtrage *par domaine consistance de σ est un CSP σ' tel que*
- *L'ensemble des solutions de σ et l'ensemble des solutions de σ' sont les mêmes ;*
- *σ' est domaine-consistant ;*
- *σ' est une extension de σ. De plus, toute extension σ'' domaine-consistante de σ est une extension de σ'.*

On peut définir un filtrage pour chaque test de consistance. En particulier, on peut définir le filtrage par intervalle-consistance en utilisant une définition similaire à la précédente. Les filtrages résultants suppriment le nombre minimal de valeurs dans les domaines de sorte à obtenir la consistance correspondante.

Résolution

Nous pouvons maintenant définir le processus général de résolution d'un CSP. Nous traitons ici la résolution des CSPs dans le cas où le domaine des variables est fini.

La résolution d'un CSP s'effectue en alternant une phase de réduction de domaine et une phase de détermination d'une variable. La réduction de domaine est effectuée par un algorithme de filtrage. Nous donnons dans la suite un schéma de l'algorithme généralement utilisé pour trouver une solution.

Définition 4.1.18.
Soit un CSP σ. La résolution de σ s'effectue en appliquant l'algorithme suivant :
1. *On choisit une variable X de σ dont le domaine n'est pas un singleton et une valeur v de son domaine. On modifie le domaine de X en le restreignant au singleton $\{v\}$. ;*
2. *Le CSP σ est remplacé par CSP σ' tel que σ' est obtenu en appliquant un algorithme de filtrage ;*
3. *S'il existe une variable avec un domaine vide, on retourne à la dernière fois où l'on a appliqué l'étape 1, on retire du domaine la valeur choisie précédemment et on reprend une autre valeur (on* backtrack*).*
4. *S'il existe des variables dont le domaine n'est pas un singleton, on retourne à l'étape 1.*

À la fin de cet algorithme, on se retrouve avec un CSP dont toutes les variables ont un singleton pour domaine s'il existe une solution. Il s'agit d'une solution du CSP. Le CSP peut, bien entendu, posséder d'autres solutions.

On remarque que des libertés sont laissées dans l'implantation de cet algorithme :

Dans l'étape 1, il n'est pas précisé quelle variable choisir. Il existe différentes heuristiques dans le choix de cette variable. On peut choisir la variable qui possède le plus petit domaine (*first-fail*) ou le plus grand domaine, la variable qui est présente dans le plus grand/petit nombre de contraintes ...

En ce qui concerne le choix de la valeur dans le domaine il existe là encore des heuristiques classiques : choisir la valeur la plus petite/grande, balayer le domaine de la variable en partant du milieu... Une autre approche consiste à ne pas réduire le domaine de la variable en un singleton mais en un sous-ensemble du domaine initial et lors d'un *backtrack* de choisir un autre sous-ensemble.

L'algorithme de filtrage à utiliser n'est pas non plus spécifié. Tous les algorithmes de filtrage existants peuvent être utilisés. On peut aussi ne pas utiliser d'algorithme de filtrage et passer directement à l'étape de détermination d'une autre variable.

Pour chacun des choix à faire dans l'implantation de la résolution d'un CSP, il n'y a pas d'heuristique qui soit meilleure qu'une autre. L'efficacité d'une heuristique dépend du problème à résoudre. Certaines heuristiques sont meilleures dans des classes de problèmes bien précises et moins bonnes dans d'autres classes. C'est pour cette raison que, généralement, les solveurs de contraintes laissent la possibilité à l'utilisateur de choisir l'heuristique qu'il veut utiliser pour résoudre les contraintes soumises.

Une autre manière de représenter la résolution d'un CSP est de la présenter sous la forme d'un arbre. Les nœuds de l'arbre sont les variables du CSP et pour chaque nœud, il y a autant de fils qu'il y a de valeurs dans son domaine. Chaque fils représente la réduction du domaine de la variable au singleton correspondant. La résolution d'un système de contraintes revient alors à explorer cet arbre.

Exemple 4.1.19. Reprenons l'exemple 4.1.2 et déroulons l'algorithme présenté pour obtenir une solution. On choisit de prendre les variables dans l'ordre présenté dans l'exemple : X_1, X_2, X_3, X_4. À chaque étape de détermination on prend la valeur la plus petite dans son domaine et nous utilisons le filtrage par domaine. À chaque étape de filtrage, nous indiquons entre parenthèses le nom des contraintes qui ont réduit le domaine.

1. On prend la variable X_1 et pour valeur 1. Le filtrage réduit les domaines comme ceci :
   $\mathcal{D}_1 = \{1\}$
   $\mathcal{D}_2 = \{3, 4\}$ (par C_1 et C_2)
   $\mathcal{D}_3 = \{2, 4\}$ (par C_1 et C_3)
   $\mathcal{D}_4 = \{2, 3\}$ (par C_1 et C_4)
   (a) On prend la variable X_2 et la valeur 3. On obtient :
   $\mathcal{D}_1 = \{1\}$
   $\mathcal{D}_2 = \{3\}$
   $\mathcal{D}_3 = \emptyset$ (par C_5)
   $\mathcal{D}_4 = \{2\}$ (par C_1)
   Le domaine de X_3 est vide on reprend donc une autre valeur pour X_2.
   (b) On prend la variable X_2 et la valeur 4. On obtient :
   $\mathcal{D}_1 = \{1\}$

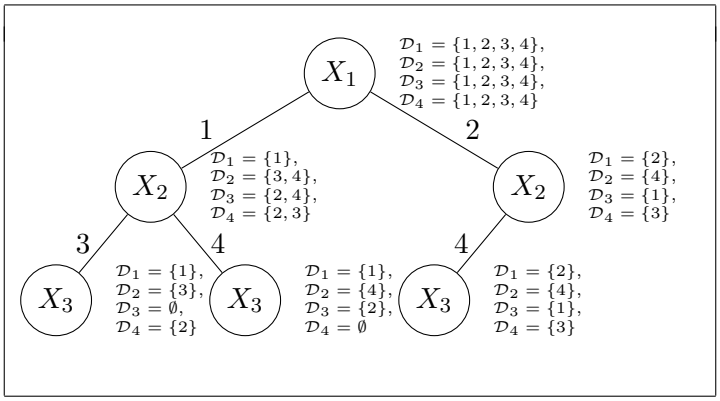

FIGURE 4.1 – Arbre de résolution du problème des dames de taille 4

$\mathcal{D}_2 = \{4\}$
$\mathcal{D}_3 = \{2\}$ (par C_1 et C_5)
$\mathcal{D}_4 = \emptyset$ (par C_1 et C_7)
Le domaine de X_4 est vide. Comme on a épuisé les valeurs possibles pour X_2 on reprend une autre valeur pour X_1.

2. On prend la variable X_1 et la valeur 2. On obtient après filtrage des domaines :
   $\mathcal{D}_1 = \{2\}$
   $\mathcal{D}_2 = \{4\}$ (par C_1 et C_2)
   $\mathcal{D}_3 = \{1\}$ (par C_1 et C_5)
   $\mathcal{D}_4 = \{3\}$ (par C_1) Ce qui donne une solution.

L'arbre correspondant à cette résolution est présenté dans la figure 4.1.

4.2 Méta-contrainte

Les contraintes que nous avons vues jusqu'à maintenant étaient définies comme une relation entre des variables. Les contraintes ainsi définies n'évoluent pas au fur et à mesure de la résolution du système. Le principe de résolution défini dans la section précédente transforme un CSP en ne modifiant pas les contraintes du système. Les contraintes sont *statiques*. Pour étendre l'expressivité des contraintes nous avons besoin de définir des contraintes spéciales qui évoluent au fur et à mesure de la résolution.

Nous appelons une telle contrainte une *méta-contrainte*. Les méta-contraintes ne sont donc pas définies par des relations entre des variables. Plus précisément, les méta-contraintes agissent comme des générateurs de contraintes. Lorsqu'une certaine action ou condition est vérifiée par le CSP, une méta-contrainte est transformée en un ensemble de contraintes (ces contraintes viennent remplacer la méta-contrainte dans le CSP). On remarque que l'ensemble des contraintes résultant de la méta-contrainte peut contenir des méta-contraintes ou peut être vide. Dans ce dernier cas, la méta-contrainte est effacée du CSP.

4.2.1 Test d'implication

Le principal mécanisme de transformation des méta-contraintes est le test d'implication d'une contrainte par un CSP. Le test d'implication consiste à chercher si une contrainte est vérifiée pour toutes les solutions d'un ensemble de contraintes ou non.

Définition 4.2.1 Implication**.**
Soient un CSP σ et une contrainte C. C est impliquée par le CSP σ si et seulement si toute solution de σ satisfait la contrainte C.

Il est clair que dans la pratique, la détection de l'implication d'une contrainte ne peut pas être complète. Il est nécessaire de connaître toutes les solutions d'un CSP. C'est pour cela que, de la même manière que pour le filtrage des domaines, il a été défini plusieurs tests de détection partiels de l'implication.

Définition 4.2.2 Domaine-implication**.**
Soient un CSP σ et une contrainte C. C est domaine-impliquée par σ si et seulement si pour toute affectation de σ, C est vérifiée.

Comme pour le filtrage, lorsque les variables sont sur un domaine entier, on peut donner une définition de l'implication fondée sur les intervalles.

Définition 4.2.3 Intervalle-implication**.**
Soient un CSP σ et une contrainte C. C est intervalle-impliquée par σ si et seulement si pour toute valeur $(v_1, \ldots, v_n)$ pour les variables $(X_1, \ldots, X_n)$ telle que $v_i \in \mathcal{D}^*(X_i)$, la contrainte C est vérifiée.

L'implication est l'opération qui permet de construire le cœur du mécanisme de fonctionnement des méta-contraintes que sont les contraintes gardées. Les contraintes gardées sont utilisées principalement pour bloquer l'action d'une contrainte et la rendre ainsi inopérante jusqu'à ce qu'une garde devienne impliquée par le CSP.

Définition 4.2.4 Contrainte gardée**.**
Une contrainte gardée est une contrainte de la forme $C_1 \rightarrow C_2$. Soient un CSP σ (qui comprend un ensemble de contraintes et des domaines pour les variables) et une contrainte $C_1 \rightarrow C_2$ de σ. Le traitement de $C_1 \rightarrow C_2$ est le suivant :

1. *si C_1 est impliquée par σ alors la contrainte gardée $C_1 \rightarrow C_2$ est retirée de σ et C_2 y est ajoutée ;*
2. *si $\neg C_1$ est impliquée par σ la contrainte gardée est supprimée de σ ;*
3. *si aucun des cas précédents n'est vérifié, la contrainte gardée est mise en attente d'un traitement futur.*

On remarque que, dans la définition précédente, il n'est pas précisé quel type d'implication utiliser (domaine-implication ...). Ce choix influe sur la capacité du système à réduire les contraintes gardées, sur le temps passé à faire les tests d'implication et sur le temps global de résolution d'un CSP. La réduction au plus tôt des contraintes gardées peut accélérer de manière significative la recherche d'une solution du système. Au contraire, le temps consacré au test d'implication d'une contrainte peut augmenter le temps de recherche d'une solution.

Les méta-contraintes sont définies par un ensemble de contraintes gardées. Il s'agit d'une généralisation des contraintes-gardées en présence de plusieurs gardes.

Définition 4.2.5 Méta-contrainte**.**
Une méta-contrainte C est la donnée d'un ensemble de contraintes gardées :

1. $C_1 \rightarrow C'_1$
2. $C_2 \rightarrow C'_2$

$\vdots$

n. $C_n \rightarrow C'_n$

Dans un CSP σ (qui comprend toujours un ensemble de contraintes et des domaines pour les variables), elle a la sémantique suivante :

1. *si la contrainte C_i est impliquée par σ alors la méta-contrainte C est transformée en C'_i ;*
2. *si la contrainte $\neg C_i$ est impliquée par σ alors la contrainte gardée i est retirée de C ;*
3. *sinon C est mise en attente d'un autre test d'implication.*

On remarque que dans certains cas il est utile de ne pas poser de contraintes après la détection d'une implication. Cela est utilisé, par exemple, lorsqu'on a détecté l'implication d'une condition nécessaire et suffisante qui signifie que la méta-contrainte n'a plus aucun effet dans le système de contraintes. Une telle règle doit alors supprimer la méta-contrainte. Pour définir ces règles, on donne dans ce cas, pour valeur de C' la contrainte `true` qui est la contrainte vraie dans tout contexte.

De par leur définition, les méta-contraintes posent immédiatement un problème de déterminisme. Lorsque pour une méta-contrainte C, deux gardes sont impliquées par le CSP alors il y deux manières de transformer la méta-contrainte. Nous ne sommes pas assurés que pour chacune des deux transformations l'ensemble des solutions des CSPs résultants sont les mêmes.

Exemple 4.2.6. Posons la méta-contrainte C avec pour définition :

1. $X = 1 \rightarrow Y > 10$
2. $X = 1 \rightarrow Y < 5$

Si on a un CSP σ qui implique la contrainte $X = 1$ alors le CSP $\sigma \cup \{C\}$ peut donner lieu à deux CSP possibles : $\sigma \cup \{Y > 10\}$ et $\sigma \cup \{Y < 5\}$. Ils ne sont pas équivalents car les ensembles des valeurs imposées pour Y dans chacun d'eux sont disjoints.

4.2.2 Contrainte de cardinalité

L'exemple le plus connu de méta-contrainte à base de contraintes gardées est la contrainte de cardinalité définie dans [VHD90]. La contrainte de cardinalité prend en entrée deux entiers (l et u) et un ensemble de contraintes. Elle impose que le nombre n de contraintes satisfaites est compris entre l (borne inférieure) et u (borne supérieure). Sa sémantique est la suivante :

Définition 4.2.7 Contrainte `card/3`**.**
La contrainte $\texttt{card}(l, u, [C_1, \ldots, C_n])$ *a la sémantique suivante :*

1. $l \leq 0 \wedge n \leq u \rightarrow \texttt{true}$
2. $l \leq u \wedge l = n \rightarrow C_1 \wedge \ldots \wedge C_n$
3. $l \leq u \wedge u = 0 \rightarrow \neg C_1 \wedge \ldots \wedge \neg C_n$
4. $C_i \rightarrow \texttt{card}(l-1, u-1, [C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n])$

5. $\neg C_i \rightarrow \texttt{card}(l, u, [C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n])$

Les trois premières contraintes gardées sont terminales dans le sens où elles ne se réécrivent pas en la contrainte `card`. La première contrainte gardée détecte la consistance de `card` si le nombre de contraintes dans l'ensemble est compris entre les bornes inférieure et supérieure. La deuxième et la troisième détectent si `card` impose que toutes les contraintes doivent être vérifiées ou non vérifiées. Les deux dernières contraintes gardées permettent de retirer de l'ensemble de contraintes une contrainte qui est détectée vraie ou fausse.

4.2.3 Synthèse

Nous avons présenté, dans ce chapitre, une présentation générale de la programmation par contraintes. Nous avons donné, une définition des méta-contraintes avec l'exemple de la contrainte de cardinalité. Dans les chapitres qui suivent, nous définissons la méthodologie de test que nous avons définie dans le cadre du développement d'un programme FoCal. En particulier dans le chapitre qui présente la transformation des programmes FoCal en contraintes, nous donnons la définition de deux nouvelles méta-contraintes qui constituent une partie de la contribution de la thèse.

Deuxième partie

Le test et FoCal

Chapitre 5

Test de propriétés

Dans ce chapitre, nous détaillons le principe du test de propriétés. Nous prenons comme hypothèse que nous possédons une propriété que nous souhaitons vérifier. La propriété porte sur un ensemble de fonctions FoCal. Chacune de ces entités est munie d'une implantation en FoCal sous la forme d'une fonction qui permet de décider la validité. Nous souhaitons vérifier que l'implantation de chaque prédicat/fonction respecte la propriété. Cela revient à « éxecuter » la propriété pour chercher à la mettre en défaut.

5.1 Contexte de test

Nous nous plaçons dans une espèce, le but du test est de vérifier une propriété de l'espèce vis-à-vis de l'implantation donnée pour chacune des fonctions utilisées dans la propriété. Pour cela, nous supposons que l'espèce dans laquelle se place la propriété est complète et que toutes les fonctions possèdent une implantation. Ceci nous assure que les fonctions sur lesquelles portent la propriété existent et que toutes les propriétés sont munies d'une preuve, la preuve pouvant être réduite à la simple utilisation du mot-clef `admitted`, ce qui sera généralement le cas puisque l'on cherche la plupart du temps à tester une propriété non encore prouvée ou qu'on ne veut pas prouver.

Ces précautions sont posées pour assurer que l'espèce peut être rendue exécutable en la transformant en collection. Nous pouvons relâcher la contrainte qui spécifie que l'espèce doit être complète. Nous n'avons besoin que de connaître les définitions des fonctions de la propriété pour effectuer notre test, les autres fonctions peuvent être définies à l'aide de bouchons. Un bouchon est un morceau de code qui vient remplacer le code réel d'une fonction. De plus, les hypothèses que nous formulons ici sont une simplification des hypothèses de tests. Elles donnent une idée du contexte dans lequel on se place pour tester une propriété. On trouvera une définition plus précise du contexte de test dans le chapitre sur la présentation de l'outil FoCalTest en section 9.3.1.

5.2 Restrictions

L'ensemble des propriétés exprimables en FoCal est très vaste. Il s'agit de toutes les propriétés du premier ordre. Comme notre méthode de test se base sur l'exécution de l'implantation, il n'est pas possible de prendre des propriétés de forme quelconque, en particulier on ne peut pas accepter des propriétés qui possèdent des quantificateurs existentiels. En effet, si une variable quantifiée l'est existentiellement, il faudrait, à des fins de test, trouver une valeur de la

variable, parmi l'ensemble des valeurs possibles, qui vérifierait le prédicat sur lequel porte la quantification.

Nous avons choisi de restreindre la forme des propriétés testables à celle décrite ci dessous :

Définition 5.2.1 Propriétés testables**.**
Les propriétés testables *sont les propriétés de la forme suivante :*

$$\begin{array}{r} \forall (x_1 : \tau_1) \ldots (x_n : \tau_n),\ \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow (A_1^1 \vee \ldots \vee A_{n_1}^1) \wedge \ldots \wedge \\ (A_1^m \vee \ldots \vee A_{n_m}^m) \\ \alpha ::= \alpha \vee \alpha \mid \alpha \wedge \alpha \mid A \end{array} \tag{5.1}$$

Les A et A_j^i dénotent des appels de fonctions FoCal, elles peuvent être imbriquées.

Les propriétés testables sont donc des propriétés en forme prénexe. Seul le quantificateur universel est autorisé. Sous les quantificateurs, le corps de la propriété est une suite d'implications dont les formules à gauche sont des disjonctions/conjonctions d'atomes et la formule à droite est sous forme normale conjonctive.

Dans une propriété testable, nous distinguons deux parties.

Définition 5.2.2 Précondition/Conclusion**.**
Soit $P \equiv \forall (x_1 : \tau_1) \ldots (x_n : \tau_n), \alpha_1 \Rightarrow \ldots \alpha_n \Rightarrow \beta$. Nous appelons précondition *(resp.* conclusion*) de P, le prédicat $Pre(P) = \alpha_1 \wedge \cdots \wedge \alpha_n$ (resp. $Conc(P) = \beta$).*

Dans la suite nous parlerons de précondition d'une propriété P pour désigner aussi bien la conjonction $Pre(P) = \alpha_1 \wedge \ldots \wedge \alpha_n$ que l'ensemble $\{\alpha_1, \ldots, \alpha_n\}$ des éléments de cette précondition. De plus, nous désignerons un α_i comme étant une *partie* ou un élément de la précondition.

La forme des propriétés testables est justifiée par le fait que la majorité des propriétés disponibles dans la bibliothèque standard sont exprimées dans une forme similaire. Nous pouvons aussi justifier ce choix en précisant que la classe des propriétés testables capture en partie la classe des propriétés fonctionnelles définies dans [AHP08]. Ces propriétés sont exprimées sous la forme $\forall x_1 \ldots, x_n, r_1(x_1, \ldots, x_n) \Rightarrow r_2(x_1, \ldots, x_n)$ où r_1 et r_2 sont des relations qui peuvent aussi bien être directement des prédicats FoCal que des propriétés.

5.3 Réécriture

Afin de tester les propriétés définies en 5.1, nous effectuons d'abord une transformation. Dans cette phase, la propriété est transformée en un ensemble de propriétés dites élémentaires. Nous réécrivons la propriété de départ pour isoler les différents comportements qui y sont spécifiés. Ainsi, si un contre exemple est trouvé, il est plus facile de d'identifier la cause de l'erreur. Après cette phase de transformation, la propriété avant transformation est oubliée dans le sens où elle n'est plus utile pour la procédure de test. Seules les propriétés de l'ensemble obtenu sont considérées et testées.

Dans un premier temps, nous nous concentrons sur des propriétés sans quantificateur. Nous

$$\left\{ \begin{array}{c} \ldots, \\ \alpha_1 \Rightarrow \ldots \Rightarrow (\beta_1 \vee \ldots \vee \beta_m) \Rightarrow \ldots \Rightarrow \alpha_n, \\ \ldots \end{array} \right\} \longmapsto \left\{ \begin{array}{c} \ldots, \\ \alpha_1 \Rightarrow \ldots \Rightarrow \beta_1 \Rightarrow \ldots \Rightarrow \alpha_n, \\ \alpha_1 \Rightarrow \ldots \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \alpha_n, \\ \vdots \\ \alpha_1 \Rightarrow \ldots \Rightarrow \beta_m \Rightarrow \ldots \Rightarrow \alpha_n, \\ \ldots \end{array} \right\} \tag{5.2}$$

$$\left\{ \begin{array}{c} \ldots, \\ \alpha_1 \Rightarrow \ldots \Rightarrow (\beta_1 \wedge \ldots \wedge \beta_m) \Rightarrow \ldots \Rightarrow \alpha_n, \\ \ldots \end{array} \right\} \longmapsto \left\{ \begin{array}{c} \ldots, \\ \alpha_1 \Rightarrow \ldots \Rightarrow \beta_1 \Rightarrow \ldots \Rightarrow \beta_m \Rightarrow \ldots \Rightarrow \alpha_n, \\ \ldots \end{array} \right\} \tag{5.3}$$

$$\left\{ \begin{array}{c} \ldots, \\ \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow (\beta_1 \wedge \ldots \wedge \beta_m), \\ \ldots \end{array} \right\} \longmapsto \left\{ \begin{array}{c} \ldots, \\ \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow \beta_1, \\ \vdots \\ \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow \beta_m, \\ \ldots \end{array} \right\} \tag{5.4}$$

FIGURE 5.1 – Système de réécriture pour les propriétés

commençons par décrire un système de réécriture qui « casse » les opérateurs logiques d'une propriété afin d'obtenir plusieurs autres propriétés.

Définition 5.3.1 Quantificateurs**.**
Soit P une propriété testable. On définit P', comme étant la propriété P dans laquelle on a ôté les quantificateurs. Ainsi, on a $P \equiv \forall x_1 \ldots x_n, P'$.

Le système de réécriture est donné dans la figure 5.1. Chaque règle du système réécrit une propriété de l'ensemble de gauche en une ou plusieurs propriétés. Le système est défini sur les propriétés testables sans quantificateurs et est composé de 3 règles. Chacune des règles est un schéma de règles qui s'instancie en fonction de la longueur de la suite d'implications et du nombre d'éléments dans la disjonction. Ainsi, la règle 5.2 décrit l'ensemble des règles pour toute valeur de n et de m. Voici la description de chacune des règles :

- la première règle coupe une disjonction qui apparaît dans la précondition. Chaque élément de cette disjonction donne lieu à une nouvelle propriété, celle-ci est la propriété d'origine dans laquelle la disjonction dans son entier a été remplacée par l'élément en question ;
- la deuxième règle reformule les conjonctions dans la précondition en une suite d'implications, elle transforme donc une propriété en une propriété équivalente ;
- enfin, la dernière règle éclate la conjonction de la conclusion d'une propriété. De même que pour la première règle, la propriété est réécrite en un ensemble de propriétés dans lesquelles la conjonction de la conclusion est changée en chacun des éléments de la conjonction initiale.

Exemple 5.3.2. La propriété

$$(\texttt{equal}(x_1 * x_2, 0) \vee \texttt{is\_zero}(x_1)) \Rightarrow (\texttt{equal}(x_1, 0) \vee \texttt{equal}(x_2, 0))$$

se réécrit par la règle 5.2 en

$$\begin{array}{ll} \{ & \texttt{is\_zero}(x_1) \Rightarrow (\texttt{equal}(x_1, 0) \vee \texttt{equal}(x_2, 0)), \\ & \texttt{equal}(x_1 * x_2, 0) \Rightarrow (\texttt{equal}(x_1, 0) \vee \texttt{equal}(x_2, 0))\} \end{array}$$

Ces règles de réécriture décrivent une étape de transformation de la propriété. Pour obtenir les propriétés réellement testées, on applique ces règles de réécriture jusqu'à obtenir des *formes normales*.

Exemple 5.3.3. Dans l'exemple précédent, les deux propriétés

$$\texttt{is\_zero}(x_1) \Rightarrow \texttt{equal}(x_1, 0) \vee \texttt{equal}(x_2, 0)$$
$$\texttt{equal}(x_1 * x_2, 0) \Rightarrow \texttt{equal}(x_1, 0) \vee \texttt{equal}(x_2, 0)$$

sont des formes normales du système.

Afin d'avoir une procédure de transformation cohérente il faut vérifier que le système de réécriture possède les propriétés minimales qu'on attend de lui, à savoir que le processus termine et que les formes normales ne dépendent pas de la stratégie de réduction.

Théorème 5.3.4. *Le système de réécriture 5.1 termine et les formes normales sont uniques.*

Démonstration. Pour prouver la terminaison, on considère pour une propriété, le couple (n, m) où n est le nombre de connecteurs de la propriété et m est le nombre de conjonctions avec l'ordre lexicographique qui est un ordre bien fondé. On remarque alors que les règles de réécriture remplacent une propriété par un ensemble fini de propriétés plus petites au sens de cet ordre. Le système de réécriture transforme ainsi un multi-ensemble par un autre multi-ensemble plus petit par rapport à l'ordre (n, m) que nous venons de définir. L'ordre multi-ensemble étant bien fondé, nous sommes assuré que la système de réécriture termine.

Pour prouver la confluence, nous remarquons que les paires critiques du système sont joignables par inversion de l'ordre d'application des règles car elles ne se chevauchent pas.

□

Le système de réécriture termine et est confluent, on est alors assuré d'obtenir une unique forme normale à partir de n'importe quelle propriété testable.

Définition 5.3.5 Propriétés élémentaires**.**
On appelle ensemble de propriétés élémentaires *la forme normale du système de réécriture. Ces propriétés sont de la forme* $A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow B_1 \vee \ldots \vee B_m$.

Les formes normales du système de réécriture sont des multi-ensembles car nous ne sommes pas assurés que les propriétés obtenues sont toutes différentes. Deux règles peuvent produire directement des doublons, les règles 5.2 et 5.4. Dans ces deux règles, la création de doublons se fait lorsque β_i et β_j sont identiques pour $i \neq j$. Si le doublon est produit par la règle 5.2 cela signifie alors que la précondition contient une disjonction avec deux éléments identiques. Tandis que pour la règle 5.4, cela veut dire que la conclusion contient dans sa conjonction deux éléments identiques. Mais, il faut bien noter que la création de doublons ne se produit pas uniquement par application des règles 5.2 et 5.4.

Exemple 5.3.6. Soient A, B, C et D des appels à des prédicats. Le singleton $\{(A \wedge B) \vee ((A \wedge B) \vee C) \Rightarrow D\}$ se réécrit avec la règle 5.2 en :

$$\{(A \wedge B) \Rightarrow D, ((A \wedge B) \vee C) \Rightarrow D\}$$

L'application de la règle 5.3 sur la première formule de l'ensemble donne :

$$\{A \Rightarrow B \Rightarrow D, ((A \wedge B) \vee C) \Rightarrow D\}$$

Si on applique les règles 5.2 et 5.3, dans cet ordre, sur la deuxième formule on obtient :

$$\{A \Rightarrow B \Rightarrow D, A \Rightarrow B \Rightarrow D, C \Rightarrow D\}$$

Les deux premières formules forment un doublon.

L'exemple précédent montre bien qu'on peut obtenir des doublons par application de la règle 5.3. Dans ce cas, on retrouve une propriété qui apparaît déjà dans l'ensemble. Dans l'exemple, la redondance se trouve au niveau de la précondition de la propriété de départ. Pour la retirer on pourrait reformuler la propriété en $(A \wedge B) \vee C \Rightarrow D$. La forme normale de la propriété n'est pas changée aux doublons près.

Si la forme normale d'une propriété contient deux propriétés identiques cela signifie que la propriété de départ contient des redondances dans sa précondition ou sa conclusion. Ces redondances sont le plus souvent révélatrices d'une erreur de conception dans la spécification et doivent être relevées.

Définition 5.3.7 Forme normale**.**
Soit P une propriété on note $P_\downarrow$ la forme normale de P.

Dans la suite, nous désignerons $P_\downarrow$ par un ensemble et non un multi-ensemble en ne prenant en compte qu'une seule occurrence de chaque élément de $P_\downarrow$.

Théorème 5.3.8. *Soit P une propriété testable dont on a ôté les quantificateurs et soit $P_\downarrow$ la forme normale de P. La propriété suivante est vérifiée :*

$$\bigwedge_{p \in P_\downarrow} p \equiv P$$

où $\equiv$ est l'équivalence des formules logiques.

Démonstration. Pour prouver l'équivalence, il suffit de vérifier que chaque règle de réécriture la préserve.

– Pour la règle 5.2 on la suite d'équivalences suivante :

$$\begin{aligned}
&\alpha_1 \Rightarrow \ldots \Rightarrow (\beta_1 \vee \ldots \vee \beta_m) \Rightarrow \ldots \Rightarrow \alpha_n \\
&\quad \equiv \quad (\alpha_1 \wedge \ldots \wedge (\beta_1 \vee \ldots \vee \beta_m) \wedge \ldots \wedge \alpha_{n-1}) \Rightarrow \alpha_n \\
&\quad \equiv \quad ((\alpha_1 \wedge \ldots \wedge \beta_1 \wedge \ldots \vee \alpha_{n-1}) \vee \ldots \vee (\alpha_1 \wedge \ldots \wedge \beta_m \wedge \ldots \vee \alpha_{n-1})) \Rightarrow \alpha_n \\
&\quad \equiv \quad ((\alpha_1 \wedge \ldots \wedge \beta_1 \wedge \ldots \vee \alpha_{n-1}) \Rightarrow \alpha_n) \wedge \ldots \wedge \\
&\qquad\quad ((\alpha_1 \wedge \ldots \wedge \beta_m \wedge \ldots \vee \alpha_{n-1}) \Rightarrow \alpha_n)
\end{aligned}$$

– Pour la règle 5.3, il s'agit d'un résultat classique :

$$\begin{aligned}
&\alpha_1 \Rightarrow \ldots \Rightarrow (\beta_1 \wedge \ldots \wedge \beta_m) \Rightarrow \ldots \Rightarrow \alpha_n \\
&\quad \equiv \quad \alpha_1 \Rightarrow \ldots \Rightarrow \beta_1 \Rightarrow \ldots \Rightarrow \beta_m \Rightarrow \ldots \Rightarrow \alpha_n
\end{aligned}$$

– Pour la règle 5.4, on utilise aussi un résultat classique :

$$\begin{aligned}
&\alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow (\beta_1 \wedge \ldots \wedge \beta_m) \\
&\quad \equiv \quad (\alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow \beta_1) \wedge \ldots (\alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow \beta_m)
\end{aligned}$$

□

Nous pouvons maintenant étendre le système de réécriture aux propriétés testables.

Définition 5.3.9 Extension des propriétés testables**.**
Soit $P = \forall(x_1 : \tau_1) \dots (x_n : \tau_n)$, P' une propriété testable. Nous étendons la notion de propriété élémentaire *aux propriétés testables en définissant $P_\downarrow = \{\forall(x_1 : \tau_1) \dots (x_n : \tau_n),\ \beta \mid \beta \in {P'}_\downarrow\}$.*

Bien entendu, le théorème 5.3.8 reste valable pour les propriétés testables.

Théorème 5.3.10. *Soit $P = \forall(x_1 : \tau_1) \dots (x_n : \tau_n)$, P' une propriété testable et soit $P_\downarrow$ la forme normale de P. L'extension naturelle du théorème 5.3.8 à P est vérifiée :*

$$\bigwedge_{p \in P_\downarrow} p \equiv P$$

Démonstration. Pour prouver la propriété il faut vérifier que l'ajout des quantificateurs préserve l'équivalence.

$$\begin{array}{lll}
\bigwedge\limits_{p \in P_\downarrow} p & \equiv & \bigwedge\limits_{p \in {P'}_\downarrow} \forall(x_1 : \tau_1) \dots (x_n : \tau_n),\ p \quad \text{par 5.3.9} \\
& \equiv & \forall(x_1 : \tau_1) \dots (x_n : \tau_n),\ \bigwedge\limits_{p \in {P'}_\downarrow} p \quad \text{résultat classique des formules du premier ordre} \\
& \equiv_{(5.3.8)} & \forall(x_1 : \tau_1) \dots (x_n : \tau_n),\ P' \quad \text{par 5.3.8} \\
& \equiv_{def} & P
\end{array}$$

□

5.4 Procédure de test

Nous présentons ici le test des propriétés testables définies en section 5.2. Celui-ci s'effectue en deux étapes. Dans la première étape nous appliquons le système de réécriture de la figure 5.1 sur la propriété sous test. La deuxième étape consiste à tester les propriétés élémentaires séparément les unes des autres. La propriété initiale n'intervient plus dans cette deuxième étape. Nous définissons dans cette section la méthodologie de test de la deuxième étape.

Définition 5.4.1 Jeu de test potentiel**.**
Soit P une propriété élémentaire et soient $x_1, \dots, x_n$ les variables quantifiées de P. Un jeu de test potentiel *pour P est une valuation des variables $x_1, \dots, x_n$.*

Les jeux de test potentiels permettent d'évaluer une propriété dans son entier. On pourrait garder les jeux de test potentiels pour effectuer directement le test mais certaines valuations ne sont pas intéressantes. Intuitivement, un jeu de test qui invalide la précondition n'est d'aucune utilité car la propriété testée est immédiatement vérifiée pour ce jeu de test. On est donc amené à définir la notion de jeu de test valide.

Définition 5.4.2 Jeu de test valide**.**
Soit P une propriété élémentaire et soient $x_1, \dots, x_n$ les variables quantifiées de P. Un jeu de test valide *(resp.* invalide*) est un jeu de test potentiel qui vérifie (resp. ne vérifie pas) la précondition.*

On dit qu'une précondition $Pre(P)$ est vérifiée (resp. n'est pas vérifiée) par une valuation si et seulement si l'évaluation de $Pre(P)$ selon cette valuation donne `true` *(resp.* `false`*).*

Un jeu de test valide est potentiellement un contre-exemple de la propriété. Afin d'obtenir le résultat du test, nous évaluons la conclusion. Cette évaluation peut donner deux résultats dont voici la signification :

- si la conclusion s'évalue à `true` alors la propriété est vérifiée pour ce jeu de test. Nous ne l'avons pas mis en défaut avec le jeu de test.
- si la conclusion s'évalue à `false` alors le jeu de test est un contre-exemple de la propriété. Ce fait doit être rapporté au testeur.

Définition 5.4.3 Verdict d'un jeu de test**.**
Soit P une propriété élémentaire et soit V un jeu de test valide pour P. Le résultat de l'évaluation de la conclusion de P sur V donne le verdict *du jeu de test.*

La sélection d'un jeu de test (resp. sa soumission) requiert l'exécution de la précondition (resp. de la conclusion). Cette exécution est immédiate car la précondition (resp. la conslusion) est une conjonction (resp. disjonction) d'appel à des fonctions booléennes. Il suffit de calculer la conjonction (resp. la disjonction) des résultats obtenus après les appels aux fonctions.

5.5 Étendre la forme des propriétés

Nous avons vu que les propriétés acceptées dans notre procédure de test étaient dans une forme bien particulière à savoir des propriétés en forme prénexe avec uniquement le quantificateur universel et se présentant comme une suite d'implications sur des formules bien précises.

Nous pouvons relâcher la contrainte sur la forme des propriétés testées dans le cas où le domaine des variables quantifiées est fini. Lorsque c'est le cas, nous pouvons, *a priori*, tester de manière automatique la classe des propriétés exprimables dans FoCal sans exception. Il s'agit alors de vérifier la propriété exhaustivement en énumérant l'ensemble des valeurs possibles des variables quantifiées.

Il faut toutefois modérer ces propos. Dans la plupart des cas, vérifier une propriété exhaustivement, quand le domaine des variables est fini, est trop coûteux en temps. C'est le cas par exemple lorsque les variables quantifiées sont entières. Tester une propriété qui contient un unique quantificateur sur une variable de type **int** [1], est envisageable (mais pas forcément raisonnable) en utilisant les machines actuelles, mais ne l'est plus dès que plusieurs variables entières interviennent dans l'énoncé de la propriété. C'est le cas, par exemple, de la propriété d'associativité de l'addition des entiers qui s'énonce comme suit :

$$\forall(x : \mathbf{int}), (y : \mathbf{int}), (z : \mathbf{int}), \sharp add\_int\,(x, \sharp add\_int\,(y, z)) = \sharp add\_int\,(\sharp add\_int\,(x, y), z)$$

Nous proposons dans la suite une méthodologie pour rendre exécutable une propriété qui ne contient que des variables sur des types finis. Il conviendra de proposer une restriction pour empêcher d'exécuter des propriétés telles que l'associativité de l'addition sur les entiers. Cela va nous définir une classe de propriétés. On va ensuite utiliser cette classe pour étendre l'ensemble des propriétés testables.

1. On rappelle qu'en focal un entier est codé sur 31 bits, soit 2^{31} valeurs possibles (environ 2 milliards)

Propriétés énumérables

Ainsi, comme nous l'avons dit, nous pouvons accepter des propriétés qui ne sont pas sous forme prénexe pourvu que les variables quantifiées dans le corps de la propriété soient de type fini. L'idée est de les exécuter, c'est-à-dire, les transformer en une expression qui soumet l'ensemble des valeurs possibles pour les variables quantifiées. Par exemple, la quantification universelle d'une variable va être transformée en une boucle qui va soumettre une à une les valeurs possibles de la variable. L'expression obtenue par ce procédé va retourner une valeur booléenne, `true` si la propriété est vérifiée par l'ensemble des valeurs possibles et `false` dans le cas contraire. Avant d'effectuer la transformation, il faut d'abord fournir une condition qui interdit d'exécuter des propriétés qui donneraient des expressions trop coûteuses en temps d'exécution.

Nous commençons par définir le cardinal d'un type. À partir de cette fonction, nous allons pouvoir définir le coût de test d'une propriété qui quantifie sur des variables de types finis.

Définition 5.5.1 Cardinal d'un type**.**
Soit τ un type. On définit card(τ) de la manière suivante :

$$
\begin{array}{ll}
\mathit{card}(\mathbf{int}) = 2^{31} & \textit{(le type } \mathbf{int} \textit{ est le type des entiers de FoCal)} \\
\multicolumn{2}{l}{\textit{Pour } \tau \textit{ tel que } \mathrm{def}_\tau(\tau) = \{C_1 : \tau_1^1 \to \ldots \to \tau_{n_1}^1 \to \tau, \ldots, C_m : \tau_1^m \to \ldots \to \tau_{n_m}^m \to \tau\}} \\
\mathit{card}(\tau) = \sum_{i=1}^{m} \mathit{card}(\tau_1^i) * \ldots * \mathit{card}(\tau_{n_i}^i) & \textit{si } \forall i \in [\![1,n]\!], \forall j \in [\![1,m]\!], \tau \neq \tau_j^i \textit{ et } \mathit{card}(\tau_j^i) \neq \top \\
 & \textit{pour } n_i = 0\textit{, on a } \mathit{card}(\tau_1^i) * \ldots * \mathit{card}(\tau_{n_i}^i) = 1 \\
\mathit{card}(\tau) = \top & \textit{sinon}
\end{array}
$$

La fonction *card* donne le nombre de valeurs que peut prendre une variable d'un type donné τ. Si le type de donnée est infini, comme par exemple le type des listes, alors *card* associe au type la valeur infinie $\top$.

Nous étendons maintenant la définition de *card* sur un ensemble de types.

Définition 5.5.2 Cardinal de types**.**
Soit T un multi-ensemble de types. On étend la définition de card sur T par :

$$\mathit{card}(T) = \prod_{\tau \in T} \mathit{card}(\tau)$$

Cette extension permet de calculer le coût total d'énumération d'une propriété qui quantifie sur plusieurs variables à partir du multi-ensemble de types des variables quantifiées. Il s'agira de donner à *card* l'ensemble des types qui apparaissent dans la liste des variables quantifiées. Intuitivement, la valeur rendue par *card*(T) indique le nombre de valeurs à soumettre à la propriété pour l'énumérer exhaustivement. La fonction *card* prend en entrée un multi-ensemble de types car plusieurs des variables quantifiées peuvent avoir le même type, il faut prendre en compte les types autant de fois qu'ils apparaissent.

Avec la fonction *card*, nous pouvons donner la définition de la classe des propriétés qui sont concernées par le processus d'énumération, les propriétés énumérables.

Définition 5.5.3 Propriétés énumérables**.**

Soit P une propriété et n un entier, soit $X_b = bv(P)$ le multi-ensemble des variables typées liées dans P. P est une propriété énumérable *vis-à-vis de n si et seulement si $card(\tau_b) \leq n$ tel que $\tau_b = \{\tau \mid \exists x, x : \tau \in X_b\}$.*

Quand une propriété est énumérable vis-à-vis d'un entier n cela signifie que la propriété peut être vérifiée par au plus n évaluations, une évaluation se faisant quand toutes les variables liées dans la propriété sont associées à une valeur. Dans la suite on parlera de propriété énumérable sans préciser vis-à-vis de quel entier lorsque l'entier est superflu dans les explications.

Nous pouvons maintenant étendre la classe des propriétés testables à l'aide des propriétés énumérables.

Définition 5.5.4 Extension des propriétés testables**.**
Nous étendons la forme des propriétés testables aux propriétés énumérables*. Les nouvelles propriétés testables sont de la forme suivante :*

$$\begin{gathered} \forall (x_1 : \tau_1) \ldots (x_n : \tau_n),\ \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow (A_1^1 \vee \ldots \vee A_{n_1}^1) \wedge \ldots \wedge \\ (A_1^m \vee \ldots \vee A_{n_m}^m) \\ \alpha ::= \alpha \vee \alpha \mid \alpha \wedge \alpha \mid P \\ P \text{ est une propriété énumérable} \end{gathered} \tag{5.5}$$

Notons que les propriétés testables sont identiques à la définition 5.1 à la différence près que la précondition contient maintenant des propriétés énumérables en tant que sous-formule.

La mise en forme normale des propriétés reste identique au processus défini plus haut. Le système de réécriture reste le même et la forme normale a la même forme avec pour seule différence que les éléments des préconditions sont maintenant des propriétés énumérables.

Définition 5.5.5 Forme normale**.**
Les propriétés apparaissant dans la forme normale d'une propriété testable sont de la forme $\forall (x_1 : \tau_i) \ldots (x_n : \tau_n), P_1 \Rightarrow \ldots \Rightarrow P_n \Rightarrow B_1 \vee \ldots \vee B_m$ où les P_i sont des propriétés énumérables.

Les propriétés énumérables sont alors traduites en une forme exécutable sous la forme d'une expression FoCal. Cette expression prend en paramètre les valeurs des variables libres de la propriété originelle et s'évalue en une valeur booléenne. Ce résultat précise ensuite si la propriété est vérifiée pour l'ensemble des valuations des variables quantifiées ou non.

Définition 5.5.6 Énumération d'un type**.**
On définit val comme étant la fonction qui calcule la liste des valeurs d'un type donné en argument.

La figure 5.2 contient les règles de transformation des propriétés testables en code FoCal. Les règles sont de la forme $P \longmapsto p$ et se lisent « la propriété P se transforme en le code p ». Les règles les plus intéressantes sont Exists et Forall. Pour ces deux règles on suppose l'existence

$$\frac{P_1 \longmapsto p_1 \qquad P_2 \longmapsto p_2}{P_1 \Leftrightarrow P_2 \longmapsto \textbf{if}\ p_1\ \textbf{then}\ p_2\ \textbf{else}\ \texttt{not}(p_2)}\ \textsc{Equiv} \qquad \frac{P_1 \longmapsto p_1 \qquad P_2 \longmapsto p_2}{P_1 \wedge P_2 \longmapsto \texttt{and}(p_1, p_2)}\ \textsc{And}$$

$$\frac{P_1 \longmapsto p_1 \qquad P_2 \longmapsto p_2}{P_1 \Rightarrow P_2 \longmapsto \textbf{if}\ p_1\ \textbf{then}\ p_2\ \textbf{else}\ \texttt{true}}\ \textsc{Imply} \qquad \frac{P_1 \longmapsto p_1 \qquad P_2 \longmapsto p_2}{P_1 \vee P_2 \longmapsto \texttt{or}(p_1, p_2)}\ \textsc{Or}$$

$$\frac{P \longmapsto p}{\exists(x:\tau), P \longmapsto \texttt{exists}\ (\textbf{fun}\, x \to p)\ \mathit{val}(\tau)}\ \textsc{Exists} \qquad \frac{}{A \longmapsto A}\ \textsc{Call}$$

$$\frac{P \longmapsto p}{\forall(x:\tau), P \longmapsto \texttt{forall}\ (\textbf{fun}\, x \to p)\ \mathit{val}(\tau)}\ \textsc{Forall}$$

FIGURE 5.2 – Traduction des propriétés énumérables

de deux fonctions `exists` et `forall` qui prennent comme argument un prédicat FoCal et une liste de valeurs. La fonction EXISTS retourne `true` si au moins un élément de la liste vérifie le prédicat et `false` sinon. La fonction `forall` retourne `true` si tous les éléments de la liste vérifient le prédicat. Les deux constructions $\forall$ et $\exists$ des propriétés sont traduites en des appels à chacune de ces deux fonctions. Pour les règles AND et OR, on suppose l'existence de deux fonctions `and` et `or` qui prennent en arguments deux valeurs booléennes et qui retournent respectivement la conjonction et la disjonction des deux paramètres.

5.6 Génération de jeux de test pseudo-aléatoire

Une première approche facile à mettre en œuvre pour obtenir des jeux de test consiste à utiliser l'approche aléatoire. Cela consiste à générer les jeux de test potentiels en prenant, pour chaque variable, une valeur prise au hasard dans son ensemble de valeurs possibles. Les jeux de test potentiels sont ensuite vérifiés en exécutant la précondition. Nous présentons brièvement notre méthode pour générer des valeurs aléatoires à partir de la définition d'un type.

La génération aléatoire de valeurs à partir de définitions algébriques n'est pas un problème aisé et en particulier, si on veut obtenir une répartition équiprobable des valeurs.

Des travaux en la matière existent, par exemple [FZVC94] décrit la *méthode récursive* qui permet d'obtenir une valeur d'une taille fixée avec une répartition équiprobable. La méthode découpe le travail de génération en deux étapes, une étape de pré-traitement sur la définition algébrique coûteuse en temps et en espace mémoire puis une étape de création de générateur. Les générateurs ont une complexité en $O(n^2)$ dans le pire cas.

Plus récemment, on trouve la *méthode de Boltzmann* [DFLS04] qui relaxe la contrainte sur la taille de l'objet généré. Cette méthode présente pour propriétés de générer des objets d'une taille qui est proche de la taille désirée et, pour un objet d'une taille fixé, de générer un objet avec une répartition uniforme sur les objets de cette taille. Les algorithmes donnés par la méthode de Boltzmann sont très souvent linéaires [Piv08].

Plus proche du test de programmes, on trouve dans le cadre de la génération de jeux de tests d'Isabelle/HOL [BN04] la définition de générateurs basés sur des combinateurs. Les générateurs n'ont pas une répartition uniforme des valeurs sur leur taille, ils se basent sur un paramètre entier qui définit une probabilité servant d'une part à spécifier, un peu comme la méthode de

Boltzmann, la taille de l'objet voulue et d'autre part à borner la profondeur de celui-ci.

Nous présentons ici succinctement la méthode que nous utilisons pour générer pseudo-aléatoirement des valeurs d'un type défini inductivement. Cette méthode n'a pas pour ambition d'avoir de propriétés spécifiques sur la répartition des objets générés. Notre procédure de test nous impose de trouver des valeurs des variables telles qu'une certaine propriété est vérifiée. Cette contrainte implique que de nombreux essais sont réalisés avant de trouver un jeu de test. Obtenir un générateur de valeurs qui respecte une propriété d'uniformité demande beaucoup d'effort et de temps pré-calcul/d'exécution. Nous cherchons avec notre approche aléatoire à obtenir des jeux de test rapidement. Dans cette optique, nous proposons une génération aléatoire qui permet d'obtenir des valeurs rapidement et qui ne rivalise pas avec les trois méthodes présentées sur la répartition des valeurs.

Soit une définition d'un type inductif τ par des constructeurs $C_1, \ldots, C_n$:

$$def_\tau(\tau) = \{C_1 : \tau_1^1 \to \ldots \to \tau_{n_1}^1 \to \tau, \ldots, C_m : \tau_1^m \to \ldots \to \tau_{n_m}^m \to \tau\}$$

On commence par créer l'ensemble des constructeurs dit « récursifs » et l'ensemble des constructeurs « non-récursifs ». Un constructeur C_i est considéré comme récursif si on a $\tau \in \{\tau_1^i, \ldots, \tau_{n_i}^i\}$ et comme non-récursif dans le cas contraire. On construit pseudo-aléatoirement une valeur de type τ de la façon suivante : on choisit avec une probabilité fixée soit de prendre un constructeur récursif soit un constructeur non-récursif. Une fois ce choix effectué, on choisit avec une répartition uniforme un constructeur dans l'ensemble choisi. Pour chacun des types attendus par le constructeur, nous prenons une valeur au hasard de ce type et les appliquons au constructeur.

Pour les types de base comme les entiers naturels, nous disposons de générateurs prédéfinis.

Exemple 5.6.1. Soit la définition des arbres binaires d'entiers :

$$def_\tau(\texttt{arbre}) = \{\texttt{Feuille} : \texttt{int} \to \texttt{arbre},\ \texttt{Noeud} : \texttt{int} \to \texttt{arbre} \to \texttt{arbre} \to \texttt{arbre}\}$$

`Feuille` est un constructeur non récursif et `Noeud` est récursif car il attend un paramètre de type `arbre`.

Si on choisit le constructeur `Feuille` on doit générer un entier i avant de rendre la valeur `Feuille`(i), si on choisit le constructeur `noeud` on doit générer un entier i et deux arbres a_1, a_2 (avec la méthode en train d'être décrite) puis retourner la valeur `Noeud`(i, a_1, a_2).

La méthode décrite précédemment a le défaut de donner des générateurs qui ne terminent pas. Pour l'exemple 5.6.1, la probabilité que la méthode termine est de $\frac{1}{2}$. Pour cela, on munit les générateurs d'un paramètre qui fixe la profondeur maximale des objets générés. Pour l'exemple, à chaque génération d'un nouvel arbre ce paramètre est décrémenté. Lorsque le paramètre atteint 0, on choisit un constructeur non récursif.

5.7 Synthèse

Nous avons défini une méthode de test de propriétés dans le cadre d'un système certifiant. Comme dans ce système les propriétés et l'implantation sont incluses dans le même fichier source nous avons pu en tirer partie. Lors du test d'une propriété nous avons accès à l'implantation à laquelle elle se rapporte. L'implantation étant exécutable nous avons pu en tirer parti pour vérifier si un jeu de test *potentiel* est *valide* ou non. De plus, nous avons pu passer outre le problème de l'oracle, la conclusion de la propriété définissant une procédure de décision/verdict.

La classe des propriétés testables est celle des propriétés en forme prénexe mais contient aussi les propriétés dont les quantificateurs dans le corps de la propriété portent sur des variables dont le type est fini. Dans ce chapitre nous avons succinctement adressé le problème de la synthèse des jeux de test en utilisant une approche aléatoire. Cette approche, bien que plaisante, est très rapidement dépassée lorsque la précondition admet peu de valuations qui la valident. Dans le chapitre suivant nous présentons une approche différente pour obtenir les jeux de test *valides*. Cela consiste à utiliser la programmation par contraintes pour synthétiser des jeux de test valides.

Dans le chapitre qui suit, nous présentons FoCal de manière plus formelle que le chapitre 3. Cette formalisation est nécessaire car nous cherchons à traduire un programme FoCal en un ensemble de contraintes tout en prouvant que cette traduction est correcte et complète.

Chapitre 6

Le langage FoCal

Nous faisons ici une présentation plus formelle du langage FoCal. Nous détaillerons les aspects du langage qui nous semblent essentiels pour la suite. Nous ne parlerons pas des aspects logiques du langage. En particulier, nous donnerons la syntaxe des propriétés exprimables dans FoCal sans expliciter leur intégration dans le langage. Nous ne détaillerons pas non plus les aspects relatifs à la preuve qui dépassent le cadre de notre discours. Les définitions qui suivent ne sont donc pas complètes mais suffisantes pour avoir une compréhension globale du langage FoCal.

Les travaux qui ont été réalisés précédemment sur la sémantique de FoCal ont successivement portés sur une étude de la cohérence de FoCal [Bou00b], le développement d'un compilateur [Pre03] ainsi que sur une étude des traits orientés objets du langage [Fec05]. La sémantique que nous donnons dans la suite reprend en grande partie les idées présentées dans [Fec05] (et quelques notations). Nous ne présentons pas les aspects de cette sémantique qui se concentrent sur les traits objets mais plutôt sur le langage fonctionnel qui fait le cœur de FoCal. La sémantique que nous donnons est classique dans le sens où elle ne présente pas d'originalité par rapport à des travaux antérieurs sur les langages fonctionnels.

Un autre aspect du langage que nous ne formalisons pas concerne le typage. Nous présentons dans la suite les principaux types utilisés dans FoCal mais ne détaillons pas le mécanisme de typage du langage des expressions. Les propriétés usuelles des langages fortement typés sont valables dans FoCal. Ainsi, on a une garantie de sûreté de typage ; toute expression correctement typée dans FoCal s'évalue en une valeur du même type.

Nous nous intéressons aussi uniquement aux traits orientés programmation. En particulier, nous introduisons les informations suffisantes pour définir la sémantique d'un programme FoCal. Cela signifie que lorsque nous introduisons les interfaces/valeurs d'espèces nous ne montrons pas la présence des propriétés. Ce manque ne nuit pas à la compréhension du fonctionnement du langage.

6.1 Les types

FoCal est un langage de programmation fortement typé au même titre que OCaml. Ainsi, les types ont une part très importante dans le fonctionnement du langage. Lors de l'écriture d'un programme FoCal, les types des données manipulées restent implicites. Ils sont inférés par la suite par le compilateur lors de la phase de typage. Toutefois, comme nous l'avons vu dans les exemples du chapitre 3, le développeur a la possibilité de spécifier le type des valeurs manipulées à l'aide d'annotations de types. Les annotations sont facultatives et un programme peut être développé sans leur aide. Malgré cela, dans certains cas, il est très utile, voire obligatoire,

$$
\begin{array}{rcl}
\tau & ::= & \iota \\
 & | & \alpha \\
 & | & \tau_1 \rightarrow \tau_2 \\
 & | & \tau_1 * \tau_2 \\
 & | & \kappa \\
 & | & \kappa(\tau_1, \ldots, \tau_n)
\end{array}
$$

FIGURE 6.1 – Les types

d'utiliser les annotations lorsqu'une valeur manipulée doit être de type **self**. Nous avons vu un tel exemple avec la fonction `from_int` décrite à la page 32.

Les types des expressions FoCal sont définis dans la figure 6.1. ι représente les types de base du langage. α représente les variables de type. Le type $\tau_1 \rightarrow \tau_2$ définit le type des fonctions qui prennent une valeur de type τ_1 en entrée et retournent une valeur de type τ_2. Le produit des types τ_1 et τ_2 est noté $\tau_1 * \tau_2$. κ représente les types concrets. Un type concret peut être paramétré, ainsi pour un type concret c, $c(\tau_1, \ldots, \tau_n)$ est le type c où les paramètres sont instanciés par τ_1, …, τ_n.

Nous détaillons dans la suite les types de base et les types concrets. Nous assimilons le type $*$ à un type concret avec un unique constructeur. Dans FoCal, il y a essentiellement quatre sortes de types. Les types sur des valeurs numériques, les types concrets, les chaînes de caractères et le type des fonctions. Nous ne présentons pas les deux derniers car, par la suite, nous utilisons les informations de typage pour générer des valeurs aléatoires. Nous avons choisi de ne pas générer des valeurs fonctionnelles et des chaînes de caractères. FoCal permet d'exprimer des propriétés de la logique du premier ordre, les variables quantifiées ne sont donc pas fonctionnelles. De plus, les propriétés exprimées dans FoCal portent rarement sur des chaînes de caractères.

6.1.1 Types numériques

FoCal définit deux types de données numériques, les entiers dont le type est `int` et les flottants de type `float`.

- les valeurs entières sont les mêmes que celles de OCaml. Elles prennent leurs valeurs dans l'intervalle $[\![-2^{31}, 2^{31} - 1]\!]$;
- les flottants de FoCal sont ceux définis par la norme IEEE 754. [Sta08].

Dans la suite, nous ne considérons que le type des entiers comme type de base. Nous présentons une syntaxe de FoCal qui n'intègre pas les flottants car plus tard, nous ne chercherons pas à traduire en contraintes des programmes qui portent sur des valeurs flottantes. Notre approche ne permet pas de résoudre des contraintes sur les flottants. Nous ne soucions pas de ce type dans la suite.

6.1.2 Types concrets

Un type concret est la donnée d'un ensemble de constructeurs typés. Les constructeurs forment une définition inductive des habitants du type. Une valeur d'un type concret est un terme.

Définition 6.1.1 Constructeurs d'un type**.**
Soit τ un type concret, l'ensemble des constructeurs de τ est donné par def_τ.
$\text{def}_\tau(\tau) = \{C_i : \tau_1^i \to \ldots \to \tau_{n_i}^i \to \tau\}$. *Les constructeurs C_i sont typés et les τ_j^i sont des types quelconques qui peuvent être égaux à τ.*

Un type concret peut être paramétré par d'autres types. Dans ce cas, les τ_j^i de sa définition peuvent faire référence à ces paramètres.

Exemple 6.1.2. Le type des listes est défini par deux constructeurs, `Nil` et `Cons`.

$$\begin{aligned} def_\tau(\texttt{list}(\alpha)) \quad = \{ \quad & \texttt{Nil} : \texttt{list}(\alpha), \\ & \texttt{Cons} : \alpha \to \texttt{list}(\alpha) \to \texttt{list}(\alpha) \\ \} & \end{aligned}$$

Le type des listes est paramétré par le type des valeurs contenues dans la liste. Le constructeur `Nil` n'attend pas de paramètre et représente la liste vide. Le constructeur `Cons` prend deux arguments, le premier argument est une valeur de type α, le paramètre du type, et le deuxième argument est une liste de valeurs de type α, ce constructeur permet d'ajouter un élément en tête d'une liste.
La liste des trois entiers 4, 6 et 2 est alors représentée par `Cons`(4,`Cons`(6,`Cons`(2,`Nil`))) et a pour type `list`(`int`).

Le type d'un constructeur spécifie son arité et le type de chacun de ses arguments. Bien que le type associé à un constructeur soit non curryfié, un constructeur ne peut pas être appliqué partiellement. Si on reprend l'exemple des listes, `Cons` 4 n'a aucune signification car `Cons` attend deux arguments. À ce titre, l'application d'un constructeur est notée de manière curryfiée.

Le langage FoCal prédéfinit un certain nombre de types concrets. Ces types sont classiques et apparaissent dans tous les langages fonctionnels. Nous les présentons brièvement :

- Le type `unit` est le type de données à une seule valeur. Il est constitué d'un unique constructeur qui ne prend pas d'argument. Ce type est généralement utilisé pour définir des fonctions qui ne produisent pas de résultat. Son constructeur est noté (). Ainsi la définition du type `unit` est la suivante : $def_\tau(unit) = \{() : \texttt{unit}\}$;

- Le type des listes. Ce type est paramétré par le type des valeurs qui sont contenues dans la liste. L'exemple 6.1.2 donne la définition des listes et une valeur de ce type ;

- Les valeurs booléennes. Ce type est composé de deux constructeurs sans argument `true` et `false`. Sa définition est $def_\tau(\texttt{bool}) = \{\texttt{true} : \texttt{bool}, \texttt{false} : \texttt{bool}\}$;

- Le type des valeurs partielles (aussi appelé type *option*). Il définit deux constructeurs : $def_\tau(\texttt{partiel}(\alpha)) = \{\texttt{Failed} : \texttt{partiel}(\alpha), \texttt{Unfailed} : \alpha \to \texttt{partiel}(\alpha)\}$. Ses deux constructeurs sont `Failed` qui ne prend pas d'argument et `Unfailed` qui attend un argument de type α. Essentiellement, le type partiel permet de définir le type de retour des fonctions qui ne sont pas définies pour certaines valeurs de leur domaine d'entrée. La valeur `Failed` désigne un résultat indéfini et `Unfailed` signifie le succès du calcul et transporte le résultat ;

- Les couples sont représentés par un type concret à un constructeur. Le type des couples prend deux arguments, les types des éléments du couple. $def_\tau(\texttt{crp}(\alpha, \beta)) = \{\texttt{crp} : \alpha \to$

$\beta \rightarrow \texttt{crp}(\alpha, \beta)\}$ à savoir, le type des couples d'entiers est défini par $\texttt{crp}(\texttt{int}, \texttt{int})$ dont une valeur possible est $\texttt{crp}(5, 8)$.

En plus des types prédéfinis, le développeur a la possibilité de définir ses propres types concrets. Pour cela, il doit donner le nom du nouveau type, les paramètres formels et la liste des constructeurs avec leurs types. À partir de ces informations, FoCal ajoute le nouveau type dans def_τ. Il n'est pas possible d'avoir plusieurs types de même nom.

6.2 Les expressions de base

Les expressions de base forment le cœur du langage. Elles fournissent à l'utilisateur les constructions nécessaires pour définir un programme.

Dans FoCal, les expressions de base apparaissent dans les définitions des fonctions, au sein des espèces. Nous présentons dans la suite la syntaxe des expressions de base de FoCal. Nous donnons la sémantique de ces expressions dans la section 6.7 qui suit la présentation des espèces et des collections.

Nous ne présentons pas la sémantique des expressions en même temps que leur syntaxe car pour donner la sémantique des expressions il faut que le contexte dans lequel se place l'expression soit complètement défini.

Plus précisément, il faut que l'ensemble des fonctions qui sont utilisées dans l'expression soit connu. Or, les expressions sont utilisées dans la définition des espèces. Le mécanisme d'héritage, et plus précisément la liaison retardée, permet de définir des expressions qui font référence à des déclarations. L'évaluation d'une expression qui contient de telles références, ne peut pas aboutir. L'environnement qui lui est associé est incomplet. Par exemple, une expression peut faire appel à une fonction dont on ne connaît que la signature et dont la définition n'est pas encore connue.

Syntaxe des expressions

La syntaxe des expressions de base de FoCal est présentée dans la figure 6.2. Elle comprend les constructions usuelles.

La construction **let** $x = e_1$ **in** e_2 permet de définir temporairement une variable x dont la valeur est le résultat de l'évaluation de e_1. L'expression **let** s'évalue en la valeur de e_2 dans laquelle la variable x a pris la valeur de e_1.

La conditionnelle **if** e_1 **then** e_2 **else** e_3 exprime une rupture dans le flot de contrôle, si la condition e_1 est vraie, la conditionnelle s'évalue dans la valeur de e_2, sinon elle s'évalue comme la troisième expression.

L'appel de fonction $sc!m^i(e_1, \ldots, e_i)$, consiste à appeler la fonction m^i d'arité i. Le symbole qui préfixe le ! indique où trouver la fonction. Dans **self**!$f^i(\ldots)$, le mot-clef **self** précise que f^i est une fonction de l'espèce courante ou héritée (l'espèce dans laquelle apparaît l'expression). Dans les exemples de la section 3, **self** est volontairement omis. Dans $\mathfrak{c}!f^i(e_1, \ldots, e_n)$, $\mathfrak{c}$ indique que la fonction $f^i(e_1, \ldots, e_n)$ se trouve dans la collection $\mathfrak{c}$. Les paramètres effectifs de l'appel sont les expressions $e_1 \ldots e_n$.

La construction $\sharp f^i(e_2, \ldots, e_i)$, est l'appel d'un opérateur de FoCal. Il s'agit des fonctions prédéfinies comme ; par exemple les opérations sur les entiers ou sur les listes. Les paramètres ont la même signification que pour les appels de fonctions.

$e ::=$	**let** $x = e_1$ **in** e_2	définition de variable locale
$\mid$	**if** e_1 **then** e_2 **else** e_3	conditionnelle
$\mid$	$sc!m^i(e_1, \ldots, e_i)$	appel de fonction d'arité i
$\mid$	$\sharp f^i(e_1, \ldots, e_i)$	appel de fonction prédéfinie
$\mid$	**match** e **with** $\mid pat_1 \rightarrow e_1$ $\vdots$ $\mid pat_i \rightarrow e_i$	filtrage
$\mid$	C^0	constructeur constant
$\mid$	$C^n(e_1, \ldots, e_n)$	constructeur d'arité n
$\mid$	i	valeur entière
$\mid$	x	variable

$sc ::=$	**self**	espèce courante
$\mid$	$\mathfrak{c}$	collection

$pat ::=$	x	définition de variable
$\mid$	_	attrape tout
$\mid$	C^0	constructeur constant
$\mid$	$C^n(pat_1, \ldots, pat_n)$	constructeur d'arité n

$v ::=$	i	valeur entière
$\mid$	C^0	constructeur constant
$\mid$	$C^n(v_1, \ldots, v_n)$	constructeur d'arité n appliqué

FIGURE 6.2 – Syntaxe des expressions de base

La construction **match** permet de faire du filtrage par motif. Celle-ci définit quelle expression évaluer en fonction du motif vérifié par la valeur filtrée.

Les quatre dernières constructions expriment les constructeurs d'arité 0, les constructeurs d'arité n appliqués à n expressions, les entiers et les variables sont des expressions autorisées par FoCal.

Exemple 6.2.1. L'expression suivante permet de faire l'addition ou le produit de deux entiers y et z en fonction de la valeur d'un booléen a :

$$\textbf{if } a \textbf{ then } \sharp int\_add\,(y, z) \textbf{ else } \sharp int\_mult\,(y, z)$$

Définition 6.2.2 Variables libres**.**
Soit une expression e. L'ensemble des variables libres de e est donné par la fonction FV en considérant que le seul lieur de variables est la construction **let** *; dans l'expression* **let** $x = e_1$ **in** e_2*, x est liée dans* e_2*.*

6.3 Les propriétés

Le langage FoCal permet de spécifier des propriétés. Les propriétés exprimables en FoCal sont les propriétés du premier ordre. La syntaxe est la suivante :

$$\begin{array}{rcl} \mathit{prop} & ::= & A \\ & | & \forall x \in \tau, \mathit{prop} \\ & | & \exists x \in \tau, \mathit{prop} \\ & | & \mathit{prop} \vee \mathit{prop} \\ & | & \mathit{prop} \wedge \mathit{prop} \\ & | & \mathit{prop} \Rightarrow \mathit{prop} \\ & | & \mathit{prop} \Leftrightarrow \mathit{prop} \end{array}$$

$$\begin{array}{rcl} A & ::= & sc!\, m^i(A_1, \ldots, A_i) \\ & | & \sharp f^i(A_1, \ldots, A_i) \\ & | & x \end{array}$$

Les propriétés logiques de FoCal sont donc classiques. On dispose des opérateurs logiques usuels : La conjonction, la disjonction, l'implication, l'équivalence et les deux types de quantificateurs. La seule originalité concerne les atomes. Ceux-ci sont définis à partir des symboles de fonctions et des opérateurs du langage appliqués au bon nombre d'arguments. Dans la suite, nous appellerons *appel à un prédicat* ces atomes. Ces arguments peuvent être des noms de variables ou des symboles de fonction appliquées à des arguments. D'un point de vue logique, ces symboles sont vus comme des prédicats et des relations sur lesquels on établit les propriétés.

En plus des propriétés, FoCal offre des définitions de propriétés logiques similaires aux macros du langage C. Ces prédicats forment une facilité d'écriture. Nous les avons déjà mentionnées page 33. Ils ne sont pas présentés dans la formalisation du langage car nous supposons que dans les propriétés qu'on teste, les utilisations des propriétés logiques sont expansées et donc remplacées par leur définition.

Exemple 6.3.1. Soient **self!** *odd* et **self!** *even* deux fonctions qui établissent la parité d'un entier. Les propriétés qui mettent en relation ces deux fonctions sont les suivantes :

$$\forall x \in \textbf{int}, \textbf{self!}\ odd(x) \Rightarrow \textbf{self!}\ even(\sharp add\_int\,(x, 1))$$
$$\forall x \in \textbf{int}, \textbf{self!}\ even(x) \Rightarrow \textbf{self!}\ odd(\sharp add\_int\,(x, 1))$$

La propriété voit **self!** *odd* et **self!** *even* comme des prédicats.

6.4 Les interfaces

Avant de donner une définition formelle des espèces et des collections on commence par donner la définition d'une interface. Si une espèce attend un paramètre collection, ce paramètre est spécifié par une interface. Les interfaces définissent les requis minimaux que doit satisfaire une collection pour être compatible. On définit ensuite une relation de compatibilité sur les interfaces. Cette relation permet de décider si une collection peut être paramètre d'une espèce paramétrée.

On distingue deux sortes d'interfaces, les interfaces de collection et les interfaces d'espèce. Cette distinction est uniquement syntaxique, les deux sortes d'interfaces définissent essentiellement la même chose. On introduit cette distinction pour, dans la suite, spécifier à quels endroits on peut placer une interface qui attend des paramètres ou non.

6.4.1 Interfaces de collections

Nous définissons les interfaces des collections de la manière suivante :

$$I^{\mathfrak{c}} \quad ::= \quad \exists \mathsf{rep\text{-}abs}.\{m_1 : \tau_1 \ldots m_n : \tau_n\}[\textbf{self} \leftarrow \mathsf{rep\text{-}abs}]$$

La notation [**self** ← rep-abs] signifie qu'on substitue toutes les occurrences de **self** dans le type des méthodes m_i par rep-abs. La quantification permet de s'assurer que le type support n'est pas une variable libre de l'interface et qu'il est masqué en dehors de l'espèce.

L'interface d'une collection est notée $I^{\mathfrak{c}}$. Elle est composée d'un ensemble de noms de fonction f_i munis pour chacun d'eux d'un type τ_i appelé *signature de la fonction*. Une collection qui doit satisfaire cette interface doit contenir au moins les fonctions m_i avec pour signature τ_i.

La quantification du type support permet de rendre compatible des interfaces pour lesquelles le type support est différent. On rappelle que le type **self** est un alias du type support de l'espèce.

Exemple 6.4.1. L'interface des treillis sur les entiers donnée dans la présentation informelle de FoCal en section 3 est la suivante :

$$\begin{aligned} I^{\mathfrak{c}}_{\text{treillis\_entiers}} = \exists \mathsf{rep\text{-}abs}.\{\ & inf && : \mathsf{rep\text{-}abs} \to \mathsf{rep\text{-}abs} \to \mathsf{rep\text{-}abs} \\ & order\_inf && : \mathsf{rep\text{-}abs} \to \mathsf{rep\text{-}abs} \to \textbf{bool} \\ & sup && : \mathsf{rep\text{-}abs} \to \mathsf{rep\text{-}abs} \to \mathsf{rep\text{-}abs} \\ & order\_sup && : \mathsf{rep\text{-}abs} \to \mathsf{rep\text{-}abs} \to \textbf{bool} \\ & equal && : \mathsf{rep\text{-}abs} \to \mathsf{rep\text{-}abs} \to \textbf{bool} \\ \} & & \end{aligned}$$

Si on avait défini l'interface d'une collection en laissant apparaître le type **self** dans les types des fonctions, l'interface de la collection des treillis sur les entiers n'aurait été compatible qu'avec des espèces dont le type support est effectivement celui des entiers machine. La substitution de **self** par rep-abs a rendu le type support abstrait. Par la suite, cette interface est compatible avec d'autres implantations des treillis du moment que les cinq fonctions requises y apparaissent et

ont la même signature. Ainsi, à tout endroit où la collection `treillis_entiers` peut apparaître une autre implantation compatible peut lui être substituée. Ces autres implantations peuvent très bien contenir d'autres fonctions en plus des cinq requises.

6.4.2 Interfaces d'espèces

$$I \quad ::= \quad \exists \mathfrak{c}_1 : I_1^{\mathfrak{c}} \ldots \mathfrak{c}_n : I_n^{\mathfrak{c}}.\exists \mathsf{rep\text{-}abs}.\{m_1 : \tau_1, \ldots, m_n : \tau_n\}[\mathbf{self} \leftarrow \mathsf{rep\text{-}abs}]$$

Les interfaces I sont définies pour les espèces. La différence notable avec les interfaces de collection vient du fait qu'une espèce peut prendre des paramètres. Une interface de collection ne possède pas de paramètre car une collection est une implantation complète dans laquelle tout est connu.

Comme déjà dit, la distinction interface collection/interface espèce est syntaxique, cela signifie que la liste des fonctions d'une interface d'espèces a la même signification que pour les interfaces de collections.

Les paramètres de l'interface sont des types. À ce titre, ils peuvent apparaître dans les signatures des fonctions. On remarque que les paramètres des interfaces d'espèce doivent satisfaire des interfaces collections. Cela traduit le fait qu'une espèce ne peut être appliquée qu'à des collections.

Exemple 6.4.2. L'espèce des polynômes à coefficient dans un anneau $\mathfrak{c}$ a l'interface suivante :

$$I^{\mathfrak{c}}_{\text{polynômes}} = \exists \mathfrak{c}.\exists \mathsf{rep\text{-}abs}.\{\begin{array}{ll} \mathit{plus} & : \mathsf{rep\text{-}abs} \to \mathsf{rep\text{-}abs} \to \mathsf{rep\text{-}abs} \\ \mathit{moins} & : \mathsf{rep\text{-}abs} \to \mathsf{rep\text{-}abs} \to \mathsf{rep\text{-}abs} \\ \mathit{from\_list} & : \mathtt{list}(\mathfrak{c} * \mathtt{int}) \to \mathsf{rep\text{-}abs} \\ \mathit{equal} & : \mathsf{rep\text{-}abs} \to \mathsf{rep\text{-}abs} \to \mathsf{rep\text{-}abs} \end{array} \}$$

On remarque que par l'abstraction du type support, le type $\mathfrak{c}$ n'est visible dans l'interface uniquement dans le type de la fonction $\mathit{from\_list}$. Les autres fonctions utilisent le type rep-abs pour représenter un polynôme.

6.4.3 Sous-interface

Maintenant que nous avons donné la définition des interfaces d'espèces et de collections, nous pouvons donner la définition formelle de la notion de compatibilité d'interface.

Définition 6.4.3 Extraction des déclarations d'une interface de collection**.**
Soit une interface de collection $I^{\mathfrak{c}} = \exists \mathsf{rep\text{-}abs}.\{m_1 : \tau_1, \ldots, m_n : \tau_n\}[\mathbf{self} \leftarrow \mathsf{rep\text{-}abs}]$*. On note* $\mathcal{M}$ *la fonction qui renvoie l'ensemble des fonctions de* $I^{\mathfrak{c}}$ *avec leur type. Ainsi, on a* $\mathcal{M}(I^{\mathfrak{c}}) = \{m_1 : \tau_1, \ldots, m_n : \tau_n\}$.

Nous pouvons maintenant définir la relation de sous-interface. Intuitivement, $I_1^{\mathfrak{c}}$ est compatible avec $I_2^{\mathfrak{c}}$ si et seulement si toutes les fonctions de $I_2^{\mathfrak{c}}$ apparaissent dans $I_1^{\mathfrak{c}}$. Dans ce cas, à tout endroit où l'on attend un objet d'interface I_2, on peut appliquer un objet d'interface I_1. La relation de sous-interface est notée $\prec_I$, elle capture cette intuition.

Définition 6.4.4 Sous-interface**.**
La relation de sous-interface est définie sur les interfaces de collection. Soient $I_1^{\mathfrak{c}}$ *et* $I_2^{\mathfrak{c}}$ *deux interfaces. On a* $I_1^{\mathfrak{c}} \prec_I I_2^{\mathfrak{c}}$ *si et seulement si* $\mathcal{M}(I_2^{\mathfrak{c}}) \subseteq \mathcal{M}(I_1^{\mathfrak{c}})$

6.4.4 Application d'interface

Informellement, on passe d'une interface d'espèce à une interface de collection en lui appliquant des collections. Cette opération consiste à substituer les occurrences des paramètres formels au sein de l'interface par les paramètres effectifs. Bien sûr, cette opération est valide si les paramètres effectifs de l'interface sont compatibles avec les paramètres formels. La définition suivante inclut toutes ces remarques.

Définition 6.4.5 Application de collections**.**
Soit une interface I *telle que*

$$I = \exists \mathfrak{c}_1 : I_1^{\mathfrak{c}} \ldots \mathfrak{c}_n : I_n^{\mathfrak{c}}.\exists \mathit{rep\text{-}abs}.\{m_1 : \tau_1, \ldots, m_n : \tau_n\}[\mathbf{self} \leftarrow \mathit{rep\text{-}abs}]$$

Soit un ensemble de n *collections* $\mathfrak{c}'_i : I_i^{\mathfrak{c}'}$ *tel que* $\forall i \in [\![0, n]\!], I_i^{\mathfrak{c}'} \prec_I I_i^{\mathfrak{c}}$*. La nouvelle interface* $I^{\mathfrak{c}'} = I(\mathfrak{c}'_1, \ldots, \mathfrak{c}'_n)$ *qui est obtenue en appliquant* I *aux collections* $\mathfrak{c}'_i$ *a pour définition :*

$$I^{\mathfrak{c}} = \exists \mathit{rep\text{-}abs}.\{m_1 : \tau_1 \ldots m_n : \tau_n\}[\mathbf{self} \leftarrow \mathit{rep\text{-}abs}][\mathfrak{c}_1 \leftarrow \mathfrak{c}'_1] \ldots [\mathfrak{c}_n \leftarrow \mathfrak{c}'_n]$$

6.5 Les espèces

Nous présentons maintenant une définition formelle des espèces. Nous rappelons que nous avons choisi de ne pas montrer les propriétés. De plus, le langage FoCal permet de faire des fonctions mutuellement récursives et impose que des preuves de terminaison de celles-ci. Par soucis de clarté, nous présentons un formalisme sur les espèces qui n'explicite pas les fonctions récursives ainsi que leur preuve. Notre méthodologie fonctionne en présence de fonctions mutuellement récursives. Essentiellement, une espèce est la donnée de cinq ensembles :

$$\begin{array}{lll}
\mathit{def}_{\mathsf{s}} ::= \{ & \mathfrak{c}_1 : I_1^{\mathfrak{c}}, \ldots, \mathfrak{c}_n : I_n^{\mathfrak{c}}; & \text{paramètres collections} \\
& \mathsf{S}_1(\mathfrak{c}_1^1, \ldots, \mathfrak{c}_{k_1}^1), \ldots, \mathsf{S}_p(\mathfrak{c}_1^p, \ldots, \mathfrak{c}_{k_p}^p); & \text{espèces héritées} \\
& \mathsf{rep} = \tau\text{-rep}; & \text{type support} \\
& m_1 : \tau_1, \ldots, m_o : \tau_o; & \text{fonctions uniquement déclarées} \\
& n_1 = <x_1^1, \ldots, x_{n_1}^1 \rightsquigarrow e_1> : \tau'_1, & \\
& \qquad \ldots, & \text{fonctions définies} \\
& n_r = <x_1^r, \ldots, x_{n_r}^r \rightsquigarrow e_r> : \tau'_r\} &
\end{array}$$

Le premier ensemble spécifie les paramètres $\mathfrak{c}_1 : I_1^{\mathfrak{c}}, \ldots, \mathfrak{c}_n : I_n^{\mathfrak{c}}$ de l'espèce. Ce sont les paramètres collections de l'espèce. Comme déjà dit plus haut, les paramètres collections jouent le rôle de type au sein de l'espèce et chaque $\mathfrak{c}_i$ peut être présent dans le type des fonctions de l'espèce ainsi que dans le type support.

La deuxième composante est la liste des espèces héritées. Il faut préciser que pour pouvoir hériter d'une espèce, il faut que tous ses paramètres soient connus. Le langage focal interdit d'hériter d'une espèce appliquée partiellement.

Le troisième champ de l'espèce correspond à la définition du type support. Cette définition peut prendre les valeurs suivantes :

$$\tau\text{-rep} \quad ::= \quad \tau \mid \mathsf{rep\text{-}abs}$$

Ainsi, le type support peut prendre comme définition un type de FoCal τ. Dans ce cas, l'espèce *définit le type support* et le mot-clef **self** devient un alias de ce type. Si le type support est déclaré par l'expression rep-abs, cela signifie que l'espèce ne précise rien sur ce type. Il y a alors deux possibilités. Soit le type support est défini dans au moins une des espèces héritées, alors le type support de l'espèce deviendra après résolution de l'héritage celui des espèces parentes si les types supports hérités sont identiques. Soit le type support n'est défini dans aucune espèce parente, le type support restera alors abstrait.

La quatrième composante contient l'ensemble des déclarations de fonctions de l'espèce. Il s'agit des fonctions qui ne sont connues qu'au travers d'une signature. Si une espèce parente contient la définition/déclaration d'une fonction de même nom, il faut vérifier que les signatures sont compatibles.

Enfin, le dernier champ spécifie l'ensemble des définitions de fonctions. Chaque fonction est définie par une fermeture qui contient l'ensemble des variables que la fonction prend en argument et l'expression qui forme le corps de la fonction.

Définition 6.5.1 Espèce bien formée**.**
Une définition d'espèce def_{s} *est* bien formée *si pour chaque définition de fonction*

$$n = <x_1, \ldots, x_n \rightsquigarrow e> : \tau \ \textit{avec} \ \ FV(e)/\{x_1, \ldots, x_n\} = \emptyset$$

Valeur d'une espèce

La valeur d'une espèce correspond à la définition de l'espèce après la résolution de l'héritage. Il s'agit d'une « définition brute » de l'espèce. Toutes les définitions des fonctions obtenues par héritage sont recopiées dans la définition.

$$\begin{array}{lll} val_{\mathsf{s}} ::= \{ \ \mathfrak{c}_1 : I_1^{\mathfrak{c}}, \ldots, \mathfrak{c}_n : I_n^{\mathfrak{c}}; & & \text{paramètres collections} \\ \quad \mathsf{rep} = \tau\text{-rep}; & & \text{type support} \\ \quad m_1 : \tau_1, \ldots, m_o : \tau_o; & & \text{fonctions uniquement déclarées} \\ \quad n_1 = <x_1^1, \ldots, x_{n_1}^1 \rightsquigarrow e_1> : \tau_1', & & \\ \qquad\qquad \ldots, & & \text{fonctions définies} \\ \quad n_r = <x_1^r, \ldots, x_{n_r}^r \rightsquigarrow e_r> : \tau_r' \} & & \end{array}$$

Ainsi la valeur d'une espèce contient les mêmes champs que la définition d'une espèce excepté le deuxième champ. On remarque en particulier qu'une valeur d'espèce contient toujours les paramètres d'origine.

Dans cette définition, l'héritage a été résolu entièrement. La résolution de l'héritage s'effectue en réduisant les commandes d'héritage de la droite vers la gauche. Lors de cette résolution, les paramètres appliqués à l'espèce héritée sont instanciés en remplaçant les occurrences des paramètres formels par les valeurs des paramètres effectifs. Pour chaque commande d'héritage l'ensemble des définitions de fonctions est recopié dans la définition de l'espèce courante. Pour chaque fonction apparaissant dans l'espèce parente, l'héritage est résolu de la façon suivante :

– si la fonction est *définie* dans l'espèce courante : La définition de l'espèce courante est retenue et les types de la fonction dans l'espèce courante et dans l'espèce parente sont comparés. Si les types ne s'unifient pas alors la résolution de l'héritage échoue ;

– si la fonction est *déclarée* dans l'espèce courante. Les signatures de la fonction dans l'espèce courante et dans l'espèce parente sont comparées. Si les signatures ne sont pas identiques, l'héritage échoue. Si la fonction est définie dans l'espèce parente alors sa définition est recopiée au sein de l'espèce courante ;

– si la fonction n'est *ni déclarée*, *ni définie* dans l'espèce courante. La signature et l'éventuelle définition de la fonction est recopiée de l'espèce parente vers l'espèce courante.

On remarque qu'une fermeture de fonction d'espèce ne contient pas d'environnement. Une fonction est un membre d'une espèce, l'environnement dans lequel est définie la fonction ne contient aucune déclaration locale. Une fonction ne dépend que de l'existence des autres fonctions de l'espèce et des opérateurs de base du langage. Le type τ_i' définit le type de la fonction.

Définition 6.5.2 Résolution d'héritage**.**
Soit def_{s} la définition d'une espèce. On définit $expand_{\mathsf{s}}$ qui permet de transformer la définition d'une espèce vers sa valeur. Ainsi, si val_{s} est la valeur de def_{s} après résolution de l'héritage et que de plus l'interface de val_{s} est I, alors l'ensemble d'espèce $\mathcal{S}$ on a $expand_{\mathsf{s}}(def_{\mathsf{s}}, \mathcal{S}) = val_{\mathsf{s}} : I$.

Pour des soucis de concision et pour ne pas développer des parties non utiles pour la compréhension de FoCal, nous ne donnons pas la définition de cette fonction et admettons qu'elle existe. On peut retrouver sa définition dans [Fec05].

Définition 6.5.3 Fonctions d'une valeur d'espèce**.**
On étend la définition de la fonction $\mathcal{M}$ sur les valeurs d'espèces pour les fonctions définies. Ainsi, pour une espèce S de valeur val_{s}, $\mathcal{M}(val_{\mathsf{s}})$ est l'ensemble des fonctions définies de S.

La fonction suivante permet d'extraire d'une espèce l'ensemble de ses paramètres.

Définition 6.5.4 Paramètres d'une espèce**.**
Soit S une espèce de valeur val_{s}. On note $\mathcal{P}(val_{\mathsf{s}}) = \{\mathfrak{c}_1 : I_1^{\mathfrak{c}}, \ldots, \mathfrak{c}_n : I_n\}$ l'ensemble des paramètres de S.

Définition 6.5.5 Espèce complète**.**
Soit S une espèce. S est une espèce complète *si et seulement si la valeur de S ne contient pas de fonctions déclarées non définies et le type support est défini par un type τ. On définit le prédicat $Comp(\mathsf{S})$ qui est vrai si S est une espèce complète.*

Les espèces complètes sont les candidates à la création d'une collection. Dans une espèce complète, toutes les fonctions sont connues. La structure algébrique décrite par l'espèce est entièrement définie, on peut envisager de créer un nouveau type correspondant à l'espèce. Il ne reste plus qu'à connaître la valeur des éventuels paramètres de l'espèce pour créer une nouvelle collection.

Définition 6.5.6 Valeur d'espèce bien formée**.**
Une valeur d'espèce val_{s} *est* bien formée *si pour toute définition de fonction* n *telle que :*

$$n = <x_1, \ldots, x_n \rightsquigarrow e> : \tau, \text{ on a } FV(e)/\{x_1, \ldots, x_n\} = \emptyset$$

6.6 Les collections

Les collections sont créées à partir des espèces complètes. Formellement, la valeur d'une collection est la donnée d'un ensemble de fonctions :

$$\begin{aligned} val_{\mathfrak{c}} ::= \{\ & n_1 = <x_1^1, \ldots, x_{n_1}^1 \rightsquigarrow e_1> : \tau_1', \\ & \ldots, \\ & n_r = <x_1^r, \ldots, x_{n_r}^r \rightsquigarrow e_r> : \tau_r' \\ \} & \end{aligned}$$

La création d'une collection consiste à effacer de la définition de l'espèce le champ `rep` et à substituer toutes les occurrences de **self** par le nom de la collection. Comme l'espèce est complète et que les paramètres de l'espèce sont connus, les seuls champs qui apparaissent dans la collection sont les fonctions définies. La définition de la collection est alors l'unique donnée des fonctions définies de l'espèce à partir de laquelle la collection a été créée.

Définition 6.6.1 Fermeture d'une fonction**.**
Soit une valeur de collection $val_{\mathfrak{c}}$*. Si* n *est une fonction de* $val_{\mathfrak{c}}$ *et* $<x_1, \ldots, x_n \rightsquigarrow e>$ *la fermeture associée à* n *alors on note* $val_{\mathfrak{c}}(n) = <x_1, \ldots, x_n \rightsquigarrow e>$*.*

Théorème 6.6.2. *Dans une collection, la définition des fonctions ne contient pas de référence au mot-clef* **self***.*

Définition 6.6.3 Collection bien formée**.**
Une valeur de collection est bien formée *si toutes ses fonctions sont bien formées.*

6.7 Sémantique des expressions

Une expression FoCal ne peut pas être évaluée tant qu'elle figure dans une espèce. Les espèces sont des objets par nature non exécutables. Si la définition d'une fonction contient un appel à une autre fonction, pour évaluer l'expression il faut que la fonction appelée soit définie. Or, une espèce peut contenir des fonctions uniquement déclarées. Dans un tel cas, l'évaluation de l'expression ne peut pas aboutir. L'évaluation des appels de fonction peut bloquer lorsqu'on se place au niveau des espèces. Bien entendu, ce n'est pas toujours le cas, l'ensemble des fonctions utilisées lors d'un calcul peut très bien être défini ou bien les espèces utilisées pour un calcul peuvent aussi être complètes. Dans ces deux cas, l'évaluation aboutit. *A contrario*, les collections sont des objets complets et fonctionnels. Toutes les fonctions contenues dans une collection sont définies par définition. L'évaluation d'une expression dans un environnement composé de collections aboutit toujours. Les collections offrent donc un environnement favorable à la définition de la sémantique des expressions.

Avant de donner la sémantique des expressions FoCal nous devons préciser la définition des environnements.

Définition 6.7.1 Environnements de collections**.**
On définit les environnements de collection de la manière suivante :

$$\begin{array}{rcl} \mathcal{C} & ::= & \emptyset \\ & | & \mathcal{C}, (\mathfrak{c} \leftarrow val_{\mathfrak{c}} : I^{\mathfrak{c}}) \end{array}$$

Les environnements de collection permettent de lier des noms de collections à des valeurs de collection. On ajoute à cette définition une notation pour dénoter l'appartenance d'une définition à un environnement $\mathcal{C}$. $(\mathfrak{c} \leftarrow val_{\mathfrak{c}} : I^{\mathfrak{c}}) \in \mathcal{C}$ est notée $\mathcal{C}(\mathfrak{c}) = val_{\mathfrak{c}} : I^{\mathfrak{c}}$.

Définition 6.7.2 Environnements de variable**.**
Nous définissons l'environnement des variables de manière similaire à l'environnement des collections :

$$\begin{array}{rcl} \mathcal{E}_v & ::= & \emptyset \\ & | & \mathcal{E}_v, (x \leftarrow v) \end{array}$$

On utilise la notation $\mathcal{E}_v(x) = v$ si et seulement si $(x \leftarrow v) \in \mathcal{E}_v$.

Cet environnement permet d'associer des valeurs à des variables. Il est utilisé pour stocker les valeurs des variables locales à une expression.

Définition 6.7.3 Surcharge**.**
On définit l'opérateur de surcharge $\oplus$ des environnements. Soit $\mathcal{E}_v$ un environnement, x une variable et v une valeur. $\mathcal{E}_v \oplus (x \leftarrow v)$ est un nouvel environnement tel que :

$$\begin{array}{l} (\mathcal{E}_v \oplus (x \leftarrow v))(x) = \mathcal{E}_v(x) \\ (\mathcal{E}_v \oplus (x \leftarrow v))(x') = \mathcal{E}_v(x') \quad \textit{pour } x' \neq x \end{array}$$

Définition 6.7.4 Antirestriction**.**
On définit l'opérateur d'antirestriction $\lhd$ sur les environnements. Soit $\mathcal{E}_v$ un environnement et X un ensemble de variables. $X \lhd \mathcal{E}_v$ est un nouvel environnement où pour tout $x \in X$, $(X \lhd \mathcal{E}_v)(x)$ n'est pas défini et pour $x \notin X$, $(X \lhd \mathcal{E}_v)(x) = \mathcal{E}_v(x)$.

Définition 6.7.5 Fusion**.**
On définit l'opérateur de fusion $\otimes$ de deux environnements. Soient $\mathcal{E}_v$ et ${\mathcal{E}_v}'$ deux environnements. $\mathcal{E}_v \otimes {\mathcal{E}_v}'$ est un nouvel environnement tel que :

$$\begin{array}{l} (\mathcal{E}_v \otimes {\mathcal{E}_v}')(x) = \mathcal{E}_v(x) \quad \textit{si } \exists v, (x \leftarrow v) \in {\mathcal{E}_v}' \\ (\mathcal{E}_v \otimes {\mathcal{E}_v}')(x) = {\mathcal{E}_v}'(x) \quad \textit{si } \nexists v, (x \leftarrow v) \in \mathcal{E}_v \\ (\mathcal{E}_v \otimes {\mathcal{E}_v}')(x) \textit{ non définie si } \mathcal{E}_v(x) \textit{ et } {\mathcal{E}_v}'(x) \textit{ ne sont pas définis} \end{array}$$

À chaque fois qu'on définira une sorte d'environnement, on supposera que les opérateurs de surcharge et de fusion de ces environnements sont définis et qu'ils ont une définition similaire à celle que nous avons donnée.

Définition 6.7.6 Interprétation des opérateurs**.**
On dispose d'une fonction d'interprétation des opérateurs de FoCal. Soit $\sharp f^i$ un opérateur FoCal

d'arité i et v_1, ..., v_i des valeurs. La fonction d'interprétation associe à chaque valeur symbole de fonction une relation et à chaque valeur de FoCal une valeur de la relation. Ainsi, on a :

$$eval\_op(\sharp f^i, v_1, \ldots, v_i) = eval\_op(\sharp f^i)(eval\_op(v_1), \ldots, eval\_op(v_i))$$

Afin de donner une sémantique au filtrage par motif, on définit la fonction de filtrage qui permet de mettre en concordance un filtre et une valeur.

Définition 6.7.7 Filtrage**.**
La fonction $\mathcal{F}$ effectue un filtrage, elle prend un motif pat et une valeur v. Elle retourne l'environnement de variable $\mathcal{E}_v$ qui lie les variables du motif si le motif et la valeur concordent et échoue en retournant `fail` *si la valeur v ne satisfait pas le motif pat. On définit cette fonction par induction structurelle sur le motif pat (on rappelle que les filtres sont linéaires) :*

$$\begin{array}{lll}
\mathcal{F}(_, v) = \emptyset & & \\
\mathcal{F}(x, v) = (x \leftarrow v) & & \\
\mathcal{F}(C^0, C^0) = \emptyset & & \\
\mathcal{F}(C^0, C'^0) = \texttt{fail} & & Si\ C^0 \neq C'^0 \\
\mathcal{F}(C^n(pat_1, \ldots, pat_n), C'^m(v_1, \ldots, v_m)) = \texttt{fail} & & Si\ C^n \neq C'^m \\
\mathcal{F}(C^n(pat_1, \ldots, pat_n), C^n(v_1, \ldots, v_n)) = & \mathcal{F}(pat_1, v_1) \otimes \ldots \otimes & Si\ \forall i \in [\![1, n]\!], \\
 & \mathcal{F}(pat_n, v_n) & \mathcal{F}(pat_i, v_i) \neq \texttt{fail} \\
\mathcal{F}(C^n(pat_1, \ldots, pat_n), C^n(v_1, \ldots, v_n)) = \texttt{fail} & & sinon
\end{array}$$

Exemple 6.7.8. Soient un motif *pat* et une valeur v tels que :

$$\begin{array}{ll}
pat = C(A, x, D(y)) & \text{et} \\
v = C(A, E(B, 33), D(F)) &
\end{array}$$

La fonction de filtrage nous donne :

$$\begin{array}{rcl}
\mathcal{F}(pat, v) = & \mathcal{F}(A, A) \otimes \mathcal{F}(x, E(B, 33)) \otimes \mathcal{F}(D(y), D(F)) \\
= & \emptyset \otimes (x \leftarrow E(B, 33)) \otimes \mathcal{F}(y, F) \\
= & (x \leftarrow E(B, 33)), (y \leftarrow F)
\end{array}$$

La sémantique des expressions FoCal est présentée à l'aide de jugements de la forme suivante :

$$\mathcal{E}_v; \mathcal{C} \vdash e \rhd v$$

Un tel jugement se lit « l'expression e s'évalue en la valeur v dans l'environnement de collections du programme $\mathcal{C}$ et l'environnement de variables $\mathcal{E}_v$ ». L'environnement $\mathcal{C}$ contient l'ensemble des collections définies dans le programme et $\mathcal{E}_v$ contient les valeurs des variables définies.

La figure 6.3 présente les règles d'évaluation des expressions FoCal. Ces règles sont, pour la plupart, très classiques. Précisons que dans cette sémantique le cas d'un appel de fonction sur le mot-clef **self** n'est pas traité. Le théorème 6.6.2, nous assure que, pendant l'évaluation d'une expression, ce cas n'arrivera jamais.

La règle FVar évalue une variable. La règle consiste à retourner la valeur de la variable enregistrée dans l'environnement $\mathcal{E}_v$.

$$\frac{\mathcal{E}_v(x) = v}{\mathcal{E}_v;\mathcal{C} \vdash x \rhd v}\ \text{FVAR} \qquad \frac{\mathcal{E}_v;\mathcal{C} \vdash e_1 \rhd \texttt{true} \quad \mathcal{E}_v;\mathcal{C} \vdash e_2 \rhd v_2}{\mathcal{E}_v;\mathcal{C} \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \rhd v_2}\ \text{FIF\_TRUE}$$

$$\frac{}{\mathcal{E}_v;\mathcal{C} \vdash i \rhd i}\ \text{FENTIER} \qquad \frac{\mathcal{E}_v;\mathcal{C} \vdash e_1 \rhd \texttt{false} \quad \mathcal{E}_v;\mathcal{C} \vdash e_3 \rhd v_3}{\mathcal{E}_v;\mathcal{C} \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \rhd v_3}\ \text{FIF\_FALSE}$$

$$\frac{}{\mathcal{E}_v;\mathcal{C} \vdash C^0 \rhd C^0}\ \text{FCONSTANTE} \qquad \frac{\mathcal{E}_v;\mathcal{C} \vdash e_1 \rhd v_1 \quad \mathcal{E}_v \oplus (x \leftarrow v_1);\mathcal{C} \vdash e_2 \rhd v_2}{\mathcal{E}_v;\mathcal{C} \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \rhd v_2}\ \text{FLET}$$

$$\frac{\mathcal{E}_v;\mathcal{C} \vdash e_1 \rhd v_1 \quad \ldots \quad \mathcal{E}_v;\mathcal{C} \vdash e_n \rhd v_n}{\mathcal{E}_v;\mathcal{C} \vdash C^n(e_1,\ldots,e_n) \rhd C^n(v_1,\ldots,v_n)}\ \text{FCONSTRUCTEUR}$$

$$\frac{\begin{array}{c}\mathcal{E}_v;\mathcal{C} \vdash e_1 \rhd v_1 \quad \ldots \quad \mathcal{E}_v;\mathcal{C} \vdash e_n \rhd v_n \\ \mathcal{C}(\mathfrak{c}) = val_{\mathfrak{c}} \quad val_{\mathfrak{c}}(m) = <x_1 \ldots x_n \rightsquigarrow e_f> \\ (x_1,v_1),\ldots,(x_n,v_n);\mathfrak{c};\mathcal{C} \vdash e_f \rhd v_f\end{array}}{\mathcal{E}_v;\mathcal{C} \vdash \mathfrak{c}!m(e_1,\ldots,e_n) \rhd v_f}\ \text{FF\_N\_COLL}$$

$$\frac{\begin{array}{c}\mathcal{E}_v;\mathcal{C} \vdash e \rhd v \quad \mathcal{F}(pat_i,v) = \mathcal{E}_v' \quad 1 \le i \le n \\ \forall 1 \le j < i,\ \mathcal{F}(pat_j,v) = \texttt{fail} \quad \mathcal{E}_v \otimes \mathcal{E}_v';\mathcal{C} \vdash e_i \rhd v_i\end{array}}{\mathcal{E}_v;\mathcal{C} \vdash \begin{array}{l}\textbf{match } e \textbf{ with} \\ \mid pat_1 \rightarrow e_1 \\ \quad\vdots \\ \mid pat_n \rightarrow e_n\end{array} \rhd v_i}\ \text{FFILTRAGE}$$

$$\frac{\mathcal{E}_v;\mathcal{C} \vdash e_1 \rhd v_1 \quad \ldots \quad \mathcal{E}_v;\mathcal{C} \vdash e_n \rhd v_n \quad \text{eval\_op}(\sharp f^n, v_1,\ldots,v_n) = v}{\mathcal{E}_v;\mathcal{C} \vdash \sharp f^n(e_1,\ldots,e_n) \rhd v}\ \text{F\_OP}$$

FIGURE 6.3 – Sémantique des expressions FoCal

Les règles FEntier et Constante permettent d'évaluer un entier et un constructeur constant. Ces deux règles retournent les deux expressions à évaluer sans modification.

Les deux règles FIf_true et FIf_false définissent la sémantique de la conditionnelle. Si la condition est vraie, la valeur de l'expression est celle de la partie **then**, si la condition est fausse, la valeur de l'expression est celle de la partie **else**.

La règle FLet évalue une construction **let** ... **in**. On commence par évaluer l'expression e_1 en une valeur v_1 dans l'environnement courant. On évalue ensuite l'expression e_2 dans l'environnement $\mathcal{E}_v$ surchargé par x qui prend la valeur v_1.

La règle FConstructeur évalue l'application d'un constructeur C à des expressions. La règle détermine la valeur v_i correspondant à chacune des expressions e_i. Ensuite, l'expression $C(e_1, \ldots, e_n)$ s'évalue en la valeur $C(v_1, \ldots, v_n)$.

La règle FF_n_coll permet d'évaluer un appel de fonction dans une collection. On commence par évaluer les e_i paramètres de la fonction. On recherche ensuite la valeur $val_{\mathfrak{c}}$ de la collection $\mathfrak{c}$ dans l'environnement des collections. Ceci fait, la règle recherche dans la valeur $val_{\mathfrak{c}}$ la définition de la fonction appelée m. On évalue ensuite le corps de la fonction e_f dans un environnement où les différents paramètres x_i de la fonction reçoivent les valeurs correspondantes aux paramètres effectifs.

La règle FOp effectue l'appel d'un opérateur. Le résultat de l'évaluation est donné par la fonction d'interprétation des opérateurs eval_op() définis précédemment.

La règle FFiltrage permet d'évaluer un filtrage pas motif. La règle commence par évaluer l'expression filtrée en une valeur v. Au premier filtre pat_i rencontré pour lequel le filtrage $\mathcal{F}$ réussit, l'expression e_i associée au filtre est évaluée dans l'environnement de départ $\mathcal{E}_v$ surchargé par les définitions de l'environnement retourné par $\mathcal{F}$. La valeur v_i obtenue est alors le résultat de l'évaluation du filtrage.

6.8 Programme FoCal

À partir des définitions des expressions FoCal, des espèces et des collections, nous pouvons définir les programmes FoCal. Nous définissons les programmes FoCal dans une syntaxe différente de celle présentée au chapitre 3. Cette syntaxe se veut courte et concise. Un programme FoCal est défini par la syntaxe suivante :

cmd	::=	$\mathsf{S} = def_{\mathsf{s}}$	création d'espèce
	|	$\mathfrak{c}$ impl S	création de collection
	|	$\mathfrak{c}$ impl $\mathsf{S}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n)$	création de collection (avec paramètres)
	|	e	expression
prog	::=	$\emptyset$	programme vide
	|	*prog*; *cmd*	séquence de commandes

Formellement, un programme FoCal est décrit par une suite de commandes. Il y a quatre sortes de commandes :

- la commande $\mathsf{S} = def_{\mathsf{s}}$ permet d'introduire dans l'environnement une nouvelle espèce S avec pour définition def_{s} ;
- la commande $\mathfrak{c}$ impl S permet de créer une nouvelle collection à partir d'une espèce complète S ;
- la commande $\mathfrak{c}$ impl $\mathsf{S}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n)$, tout comme la commande précédente permet de créer une nouvelle collection. La nouvelle collection est créée à partir d'une espèce S dont les paramètres ont été instanciés par les collections $\mathfrak{c}_1, \ldots, \mathfrak{c}_n$;

– la commande *e* permet de définir une expression à évaluer, par exemple, l'appel d'une fonction d'une collection.

Dans ce qui suit, nous donnons une définition de la sémantique des programmes FoCal. Cette sémantique est présente pour donner un aperçu du fonctionnement d'un programme FoCal et est une simplification de la sémantique réellement définie. La sémantique de FoCal est définie en fait à partir d'un langage intermédiaire. Les programmes FoCal sont compilés vers ce langage. Dans le langage intermédiaire, les directives d'héritage sont résolues et il ne reste plus que les définitions de collections. La sémantique est alors définie sur ce langage intermédiaire. Ces deux étapes de définition de la sémantique sont mélangées dans le discours qui suit.

Les programmes FoCal sont évalués dans un environnement d'espèce et de collection. Définissons les.

Définition 6.8.1 Environnement d'espèce**.**
On définit les environnements d'espèce $\mathcal{S}$ de la manière suivante :

$$\begin{array}{rcl} \mathcal{S} & ::= & \emptyset \\ & | & \mathcal{S}, (\mathsf{S} \leftarrow val_{\mathsf{s}} : I) \end{array}$$

Les environnements d'espèce lient des noms d'espèce à des valeurs. On définit $\mathcal{S}(\mathsf{S}) = val_{\mathsf{s}} : I$ si et seulement si $(\mathsf{S} \leftarrow val_{\mathsf{s}} : I) \in \mathcal{S}$.

Nous pouvons maintenant donner la sémantique des programmes FoCal. Cette sémantique est donnée par des jugements de la forme suivante :

$$\mathcal{C}; \mathcal{S} \vdash prog \rhd \mathcal{C}'; \mathcal{S}'$$

Ces jugements d'évaluation se lisent de la manière suivante : dans les environnements de collection $\mathcal{C}$ et d'espèce $\mathcal{S}$, le programme *prog* s'évalue en un environnement de collection $\mathcal{C}$' et un environnement d'espèce $\mathcal{S}$'.

Les règles d'évaluation des commandes FoCal sont données dans la figure 6.4. Voici la signification de chacune de ces règles :

La règle CREAT_SPEC permet d'ajouter dans l'environnement des espèces une nouvelle espèce. La nouvelle espèce est S et sa définition est def_{s}. La définition de l'espèce est d'abord transformée en une valeur en résolvant l'héritage, on obtient alors la valeur val_{s} d'interface I. L'environnement des espèces est ensuite augmenté de la nouvelle espèce S liée à sa valeur et à son interface.

La règle IMPL permet d'ajouter à l'environnement des collections une nouvelle collection $\mathfrak{c}$ créée à partir d'une espèce S non paramétrée. On cherche d'abord dans l'environnement des espèces $\mathcal{S}$ la valeur val_{s} d'interface I de l'espèce S. La règle vérifie ensuite si val_{s} définit bien une espèce complète et qu'elle n'attend pas de paramètre. Si tel est le cas, l'environnement des collections est augmenté de la nouvelle collection $\mathfrak{c}$ dont la définition est l'ensemble des fonctions définies dans val_{s} dans lesquelles on remplace les occurrences de **self** dans leur type par $\mathfrak{c}$.

La règle IMPL_PARAM est similaire à la règle précédente. Elle permet de créer une nouvelle collection $\mathfrak{c}$ à partir d'une espèce S appliquée à des collections $\mathfrak{c}_i$. On commence par obtenir la valeur de S val_{s} et son interface I. La règle vérifie ensuite si S est complète et obtient ses paramètres $\mathfrak{c}_i$ d'interface I'_i. On obtient ensuite les interfaces des paramètres effectifs $I^{\mathfrak{c}}_i$. Il faut

$$\frac{\text{expand}_{\mathsf{s}}(def_{\mathsf{s}}) = val_{\mathsf{s}} : I}{\mathcal{C};\mathcal{S} \vdash \mathsf{S} = def_{\mathsf{s}} \rhd \mathcal{C};\mathcal{S},(\mathsf{S} \leftarrow val_{\mathsf{s}} : I)}\ \textsc{Creat\_spec}$$

$$\frac{\begin{array}{c}\mathcal{S}(\mathsf{S}) = val_{\mathsf{s}} : I \\ \text{Comp}(val_{\mathsf{s}}) \qquad \mathcal{P}(val_{\mathsf{s}}) = \emptyset\end{array}}{\mathcal{C};\mathcal{S} \vdash \mathfrak{c}\ \mathsf{impl}\ \mathsf{S} \rhd \mathcal{C},(\mathfrak{c} \leftarrow \{\mathcal{M}(S)[\mathbf{self} \leftarrow \mathfrak{c}]\});\mathcal{S}}\ \textsc{Impl}$$

$$\frac{\begin{array}{c}\mathcal{S}(\mathsf{S}) = val_{\mathsf{s}} : I \qquad \text{Comp}(val_{\mathsf{s}}) \\ \mathcal{P}(val_{\mathsf{s}}) = \{\mathfrak{c}'_1 : {I^{\mathfrak{c}'}}_1, \ldots, \mathfrak{c}'_n : {I^{\mathfrak{c}'}}_n\} \\ \mathcal{C}(\mathfrak{c}_1) = val_{\mathfrak{c}1} : I_1^{\mathfrak{c}} \qquad \ldots \qquad \mathcal{C}(\mathfrak{c}_n) = val_{\mathfrak{c}n} : I_n^{\mathfrak{c}} \\ I_1^{\mathfrak{c}} \prec_I {I^{\mathfrak{c}'}}_1 \qquad \ldots \qquad I_n^{\mathfrak{c}} \prec_I {I^{\mathfrak{c}'}}_n \qquad I^{\mathfrak{c}} = I(\mathfrak{c}_1, \ldots, \mathfrak{c}_n)\end{array}}{\mathcal{C};\mathcal{S} \vdash \mathfrak{c}\ \mathsf{impl}\ \mathsf{S}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n) \rhd \mathcal{C},(\mathfrak{c} \leftarrow \{\mathcal{M}(S)[\mathbf{self} \leftarrow \mathfrak{c}][\mathfrak{c}'_i \leftarrow \mathfrak{c}_i] : I^{\mathfrak{c}}\});\mathcal{S}}\ \textsc{Impl\_param}$$

$$\frac{\emptyset;\mathcal{C} \vdash e \rhd v}{\mathcal{C};\mathcal{S} \vdash e \rhd \mathcal{C};\mathcal{S}}\ \textsc{Eval\_expr}$$

FIGURE 6.4 – Sémantique des programmes FoCal

vérifier alors que les paramètres effectifs sont bien compatibles avec les paramètres attendus. Pour cela, on regarde si les relations de sous-interface $I_i^{\mathfrak{c}} \prec_I {I^{\mathfrak{c}'}}_n$ sont correctes. La règle calcule ensuite l'interface de la nouvelle collection $I^{\mathfrak{c}}$ en appliquant l'interface de I de l'espèce aux paramètres. Quand tout cela est fait, la règle retourne l'environnement des collections augmenté de la nouvelle collection qui contient les fonctions de S dans lesquelles les occurrences du type **self** sont remplacées par $\mathfrak{c}$ et les occurrences des $\mathfrak{c}'_i$ sont remplacées par les $\mathfrak{c}_i$.

La règle EVAL_EXPR permet d'évaluer une expression. Cette règle se contente d'évaluer e en utilisant les règles d'évaluations définies en 6.7. Elle ne modifie pas les environnements de collection et d'espèce.

Une fois les règles d'évaluations des commandes définies, on peut donner les règles d'évaluation des programmes FoCal. Pour évaluer un programme FoCal, on part des environnements de collection et d'espèce vides et on évalue séquentiellement les commandes qui forment le programme FoCal à l'aide des règles de la figure 6.4.

Théorème 6.8.2. *Soit une collection* $\mathfrak{c}$*. Les expressions liées aux fonctions apparaissant dans* $\mathfrak{c}$ *ne contiennent pas de référence à* **self***. Ceci est vrai grâce aux deux règles de création de collection* IMPL *et* IMPL_PARAM *qui substituent toutes les occurrences de* **self** *par le nom de la collection créée.*

6.9 Synthèse

Dans ce chapitre, nous avons défini formellement le langage FoCal. Nous avons volontairement présenté un sous-ensemble du langage afin de nous concentrer sur les aspects les plus importants pour tester les propriétés. En particulier, nous avons mis de côté les traits orientés spécification des programmes FoCal. Nous n'avons pas parlé des preuves et de leur traitement au sein des espèces et des collections dans le cadre de l'héritage. Le langage d'expression a été

alléger de l'ordre supérieur, bien qu'il fasse partie du langage FoCal, car notre méthode de test n'intègre pas ces aspects.

Dans le chapitre qui suit nous présentons notre méthode de génération de jeux de test qui utilise la programmation par contraintes.

Chapitre 7

Jeux de test par résolution de contraintes

Dans ce chapitre nous présentons notre technique de génération des jeux de test qui utilise la résolution de contraintes. Schématiquement, nous convertissons la précondition des propriétés élémentaires sous test en un système de contraintes. La résolution du système nous donne alors une solution qui est directement un des jeux de test valides recherchés. Nous fabriquons donc des jeux de test qui sont valides par construction.

Nous considérons dans ce chapitre que nous cherchons à traduire une expression FoCal en un ensemble de contraintes. Nous qualifions de programme l'expression et l'environnement dans lequel se place cette expression, c'est-à-dire, l'ensemble des collections.

La traduction de la précondition implique qu'on convertisse un programme FoCal, ou plus précisément les fonctions concernées par la précondition, en un ensemble de contraintes. Nous effectuons cette traduction en deux étapes.

La première étape consiste à transformer le programme testé, ce programme est la donnée de l'ensemble des fonctions qui interviennent dans la définition de la précondition. Cette étape transforme les fonctions de manière à faciliter leur traduction en un ensemble de contraintes. Les fonctions que nous souhaitons transformer sont celles qui sont utilisées dans la précondition ainsi que celles qui sont utilisées par ces dernières. Le programme testé est transformé en un programme sous une forme intermédiaire monadique, un programme dans lequel tous les calculs intermédiaires sont nommés. Seuls les aspects fonctionnels du programme FoCal nous intéressent. Nous ignorons donc ici les notions d'espèces et de collection car nous souhaitons nous concentrer sur les descriptions des fonctions.

Dans la deuxième étape, nous convertissons le programme normalisé en un « environnement de contraintes ». Cet environnement de contraintes fournit l'équivalent des fonctions du programme sous test sous la forme de fonctions générant des contraintes. Une fois le programme FoCal converti en contraintes, nous pouvons convertir la précondition en un ensemble de contraintes.

Dans la suite, nous commençons par présenter le langage FoCal Monadique (FMON). Il s'agit du langage intermédiaire vers lequel nous transformons les programmes FoCal. Ensuite, nous formaliserons la traduction et la normalisation des programmes FoCal vers FMON. Nous accompagnons cette traduction d'une preuve de correction et de complétude qui nous assurent que le programme FMON résultant donne les mêmes résultats que le programme FoCal d'origine. Une fois l'étape de traduction vers FMON présentée, nous présenterons le langage de contraintes

sur lequel nous nous appuyons. Cette présentation est suivie de la formalisation de la traduction des programmes FMON vers le langage à contraintes. Nous n'aborderons pas ici la résolution des contraintes mais nous définissons un prédicat qui permet de vérifier une solution du système de contraintes. Ce prédicat permet de décider uniquement des solutions sur les contraintes obtenues par la traduction. Il ne décide pas de manière générale des solutions sur l'ensemble des contraintes définissables dans le langage de contraintes. Enfin, nous donnerons une preuve de correction et de complétude de cette deuxième traduction.

7.1 Le langage FMON

Avant de traduire les programmes FoCal en un système de contraintes, nous effectuons d'abord une mise en forme normale de ceux-ci. Cette normalisation a deux buts. D'une part, il s'agit d'obtenir un programme qui ne contient plus les traits orientés objet et spécifications du langage FoCal. D'autre part, on met les programmes sous une forme intermédiaire monadique, forme dans laquelle tous les calculs intermédiaires sont nommés.

On se place ici après résolution de l'héritage et on considère un ensemble de fonctions FoCal, accompagné de leur type. **self** est remplacé par son type de définition. Cela signifie qu'il faut extraire les fonctions de leur collection d'origine, les mettre dans un environnement de fonction FMON, et modifier les appels de fonctions en un appel de fonction de cet environnement. Nous effectuons aussi une normalisation du filtrage par motif afin d'obtenir un filtrage simple (*i.e.* sur des motifs constants avec un constructeur et des variables, autrement appelé *filtrage de tête*).

7.1.1 Syntaxe des expressions FMON

Un programme FMON est la donnée d'un ensemble de fonctions. Nous commençons donc par définir les symboles de fonctions de FMON.

Définition 7.1.1 Symboles de fonctions**.**
On note F *l'ensemble des symboles de fonctions de FMON. Un élément de cet ensemble est noté f.*

La syntaxe des expressions FMON est donnée dans la figure 7.1. Cette syntaxe est proche de celle des expressions FoCal. De plus, les arguments des appels de fonctions, les décisions des conditionnelles et les valeurs filtrées ne peuvent être que des variables. Le filtrage ne se fait que sur les constructeurs de tête.

Les constructions de filtrage portent uniquement sur les constructeurs de tête de la variable filtrée. De plus, les motifs attrape-tout n'apparaissent qu'en fin de filtrage.

7.1.2 Sémantique des expressions FMON

Pour définir la sémantique des fonctions FMON il faut d'abord donner la définition des environnements qui permettent d'évaluer une expression.

Définition 7.1.2 Fonctions normalisées**.**
On définit l'environnement de fonction FMON normalisé de la manière suivante :

$$
\begin{array}{lcll}
e & ::= & \textbf{let}\ x = e_1\ \textbf{in}\ e_2 & \text{déclaration de variable locale} \\
& | & \textbf{if}\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \text{conditionnelle} \\
& | & \mathsf{f}(x_1, \ldots, x_i) & \text{appel de fonction d'arité } i \\
& | & \sharp f(x_1, \ldots, x_i) & \text{appel de fonction prédéfinie} \\
& & \quad \textbf{match}\ x\ \textbf{with} & \\
& & \quad | \ pat_1 \rightarrow e_1 & \\
& | & \qquad \vdots & \text{filtrage} \\
& & \quad | \ pat_i \rightarrow e_i & \\
& & \quad [| \ \_ \rightarrow e_{i+1}] & \\
& | & n & \text{valeur entière} \\
& | & C^0 & \text{constructeur constant} \\
& | & C^n(x_1, \ldots, x_n) & \text{constructeur appliqué à des variables} \\
& | & x & \text{variable}
\end{array}
$$

$$
\begin{array}{lcll}
pat & ::= & C^0 & \text{constructeur constant} \\
& | & C^n(pat\_arg, \ldots, pat\_arg) & \text{constructeur d'arité } n
\end{array}
$$

$$
\begin{array}{lcll}
pat\_arg & ::= & \_ & \text{motif attrape-tout} \\
& | & x & \text{variable de motif}
\end{array}
$$

$$
\begin{array}{lcll}
v & ::= & n & \text{valeur entière} \\
& | & C^0 & \text{constructeur constant} \\
& | & C^n(v_1, \ldots, v_n) & \text{constructeur appliqué}
\end{array}
$$

FIGURE 7.1 – Syntaxe de FMON

$$\begin{array}{rcl} \mathcal{E}_f & ::= & \emptyset \\ & | & \mathcal{E}_f, (f \leftarrow < x_1, \ldots, x_n \rightsquigarrow e >) \end{array}$$

Cet environnement lie un symbole de fonction à une fermeture FMON. Dans la suite, $\mathcal{E}_f$ est l'environnement de fonction définies dans le programme.

On remarque que dans la dernière définition, comme en FoCal, les fermetures de fonctions ne contiennent pas d'environnement. Dans un programme FMON, toutes les fonctions sont au même niveau, cela signifie que chaque fonction a la possibilité d'appeler toutes les autres fonctions et en particulier elles sont toutes implicitement mutuellement récursives. Dans ce cas, et comme il n'y a pas de traits d'ordre supérieur, il n'est pas nécessaire d'enfermer un environnement local dans les fermetures

Définition 7.1.3 Environnement de variables**.**
Nous reprenons les environnements de variable $\mathcal{E}_v$ définis dans 6.7.2. Dans la suite, le terme environnement désigne un environnement de variable.

Nous définissons la sémantique des expressions normalisées à l'aide de jugements d'évaluation de la forme suivante :

$$\mathcal{E}_v; \mathcal{E}_\mathsf{f} \vdash_N e \rhd v$$

Un tel jugement se lit « l'expression e s'évalue en la valeur v dans l'environnement $\mathcal{E}_v$ et dans l'environnement de fonction $\mathcal{E}_\mathsf{f}$ ».

La figure 7.2 présente la sémantique des expressions FMON. Cette sémantique est similaire à celle des expressions FoCal, nous ne les détaillons pas. Les seules différences sont celles qui viennent de la syntaxe restrictive de FMON. Ainsi, là où des règles d'évaluations de FoCal doivent évaluer des expressions, les règles de FMON correspondantes attendent des variables qui doivent être définies dans l'environnement de variable.

Nous énonçons ci-dessous un certain nombre de propriétés de la sémantique qui seront nécessaires par la suite pour montrer la correction et la complétude de la transformation des programmes FoCal en FMON.

Les deux premiers théorèmes sont classiques, il s'agit d'abord de l'affaiblissement qui énonce que si une expression s'évalue en une valeur dans un certain environnement alors elle s'évalue en la même valeur dans l'environnement auquel on a ajouté une définition. Le deuxième est le renforcement qui affirme qu'on peut sous certaines conditions supprimer une définition dans l'environnement.

Théorème 7.1.4. *Soient $\mathcal{E}_v$ un environnement et x' une variable non libre dans e. Soit encore, un environnement de fonction $\mathcal{E}_f$, une expression e et deux valeurs v et v'.*

$$\textit{Si } (\mathcal{E}_v; \mathcal{E}_f \vdash_N e \rhd v) \textit{ alors } (\mathcal{E}_v \oplus (x' \leftarrow v'); \mathcal{E}_f \vdash_N e \rhd v)$$

Démonstration. La preuve se fait par induction sur l'arbre de dérivation de $\mathcal{E}_v; \mathcal{E}_\mathsf{f} \vdash_N e \rhd v$.

□

$$\frac{\mathcal{E}_v(x) = v}{\mathcal{E}_v; \mathcal{E}_\mathsf{f} \vdash_N x \rhd v}\ \textsc{NVar}$$

$$\frac{}{\mathcal{E}_v; \mathcal{E}_\mathsf{f} \vdash_N i \rhd i}\ \textsc{NEntier}$$

$$\frac{}{\mathcal{E}_v; \mathcal{E}_\mathsf{f} \vdash_N C^0 \rhd C^0}\ \textsc{NConstante}$$

$$\frac{\mathcal{E}_v(x_1) = v_1 \qquad \ldots \qquad \mathcal{E}_v(x_n) = v_n}{\mathcal{E}_v; \mathcal{E}_\mathsf{f} \vdash_N C(x_1, \ldots, x_n) \rhd C(v_1, \ldots, v_n)}\ \textsc{NConstructeur}$$

$$\frac{\begin{array}{c}\mathcal{E}_v(x) = \texttt{true} \\ \mathcal{E}_v; \mathcal{E}_\mathsf{f} \vdash_N e_1 \rhd v_1\end{array}}{\mathcal{E}_v; \mathcal{E}_\mathsf{f} \vdash_N \textbf{ if } x \textbf{ then } e_1 \textbf{ else } e_2 \rhd v_1}\ \textsc{NIf\_true}$$

$$\frac{\begin{array}{c}\mathcal{E}_v(x) = \texttt{false} \\ \mathcal{E}_v; \mathcal{E}_\mathsf{f} \vdash_N e_2 \rhd v_2\end{array}}{\mathcal{E}_v; \mathcal{E}_\mathsf{f} \vdash_N \textbf{ if } x \textbf{ then } e_1 \textbf{ else } e_2 \rhd v_2}\ \textsc{NIf\_false}$$

$$\frac{\begin{array}{c}\mathcal{E}_v; \mathcal{E}_\mathsf{f} \vdash_N e_1 \rhd v_1 \\ \mathcal{E}_v \oplus (x \leftarrow v_1); \mathcal{E}_\mathsf{f} \vdash_N e_2 \rhd v_2\end{array}}{\mathcal{E}_v; \mathcal{E}_\mathsf{f} \vdash_N \textbf{ let } x = e_1 \textbf{ in } e_2 \rhd v_2}\ \textsc{NLet}$$

$$\frac{\begin{array}{c}\mathcal{E}_v(x_1) = v_1 \qquad \ldots \qquad \mathcal{E}_v(x_n) = v_n \\ \mathcal{E}_\mathsf{f}(\mathsf{f}) = <x_1, \ldots, x_n \rightsquigarrow e_f> \\ (x_1, v_1), \ldots, (x_n, v_n); \mathcal{E}_\mathsf{f} \vdash_N e_f \rhd v_f\end{array}}{\mathcal{E}_v; \mathcal{E}_\mathsf{f} \vdash_N \mathsf{f}(x_1, \ldots, x_n) \rhd v_f}\ \textsc{NApply}$$

$$\frac{\begin{array}{c}\mathcal{E}_v(x) = v \qquad \mathcal{F}(pat_i, v) = \mathcal{E}_v{}' \\ \forall j < i,\ \mathcal{F}(pat_j, v) = \texttt{fail} \qquad \mathcal{E}_v, \mathcal{E}_v{}'; \mathcal{E}_\mathsf{f} \vdash_N e_i \rhd v_i\end{array}}{\begin{array}{l}\mathcal{E}_v; \mathcal{E}_\mathsf{f} \vdash_N \textbf{match } x \textbf{ with} \\ \qquad \mid pat_1 \rightarrow e_1 \\ \qquad \quad \vdots \qquad\qquad \rhd v_i \\ \qquad \mid pat_n \rightarrow e_n\end{array}}\ \textsc{NFiltrage}$$

$$\frac{\begin{array}{c}\mathcal{E}_v(x_1) = v_1 \qquad \ldots \qquad \mathcal{E}_v(x_n) = v_n \\ \text{eval\_op}(\sharp f^n, v_1, \ldots, v_n) = v\end{array}}{\mathcal{E}_v; \mathcal{E}_\mathsf{f} \vdash_N \sharp f^n(x_1, \ldots, x_n) \rhd v}\ \textsc{NOp}$$

FIGURE 7.2 – Sémantique des expressions FMON

Théorème 7.1.5. *Soient $\mathcal{E}_v$ un environnement et x' une variable non libre dans e. Soit encore, un environnement de fonction $\mathcal{E}_f$, une expression e et deux valeurs v et v'.*

$$Si\ (\mathcal{E}_v \oplus (x' \leftarrow v'); \mathcal{E}_f \vdash_N e \rhd v)\ alors\ (\mathcal{E}_v; \mathcal{E}_f \vdash_N e \rhd v)$$

Démonstration. La preuve se fait par induction sur e. □

Les deux théorèmes suivants sont similaires aux deux précédents, mais concernent les environnements de fonction.

Théorème 7.1.6. *Soient $\mathcal{E}_f$ et $\mathcal{E}_f'$ deux environnements de fonction tels que $\mathcal{E}_f \subseteq \mathcal{E}_f'$, un environnement $\mathcal{E}_v$, une expression e et une valeur v.*

$$Si\ (\mathcal{E}_v; \mathcal{E}_f \vdash_N e \rhd v)\ alors\ (\mathcal{E}_v; \mathcal{E}_f' \vdash_N e \rhd v)$$

Démonstration. La preuve se fait par induction sur l'arbre de dérivation de $\mathcal{E}_v; \mathcal{E}_f \vdash_N e \rhd v$. □

Théorème 7.1.7. *Soient un environnement $\mathcal{E}_v$, un environnement de fonction $\mathcal{E}_f$, une expression e, une variable x et deux valeurs v et v'. La propriété suivante est vérifiée :*

$$Si\ (\mathcal{E}_v, (x \leftarrow v); \mathcal{E}_f \vdash_N e \rhd v')\ alors\ (\mathcal{E}_v; \mathcal{E}_f \vdash_N e[x \leftarrow v] \rhd v')$$

La notation $e[x \leftarrow v]$ désigne l'expression obtenue en remplaçant toutes les occurrences libres de x dans e par v.

Démonstration. Preuve par induction sur la dérivation. □

Le dernier théorème permet de déterminer l'ensemble minimum des variables qui doivent être définies dans un environnement si une expression s'évalue en une valeur.

Théorème 7.1.8. *Soient un environnement de variable $\mathcal{E}_v$, un environnement de fonction $\mathcal{E}_f$, une expression e et une valeur v.*

$$Si\ \mathcal{E}_v, \mathcal{E}_f \vdash_N e \rhd v\ alors\ \mathcal{E}_v\ définit\ au\ moins\ l'ensemble\ des\ variables\ libres\ de\ e.$$

Démonstration. Se montre par induction sur e. □

7.2 Normalisation des programmes FoCal

Nous présentons dans cette section la normalisation d'un programme FoCal appelée *mise en forme normale monadique*. Le principe de cette normalisation est de nommer tous les calculs au sein du programme. Nous avons besoin de nommer les calculs apparaissant dans le programme initial car, plus tard, nous traduirons ces programmes en contraintes. En effet, dans un système de contraintes, on pose des contraintes qui expriment des liens entre une variable et le résultat d'un calcul. Par exemple, la contrainte qui impose que la somme de deux entiers doit être égale à la somme de deux autres entiers n'est pas exprimée dans un système de contraintes par : $X+Y = Z+T$. Elle est exprimée en utilisant une cinquième variable devant être égale aux deux sommes : A = X + Y, A = Z + T. Utiliser une forme intermédiaire monadique est appropriée dans le sens où dès qu'on a nommé tous les calculs, la traduction du programme en contraintes se fait plus naturellement.

Dans la suite, nous supposons qu'on cherche à traduire un ensemble de fonctions d'un programme FoCal vers FMON. Bien entendu, il doit être possible d'évaluer une application de chacune des fonctions FMON obtenues. Pour cela, il faut prendre le soin de traduire toutes les fonctions FoCal dépendantes des fonctions à traduire.

7.2.1 Normalisation des expressions FoCal

Nous avons choisi de présenter la normalisation des expressions FoCal en deux parties. En effet, normaliser le filtrage demande d'effectuer du calcul matriciel. Ce calcul matriciel est utilisé pour isoler les portions des motifs du programme FoCal et effectuer les traitements spécifiques pour arriver à une forme monadique. Ce calcul matriciel n'est pas nécessaire pour faire la normalisation des autres constructions du langage. Nous avons donc choisi de définir une fonction dédiée à la normalisation du filtrage.

Nous présentons d'abord, la fonction qui normalise le filtrage par motif. Puis, nous exposons la fonction qui transforme une expression FoCal en une expression FMON.

Normalisation du filtrage par motif

Le but de la normalisation du filtrage est de désambiguïser les motifs présents, c'est-à-dire, faire en sorte qu'une valeur ne puisse être capturée que par un seul des motifs du filtrage. Ainsi, l'ordre d'apparition des motifs ne modifie pas le comportement du programme.

Exemple 7.2.1. Soit le programme suivant :

```
match x with
| Nil → 1
| Cons(e,Cons(e',l)) → 2
| Cons(e,l) → 3
```

`x` est une liste d'entiers, le programme retourne 1 si `x` est la liste vide, 2 si la liste est de longueur au moins 2 et 3 si la liste est de longueur 1. On remarque que le filtrage n'est pas exhaustif. On souhaite normaliser ce filtrage vers les filtrages suivants :

```
match x with
| Nil → 1
| Cons(e,ll) → match ll with
                | Nil → 3
                | Cons(e',l) → 2
```

On remarque au travers de l'exemple précédent que la normalisation du filtrage par motif implique de traduire un filtrage en un ensemble de filtrages imbriqués. Nous ne donnons aucune restriction aux filtrages de FoCal. En particulier, les motifs peuvent être de n'importe quelle forme et ne sont pas nécessairement exhaustifs. Pour ce dernier point, nous rappelons que notre but dans ce chapitre est de trouver des valeurs qui satisfont une propriété (précondition). La non exhaustivité du filtrage et en particulier trouver une valeur qui n'est pas filtrée par un filtrage n'entre pas dans notre propos. Ceci implique que détecter (ou imposer) qu'un filtrage soit non exhaustif, bien que les valeurs qui mettent en défaut le filtrage sont la plupart du temps révélatrices d'une erreur de programmation, ne nous intéressent pas.

La normalisation du filtrage que nous présentons ici est une adaptation de l'approche proposée par Luc Maranget dans [Mar92]. L'adaptation porte sur le fait que Maranget accepte le filtrage de plusieurs expressions simultanément (alors que nos filtrages n'en concernent qu'une

seule) et que le langage étudié par Maranget suit une évaluation paresseuse (alors que celle de FoCal réalise des appels pas valeur).

Dans notre normalisation, nous avons un langage qui ne permet que de filtrer une seule valeur dans un langage à appel par valeur. Nous avons adapté la fonction de Maranget pour qu'elle n'échoue pas.

Définition 7.2.2 Normalisation du filtrage**.**
$\mathcal{N}_f$ est la fonction de normalisation du filtrage par motif. Sa définition est donnée en figure 7.3. Elle prend trois arguments : une liste de variables filtrées, une matrice de motifs et une matrice colonne d'expressions.

Nous remarquons que la définition que nous avons donnée pour la fonction $\mathcal{N}_f$ est proche de celle de Maranget. Nous l'avons modifiée pour qu'elle n'échoue pas dans les cas où le filtrage n'est pas exhaustif. Pour Maranget, les valeurs non prévues par le filtrage pour une variable sont problématiques. Un programme paresseux qui contient des filtrages non exhaustifs est non déterministe et la normalisation induit l'ordre de filtrage des variables et rend le programme déterministe. De plus, nous avons gardé dans notre fonction de traduction la notion de filtrage simultané de plusieurs variables car le développement de la normalisation de motifs non triviaux (par exemple : $C_1(C_2(\ldots), Y)$) demande de faire un traitement pour $C_2(\ldots)$ et un traitement pour Y. Intuitivement, la matrice des motifs représente ce filtrage simultané, chaque ligne de la matrice étant une ligne du filtrage.

Le premier argument de $\mathcal{N}_f$ est la liste des n variables qui sont filtrées simultanément.

Le deuxième argument est une matrice P de motifs à appliquer à chaque variable. Chaque ligne contient, dans l'ordre des variables, les motifs à appliquer. La matrice P doit donc contenir autant de colonnes que de variables. Le troisième argument est une matrice colonne E d'expressions e_i à évaluer lorsque dans une ligne de P l'ensemble des filtres correspondent aux valeurs des variables. E contient donc autant de lignes que P.

La figure 7.3 présente la définition de $\mathcal{N}_f$ avec une analyse par cas :

- la première définition correspond au cas où tous les motifs de la première ligne sont des variables. Dans ce cas, le calcul se réduit en l'expression e_1 dans laquelle les variables des motifs sont remplacées par les variables filtrées ;

- la deuxième définition est le cas où la première colonne de motifs ne contient que des variables. Dans ce cas, il devient inutile de filtrer la première variable. On effectue un appel récursif dans lequel la première variable est retirée de la liste des variables filtrées, la première colonne de P est supprimée et dans chaque expression on substitue les occurrences libres de la variable de motifs par la variable supprimée ;

- le troisième et dernier cas est appliqué lorsque les deux précédents ne peuvent pas l'être. Dans ce cas, on transforme l'expression en un filtrage qui contient les constructeurs de tête apparaissant dans les pat_1^i. Les appels récursifs sont déterminés à partir des règles résumées dans les deux tableaux de la figure.

Le théorème de correction qui suit indique que pour un ensemble de variables et un environnement qui définit leur valeur, si on a une matrice de motifs telle qu'une de ses lignes filtre l'ensemble des variables et si l'expression associée à cette ligne s'évalue en une valeur v alors la

$$\mathcal{N}_f([x_1;\ldots;x_n], \begin{pmatrix} y_1 & y_2 & \ldots & y_n \\ pat_1^2 & pat_2^2 & \ldots & pat_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ pat_1^m & pat_2^m & \ldots & pat_n^m \end{pmatrix}, \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{pmatrix}) = e_1[y_1 \leftarrow x_1]\ldots[y_n \leftarrow x_n]$$

$$\mathcal{N}_f([x_1;\ldots;x_n], \begin{pmatrix} y^1 & pat_2^1 & \ldots & pat_n^1 \\ y^2 & pat_2^2 & \ldots & pat_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ y^m & pat_2^m & \ldots & pat_n^m \end{pmatrix}, \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{pmatrix}) = \mathcal{N}_f([x_2;\ldots;x_n], \begin{pmatrix} pat_2^1 & \ldots & pat_n^1 \\ pat_2^2 & \ldots & pat_n^2 \\ \vdots & \ddots & \vdots \\ pat_2^m & \ldots & pat_n^m \end{pmatrix}, \begin{pmatrix} e_1[y^1 \leftarrow x_1] \\ e_2[y^2 \leftarrow x_2] \\ \vdots \\ e_m[y^m \leftarrow x_1] \end{pmatrix})$$

$$\mathcal{N}_f([x_1;\ldots;x_n], P, E) = \begin{array}{l} \textbf{match } x_1 \textbf{ with} \\ \mid C_1^{k_1}(y_1^1,\ldots,y_{k_1}^1) \rightarrow \mathcal{N}_f([y_1^1;\ldots;y_{k_1}^1;x_2;\ldots;x_n], P_1, E_1) \\ \mid C_2^{k_2}(y_1^2,\ldots,y_{k_2}^2) \rightarrow \mathcal{N}_f([y_1^1;\ldots;y_{k_2}^1;x_2;\ldots;x_n], P_2, E_2) \\ \qquad\vdots \qquad\qquad\qquad \vdots \qquad\qquad\qquad\qquad \vdots \\ \mid C_j^{k_j}(y_1^j,\ldots,y_{k_j}^j) \rightarrow \mathcal{N}_f([y_1^1;\ldots;y_{k_j}^1;x_2;\ldots;x_n], P_j, E_j) \\ \mid \_ \qquad\qquad\qquad\quad \rightarrow \mathcal{N}_f([x_2;\ldots;x_n], P_d, E_d) \end{array}$$

La dernière ligne du filtrage n'apparait pas si la première colonne de P ne contient pas de variables.

Pour $P = \begin{pmatrix} pat_1^1 & \ldots & pat_n^1 \\ \vdots & \ddots & \vdots \\ pat_1^m & \ldots & pat_n^m \end{pmatrix}$ $E = \begin{pmatrix} e^1 \\ \vdots \\ e^m \end{pmatrix}$

y_i^j variables fraîches

C_i^j sont les constructeurs du type de la variable filtrée

Les P_o et E_o sont obtenues en transformant chaque ligne des matrices P et E selon le principe suivant :

Première colonne de la i^e ligne de P	Ligne de P_o	Ligne de E_o
y	$\_ \ \ldots \ \_ \ pat_n^2 \ \ldots \ pat_n^i$	$e^i[y \leftarrow x_1]$
$\_$	$\_ \ \ldots \ \_ \ pat_n^2 \ \ldots \ pat_n^i$	e^i
$C_o^{k_o}(q_1^i,\ldots,q_{k_o}^i)$	$q_1^i \ \ldots \ q_{k_o}^i \ pat_n^2 \ \ldots \ pat_n^i$	e^i
$C_{o'}^{k_{o'}}(\ldots)(o' \neq p)$	la ligne est supprimée	

Les deux matrices P_d et E_d sont obtenues en transformant chaque ligne de la matrice P et E selon le principe suivant :

Première colonne de la i^e ligne de P	Ligne de P_d	Ligne de E_d
y	$pat_n^2 \ \ldots \ pat_n^i$	$e^i[y \leftarrow x_1]$
$\_$	$pat_n^2 \ \ldots \ pat_n^i$	e^i
$C_{o'}^{k_{o'}}(\ldots)$	la ligne est supprimée	

FIGURE 7.3 – Traduction du filtrage par motif

fonction $\mathcal{N}_f$ retourne une expression qui s'évalue en cette même valeur.

Théorème 7.2.3. *Soient $n+1$ environnements de variable $\mathcal{E}_v$, $\mathcal{E}_{v1}$ ..., $\mathcal{E}_{vn}$, un environnement de fonction $\mathcal{E}_f$. Soient n variables x_1, ..., x_n. Soient $n+1$ valeurs v, v_1, ..., v_n. Soient n expressions e_1, ... e_n. Et enfin, soit $m * n$ motifs notés pat_i^j.*

Sous les hypothèses suivantes :

$$\mathcal{E}_v(x_1) = v_1 \ldots \mathcal{E}_v(x_n) = v_n \tag{7.1}$$
$$\forall j \in [\![1, k-1]\!],\ \exists i \in [\![1, n]\!],\ \mathcal{F}(v_i, pat_i^j) = \texttt{fail} \tag{7.2}$$
$$\forall i \in [\![1, n]\!],\ \mathcal{F}(v_i, pat_i^k) = \mathcal{E}_{vi} \tag{7.3}$$
$$\mathcal{E}_v \otimes (\mathcal{E}_{v1} \otimes \ldots \otimes \mathcal{E}_{vn}); \mathcal{E}_f \vdash_N e_k \rhd v \tag{7.4}$$

Le séquent suivant est vérifié :

$$\mathcal{E}_v; \mathcal{E}_f \vdash_N \mathcal{N}_f([x_1, \ldots, x_n], \begin{pmatrix} pat_1^1 & \ldots & pat_n^1 \\ \vdots & \ddots & \vdots \\ pat_1^m & \ldots & pat_n^m \end{pmatrix}, \begin{pmatrix} e_1 \\ \vdots \\ e_n \end{pmatrix}) \rhd v$$

Démonstration. Par induction sur la définition de la fonction $\mathcal{N}_f$.

□

Le théorème suivant est le cas particulier où une seule variable est filtrée. Intuitivement, il s'agit de l'utilisation qu'on fait de la fonction $\mathcal{N}_f$ pour normaliser les programmes FoCal.

Théorème 7.2.4. *Sous les hypothèses :*

$$\mathcal{E}_v(x) = v$$
$$\forall i \in [\![1, k-1]\!],\ \mathcal{F}(v, pat_i) = \texttt{fail}$$
$$\mathcal{F}(v, pat_k) = \mathcal{E}_v{}'$$
$$\mathcal{E}_v \otimes \mathcal{E}_v{}'; \mathcal{E}_f \vdash_N e_k \rhd v'$$

La proposition suivante est vraie :

$$\mathcal{E}_v; \mathcal{E}_f \vdash_N \mathcal{N}_f([x], \begin{pmatrix} pat_1 \\ \vdots \\ pat_n \end{pmatrix}, \begin{pmatrix} e_1 \\ \vdots \\ e_n \end{pmatrix}) \rhd v'$$

Démonstration. Par 7.2.3. □

Pour le théorème suivant on commence par étendre localement la syntaxe du filtrage de FoCal en autorisant à filtrer simultanément plusieurs variables. Cette extension de syntaxe n'est utile que pour faciliter la preuve de correction de la traduction. Elle n'est pas utilisée dans la traduction.

$$\begin{array}{l} \textbf{match } x_1, \ldots, x_n \textbf{ with} \\ \mid pat_1^1,\ \ldots,\ pat_n^1 \ \rightarrow e_1 \\ \quad \vdots \quad \ddots \quad \vdots \quad \vdots \ \vdots \\ \mid pat_1^m, \ldots,\ pat_n^m \rightarrow e_m \end{array}$$

La règle d'évaluation de cette forme de filtrage est alors la suivante :

$$\frac{\begin{array}{c}\mathcal{E}_v(x_i) \rhd v_i \\ \forall i \in [\![1,n]\!],\ \mathcal{F}(pat_i^k, v) = \mathcal{E}_{v_i}' \\ \forall j < k,\ \exists l,\ \mathcal{F}(pat_l^j, v_l) = \texttt{fail} \\ \mathcal{E}_v \otimes (\mathcal{E}_{v1}' \otimes \ldots \otimes \mathcal{E}_{vn}'); \mathcal{C} \vdash e_k \rhd v_k\end{array}}{\mathcal{E}_v; \mathcal{C} \vdash \begin{array}{l}\textbf{match } x_1, \ldots, x_n \textbf{ with} \\ \begin{array}{|lllll} pat_1^1, & \ldots, & pat_n^1 & \to & e_1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ pat_1^m, & \ldots, & pat_n^m & \to & e_m \end{array}\end{array} \rhd v_k} \text{ FFiltrage\_n}$$

La ligne de filtrage activée est la première ligne où les variables s'unifient avec leur filtre respectif. Cette règle reste cohérente avec la version du filtrage de FoCal car lorsqu'une seule variable est filtrée, elle est identique à la règle FFiltrage.

Théorème 7.2.5. *S'il existe un environnement de collection $\mathcal{C}$ et un environnement de fonction $\mathcal{E}_f$ tels que les hypothèses suivantes sont vérifiées :*

$$\begin{array}{l}\mathcal{E}_v; \mathcal{C} \vdash e_1 \rhd v_1 \\ \mathcal{E}_v; \mathcal{E}_f \vdash_N e_1 \rhd v_1 \\ \quad\vdots \\ \mathcal{E}_v; \mathcal{C} \vdash e_m \rhd v_m \\ \mathcal{E}_v; \mathcal{E}_f \vdash_N e_m \rhd v_m \\ \exists i,\ \mathcal{E}_v; \mathcal{E}_f \vdash_N \mathcal{N}_f \begin{pmatrix}\textbf{match } x_1, \ldots, x_n \textbf{ with} \\ \begin{array}{|lllll} pat_1^1, & \ldots, & pat_n^1 & \to & e_1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ pat_1^m, & \ldots, & pat_n^m & \to & e_m \end{array}\end{pmatrix} \rhd v_i\end{array}$$

Alors on a l'affirmation suivante :

$$\mathcal{E}_v; \mathcal{C} \vdash \begin{array}{l}\textbf{match } x_1, \ldots, x_n \textbf{ with} \\ \begin{array}{|lllll} pat_1^1, & \ldots, & pat_n^1 & \to & e_1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ pat_1^m, & \ldots, & pat_n^m & \to & e_m \end{array}\end{array} \rhd v_i$$

Démonstration. Ce théorème se prouve par induction sur $\mathcal{N}_f$. □

Le dernier théorème est un corollaire du précédent. Il s'agit du cas où il n'y a qu'une seule variable filtrée :

Théorème 7.2.6. *Sous les hypothèses suivantes :*

$$\begin{array}{l}\mathcal{E}_v; \mathcal{C} \vdash e_1 \rhd v_1 \\ \mathcal{E}_v; \mathcal{E}_f \vdash_N e_1 \rhd v_1 \\ \quad\vdots \\ \mathcal{E}_v; \mathcal{C} \vdash e_n \rhd v_n \\ \mathcal{E}_v; \mathcal{E}_f \vdash_N e_n \rhd v_n \\ \exists i,\ \mathcal{E}_v; \mathcal{E}_f \vdash_N \mathcal{N}_f \begin{pmatrix}\textbf{match } x \textbf{ with} \\ \begin{array}{|lll} pat_1 & \to & e_1 \\ \vdots & \ldots & \vdots \\ pat_n & \to & e_m \end{array}\end{pmatrix} \rhd v_i\end{array}$$

Alors on a l'affirmation suivante :

$$\mathcal{E}_v; \mathcal{C} \vdash \begin{array}{l}\textbf{match } x \textbf{ with} \\ \mid pat_1 \rightarrow e_1 \\ \quad \vdots \quad \ldots \quad \vdots \\ \mid pat_n \rightarrow e_m\end{array} \triangleright v_i$$

Démonstration. Par 7.2.5. □

Normalisation des autres constructions

Afin de traduire les expressions FoCal en FMON, nous devons donner une fonction qui permet de faire le lien entre les symboles de fonctions FoCal et les fonctions FMON.

Définition 7.2.7 Traduction des symboles de fonctions**.**
On se donne une fonction $\mathcal{N}_{\mathfrak{c}!f}$ injective qui prend en entrée une collection et un symbole de fonction FoCal. Cette fonction retourne un identificateur de fonction de F de FMON.

Nous nous donnons aussi un ensemble de variables fraîches nécessaire pour notre traduction.

Définition 7.2.8 Variables fraîches**.**
Soit $\mathcal{F}reshF$ un ensemble de variables fraîches. Les variables de l'ensemble $\mathcal{F}reshF$ sont distinctes des noms des variables qui apparaissent dans une expression FoCal.

Définition 7.2.9 Traduction d'une expression FoCal**.**
$\mathcal{N}_e(e)$ traduit l'expression FoCal e en une expression en forme intermédiaire monadique. Sa définition est donnée dans la figure 7.4.

La fonction $\mathcal{N}_e$ est définie sur la structure de l'expression à mettre en forme normale. Pour la plupart des constructions du langage, elle ajoute des définitions locales. Pour le filtrage par motif, elle utilise la fonction $\mathcal{N}_f$ définie en 7.2.1. On remarque qu'on pourrait optimiser la traduction de manière à ne pas introduire de variable locale lorsque le calcul intermédiaire est déjà une variable, par exemple dans l'appel de fonction $f(x, 1 + 3)$, x n'a pas besoin d'être introduit par un nouveau **let**.

On remarque que la normalisation d'une expression FoCal est syntaxique. Ainsi, un appel de fonction $\mathfrak{c}!f$ FoCal, est traduit en un appel à la fonction $\mathsf{f} = \mathcal{N}_{\mathfrak{c}!f}(\mathfrak{c}, f)$ de FMON. Afin d'assurer que l'expression traduite puisse être évaluée, il faut aussi, dans ce cas, traduire la fonction $\mathfrak{c}!f$.

Pour mettre en forme normale une fonction, il s'agit de retrouver la fermeture de la fonction FoCal qui se trouve dans la collection et de créer la fermeture FMON correspondante.

Définition 7.2.10 Normalisation d'une fonction**.**
Soit $\mathcal{C}$ un environnement de collection FoCal. On définit la mise en forme intermédiaire monadique d'une fonction FoCal $\mathfrak{c}!f$ par la fonction $\mathcal{N}_f$. $\mathcal{N}_f$ prend en argument l'environnement de

$$\mathcal{N}_e(\textbf{let } x = e_1 \textbf{ in } e_2) = \textbf{let } x = \mathcal{N}_e(e_1) \textbf{ in } \mathcal{N}_e(e_2)$$

$$\mathcal{N}_e(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) = \begin{vmatrix} \textbf{let } x = \mathcal{N}_e(e_1) \textbf{ in} \\ \textbf{if } x \textbf{ then } \mathcal{N}_e(e_2) \textbf{ else } \mathcal{N}_e(e_3) \end{vmatrix} \quad x \in \mathcal{F}reshF$$

$$\mathcal{N}_e(\mathfrak{c}!f(e_1, \ldots, e_n)) = \begin{vmatrix} \textbf{let } x_1 = \mathcal{N}_e(e_1) \textbf{ in} \\ \vdots \\ \textbf{let } x_n = \mathcal{N}_e(e_n) \textbf{ in} \\ \mathcal{N}_{\mathfrak{c}!f}(\mathfrak{c}, f)(x_1, \ldots, x_n) \end{vmatrix} \quad \forall i \in [\![1, n]\!],\ x_i \in \mathcal{F}reshF$$

$$\mathcal{N}_e(\sharp f(e_1, \ldots, e_n)) = \begin{vmatrix} \textbf{let } x_1 = \mathcal{N}_e(e_1) \textbf{ in} \\ \vdots \\ \textbf{let } x_n = \mathcal{N}_e(e_n) \textbf{ in} \\ \sharp f(x_1, \ldots, x_n) \end{vmatrix} \quad \forall i \in [\![1, n]\!],\ x_i \in \mathcal{F}reshF$$

$$\mathcal{N}_e \begin{pmatrix} \textbf{match } e \textbf{ with} \\ \mid pat_1 \to e_1 \\ \vdots \\ \mid pat_i \to e_i \end{pmatrix} = \begin{array}{l} \textbf{let } x = \mathcal{N}_e(e) \textbf{ in} \\ \mathcal{N}_f([x], \begin{pmatrix} pat_1 \\ \vdots \\ pat_i \end{pmatrix}, \begin{pmatrix} \mathcal{N}_e(e_1) \\ \vdots \\ \mathcal{N}_e(e_i) \end{pmatrix}) \end{array} \quad x \in \mathcal{F}reshF$$

$$\mathcal{N}_e(C) = C$$

$$\mathcal{N}_e(C(e_1, \ldots, e_n)) = \begin{vmatrix} \textbf{let } x_1 = \mathcal{N}_e(e_1) \textbf{ in} \\ \vdots \\ \textbf{let } x_n = \mathcal{N}_e(e_n) \textbf{ in} \\ C(x_1, \ldots, x_n) \end{vmatrix} \quad \forall i \in [\![1, n]\!],\ x_i \in \mathcal{F}reshF$$

$$\mathcal{N}_e(i) = i$$

$$\mathcal{N}_e(x) = x$$

FIGURE 7.4 – Traduction d'une expression FoCal en une expression FMON

collection $\mathcal{C}$ et le nom de la fonction FoCal à traduire $\mathfrak{c}!f^i$. Elle retourne la définition normalisée de f.

$$\mathcal{N}_f(\mathcal{C}, \mathfrak{c}!f) = <x_1, \ldots, x_n \rightsquigarrow \mathcal{N}_e(e)> \quad \begin{array}{ll} si & \mathcal{C}(\mathfrak{c}) = val_{\mathfrak{c}} : C^{\mathfrak{c}} \\ et & val_{\mathfrak{c}}(f) = <x_1, \ldots, x_n \rightsquigarrow e> \end{array}$$

7.2.2 Environnement minimal d'une expression

Afin d'obtenir un programme complet, nous devons maintenant définir la traduction des environnements qui permettent d'assurer que l'évaluation aboutit. Il s'agit de l'ensemble des fonctions utilisées par l'expression.

Pour cela, nous définissons la notion de dépendance d'une expression FoCal. Intuitivement, une fonction f dépend d'une fonction g si lors de l'évaluation d'une application de f la fonction g peut être appelée. Les dépendances d'une fonction sont calculées à partir de sa définition.

Définition 7.2.11 Dépendances d'une expression**.**
Soit e une expression FoCal. La fonction Dep *est telle que* Dep(e) *retourne l'ensemble des fonctions FoCal dont e dépend. Elle est définie sur la structure de e et a pour définition :*

$$\mathrm{Dep}(\textbf{let}\ x = e_1\ \textbf{in}\ e_2) = \mathrm{Dep}(e_1) \cup \mathrm{Dep}(e_2)$$

$$\mathrm{Dep}(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) = \mathrm{Dep}(e_1) \cup \mathrm{Dep}(e_2) \cup \mathrm{Dep}(e_3)$$

$$\mathrm{Dep}(\mathfrak{c}!f(e_1, \ldots, e_n)) = \{\mathfrak{c}!f\} \cup \bigcup_{i \in [\![1,n]\!]} \mathrm{Dep}(e_i)$$

$$\mathrm{Dep}\begin{pmatrix} \textbf{match}\ e\ \textbf{with} \\ |pat_1 \rightarrow e_1 \\ \vdots \\ |pat_n \rightarrow e_n \end{pmatrix} = \mathrm{Dep}(e) \cup \bigcup_{i \in [\![1,n]\!]} \mathrm{Dep}(e_i)$$

$$\mathrm{Dep}(i) = \mathrm{Dep}(x) = \mathrm{Dep}(C^0) = \emptyset$$

$$\mathrm{Dep}(C(e_1, \ldots, e_n)) = \bigcup_{i \in [\![1,n]\!]} \mathrm{Dep}(e_i)$$

Définition 7.2.12 Dépendance d'une fonction**.**
On étend la définition de la fonction Dep *à une fonction FoCal par rapport à un environnement de collection $\mathcal{C}$.*

$$\mathrm{Dep}(\mathfrak{c}!f^i, \mathcal{C}) = \mathrm{Dep}(e) \quad \begin{array}{ll} si & \mathcal{C}(\mathfrak{c}) = val_{\mathfrak{c}} : I^{\mathfrak{c}} \\ et & val_{\mathfrak{c}}(f^i) = <x_1, \ldots, x_i \rightsquigarrow e> \end{array}$$

La fonction Dep donne l'ensemble des fonctions qui doivent être définies dans l'environnement pour qu'un appel de la fonction $\mathfrak{c}!f$ puisse être évalué en une valeur. Chacune des fonctions retournées dépend elle aussi de fonctions qui doivent être définies. C'est pour cela qu'on définit l'ensemble des dépendances totales d'une fonction dans un environnement de collection.

Définition 7.2.13 Dépendances totales d'une expression**.**

Soit un environnement de collection $\mathcal{C}$ et une expression e. On définit la fonction Dep^+ *qui retourne l'ensemble des fonctions dont e dépend, comme la fermeture transitive de* Dep.

Définition 7.2.14 Environnement minimal**.**
Soit un environnement de collection $\mathcal{C}$. L'environnement minimal d'une expression FoCal e est noté $\mathcal{E}_f^*(e, \mathcal{C})$. Il est tel que

$$\mathcal{E}_f^*(e, \mathcal{C}) = \{\mathcal{N}_{\mathfrak{c}!f}(\mathfrak{c}, f) \leftarrow \mathcal{N}_f(\mathcal{C}, \mathfrak{c}!f) \mid \mathfrak{c}!f \in \mathrm{Dep}^+(e)\}$$

Lorsqu'il n'y a pas de confusion possible, nous omettons le deuxième argument de $\mathcal{E}_f^*$ et notons $\mathcal{E}_f^*(e)$ au lieu de $\mathcal{E}_f^*(e, \mathcal{C})$.

La notion de dépendance définie ici est à comparer avec celle définie par Prevosto pour FoCal dans [Pre03]. Dans FoCal, il existe deux sortes de dépendance, les *def-dépendances* et les *decl-dépendances*.

Les *def-dependances* sont définies dans FoCal pour les propriétés. Intuitivement, une propriété *def-dépend* d'une fonction FoCal signifie que la définition de la fonction doit être connue par la propriété. Les *def-dépendances* d'une propriété sont induites par la preuve de celle-ci. Si la preuve de P utilise la définition d'une fonction m, alors la propriété P *def-dépend* de m.

Les *decl-dépendances* se rapprochent plus de notre notion de dépendance. Une fonction m_1 *def-dépend* d'une fonction m_2 si m_1 utilise la signature de m_2. En d'autres termes, une m_1 *def-dépend* de m_2 si m_1 utilise la déclaration de m_2.
Par exemple, une propriété P *decl-dépend* d'une fonction f si l'énoncé de P contient un appel à f. Dans ce cas, P a besoin de connaître la signature de f.
Une fonction f *decl-dépend* d'une fonction g, si la définition de f contient un référence à g. De plus, la notion de *decl-dépendance* est transitive. Si une fonction f *decl-dépend* de g et si g *decl-dépend* à son tour d'une fonction h, alors f *decl-dépend* aussi de h.

On peut une définition plus précises des dépendances et leur utilité dans [Pre03].

7.2.3 Correction et complétude de la normalisation

Dans cette section on montre la correction et la complétude de la normalisation des programmes FoCal. Le premier théorème indique que si une expression FoCal s'évalue en une valeur alors sa forme normalisée s'évalue dans FoCal en la même valeur. Ce théorème est utile pour montrer la complétude de la normalisation.

Théorème 7.2.15. *Si on a*

$$\mathcal{E}_v; \mathcal{C} \vdash e \rhd v$$

alors on a

$$\mathcal{E}_v; \mathcal{C} \vdash \mathcal{N}_e(e) \rhd v$$

Démonstration. Se prouve par induction sur $\mathcal{N}_e$. □

Le théorème qui suit exprime que si une expression FoCal s'évalue en une valeur alors le programme FMON qui est sa traduction s'évalue en la même valeur.

Théorème 7.2.16 (Correction de la transformation)**.** *Soient un environnement de variable $\mathcal{E}_v$, un environnement de collection $\mathcal{C}$, une expression FoCal e et une valeur v.*

$$\textit{Si } \mathcal{E}_v;\mathcal{C} \vdash e \rhd v \textit{ alors } \mathcal{E}_v;\mathcal{E}_f^*(e) \vdash_N \mathcal{N}_e(e) \rhd v$$

Démonstration. La preuve se fait par induction sur l'arbre de dérivation de $\mathcal{E}_v, \mathcal{C} \vdash e \rhd v$.

Cas FLet, FIf_false et FIf_true

Ces cas sont immédiats avec les hypothèses d'induction et les théorèmes 7.1.6 et 7.1.4.

Cas FF_n_coll ($e = \mathfrak{c}!m(e_1, \ldots e_n)$)

$$\frac{\begin{array}{c}\mathcal{E}_v;\mathcal{C} \vdash e_1 \rhd v_1 \qquad \ldots \qquad \mathcal{E}_v;\mathcal{C} \vdash e_n \rhd v_n \\ \mathcal{C}(\mathfrak{c}) = val_{\mathfrak{c}} \qquad val_{\mathfrak{c}}(m) = <x_1 \ldots x_n \rightsquigarrow e_f> \\ (x_1, v_1), \ldots, (x_n, v_n); \mathfrak{c}; \mathcal{C} \vdash e_f \rhd v_f\end{array}}{\mathcal{E}_v;\mathcal{C} \vdash \mathfrak{c}!m(e_1, \ldots, e_n) \rhd v_f}\ \text{FF\_n\_coll}$$

Nous avons les hypothèses d'induction suivantes :

$$\begin{array}{c}\mathcal{E}_v;\mathcal{E}_{\mathsf{f}}^*(e_i) \vdash_N \mathcal{N}_e(e_i) \rhd v_i \qquad\qquad \forall i \in [\![1, n]\!] \\ (x_1, v_1), \ldots, (x_n, v_n);\mathcal{E}_{\mathsf{f}}^*(e_f) \vdash_N \mathcal{N}_e(e_f) \rhd v_f\end{array}$$

Par définition de Dep^+, on a $\mathcal{N}_f(\mathfrak{c}, m) \in \text{Dep}^+(e)$. Par la définition(7.2.14) de $\mathcal{E}_{\mathsf{f}}^*$, on a $\mathcal{E}_{\mathsf{f}}^*(e)(m) = <x_1, \ldots, x_n \rightsquigarrow \mathcal{N}_e(e_f)>$.

La transformation de e en forme normalisée donne l'expression

$$\textbf{let } x_1 = \mathcal{N}_e(e_1) \textbf{ in } \ldots \textbf{let } x_n = \mathcal{N}_e(e_n) \textbf{ in } \mathcal{N}_f(\mathfrak{c}, m)(x_1, \ldots, x_n)$$

En utilisant les théorèmes 7.1.4 et 7.1.6, on obtient les propositions suivantes (car les x_i sont des variables fraîches) :

$$\begin{array}{lll}\mathcal{E}_v & ;\mathcal{E}_{\mathsf{f}}^*(e) \vdash_N \mathcal{N}_e(e_1) \rhd v_1 \\ \mathcal{E}_v, (x_1 \leftarrow v_1) & ;\mathcal{E}_{\mathsf{f}}^*(e) \vdash_N \mathcal{N}_e(e_2) \rhd v_2 \\ \vdots & \vdots \qquad \vdots \\ \mathcal{E}_v, (x_1 \leftarrow v_1), \ldots, (x_{j-1} \leftarrow v_{j-1}) & ;\mathcal{E}_{\mathsf{f}}^*(e) \vdash_N \mathcal{N}_e(e_j) \rhd v_f \\ (x_1, v_1), \ldots, (x_n, v_n);\mathcal{E}_{\mathsf{f}}^*(e) \vdash_N \mathcal{N}_e(e_f) \rhd v_f & \end{array}$$

qui sont les hypothèses nécessaires pour évaluer $\mathcal{N}_e(e)$ en v_f. L'arbre de preuve final est donc :

$$\frac{\begin{array}{c}\mathcal{E}_{vn}(x_1) = v_1 \qquad \ldots \qquad \mathcal{E}_{vn}(x_n) = v_n \\ \mathcal{E}_{\mathsf{f}}^*(e)(f) = <x_1, \ldots, x_n \rightsquigarrow \mathcal{N}_e(e_f)> \\ (x_1, v_1), \ldots, (x_n, v_n);\mathcal{E}_{\mathsf{f}}^*(e) \vdash_N \mathcal{N}_e(e_f) \rhd v_f\end{array}}{\mathcal{E}_{vn};\mathcal{E}_{\mathsf{f}}^*(e) \vdash_N \mathcal{N}_f(\mathfrak{c}, m)(x_1, \ldots, x_n) \rhd v_f}\ \text{NApply} \tag{7.5}$$

$$\dfrac{\mathcal{E}_v;\mathcal{E}_{\mathsf{f}}^*(e) \vdash_N e_1 \rhd v_1 \qquad \dfrac{\mathcal{E}_{v1};\mathcal{E}_{\mathsf{f}}^*(e) \vdash_N e_2 \rhd v_2 \qquad \dfrac{\mathcal{E}_{vn-1};\mathcal{E}_{\mathsf{f}}^*(e) \vdash_N e_n \rhd v_n}{\text{(arbre 7.5)}} \quad \ddots}{\mathcal{E}_{v1};\mathcal{E}_{\mathsf{f}}^*(e) \vdash_N e_2' \rhd v_f}}{\mathcal{E}_v;\mathcal{E}_{\mathsf{f}}^*(e) \vdash_N \textbf{let } x_1 = e_1 \textbf{ in } \ldots \textbf{let } x_n = e_n \textbf{ in } \mathcal{N}_f(\mathfrak{c}, m)(x_1, \ldots, x_n) \rhd v_f}$$

avec

$$e'_i = \textbf{let } x_i = e_i \textbf{ in } \dots \textbf{let } x_n = e_n \textbf{ in } \mathcal{N}_f(\mathfrak{c}, m)(x_1, \dots x_n)$$
$$\mathcal{E}_{vi} = \mathcal{E}_v, (x_1 \leftarrow v_1), \dots, (x_i \leftarrow v_i)$$

Cas FConstructeur et FOperateur

Ces cas sont similaires à l'appel de fonction.

Cas FFiltrage $\left(e = \begin{array}{l} \textbf{match } e' \textbf{ with} \\ \mid pat_1 \rightarrow e_1 \\ \quad \vdots \\ \mid pat_n \rightarrow e_n \end{array} \right)$

Dans ce cas,

$$\mathcal{N}_e(e) = \left| \begin{array}{l} \textbf{let } x = \mathcal{N}_e(e') \textbf{ in} \\ \mathcal{N}_f([x], \begin{pmatrix} pat_1 \\ \vdots \\ pat_i \end{pmatrix}, \begin{pmatrix} \mathcal{N}_e(e_1) \\ \vdots \\ \mathcal{N}_e(e_i) \end{pmatrix}) \end{array} \right.$$

On a, par induction, les hypothèses suivantes :

$$\mathcal{E}_v; \mathcal{E}^*_{\mathsf{f}}(e') \vdash_N \mathcal{N}_e(e') \rhd v'$$
$$\mathcal{E}_v, \mathcal{E}_v'; \mathcal{E}^*_{\mathsf{f}}(e_i) \vdash_N \mathcal{N}_e(e_i) \rhd v_i$$

De plus, il existe un entier $i \in [\![1, n]\!]$ tel que

$$\forall j < i,\ \mathcal{F}(pat_j, v') = \texttt{fail}$$
$$\mathcal{F}(pat_i, v') = \mathcal{E}_v{}'$$

En utilisant les théorèmes 7.1.4 et 7.1.6 on obtient les propositions suivantes (car e' est une sous expression de e/e_i et x est une variable fraîche) :

$$\mathcal{E}_v; \mathcal{E}^*_{\mathsf{f}}(e) \quad \vdash_N \quad \mathcal{N}_e(e') \rhd v' \tag{7.6}$$
$$\mathcal{E}_v, \mathcal{E}_v', (x \leftarrow v); \mathcal{E}^*_{\mathsf{f}}(e) \quad \vdash_N \quad \mathcal{N}_e(e_i) \rhd v_i \tag{7.7}$$

Par le théorème 7.2.4 on en déduit :

$$\mathcal{E}_v, (x \leftarrow v'); \mathcal{E}^*_{\mathsf{f}}(e) \vdash_N \mathcal{N}_f([x], \begin{pmatrix} pat_1 \\ \vdots \\ pat_n \end{pmatrix}, \begin{pmatrix} e_1 \\ \vdots \\ e_n \end{pmatrix}) \rhd v_i$$

On obtient l'arbre d'évaluation suivant grâce à 7.6 et 7.7 :

$$\frac{\begin{array}{c} \mathcal{E}_v; \mathcal{E}^*_{\mathsf{f}}(e) \vdash_N \mathcal{N}_e(e') \rhd v' \\ \mathcal{E}_v, (x \leftarrow v'); \mathcal{E}^*_{\mathsf{f}}(e) \vdash_N \mathcal{N}_f([x], \begin{pmatrix} pat_1 \\ \vdots \\ pat_n \end{pmatrix}, \begin{pmatrix} e_1 \\ \vdots \\ e_n \end{pmatrix}) \rhd v_i \end{array}}{\mathcal{E}_v; \mathcal{E}_{\mathsf{f}} \vdash_N \textbf{let } x = \mathcal{N}_e(e') \textbf{ in } \mathcal{N}_f([x], \begin{pmatrix} pat_1 \\ \vdots \\ pat_n \end{pmatrix}, \begin{pmatrix} e_1 \\ \vdots \\ e_n \end{pmatrix}) \rhd v_i} \text{ NLet}$$

Cas FConstante et FEntier et FVar

La propriété est immédiatement vérifiée car ces expressions ne sont pas transformées et les règles de sémantique des deux langagges, toujours pour ces expressions, sont identiques.

□

Le théorème suivant exprime la complétude de la transformation. À savoir, si une expression FMON e_1 est la traduction d'une expression FoCal e_2, alors toute évaluation de e_2 en une valeur correspond à une évaluation de e_1 en cette même valeur.

Ce théorème est important car il assure que l'évaluation d'une expression FMON donne les mêmes valeurs que l'expression FoCal correspondante. Pour comprendre cela, il faut voir la contraposée de ce théorème. La contraposée indique que si une expression FoCal ne s'évalue pas en une valeur v, alors l'expression FMON correspondante ne s'évalue pas en cette valeur. Vu sous cet angle, ce théorème complète le précédent en exprimant le fait que les valeurs obtenues par les expressions FMON sont *ni plus ni moins* les mêmes valeurs que celles de FoCal.

Théorème 7.2.17 (Complétude de la transformation)**.** *Soit un environnement de variable $\mathcal{E}_v$, un environnement de collection $\mathcal{C}$, une expression FoCal e et une valeur v.*

$$\textit{Si } \mathcal{E}_v; \mathcal{E}_f^*(e) \vdash_N \mathcal{N}_e(e) \rhd v \textit{ alors } \mathcal{E}_v; \mathcal{C} \vdash e \rhd v$$

Démonstration. La preuve se fait par induction sur l'arbre de dérivation de $\mathcal{E}_v, \mathcal{E}_\mathsf{f}^*(e) \vdash_N \mathcal{N}_e(e) \rhd v$.

Cas NLet ($\mathcal{N}_e(e) = \textbf{let}\, x = e_1\, \textbf{in}\, e_2$)

Pour obtenir ce cas, il y a plusieurs possibilités. Soit le $\textbf{let}\, x = e_1$ de $\mathcal{N}_e(e)$ vient directement de e (on a $e = \textbf{let}\, x = e_1\, \textbf{in}\, \ldots$) soit il s'agit d'un **let** ajouté par la fonction de normalisation. Nous détaillons la preuve pour chacune des formes que e peut prendre pour obtenir un **let** dans la normalisation.

- $e = \textbf{let}\, x = e_3\, \textbf{in}\, e_4$
  On a alors $e_1 = \mathcal{N}_e(e_3)$ et $e_2 = \mathcal{N}_e(e_4)$. Ce cas se montre directement par les hypothèses d'induction.
- $e = \textbf{if}\, e_1\, \textbf{then}\, e_2\, \textbf{else}\, e_3$
  On a alors $\mathcal{N}_e(e) = \textbf{let}\, x = e_1\, \textbf{in}\, \textbf{if}\, x\, \textbf{then}\, e_2\, \textbf{else}\, e_3$ avec x frais.
  L'évaluation de $\mathcal{N}_e(e)$ donne les hypothèses suivantes :

$$\mathcal{E}_v; \mathcal{E}_\mathsf{f}^*(e) \vdash_N e_1 \rhd v_1 \tag{7.8}$$
$$\mathcal{E}_v \oplus (x \leftarrow v_1); \mathcal{E}_\mathsf{f}^*(e) \vdash_N \textbf{if}\, x\, \textbf{then}\, e_1\, \textbf{else}\, e_2 \rhd v_2 \tag{7.9}$$

Selon la valeur d'évaluation de e_1 on a $v_1 = \texttt{true}$ ou $v_1 = \texttt{false}$. On détaille le cas $v_1 = \texttt{true}$, l'autre cas est similaire.
Pour $v_1 = \texttt{true}$, on obtient l'hypothèse supplémentaire :

$$\mathcal{E}_v \oplus (x \leftarrow v_1); \mathcal{E}_\mathsf{f}^*(e) \vdash_N e_2 \rhd v_2$$

En utilisant les hypothèses d'induction on obtient les hypothèses suivantes :

$$\mathcal{E}_v; \mathcal{C} \vdash e_1 \rhd \texttt{true} \tag{7.10}$$
$$\mathcal{E}_v \oplus (x \leftarrow v_1); \mathcal{C} \vdash e_2 \rhd v_2 \tag{7.11}$$

Comme x est une variable fraîche, on peut utiliser le théorème de renforcement sur 7.11 pour supprimer dans l'environnement la valuation de x. De l'hypothèse obtenue conjointement avec 7.10. On arrive à conclure :

$$\frac{\mathcal{E}_v; \mathcal{C} \vdash e_1 \rhd \texttt{true} \qquad \mathcal{E}_v; \mathcal{C} \vdash e_2 \rhd v_2}{\mathcal{E}_v; \mathcal{C} \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \rhd v_2} \text{ FIF\_TRUE}$$

– $e = \begin{array}{l} \textbf{match } e' \textbf{ with} \\ \mid pat_1 \rightarrow e_1 \\ \quad \vdots \\ \mid pat_n \rightarrow e_n \end{array}$

Ce cas se montre en utilisant le théorème 7.2.6 sur les hypothèses d'induction, le théorème de renforcement et le théorème 7.2.15.

– $e = \mathfrak{c}!f(e_1, \ldots, e_n)$

On a alors $\mathcal{N}_e(e) = \begin{array}{l} \textbf{let } y_1 = e_1 \textbf{ in} \\ \quad \vdots \\ \textbf{let } y_n = e_n \textbf{ in} \\ \mathsf{f}(y_1, \ldots, y_n) \end{array}$ avec $y_1, \ldots, y_n$ frais et $\mathsf{f} = \mathcal{N}_f(\mathfrak{c}, f)$.

Les hypothèses sont alors les suivantes :

$$\mathcal{E}_v; \mathcal{E}^*_{\mathsf{f}}(e) \vdash_N e_1 \rhd v_1 \tag{7.12}$$
$$\mathcal{E}_v \oplus (y_2 \leftarrow v_2); \mathcal{E}^*_{\mathsf{f}}(e) \vdash_N e_1 \rhd v_1 \tag{7.13}$$

$$\mathcal{E}_v \oplus (y_1 \leftarrow v_1) \oplus \ldots \oplus (y_n \leftarrow v_n); \mathcal{E}^*_{\mathsf{f}}(e) \vdash_N \mathsf{f}(y_1, \ldots, y_n) \rhd v_f \tag{7.14}$$
$$\mathcal{E}^*_{\mathsf{f}}(e)(f) = <x_1, \ldots, x_n \rightsquigarrow e_f> \tag{7.15}$$
$$(x_1, v_1), \ldots, (x_n, v_n); \mathcal{E}^*_{\mathsf{f}}(e) \vdash_N \mathcal{N}_e(e_f) \rhd v_f \tag{7.16}$$

On obtient par hypothèses d'induction :

$$\mathcal{E}_v; \mathcal{C} \vdash e_1 \rhd v_1 \tag{7.17}$$
$$\mathcal{E}_v \oplus (y_1 \leftarrow v_1); \mathcal{C} \vdash e_2 \rhd v_2 \tag{7.18}$$

$$\mathcal{E}_v \oplus (y_1 \leftarrow v_1) \oplus \ldots \oplus (y_n \leftarrow v_n); \mathcal{E}^*_{\mathsf{f}}(e) \vdash \mathsf{f}(y_1, \ldots, y_n) \rhd v_{n+1} \tag{7.19}$$
$$(x_1, v_1), \ldots, (x_n, v_n); \mathcal{C} \vdash e_f \rhd v_f \tag{7.20}$$

Comme $y_1, \ldots, y_n$ sont des variables fraîches, on peut appliquer les théorèmes de renforcement sur 7.12, …, 7.13. De plus, on utilise la définition de $\mathcal{E}^*_{\mathsf{f}}(e)$ pour conclure :

$$\frac{\begin{array}{c} \mathcal{C} \vdash e_i \rhd v_i \qquad \forall i \in [\![1, n]\!] \\ \mathcal{C}(\mathfrak{c}) = val_{\mathfrak{c}} \qquad val_{\mathfrak{c}}(m) = <x_1 \ldots x_n \rightsquigarrow e_f> \\ (x_1, v_1), \ldots, (x_n, v_n); \mathcal{C} \vdash e_f \rhd v_f \end{array}}{\mathcal{E}_v; \mathcal{C} \vdash e \rhd v_f} \text{ FF\_N\_COLL}$$

– $e = \sharp f(e_1, \ldots, e_n)$ et $e = C(e_1, \ldots, e_n)$

Ces cas sont similaires à l'appel de fonction.

Cas NIF_TRUE, NIF_FALSE, NAPPLY, NOPERATEUR, NFILTRAGE et NCONSTRUCTEUR

Ces cas ne sont pas possibles car pour chacun d'eux la fonction $\mathcal{N}_e$ ne retourne pas d'expression de cette forme là.

$$
\begin{array}{rcll}
\sigma & ::= & c & \text{contrainte simple} \\
& | & c, \sigma & \text{conjonction de contraintes} \\
& | & \sigma_1 \wedge \sigma_2 & \text{conjonction de store} \\
\\
c & ::= & X =_{fd} a & \text{égalité entière} \\
& | & X \neq_{fd} a & \text{inégalité entière} \\
& | & X =_h t & \text{égalité de valeurs concrètes} \\
& | & X \neq_h t & \text{inégalité de valeurs concrètes} \\
& | & \texttt{f}(X_1, \ldots, X_n) & \text{appel à une clause} \\
& | & \begin{array}{l}\texttt{match}(X, [\ \texttt{pattern}(pat, \sigma), \\ \qquad \ldots, \\ \qquad \texttt{pattern}(pat, \sigma)], \sigma)\end{array} & \text{contrainte de filtrage} \\
& | & \texttt{ite}(X, \sigma, \sigma) & \text{contrainte conditionnelle} \\
\\
pat & ::= & \texttt{c}(X_1, \ldots, X_n) & \text{motif} \\
\\
a & ::= & i & \text{valeur} \\
& | & X & \text{variable entière} \\
& | & op(X_1, \ldots, X_n) & \text{opérateur arithmétique} \\
\\
t & ::= & X & \text{variable de valeur concrete} \\
& | & \texttt{c}(X_1, \ldots, X_n) & \text{terme construit}
\end{array}
$$

FIGURE 7.5 – Syntaxe des contraintes

Cas NVAR, NENTIER et NCONSTANTE

Ces cas sont immédiats car aucun calcul n'est nécessaire.

□

7.3 Langage de contraintes

Nous présentons ici le langage de contraintes que nous allons cibler. Ce langage se veut minimaliste et suffisant pour permettre d'exprimer les préconditions des propriétés FoCal. Dans la suite nous présentons un prédicat qui permet de vérifier si une affectation, c'est-à-dire, une valuation des variables d'un système de contraintes, est une solution.

7.3.1 Syntaxe

La syntaxe du langage de contraintes est donnée dans la figure 7.5. Dans le chapitre 4 nous avons présenté un système de contraintes comme un triplet $\sigma = (X, D, C)$ de variables, domaine des variables et ensemble de contraintes. Pour ne pas alourdir le propos, nous considérons dans la suite un système de contraintes σ comme l'unique donnée de ses contraintes. Ainsi, un système de contraintes est représenté par σ appelé *store de contraintes*. Il s'agit d'une conjonction, potentiellement vide, de contraintes élémentaires ou d'autres stores de contraintes.

Les contraintes élémentaires c du langage de contraintes sont les suivantes :

- la contrainte d'égalité sur les entiers $X =_{fd} a$ impose que la valeur de la variable X et la valeur de l'expression droite de l'égalité soient identiques. Les expressions possibles comme élément à droite de l'égalité sont : i, une valeur entière ; X une variable de contrainte entière ; $op(X_1, \ldots, X_n)$ l'application d'un opérateur arithmétique à des variables ;
- la contrainte d'inégalité sur les entiers est similaire à la contrainte précédente à ceci près qu'elle impose que les valeurs des deux côtés de $\neq_{fd}$ soient différentes ;
- la contrainte d'égalité sur les valeurs concrètes $X =_h t$ a la même signification que la contrainte d'égalité sur les entiers. La partie droite de l'égalité est une valeur concrète (un terme) et peut être : X une variable de valeur concrète ou $\texttt{c}(X_1, \ldots, X_n)$ la contrainte qui impose le constructeur de tête variable concrète ;
- la contrainte d'inégalité sur les valeurs concrètes impose que les deux arguments aient des valeurs différentes ;
- la contrainte $\texttt{f}(X_1, \ldots, X_n)$ correspond à un appel de fonction de contraintes. Il s'agit d'une méta-contrainte qui va se déplier en un ensemble de contraintes sur les variables $X_1, \ldots, X_n$;
- les contraintes $\texttt{match}(X, \ldots)$ et $\texttt{ite}(X, \sigma, \sigma)$ sont des méta-contraintes explicitées plus bas.

Exemple 7.3.1. Par exemple, le système de contraintes suivant impose que la variable R soit de la forme $\texttt{c}(2 * X, \texttt{cons}(X, nil))$ où X est une valeur entière.

$$R = \texttt{c}(X_1, R_1), R_1 = \texttt{cons}(X_2, R_2), R_2 = \texttt{nil}, X_1 = 2 * X_2$$

Définition 7.3.2 Conjonction de stores**.**
Soient deux stores de contraintes σ_1 et σ_2. La conjonction de ces deux stores est notée $\sigma_1 \wedge \sigma_2$. $\wedge$ est un combinateur dont voici la définition :

$$\begin{array}{rcl} c \wedge \sigma & = & c, \sigma \\ (c, \sigma_1) \wedge \sigma_2 & = & c, (\sigma_1 \wedge \sigma_2) \end{array}$$

7.3.2 Domaine des variables

Chaque variable du système est dotée d'un domaine qui est l'ensemble des valeurs qu'elle peut prendre. Par exemple, si X est une liste d'entiers, le domaine de x est :

$$\{\texttt{nil}, \texttt{cons}(I_1, \texttt{nil}), \texttt{cons}(I_2, \texttt{cons}(I_3, \texttt{nil})), \ldots\}$$

Les I_i sont des variables dotées du domaine des entiers.

Avant de définir le domaine des variables, nous allons d'abord donner la définition des types manipulés par le système. Ces types seront ensuite utilisés pour définir le domaine des variables de contraintes.

Types et contraintes

Nous commençons par définir les environnements de définition de types dans le système de contraintes. Ces environnements permettent de définir des types concrets. Ils seront, *in fine*, définis à partir de l'environnement de définition de types du programme FoCal considéré.

La figure 7.6 montre la syntaxe des environnements de type de notre langage de contraintes. Un environnement de définition de types est la donnée d'un ensemble de noms de type (τ)

$$
\begin{array}{lcll}
\theta & ::= & \emptyset & \text{environnement vide} \\
 & | & \theta, (\tau, [\delta_t, \ldots, \delta_t]) & \text{ajout d'un type} \\
 & & & \\
\delta_t & ::= & \delta_c & \text{constructeur} \\
 & | & \delta_c, \delta_t & \text{constructeurs} \\
 & & & \\
\delta_c & ::= & \mathtt{c} & \text{constante} \\
 & | & \mathtt{c}(carg_1, \ldots, carg_n) & \text{constructeur paramétré} \\
 & & & \\
carg & ::= & \alpha & \text{argument de type} \\
 & | & \tau & \text{type} \\
 & & & \\
\tau & ::= & \iota & \text{type} \\
 & | & \iota(\alpha_1, \ldots, \alpha_n) & \text{type paramétré}
\end{array}
$$

FIGURE 7.6 – Syntaxe des environnements de définition de types

associé à une définition sous la forme d'un ensemble de constructeurs. Chaque constructeur est spécifié par son nom et le type de ses arguments.

Exemple 7.3.3. Soit θ l'environnement :

$$(\mathtt{list}(\alpha), [\mathtt{nil}, \mathtt{cons}(\alpha, \mathtt{list}(\alpha))]), (\mathtt{option}(\alpha), [\mathtt{none}, \mathtt{some}(\alpha)])$$

θ définit deux types, le type des listes dont le paramètre α définit le type des éléments des listes, ainsi que le type option.

Définition 7.3.4 Variables libres d'un type**.**
Soit une définition de type, l'ensemble des variables de type libres de la définition est l'ensemble des variables de type qui apparaissent dans les constructeurs auquel on retire les variables de type paramètres du type.

Exemple 7.3.5. Soit la définition de type $t(\alpha_1, \alpha_2), [\mathtt{c}_1(\alpha_3), \mathtt{c}_2(\alpha_4, \alpha_1, \alpha_2)]$. Ce type prend deux arguments α_1 et α_2, il est constitué de deux constructeurs $\mathtt{c}_1$ et $\mathtt{c}_2$.

L'ensemble des variables libres de t est $\{\alpha_3, \alpha_4\}$.

Définition 7.3.6 Environnement bien formé**.**
Un environnement de définition de types est dit bien formé lorsque pour chaque définition de type, l'ensemble des variables de type libres est vide.

Dans la suite, on ne considère que les environnements de définition de types bien formés.

Définition 7.3.7 Constructeurs d'un type**.**
Soit τ un type et un environnement de définition de types θ, l'ensemble des constructeurs de τ est noté $\mathbb{C}(\tau, \theta)$.

Lorsqu'il n'y a pas d'ambiguïté, on omettra l'environnement de définition de types dans $\mathbb{C}(\tau, \theta)$ *et notera* $\mathbb{C}(\tau)$.

Exemple 7.3.8. Pour l'environnement de définition de types θ de l'exemple 7.3.3, l'ensemble des constructeurs du type des listes d'entiers est :

$$\mathbb{C}(\texttt{list(int)}) = \{\texttt{nil}, \texttt{cons}\}$$

Comme l'ensemble des constructeurs d'un type ne dépend pas de ses arguments, on s'autorisera par la suite à obtenir l'ensemble des constructeurs d'un type sans préciser les arguments. Ainsi, on a $\mathbb{C}(\texttt{list}) = \mathbb{C}(\texttt{list(int)})$.

Définition 7.3.9 Instanciation d'un constructeur**.**
Soit un type τ *et un constructeur* $\texttt{c} \in \mathbb{C}(\tau)$. *La fonction* $\mathcal{C}_{args}(\tau, \texttt{c})$ *retourne la liste des arguments de* c *où les variables de type ont été remplacées par leur définition.*

Exemple 7.3.10. Pour l'environnement de définition de types de l'exemple 7.3.3, on a :

$$\mathcal{C}_{args}(\texttt{list(int)}, \texttt{cons}) = [\texttt{int}, \texttt{list(int)}]$$
$$\mathcal{C}_{args}(\texttt{option(list(int))}, \texttt{some}) = [\texttt{list(int)}]$$

Environnement de types

Définition 7.3.11 Environnement de type**.**
Les environnements de type permettent d'associer un type à chaque variable d'un ensemble de variables. La définition est la suivante :

$$\begin{array}{rcl} \Gamma & ::= & \emptyset \\ & | & \Gamma, (X : \tau) \end{array}$$

Dans la suite on considérera implicitement que toutes les variables des contraintes sont typées par un environnement noté Γ.

Variables entières

Définition 7.3.12 Domaine d'une variable entière**.**
Soit X *une variable entière. Le domaine de* X *est défini par la fonction* DOM_i *telle que* $DOM_i(X) \subseteq \mathbb{N}$ *où* $\mathbb{N}$ *est l'ensemble des valeurs entières.*

Contraintes sur les entiers

Le langage de contraintes dispose de deux opérateurs de contrainte sur les entiers. La contrainte d'égalité $=_{fd}$ permet de contraindre une variable à avoir la même valeur que le résultat d'une opération arithmétique. La contrainte d'inégalité $\neq_{fd}$ contraint, quant à elle, une variable à avoir une valeur différente de l'expression à droite de $\neq_{fd}$.

Variables algébriques

On définit ici le domaine des variables algébriques. Il s'agit d'un domaine qui a été défini dans le cadre de notre travail de traduction des programmes FoCal en contraintes.

Définition 7.3.13 Domaine d'une variable concrète**.**
Soit une variable X et un type τ tel que $\Gamma(X) = \tau$. Le domaine de X est donné par la fonction DOM_h. On a $DOM_h(X) \subseteq \mathbb{C}(\tau)$.

Le domaine d'une variable algébrique est un ensemble de constructeurs. Intuitivement, pour une variable X et un constructeur $\mathtt{c}$ tel que $\mathtt{c} \in DOM_h(X)$, une valeur possible pour X est un terme $\mathtt{c}(\ldots)$ où les arguments de $\mathtt{c}$ sont des termes qui respectent les types attendus.

Exemple 7.3.14. Si B est une variable booléenne alors son domaine est

$$DOM_h(B) = \{\mathtt{true}, \mathtt{false}\}$$

Lorsque le domaine est un singleton, par exemple, $DOM_h(X) = \{\mathtt{c}\}$, on a une instanciation partielle implicite de la variable par l'unique constructeur du domaine. Ainsi la valeur de X est de la forme $\mathtt{c}(X_1, \ldots, X_n)$ où $X_1, \ldots, X_n$ sont des variables fraîches telles que $\mathcal{C}_{args}(\tau, \mathtt{c}) = [\tau_1, \ldots, \tau_n]$ et $\forall i \in [\![1, n]\!], DOM_h(X_i) = \mathbb{C}(\tau_i)$ pour les variables de type concret et $DOM_i(X_i) = \mathbb{N}$ pour les variables entières et l'environnement de type Γ est augmenté des variables typées $X_1 : \tau_1, \ldots, X_n : \tau_n$.

Ce domaine est à comparer avec les *features-trees* [AKPS92]. Les *features-trees* sont une généralisation du domaine de Herbrand (domaine des termes non interprétés) dans laquelle chaque constructeur de terme est vu comme un enregistrement, chaque argument est optionnel et porte un nom. Le domaine de Herbrand est donc le domaine spécial où chaque argument a pour nom $1, 2, \ldots$ et où les arguments sont obligatoires. Ainsi, les *features-trees* apportent une sorte de typage sur les termes. Elle est différente de la nôtre dans la mesure le typage où nous imposons le symbole de constructeur à être appliqué à des valeurs qui doivent elles aussi respecter un certain type.

Contraintes sur les types algébriques

Les types algébriques possèdent deux opérateurs de contrainte, l'égalité et l'inégalité.

Définition 7.3.15 Égalité**.**
On définit l'égalité entre variables de type concret. Soient X et Y deux variables de contraintes. L'opérateur d'égalité est noté $=_h$. La contrainte $X =_h Y$ impose que les variables X et Y aient la même valeur.

Définition 7.3.16 Inégalité**.**
Soient X et Y deux variables de contraintes de type concret. L'opérateur d'inégalité est noté $\neq_h$. La contrainte $X \neq_h Y$ impose que les variables X et Y n'aient pas la même valeur.

Opérateurs prédéfinis

Un certain nombre d'opérateurs sont prédéfinis. Afin de ne pas alourdir le discours, nous avons regroupé dans l'annexe A, l'ensemble des opérateurs du langage de contraintes.

Nous avons présenté dans cette section le type des variables. Dans la suite, l'environnement de types Γ n'est pas montré. On supposera qu'il est présent implicitement et que chaque ajout de nouvelles variables dans le système de contraintes, en particulier les variables ajoutées par

le dépliage d'une méta-contrainte, occasionne automatiquement un ajout de ces variables avec leurs types dans l'environnement de types. Nous obtenons le type des variables à partir du programme FoCal.

7.3.3 Clauses

Définition 7.3.17 Environnement de clause**.**
Les environnements de clause sont donnés par la syntaxe suivante :

$$\begin{array}{rcl} \mathcal{E}_{\texttt{cl}} & ::= & \emptyset \\ & | & \mathcal{E}_{\texttt{cl}}, (\texttt{f}, <X_1, \ldots, X_n \rightsquigarrow \sigma>) \end{array}$$

Ils permettent de lier un symbole de fonction à un environnement de typage et une abstraction. On appelle clause une définition $\texttt{f}, <X_1, \ldots, X_n \rightsquigarrow \sigma>$ *de l'environnement.*

Informellement, les clauses permettent de définir un équivalent aux fonctions des langages de programmation. La définition $(\texttt{f}, <X_1, \ldots, X_n \rightsquigarrow \sigma>)$ d'une clause `f` donne une contrainte sur les variables $X_1, \ldots, X_n$ à l'aide de l'ensemble de contraintes σ. Les variables σ autre que X_1, X_2, …, X_n sont implicitement quantifiées existentiellement. Cela signifie que la contrainte $\texttt{f}(X'_1, \ldots, X'_n)$ est vérifiée pour une affectation sur les variables $X'_1, \ldots, X'_n$ si et seulement s'il existe des valeurs des variables de σ (autre que X_1, …, X_n) telles que σ dans lequel on a remplacé les X_i par les X'_i.

Exemple 7.3.18. Pour définir une contrainte qui impose à une liste de contenir une unique valeur donnée (ici un entier), on peut donner la définition de clause suivante :

$$\begin{array}{c} \texttt{singleton}, <I, L \rightsquigarrow \sigma> \\ \text{avec} \\ \sigma = \{Y =_h \texttt{nil}, L =_h \texttt{cons}(I, Y)\} \end{array}$$

7.3.4 Test de solution d'un système de contraintes

Habituellement, pour tester si une affectation est une solution d'un système de contrainte on se contente de vérifier si l'ensemble des variables du système de contrainte est instancié et si toute les contraintes sont vérifiées.

En présence des clauses, nous ne pouvons pas faire de test aussi simple. Le dépliage de la définition d'une clause peut ajouter de nouvelles variables dans le système de contraintes. Ces variables sont quantifiées existentiellement et ne sont pas connues à l'avance. Il n'est pas possible que l'affectation qu'on cherche à vérifier les définissent. De plus, comme nous sommes en présence de fonctions récursives. Le nombre de ces variables n'est pas connu et n'est pas figé.

Pour résoudre ce problème, nous avons défini un prédicat qui permet de tester si une affectation est consistante par rapport à un système de contraintes. Ce prédicat de test de solution décide de la consistance d'une affectation par rapport à un système de contraintes. Nous ne considérons pas les contraintes d'inégalités. En effet ce prédicat est utilisé dans la suite pour montrer la correction de la traduction des programmes normalisés en contraintes. La traduction d'un programme normalisé en contraintes ne produisant pas d'inégalités, il n'est pas utile que ce prédicat décide de la consistance d'une affectation totale sur de telles contraintes.

Le prédicat a pour originalité de prendre en entrée une affectation et de donner en sortie une nouvelle affectation. Lorsque c'est le cas, cela signifie que l'affectation d'entrée est consistante. La

nouvelle affectation surcharge l'ancienne par ajout de nouvelles valeurs de variables trouvées par évaluation. Il agit ainsi comme un transformateur d'affectation. De plus, l'affectation de sortie a la particularité d'être totale. Ce prédicat est bien formé car il traite toutes les contraintes du systèmes les unes après les autres sans remettre en question les contraintes déjà traitées.

Le prédicat de test de consistance d'une affectation utilise un jugement de la forme :

$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}'$$

Un tel jugement signifie « l'affectation $\mathcal{A}$ est consistante par rapport au store de contraintes σ et retourne la nouvelle affectation $\mathcal{A}'$ ». Sa définition est donnée dans les figures 7.7 et 7.8.

Exemple 7.3.19. Soit l'environnement de clause suivant

$$\begin{array}{l}\mathcal{E}_{\mathtt{cl}} = (\mathsf{fact}, <R_1, N_1 \rightsquigarrow \sigma>) \\ \sigma = (C =_h (N_1 \leq 1),\ \mathtt{ite}(C, [N_1 =_{fd} 1], \\ \qquad\qquad\qquad\qquad [N_2 =_{fd} N_1 - 1, \mathsf{fact}(R_2, N_2), R_1 =_{fd} R_2 * N_1]))\end{array}$$

Soit l'affectation $\mathcal{A} = (S \leftarrow 2), (E \leftarrow 2)$. Le séquent $\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \mathsf{fact}(S, E) \longmapsto \mathcal{A}$ est valide.

À chaque dépliage de la définition de fact, il suffit que les variables E et S soient définies dans l'affectation de gauche pour que l'ensemble des variables qui apparaissent dans fact obtiennent une valeur de définition dans l'affectation de gauche. Ces valeurs sont oubliées après la vérification des contraintes de fact (règle CALL).

L'exemple précédent montre le cas le moins intuitif de $\vdash_S$. Nous avons une système de contrainte qui définit des clauses. Chaque clause peut utiliser des variables intermédiaires qui ne sont pas visible de l'extérieur. Comme nous sommes en présence de fonctions récursives, le nombre de ces variables n'est pas fixe. Il dépend du nombre d'appels récursif. Nous ne pouvons donc pas les rendre visibles en les ajoutant en tant que paramètres des clauses.

Par conséquent, il n'est pas possible, en général, de déterminer si une affectation totale est consistante sur un système de contraintes sans faire une phase d'énumération. Par exemple, on ne peut pas vérifier pour une couple de valeurs données de E_1 E_2 si la clause

$$<E_1, E_2 \rightsquigarrow E_1 =_{fd} X * 4352 + Y * 50, E+ =_{fd} X * 64523 + Y * 450>$$

est vérifiée tant qu'on a pas de valeur pour X et Y.

Ici, nous sommes dans le cas particulier où les systèmes de contraintes que nous considérons proviennent de la traduction d'un programme FMON. Les variables intermédiaires des clauses peuvent être entièrement déterminées à partir des valeurs des paramètres. Pour l'exemple, 7.3.19 nous avons utilisé la définition de fact qui est obtenue à partir de la définition FMON de la fonction factorielle. Plus exactement, $\vdash_S$ est capable de déterminer les valeurs des variables qui se trouve à gauche d'une égalité.

Ce qui justifie que $\vdash_S$ nous donne bien une affectation consistante est le fait qu'à chaque étape de vérification, le prédicat a vérifié qu'un sous ensemble des contraintes est satisfait et que pour vérifier une nouvelle contrainte, il ajoute dans l'affectation une valuation pour des variables qui ne sont pas encore déterminée (et par conséquent qui n'apparaissent pas dans les contraintes déjà satisfaites). Ainsi, il ne remet pas en cause la validité des contraintes déjà vérifiées et construit les conditions suffisantes pour que la contrainte en cours de traitement soit vérifiée.

$$\frac{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S c \longmapsto \mathcal{A}_1 \qquad \mathcal{A}_1;\mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}_2}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S c,\sigma \longmapsto \mathcal{A}_2}\ \textsc{Conjonct1}$$

$$\frac{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}_1 \qquad \mathcal{A}_1;\mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_2 \longmapsto \mathcal{A}_2}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \wedge \sigma_2 \longmapsto \mathcal{A}_2}\ \textsc{Conjonct2}$$

$$\frac{\mathcal{A}(X) = i}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{fd} i \longmapsto \mathcal{A}}\ \textsc{EqFdI}$$

$$\frac{X \notin \mathcal{A}}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{fd} i \longmapsto \mathcal{A} \oplus (X \leftarrow i)}\ \textsc{EqFdI}$$

$$\frac{\mathcal{A}(X) = \mathcal{A}(Y)}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{fd} Y \longmapsto \mathcal{A}}\ \textsc{EqFdX}$$

$$\frac{X \notin \mathcal{A} \qquad Y \in \mathcal{A}}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{fd} Y \longmapsto \mathcal{A} \oplus (X \leftarrow \mathcal{A}(Y))}\ \textsc{EqFdX}$$

$$\frac{\mathcal{A}(X) = \text{eval\_op}(op, \mathcal{A}(X_1), \ldots, \mathcal{A}(X_n))}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{fd} op(X_1, \ldots, X_n) \longmapsto \mathcal{A}}\ \textsc{EqFdOp}$$

$$\frac{X \notin \mathcal{A} \qquad v = \text{eval\_op}(op, \mathcal{A}(X_1), \ldots, \mathcal{A}(X_n))}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{fd} op(X_1, \ldots, X_n) \longmapsto \mathcal{A} \oplus (X, v)}\ \textsc{EqFdOp}$$

$$\frac{\mathcal{A}(X) = \mathtt{c}(\mathcal{A}(X_1), \ldots, \mathcal{A}(X_n))}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{h} \mathtt{c}(X_1, \ldots, X_n) \longmapsto \mathcal{A}}\ \textsc{EqHC}$$

$$\frac{X \notin \mathcal{A} \qquad v = \mathtt{c}(\mathcal{A}(X_1), \ldots, \mathcal{A}(X_n))}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{fd} op(X_1, \ldots, X_n) \longmapsto \mathcal{A} \oplus (X, v)}\ \textsc{EqHC}$$

$$\frac{\mathcal{A}(X) = \mathcal{A}(Y)}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{h} Y \longmapsto \mathcal{A}}\ \textsc{EqHX}$$

$$\frac{X \notin \mathcal{A} \qquad \mathcal{A}(Y) = \text{eval\_op}(op, \mathcal{A}(X_1), \ldots, \mathcal{A}(X_n))}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{fd} Y \longmapsto \mathcal{A} \oplus (X, \mathcal{A}(Y))}\ \textsc{EqHX}$$

FIGURE 7.7 – Prédicat de test de solution d'un système de contraintes

$$\frac{\mathcal{E}_{\mathtt{cl}}(\mathtt{f}) = <X'_1, \ldots, X'_n \rightsquigarrow \sigma> \quad (X'_1 \leftarrow \mathcal{A}(X_1)), \ldots, (X'_n \leftarrow \mathcal{A}(X_n)); \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}'}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \mathtt{f}(X_1, \ldots, X_n) \longmapsto \mathcal{A}} \textsc{Call}$$

$$\frac{\mathcal{A}(X) = \mathtt{true} \qquad \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}'}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \mathtt{ite}(X, \sigma_1, \sigma_2) \longmapsto \mathcal{A}'} \textsc{Ite\_true}$$

$$\frac{\mathcal{A}(X) = \mathtt{false} \qquad \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_2 \longmapsto \mathcal{A}'}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \mathtt{ite}(X, \sigma_1, \sigma_2) \longmapsto \mathcal{A}'} \textsc{Ite\_false}$$

$$\frac{\mathcal{A}(X) = \mathtt{c}_k(\mathcal{A}(X_1^k), \ldots, \mathcal{A}(X_{n_k}^k)) \qquad 1 \leq k \leq i \qquad \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_k \longmapsto \mathcal{A}'}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \begin{array}{l} \mathtt{match}(X, [\\ \quad \mathtt{pattern}(X =_h \mathtt{c}_1(X_1^1, \ldots, X_{n_1}^1), \sigma_1), \\ \qquad \vdots \\ \quad \mathtt{pattern}(X =_h \mathtt{c}_i(X_1^i, \ldots, X_{n_i}^i), \sigma_i) \ ], \\ \quad \sigma_{i+1}) \end{array} \longmapsto \mathcal{A}'} \textsc{Match\_pat}$$

$$\frac{\mathcal{A}(X) \neq \mathtt{c}_k(\mathcal{A}(X_1^k), \ldots, \mathcal{A}(X_{n_k}^k)) \qquad 1 \leq k \leq i \qquad \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_{i+1} \longmapsto \mathcal{A}'}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \begin{array}{l} \mathtt{match}(X, [\\ \quad \mathtt{pattern}(X =_h \mathtt{c}_1(X_1^1, \ldots, X_{n_1}^1), \sigma_1), \\ \qquad \vdots \\ \quad \mathtt{pattern}(X =_h \mathtt{c}_i(X_1^i, \ldots, X_{n_i}^i), \sigma_i) \ ], \\ \quad \sigma_{i+1}) \end{array} \longmapsto \mathcal{A}'} \textsc{Match\_other}$$

FIGURE 7.8 – Prédicat de test de solution d'un système de contraintes (suite et fin)

Théorème 7.3.20. *Soient deux affectations $\mathcal{A}$, $\mathcal{A}'$ et un store de contraintes σ. Si on a $\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}'$ alors $\mathcal{A}'$ est une affectation totale consistante par rapport à σ.*

Démonstration. Se montre par induction sur la hauteur de l'arbre de dérivation $\mathcal{A} \vdash_S \sigma \longmapsto \mathcal{A}'$. Au fur et mesure de la dérivation du jugement, le prédicat définit $\mathcal{A}'$ en ajoutant à $\mathcal{A}$ des affectations pour des variables qui n'y sont pas définies. □

Théorème 7.3.21. *Soient deux affectations $\mathcal{A}_1$ et $\mathcal{A}_2$ et un store de contraintes telles que $\mathcal{A}_1; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}_2$. Si on a une variable X et une valeur $\mathtt{v}$ telles que $\mathcal{A}_2(X) = \mathtt{v}$ alors on a $\mathcal{A}_1 \oplus (X \leftarrow \mathtt{v}); \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}_2$.*

Démonstration. La preuve se fait par induction structurelle sur le store de contraintes σ. □

7.4 Traduction des programmes normalisés

7.4.1 Variables, motifs et noms de fonctions

On commence par donner la définition des traductions syntaxiques des variables et des différents symboles utilisés dans les programmes FoCal normalisés.

Définition 7.4.1 Variables fraîches**.**
Tout comme pour la normalisation des expressions FoCal, on se donne un ensemble de variables fraîches noté $\mathcal{F}reshC$.

Définition 7.4.2 Traduction de variables**.**
On définit la notion d'environnement de traduction de variables. Un tel environnement fait la correspondance entre une variable FoCal et une variable du monde des contraintes. On note l'environnement de traduction des variables $\mathcal{T}_x$. Pour une variable de contraintes X, si $\mathcal{T}_x$ l'associe à la variable FoCal y alors on note : $X = \mathcal{T}_x(y)$.

La fonction $\mathcal{T}_x$ permet de faire le lien entre un programme FoCal et sa traduction. Intuitivement, à chaque fois qu'une nouvelle variable x est introduite par la construction **let**, une nouvelle variable fraîche lui est associée.

En plus de l'environnement de traduction de variables, nous avons une fonction qui fait le lien entre les constructeurs des types concrets de FoCal et les constructeurs de termes du système de contraintes.

Définition 7.4.3 Constructeurs de types**.**
Nous traduisons un constructeur de valeurs FoCal en symbole de terme du système de contraintes par l'intermédiaire d'une fonction injective. Cette fonction n'est pas explicitée, la traduction est effectuée syntaxiquement. Un constructeur de valeur de FoCal, noté C^n est traduit en un symbole de terme noté $\mathtt{c}^n$.

Cette fonction a pour propriété d'être injective afin d'assurer que des constructeurs différents sont traduits dans des symboles de terme différents.

Nous pouvons maintenant donner la fonction de traduction des motifs utilisés dans le filtrage par rapport à un environnement de traduction de variables. La traduction des motifs est définie à partir des deux définitions précédentes.

Définition 7.4.4 Traduction d'un filtre**.**
On définit la fonction $\mathcal{T}_p$ qui traduit un motif vers un terme du système de contraintes en fonction d'un environnement de traduction de variables $\mathcal{T}_x$ de la manière suivante :

$$\mathcal{T}_p(C^0) = \mathtt{c}^0 \qquad \mathcal{T}_p(C^n(x_1, \ldots, x_n) = \mathtt{c}^n(\mathcal{T}_x(x_1), \ldots, \mathcal{T}_x(x_n))$$

Il ne nous reste plus qu'à définir la traduction des symboles de fonction. Comme pour les constructeurs, nous définissons une traduction à partir d'une fonction non nommée. La traduction apparaît uniquement au travers des notations.

Définition 7.4.5 Symbole de fonction**.**
Chaque symbole de fonction FoCal est traduit en un symbole de clause dans le domaine des contraintes. Cette traduction est effectuée par l'intermédiaire d'une fonction non nommée. Ainsi, pour un symbole de fonction f*, on note le symbole associé dans le monde des contraintes* `f`*. Afin d'assurer que deux fonctions différentes soient associées à deux symboles différents, cette fonction est injective.*

Définition 7.4.6 Valeurs**.**
Soit v une valeur, on note `v` *la valeur correspondante dans le domaine des contraintes.*

On peut faire le lien entre les environnements de variable des programmes en forme normale et les affectations.

Définition 7.4.7 Affectation/Environnement**.**
Soient un environnement de variable $\mathcal{E}_\mathtt{f}$ et un environnement de traduction de variables $\mathcal{T}_x$ défini sur les variables de $\mathcal{E}_\mathtt{f}$. On définit l'affectation $\mathcal{A}$ correspondant à $\mathcal{E}_\mathtt{f}$ modulo la traduction des variables $\mathcal{T}_x$, par le séquent $\mathcal{T}_x, \mathcal{A} \models \mathcal{E}_\mathtt{f}$, il est défini par :

$$\frac{}{\emptyset \models \emptyset}\ \textsc{Empty} \qquad \frac{\mathcal{T}_x(x) = X \qquad \mathcal{A} \models \mathcal{E}_v}{\mathcal{A} \oplus (X, \mathtt{v}) \models \mathcal{E}_v \oplus (x, v)}\ \textsc{Overload}$$

Dans la suite, nous omettrons $\mathcal{T}_x$ et noterons $\mathcal{A} \models \mathcal{E}_\mathtt{f}$ lorsqu'il n'y a pas d'ambiguïté.

7.4.2 Les expressions

La traduction en contraintes des expressions normalisées est donnée dans la figure 7.9. La traduction est effectuée en utilisant des jugements de traduction de la forme :

$$\mathcal{T}_x; R \vdash_C e \longmapsto \sigma$$

Un tel jugement se lit « l'expression e se traduit en le store de contraintes σ dans l'environnement de traduction de variables $\mathcal{T}_x$; la valeur vers laquelle s'évalue e est concrétisée par la variable de contrainte R ». Les règles de traduction sont données dans la figure 7.9 :

$$\frac{\mathcal{T}_x(x_1) = X_1 \qquad \dots \qquad \mathcal{T}_x(x_n) = X_n}{\mathcal{T}_x; R \vdash_C \mathtt{f}(x_1, \dots, x_n) \longmapsto \mathtt{f}(R, X_1, \dots, X_n)} \textsc{Function}$$

$$\frac{\begin{array}{c} X \in \mathcal{F}reshC \qquad \mathcal{T}_x; X \vdash_C e_1 \longmapsto \sigma_1 \\ \mathcal{T}_x \oplus (x \leftarrow X); R \vdash_C e_2 \longmapsto \sigma_2 \end{array}}{\mathcal{T}_x; R \vdash_C \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \longmapsto \sigma_1 \wedge \sigma_2} \textsc{Let}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C v \longmapsto R =_\diamond v} \textsc{Value} \qquad \frac{\mathcal{T}_x(x) = X}{\mathcal{T}_x; R \vdash_C x \longmapsto R =_\diamond X} \textsc{Var}$$

$\diamond \in \{fd, h\}$ selon le type de la valeur/variable

$$\frac{\begin{array}{c} \mathcal{T}_x(x) = X \qquad \mathcal{T}_x; R \vdash_C e_1 \longmapsto \sigma_1 \\ \mathcal{T}_x; R \vdash_C e_2 \longmapsto \sigma_2 \end{array}}{\mathcal{T}_x; R \vdash_C \textbf{if}\, x \,\textbf{then}\, e_1 \,\textbf{else}\, e_2 \longmapsto \mathtt{ite}(X, \sigma_1, \sigma_2)} \textsc{If}$$

$$\frac{\mathcal{T}_x(x) = X \qquad \mathcal{T}_x; R \vdash_C e_i \longmapsto \sigma_i \qquad \forall i \in [\![1, n]\!]}{\mathcal{T}_x; R \vdash_C \begin{array}{l} \textbf{match}\, x \,\textbf{with} \\ \mid pat_1 \to e_1 \\ \quad \vdots \\ \mid pat_n \to e_n \end{array} \longmapsto \begin{array}{l} \mathtt{match}(X, [\\ \quad \mathtt{pattern}(X =_h \mathcal{T}_p(pat_1), \sigma_1) \\ \quad \vdots \\ \quad \mathtt{pattern}(X =_h \mathcal{T}_p(pat_1), \sigma_n)], \mathtt{fail}) \end{array}} \textsc{Match}$$

$$\frac{\mathcal{T}_x(x) = X \qquad \mathcal{T}_x; R \vdash_C e_i \longmapsto \sigma_i \qquad \forall i \in [\![1, n+1]\!]}{\mathcal{T}_x; R \vdash_C \begin{array}{l} \textbf{match}\, x \,\textbf{with} \\ \mid pat_1 \to e_1 \\ \quad \vdots \\ \mid pat_n \to e_n \\ \mid \_ \quad \to e_{n+1} \end{array} \longmapsto \begin{array}{l} \mathtt{match}(X, [\\ \quad \mathtt{pattern}(X =_h \mathcal{T}_p(pat_1), \sigma_1) \\ \quad \vdots \\ \quad \mathtt{pattern}(X =_h \mathcal{T}_p(pat_1), \sigma_n) \quad], \\ \quad \sigma_{i+1}) \end{array}} \textsc{MatchCatch}$$

FIGURE 7.9 – Traduction des expressions normalisées en contraintes

La règle FUNCTION permet de traduire un appel à une fonction de programme normalisé $\mathtt{f}(x_1, \ldots, x_n)$. On commence par chercher dans l'environnement les symboles de variable de contrainte correspondant aux variables de l'appel à l'aide de $\mathcal{T}_x$. La contrainte correspondante est alors l'appel à la clause `f` sur les variables trouvées.

La règle LET permet de traduire une définition de variable locale $\textbf{let}\, x = e_1\ \textbf{in}\ e_2$. Elle traduit d'abord l'expression e_1 en précisant que la valeur de retour doit se placer dans une variable fraîche X. Ensuite, l'expression e_2 est traduite dans l'environnement $\mathcal{T}_x$ auquel on ajoute la traduction de la variable x en X.

La règle VAR permet de traduire une expression qui est une variable. La contrainte correspondante impose alors que la variable de retour soit identique à la variable.

La règle VALUE effectue la même chose que VAR dans le cas où l'expression est une valeur.

La règle IF traduit une conditionnelle. Elle commence par traduire les deux expressions e_1 et e_2 en les ensembles de contraintes respectifs σ_1 et σ_2. La conditionnelle est alors traduite en la contrainte $\mathtt{ite}(X, \sigma_1, \sigma_2)$ où X est la variable de contraintes associée à la condition x.

Les règles MATCH et MATCHCATCH traduisent un filtrage par motif avec ou sans motif attrape-tout final. Cela consiste à traduire d'abord l'ensemble des expressions e_i associées aux motifs en un ensemble de système de contraintes σ_i. Par ailleurs, les motifs sont traduits dans le terme de contrainte adéquat, la variable de contrainte X correspondant à la variable filtrée x est retrouvée. La contrainte finale est alors un appel à la contrainte `match` sur X, la liste de $\mathtt{pattern}(X =_h \mathcal{T}_p(pat_i), \sigma_i)$ avec pour dernier argument `fail` dans le cas où le motif attrape-tout n'est pas présent et l'ensemble de contraintes σ_{n+1} dans le cas contraire.

Exemple 7.4.8. Soit l'expression e suivante

$$\begin{array}{ll} \textbf{let}\, x = & \textbf{let}\, y = (t < 6)\ \textbf{in} \\ & \textbf{if}\, y\ \textbf{then}\ 5\ \textbf{else}\ 10\ \textbf{in} \\ x * t & \end{array}$$

La traduction en contrainte de l'expression donne le jugement

$$(x \leftarrow X), (t \leftarrow T); R \vdash_C e \longmapsto \sigma$$

donne le système suivant de contraintes :

$$\begin{array}{l} Y =_{fd} (T < 6), \\ \mathtt{ite}(Y, X =_{fd} 5, X =_{fd} 10), \\ R =_{fd} X * T \end{array}$$

On remarque que les **let** imbriqué ont donnés lieu à des contrainte qui se suivent. La variable x a été transférée à l'intérieur de la contrainte `ite`

Nous pouvons ensuite formuler des théorèmes qui sont utilisés dans la suite pour prouver les théorèmes fondamentaux de notre traduction.

Théorème 7.4.9. *Soient une expression e, un environnement de traduction de variables $\mathcal{T}_x$, une variable R et un store de contraintes σ tels que $\mathcal{T}_x; R \vdash_C e \longmapsto \sigma$.*
Si on a deux affectations $\mathcal{A}_1$ et $\mathcal{A}_2$ et un environnement de clause tels que $\mathcal{A}_1; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}_2$ alors il existe une valeur `v` telle que $\mathcal{A}_2(R) = v$.

Démonstration. On sait par le théorème 7.3.20 que $\mathcal{A}_2$ est totale. Il reste à montrer que la variable R apparaît dans $\mathcal{A}_2$. On montre que R est dans le store σ par induction structurelle sur e. Il en découle que R est définie par $\mathcal{A}_2$ car elle est totale. □

Théorème 7.4.10. *Soient deux expressions e_1 et e_2, deux variables x et R, deux stores σ_1, σ_2, un environnement de traduction de variables $\mathcal{T}_x$ tels que $\mathcal{T}_x; R \vdash_C \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \longmapsto \sigma_1 \wedge \sigma_2$. Alors, la variable X associée à x n'apparaît pas dans le store de contraintes σ_1.*

Démonstration. Cela se montre par définition de la règle LET. Lorsqu'on traduit e_1, la variable R n'est pas la variable de retour et n'est pas dans $\mathcal{T}_x$. □

Théorème 7.4.11. *Soient deux affectations $\mathcal{A}_1$ et $\mathcal{A}_2$, une variable X ainsi qu'un store de contraintes σ tels que $\{X\} \lhd \mathcal{A}_1; \mathcal{E}_{\texttt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}_2$. On supposant que X n'apparait pas dans σ nous avons $\mathcal{A}_1; \mathcal{E}_{\texttt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}_2$.*

Démonstration. Ceci se montre en remarquant que X n'influence pas l'ensemble des solutions de la σ. La valeur donnée à X dans $\mathcal{A}_1$ n'a pas d'importance. □

Théorème 7.4.12. *Soient deux affectations $\mathcal{A}_1$ et $\mathcal{A}_2$ et un store de contraintes tels que $\mathcal{A}_1; \mathcal{E}_{\texttt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}_2$. Pour une expression e et une variable X, si $\mathcal{T}_x; X \vdash_C e \longmapsto \sigma$, alors on a $\{X\} \lhd \mathcal{A}_1; \mathcal{E}_{\texttt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}_2$.*

Démonstration. La preuve se fait par induction structurelle sur le store de contraintes σ. □

Théorème 7.4.13. *Soient une expression $e = \textbf{let}\, x = e_1 \,\textbf{in}\, e_2$ et deux stores σ_1 et σ_2 et une variable R. Si $\mathcal{T}_x; R \vdash_C e \longmapsto \sigma_1 \wedge \sigma_2$ alors les variables, autre que celle correspondant à x, de σ_1 à gauche d'une égalité n'apparaissent pas dans σ_2.*

Démonstration. La preuve se fait par induction structurelle sur la traduction. □

7.4.3 Environnement de fonction

Définition 7.4.14 Environnement de fonction**.**
Soit $\mathcal{E}_{\texttt{f}}$ un environnement de fonction. On définit $\mathcal{E}_{\texttt{cl}}$ comme étant l'environnement de clause correspondant à $\mathcal{E}_{\texttt{f}}$ tel que :

$$\mathcal{E}_{\texttt{cl}} = \{\texttt{f} \leftarrow <R, X_1, \ldots, X_n \rightsquigarrow \sigma> \mid \begin{array}{l} \mathcal{E}_f(f) = <x_1, \ldots, x_n \rightsquigarrow e> \quad \wedge \\ (x_1 \leftarrow X_1), \ldots, (x_n \leftarrow X_n); R \vdash_C e \longmapsto \sigma \ \} \end{array}$$

Où R, X_1, …, X_n sont des variables fraîches

7.5 Correction et complétude de la traduction

Dans cette section, nous montrons les deux théorèmes fondamentaux de notre traduction des fonctions FMON vers les contraintes. Le premier théorème est la correction de la traduction, à savoir que si un programme FMON s'évalue en une valeur v alors il existe une solution du système de contraintes correspondantes qui associe aux variables d'entrées et à la variable de retour les valeurs correspondantes. Le deuxième est le théorème de complétude qui est la réciproque du précédent.

Le premier théorème nous assure que tous les comportements du programme sont capturés par les solutions du système de contrainte. Il fait correspondre toutes les évaluations possibles d'une expression à au moins une solution du système de contraintes. Ainsi, la traduction d'un programme FMON donne un système de contraintes qui contient toutes les informations du dit programme.

Le deuxième théorème nous indique que toutes les solutions d'un système de contraintes issu d'un programme FMON correspondent à un couple (valeurs d'entrées, valeur de retour) du programme FMON en question. Intuitivement, il s'agit du théorème qui justifie notre travail de traduction en contrainte, n'importe quelle solution du système de contraintes est susceptible de sensibiliser un comportement du programme. En parallèle du premier théorème, ce théorème indique qu'une solution d'un système de contraintes correspond à au moins une évaluation du programme FMON initial.

Pour résumer, à eux deux, ces théorèmes assurent qu'il y a une correspondance biunivoque entre les solutions du système de contraintes obtenu à partir d'un programme FMON et les valeurs obtenues par évaluation du programme FMON.

Théorème 7.5.1 (intermédiaire pour la correction)**.** *Soient une expression e, un système de contraintes σ et une valeur v telle que :*

$$\mathcal{E}_v; \mathcal{E}_f \vdash_N e \rhd v$$
$$\mathcal{T}_x; R \vdash_C e \longmapsto \sigma$$

soit encore une affectation $\mathcal{A}$ telle que

$$\{R\} \lhd \mathcal{A} \models \mathcal{E}_v$$
$$\mathcal{A}(R) = \mathtt{v}$$

Il existe une affectation $\mathcal{A}'$ telle que :

$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}'$$

Démonstration. La preuve se fait par induction sur l'évaluation de e :

Cas NVar

On a alors $\sigma = \{R =_{fd} X\}$ ou $\sigma = \{R =_h X\}$ avec $\mathcal{T}_x(x) = X$, l'égalité utilisée est définie en fonction du type de x. La conclusion est trivialement vérifiée.

Cas NEntier NConstructeur NConstante

Ces deux cas sont similaires à NVar, ils sont vérifiés en utilisant les hypothèses.

Cas NIf_true NIf_false NFiltrage

Ces trois cas sont faciles, ils se prouvent par analyse de cas sur la valeur de la variable testée et puis en appliquant la bonne hypothèse d'induction.

Cas NApply

Sous les hypothèses :

$$\mathcal{E}_v(x_1) = v_1, \ldots, \mathcal{E}_v(x_n) = v_n \tag{7.21}$$
$$\mathcal{E}_{\mathsf{f}}(\mathsf{f}) = <x'_1, \ldots, x'_n \rightsquigarrow e_f> \tag{7.22}$$
$$(x'_1, v_1), \ldots, (x'_n, v_n); \mathcal{E}_{\mathsf{f}} \vdash_N e_f \rhd v \tag{7.23}$$
$$\mathcal{E}_v; \mathcal{E}_{\mathsf{f}} \vdash_N \mathsf{f}(x_1, \ldots, x_n) \rhd v \tag{7.24}$$
$$\mathcal{T}_x; R \vdash_C \mathsf{f}(x_1, \ldots, x_n) \longmapsto \sigma \tag{7.25}$$
$$\{R\} \lhd \mathcal{A} \models \mathcal{E}_v \tag{7.26}$$
$$\mathcal{A}(R) = v \tag{7.27}$$

Avec en plus l'hypothèse d'induction suivante :

$$\begin{array}{l}\forall\ \mathcal{T}_x\ \ R\ \ \mathcal{A}\ \ \sigma \\ \quad (x'_1, v_1), \ldots, (x'_n, v_n); \mathcal{E}_{\mathtt{f}} \vdash_N e_f \rhd v \Rightarrow \\ \quad \mathcal{T}_x; R \vdash_C e_f \longmapsto \sigma \Rightarrow \\ \quad \mathcal{A} \models (x'_1, v_1), \ldots, (x'_n, v_n) \Rightarrow \\ \qquad\qquad \exists\, \mathcal{A}', \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}'\end{array} \tag{7.28}$$

Il faut prouver qu'il existe une affectation $\mathcal{A}'$ tel que :

$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}'$$

Par l'hypothèse 7.24 on en déduit que $\sigma = \mathtt{f}(R, X_1, \ldots, X_n)$ avec $\mathcal{T}_x(x_1) = X_1$, ..., $\mathcal{T}_x(x_n) = X_n$. On ne peut alors appliquer que la règle CALL pour arriver à ce but. On aura alors $\mathcal{A} = \mathcal{A}'$ et il faut cependant montrer qu'il existe un store de contraintes σ' et une affectation $\mathcal{A}_2$ telle :

$$\mathcal{E}_{\mathtt{cl}}(\mathtt{f}) = <R', X'_1, \ldots, X'_n \rightsquigarrow \sigma'> \tag{7.29}$$

$$(R' \leftarrow \mathcal{A}(R)), (X'_1 \leftarrow \mathcal{A}(X_1)) \ldots, (X'_n \leftarrow \mathcal{A}(X_n)); \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma' \longmapsto \mathcal{A}_2 \tag{7.30}$$

7.29 est vérifié par définition de $\mathcal{E}_{\mathtt{cl}}$.
7.30 se montre en utilisant l'hypothèse d'induction 7.28. Les quatre prémisses du théorème sont vraies respectivement par l'hypothèse 7.23, la définition de $\mathcal{E}_{\mathtt{cl}}$, la définition de $\models$ et enfin l'hypothèse 7.27.

Cas NLET

Dans ce cas, on a $e = \mathbf{let}\, x = e_1\ \mathbf{in}\ e_2$. On part alors des hypothèses suivantes obtenues par la décomposition de l'expression :

$$\mathcal{E}_v; \mathcal{E}_{\mathtt{f}} \vdash_N e_1 \rhd v_1 \tag{7.31}$$

$$\mathcal{E}_v \oplus (x \leftarrow v_1); \mathcal{E}_{\mathtt{f}} \vdash_N e_2 \rhd v_2 \tag{7.32}$$

$$\mathcal{T}_x; R \vdash_C \mathbf{let}\, x = e_1\ \mathbf{in}\ e_2 \longmapsto \sigma \tag{7.33}$$

$$\mathcal{E}_v; \mathcal{E}_{\mathtt{f}} \vdash_N \mathbf{let}\, x = e_1\ \mathbf{in}\ e_2 \rhd v_2 \tag{7.34}$$

$$\{R\} \lhd \mathcal{A} \models \mathcal{E}_v \tag{7.35}$$

$$\mathcal{A}(R) = v_2 \tag{7.36}$$

Et on a les deux hypothèses d'induction suivantes :

$$\begin{array}{l}\forall\ \mathcal{T}_x\ \ R\ \ \mathcal{A}\ \ \sigma \\ \quad \mathcal{E}_v; \mathcal{E}_{\mathtt{f}} \vdash_N e_1 \rhd v_1 \Rightarrow \\ \quad \mathcal{T}_x; R \vdash_C e_1 \longmapsto \sigma \Rightarrow \\ \quad \{R\} \lhd \mathcal{A} \models \mathcal{E}_v \Rightarrow \\ \quad \mathcal{A}(R) = v_1 \Rightarrow \\ \quad\ \exists\, \mathcal{A}', \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}'\end{array} \tag{7.37}$$

$$\begin{array}{l}\forall\ \mathcal{T}_x\ \ R\ \ \mathcal{A}\ \ \sigma \\ \quad \mathcal{E}_v \oplus (x \leftarrow v_1); \mathcal{E}_{\mathtt{f}} \vdash_N e_2 \rhd v_2 \Rightarrow \\ \quad \mathcal{T}_x; R \vdash_C e_2 \longmapsto \sigma \Rightarrow \\ \quad \{R\} \lhd \mathcal{A} \models \mathcal{E}_v \oplus (x \leftarrow v_1) \Rightarrow \\ \quad \mathcal{A}(R) = v_2 \Rightarrow \\ \qquad\qquad \exists\, \mathcal{A}', \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}'\end{array} \tag{7.38}$$

Il faut prouver qu'il existe une affectation $\mathcal{A}'$ telle que :

$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}' \tag{7.39}$$

Par l'hypothèse 7.33 on déduit que $\sigma = \sigma_1 \wedge \sigma_2$ avec (pour un X frais) :

$$\mathcal{T}_x; X \vdash_C e_1 \longmapsto \sigma_1 \tag{7.40}$$
$$\mathcal{T}_x, (x \leftarrow X); R \vdash_C e_2 \longmapsto \sigma_2 \tag{7.41}$$

La seule règle applicable de $\vdash_S$ pour prouver 7.39 est Conjonct2. Il faut donc trouver deux affectations $\mathcal{A}_1$ et $\mathcal{A}'$ telles que les deux affirmations suivantes soient valables :

$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}_1 \tag{7.42}$$
$$\mathcal{A}_1; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_2 \longmapsto \mathcal{A}' \tag{7.43}$$

Pour prouver 7.42, on commence par prouver l'affirmation suivante et on conclut à l'aide des théorèmes 7.4.12 et 7.4.11 (on remarque que R n'apparaît pas dans σ_1 grâce à l'hypothèse 7.40 et parce que $\mathcal{T}_x$ n'associe R à aucune variable d'expression) :

$$(\{R\} \lhd \mathcal{A}) \oplus (X \leftarrow v_1); \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}_1$$

ce qui est possible car R n'est pas une variable libre de σ_1 et par 7.40. Ce dernier but se montre par l'hypothèse 7.37 dont les prémisses deviennent :

$$\begin{array}{l} \mathcal{E}_v; \mathcal{E}_{\mathtt{f}} \vdash_N e_1 \rhd v_1 \\ \mathcal{T}_x; X \vdash_C e_1 \longmapsto \sigma_1 \\ \{X\} \lhd ((\{R\} \lhd \mathcal{A}) \oplus (X \leftarrow v_1)) \models \mathcal{E}_v \\ ((\{R\} \lhd \mathcal{A}) \oplus (X \leftarrow v_1))(X) = v_1 \end{array}$$

Ils se prouvent respectivement par les hypothèses 7.31, 7.40, 7.35 (après simplification de l'affectation) et par définition de la surcharge d'une affectation.

Pour prouver 7.43, on commence pour montrer l'affirmation intermédiaire :

$$\mathcal{A} \oplus (X \leftarrow v_1); \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_2 \longmapsto \mathcal{A}' \tag{7.44}$$

Elle se prouve par l'hypothèse d'induction 7.38 avec les hypothèses 7.32, 7.41, 7.35 et l'hypothèse 7.36.

Nous remarquons par le théorème 7.4.13 qu'en dehors de X, les variables qui se trouvent à gauche d'une égalité dans σ_1 n'apparaissent pas dans σ_2. Or, comme le prédicat de test de solution surcharge une affectation en ajoutant que des variables qui apparaissent à gauche d'une égalité, on en déduit que $\mathcal{A}_1$ est $\mathcal{A}$ dans lequel on a ajouté les définitions de variables qui ne sont pas dans σ_2. On peut appliquer des renforcements sur 7.43 pour retrouver 7.44. Il faut ajouter que lors de la preuve du premier but, on a montré que $\mathcal{A}_1(X) = v_1$.

□

Nous pouvons maintenant formuler le théorème de correction de notre traduction.

Théorème 7.5.2 (correction)**.** *Soient une expression e, un système de contraintes σ et une valeur v telle que :*

$$\begin{array}{l}\mathcal{E}_v; \mathcal{E}_f \vdash_N e \rhd v \\ \mathcal{T}_x; R \vdash_C e \longmapsto \sigma\end{array}$$

soit encore une affectation $\mathcal{A}$ telle que

$$\mathcal{A} \models \mathcal{E}_v$$

Il existe une affectation $\mathcal{A}'$ telle que :

$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S R = \mathtt{v}, \sigma \longmapsto \mathcal{A}'$$

Démonstration. Se prouve à l'aide du théorème 7.5.1. □

Théorème 7.5.3 (intermédiaire pour la complètude)**.** *Soit une expression e et un système de contraintes σ tels que :*

$$\mathcal{T}_x; R \vdash_C e \longmapsto \sigma$$

soit encore un environnement de variable $\mathcal{E}_v$ et deux affectations $\mathcal{A}$ et $\mathcal{A}'$ tels que :

$$\begin{array}{l}\{R\} \lhd \mathcal{A} \models \mathcal{E}_v \\ \mathcal{A}(R) = \mathtt{v} \\ \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}'\end{array}$$

on a l'affirmation suivante :

$$\mathcal{E}_v; \mathcal{E}_f \vdash_N e \rhd v$$

Démonstration. Par induction sur la définition de $\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}'$:

Cas CALL ($\mathtt{f}(X_1, \ldots, X_n)$)

Pour des variables $X_1, X_1', \ldots, X_n, X_n'$, un symbole de fonction $\mathtt{f}$ et deux stores de contraintes σ et σ'. Sous les hypothèses :

$$(X_1' \leftarrow \mathcal{A}(X_1)), \ldots, (X_n' \leftarrow \mathcal{A}(X_n)); \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}' \tag{7.45}$$

$$\mathcal{E}_{\mathtt{cl}}(\mathtt{f}) = <X_1', \ldots, X_n' \rightsquigarrow \sigma> \tag{7.46}$$

de l'hypothèse d'induction :

$$\begin{array}{l}\forall\ R\ v\ e\ \mathcal{T}_x\ \mathcal{E}_v \\ \quad \mathcal{T}_x; R \vdash_C e \longmapsto \sigma \Rightarrow \\ \quad \{R\} \lhd ((X_1' \leftarrow \mathcal{A}(X_1)), \ldots, (X_n' \leftarrow \mathcal{A}(X_n))) \models \mathcal{E}_v \Rightarrow \\ \quad ((X_1' \leftarrow \mathcal{A}(X_1)), \ldots, (X_n' \leftarrow \mathcal{A}(X_n)))(R) = \mathtt{v} \Rightarrow \\ \quad (X_1' \leftarrow \mathcal{A}(X_1)), \ldots, (X_n' \leftarrow \mathcal{A}(X_n)); \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}' \Rightarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathcal{E}_v; \mathcal{E}_{\mathtt{f}} \vdash_N e \rhd v\end{array} \tag{7.47}$$

et des hypothèses :

$$\mathcal{T}_x; R \vdash_C e \longmapsto \mathtt{f}(X_1, \ldots, X_n) \tag{7.48}$$
$$\{R\} \lhd \mathcal{A} \models \mathcal{E}_v \tag{7.49}$$
$$\mathcal{A}(R) = \mathtt{v} \tag{7.50}$$
$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \mathtt{f}(X_1, \ldots, X_n) \longmapsto \mathcal{A} \tag{7.51}$$

Il faut montrer qu'on obtient : $\mathcal{E}_v; \mathcal{E}_{\mathtt{f}} \vdash_N e \rhd v$.
De 7.48 il découle que $e = \mathtt{f}(x_2, \ldots, x_n)$ avec $X_i = \mathcal{T}_x(x_i)$ pour $2 \leq i \leq n$ et $X_1 = R$. De plus, par la définition 7.4.14 et l'hypothèse 7.46, on obtient les hypothèses suivantes :

$$\mathcal{E}_{\mathtt{f}}(\mathtt{f}) = <x'_2, \ldots, x'_n \rightsquigarrow e_f> \tag{7.52}$$
$$(x'_2 \leftarrow X'_2), \ldots (x'_n \leftarrow X'_n); R \vdash_C e_f \longmapsto \sigma \tag{7.53}$$
$$R = X'_1 \tag{7.54}$$

Comme $e = \mathtt{f}(x_2, \ldots, x_n)$ alors la conclusion s'obtient par application de la règle NApply. Il faut donc prouver :

$$(x'_2, \mathcal{E}_v(x_2)), \ldots, (x'_n, \mathcal{E}_v(x_n)); \mathcal{E}_{\mathtt{f}} \vdash_N e_f \rhd v$$

Ce qui se montre en utilisant l'hypothèse 7.47. Les quatre pré-requis de cette hypothèse se montrent facilement. La première avec l'hypothèse 7.53, la deuxième par la définition de $\models$ de $\lhd$ et de $X = X'_1$, la troisième par $X = R$ et l'hypothèse 7.50 et la dernière est exactement l'hypothèse 7.45.

Cas Conjonct1

Ce cas n'est pas possible car le jugement $\vdash_C$ ne permet pas d'obtenir directement des stores de la forme c, σ.

Cas Conjonct2 ($\sigma_1 \wedge \sigma_2$)

Dans ce cas, il faut montrer sous les hypothèses :

$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}_1 \tag{7.55}$$
$$\begin{array}{l} \forall\ R\ v\ e\ \mathcal{T}_x\ \mathcal{E}_v \\ \quad \mathcal{T}_x; R \vdash_C e \longmapsto \sigma_1 \Rightarrow \\ \quad \{R\} \lhd \mathcal{A} \models \mathcal{E}_v \Rightarrow \\ \quad \mathcal{A}(R) = \mathtt{v} \Rightarrow \\ \quad \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}_1 \Rightarrow \\ \qquad \mathcal{E}_v; \mathcal{E}_{\mathtt{f}} \vdash_N e \rhd v \end{array} \tag{7.56}$$
$$\mathcal{A}_1; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_2 \longmapsto \mathcal{A}_2 \tag{7.57}$$
$$\begin{array}{l} \forall\ R\ v\ e\ \mathcal{T}_x\ \mathcal{E}_v \\ \quad \mathcal{T}_x; R \vdash_C e \longmapsto \sigma_2 \Rightarrow \\ \quad \{R\} \lhd \mathcal{A}_1 \models \mathcal{E}_v \Rightarrow \\ \quad \mathcal{A}_1(R) = \mathtt{v} \Rightarrow \\ \quad \mathcal{A}_1; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_2 \longmapsto \mathcal{A}_2 \Rightarrow \\ \qquad \mathcal{E}_v; \mathcal{E}_{\mathtt{f}} \vdash_N e \rhd v \end{array} \tag{7.58}$$

Et sous les hypothèses suivantes :

$$\mathcal{T}_x; R \vdash_C e \longmapsto \sigma_1 \wedge \sigma_2 \tag{7.59}$$
$$\{R\} \lhd \mathcal{A} \models \mathcal{E}_v \tag{7.60}$$
$$\mathcal{A}(R) = \mathtt{v} \tag{7.61}$$
$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \wedge \sigma_2 \longmapsto \mathcal{A}_2 \tag{7.62}$$

Que $\mathcal{E}_v; \mathcal{E}_{\mathtt{f}} \vdash_N e \rhd v$.
Par l'hypothèse 7.59, on a $e = \textbf{let}\, x = e_1\, \textbf{in}\, e_2$, on obtient alors les hypothèses suivantes (pour X frais) :

$$\mathcal{T}_x; X \vdash_C e_1 \longmapsto \sigma_1 \tag{7.63}$$
$$\mathcal{T}_x \oplus (x \leftarrow X); R \vdash_C e_2 \longmapsto \sigma_2 \tag{7.64}$$

En appliquant le théorème 7.4.9 sur les hypothèses 7.63 et 7.55, on montre qu'il y a une valeur $\mathtt{v}_1$ telle que $\mathcal{A}_1(X) = \mathtt{v}_1$. Avec le théorème 7.3.21 et l'hypothèse 7.55 on en déduit :

$$\mathcal{A} \oplus (X \leftarrow \mathtt{v}_1), \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}_1 \tag{7.65}$$

De plus, par le théorème 7.4.10, la variable R n'est pas dans σ_1 et donc on peut effacer la définition de R dans $\mathcal{A}$ dans la dernière hypothèse :

$$(\{R\} \lhd \mathcal{A}) \oplus (X \leftarrow \mathtt{v}_1); \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}_1 \tag{7.66}$$

Nous remarquons maintenant que les deux affirmations suivantes sont vérifiées :

$$\mathcal{E}_v; \mathcal{E}_{\mathtt{f}} \vdash_N e_1 \rhd v_1 \tag{7.67}$$
$$\mathcal{E}_v, (x \leftarrow v_1); \mathcal{E}_{\mathtt{f}} \vdash_N e_2 \rhd v \tag{7.68}$$

La première de ces deux affirmations se montre avec l'hypothèse d'induction 7.56. Sur les quatre pré-requis de 7.56 la première se montre par 7.63, la deuxième en utilisant la définition de $\oplus$ et $\lhd$ et le fait que X est frais et par 7.60, la troisième se montre par la définition de la surcharge et la dernière est l'hypothèse 7.66.
Pour prouver la deuxième affirmations, il faut lui appliquer le théorème de renforcement avant d'appliquer l'hypothèse 7.57. Soit V l'ensemble des variables définies dans $\mathcal{A}_1$ et non définies dans $\mathcal{A}$ (en dehors de X). On a $V = \{Y_1, \ldots, Y_n\}$ et $\mathcal{A}_1(Y_i) = \mathtt{v}'_i$ pour $1 \leq i \leq n$. Soit n variables fraîches $y_1, \ldots, y_n$.
On applique le théorème de renforcement sur cette dernière et on obtient :

$$\mathcal{E}_v, (x \leftarrow v_1), (y_1 \leftarrow v'_1), \ldots, (y_n \leftarrow v_n); \mathcal{E}_{\mathtt{f}} \vdash_N e_2 \rhd v_2$$

On peut maintenant appliquer l'hypothèse 7.56, les quatre pré-requis sont vérifiés soit directement par une autre hypothèse, soit par la définition de $\models$.
Des deux affirmations 7.67, 7.68, on en déduit par la règle NLet :

$$\mathcal{E}_v; \mathcal{E}_{\mathtt{f}} \vdash_N \textbf{let}\, x = e_1\, \textbf{in}\, e_2 \rhd v$$

Qui est le but qu'il fallait démontrer.

Cas EqFdI EqFdX EqFdHC EqFdHX

Nous développons le cas EQFdI, les autres cas sont similaires.

Dans ce cas, on a $\sigma = (R =_{fd} i)$ pour un entier i.

Des hypothèses :

$$\mathcal{T}_x; R \vdash_C e \longmapsto \sigma \tag{7.69}$$
$$\{R\} \lhd \mathcal{A} \models \mathcal{E}_v \tag{7.70}$$
$$\mathcal{A}(R) = \mathtt{v} \tag{7.71}$$
$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}' \tag{7.72}$$

On doit démontrer

$$\mathcal{E}_v; \mathcal{E}_{\mathtt{f}} \vdash_N e \rhd v$$

De l'hypothèse 7.69 on en déduit que $e = i$. D'autre part, la seule règle de $\vdash_S$ applicable pour obtenir 7.72 est EqFdI (dans sa version ou $R \in \mathcal{A}$ par l'hypothèse 7.71). On en déduit que $\mathtt{v} = i$.

De $e = i$ et $v = i$ on déduit trivialement la conclusion.

Cas EqFdOp

Dans ce cas, la valeur de la variable contrainte est donnée par l'interprétation des opérateurs arithmétiques. Elle est identique dans FoCal et le langage de contraintes.

Cas Ite Match_pat

Dans ces deux cas, il faut faire une analyse de cas sur la valeur de la variable testée/filtrée. On conclut en utilisant directement l'hypothèse d'induction car les environnements et affectations ne varient pas dans ces cas.

□

Théorème 7.5.4 (complétude)**.** *Soient une expression e, un système de contraintes σ, une valeur v et deux affectations $\mathcal{A}$ et $\mathcal{A}'$ telles que :*

$$\mathcal{T}_x; R \vdash_C e \longmapsto \sigma$$
$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S R = \mathtt{v}, \sigma \longmapsto \mathcal{A}'$$
$$\mathcal{A} \models \mathcal{E}_v$$

Nous avons :

$$\mathcal{E}_v; \mathcal{E}_f \vdash_N e \rhd v$$

Démonstration. Se prouve à l'aide du théorème 7.5.3. □

7.6 Résumé sur la génération de jeux de test

Maintenant que nous avons donné le formalisme de traduction des programmes FoCal en système de contraintes et donné une preuve de correction et de complétude de cette traduction. Nous pouvons faire le lien avec le test de propriété sur un exemple.

Considérons le programme FoCal suivant composé d'une unique espèce (nous ne présentons pas la version formelle de la définition de l'espèce pour des raisons de clarté) :

```
species fusion =
  rep = list(int);

  let rec fusion(l1, l2) =
    match l1 with
    | Nil → l2
    | Cons(e1,r1) → match l2 with
                | Nil → l1
                | Cons(e2,r2) → if e1 < e2 then
                                  !fusion(r1, l1)
                                else
                                  !fusion(r2, l2)
                end
    end;

  let rec sorted(l) =
    match l with
    | Nil → True
    | Cons(e, r) → True
    | Cons(e1, Cons(e2, r)) →
      if e1 < e2 then !sorted(Cons(e2;r)) else False
    end;

  theorem correct_fusion =
    all l1 l2 in self,
      !sorted(l1) → !sorted(l2) → !sorted(!fusion(l1,l2))
  proof:
    assumed

end

collection fusion_coll implements fusion;
```

Si nous souhaitons tester la propriété `correct_fusion`, nous devons passer par les étapes suivantes :

- On commence par identifier la précondition de la propriété :
  `fusion_coll!sorted(l1)` **and** `fusion_coll!sorted(l2)`
  la précondition est donc constituée de deux appels de fonctions à `sorted` ;

- pour chacun des éléments de la précondition, on calcule l'ensemble de dépendances par Dep :
  Dep(`fusion_coll!sorted(l1)`) = { `fusion_coll!sorted` }

  Dep(`fusion_coll!sorted(l2)`) = { `fusion_coll!sorted` }
  Ceci nous donne l'ensemble des fonctions FoCal qui vont donner lieu au programme FMON ;

- Après la traduction de `sorted`, nous obtenons l'environnement de fonctions FMON suivante :

$$(\texttt{sorted} \leftarrow <l \rightsquigarrow \left(\begin{array}{l} \textbf{match } l \textbf{ with} \\ |\text{Nil} \rightarrow \texttt{true} \\ |\text{Cons}(e, r) \rightarrow \\ \quad \textbf{match } r \textbf{ with} \\ \quad |\text{Nil} \rightarrow \texttt{true} \\ \quad |\text{Cons}(e', r') \rightarrow \textbf{let } a = e < e' \textbf{ in} \\ \quad\quad \textbf{if } a \textbf{ then } \texttt{sorted}(\text{Cons}(e', r')) \textbf{ else } \texttt{false} \end{array}\right) >)$$

– Cet environnement de fonction FMON est traduit en un environnement de clauses.

$$(\texttt{sorted}, <Z, L \rightsquigarrow \left(\begin{array}{l} \texttt{match}(X, [\\ \quad \texttt{pattern}(\texttt{nil}, Z =_h \texttt{true}), \\ \quad \texttt{pattern}(\texttt{cons}(E, R), \\ \quad\quad \texttt{match}(R, [\\ \quad\quad\quad \texttt{pattern}(\texttt{nil}, Z =_h \texttt{true}), \\ \quad\quad\quad \texttt{pattern}(\\ \quad\quad\quad\quad \texttt{cons}(e', r'), \\ \quad\quad\quad\quad \mathit{int\_lt}(A, E, E') \wedge A' =_h \texttt{cons}(E', R') \wedge \\ \quad\quad\quad\quad \texttt{ite}(A, \texttt{sorted}(Z, A'), Z =_h \texttt{false}))))) \end{array}\right) >)$$

– Nous pouvons maintenant traduire la précondition en contraintes. Il s'agit de commencer par transformer chacun des éléments de la précondition en expression FMON. Nous obtenons :

$$\texttt{sorted}(l1)$$
$$\texttt{sorted}(l2)$$

– La précondition est ensuite transformé en contraintes :

$$\texttt{sorted}(\text{R1, L1})$$
$$\texttt{sorted}(\text{R2, L2})$$

Le système de contrainte qui spécifie l'ensemble des jeux de test qu'on cherche est alors :

$$R1 = \texttt{true}, R2 = \texttt{true}, \texttt{sorted}(R1, L1), \texttt{sorted}(R2, L2)$$

À partir des solutions de ce système de contrainte, on obtient les jeux de test pour la propriété de départ.

En dernier lieu, nous remarquons que si on souhaite avoir une couverture MC/DC de la propriété, nous avons qu'à modifier les contraintes $R1 = \ldots, R2 = \ldots$, en mettant les bonnes valeurs de vérité pour chacun des buts requis par MC/DC.

7.7 Synthèse

Nous avons défini dans ce chapitre la méthodologie de transformation d'un programme FoCal en un ensemble de contraintes. Cette transformation se fait en deux étapes : une première étape de traduction des programmes FoCal en programme FMON et une deuxième étape de traduction des programmes FMON vers un ensemble de contraintes.

Nous avons prouvé la correction et la complétude des deux étapes de traduction. Pour prouver la correction de la traduction des programmes FMON en contraintes, nous avons défini un prédicat de test de solution qui prend une affectation et la complète en une affectation

totale. Ce prédicat est défini pour décider des solutions des systèmes de contraintes obtenus par traduction des programmes FMON.

Pour avoir une représentation des types concrets dans le système de contraintes, nous avons défini un domaine spécifique. Pour une variable de contraintes sur un type concret, son domaine est défini comme l'ensemble des constructeurs de tête qui peuvent lui être appliqués. Nous avons défini pour ce domaine deux contraintes : l'égalité et l'inégalité.

Nous avons défini une sémantique des contraintes `ite` et `match` dans le cadre de la vérification de solution d'un système de contraintes. Cette sémantique n'est clairement pas adapté pour la recherche d'une solution puisqu'il faut connaître par avance les valeurs des variables conditions/filtrées. Nous développons dans la section suivante une sémantique qui tire parti du mécanisme de *backtracking* de la programmation par contrainte.

Troisième partie

Implantation et conclusion

Chapitre 8

Les méta-contraintes `ite` et `match`

Dans cette partie, nous présentons les aspects implantations de notre approche. Il est découpé en deux parties. Ce présent chapitre présente la sémantique utilisée dans l'implantation des contraintes `ite` et `match` et donne une preuve d'équivalence avec la sémantique donnée dans la partie II. Le chapitre suivant décrit l'implantation de l'outil FoCalTest.

Dans le chapitre précédent, nous avons donné une sémantique pour les contraintes `ite` et `match` qui consiste à vérifier si une affectation est une solution ou non. Cette sémantique n'est pas suffisante car elle ne permet pas de faire de la recherche de solution sur les contraintes. Dans ce chapitre, nous donnons une sémantique à `ite` et à `match` sous la forme de méta-contraintes. Nous rappelons que nous avons donné la définition d'une méta-contrainte au chapitre 4. Nous montrons aussi l'équivalence entre cette sémantique et la sémantique de vérification de solutions.

8.1 Contrainte `ite`

La méta-contrainte `ite` a initialement été définie dans [GBR98, GBR00] pour simuler la construction conditionnelle des langages de programmation impératifs. Nous l'adaptons à notre cas avec la définition suivante.

Définition 8.1.1 Contrainte `ite`.
La contrainte `ite/3` correspond à un équivalent de la structure conditionnelle des langages de programmation. Elle prend en argument un nom de variable et deux ensembles de contraintes. La sémantique de $\texttt{ite}(X, T, E)$ est donnée par les quatre contraintes gardées suivantes :

1. $X = \texttt{true} \rightarrow T$
2. $X = \texttt{false} \rightarrow E$
3. $\neg(X = \texttt{true} \wedge T) \rightarrow X = \texttt{false} \wedge E$
4. $\neg(X = \texttt{false} \wedge E) \rightarrow X = \texttt{true} \wedge T$

Les deux premières contraintes gardées signifient d'une part que si la contrainte $X = \texttt{true}$ est impliquée par le système de contraintes alors le `ite` est réécrit en T et si sa négation, c'est-à-dire $X = \texttt{false}$ est impliquée par le système de contraintes, le `ite` est réécrit en E. Elles correspondent à un raisonnement en avant d'une conditionnelle.

Les deux autres contraintes gardées correspondent à un raisonnement en arrière d'une conditionnelle. Chacune d'elle teste d'abord si l'un des deux cas possibles (à savoir $X = \texttt{true} \wedge T$ et

$X = \texttt{false} \wedge E$) est détecté comme non possible et réécrit `ite` dans l'autre cas.

Exemple 8.1.2. Prenons le système de contraintes suivant :

$$\begin{array}{l} Y =_{fd} 0, \\ C_1 = (X =_{fd} 3), \texttt{ite}(C_1, Z =_{fd} 0, Z =_{fd} 1), \\ C_2 = (Z > 0), \texttt{ite}(C_2, Y =_{fd} X, Y =_{fd} 5) \end{array}$$

avec pour domaine des variables $X \in [\![0, 2^{32} - 1]\!]$ et $Z \in [\![0, 2^{32} - 1]\!]$.

D'après le première contrainte `ite` nous ne pouvons rien déduire des quatre contraintes gardées. Sur la deuxième contrainte `ite`, on constate que $Y =_{fd} 5$ est incompatible avec la première contrainte du système ($Y =_{fd} 0$). La quatrième règle de `ite` peut être appliquée pour obtenir le système de contraintes suivant :

$$\begin{array}{l} Y =_{fd} 0, \\ C_1 = (X =_{fd} 3), \texttt{ite}(C_1, Z =_{fd} 0, Z =_{fd} 1), \\ C_2 = (Z > 0), C_2 = \texttt{true}, Y =_{fd} X \end{array}$$

Maintenant, nous avons $X =_{fd} 0$ et $Z \in\in [\![1, 2^{32} - 1]\!]$. On appliquant la troisième règle de la contrainte `ite` restante nous constatons que la $Z =_{fd} 0$ est incompatible avec le domaine de Z. Et donc le système de contraintes se réécrit en

$$\begin{array}{l} Y =_{fd} 0, \\ C_1 = (X =_{fd} 3), C_1 = \texttt{false}, Z =_{fd} 1, \\ C_2 = (Z > 0), C_2 = \texttt{true}, Y =_{fd} X \end{array}$$

et par des raisonnements sur les domaines finis, on obtient la seule solution du système :

$$(X = 0, Y = 0, Z = 1)$$

Proposition 8.1.3. *Soient une variable X et deux contraintes T et E ainsi qu'un ensemble de contraintes σ.*

Sur l'ensemble de contraintes $\sigma, \texttt{ite}(X, T, E)$, si la contrainte `ite` *se réduit d'une part en C et d'autre part en C' alors les deux ensembles de contraintes σ, C et σ, C' possèdent les mêmes solutions.*

Démonstration. Pour le montrer, il faut voir la sémantique de `ite` comme un système de réécriture et montrer que les paires critiques sont joignables.

□

8.2 Contrainte `match`

La méta-contrainte `match` est le pendant sous forme de contraintes du filtrage par motif des langages fonctionnels. Tout comme la contrainte `ite` qui simule la structure conditionnelle avec la possibilité de faire du raisonnement arrière, la contrainte `match` permet de faire du raisonnement sur un ensemble de règles de filtrage en avant et en arrière.

Définition 8.2.1 Sémantique de `match/3`.
La contrainte `match` *prend 3 arguments, la variable à filtrer, une liste de motif-contraintes et un ensemble de contraintes par défaut à utiliser lorsqu'aucun des motifs n'est applicable. Les motifs utilisés dans la contrainte sont de la forme $c(X_1, \ldots, X_n)$ où c est un constructeur de termes. Les constructeurs de termes de chaque filtre sont supposés différents.*

La contrainte $\texttt{match}(X, [\texttt{pattern}(X = pat_1, \sigma_1), \ldots, \texttt{pattern}(X = pat_n, \sigma_n)], \sigma_{n+1})$ *a la sémantique suivante :*

$\forall i \in [\![1, n]\!]$

1. $X = pat_i \rightarrow \sigma_i$
2. $\neg(X = pat_i \wedge \sigma_i) \rightarrow \texttt{match}(X, [\ \texttt{pattern}(X = pat_1, \sigma_1), \ldots, \texttt{pattern}(X = pat_{i-1}, \sigma_{i-1}), \texttt{pattern}(X = pat_{i+1}, \sigma_{i+1}), \ldots, \texttt{pattern}(X = pat_n, \sigma_n)])$
3. $n = 1 \wedge \sigma_{n+1} =_{def} \texttt{fail} \rightarrow X = pat_1 \wedge \sigma_1$
4. $n = 0 \rightarrow \sigma_{n+1}$

Les deux premières règles définissent les raisonnements à partir des contraintes. La première règle définit la « sémantique avant » de l'opérateur, elle spécifie que si le motif est impliqué par l'ensemble de contraintes alors la contrainte `match` initiale est remplacée par le système de contraintes qui lui est associé. La deuxième règle donne la sémantique par raisonnement arrière d'une règle de filtrage, si le motif et le système de contraintes ne sont pas impliqués, la clause correspondante est supprimée du filtrage.

Les deux dernières règles proposent des raisonnements sur le nombre de filtres présents dans le filtrage. La troisième règle permet de déduire un filtre lorsqu'il est le seul présent dans l'ensemble des filtres et que les contraintes par défaut sont définies par `fail`, la contrainte qui échoue dans tout contexte. Enfin la dernière règle impose les contraintes par défaut lorsque tous les filtres ont été détectés non valides.

Les deux dernières règles se justifient car elles expriment les cas où n'y a qu'un seul motif pour effectuer le filtrage. Ces cas s'obtiennent après la suppression d'un certain nombre de motifs détectés comme non possible. On a alors montré que la règle restante est la seule qui peut correspondre au terme (puisque les autres filtres ont été retirés après avoir constaté qu'ils n'étaient pas compatibles).

Exemple 8.2.2. Prenons le système de contrainte suivant :

$$\texttt{match}(L, [\texttt{pattern}(\texttt{nil}, R =_{fd} 0), pattern(\texttt{cons}(X, Y), R =_{fd} X + 10)], \texttt{fail})$$

Où le domaine de L est $\{\texttt{nil}, \texttt{cons}\}$ et le domaine de R est $[\![6, 14]\!]$. Comme la contrainte $\neg(l = \texttt{nil} \wedge r = 0)$ est impliquée par le domaine courant des variables, la deuxième règle appliquée au deuxième filtre permet de réécrire le système en

$$\texttt{match}(L, [\texttt{pattern}(\texttt{cons}(x, y), R =_{fd} X + 10)], \texttt{fail})$$

Comme le dernier filtre est `fail`, nous pouvons appliquer la troisième règle est réécrire la système en

$$L =_h \texttt{cons}(x, y), R =_{fd} X + 10$$

Le domaine des variables est ensuite réduit en $R \in [\![6, 14]\!]$, $X \in [\![-4, 4]\!]$, $Y \in \{\texttt{nil}, \texttt{cons}\}$ et $L = \texttt{cons}(X, Y)$. Tous les raisonnements sont terminés.

Il faut maintenant montrer que malgré le recouvrement des contraintes gardées, la sémantique de `match` est déterministe. Par exemple, s'il ne reste qu'un seul motif il est possible que les règles 1 et 3 soient applicables simultanément tout comme pour les règles 2 et 3.

Proposition 8.2.3. *Soient une contrainte :*

$$C = \texttt{match}(X, [\texttt{pattern}(X = pat_1, \sigma_1), \ldots, \texttt{pattern}(X = pat_n, \sigma_n)], \sigma_{n+1})$$

et un ensemble de contraintes σ. Soient σ' et σ'' deux ensembles de contraintes obtenus après application d'un maximum de règles de sémantique de `match` *sur σ. On a $\sigma \cup \sigma' \equiv \sigma \cup \sigma''$.*

Démonstration. Le théorème affirme que le système de réécriture formé par la sémantique de `match` donne les mêmes résultats en terme d'équivalence entre ensembles de contraintes. Ce système de réécriture est nœthérien car le nombre de filtres diminue strictement après l'application d'une règle qui redonne une contrainte `match`. On est ainsi assuré qu'appliquer un maximum de fois les règles est un processus qui termine. Pour montrer le théorème il faut montrer que les paires critiques du système de réécriture sont joignables.

Les paires critiques sont classées selon le nombre de filtres présents dans le filtrage. On distingue trois cas $n = 0$, $n = 1$ et $n > 1$:

- $n = 0$, la seule règle applicable est la règle 4 et il n'y a pas de paire critique.
- $n = 1$, dans ce cas, on trouve trois paires critiques entre les règles 1 2, 1 3 et 2 3.
  1 2 : la paire critique est $(\sigma_1, \texttt{match}(X, [], \sigma_2))$, σ implique $X = pat_1$ et $\neg(X = pat_1)$. Le système de contraintes d'origine n'admet pas de solution et le théorème est vrai dans ce cas.

  1 3 : la paire critique est $(\sigma_1, X = pat_1 \wedge \sigma_1)$, σ implique $X = pat_1$. Comme σ implique $X = pat_1$, alors $\sigma \cup \sigma_1 \equiv (\sigma \cup X = pat_1) \cup \sigma_1 \equiv \sigma \cup X = pat_1 \wedge \sigma_1$.

  2 3 : la paire critique est $(\texttt{match}(X, [], \sigma_2), X = pat_1 \wedge \sigma_1)$ et σ implique $\neg(X = pat_1)$ et $\sigma_2 =_{def} \texttt{fail}$. Donc, $\sigma \cup X = pat_1 \wedge \sigma_1 \equiv (\sigma \wedge \neg(X = pat_1)) \cup (X = pat_1 \wedge \sigma_1) \equiv \texttt{fail}$. De plus, en appliquant la règle 4 sur le premier élément de la paire critique on obtient `fail`.
- $n > 1$, il y a trois paires critiques entre les règles 1 1, 1 2 et 2 2.
  1 1 : soit $i, j \in [\![1, n]\!]$, la paire critique est (σ_i, σ_j) avec $i \neq j$ et les contraintes impliquées par σ sont $X = pat_i$ et $X = pat_j$. Or, les motifs du match ne se recouvrent pas par hypothèse donc $X = pat_i \wedge X = pat_j \equiv \texttt{fail}$ pour $i \neq j$. On a alors $\sigma \equiv \texttt{fail}$ et les deux ensembles de contraintes obtenus après réécriture sont équivalents car $\sigma \cup \sigma_i \equiv \texttt{fail} \cup \sigma_i \equiv \texttt{fail}$ et $\sigma \cup \sigma_j \equiv \texttt{fail} \cup \sigma_j \equiv \texttt{fail}$.

  1 2 : la paire critique est

$$\begin{array}{l}(\sigma_i,\\ \texttt{match}(X, [\texttt{pattern}(X = pat_1, \sigma_1),\\ \qquad \ldots,\\ \qquad \texttt{pattern}(X = pat_{j-1}, \sigma_{j-1}),\\ \qquad \ldots,\\ \qquad \texttt{pattern}(X = pat_{j+1}, \sigma_{j+1}),)\\ \qquad \ldots,\\ \qquad \texttt{pattern}(X = pat_n, \sigma_n)],\\ \qquad \sigma_{n+1})\end{array}$$

  les contraintes impliquées par σ sont $X = pat_i$ et $\neg(X = pat_j)$. Deux cas se présentent, soit $i = j$ soit $i \neq j$.

Si $i = j$ alors σ implique $X = pat_i$ et $\neg(X = pat_i)$ et donc $\sigma \equiv \texttt{fail}$. Par le raisonnement analogue au cas précédent on conclut que les deux systèmes de contraintes obtenus sont équivalents.
Si $i \neq j$ alors la paire critique est joignable par application de la règle 1 sur le deuxième terme de la paire. Car $X = pat_i$ est impliqué par σ par hypothèse.

2 2 : la paire critique est :

$$\left(\begin{array}{l}\texttt{match}(X, [\texttt{pattern}(X = pat_1, \sigma_1),\\ \quad \ldots,\\ \quad \texttt{pattern}(X = pat_{i-1}, \sigma_{i-1}),\\ \quad \ldots,\\ \quad \texttt{pattern}(X = pat_{i+1}, \sigma_{i+1}),\\ \quad \ldots,\\ \quad \texttt{pattern}(X = pat_n, \sigma_n)],\\ \quad \sigma_{n+1})\end{array}\right., \quad \left(\begin{array}{l}\texttt{match}(X, [\texttt{pattern}(X = pat_1, \sigma_1),\\ \quad \ldots,\\ \quad \texttt{pattern}(X = pat_{j-1}, \sigma_{j-1}),\\ \quad \ldots,\\ \quad \texttt{pattern}(X = pat_{j+1}, \sigma_{j+1}),\\ \quad \ldots,\\ \quad \texttt{pattern}(X = pat_n, \sigma_n)],\\ \quad \sigma_{n+1})\end{array}\right)$$

avec $i \neq j$ et les contraintes impliquées par σ sont $\neg(X = pat_i)$ et $\neg(X = pat_j)$. On résout la paire en appliquant la règle 1 avec $X = pat_j$ sur le premier élément de la paire et en appliquant la règle 1 avec $X = pat_i$ sur le deuxième élément de la paire.

□

8.3 Équivalence entre les deux sémantiques de ite et de match

Pour l'instant nous nous sommes concentrés sur la traduction des programmes FMON. Nous avons défini un prédicat qui vérifie si une affectation totale est une solution d'un système de contraintes qui contient les contraintes `ite` et `match`. À cette occasion, nous avons muni notre prédicat de règles spécifiques à ces contraintes. D'autre part, nous avons donné une sémantique sous la forme d'un ensemble de règles à base de contraintes gardées.

Dans les deux sections qui suivent, nous montrons l'équivalence des deux sémantiques données à `ite` puis à `match`. Le schéma de la preuve est le suivant. Nous montrons d'abord que ces méta-contraintes sont équivalentes à une forme spécifique de la contrainte de cardinalité `card` qui exprime une disjonction exclusive. Ensuite, à partir de cette disjonction, nous montrons que la sémantique donnée aux méta-contraintes dans la relation $\vdash_S$ est équivalente à cette disjonction.

Dans la suite, afin de ne pas alourdir le propos, nous détaillons les preuves d'équivalences uniquement pour la méta-contrainte `ite`. Les preuves pour la contrainte `match` sont similaires.

8.3.1 Équivalence pour ite

Avant de montrer l'équivalence entre les sémantiques, nous remarquons que la structure conditionnelle exprime une forme de la contrainte `card`.

Proposition 8.3.1. *Soient une variable X et deux stores de contraintes σ_1 et σ_2. Les deux contraintes suivantes sont équivalentes, c'est-à-dire, admettent les mêmes solutions :*

$$\texttt{ite}(X, \sigma_1, \sigma_2) \tag{8.1}$$
$$\texttt{card}(1, 1, [X = \texttt{true} \wedge \sigma_1, X = \texttt{false} \wedge \sigma_2]) \tag{8.2}$$

Démonstration. Pour le prouver, il suffit de prendre les gardes de chacune d'elles et de vérifier que si la garde est vérifiée alors les déductions des deux méta-contraintes sont les mêmes.

On commence par prendre les gardes de 8.1.1

$X = \texttt{true}$**.** Alors la contrainte $X = \texttt{false} \wedge \sigma_2$ ne peut pas être vérifiée à cause de la valeur de X. On a donc $\neg(X = \texttt{false} \wedge \sigma_2)$. De 8.1 et par définition de `card` on obtient $\texttt{card}(1, 1, [X = \texttt{true} \wedge \sigma_1])$ et donc la contrainte $X = \texttt{true} \wedge \sigma_1$ qui se déduit aussi de 8.1.

$X = \texttt{false}$**.** Par un raisonnement analogue on retrouve l'équivalence des deux méta-contraintes.

$\neg(X = \texttt{true} \wedge \sigma_1)$**.** Par définition de `card` on obtient $\texttt{card}(1, 1, [X = \texttt{false} \wedge \sigma_2])$ qui se réécrit en $X = \texttt{false} \wedge \sigma_2$ ce qui se retrouve par la contrainte 8.1 avec la troisième contrainte gardée.

$\neg(X = \texttt{false} \wedge \sigma_2)$**.** Alors par définition de `card` 8.2 se réécrit en $\texttt{card}(1, 1, [X = \texttt{true} \wedge \sigma_1])$ qui se réécrit en $S = \texttt{true} \wedge \sigma_1$. Ce qui est le résultat de 8.1 par la dernière contrainte gardée.

On considère maintenant les contraintes gardées qui définissent `card`.

$(1 \leq 0) \wedge (2 \leq 1)$**.** Ce cas ne peut pas arriver.

$(1 \leq 1) \wedge (1 = 2)$**.** Ce cas ne peut pas arriver.

$X = \texttt{true} \wedge \sigma_1$**.** Dans ce cas, la contrainte 8.2 donne $\neg(X = \texttt{false} \wedge \sigma_2)$. On retrouve les contraintes en question par 8.1 on appliquant la 3$^{\text{e}}$ contrainte gardée de `ite`.
Par un raisonnement analogue on retrouve un résultat analogue si on a $X = \texttt{false} \wedge \sigma_2$

$\neg(X = \texttt{true} \wedge \sigma_1)$**.** De 8.1 on trouve $X = \texttt{false} \wedge \sigma_2$ et les deux formules se retrouvent avec la 4$^{\text{e}}$contrainte gardée de 8.1.
Lorsqu'on a $\neg(X = \texttt{false} \wedge \sigma_2)$, le raisonnement est similaire.

□

Ainsi, par l'intermédiaire de la contrainte de cardinalité, on a montré que la contrainte `ite` exprime qu'un seul des cas $x = \texttt{true} \wedge \sigma_1$ et $X = \texttt{false} \wedge \sigma_2$ est possible. On a donc le corollaire suivant au théorème 8.3.1.

Proposition 8.3.2. *Soient une variable X et deux ensembles de contraintes σ_1 et σ_2. La contrainte $\texttt{ite}(X, \sigma_1, \sigma_2)$ est équivalente à $(X = \texttt{true} \wedge \sigma_1) \vee (X = \texttt{false} \wedge \sigma_2)$ où $\vee$ est la disjonction logique.*

Nous remarquons que les prémisses de cette disjonction sont exclusives l'une de l'autre car elles imposent des valeurs différentes pour X.

Pour prouver l'équivalence entre les deux sémantiques, il suffit maintenant de remarquer que les deux règles ITE_TRUE et ITE_FALSE sont équivalentes à cette disjonction. C'est ce qu'expriment les deux théorèmes suivants.

Proposition 8.3.3. *Soient une variable X, deux stores de contraintes σ_1 et σ_2 et une affectation totale $\mathcal{A}$. Si on a un des deux cas suivants :*

$$\mathcal{A}; \mathcal{E}_{\texttt{cl}} \vdash_S X = \texttt{true}, \sigma_1 \longmapsto \mathcal{A}' \tag{8.3}$$

$$\mathcal{A}; \mathcal{E}_{\texttt{cl}} \vdash_S X = \texttt{false}, \sigma_2 \longmapsto \mathcal{A}' \tag{8.4}$$

Alors nous avons :

$$\mathcal{A}; \mathcal{E}_{\texttt{cl}} \vdash_S \texttt{ite}(X, \sigma_1, \sigma_2) \longmapsto \mathcal{A}'$$

Démonstration. La preuve se fait en suivant une analyse de cas. On commence par prendre en hypothèse soit 8.3 soit 8.4 et on montre le but avec les déductions qu'on peut obtenir dans chacun des cas.

Si l'hypothèse 8.3 est vérifiée alors on a une affectation $\mathcal{A}_1$ telle que

$$\mathcal{A}; \mathcal{E}_{\texttt{cl}} \vdash_S X = \texttt{true} \longmapsto \mathcal{A}_1 \tag{8.5}$$
$$\mathcal{A}_1; \mathcal{E}_{\texttt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}' \tag{8.6}$$

On a donc $\mathcal{A}_1(X) = \texttt{true}$ et comme $\mathcal{A}$ est totale on a aussi $\mathcal{A}(X) = \texttt{true}$ et donc la règle ITE_TRUE permet de conclure.

Pour le deuxième cas, le raisonnement est similaire. □

Proposition 8.3.4. *Soient une variable X et deux stores de contraintes σ_1 et σ_2 et une affectation totale $\mathcal{A}$. Si on a :*

$$\mathcal{A}; \mathcal{E}_{\texttt{cl}} \vdash_S \texttt{ite}(X, \sigma_1, \sigma_2) \longmapsto \mathcal{A}' \tag{8.7}$$

Nous avons l'un des deux cas suivants :

$$\mathcal{A}; \mathcal{E}_{\texttt{cl}} \vdash_S X = \texttt{true}, \sigma_1 \longmapsto \mathcal{A}' \tag{8.8}$$
$$\mathcal{A}; \mathcal{E}_{\texttt{cl}} \vdash_S X = \texttt{false}, \sigma_2 \longmapsto \mathcal{A}' \tag{8.9}$$

Démonstration. La preuve se fait par analyse de cas de la dernière règle appliquée pour montrer l'hypothèse 8.7.

Si l'hypothèse 8.7 a été obtenue par la règle ITE_TRUE alors :

$$\mathcal{A}(X) = \texttt{true} \tag{8.10}$$
$$\mathcal{A}; \mathcal{E}_{\texttt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}' \tag{8.11}$$

Ce qui suffit pour montrer le but 8.8 à l'aide des règles CONJONCT1 et EQFDI.

Pour le deuxième cas qui montre 8.7, on effectue un raisonnement similaire avec la règle ITE_FALSE. □

8.3.2 Équivalence pour `match`

La démonstration de l'équivalence entre les deux sémantiques suit le même cheminement que pour `ite`. On commence par montrer que `match` est équivalent à une forme de `card`.

Proposition 8.3.5. *Soient une variable X, un ensemble de motifs pat_i qui ne se recouvrent pas, un ensemble de stores de contraintes σ_i pour $i \in [\![1, n]\!]$ et un store σ. Les deux contraintes suivantes sont équivalentes :*

$$\begin{aligned} &\texttt{match}(X, [\ \texttt{pattern}(X = pat_1, \sigma_1), \\ &\qquad \ldots, \\ &\qquad \texttt{pattern}(X = pat_n, \sigma_n)], \sigma) \end{aligned} \tag{8.12}$$

$$\begin{aligned} &\texttt{card}(1, 1, [\ X = pat_1 \wedge \sigma_1, \\ &\qquad \ldots, \\ &\qquad X = pat_n \wedge \sigma_n, \\ &\qquad X \neq pat_1 \wedge \ldots \wedge X \neq pat_n \wedge \sigma]) \end{aligned} \tag{8.13}$$

Démonstration. Nous le prouvons par induction sur n le nombre de motifs.

n = 0. Dans ce cas, les contraintes 8.12 8.13 se réécrivent toutes les deux en σ.

n = 1. Si $\sigma = \texttt{fail}$, les deux contraintes se réécrivent en σ_1. Sinon, on fait une analyse par cas sur les règles de réécriture de chacune des deux contraintes et on constate qu'elles arrivent aux mêmes contraintes.

n > 1. On développe deux cas, un cas pour les règles de sémantique de `match` et un cas pour les règles de sémantiques de `card`. Soit on constate qu'elles se réécrivent toutes les deux en $X = pat_i \wedge \sigma_i$ pour $i \in [\![1, n]\!]$. Soit elles se réécrivent en une version d'elles-mêmes avec $n - 1$ motifs et l'hypothèse d'induction permet de conclure.

□

Nous avons donc le corollaire à ce théorème qui établit que `match` est une disjonction exclusive.

Proposition 8.3.6. *Soient une variable X, un ensemble de motifs pat_i qui ne se recouvrent pas, un ensemble de stores de contraintes σ_i pour $i \in [\![1, n]\!]$ et un store σ. Les deux expressions suivantes sont équivalentes :*

$$\begin{aligned} &\texttt{match}(X, [\ \texttt{pattern}(X = pat_1, \sigma_1), \\ &\qquad\qquad \ldots, \\ &\qquad\qquad \texttt{pattern}(X = pat_n, \sigma_n)], \sigma) \end{aligned} \tag{8.14}$$

$$\begin{aligned} &(X = pat_1 \wedge \sigma_1) \vee \\ &\ldots \\ &(X = pat_n \wedge \sigma_n) \vee \\ &(X \neq pat_1 \wedge \ldots \wedge X \neq pat_n \wedge \sigma) \end{aligned} \tag{8.15}$$

La disjonction 8.15 est exclusive car les motifs pat ne se recouvrent pas par hypothèse.

Nous pouvons maintenant prouver l'équivalence des deux sémantiques définies pour `match`.

Proposition 8.3.7. *Soient une variable X, deux stores de contraintes σ_1 et σ_2 et une affectation $\mathcal{A}$. Si on a un des cas suivants :*

$$\begin{aligned} &\mathcal{A}; \mathcal{E}_{\texttt{cl}} \vdash_S X = pat_1 \wedge \sigma_1 \longmapsto \mathcal{A}' \\ &\qquad\qquad\qquad\qquad \vdots \\ &\mathcal{A}; \mathcal{E}_{\texttt{cl}} \vdash_S X = pat_n \wedge \sigma_2 \longmapsto \mathcal{A}' \\ &\mathcal{A}; \mathcal{E}_{\texttt{cl}} \vdash_S X \neq pat_1 \wedge \ldots \wedge X\, pat_n \wedge \sigma \longmapsto \mathcal{A}' \end{aligned}$$

Alors nous avons :

$$\begin{aligned} &\texttt{match}(X, [\ \texttt{pattern}(X = pat_1, \sigma_1), \\ \mathcal{A}; \mathcal{E}_{\texttt{cl}} \vdash_S &\qquad\qquad \ldots, \qquad\qquad \longmapsto \mathcal{A}' \\ &\qquad\qquad \texttt{pattern}(X = pat_n, \sigma_n)], \sigma) \end{aligned}$$

Démonstration. La preuve est similaire à celle du théorème 8.3.3. □

Proposition 8.3.8. *Soient une variable X, deux stores de contraintes σ_1 et σ_2 et une affectation $\mathcal{A}$. Si on a :*

$$\begin{aligned} &\texttt{match}(X, [\ \texttt{pattern}(X = pat_1, \sigma_1), \\ \mathcal{A}; \mathcal{E}_{\texttt{cl}} \vdash_S &\qquad\qquad \ldots, \qquad\qquad \longmapsto \mathcal{A}' \\ &\qquad\qquad \texttt{pattern}(X = pat_n, \sigma_n)], \sigma) \end{aligned}$$

Nous avons l'un des cas suivants :

$$\begin{array}{l}\mathcal{A};\mathcal{E}_{\texttt{cl}} \vdash_S X = pat_1 \wedge \sigma_1 \longmapsto \mathcal{A}' \\ \vdots \\ \mathcal{A};\mathcal{E}_{\texttt{cl}} \vdash_S X = pat_n \wedge \sigma_2 \longmapsto \mathcal{A}' \\ \mathcal{A};\mathcal{E}_{\texttt{cl}} \vdash_S X \neq pat_1 \wedge \ldots \wedge X pat_n \wedge \sigma \longmapsto \mathcal{A}'\end{array}$$

Démonstration. La preuve est similaire à celle du théorème 8.3.4. □

8.4 Synthèse

Nous avons donc introduit des méta-contraintes pour modéliser les structures conditionnelles et les filtrages par motif dans les contraintes. Nous avons donné deux sémantiques à chacune de ces méta-contraintes, la première définie par l'intermédiaire du prédicat de test de solution, qui vérifie si une affectation satisfait les contraintes et la deuxième définie avec des contraintes gardées, qui donne la méthode de résolution des deux méta-contraintes. Nous avons montré que ces deux sémantiques sont équivalentes.

Dans le chapitre suivant, nous présentons l'outil qui a été développé dans le cadre ce travail. Nous évaluons aussi cet outil sur des exemples. Le but de ces évaluations est de montrer la pertinence de la sémantique de nos méta-contraintes ainsi que les apports par rapport à d'autres implantations possibles pour `ite` et `match`.

Chapitre 9

L'outil FoCalTest

Nous présentons ici l'outil qui a été développé à l'issue des travaux présentés dans les chapitres précédents. FoCalTest est écrit en OCaml. Les contraintes obtenues après la transformation présentée dans le chapitre 7 sont exprimées en Sicstus Prolog. De même, les méta-contraintes `ite` et `match` et la résolution des contraintes générées à partir d'une précondition sont implantées en Sicstus Prolog.

Dans la première partie de ce chapitre nous présenterons l'architecture générale de FoCalTest et son fonctionnement. Nous détaillerons ensuite comment FoCalTest pose son harnais dans le programme sous test. Enfin, nous montrerons des résultats expérimentaux.

9.1 Architecture générale de FoCalTest

La figure 9.1 présente l'architecture générale de FoCalTest. Principalement, FoCalTest est composé de quatre modules. FoCalTest a d'abord été intégré à *focc*, la première implantation de FoCal et plus récemment, il a été porté sur la dernière implantation de FoCal nommée *Focalize*. Il prend en entrée une liste de noms de propriétés à tester et le nom de l'espèce dans laquelle se trouvent ces propriétés sous la forme d'un *contexte de test* (défini en 9.3.1).

1. le module *décomposition* se charge de prendre une propriété et de la décomposer en un ensemble de propriétés élémentaires. Il s'agit de l'application du système de réécriture sur la propriété sous test ;
2. le module *Analyse de dépendance/extraction* se charge de prendre les formes normales des propriétés testées et le programme sous test. Il effectue l'analyse de dépendance des fonctions présentes dans la précondition et calcule l'ensemble des fonctions qui devront être extraites. Ensuite, ce module traduit les fonctions vers le langage FMON dans une forme normalisée ;
3. le module *Traduction* prend l'ensemble des fonctions FMON produites par l'extraction et les formes normales des propriétés. D'une part, il convertit les fonctions FMON en un environnement de clause, chaque fonction donnant lieu à une clause. D'autre part, il convertit les préconditions des formes normales en un ensemble de contraintes en ajoutant les directives propres à Prolog pour lancer la procédure de résolution des contraintes ;
4. le module *harnais* prend l'ensemble des formes normales et détermine le harnais de test. Il s'agit d'un ensemble d'espèces et de collections qui héritent de l'espèce dans laquelle se trouve la propriété testée et ajoutent les fonctions nécessaires au test (voir 9.3). Ce module génère un fichier source FoCal qui contient des directives pour exécuter le programme Prolog et récupérer les résultats ;

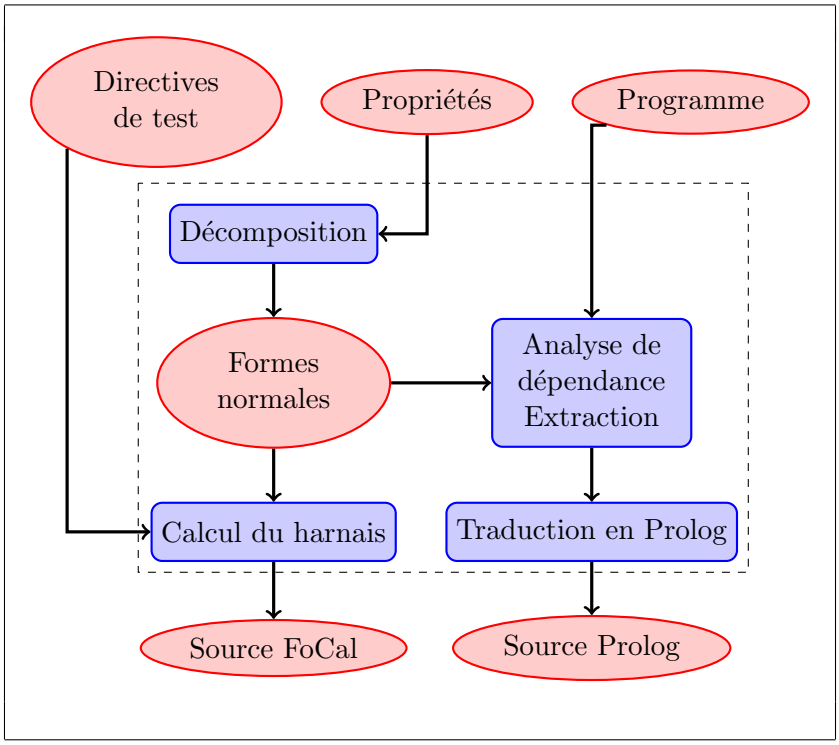

FIGURE 9.1 – Architecture de FoCalTest

L'exécutable obtenu après la compilation du fichier FoCal généré par FoCalTest, lance la procédure de test des propriétés. Il génère les jeux de test en utilisant les contraintes du fichier Prolog, les soumet (évalue les conclusions) et génère un rapport de test au format XML.

9.2 Implantation

Dans la présentation de notre méthodologie nous avons développé un formalisme sur les contraintes qui se veut indépendant de leur implantation dans FoCalTest. Nous détaillons ici les choix d'implantation que nous avons fait.

FoCalTest profite de l'API de *Focalize* pour obtenir les programmes FoCal testés sous la forme d'un arbre de syntaxe abstraite. Il définit sa propre structure pour représenter les programmes FMON et les contraintes. Les deux étapes de transformation de programmes (FoCal → FMON et FMON → contraintes) sont effectuées en interne. FoCalTest génère deux fichiers en sortie.

- le premier fichier est un programme FoCal qui contient le harnais de test. Un ensemble d'espèces et de collections qui contiennent trois sortes de fonctions :
  - les fonctions qui permettent d'obtenir des éléments du type support de l'espèce considérée en passant outre l'abstraction de la représentation des données ;
  - les fonctions qui testent la valeur de vérité de la précondition et de la conclusion ;
  - les fonctions qui se chargent d'effectuer la procédure de test et d'écrire le rapport de test.

- le deuxième fichier est un programme Sicstus Prolog contenant l'environnement de contraintes issu de la traduction du programme FoCal vers un ensemble de contraintes.

Pour l'implantation des contraintes, nous avons choisi de représenter les clauses telles que présentées en 7.3.3 par des prédicats Prolog car leur fonctionnement est similaire à la sémantique que nous avons donnée à nos clauses. L'implantation des contraintes repose en très grande partie sur CLP(FD), la bibliothèque des contraintes sur les domaines finis. Nous avons utilisé les interfaces de CLP(FD) pour implanter des contraintes utilisateurs pour `ite` et `match`. Ainsi, ces deux dernières sont manipulées directement par le solveur de contraintes de CLP(FD) et bénéficient d'un traitement identique aux contraintes prédéfinies par CLP(FD). Les contraintes `ite` et `match` sont réveillées dans le solveur sur modification du domaine des variables sur lesquelles elles portent.

Nous avons défini le domaine des variables algébriques en utilisant des variables Prolog attribuées. Nous utilisons deux attributs, un attribut pour spécifier le type de la variable et un attribut qui spécifie l'ensemble des constructeurs possibles. Nous avons défini trois contraintes pour les variables algébriques, l'égalité entre deux variables, l'inégalité et le filtrage de tête. Afin de vérifier le respect du bon typage des variables algébriques nous exportons l'environnement de type de FoCal dans un environnement accessible dans Prolog. Chacune des trois contraintes utilise cet environnement. Les variables algébriques prennent leurs valeurs dans le domaine d'Herbrand en respect du type de la variable.

Pour éviter que la résolution des contraintes ne diverge et n'aboutisse pas, nous avons intégré à FoCalTest des limitations dans la recherche de solutions. D'une part, nous avons fixé un seuil dans le nombre maximal de contraintes `ite` et `match` réveillées simultanément. Le traitement des méta-contraintes cherche si une contrainte donnée est impliquée par l'ensemble des contraintes courantes. Cette recherche peut réveiller une autre méta-contrainte que va relancer un autre test d'implication. Réveiller trop de méta-contraintes simultanément nuit énormément aux performances du solveur de contraintes et peut dans certain cas ne pas terminer. Il faut remarquer que cette limitation existe aussi bien pour éviter que la résolution diverge mais aussi pour éviter que le test d'implication nuise au performance. La deuxième limitation que nous avons fixé concerne les variables algébriques. Nous avons fixé une profondeur maximum pour les variables algébrique. Cette limitation est obligatoire dans la mesure où l'ensemble des valeurs possibles pour des variables algébriques est le plus souvent infini. Comme la résolution de contraintes peut aboutir à une énumération exhaustive du domaine des variables, cette limitation parait évidente.

Lors de la génération du rapport de test, la précondition est évaluée à l'aide des valeurs trouvées pour le solveur de contraintes et le résultat est mis dans le rapport. Ainsi, si le solveur de contraintes présente un défaut et trouve une affectation qui n'est pas une solution du système de contrainte, le rapport de test garde une trace de cette erreur.

9.3 Pose d'un Harnais sur une espèce

Au sein de l'outil FoCalTest, l'espèce à tester est donnée sous la forme d'un contexte de test. Intuitivement, un contexte de test indique dans quelle espèce se trouve la propriété testée ainsi que l'ensemble des espèces que l'on souhaite utiliser pour instancier les paramètres éventuels de l'espèce indiquée. Rappelons que, les paramètres d'une espèce sont des collections. Toutefois, le harnais de test a besoin d'ajouter dans chaque paramètre les fonctions utiles au test. Les collections étant figées par définition, il n'est pas possible d'ajouter les fonctions du harnais à partir d'une collection. De plus, l'abstraction du type support interdit la manipulation et la création d'une valeur de collection à l'extérieur de la dite collection en utilisant directement les constructeurs de valeurs. Nous avons besoin de connaître le type support pour créer les

générateurs de valeurs ou bien initialiser les domaines des variables de contrainte. L'abstraction nous en empêche. C'est pour cela que les contextes de test sont définis à partir de noms d'espèces et non de collections.

Nous présentons donc, dans la suite, la définition d'un contexte de test ainsi que les définitions qui permettent de vérifier qu'un contexte de test est bien formé. Nous donnerons, ensuite, les règles utilisées pour poser le harnais de test sur l'ensemble des espèces utilisées dans le contexte de test.

9.3.1 Contexte de test

Définition

On appelle *contexte de test* la donnée de l'espèce à tester et de ses paramètres effectifs. Nous définissons d'abord le langage des contextes de test. La syntaxe concrète de déclaration de l'espèce à tester est la suivante :

$$
\begin{array}{lcll}
c & ::= & \mathfrak{c} \text{ impl } s \text{ in } c & \text{déclaration de collection} \\
 & | & s & \text{espèce sous test} \\
 & & & \\
s & ::= & \mathsf{S}(cl) & \text{espèce appliquée} \\
 & | & \mathsf{S} & \text{espèce} \\
 & & & \\
cl & ::= & \mathfrak{c}, cl & \text{liste de collections} \\
 & | & \mathfrak{c} & \text{collection simple}
\end{array}
$$

Un contexte de test est donc la donnée d'une suite de déclarations de collection de la forme : $\mathfrak{c}$ impl ... in ... Les collections sont définies à partir d'une espèce (mot-clef impl) éventuellement appliquée à des paramètres. Comme dans FoCal, les paramètres des espèces sont des collections.

Une fois l'ensemble des collections défini, on spécifie l'espèce à tester. Il s'agit d'une espèce qui peut être appliquée à des collections.

Intuitivement, la liste de déclarations de collections qui précède l'énoncé de l'espèce sous test permet de définir les collections qui seront utilisées en tant que paramètre de l'espèce testée. Ces collections ne doivent pas exister préalablement dans le programme. Ce sont des nouvelles collections qui sont créées par le harnais de test et qui vont correspondre aux espèces spécifiées dans lesquelles nous avons ajouté les fonctions du harnais.

Exemple 9.3.1. Soient l'interface Setoid, une espèce complète Integer compatible avec l'interface Setoid, une espèce Set qui prend en paramètre une collection d'interface Setoid et une espèce Treillis. Un contexte qui permet de tester l'espèce Treillis(Set(Integer)) se déclare comme ceci :

$$
\begin{array}{l}
\mathfrak{c}_1 \text{ impl Integer in} \\
\mathfrak{c}_2 \text{ impl Set}(\mathfrak{c}_1) \text{ in} \\
\text{Treillis}(\mathfrak{c}_2)
\end{array}
$$

À partir de l'exemple précédent, on remarque qu'un contexte de test est une manière de spécifier un arbre d'héritage.

Paramètre de test et dépendances

Nous avons besoin d'identifier l'ensemble des collections définies dans un contexte de test. Nous appelons les collections définies dans un contexte les *paramètres de test*.

Définition 9.3.2 Paramètre de test**.**
Nous appelons, et notons $\boldsymbol{PC}(\mathsf{c})$*, paramètres de test d'un contexte* c*, l'ensemble des collections obtenues par application de la définition inductive suivante :*

$$\begin{array}{l}\boldsymbol{PC}(\mathsf{S}) = \boldsymbol{PC}(\mathsf{S}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n)) = \emptyset \\ \boldsymbol{PC}(\mathfrak{c} \text{ impl } \mathsf{S} \text{ in e}) = \{\mathfrak{c}\} \\ \boldsymbol{PC}(\mathfrak{c} \text{ impl } \mathsf{S}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n) \text{ in e}) = \{\mathfrak{c}\} \cup \boldsymbol{PC}(e)\end{array}$$

Exemple 9.3.3. Les paramètres de test de l'exemple 9.3.1 sont $\{\mathfrak{c}_1, \mathfrak{c}_2\}$.

À partir de la définition des paramètres de test d'un contexte, nous introduisons la notion de dépendance d'un paramètre de contexte. Cette notion est importante car elle détermine l'ensemble des collections qui doivent être définies pour permettre de créer la collection qui va implanter le paramètre de test.

Définition 9.3.4 Dépendances d'un paramètre de test**.**
Soit un contexte c *et une collection* $\mathfrak{c} \in \boldsymbol{PC}(\mathsf{c})$*.* $\boldsymbol{Dep}_{\mathfrak{c}}(\mathsf{c})$ *est l'ensemble des dépendances de* $\mathfrak{c}$ *sur le contexte* c*.*

$$\begin{array}{lll}\boldsymbol{Dep}_{\mathfrak{c}}(\mathsf{S}) = \boldsymbol{Dep}_{\mathfrak{c}}(\mathsf{S}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n)) = \boldsymbol{Dep}_{\mathfrak{c}}(\mathfrak{c}' \text{ impl } \mathsf{S} \text{ in e}) = \emptyset & & \\ \boldsymbol{Dep}_{\mathfrak{c}}(\mathfrak{c}' \text{ impl } \mathsf{S}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n) \text{ in e}) = \{\mathfrak{c}_1, \ldots, \mathfrak{c}_n\} & \textit{si} & \mathfrak{c}' = \mathfrak{c} \\ \boldsymbol{Dep}_{\mathfrak{c}}(\mathfrak{c}' \text{ impl } \mathsf{S}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n) \text{ in e}) = \emptyset & \textit{si} & \mathfrak{c}' \neq \mathfrak{c}\end{array}$$

Intuitivement, les dépendances d'un paramètre de test $\mathfrak{c}$ sont l'ensemble des collections paramètres de l'espèce à partir de laquelle est définie $\mathfrak{c}$.

Exemple 9.3.5. L'ensemble des dépendances de $\mathfrak{c}_2$ dans l'exemple 9.3.1 est $\{\mathfrak{c}_1\}$. L'ensemble des dépendances de $\mathfrak{c}_1$ est $\emptyset$.

Les dépendances telles qu'elles sont définies par la fonction $\mathbf{Dep}_{\mathfrak{c}}$ ne sont pas suffisantes pour exprimer l'ensemble des collections qui doivent être définies pour créer la collection. La fonction retourne les dépendances directes du paramètre de test. Nous avons besoin d'obtenir l'ensemble des dépendances directes et indirectes pour la suite.

Définition 9.3.6 Dépendance totale des paramètres de test**.**
On définit maintenant $\boldsymbol{Dep}^*_{\mathfrak{c}}(\mathsf{c})$ *comme la fermeture transitive de* $\boldsymbol{Dep}_{\mathfrak{c}}(\mathsf{c})$*.*

Définition 9.3.7 Ordre sur les dépendances**.**
Soit un contexte de test c*.* c *définit un ordre noté* $\prec_{\mathsf{c}}$ *sur les paramètres de test :*

$$\mathfrak{c}_1 \prec_{\mathsf{c}} \mathfrak{c}_2 \Leftrightarrow \mathfrak{c}_1 \in \boldsymbol{Dep}^*_{\mathfrak{c}_2}(\mathsf{c})$$

Validité d'un contexte

Nous allons définir l'ensemble des contextes valides. Intuitivement, un contexte valide est un contexte qui ne fait référence à aucune collection de l'environnement. Nous avons besoin de cette hypothèse car nous souhaitons étendre la définition de chacun des composants utilisés dans le contexte à un harnais de test, par ajout de fonctions. Une collection est un composant figé qui ne peut pas être étendu par héritage. Si un contexte de test utilise une collection de

l'environnement FoCal, cela signifie qu'on ne pourra pas y ajouter les fonctions du harnais de test. La notion de contexte valide permet aussi de définir la portée d'un paramètre de test. La portée d'un paramètre de test est similaire à la portée d'une variable locale définie par la construction **let** ... **in** des langages ML.

Définition 9.3.8 Collections libres d'un contexte de test**.**
On définit l'ensemble des collections libres d'un contexte de test par :

$$\begin{array}{l}\boldsymbol{FC}(\mathsf{S}) = \emptyset \\ \boldsymbol{FC}(\mathsf{S}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n)) = \{\mathfrak{c}_1, \ldots, \mathfrak{c}_n\} \\ \boldsymbol{FC}(\mathfrak{c} \text{ impl } \mathsf{S} \text{ in e}) = \boldsymbol{FC}(e) \backslash \{\mathfrak{c}\} \\ \boldsymbol{FC}(\mathfrak{c} \text{ impl } \mathsf{S}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n) \text{ in e}) = (\boldsymbol{FC}(e) \backslash \mathfrak{c}) \cup \{\mathfrak{c}_1, \ldots, \mathfrak{c}_n\}\end{array}$$

Définition 9.3.9 Contexte de test valide**.**
Soit un contexte de test c. c *est* valide *si et seulement si* $\boldsymbol{FC}(\mathsf{c}) = \emptyset$.

Exemple 9.3.10. Le contexte de l'exemple 9.3.1 est valide.

Comme déjà dit, les contextes valides sont ceux qui vont permettre de poser un harnais sur chacun des composants impliqués. Dans un contexte valide, les seuls éléments qui doivent être prédéfinis sont les espèces qui servent à créer les paramètres du contexte et l'espèce sous test. Dans la suite, nous supposerons que les contextes de test utilisés sont valides.

Il faut toutefois vérifier que, dans l'environnement dans lequel se place le contexte de test, les espèces utilisées du contexte de test ne dépendent pas de collections. Par exemple, si nous avons une espèce S paramétrée par une collection $\mathfrak{c}_1$, nous pouvons créer une nouvelle espèce S' qui n'attend pas de paramètres et qui hérite de la première appliquée à une collection $\mathfrak{c}_2$ (présente dans l'environnement). Dans ce cas, S' utilise des entités de type $\mathfrak{c}_2$. L'abstraction du type support de $\mathfrak{c}_2$ pose problème pour poser le harnais de test (qui consiste à pouvoir créer une valeur de $\mathfrak{c}_2$ sans passer par les fonctions de $\mathfrak{c}_2$).

Nous présentons donc la définition suivante qui permet d'obtenir les collections dont dépend une espèce.

Définition 9.3.11 Collections libres d'une interface**.**
Soit une interface I dont les types des fonctions sont τ_1, ..., τ_n. L'ensemble des collections libres de I est donné par $\boldsymbol{FC}_I(I)$:

$$\begin{array}{l}\boldsymbol{FC}_I(\exists \mathit{rep\text{-}abs}.\{m_1 : \tau_1, \ldots, m : \tau_n\}[\mathbf{self} \leftarrow \mathit{rep\text{-}abs}]) = \bigcup_{i \in 1..n} \boldsymbol{FC}_m(\tau_i[\mathbf{self} \leftarrow \mathit{rep\text{-}abs}]) \\ \boldsymbol{FC}_I(\exists \mathfrak{c} : I^{\mathfrak{c}}.I') = \boldsymbol{FC}_I(I')/\{\mathfrak{c}\}\end{array}$$

Où $\boldsymbol{FC}_m(\tau)$ retourne l'ensemble des symboles de collection qui apparaissent dans τ.

Définition 9.3.12.
Une espèce S *d'interface I est close si et seulement si $\boldsymbol{FC}_I(I) = \emptyset$*

Contexte de test bien formé

On définit maintenant la notion de contexte de test bien formé par rapport à un environnement FoCal. On rappelle qu'un environnement FoCal est la donnée d'un ensemble de collections ou d'espèces.

Définition 9.3.13 Espèces de test**.**
Nous appelons espèces de test d'un contexte c *l'ensemble* $\boldsymbol{SC}(\mathsf{c})$ *défini par* c *:*

$$\begin{aligned}
&\boldsymbol{SC}(\mathsf{S}) = \{S\}\\
&\boldsymbol{SC}(\mathsf{S}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n)) = \{S\}\\
&\boldsymbol{SC}(\mathfrak{c} \text{ impl } \mathsf{S} \text{ in } \mathsf{c}) = \{S\} \cup \boldsymbol{SC}(\mathsf{c})\\
&\boldsymbol{SC}(\mathfrak{c} \text{ impl } \mathsf{S}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n) \text{ in } \mathsf{c}) = \{S\} \cup \boldsymbol{SC}(\mathsf{c})
\end{aligned}$$

À partir des espèces de test, nous pouvons maintenant définir les collections libres d'un contexte de test. Pour cela, nous allons nous servir de $\boldsymbol{FC}_I$.

Définition 9.3.14 Collections de test libres**.**
Soit un contexte de test c*. On définit l'ensemble des collections libres d'un contexte de test par l'ensemble* $\boldsymbol{FC}_{\mathsf{c}}(\mathsf{c}) = \{\mathfrak{c} \mid \exists \mathsf{S}\,.\ \mathfrak{c} \in \boldsymbol{FC}_I(I) \wedge \mathsf{S} \in \boldsymbol{SC}(\mathsf{c})\}$*.*

Définition 9.3.15 contexte de test clos**.**
Un contexte de test c *est clos si et seulement si* $\boldsymbol{FC}_{\mathsf{c}}(\mathsf{c}) = \emptyset$*.*

Définition 9.3.16 Contexte de test bien formé par rapport à un environnement**.**
Soit un environnement d'espèces $\mathcal{S}$*, un ensemble de collections* $\mathcal{C}$ *et un contexte de test* c*. Le contexte* c *est bien formé par rapport à l'environnement* $\mathcal{S};\mathcal{C}$ *si et seulement si le contexte est valide et clos dans l'environnement.*

9.3.2 Harnais de test

Pour la pose du harnais de test, le principe est de prendre chaque espèce de test du contexte de test et de rajouter par héritage l'ensemble des fonctions nécessaires au test.

Hypothèses sur le harnais

On suppose l'existence d'une fonction $\boldsymbol{Harness}_m : \mathcal{S} \to \mathcal{M}$ qui à partir de la définition d'une espèce crée l'ensemble des fonctions qui constitue le harnais de l'espèce. $\boldsymbol{Harness}_m(S)$ utilise la définition du type support pour créer les fonctions de générations de valeurs.

Pose du harnais

On part d'un contexte de test bien formé c. La pose du harnais de test se fait par induction sur c. Pour chaque collection de test de c, on crée une espèce qui ajoute dans l'espèce de test les fonctions retournées par $\boldsymbol{Harness}_m$ et la collection qui implante cette nouvelle collection.

La figure 9.2 illustre la pose du harnais dans une espèce S paramétrée par une collection $\mathfrak{c}$. La collection $\mathfrak{c}$ paramètre de S doit implanter S_1. Le harnais consiste à étendre par héritage

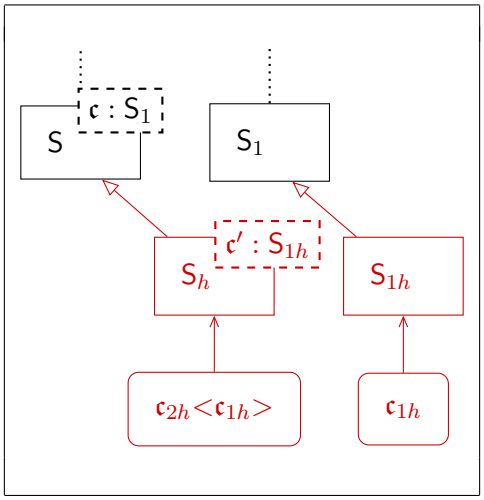

FIGURE 9.2 – Ajout du harnais dans une espèce

$$\frac{def_{\mathsf{s}} = \{\emptyset; \mathsf{S}; \emptyset; \boldsymbol{Harness}_m(\mathsf{S})\}}{\mathsf{S} \longmapsto def_{\mathsf{s}}} \text{ SPECHARNESS}$$

$$\frac{def_{\mathsf{s}} = \begin{array}{l}\{\mathfrak{c}_1 : I_1, \ldots \mathfrak{c}_n : I_n; \\ \mathsf{S}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n); \\ \emptyset; \\ \boldsymbol{Harness}_m(\mathsf{S})\}\end{array}}{\mathsf{S}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n) \longmapsto def_{\mathsf{s}}} \text{ PARAMHARNESS}$$

FIGURE 9.3 – Pose du harnais sur une espèce

l'espèce S_1. Cela nous donne l'espèce S_{1h}. Ensuite, nous étendons la définition de S par héritage dans l'espèce S_h. Les deux collections $\mathfrak{c}_{1h}$ et $\mathfrak{c}_{2h}$ (qui implantent respectivement S_{1h} et S_h) sont ensuite créées.

On commence par donner les règles qui permettent de poser le harnais sur une espèce. L'espèce doit être typée. Les règles de transformation sont données par des jugements de la forme :

$$\mathsf{S} \longmapsto def_{\mathsf{s}}$$

Elle signifie, « à partir de l'espèce S on crée la définition d'espèce def_{s} qui contient les fonctions du harnais ». La figure 9.3 présente les définitions des deux règles. La première règle pose un harnais sur une espèce non paramétrée et la deuxième pose le harnais dans le cas d'une espèce paramétrée. Les deux règles retournent la définition d'une nouvelle espèce qui hérite de l'espèce d'origine et définissent les fonctions retournées par $\boldsymbol{Harness}_m$.

Maintenant qu'on est capable de poser le harnais sur une espèce, on peut définir la pose du harnais sur un contexte de test. La pose d'un harnais de test se fait à l'aide de jugements de la forme :

$$\frac{\mathsf{S} \longmapsto def_{\mathsf{s}} \qquad \mathcal{C}, \mathfrak{c}; \mathcal{S} \vdash \mathsf{c} \longmapsto cmd \qquad \mathsf{S'} \text{ frais}}{\mathcal{C}; \mathcal{S} \vdash \mathfrak{c} \text{ impl } \mathsf{S} \text{ in } \mathsf{c} \rightsquigarrow \mathsf{S'} = def_{\mathsf{s}}; \mathfrak{c} \text{ impl } \mathsf{S'}; cmd} \textsc{ContextCollHarness}$$

$$\frac{\mathsf{S} \longmapsto def_{\mathsf{s}} \qquad \mathsf{S'} \text{ frais}}{\mathcal{C}; \mathcal{S} \vdash \mathsf{S} \rightsquigarrow \mathsf{S'} = def_{\mathsf{s}}; \mathfrak{c} \text{ impl } \mathsf{S'}} \textsc{ContextSpecHarness}$$

$$\frac{\mathsf{S}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n) \longmapsto def_{\mathsf{s}} \qquad \mathsf{S'} \text{ frais}}{\mathcal{C}; \mathcal{S} \vdash \mathsf{S}(\mathfrak{c}_1, \ldots, \mathfrak{c}_n) \rightsquigarrow \mathsf{S'} = def_{\mathsf{s}}; \mathfrak{c} \text{ impl } \mathsf{S'}} \textsc{ContextSpecParamHarness}$$

FIGURE 9.4 – Pose du harnais sur un contexte de test

$$\mathcal{C}; \mathcal{S} \vdash \mathsf{c} \rightsquigarrow cmd$$

Ces jugements se lisent : « dans l'environnement FoCal $\mathcal{C}$; $\mathcal{S}$, la pose du harnais sur le contexte de test c donne lieu aux instructions FoCal *cmd* ».

Les 3 règles sont données dans la figure 9.4.

La règle CONTEXTCOLLHARNESS correspond au cas où un paramètre de collection est défini dans le contexte de test. On commence par poser le harnais sur l'espèce S, cela donne lieu à une définition def_{s}. On pose ensuite le harnais sur le reste du contexte, ce qui donne lieu à une suite d'instructions *cmd*. Le harnais total est alors la suite d'instruction $\mathsf{S'} = def_{\mathsf{s}}, \mathfrak{c}$ impl $\mathsf{S'}$ *cmd* qui définit une nouvelle espèce $\mathsf{S'}$ de définition def_{s} puis crée la collection $\mathfrak{c}$ à partir de $\mathsf{S'}$ et contient ensuite les instructions *cmd*.
Les règles CONTEXTSPECHARNESS et CONTEXTSPECPARAMHARNESS sont similaires, il s'agit de poser le harnais sur l'espèce sous test puis de créer une espèce et une collection à partir de la définition obtenue.

Enfin, le programme obtenu par ces règles est augmenté des instructions nécessaires pour lancer la recherche de jeux de test. Le résultat est le fichier FoCal généré par FoCalTest.

9.4 Expérimentations et résultats

Nous présentons dans cette section les résultats expérimentaux qu'on a pu obtenir à partir de FoCalTest. Le but de ces expérimentations est double. Elles consistent à comparer les apports des contraintes d'une part par rapport à l'approche aléatoire, et d'autre part, il s'agit de montrer que l'implantation de contraintes spécifiques pour les constructions fonctionnelles `ite` et `match` à l'aide de méta-contraintes, et avec la sémantique décrite dans le chapitre 8, apporte un gain en terme de recherche de solution par rapport à une implantation « naïve » de ceux-ci.

9.4.1 Conditions d'expérimentation

Dans ce qui suit, nous avons demandé à FoCalTest de générer 10 jeux de test pour chaque propriété testée, et donc 10 valuations des variables quantifiées qui valident la précondition. Nous avons comparé 4 stratégies de génération de jeux de test. 3 d'entre-elles utilisent la résolution de contraintes pour obtenir les jeux de test. La dernière est la stratégie de génération aléatoire des valuations jusqu'à obtention de 10 jeux de test valides.

Pour chacune des propriétés, nous avons mesuré le temps pris par FoCalTest pour trouver les 10 jeux de test. Nous avons mesuré les temps de génération à l'aide de la fonction `Unix.time` et avons mesuré le temps entre le lancement de la génération des jeux de test et la fin de celle-ci.

Pour la recherche des jeux de test avec la stratégie aléatoire, nous avons en plus du temps de recherche, le nombre de valuations rejetées pour raison de préconditions non vérifiées. Afin d'assurer que l'expérimentation termine, nous avons fixé à 10000000 le nombre maximum de valuations successives qui ne forment pas un jeu de test. Lorsque ce seuil est atteint, la recherche aléatoire des jeux de test est interrompue. Ce seuil est réactivé à chaque propriété élémentaire.

Pour les stratégies de recherches de jeux de test par résolution de contraintes, le lancement de la recherche des jeux de test se fait en deux temps. Dans un premier temps, nous avons lancé Sicstus Prolog et chargé en mémoire l'environnement de recherche de solutions. Nous avons sauvegardé l'état-mémoire de Sicstus Prolog dans un fichier à l'aide de la primitive `save_program`. Nous avons, ensuite, relancé Sicstus Prolog, chargé le fichier qui contient l'état-mémoire et lancé la recherche de solutions. Nous avons mesuré le temps de recherche de la deuxième étape. Cela permet de mesurer le temps effectif de recherche de solutions, sans mesurer le chargement des différentes bibliothèques, et donc dans les mêmes conditions que pour la recherche de solution aléatoire.

Afin d'avoir des mesures de comparaison significative entre les différentes stratégies de recherche, nous avons retiré des jeux de test possibles les cas de test « dégénérés », c'est-à-dire, les jeux de test qui contiennent des valeurs évidentes. Pour cela, nous avons imposé la génération de valeurs d'une taille supérieure à une limite donnée lorsque c'était possible. Pour l'énumération, nous avons choisi de déterminer la structure des valeurs des types algébriques (longueur de la liste, forme de l'arbre) puis d'énumérer les valeurs entières.

Nous avons exécuté l'outil sur un ordinateur muni d'un processeur « 2.33 GHz Intel Core 2 Duo ». Les valeurs entières utilisées dans les exemples sont représentées par des entiers 16 bits.

9.4.2 Exemples traités

Nous avons évalué l'outil sur les 7 exemples suivants :

1. le programme dénommé `avl` est une implantation des arbres binaires de recherche équilibrés. La propriété testée est la correction de l'insertion d'un élément dans un arbre dont voici l'énoncé :

$$\forall t \in \texttt{tree(int)}, \forall e \in \texttt{int}, \mathit{is\_avl}(t) \Rightarrow \mathit{is\_avl}(\mathit{insert\_avl(e,\ t)})$$

2. le programme `liste_trie` est une implantation d'une liste triée. Nous testons une propriété de même nature que la précédente.

$$\forall t \in \texttt{list(int)}, \forall e \in \texttt{int}, \mathit{sorted}(t) \Rightarrow \mathit{sorted}(\mathit{insert\_list(e,\ t)})$$

3. la propriété `min_max` est relative au minimum et au maximum d'une liste. Elle exprime que lorsqu'on ajoute un élément dans une liste, le minimum de la liste `cons(e, l)` est le minimum entre `e` et le minimum de `l` :

$$\begin{array}{l}\forall l \in \texttt{list(int)},\\ \forall min\ max\ e \in \texttt{int},\\ \quad is\_min(min, l) \Rightarrow\\ \quad\quad is\_max(max, l) \Rightarrow\\ \quad\quad\quad (min\_list(e :: l) = min\_int(min, e) \wedge\\ \quad\quad\quad max\_list(e :: l) = max\_int(max, e))\end{array}$$

4. le programme `sum_list`, est une fonction qui calcule la somme des éléments d'une liste. Nous testons la propriété suivante :

$$\begin{array}{l}\forall s1, s2 \in \texttt{int},\\ \quad s1 = plus\_list(l1) \Rightarrow\\ \quad\quad s2 = plus\_list(l2) \Rightarrow\\ \quad\quad\quad s1 + s2 = plus\_list(append(l1, l2))\end{array}$$

5. le programme `triangle` est un exemple classique dans la communauté du test. La fonction `triangle` prend en argument trois entiers représentant trois longueurs. Elle renvoie la nature du triangle formé par la composition de trois segments des longueurs données. Les résultats possibles sont : `Équilatéral`, `Isocèle`, `Scalène` ou `Non_triangle`. Nous testons les propriétés de correction de la fonction. Nous montrons celle relative à `Équilatéral` :

$$\forall x, y, z \in \texttt{int}, triangle(x, y, z) = \texttt{Équilatéral} \Rightarrow x = y \wedge y = z$$

6. L'avant dernier cas d'étude est celui du voteur, composant usuel dans l'industrie. Nous nous intéressons à la fonction `vote` qui prend en entrée trois valeurs et retourne une valeur qui est considérée comme *certifiée* si elle est compatible avec au moins deux des valeurs d'entrée. Ce composant est utilisé dans le cadre d'un système qui effectue de la redondance de mesures (3 capteurs). Il permet ainsi d'assurer la mesure avec au plus un capteur défaillant.

   Le voteur plus précisément, retourne un couple composé d'une valeur entière et d'une constante appartenant à l'ensemble { `match`, `nomatch`, `perfect_match` }. Le premier composant est le résultat du vote (l'un des trois entiers en entrée). Le deuxième indique le statut de ce vote : `match` signifie que deux des trois entiers sont compatibles et la valeur de sortie est l'un d'eux. `nomatch` signifie qu'aucun entier n'est compatible avec aucun autre. `perfect_match` signifie que les trois entiers sont compatibles. Ici, deux entiers sont compatibles si leur différence est d'au plus 10. Nous testons plusieurs propriétés de cette fonction. Il s'agit des propriétés de correction, en voici une :

$$\begin{array}{l}\forall v1, v2, v3 \in \texttt{int},\\ \quad compatible(v1, v2) \Rightarrow\\ \quad\quad compatible(v2, v3) \Rightarrow\\ \quad\quad\quad compatible(\sharp fst\,(vote(v1, v2, v3), v1)) \wedge\\ \quad\quad\quad vote(v1, v2, v3) = \texttt{perfect\_match}\end{array}$$

7. Le dernier cas d'étude concerne le test d'un modèle de sécurité de système de fichiers avec droits à la UNIX. Plus précisément, nous testons la propriété de correction de la fonction `create_file`. Cette propriété exprime que si un fichier n'existe pas dans le système de fichiers courant, alors la fonction `create_file` l'ajoute ; il est vide et est initialisé avec les bons droits :

$$\begin{array}{l} \forall x \in self, \\ \forall y \in filename, \\ \forall u \in userprop, \\ \quad not\_in(y, x) \Rightarrow \end{array} \left(\begin{array}{l} in(y, create(x, u, y)) \wedge \\ eq\_uid(get\_uid(y, create(x, u, y)), get\_user\_uid(u)) \wedge \\ sub\_set\_gid(get\_gid(y, create(x, u, y)), u) \wedge \\ get\_content(y, create(x, u, y)) = Unfailed(empty) \end{array} \right)$$

Les deux premiers exemples sont intéressants car ils permettent d'évaluer la capacité de FoCalTest à faire du raisonnement sur des types de données algébriques en présence d'un invariant de représentation. L'exemple `avl` requiert de générer des arbres binaires de recherche équilibrés, `liste_trie` des listes triées. Les fonctions utilisées dans les préconditions sont récursives et utilisent les structures conditionnelles et le filtrage par motif. Ainsi, ces exemples permettent de vérifier l'efficacité des méta-contraintes `ite` et `match`.

Les quatre derniers exemples utilisent essentiellement des structures conditionnelles et les variables utilisées dans le programme sont entières ou d'un type énuméré. Ils permettent de vérifier la capacité de l'outil à trouver des jeux de test à partir de contraintes entières.

9.4.3 Mesures effectuées

Pour chaque exemple traité, nous avons utilisé les trois stratégies suivantes de recherche par résolution de contraintes :

- `Méta-contrainte`, il s'agit de la recherche des jeux de test, présentée dans ce manuscrit, en utilisant les méta-contraintes `ite` et `match` définies au chapitre 8 ;
- `Sans règle arrière`, il s'agit de la stratégie où les méta-contraintes sont pourvues d'une sémantique qui n'utilise pas le « raisonnement arrière ». Nous avons amputé la sémantique de `ite` et `match` des règles qui testent l'implication de leurs sous-branches. Nous obtenons alors la sémantique suivante pour $\texttt{ite}(C, T, E)$:
  1. $C \rightarrow T$
  2. $\neg C \rightarrow E$

  Pour $\texttt{match}(X, [\texttt{pattern}(pat_1, \sigma_1), \ldots, \texttt{pattern}(pat_n, \sigma_n)], \sigma_{n+1})$ on arrive à la sémantique suivante :
  $\forall i \in [\![1, n]\!]$
  1. $X = pat_i \rightarrow \sigma_i$
  2. $n = 1 \wedge \sigma_{n+1} =_{def} \texttt{fail} \rightarrow X = pat_1 \wedge \sigma_1$
  3. $n = 0 \rightarrow \sigma_{n+1}$
- `Naïve`, il s'agit d'une implantation des contraintes `ite` et `match` sous la forme de point de choix Prolog. La définition de $\texttt{ite}(C, T, E)$ est la suivante :

  $$(C \wedge T); (\neg C \wedge T)$$

  Le ; désigne la disjonction Prolog, on pose les contraintes de gauche puis en cas d'échec on pose les contraintes de droite.
  La contrainte `match` est munie d'une sémantique similaire.

Le but est de montrer les différences entre ces trois stratégies et par conséquent de montrer l'apport des méta-contraintes par rapport à une implantation naïve des structures conditionnelles et du filtrage par motif, et d'autre part de montrer que les règles de sémantique « arrière » apportent un gain en terme de temps de résolution.

Programmes	Propriétés	Méta-contrainte	Sans règle arrière	Naïve
`avl`	*Voir section 9.4.2*	864	10926	Out of memory †
`liste_trie`	*Voir section 9.4.2*	39	178335	Out of memory †
`liste avec min/max`	*Voir section 9.4.2*	104	314	Out of memory †
`sum_list`	*Voir section 9.4.2*	34	57	Out of memory †
`triangle`	triangle_type_correct_equi	133	112	115
	triangle_type_correct_iso	204	170	171
	triangle_type_correct_scal	233	196	196
	triangle_type_correct_err	34	27	29
`voteur`	vote_perfect	225	94	54
	vote_range_c1	113	74	54
	vote_range_c2	110	77	243
	vote_range_c3	152	65	54
	vote_partial_c1	218	181	752
	vote_partial_c2	278	191	45
	vote_partial_c3	289	219915	487
create_file	*Voir section 9.4.2*	160	160	160

† le processus de recherche s'est arrêté par manque de mémoire. Malgré une allocation de pile et de tas plus importante, il n'a pas été possible d'éviter cette erreur et de trouver le moindre jeu de test.

FIGURE 9.5 – Recherche des jeux de test par résolution de contraintes

9.4.4 Résultats et conclusions

Nous présentons ici les temps mesurés pour la recherche de jeux de test utilisant les différentes stratégies. Nous présentons cela en deux tableaux. Le premier en figure 9.5 regroupe les temps des trois approches par résolution de contraintes et le deuxième en figure 9.6 montre les résultats de l'approche aléatoire. Tous les temps indiqués sont exprimés en millisecondes.

Nous présentons les résultats de la génération des jeux de test avec l'approche par contraintes dans la figure 9.5. Les temps indiqués sont exprimés en milliseconde.

La première remarque est que pour les exemples qui demandent de manipuler des structures de données algébriques, une implantation naïve de `ite` et `match` ne parvient pas à produire un seul jeu de test sans une explosion de la taille de la pile et ce malgré une allocation plus importante. Cela provient du fait que les points de choix produits par la sémantique naïve impliquent une explosion combinatoire. Pour ces exemples, il est clairement montré que cette sémantique n'est pas appropriée. Pour les autres exemples, les résultats obtenus avec l'implantation naïve des contraintes `ite` et `match` sont comparables aux autres approches. Comme dernière remarque, nous précisons que les écarts de temps entre l'approche naïve et méta-contrainte sont trop faibles pour être significatifs.

Pour les stratégies **Méta-contrainte** et **Sans règle arrière**, nous constatons que pour la majorité des propriétés testées, les différences de temps ne sont pas significatives. Toutefois, pour les trois propriétés `avl`, `liste_trie` et `vote_partial_c3`, la différence de temps est très fortement significative. Sur ces propriétés, le rapport de temps va de 10 à 4000. Il faut ajouter que la propriété `liste_trie` se teste très facilement avec une approche aléatoire et ne justifie pas l'utilisation de techniques de résolution par contraintes. Elle montre les différences qu'il peut y avoir entre les stratégies avec règles arrière et sans règles arrière. Cela montre que les règles arrière ne ralentissent pas la résolution des contraintes mais au contraire sur des exemples précis, accélèrent la détection d'une solution.

Ces différences de temps s'expliquent par l'utilisation des contraintes gardées et plus particulièrement par la sémantique utilisée des méta-contraintes. Dans les méta-contraintes, les tests

Programmes	Propriétés	Random	Nombre de valuation générés
avl		9180235	20000000
liste_trie		< 10	1123
liste avec min/max		162349	20000000 †
Liste d'entier	sum_list	96826	10000000 †
Triangle	triangle_type_correct_equi	66855	10561778
	triangle_type_correct_iso	67001	10593996
	triangle_type_correct_scal	< 10	1042
	triangle_type_correct_err	< 10	12
Voteur	vote_perfect	144891	20000000 †
	vote_range_c1	1693	225104
	vote_range_c2	1608	213720
	vote_range_c3	1898	251580
	vote_partial_c1	149947	20000000 †
	vote_partial_c2	148905	20000000 †
	vote_partial_c3	149965	20000000 †
create_file	*Voir section 9.4.2*	48	40

† le processus de recherche a été arrêté avant de trouver les 10 jeux de test demandés. Pour le temps indiqué, aucun jeu de test n'a été trouvé.

FIGURE 9.6 – Recherches de jeux de test avec la stratégie aléatoire

de consistance se font uniquement sur les gardes. Ainsi, les seules contraintes détectées insatisfaites sont celles qui se trouvent dans les gardes. Dans l'implantation des méta-contraintes sans les règles arrière, les seules contraintes qui se trouvent dans les gardes sont le test de la condition (`ite`) et les filtres (`match`). Lorsqu'une contrainte du corps d'une conditionnelle (ou d'un filtrage) est insatisfaite, l'inconsistance n'est pas détectée au réveil de la contrainte. Au contraire, sur l'implantation avec règles arrière, les contraintes du corps des méta-contraintes se retrouvent dans les gardes. Ainsi, les inconsistances sont détectées au plus tôt, ce qui accélère la résolution.

La figure 9.6 montre les résultats de l'expérience en utilisant la stratégie aléatoire des recherches de jeux de test. Sont indiqués dans le tableau les temps de génération des jeux de test ainsi que le nombre total de valuations générées avant de trouver les 10 jeux de test valides demandés. Pour certains exemples aucun jeu de test n'a été trouvé dans le temps imparti, nous les avons marqués du symbole †.

Avant toute chose, nous remarquons que pour l'exemple `avl`, la recherche des jeux de test est très longue (environ 2h30). Le gain de temps obtenu en utilisant une résolution par contraintes est très clairement montré. À l'exception de trois d'entre eux, les autres exemples tendent à confirmer cette conclusion. Nous remarquons que nous avons arrêté l'expérience après la génération de 10000000 valuations successives qui ne forment pas un jeu de test valide. Pour les exemples où le nombre de valuations générées est de 20000000 nous précisons que ces propriétés mènent à 2 propriétés élémentaires. De manière générale, dès que les préconditions sont un peu contraignantes, la recherche des jeux de test est plus longue avec la stratégie aléatoire qu'avec la stratégie par résolution de contraintes.

9.5 Conclusion

Ces expérimentations montrent que l'utilisation des méta-contraintes est une approche pertinente par rapport à des contraintes plus classiques comme l'implantation `naïve` qui utilise un point de choix Prolog et aussi par rapport à une recherche aléatoire des jeux de test. L'absence

de point de choix dans les méta-contraintes est un atout qui évite de figer l'arbre de résolution. La sémantique particulière donnée aux méta-contraintes contribue en grande partie à l'efficacité de la résolution. En particulier, la définition des règles arrières est pertinente et améliore substantiellement la recherche de solution.

Conclusion et perspectives

Résumé et contributions

Nous avons mis au point une technique de test au sein de l'atelier FoCal. Les propriétés d'un programme FoCal constituent la spécification de celui-ci. Ainsi pour vérifier l'adéquation entre la spécification et l'implantation, nous avons défini une méthode de test de propriétés.

La classe des propriétés testables est une sous-classe des propriétés exprimables dans FoCal. Il s'agit de propriétés en forme prenexe décomposables en un couple précondition/conclusion. Pour une propriété, un jeu de test est une valuation des variables quantifiées. Trouver ces jeux de test implique d'avoir des valeurs d'entrées qui valident la précondition, soit, un ensemble de prédicats. Nous avons présenté deux approches pour obtenir les jeux de test :

1. la première approche est fondée sur la génération aléatoire. Elle consiste à choisir des valeurs au hasard pour chacune des variables quantifiées de la propriété et à vérifier *a posteriori* si la précondition est vérifiée. La vérification de la précondition est faite par l'exécution de l'implantation, c'est-à-dire, des fonctions qu'elle utilise ;
2. la deuxième approche utilise la programmation par contraintes. Elle consiste à traduire la précondition en contraintes. Comme pour l'approche aléatoire, nous utilisons l'implantation des fonctions utilisées dans la propriété. Nous commençons par traduire l'implantation des fonctions impliquées dans la précondition dans un langage intermédiaire appelé FMON. Dans ce langage, les aspects spécifiques à FoCal (espèces, collections, héritage, ...) sont retirés. De plus, ce langage permet de mettre les implantations sous une forme intermédiaire monadique, forme dans laquelle les calculs intermédiaires ont été nommés. Cette traduction a été montrée correcte et complète. Cela signifie que si un programme FoCal s'évalue en une valeur alors le programme en forme intermédiaire monadique s'évalue en cette même valeur et *vice versa*. Nous avons aussi défini un langage de contraintes ainsi qu'un prédicat de test de solution. Nous avons aussi défini une traduction des programmes FMON vers des contraintes du langage de contraintes. Comme pour la première traduction, nous avons prouvé la correction et la complétude de la traduction des programmes FMON vers les contraintes. Cela signifie que si un programme FMON s'évalue en une valeur, alors le système de contraintes obtenu par traduction admet une solution qui associe la valeur de sortie du programme FMON aux valeurs d'entrée qui ont permis de l'obtenir et *vice versa*. Pour obtenir des jeux de test pour une propriété donnée, la précondition est alors traduite en un ensemble de contraintes par la méthode énoncée, puis la résolution du système de contraintes donne directement des valuations des variables qui valident la précondition.

   Nous avons utilisé cette méthodologie pour générer des jeux de test valides mais elle est générique. Elle peut être utilisée avec tout terme combinant des fonctions FoCal et toute contrainte de valeur. Si l'on souhaite obtenir des valeurs d'entrée d'une expression FoCal qui permettent d'arriver à une valeur de sortie v, il suffit de fixer la valeur de retour R

associée à l'expression initiale à la valeur attendue v en ajoutant au système de contraintes résultant de la traduction la contrainte $X = v$. Nous pourrions, par exemple, imposer les valeurs de sortie de chacune des parties de la précondition pour obtenir uns-à-uns les jeux de test qui couvre le critère MC/DC. Cela permettrait d'obtenir une assurance que toutes les parties de la précondition sont indépendantes les unes des autres. Pour aller plus loin, on peut imaginer chercher à appliquer aussi le critère MC/DC sur la conclusion.

Afin d'exprimer les structures particulières à un langage fonctionnel comme FoCal. Nous avons dû définir des contraintes spécifiques. Les méta-contraintes `ite` et `match` permettent d'exprimer dans le langage de contraintes les structures conditionnelles et le filtrage par motif. Nous avons donné deux sémantiques à chacune de ces deux contraintes.

- la première sémantique permet de vérifier si une affectation est une solution des méta-contraintes. Il s'agit d'une vision exécutable des contraintes, cette sémantique est inspirée de la sémantique de FoCal et de FMON.
- la deuxième sémantique est orientée vers la résolution de contraintes. Elle s'exprime à l'aide de contraintes gardées et permet de faire du raisonnement arrière.

Enfin, nous avons montré que les deux sémantiques données aux contraintes `ite` et `match` sont équivalentes.

Ces travaux ont débouché sur l'outil de test FoCalTest. Cet outil est intégré à l'atelier FoCal. Nous avons présenté la méthode utilisée pour poser un harnais de test dans un programme FoCal afin d'automatiser totalement le test d'une propriété. FoCalTest génère un rapport de test au format XML qui peut être réinjecté pour faire du test de non régression.

Nous avons évalué l'outil sur des exemples dans le but de valider l'approche par résolution de contraintes et aussi la sémantique donnée à `ite` et `match`. La sémantique à contraintes gardées de ces deux contraintes est décomposable en deux parties. Une partie raisonnement avant qui correspond à une exécution naturelle de la structure modélisée par la contrainte et une partie raisonnement arrière dont le but est d'éliminer les chemins de la structure qui ne sont pas vérifiés. L'expérimentation a permis de montrer que les règles de raisonnement arrière, bien qu'elles n'augmentent pas l'expressivité des contraintes, réduisent le temps de résolution de manière significative.

Perspectives

Tout au long de ce manuscrit, des hypothèses ont été formulées pour obtenir les résultats. Certaines d'entre elles se retrouvent dans les perspectives.

Formalisation en Coq

Nous avons donné dans ce manuscrit les preuves de correction et de complétude de la traduction des programmes FMON vers les contraintes. Nous avons aussi donné des preuves d'équivalence entre les différentes sémantiques des opérateurs `ite` et `match`. Toutes ces preuves ont été faites manuellement.

De nombreux travaux existent autour de la formalisation de traductions sémantiques en utilisant des assistants à la preuve ou de certification d'algorithmes en général. Nous pouvons par exemple citer la preuve d'équivalence sémantique dans la traduction de MiniML en Cminor [Dar09], langage impératif de bas de niveau du projet de compilateur certifié [BDL06a], ou bien la preuve Coq de l'algorithme de typage W [DMM99].

Mécaniser nos preuves serait un plus qui offrirait un niveau de confiance supérieur dans l'exactitude de notre traduction. C'est pour cela que il est important de formaliser cette traduction dans un assistant à la preuve. FoCal est toujours en développement actif, certaines fonctionnalités ne sont pas encore figées. De plus, FoCal ne propose pas encore la génération de principe d'induction. Les preuves que nous avons élaborées utilisent fortement des raisonnements inductifs. C'est pour ces deux raisons que nous n'envisageons pas de faire une formalisation en FoCal des preuves de correction/complétude. En dehors de FoCal, l'outil qui nous semble le plus naturel pour effectuer une telle preuve est l'assistant à la preuve Coq. Le travail de formalisation en Coq des preuves de ce présent manuscrit est en cours.

Ordre supérieur

Nous n'avons pas traité les aspects liés à l'ordre supérieur dans notre méthode de test.FoCal est un langage de programmation fonctionnelle, l'ordre supérieur est donc le trait principal du langage et l'impossibilité de synthétiser des jeux de test avec la résolution de la précondition est un cruel manque. Ajouter la possibilité de faire du test sur des préconditions qui contiennent de l'ordre supérieur rentre donc dans nos perspectives. La programmation par contraintes n'offre pas la possibilité de faire du raisonnement sur des variables d'ordre supérieur, il faut trouver un moyen de le simuler. Pour cela, il faut surmonter plusieurs obstacles :

- une transformation du code FoCal pourrait permettre de retirer certaines formes liées à l'ordre supérieur, par exemple, les définitions de fonctions anonymes peuvent être extériorisées et transformées en fonctions nommées ;
- un réordonnancement des déclarations d'arguments permet de tirer les **fun** en tête des définitions (**let** $x = 4$ **in fun** $y \rightarrow \ldots$ donne alors **fun** $y \rightarrow$ **let** $x = 4$ **in** $\ldots$), ceci implique de vérifier que la sémantique du programme n'est pas modifiée ;
- l'application d'une fonction à une autre fonction est l'obstacle majeur dans la traduction des programmes FoCal en contraintes. Définir une méta-contrainte spécifique à ce cas pourrait aider à résoudre ce problème.

Améliorations de la résolution de contraintes

Calculs flottants

Le type des flottants n'est pas traduit dans le langage de contraintes, compte tenu du manque de prise en compte de flottants dans la programmation par contraintes. Le calcul sur les flottants pose des difficultés car les opérations d'addition et de multiplication n'ont pas les propriétés usuelles (associativité, distributivité). Des travaux de recherche fondés sur des techniques de *box*-consistance pour réduire la taille des domaines [MRL01] ou de *2b*-consistance [Mic02] permettent d'envisager d'intégrer un solveur pour résoudre des contraintes sur les flottants.

Domaines algébriques/numériques

Le prototype FoCalTest que nous avons développé est une extension de CLP(FD) dans Sicstus Prolog qui utilise les primitives fournies par ce denier pour ajouter des contraintes utilisateurs. Dans le cadre du développement d'un prototype, cette approche est satisfaisante pour obtenir des résultats rapidement. Toutefois elle présente des inconvénients majeurs :

- D'une part, le réveil des contraintes est contrôlé par Sicstus Prolog. Il peut être intéressant de définir nous propres conditions de réveil des méta-contraintes. À la création de la contrainte on spécifie un ensemble de variables de domaine fini pour lesquelles une modification du domaine provoque le réveil de la contrainte. Cette possibilité n'est pas naturellement utilisable pour les variables algébriques. Le solveur de contraintes est orienté

type fini, un type dont les valeurs sont unidimensionnelles. Les conditions de réveil sont dans orienté vers cette caractéristique. Les types algébriques définissent un domaine sur des termes de profondeur quelconque. Le solveur n'offre pas la possibilité de poser des conditions de réveil qui couvre l'intégralité de la forme du terme sous-jacent à une variable. Dans notre implantation, nous déclenchons le réveil des méta-contraintes lorsque le constructeur de tête des variables algébriques est instancié. Nous aimerions être capable de faire des traitements différents selon que la variable dont le domaine est modifié est algébrique ou de domaine fini. Nous pourrions envisager de réveiller en priorité certaines contraintes si la variable est algébrique et d'autres contraintes si la variable est de domaine fini ;

– Nous avons remarqué lors d'essais de résolution de contraintes qui mélangeaient des variables algébriques et numériques que le temps de résolution est très sensible à la manière dont les variables sont énumérées. Pour certains exemples, commencer par définir la structure des variables algébriques (taille d'une liste, forme d'un arbre, ...) puis chercher à trouver des valeurs sur les variables de domaine fini permet d'obtenir une résolution efficace. À l'inverse d'autres exemples se comportent très bien en utilisant un énumération qui précise la valeur des variables de domaine fini dès leur création (au sein d'une valeur algébrique par exemple). Aucune de ces deux méthodes n'est meilleure que l'autre. L'efficacité d'une approche dépend grandement du problème traité. Actuellement, l'énumération implantée dans le prototype ne peut être modifiée. Une amélioration consiste à étudier comment faire interagir deux solveurs de contraintes dédiées aux contraintes des deux domaines. Pendant la phase d'énumération, de trouver des critères pour déterminer l'ordre d'énumération des variables et implanter les heuristiques ainsi créées.

De manière générale les deux problèmes posés plus haut sont imputables au manque de souplesse qu'offre l'actuelle implantation des méta-contraintes. Une perspective à long terme serait de développer un solveur de contraintes dédié au domaine des variables algébriques possédant ses propres heuristiques et paramètres. Un tel solveur pourrait ensuite être interfacé avec un solveur de type CLP(FD) à qui les contraintes numériques seraient sous-traitées.

Test et preuves

L'aspect le plus important à ne pas oublier et le contexte d'intégration de FoCalTest. FoCal est avant tout un atelier de développement dédiée à la preuve de programme. Nous pensons que le test et la preuve ne sont pas des méthodes antinomique mais au contraire peuvent être utilisées conjointement dans l'amélioration de la qualité des logiciels.

Nous n'avons pas eu le temps d'aborder durant cette thèse la thématique du mélange entre le test et la preuve et il nous semble que les deux peuvent subsister l'un avec l'autre pour plusieurs raisons :

– dans une vision pragmatique, nous pouvons constater que le travail de preuve de programme est long et laborieux. Faire une preuve consiste en grande partie à découper la structure de la preuve à l'aide de propriétés intermédiaires. Ces propriétés sont généralement prouvées en dernier lieu lorsqu'elles ont permis d'aboutir à une preuve de la propriété initiale. Bien souvent l'une des propriétés intermédiaires est fausse car elle expose un cas particulier qui n'a pas été pensé par son concepteur. Bien entendu, dans l'état actuel des choses il est possible de tester une propriété intermédiaire avec FoCalTest mais cela demande de sortir de l'environnement de preuve. Une première approche dans l'interaction test et preuve consisterait à intégrer FoCalTest directement dans l'environnement dédié à la preuve et pouvoir exhiber des contres exemples à une propriété intermédiaire au plus tôt. Ce travail à déjà été effectué pour l'assistant à la preuve AGDA [DHT03].

– les preuves des propriétés sont structurées : analyses de cas, inductions sur une valeur .... Les techniques de test structurelle définissent des critères de couverture à atteindre à partir d'une représentation structurée d'un programme. Nous pouvons définir des critères de couverture sur la structure d'une preuve (même incomplète, c'est-à-dire avec des parties admises) et étendre FoCalTest pour qu'il assure une couverture sur une preuve.

Quatrième partie

Annexe

Annexe A

Les opérateurs et leur traduction

1 Opérateurs prédéfinies de FoCal

Nous présentons ici les opérateurs prédéfinis de FoCal et de FMON. Nous rappelons que ces deux langages partagent les mêmes opérateurs de base. Les opérateurs sont présentés en fonction du type de données manipulé.

Opérateurs arithmétiques

$\text{eval\_op}(\sharp \mathit{int\_add}\,(i_1, i_2)) = \text{eval\_op}(i_1) + \text{eval\_op}(i_2)$	$\text{eval\_op}(\sharp \mathit{int\_minus}\,(i_1, i_2)) = \text{eval\_op}(i_1) - \text{eval\_op}(i_2)$
$\text{eval\_op}(\sharp \mathit{int\_mult}\,(i_1, i_2)) = \text{eval\_op}(i_1) * \text{eval\_op}(i_2)$	$\text{eval\_op}(\sharp \mathit{int\_div}\,(i_1, i_2)) = \text{eval\_op}(i_1)/\text{eval\_op}(i_2)$
$\text{eval\_op}(\sharp \mathit{int\_mod}\,(i_1, i_2)) = \text{eval\_op}(i_1) \bmod \text{eval\_op}(i_2)$	$\text{eval\_op}(\sharp \mathit{int\_opp}\,(i)) = -\text{eval\_op}(i)$
$\text{eval\_op}(\sharp \mathit{int\_min}\,(i_1, i_2)) = \min(\text{eval\_op}(i_1), \text{eval\_op}(i_2))$	$\text{eval\_op}(\sharp \mathit{int\_min}\,(i_1, i_2)) = \max(\text{eval\_op}(i_1), \text{eval\_op}(i_2))$
$\text{eval\_op}(\sharp \mathit{int\_succ}\,(i)) = \text{eval\_op}(i) + 1$	$\text{eval\_op}(\sharp \mathit{int\_pred}\,(i)) = \text{eval\_op}(i) - 1$

FIGURE A.1 – Sémantique des opérateurs arithmétiques

La figure A.1 présente la sémantique des opérateurs arithmétiques de FoCal. Il s'agit des opérateurs usuels de manipulation sur les entiers, à savoir l'addition, la soustraction, la multiplication et la division. En plus de ces quatre opérateurs, FoCal fournit les opérateurs de calcul de modulo d'un entier par rapport à un autre, d'opposé d'un entier, des calculs du maximum/-minimum de deux entiers ainsi que l'incrémentation et la décrémentation d'un entier.

Conjointement à ces opérateurs, nous donnons la sémantique des opérateurs de comparaisons et d'égalité des entiers FoCal dans la figure A.2. Il s'agit des opérateurs de comparaison usuels des entiers.

Opérateurs sur les *booléens*

FoCal prédéfinit quatre opérateurs sur les booléens, la figure A.3 les présente. Ces quatre opérateurs sont classiques, il s'agit des trois opérateurs binaires de *conjonction*, *disjonction* et *disjonction exclusive* ainsi que de l'unique opérateur unaire de *négation*.

$$\text{eval\_op}(\sharp \mathit{int\_eq}\,(i_1, i_2)) = (\text{eval\_op}(i_1) = \text{eval\_op}(i_2))$$

$$\text{eval\_op}(\sharp \mathit{int\_lt}\,(i_1, i_2)) = \text{eval\_op}(i_1) < \text{eval\_op}(i_2)$$
$$\text{eval\_op}(\sharp \mathit{int\_leq}\,(i_1, i_2)) = \text{eval\_op}(i_1) \leq \text{eval\_op}(i_2)$$

$$\text{eval\_op}(\sharp \mathit{int\_gt}\,(i_1, i_2)) = \text{eval\_op}(i_1) > \text{eval\_op}(i_2)$$
$$\text{eval\_op}(\sharp \mathit{int\_geq}\,(i_1, i_2)) = \text{eval\_op}(i_1) \geq \text{eval\_op}(i_2)$$

FIGURE A.2 – Sémantique des opérateurs de comparaisons

$$\text{eval\_op}(\sharp \mathit{and\_b}\,(v_1, v_2)) = \text{eval\_op}(v_1)\ \&\ \text{eval\_op}(v_2)$$

$$\text{eval\_op}(\sharp \mathit{xor\_b}\,(v_1, v_2)) = \text{eval\_op}(v_1)\ \text{xor}\ \text{eval\_op}(v_2) \qquad \text{eval\_op}(\sharp \mathit{not\_b}\,(v)) = \neg\text{eval\_op}(v)$$

$$\text{eval\_op}(\sharp \mathit{or\_b}\,(v_1, v_2)) = \text{eval\_op}(v_1)\ |\ \text{eval\_op}(v_2)$$

FIGURE A.3 – Sémantique des opérateurs booléens

Opérateurs sur les *couples*

$$\text{eval\_op}(\sharp \mathit{first}\,(\mathit{Crp}(v_1, v_2))) = \text{eval\_op}(v_1) \qquad \text{eval\_op}(\sharp \mathit{crp}\,(v_1, v_2)) = \mathit{Crp}(v_1, v_2)$$

$$\text{eval\_op}(\sharp \mathit{scnd}\,(\mathit{Crp}(v_1, v_2))) = \text{eval\_op}(v_2)$$

FIGURE A.4 – Sémantique des opérateurs sur les couples

La figure A.4 présente les opérateurs de manipulations des valeurs de couple. Ils sont au nombre de trois, il y a un opérateur qui construit un couple à partir de deux valeurs et les deux opérateurs de projection sur le premier élément et le deuxième élément.

Opérateurs sur le type *partiel*

La figure A.4 présente les opérateurs du type partiel. Le premier opérateur $\sharp \mathit{is\_failed}$ permet de tester si la valeur est `true` ou non. Le deuxième opérateur $\sharp \mathit{non\_failed}$ permet de faire une projection.

2 Opérateurs prédéfinies du langage de contraintes

Nous présentons maintenant les opérateurs prédéfinis du langage de contraintes présenté à la section 7.3.

Ces opérateurs sont définis de manière à établir une correspondance avec les opérateurs de FoCal. Cela permet de faciliter la traduction des opérateurs de FoCal vers les opérateurs de contraintes. De plus, les preuves de corrections et de complétudes de la traduction sur les opérateurs sont plus directes.

$$\text{eval\_op}(\sharp \textit{is\_failed}\,(\textit{Failed})) = \texttt{true} \qquad \text{eval\_op}(\sharp \textit{non\_failed}\,(\textit{Unfailed}(v))) = \text{eval\_op}(v)$$

$$\text{eval\_op}(\sharp \textit{is\_failed}\,(\textit{Unfailed}(v))) = \texttt{false}$$

FIGURE A.5 – Sémantique des opérateurs sur le type partiel

Opérateurs arithmétiques

$$\text{eval\_op}(+_{fd}, i_1, i_2) = \text{eval\_op}(i_1) + \text{eval\_op}(i_2) \qquad \text{eval\_op}(*_{fd}, i_1, i_2) = \text{eval\_op}(i_1) * \text{eval\_op}(i_2)$$

$$\text{eval\_op}(-_{fd}, i_1, i_2) = \text{eval\_op}(i_1) - \text{eval\_op}(i_2) \qquad \text{eval\_op}(/_{fd}, i_1, i_2) = \text{eval\_op}(i_1)/\text{eval\_op}(i_2)$$

$$\text{eval\_op}(\textit{mod}_{fd}, i_1, i_2)) = \text{eval\_op}(i_1) \bmod \text{eval\_op}(i_2)$$

FIGURE A.6 – Sémantique des opérateurs arithmétiques

La figure A.6 présente la sémantique des opérateurs arithmétiques des contraintes. Il s'agit des quatre opérateurs classiques d'addition, de soustraction, de division et de multiplication ainsi que du calcul du modulo de deux entiers.

On remarque que certains opérateurs de FoCal n'ont pas de correspondance directe avec les opérateurs définis ici. Par exemple, nous ne donnons pas de définition correspondant à $\sharp \textit{succ}$ et à $\sharp \textit{pred}$. Néanmoins, les opérateurs de la figure A.6 peuvent être combinés pour retrouver les opérateurs FoCal manquants.

$$\text{eval\_op}(=_{fd}, v_1, v_2) = (\text{eval\_op}(v_1) = \text{eval\_op}(v_2))$$

$$\text{eval\_op}(\leq_{fd}, v_1, v_2, v_3) = (\text{eval\_op}(v_1) = \text{eval\_op}(v_2) \leq \text{eval\_op}(v_3))$$
$$\text{eval\_op}(<_{fd}, v_1, v_2, v_3) = (\text{eval\_op}(v_1) = \text{eval\_op}(v_2) < \text{eval\_op}(v_3))$$

$$\text{eval\_op}(\geq_{fd}, v_1, v_2, v_3) = (\text{eval\_op}(v_1) = \text{eval\_op}(v_2) \geq \text{eval\_op}(v_3))$$
$$\text{eval\_op}(>_{fd}, v_1, v_2, v_3) = (\text{eval\_op}(v_1) = \text{eval\_op}(v_2) > \text{eval\_op}(v_3))$$

FIGURE A.7 – Sémantique des opérateurs de comparaisons

La figure A.7 présente les opérateurs de comparaison des entiers. Il s'agit là aussi des opérateurs classiques identiques à ceux de FoCal.

Opérateurs sur les *booléens*

Les opérateurs arithmétiques booléens des contraintes sont définis avec une arité différente de ceux de FoCal. Il s'agit des versions réifiées des contraintes booléennes. C'est-à-dire, ils ont un argument suplémentaire qui permet, pour d'eux, de forcer la valeur de vérité de l'opération logique. Ils sont définis dans la figure A.8

$$\text{eval\_op}(\textit{and\_b}, v_1, v_2, v_3) = (\text{eval\_op}(v_1) = \text{eval\_op}(v_2)\ \&\ \text{eval\_op}(v_3))$$

$$\text{eval\_op}(\textit{or\_b}, v_1, v_2, v_3) = (\text{eval\_op}(v_1) = \text{eval\_op}(v_2)\ |\ \text{eval\_op}(v_3))$$

$$\text{eval\_op}(\textit{xor\_b}, v_1, v_2, v_3) = (\text{eval\_op}(v_1) = \text{eval\_op}(v_2)\ \text{xor}\ \text{eval\_op}(v_3))$$

$$\text{eval\_op}(\textit{not\_b}, v_1, v_2) = (\text{eval\_op}(v_2) = \neg\text{eval\_op}(v_2))$$

FIGURE A.8 – Sémantique des opérateurs de comparaison

Opérateurs sur les *couples*

$$\text{eval\_op}(\textit{first}, v_1, \texttt{crp}(v_2, v_3)) = (\text{eval\_op}(v_1) = \text{eval\_op}(v_2))$$

$$\text{eval\_op}(\textit{scnd}, v_1, \texttt{crp}(v_2, v_3)) = (\text{eval\_op}(v_1) = \text{eval\_op}(v_3))$$

FIGURE A.9 – Sémantique des opérateurs sur les couples

La figure A.9 présente les opérateurs de contraintes qui manipulent les couples. Ce sont les mêmes opérateurs que FoCal mais en version réifiée.

L'opérateur de création d'un couple est directement défini à partir du type de FoCal. On rappelle que `crp` est le symbole de constructeur du monde des contraintes associé au symbole de constructeur FoCal *Crp* des couples.

Opérateurs sur le type *partiel*

$$\text{eval\_op}(\textit{is\_failed}, v, \texttt{failed}) = (v_1 = \texttt{true})$$

$$\text{eval\_op}(\textit{is\_failed}, v, \texttt{unfailed}(v_2)) = (v_1 = \texttt{false})$$

$$\text{eval\_op}(\textit{non\_failed}, v_1, \texttt{unfailed}(v_2)) = (v_1 = v_2)$$

FIGURE A.10 – Sémantique des opérateurs sur le type partiel

La figure A.10 récapitule la sémantique les opérateurs de manipulation de valeurs sur le type partiel. Il sont au nombre de deux :

- le premier opérateur $\sharp\textit{is\_failed}$ permet de convertir une valeur partielle en un booléen qui spécifie si le constructeur est `fail` ou `unfailed` ;
- le deuxième opérateur $\sharp\textit{non\_failed}$ permet de faire une projection, il n'est défini que lorsque la valeur du type partiel est `unfailed`.

3 Traduction des opérateurs en contraintes

Les figures A.11, A.12, A.13, A.14 et A.15 présentent les fonction de traductions des opérateurs de FoCal vers les contraintes.

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp int\_add\,(x,y) \longmapsto R =_{fd} \mathcal{T}_x(x) +_{fd} \mathcal{T}_x(y)}\ \text{INT\_ADD}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp int\_minus\,(x,y) \longmapsto R =_{fd} \mathcal{T}_x(x) -_{fd} \mathcal{T}_x(y)}\ \text{INT\_MINUS}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp int\_mult\,(x,y) \longmapsto R =_{fd} \mathcal{T}_x(x) *_{fd} \mathcal{T}_x(y)}\ \text{INT\_MULT}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp int\_div\,(x,y) \longmapsto R =_{fd} \mathcal{T}_x(x) /_{fd} \mathcal{T}_x(y)}\ \text{INT\_DIV}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp int\_mod\,(x,y) \longmapsto R =_{fd} mod_{fd}(\mathcal{T}_x(x), \mathcal{T}_x(y))}\ \text{INT\_MOD}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp int\_opp\,(x) \longmapsto R =_{fd} 0 -_{fd} \mathcal{T}_x(x)}\ \text{INT\_OPP}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp int\_min\,(x,y) \longmapsto R =_{fd} min_{fd}(\mathcal{T}_x(x), \mathcal{T}_x(y))}\ \text{INT\_MIN}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp int\_max\,(x,y) \longmapsto R =_{fd} max_{fd}(\mathcal{T}_x(x), \mathcal{T}_x(y))}\ \text{INT\_MAX}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp int\_succ\,(x) \longmapsto R =_{fd} \mathcal{T}_x(x) + 1}\ \text{INT\_SUCC}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp int\_pred\,(x) \longmapsto R =_{fd} \mathcal{T}_x(x) - 1}\ \text{INT\_PRED}$$

FIGURE A.11 – Traduction des opérateurs arithmétiques

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp int\_eq\,(x,y) \longmapsto R =_{fd} \mathcal{T}_x(x) =_{fd} \mathcal{T}_x(y)}\ \text{INT\_EQ}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp int\_lt\,(x,y) \longmapsto R =_{fd} \mathcal{T}_x(x) <_{fd} \mathcal{T}_x(y)}\ \text{INT\_LT}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp int\_leq\,(x,y) \longmapsto R =_{fd} \mathcal{T}_x(x) \leq_{fd} \mathcal{T}_x(y)}\ \text{INT\_LEQ}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp int\_gt\,(x,y) \longmapsto R =_{fd} \mathcal{T}_x(x) >_{fd} \mathcal{T}_x(y)}\ \text{INT\_GT}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp int\_geq\,(x,y) \longmapsto R =_{fd} \mathcal{T}_x(x) \geq_{fd} \mathcal{T}_x(y)}\ \text{INT\_GEQ}$$

FIGURE A.12 – Traduction des opérateurs arithmétiques de comparaison

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp and\_b\,(x,y) \longmapsto R = and\_b(\mathcal{T}_x(x), \mathcal{T}_x(y))}\ \text{BOOL\_AND}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp or\_b\,(x,y) \longmapsto R = or\_b(\mathcal{T}_x(x), \mathcal{T}_x(y))}\ \text{BOOL\_OR}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp xor\_b\,(x,y) \longmapsto R = xor\_b(\mathcal{T}_x(x), \mathcal{T}_x(y))}\ \text{BOOL\_XOR}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp not\_b\,(x) \longmapsto R = not\_b(\mathcal{T}_x(x))}\ \text{BOOL\_NOT}$$

FIGURE A.13 – Traduction des opérateurs booléens

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp crp\,(x,y) \longmapsto R = \texttt{crp}(\mathcal{T}_x(x), \mathcal{T}_x(y))}\ \text{FIRST}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp first\,(x) \longmapsto first(R, \mathcal{T}_x(x))}\ \text{FIRST}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp scnd\,(x) \longmapsto scnd(R, \mathcal{T}_x(x))}\ \text{SCND}$$

FIGURE A.14 – Traduction des opérateurs sur les couples

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp is\_failed\,(x) \longmapsto is\_failed(R, \mathcal{T}_x(x))}\ \text{IS\_FAILED}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C \sharp non\_failed\,(x) \longmapsto non\_failed(R, \mathcal{T}_x(x))}\ \text{NON\_FAILED}$$

FIGURE A.15 – Traduction des opérateurs sur le type partiel

Table des figures

Bibliographie

[ABC+02] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. BZ-TT : A Tool-Set for Test Generation from Z and B using Constraint Logic Programming. In *Proc. of Formal Approaches to Testing of Software (FATES 2002) workshop of CONCUR'02)*, pages 105–120, Brno, République Tchèque, August 2002. INRIA report. Cité page 19

[ACD+08] P. Ayrault, M. Carlier, D. Delahaye, C. Dubois, D. Doligez, L. Habib, T. Hardin, M. Jaume, C. Morisset, F. Pessaux, and P. Weis. Trusted Software within FoCaL. In *CE&SAR*, pages 142–157, Rennes, France, December 2008. Cité page 27

[AH00] S. Antoy and R. G. Hamlet. Automatically Checking an Implementation against Its Formal Specification. *IEEE Trans. Software Eng.*, 26(1) :55–69, 2000. Cité page 21

[AHP08] P. Ayrault, T. Hardin, and F. Pessaux. Development life cycle of critical software under FoCal. In *2nd International Workshop on Harnessing Theories for Tool Support in Software (TTSS'08)*, Istanbul, 2008. Cité page 50

[AKPS92] H. Aït-Kaci, A. Podelski, and G. Smolka. A Feature-based Constraint System for Logic Programming with Entailment. Research Report RR-92-17, Deutches Forschungszentrum für Künstliche Intelligenz GmbH, March 1992. Cité page 104

[BaFG+03] C. Bigot, alain Faivre, J.-P. Gallois, A. Lapitre, D. Lugato, J.-Y. Pierron, and N. Rapin. Automatic Test Generation with AGATHA. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619/2003 of *Lecture Notes in Computer Science*, pages 591–596. Springer Berlin, 2003. Cité page 26

[BBD+09] S. Bardin, B. Botella, F. Dadeau, F. Charreteur, A. Gotlieb, B. Marre, C. Michel, M. Rueher, and N. Williams. Constraint-Based Software Testing. In I. Press, editor, *Journée du GDR-GPL 09*, ENSEEIHT, Toulouse, January 2009. Yves Ledru and Marc Pantel. Cité page 26

[BDD07] R. Bonichon, D. Delahaye, and D. Doligez. Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs. In *Proceedings ofLogic for Programming Artificial Intelligence and Reasoning (LPAR)*, pages 151–165, 2007. Cité pages 27 et 32

[BDL06a] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *Formal Methods (FM06)*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer Berlin, June 2006. Cité page 154

[BDL06b] F. Bouquet, F. Dadeau, and B. Legeard. Automated Boundary Test Generation from JML Specifications. In *Formal Methods (FM)*, volume 4085 of *Lecture Notes in Computer Science*, pages 428–443, Hamilton, Ontario (Canada), August 2006. Springer. Cité page 19

[BGL+07] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. A subset of precise UML for model-based testing. In *A-MOST 2007*, pages 95–104, 2007. Cité page 26

[BGM90] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications : a theory and a tool. Research Report Rapport 581, L.R.I, Université Paris Sud/Orsay, June 1990. Cité page 21

[BGM+02] B. Botella, A. Gotlieb, C. Michel, M. Rueher, and P. Taillibert. Utilisation des contraintes pour la génération automatique de cas de test structurels. *Technique et Science Informatiques*, 21(9) :1163–1187, 2002. Cité page 19

[BGS+03] M. Barnett, W. Grieskamp, W. Schulte, N. Tillmann, and M. Veanes. Validating use-cases with the asml test tool. In *QSIC '03 : Proceedings of the Third International Conference on Quality Software*, page 238, Washington, DC, USA, 2003. IEEE Computer Society. Cité page 26

[BHH+99] S. Boulmé, T. Hardin, D. Hirschkoff, V. Ménissier-Morain, and R. Rioboo. On the way to certify Computer Algebra Systems. In *Proceedings of the Calculemus workshop of FLOC'99 (Federated Logic Conference, Trento, Italie)*, volume 23 of *ENTCS*. Elsevier, 1999. Cité page 27

[BHR00] S. Boulmé, T. Hardin, and R. Rioboo. Polymorphic Data types, Objects, Modules and Functors : is it too much ? Research report, Laboratoire Informatique de Paris 6, 2000. Cité page 27

[BKM02] C. Boyapati, S. Khurshid, and D. Marinov. Korat : Automated Testing Based on Java Predicates. In *International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 123–133, Rome, Italie, July 2002. Cité page 18

[BLLP04] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications : GSM 11.11 standard case-study. Research Report RR2004-16, LIFC - Laboratoire d'Informatique de l'Université de Franche Comté, December 2004. Cité page 19

[BN04] S. Berghofer and T. Nipkow. Random Testing in Isabelle/HOL. In *Software Engineering and Formal Methods (SEFM'04)*, pages 230–239. IEEE Computer Society, 2004. Cité pages 23 et 58

[Bou00a] S. Boulmé. Opérateurs de raffinement sur les structures algébriques. In *Actes des Journées Francophones des Langages Applicatifs*, 2000. Cité page 27

[Bou00b] S. Boulmé. *Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel*. Thèse de Doctorat, Université Paris 6, 2000. Cité pages 27 et 61

[BW] A. Brucker and B. Wolff. Le site internet de HOL-TestGen. http://www.brucker.ch/projects/hol-testgen/. Cité page 23

[BW07] A. Brucker and B. Wolff. Test-Sequence Generation with HOl-TestGen – With an Application to Firewall Testing. In B. Meyer and Y. Gurevich, editors, *TAP 2007 : Tests and Proofs*, Lecture Notes in Computer Science, pages 149–168, Heidelberg, 2007. Springer-Verlag. Cité page 23

[CCCL08] Y. Cheon, A. Cortes, M. Ceberio, and G. T. Leavens. Integrating Random Testing with Constraints for Improved Efficiency and Diversity. In *Proceedings of SEKE 2008, The 20-th International Conference on Software Engineering and Knowledge Engineering*, pages 861–866, San Francisco, CA, July 2008. Cité page 20

[CCY98] T. Chen, S. Cheung, and S. Yiy. Metamorphic testing : a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998. Cité page 21

[CFR+91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficently Computing Static Single Assignment Form and the Control Dependence Graph. *Transactions on Programming Languages and Systems*, 13(4) :451–490, 1991. Cité page 18

[CH00] K. Claessen and J. Hughes. QuickCheck : a lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP 2000)*, pages 268–279, 2000. Cité page 22

[CH02] K. Claessen and J. Hughes. Testing monadic code with QuickCheck. *SIGPLAN Notices*, 37(12) :47–59, 2002. Cité page 22

[CL02] Y. Cheon and G. T. Leavens. A simple and Practical Approach to Unit Testing : The JML and JUnit Way. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, volume 2374, pages 231–255, London, UK, 2002. Springer-Verlag. Cité page 20

[Com06] Common Criteria Recognition Arrangement. *Common Criteria for Information Technology Security Evaluation, version 3.1*, 2006. Further information available at http://www.commoncriteriaportal.org. Cité page 1

[CRM07] Y. Cheon and C. E. Rubio-Medrano. Random Test Data Generation for Java Classes Annotated with JML Specifications. In *Proceedings of the 2007 International Conference on Software Engineering Research and Practice*, volume 2, pages 385–392, Las Vegas, Nevada, June 2007. Cité page 20

[Dar09] Z. Dargaye. *Vérification formelle d'un compilateur optimisant pour un langage fonctionel*. Thèse de Doctorat, Université Paris 7, July 2009. Cité page 154

[dBOP+98] L. du Bousquet, F. Ouabdesselam, I. Parissis, J.-L. Richier, and N. Zuanon. Lutess : a testing environment for synchronous software. In *Tool support for System Specification, Development, and Verification, Advances in Computing Science (TOOLS'98)*, pages 48–61, Malente, Germany, 1998. Springer. Cité pages 24 et 25

[dBORZ99] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess : a specification-driven testing environment for synchronous software. In *Proceedings of the 21th international conference on software engineering*, pages 267–276, Los angeles, California, United States, 1999. ACM. Cité pages 24 et 25

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *International Symposium on Formal Methods Europe (FME'93)*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284. Springer-Verlag, April 1993. Cité page 19

[DFLS04] P. Duchon, P. Flajolet, G. Louchard, and G. Schaeffer. Boltzmann samplers for random generation of combinatorial structures. *Combinatorics, Probability, and Computing*, 13(3–4) :577–625, 2004. Cité page 58

[DGD07] T. Denmat, A. Gotlieb, and M. Ducassé. An Abstract Interpretation Based Combinator for Modelling While Loops in Constraint Programming. In *International Conference on Principles and Practice of Constraint Programming (CP'07)*, volume 4741/2007, pages 241–255. Springer Berlin/Heidelberg, October 2007. Cité page 19

[DGG04] A. Denise, M.-C. Gaudel, and S.-D. Gouraud. A Generic Method for Statistical Testing. In *International Symposium on Software Reliability Engineering (ISSRE 2004)*, pages 25–34, 2004. Cité page 15

[DHT03] P. Dybjer, Q. Haian, and M. Takeyama. Combining testing and proving in dependent type theory. In David Basin and Burkhart Wolff, editors, *Internatianal Conference on Theorem Proving in Higher Order Logics (TPHOL'03)*, volume 2758 of *Lecture Notes in Computer Science*, pages 188–203. Springer-Verlag, 2003. Cité pages 23 et 156

[DHVG04] C. Dubois, T. Hardin, and V. Viguié Gouge. Building certified components within FOCAL. In *Symposium on Trends in Functional Programming (TFP04)*, volume 5, pages 33–48, 2004. Cité page 27

[Dij72] E. W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10) :859–866, 1972. Turing Award lecture. Cité page 9

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection : Help for the practicing programmer. In *Computer*, volume 11, pages 34–41. IEEE, April 1978. Cité page 13

[dMB08] L. de Moura and N. Bjørner. Z3 : An Efficient SMT Solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Budapest, Hungary, 2008. Cité page 18

[DMM99] C. Dubois and V. Ménissier-Morain. Certification of a type inference tool for ML : Damas-Milner within Coq. *Journal of Automated Reasoning*, 23(3) :319–346, 1999. Cité page 154

[DÉVDG06] D. Delahaye, J.-F. Étienne, and V. Viguié Donzeau-Gouge. Certifying Airport Security Regulations using the Focal Environment. In *Formal Methods (FM)*, volume 4085 of *Lecture Notes in Computer Science*, pages 48–63, Hamilton, Ontario (Canada), August 2006. Springer. Cité page 28

[Fec01] S. Fechter. Une sémantique pour FoC. Rapport de D.E.A., Université Paris 6, Septembre 2001. Cité page 27

[Fec05] S. Fechter. *Sémantique des traits orientés objet de* Focal. Thèse de Doctorat, Université Paris 6, July 2005. Cité pages 61 et 71

[Frü95] T. Frühwirth. Constraint Handling Rules. In *Constraint Programming : Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 90–107. Springer-Verlag, 1995. Cité page 25

[FZVC94] P. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, 132(1-2) :1–35, 1994. Cité page 58

[GA95] P. L. Gall and A. Arnould. Formal Specifications and Test : Correctness and Oracle. In *COMPASS/ADT*, pages 342–358, 1995. Cité page 21

[GB03] A. Gotlieb and B. Botella. Automated metamorphic testing. Publication interne 1516, Institut de recherche en informatique et systèmes aléatoires, 2003. Cité page 22

[GBR98] A. Gotlieb, B. Botella, and M. Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In *International Symposium on Software Testing and Analysis (ISSTA'98)*, pages 53–62, 1998. Cité pages 19 et 127

[GBR00] A. Gotlieb, B. Botella, and M. Rueher. A CLP Framework for Computing Structural Test Data. In *International Conference on Computational Logic (CL'00)*, pages 399–413, 2000. Cité pages 18 et 127

[GG08] M.-C. Gaudel and P. L. Gall. Testing Data Types Implementations from Algebraic Specifications. In *Formal Methods and Testing*, pages 209–239, 2008. Cité page 21

[Got09] A. Gotlieb. Euclide : A Constraint-Based Testing framework for critical C programs. In *International Conference on Software Testing, Verification and Validation(ICST'09)*, 2009. Cité page 18

[GRT07] P. L. Gall, N. Rapin, and A. Touil. Symbolic Execution Techniques for Test Refinement. In *Test And Proof 2007*, Lecture Notes in Computer Science, Zurich, Suisse, February 2007. Cité page 17

[HJL+06] J. J. Hunt, E. Jenn, S. Leriche, P. Schmitt, I. Tonin, and C. Wonnemann. A Case Study of Specification and Verification using JML in an Avionics Application. In *4th Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, 2006. Cité page 20

[How77] W. E. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Trans. Softw. Eng.*, 3(4) :266–278, 1977. Cité page 17

[IEE98] IEEE. IEEE standard for software test documentation, 1998. Cité page 9

[JM99] T. Jéron and P. Morel. Test Generation Derived from Model-checking. In *Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*. Springer, 1999. Cité page 26

[JM06] M. Jaume and C. Morisset. A formal approach to implement access control. *Journal of Information Assurance and Security*, 2 :137–148, June 2006. Cité page 28

[KAaRP03] P. Koopman, A. Alimarine, and J. T. ans Rinus Plasmeijer. Gast : Generic Automated Software Testing. In *14th International Workshop on the Implementation Functional Languages*, pages 84–100. Springer, 2003. Cité page 23

[KATP02] P. W. M. Koopman, A. Alimarine, J. Tretmans, and M. J. Plasmeijer. Gast : Generic Automated Software Testing. In *Implementation Functional Languages (IFL 2002)*, pages 84–100, 2002. Cité page 23

[KOAB08] S. Kansomkeat, J. Offutt, A. Abdurazik, and A. Baldini. A Comparative Evaluation of Tests Generated from Different UML Diagrams. In *Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2008)*, pages 867–872, Phuket Thailand, August 2008. Cité page 26

[LP01] B. Legeard and F. Peureux. Génération de séquences de test à partir d'une spécification B en PLC ensembliste. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'01)*, pages 113–130, Nancy, France, June 2001. Cité page 19

[LP05] A. Lakehal and I. Parissis. Lustructu : A tool for the automatic coverage assessment of lustre programs. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pages 301–310, Washington DC, USA, 2005. IEEE Computer Society. Cité page 24

[LPU04] B. Legeard, F. Peureux, and M. Utting. Controlling Test Case Explosion in Test Generation from B Formal Models. Research Report RR2004-14, LIFC - Laboratoire d'Informatique de l'Université de Franche Comté, December 2004. Cité page 19

[Mar91] B. Marre. Toward automatic test data set selection using algebraic specifications and logic programming. In *International Conference on Logic Programming (ICLP'91)*, pages 202–219, Paris, June 1991. MIT press. Cité pages 21 et 25

[Mar92] L. Maranget. *La stratégie paresseuse*. Thèse de Doctorat, Université Paris 7, July 1992. Cité page 87

[Mar98] B. Marre. Vers une génération automatique de jeux de tests en utilisant les spécifications algébriques et la programmation logique. In *Congrès AFCET de Génie Logiciel CGL4*, pages 19–29, Paris, October 1998. Cité page 21

[Mic02] C. Michel. Exact projection functions for floating point number constraints. In *7th International symposium on Artificial Intelligence and MAthematics (AIMA'02)*, Fort Lauderdale, Floride (US), January 2002. Cité page 155

[MMW04] B. Marre, P. Mouy, and N. Williams. On-the-Fly Generation of K-Path Tests for C Functions. In *19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, Linz, Austria, September 2004. Cité page 18

[MPMU04] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58 :89–106, 2004. Cité page 20

[MRL01] C. Michel, M. Rueher, and Y. Lebbah. Solving constraints over floating-point numbers. In *International Conference on Principles and Practice of Constraint Programming (CP'01)*, volume 2239 of *Lecture Notes in Computer Science*, pages 524–438, November 2001. Cité page 155

[Mye04] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., July 2004. Cité page 15

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. Cité page 23

[OA99] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference*, volume 1723, pages 416–429, Fort Collins, CO, USA, October 1999. Springer. Cité page 26

[Ori05] C. Oriat. Jartege : A Tool for Random Generation of Unit Tests for Java Classes. In *Second International Workshop on Software Quality (SOQUA'05)*, volume 3712 of *Lecture Notes in Computer Science*, pages 242–256, Erfurt, Germany, September 2005. Cité page 20

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS : A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag. Cité page 23

[Piv08] C. Pivoteau. *Génération aléatoire de structures combinatoires : méthode de Boltzmann effective*. Thèse de Doctorat, Université Paris 6, 2008. Cité page 58

[PJJ+07] S. Pickin, C. Jard, T. Jéron, J.-M. Jézéquel, and Y. L. Traon. Test synthesis from UML models of distributed software. In *IEEE Transactions on Software Engineering*, volume 33, pages 252–268, April 2007. Cité page 26

[Pre03] V. Prevosto. *Conception et implantation du langage FoC pour le développement de logiciels certifiés*. Thèse de Doctorat, Université Paris 6, 2003. Cité pages 61 et 95

[Pre05] V. Prevosto. Certified mathematical hierarchies : the FoCal system. In Thierry Coquand, Henri Lombardi, and Marie-Françoise Roy, editors, *Mathematics, Algorithms, Proofs*, number 05021 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany. Cité page 27

[RTC92] RTCA. Software considerations in airborne systems and equipment certification. *RTCA*, 1992. Cité page 1

[SH99] B. Schätz and F. Huber. Integrating Formal Description Techniques. In *Formal Methods (FM'99)*, number 1709 in Lecture Notes in Computer Science, pages 1206–1225, 1999. Cité page 25

[SO06] Sam Owre. Random Testing in pvs. In *Workshop on Automated Formal Methods (AFM'06)*, 2006. Cité page 23

[Sta08] I. Staff. IEEE standard for Floating-Point Arithmetic. Research Report 754-2008, IEEE, August 2008. Cité page 62

[TdH08] N. Tillmann and J. de Halleux. Pex – White Box Test Generation for .NET. In B. Beckert and R. Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008*, volume 4966 of *Lecture Notes in Computer Science*, pages 182–191, Prato, Italy, 2008. Springer. Cité page 18

[TF89] P. Thévenod-Fosse. Software validation by means of statistical testing : Retrospect and future direction. In *International Working Conference on Dependable Computing for critical Applications (DCCA 1989)*, pages 15–22, 1989. Cité page 15

[TFW93] P. Thévenod-Fosse and H. Waeselynck. STATEMATE applied to statistical software testing. In *International Symposium on Software Testing and Analysis (ISSTA 1993)*, pages 99–109, Cambridge, Massachusetts, United States, 1993. Cité page 15

[VCG+08] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In *Formal Methods and Testing*. Springer Verlag, 2008. Cité page 26

[VHD90] P. Van Hentenryck and Y. Deville. The Cardinality Operator : A New Logical Connective for Constraint Logic Programming. Research Report CS-90-24, Brown University, October 1990. Cité page 45

[VPK04] W. Visser, C. Păsăreanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT internatinal symposium on Software testing and analysis*, volume 29, pages 97–107, July 2004. Cité page 18

[WS08] S. Weißleder and D. Sokenou. Automatic Test Case Generation from UML Models and OCL Expressions. In *Testmethoden für Software - Von der Forschung in die Praxis (TESO'08)*, pages 423–426, München 2008. Cité page 25

Résumé

L'environnement FoCal, développé conjointement par des chercheurs des laboratoires CÉDRIC, LIP6 et de l'INRIA, permet de construire de manière incrémentale des composants de bibliothèque avec un haut niveau de confiance et de qualité. Un composant de bibliothèque FoCal peut contenir des spécifications, l'implantation des opérations spécifiées et des preuves que la spécification et l'implantation sont en conformité. Les composants FoCal sont traduits en code exécutable OCaml et vérifiés par l'assistant à la preuve Coq.

Même si un développement en FoCal nous assure un haut degré de confiance dans le programme, de par une spécification formelle et les preuves de correction, nous ne pouvons nous passer du test. Les propriétés de bas niveau, comme les axiomes, ne sont en général pas prouvées. FoCal autorise l'importation de code non sûr au sein de code FoCal certifié, ce qui peut amener à « casser » la spécification. Finalement, le test peut être utilisé pour avoir confiance en une implantation dont la preuve de correction n'est pas encore effectuée ou terminée ou lorsqu'une tentative de preuve échoue, pour trouver des contre-exemples.
Ce manuscrit présente le développement d'un outil de test intégré à FoCal. La méthodologie repose sur le test automatique de propriétés. Les jeux de test sont définis comme des valuations des variables quantifiées de la propriété sous test, qui satisfont les prémisses de la propriété (précondition). La conclusion de la propriété sert d'oracle pour le test. Nous proposons deux stratégies de recherche des jeux de test : aléatoire et par résolution de contraintes. Pour cette dernière, nous traduisons la précondition de la propriété en un système de contraintes. Sa résolution nous permet d'obtenir les jeux de test recherchés. Cette méthodologie est formellement définie et nous prouvons que les jeux de test ainsi construits satisfont la précondition.

Mots-clefs : test automatique, test de propriété, méta-contrainte, préservation sémantique, programmation par contraintes, programme certifié, FoCal

Abstract

The FoCal environment, developed by researchers coming from the laboratories LIP6, CÉDRIC and INRIA, allows one to incrementally build library components with a high level of confidence and quality. A component of a FoCal library can contain specifications, implementations of operations and proofs that the implementations satisfy their specifications. The components are translated into OCaml code and verified by the Coq proof assistant.

Even if the FoCal environment ensures a high level of confidence because of its methodology based on specification and proofs, we cannot do without testing. Low level properties, for example axioms, are usually not proven. FoCal permits one to import OCaml untrusted code into a FoCal certified code, which can *break* the specification. Finally, testing could be used to check an implementation when a proof attempt fails or when the proof is not yet done, in order to find counter-examples for examples.

This thesis deals with the design and development of a testing tool integrated into FoCal. The methodology consists in testing properties automatically. Test data are defined as valuations of the quantified variables of the property under test which satisfy the premises (called the precondition) of the property. The conclusion of the property serves as an oracle. We propose two approaches to select test data: randomly or through the usage of constraint reasoning. For the latter case, we translate the precondition into a constraint system whose resolution will provide us with the expected test data. Our approach is formally defined and we prove that the test data obtained by constraint resolution satisfy the precondition.

Keywords : automatic testing, test of property, meta-constraint, semantic preservation, constraint programming, certified program, FoCal