Conception et implantation du langage FoC pour le développement de logiciels certifiés

Virgile Prévosto

THÈSE de DOCTORAT de l'UNIVERSITÉ PARIS 6

Spécialité :
INFORMATIQUE

présentée par :

Virgile Prevosto

pour obtenir le grade de
DOCTEUR de l'UNIVERSITÉ PARIS 6

Conception et implantation du langage Foc pour le développement de logiciels certifiés

Soutenue le 15 septembre 2003,

Devant le jury composé de :

M. Damien DOLIGEZ	INRIA	Directeur
Mme. Thérèse HARDIN	LIP 6	Directrice
M. Gérard HUET	INRIA	Examinateur
M. Claude KIRCHNER	LORIA	Rapporteur
M. Cesar MUÑOZ	NIA	Rapporteur
M. Christian QUEINNEC	LIP 6	Président
M. Sylvain BOULMÉ	IMAG	Invité

Remerciements

Mes remerciements iront bien sûr avant tout à mes deux co-directeurs de thèse, Thérèse Hardin et Damien Doligez. Leurs indications et leurs conseils m'ont été précieux tout au long de ces trois ans, et ils ont toujours été présents quand il le fallait. De même, je voudrais remercier Claude Kirchner et César Muñoz d'avoir accepté de rapporter cette thèse, et d'avoir consacré une partie de leurs mois de juillet et d'août à relire le manuscrit. Leurs suggestions m'ont été particulièrement utiles. Merci aussi à Christian Queinnec, Gérard Huet et Sylvain Boulmé d'avoir eu la gentillesse de bien vouloir faire partie du jury.

Je remercie les membres du projet FoC, passés et présents, et en particulier ceux qui ont utilisé les premières versions du compilateur et ont par là même contribué grandement à son amélioration : Renaud Rioboo, dont les algorithmes ont souvent poussé les mécanismes de traduction en ocaml dans leurs derniers retranchement, Mathieu Jaume, friand de traduction vers Coq, et les différentes générations de stagiaires qui se sont succédées : Louis Mandel, Lazar Bibin, Matthieu Grandjean et Olivier Bonnet. Je n'oublie pas non plus les autres membres du projet, Valérie Ménissier-Morain et Stéphane Fechter, ni les autres stagiaires, en particulier Manuel Maarek et Violaine Ruffié, qui ont apporté une contribution importante au parseur de FoC.

Le projet FoC n'est toutefois pas la seule présente au 4ème étage du LIP6. Je voudrais donc remercier les membres des équipes SPI et CALFOR pour l'ambiance générale dans laquelle s'est déroulée cette thèse. Merci en particulier à David Massot, notre secrétaire particulièrement habile à dénouer n'importe quel problème administratif, et à Grégoire Hamon pour ses récits d'aventure groënlandaises et islandaises, ainsi que sa participation à une des randonnées les plus humides de l'histoire des Alpes italiennes.

La place me manque un peu pour remercier individuellement tous les membres de l'INRIA Rocquencourt qui ont contribué à la réussite de cette thèse. Je remercie donc collectivement les membres des projets Moscova, chez qui j'ai toujours trouvé le meilleur accueil lors de mes visites hebdomadaires, Cristal et Logical, sans qui FoC n'existerait pas et qui ont toujours été d'une grande disponibilité pour répondre à mes questions concernant leurs langages favoris respectifs.

Dans un tout autre registre, il me faut dire un grand merci à Ludo, Pascal, Nico, Sémi, et tous ceux qui d'innombrables fois m'ont accueilli en gare d'Antibes, Gap ou Embrun afin d'aller admirer avec eux les panoramas des Écrins, du Queyras ou du Mercantour. Ces parenthèses d'un week-end ou d'une semaine à l'air pur des montagnes m'ont apporté les moment de détente nécessaire.

Enfin, je veux remercier mes parents qui m'ont toujours fait confiance et soutenu dans mes choix. C'est avant tout grâce à eux que j'ai pu en arriver là. Je dois également leur associer mes grands-parents, qui m'ont beaucoup

appris pendant mes jeunes années. Merci enfin à mon frère, avec qui j'ai pu explorer de nombreux recoins de cette vallée de Postua qui reste le berceau de la famille. J'en profite d'ailleurs pour étendre ces remerciements à mes oncles, tantes et cousins (quel qu'en soit le degré, comme il se doit dans les familles italiennes), que j'ai toujours autant de plaisir à revoir à quelque occasion que ce soit.

Pour conclure, je souhaiterais remercier tous ceux qui à un moment quelconque, par leur accueil, leurs conseils, leurs paroles, ou simplement leur présence m'ont permis de mener à bien toutes les les étapes qui ont mené à cette thèse. Je n'ai bien évidemment pas la place de tous les citer, mais qu'ils soient assurés que je ne les oublie pas.

Table des matières

Introduction

Cette thèse s'est déroulée dans le cadre du projet Foc, acronyme de Formel, Ocaml, Coq. Le but de ce projet était à l'origine de construire un environnement dédié au développement de librairies de calcul formel certifié [BHMR98]. La contribution principale de la thèse à ce projet est l'élaboration d'une syntaxe concrète, le langage Foc, ainsi que la sémantique de l'héritage de ce langage, et sa compilation vers Ocaml et Coq. Ce langage permet de décrire le passage progressif de la spécification à l'implantation des algorithmes, et de faire la preuve, vérifiée par Coq, que ces implantations vérifient bien les spécifications correspondantes.

Dans le reste de cette introduction, nous allons brièvement (section 1) présenter l'enjeu de la certification de programmes et plus généralement de l'utilisation de méthodes formelles pour la sécurité du logiciel. On verra ensuite (section 2) pourquoi le calcul formel est un domaine intéressant pour l'étude d'un modèle de développement basé sur les méthodes formelles. Cela permettra en outre de replacer ces travaux de thèse dans le contexte plus général du projet Foc, et de présenter les travaux antérieurs du projet qui ont permis l'élaboration du langage. Enfin, on détaillera dans la section 5 le cahier des charges initial de la thèse proprement dite, et le plan de cette dernière.

1 Programmation certifiée et logiciels "sûrs"

1.1 Quelle confiance accorder aux programmes ?

Les systèmes informatiques sont présents à peu près partout dans notre vie quotidienne. Chaque nouvelle génération de téléphone portable présente de nouvelles fonctions, dont la plupart font massivement appels aux puces de plus en plus puissantes dont sont munis ces appareils. Par ailleurs toute l'infrastructure (antennes relais, centraux téléphoniques, etc) permettant d'établir une communication est entièrement contrôlée par des ordinateurs. Mais les télécommunications ne sont pas un exemple isolé : il arrive désormais couramment que ce soit un système informatique qui contrôle la chaudière d'une maison, le système de freinage et le moteur d'une voiture, ou simple-

ment les différents lecteurs (cd, dvd, mp3) qui commencent à équiper nos salons. On ne parlera même pas des appareils photos numériques, dont le développement fulgurant semble en passe de détrôner la pellicule dans un futur relativement proche.

Ces systèmes offrent deux grands avantages : ils sont capable de traiter automatiquement de grandes quantités d'informations, et d'effectuer des tâches répétitives. De plus, par rapport à leurs concurrents mécaniques ou éléctroniques, ils sont généralement plus facilement adaptables : il est plus facile de modifier quelques lignes du code source d'un programme que de reconstruire tout un mécanisme effectuant la même tâche. De plus, les programmes sont généralement moins coûteux à réaliser.

Cependant, la complexité croissante des tâches demandées aux logiciels rend leur comportement de plus en plus difficile à prévoir, en particulier lorsqu'ils sont soumis à des conditions qui n'avaient pas été incluses dans la spécification de départ. L'exemple le plus frappant à cet égard est le désormais fameux "bug de l'an 2000". Si ce dernier n'a eu que peu de conséquences dramatiques, il a cependant montré qu'il est extrèmement difficile de prévoir comment un système réagit lorsqu'on sort du cadre pour lequel il a été conçu. Ce fait a également été illustré par l'échec du vol Ariane 501, du à la réutilisation sans grande précaution d'une partie du code contrôlant Ariane 4. En effet, les moteurs d'Ariane 5 étant beaucoup plus puissants que ceux de la version précédente, les capteurs ont envoyés des données en dehors de l'intervalle où le programme s'attendait à les recevoir. Ce dernier n'a donc pas pu les interpréter correctement, provoquant la déviation de la trajectoire, et finalement la destruction de la fusée.

Plus grave encore, il est souvent extrèmement difficile d'être sûr qu'un logiciel effectue correctement les tâches qu'on attend de lui. De nombreux contrats de licence signalent d'ailleurs que le développeur du programme ne saurait être tenu responsable d'éventuels préjudices causés par le dysfonctionnement du logiciel. En effet, depuis le début de l'informatique, on sait qu'il n'existera jamais d'algorithme permettant de valider automatiquement un programme : le problème est indécidable. Cependant, la généralisation de l'informatique évoquée ci-dessus impose qu'on fournisse des systèmes fiables, et qu'on dispose de moyens efficaces de contrôler cette fiabilité.

1.2 Méthodes formelles et sûreté de fonctionnement

Un premier moyen d'obtenir une telle fiabilité est bien sûr de multiplier les tests, aussi bien dans les conditions "normales" de fonctionnement que dans des situations "exceptionnelles" (ces deux notions étant à définir au préalable pour le système considéré). Ces tests permettent de repérer et d'éliminer un certain nombre de bugs, avant que le système soit utilisé réellement. Si ce processus de tests est indispensable dans le cycle de développement d'un

logiciel, il n'est toutefois bien souvent pas suffisant. En effet, il souffre d'un certain nombre d'inconvénients. En premier lieu, il peut être parfois difficile d'effectuer la simulation du système physique que le programme doit contrôler, ou de vérifier que les résultats du test sont corrects. Par exemple, dans le cas particulier du calcul formel, si l'implantation d'un nouvel algorithme produit des résultats sur des données qu'aucun autre système n'est en mesure de traiter, de quelle garantie disposons-nous sur la correction des résultats en question ?

Un autre problème concerne l'exhaustivité des tests. Il n'est en effet pas toujours facile de prédire l'ensemble des situations auxquelles le système devra faire face, ni si les jeux de tests choisis constituent un bon modèle de cet ensemble. Les données avec lesquelles on va tester un logiciel doivent donc être choisies avec soin, et nécessitent un dialogue entre l'équipe qui développe le logiciel, et les spécialistes du domaine d'application afin de juger de la pertinence des modèles obtenus.

Enfin, les tests ne peuvent intervenir, par définition, qu'à la fin du cycle de développement d'un logiciel : pour pouvoir tester un programme, il faut que ce dernier sous une forme compréhensible par la machine sur laquelle il doit être exécutée, c'est à dire généralement une suite de bits. Or le développement d'un logiciel est une succession de traductions de l'algorithme à implanter dans divers langages (que ces traductions soit effectuées à la main ou automatiquement), depuis une spécification (en français, par le biais de schémas, etc), jusqu'à la version exécutable. Chaque étape est susceptible d'introduire des erreurs, depuis une mauvaise spécification de ce qu'on attend du logiciel, jusqu'à une erreur de programmation, en passant par une prise en compte incomplète des spécifications exigées. Sachant que la prise en compte d'une erreur est d'autant plus facile à faire que sa détection a lieu tôt dans le développement, le génie logiciel tend à l'heure actuelle à promouvoir des méthodes permettant la détection des bugs dès les première phases de la conception du système. La validation par des tests reste indispensable, mais ce n'est plus le seul moyen d'éviter des erreurs.

Dans ce contexte, les *méthodes formelles* cherchent à donner une preuve mathématique qu'un système est sans bug. Pour cela, on doit décrire mathématiquement les spécification du système et fournir un modèle mathématique de l'implantation censée répondre aux spécifications. Il est alors possible de raisonner sur ce modèle, et d'en prouver des propriétés de correction vis-à-vis de la spécification. Cette approche ne peut évidemment prétendre éliminer tous les bugs, dans la mesure où les preuve sont faites à un assez haut niveau d'abstraction, et ne garantissent pas la correction du code qui sera finalement exécuté.

Parmi les différentes méthodes possibles, la *programmation certifiée* s'attache à intégrer les preuves dans la programmation même des logiciels. Pour cela, on doit donc disposer d'un environnement dans lequel il est possible d'écrire en même temps des spécifications, des programmes, et des preuves

que les programmes sont corrects. Le but de cette approche est d'obliger le programmeur à formaliser explicitement les arguments permettant de prouver que le programme est correct. En particulier, dans le cas d'une boucle (ou d'une récursion dans le monde des langages fonctionnels), le programmeur connaît généralement les invariants (propriétés qui sont vraies à chaque passage dans la boucle) et les variants (propriétés qui garantissent la terminaison, généralement en montrant qu'une certaine quantité décroît strictement vers 0) de la boucle. Ces informations reste généralement implicites, par exemple sous la forme d'un commentaire dans le code source. En les explicitant, on se donne la possibilité de les faire vérifier mécaniquement par le système, et ainsi de détecter une éventuelle erreur dans le code source du programme.

Toutefois, la programmation certifiée pose deux problèmes particuliers (en dehors du fait que là encore il faut faire confiance au compilateur qui va transformer le code source certifié en un code exécutable). Tout d'abord, il faut sélectionner avec soin les instructions du langage. En effet, si les langages de programmation actuels offrent de nombreuses constructions facilitant la tâche du programmeur, toutes ne se laissent pas facilement interpréter en termes plus mathématiques. À l'inverse, il paraît illusoire de réaliser de grands développements en se contentant du minimum de commandes exigé par la théorie. Il est donc important de fournir à l'utilisateur d'un environnement de programmation certifié un ensemble bien étudié de constructions, qui d'une part lui permettront d'implanter facilement les algorithmes qu'il souhaite, et d'autre part possèdent une sémantique claire, permettant de raisonner facilement dessus.

2 Un exemple de développement : une librairie de calcul formel

Dans ce contexte, l'implantation d'un système dédié au développement de librairies de calcul formel certifiées semble constituer un bon exemple d'un environnement de programmation certifiée. En premier lieu, le domaine du calcul formel s'appuie fortement sur des résultats mathématiques. La formalisation des spécifications est donc déjà faite, et on peut se concentrer sur leur expression à l'intérieur du système. Par ailleurs, les algorithmes utilisés dans le cadre du calcul formel sont souvent assez, voire très complexes. De ce fait, un environnement de programmation adapté doit impérativement disposer d'un langage suffisamment expressif afin d'être utilisable en pratique.

Très grossièrement, le calcul formel (*Computer Algebra* en anglais) s'attache à réaliser des calculs sur les structures mathématiques usuelles, comme les entiers, les polynômes, les matrices, etc. Le principal intérêt des algorithmes de calcul formel est qu'ils fournissent des résultats *exacts*, par comparaison avec les méthodes numériques qui en général ne peuvent donner

qu'une *approximation* de la solution. En effet, dans ce second cas, lorsque les erreurs d'arrondi s'additionnent, le résultat final peut être notablement différent de la solution mathématique, contrairement aux algorithmes de calcul formel.

Toutefois, calculer des solutions exactes est en général beaucoup plus coûteux que calculer des solutions approchées, et les implantations concrètes d'algorithmes de calcul formel font généralement appel à des optimisations plus ou moins compliquées afin de pouvoir mener à bien les calculs pour des problèmes de taille importante. Or quand une nouvelle optimisation permet d'obtenir des résultats pour des données qui étaient auparavant trop grosses, on peut se demander quelle confiance accorder à la valeur obtenue. En effet, on ne dispose d'aucun point de comparaison. Comme par ailleurs les temps de calculs sont généralement assez longs, il serait intéressant d'avoir un certain niveau de certification dans l'implantation des algorithmes.

3 Pourquoi un nouveau langage ?

Une première version de la librairie Foc [BHMR98] a été réalisée en Ocaml. Comme nous le verrons plus en détail dans les chapitres 4 et 8, c'est également vers Ocaml que sont compilés les programmes Foc. La première traduction proposée, celle du chapitre 4, est d'ailleurs très proche de la librairie écrite directement en Ocaml. Un certain nombre de raisons ont toutefois conduit à la définition d'un nouveau langage au-dessus d'Ocaml.

En premier lieu, il s'agissait d'incorporer des propriétés et des preuves aux fonctions chargées de faire les calculs. Cette dimension n'est absolument pas prise en compte par Ocaml. C'est à Coq (chapitre 6) que revient la tâche de vérifier les preuves fournies dans un source Foc. De ce fait, disposer d'un langage pouvant être compilé à la fois vers Ocaml et vers Coq apporte une garantie certaine sur la cohérence globale de la bibliothèque, que n'auraient pas deux développements distincts, l'un en Ocaml et l'autre en Coq.

Il aurait cependant été possible de faire un développement complet en Coq uniquement, et de se servir des fonctionnalités d'extraction de ce langage pour effacer les composantes de preuves et obtenir un programme Ocaml. Toutefois cette approche souffre de deux problèmes majeurs. D'une part, la formalisation complète des différentes structures utilisées en Coq, telle qu'elle a été proposée dans [Bou00] conduit à des termes Coq de très grande taille, à tel point qu'en pratique le vérificateur de type de Coq atteint assez vite ses limites, même pour des structures relativement simples. D'autre part, la composante "langage de programmation" de Coq manque d'un certain nombre de traits importants pour le développement de la bibliothèque, comme l'héritage ou la liaison retardée. Il est bien entendu possible d'utiliser différents codages pour retrouver ces traits, mais l'extraction vers Ocaml aurait porté la marque de ce codage, empêchant de tirer parti de certains

traits de OCAML. Au contraire, un langage compilable à la fois vers OCAML et COQ permet d'utiliser pleinement toutes les constructions de OCAML.

Par ailleurs, le développement direct de la librairie en OCAML a été effectué en respectant des règles de codage très strictes. Les analyses effectuées par le compilateur OCAML ne peuvent pas à elles seules garantir le respect de ces règles. Au contraire, le compilateur FOC a été conçu pour effectuer les vérifications nécessaires. De plus, il engendre bien sûr du code OCAML respectant les règles précédemment définies.

Enfin, l'existence de ce nouveau langage permet de tester plus facilement de nouvelles représentations des structures mathématiques en OCAML : il suffit de modifier une petite partie du compilateur, et non l'intégralité de la librairie.

4 Lien avec d'autres travaux

Le travail qui est présenté ici s'appuie avant tout sur les travaux antérieurs du projet FOC [BHMR98, BHR99, BHR00, Bou00]. En premier lieu, la spécification des différentes structures nécessaires à l'élaboration d'une librairie de calcul formel, ainsi que leurs relations mutuelles a été réalisée dans [BHMR98], et a formé la base du développement du langage FOC. De même, les différents prototype de la librairie écrits en OCAML, ainsi que la discipline de codage décrite dans [BHR00] ont directement inspiré la première version de la traduction de FOC en OCAML décrite dans le chapitre 4.1.1. Enfin, la formalisation concrète en COQ des opérations de base entre les diverses structures de la librairie [Bou00] a constitué un modèle pour le développement des différentes analyses décidant de la correction d'un programme FOC. Les chapitres 9 et 10 montrent d'ailleurs les liens qui existent entre le langage FOC actuel et cette formalisation.

Par ailleurs, le développement du langage FOC peut être rapproché d'un certain nombre d'autres travaux. On peut diviser ces derniers en quatre grandes catégories :

1. Le développement de bibliothèque de calcul formel dans un langage objet (existant ou développé simultanément).
2. L'interfaçage d'un système de calcul formel avec un système d'aide à la preuve.
3. La spécification (et la preuve) d'algorithmes mathématiques dans un système d'aide à la preuve et/ou de démonstration automatique.
4. Les systèmes permettant de mêler abstraction de type et programmation orientée objet.

4.1 Calcul formel et objets

Axiom [JS92, WBD+95] est un système de calcul formel qui peut être vu par certains côtés comme un parent de Foc. Il repose en effet sur une librarie composée de structures construites par héritages successifs à partir d'ensembles très abstraits. Toutefois, le système de type d'Axiom est plus "permissif" que celui de Foc, ce qui rend la vérification de programmes Axiom relativement difficile. Des travaux sont en cours pour munir le successeur d'Axiom, Aldor, d'un langage de types dépendants, afin d'obtenir plus de garanties de la part du vérificateur de type [RT99]. Le système de type d'Aldor est d'ailleurs assez puissant pour permettre d'exprimer des propriétés [PT98] que doivent vérifier les implantations. Ces propriétés pourraient alors être prouvées dans des systèmes d'aide à la preuve interfacés avec Axiom, comme Coq. Un sous-ensemble du système de type d'Aldor, Aldor--, qui pourrait être représenté entièrement en Coq est en cours de définition.

Par ailleurs, ObjectMath [VF92, FEV93] est une extension ajoutant des traits orientés-objet au dessus du langage de Mathematica. Le but principal de cette extension est de factoriser au maximum le développement du code, en donnant une structure au différents modèles mathématiques utilisés et en décrivant leurs interactions. En outre, cela favorise le prototypage de différents modèles répondant au même problème. La même idée est présente dans [LGF00], qui propose un ensemble de classes et d'interfaces Java pour faciliter la manipulation des différentes représentations possibles des matrices (creuses, denses, par blocs, par bandes, etc.).

4.2 Interfacer un système de calcul formel et un système d'aide à la preuve

Un certain nombre de projets ont eu pour but de faire communiquer un système de calcul formel et un système d'aide à la preuve. En particulier, Harrison et Théry [HT98] ont utilisé HOL pour vérifier formellement certains résultats fournis par Maple, comme une intégration (il "suffit" de demander à HOL de dériver ce résultat, et éventuellement de lui faire faire quelques simplifications pour vérifier la correction). Cette approche permet donc de s'assurer que Maple renvoie un résultat correct sur un calcul donné, mais pas d'avoir des garanties sur l'algorithme lui-même : le système de calcul formel reste une "boîte noire".

Dans le même ordre d'idées, il est possible de faire interagir plus étroitement un système de calcul formel et un système d'aide à la preuve. En effet, dans de nombreux cas, les calculs effectués par le premier ne sont valides que sous certaines hypothèses, dont la vérification est souvent hors de portée du calcul formel. Par exemple, les routines d'intégration de Maple ne fonctionnent correctement que si la fonction à intégrer est continue sur l'in-

tervalle demandé. Au lieu de se contenter de vérifier le résultat, [DGKM01] utilise PVS pour vérifier cette hypothèse de continuité.

Inversement, il est aussi possible d'utiliser un système de calcul formel pour réaliser des étapes de calcul intervenant au cours d'une preuve. Ainsi, dans [BHC95], Maple est utilisé comme "oracle" pour produire des témoins dans des preuves d'existence en Isabelle.

Enfin, il est possible d'adopter une approche ne privilégiant aucun des deux côtés (calcul formel ou système de preuve) et laissant l'utilisateur d'un système quelconque communiquer avec les autres. Pour cela, il est nécessaire de disposer d'un langage commun, permettant d'échanger des connaissances mathématiques entre différents systèmes. En particulier, deux standards XML s'attachent à développer ce point. Il s'agit d'OpenMath [DGW97] pour la représentation de données mathématiques, et d'OMDoc [Koh03] pour la description des structures algébriques. Ces standards sont en particulier utilisés dans le MathWeb Software Bus [FK00], qui fait communiquer divers systèmes de calcul formel et d'aide à la preuve par le biais d'Internet. Cette possibilité d'échanger rapidement des connaissances présente un intérêt certain pour Foc, et des travaux sont en cours pour interfacer Foc et OMDoc [MP03].

4.3 Spécification et preuve

La représentation de hiérarchies algébriques a aussi été étudiée du point de vue des systémes d'aides à la preuve, et dans le cadre de la théorie des types. En particulier, Betarte [Bet98] a donné une axiomatisation des enregistrements avec champs dépendants dans le cadre de la théorie des types de Martin-Löf. Cette axiomatisation s'attache en outre à définir une relation de sous-typage entre enregistrement, qui dans le cas des structures algébriques est intimement liée à la notion d'héritage.

Par ailleurs, Pollack [Pol00] propose une théorie d'enregistrements extensibles, basée sur les enregistrements de Coq. D'après sa nomenclature, deux sortes d'extensions sont possibles : "vers la droite" (on ajoute les champs à la fin de l'enregistrement préexistant), ou "vers la gauche" (on ajoute de nouveaux champs au début de l'enregistrement). Dans son travail, Pollack privilégie les enregistrements extensibles "à gauche". Ces derniers sont en effet plus facile à exprimer dans le calcul des constructions. Des hiérarchies de structures mathématiques basées sur un codage de ces enregistrements extensibles en Coq ont permis de montrer le théorème fondamental de l'algèbre [GPWZ01]. Par ailleurs, Pottier [Pot99] a également utilisé les enregistrements de Coq pour développer une telle hiérarchie, jusqu'aux polynômes. Contrairement à Foc, ces développements ne s'intéressent toutefois pas à la redéfinition de méthodes et à la liaison tardive. Ces deux points intéressent en effet plus spécifiquement la programmation : on veut faire des calculs avec

la définition la plus "récente" d'une fonction donnée. Ils posent cependant des problèmes particuliers, matérialisés par la gestion des dépendances entre les différents champs.

Des travaux similaires ont été effectués dans d'autres systèmes. On peut citer en particulier Jackson, qui propose dans [Jac94] le codage d'une hiérarchie (montant là encore jusqu'aux polynômes) en Nuprl. Son travail s'éloigne toutefois des précédents et de FOC dans la mesure où il n'y a pas de sous-typage à proprement parler. En effet, si on peut raffiner les propriétés d'une structure donnée (passer d'un groupe à un groupe abélien par exemple), il est impossible d'ajouter de nouvelles opérations. Un tel ajout oblige, dans sa représentation, à changer complètement l'univers dans lequel vit la structure. En d'autres termes, les relations d'héritage entre les structures mathématiques ne sont pas reflétées dans cette hiérarchie. De même, les *interprétations de théories* de PVS [OS01] permettent de spécifier des structures algébrique, puis de les raffiner pour obtenir différentes implantations.

Enfin, le système Theorema [BJV98] est une extension de Mathematica dans laquelle il est possible de faire des preuves. il repose sur un démonstrateur générique, qui fait appel à des prouveurs plus spécialisés suivant la forme de la preuve à obtenir (induction, réécriture, etc.). Si Theorema n'est pas basé à proprement parler sur une hiérarchie de structures algébriques, il constitue un bon modèle de ce que pourrait être un système de démonstration adapté au calcul formel.

4.4 Systèmes de type

La hiérarchie de structures de FOC rappelle avant tout celle des langages orientés-objets, décrite par exemple dans [AC94]. En effet, l'héritage, la liaison tardive et la coexistence de méthodes définies et déclarées sont communes à FOC et aux modèles "à classes" des langages objet. On peut toutefois constater deux différences importantes. En premier lieu, le type support de FOC n'a pas vraiment d'équivalent. À l'inverse, les objets sont en général caractérisés par un état, c'est à dire une liste de variables d'instances susceptibles d'être modifiées au cours de l'exécution d'un programme. Ces variables n'existent pas en FOC. Toutefois, comme on le verra dans les chapitres qui suivent, la couche objet de OCAML [RV98] offre des mécanismes assez puissants pour exprimer relativement aisément les différentes constructions de FOC. Par ailleurs, Stéphane Fechter étudie la sémantique opérationnelle de FOC [Fec01] en s'inspirant de la sémantique d'Objective-ML.

Un autre système particulièrement intéressant de notre point de vue est celui des classes de types de Haskell [SP01]. On peut les rapprocher des modèles "multiméthodes" des langages objets. Il s'agit en fait de marier surcharge et inférence de type, en utilisant des mécanismes de contraintes de type. Par exemple, on peut définir une fonction d'exponentiation générique

sur tout type A pour lequel il existe une fonction *mult* de type $A \to A \to A$. Une librairie de calcul formel allant jusqu'aux polynômes a été écrite en utilisant ce système [Mec01].

Nous avons dit précédemment que les classes et les objets présentent de nombreux point communs avec Foc, en particulier au niveau de l'héritage. Toutefois, les objets ne permettent pas d'obtenir l'abstraction voulue lorsqu'on crée une collection. Cette dernière fait plutôt appel aux mécanismes de modules des langages fonctionnels. Les modules mixins de [AZ98] permettent de mélanger abstraction et héritage, et leur théorie est donc précieuse pour Foc. De plus, les mixins, ainsi que les modules récursifs, sont en cours d'intégration en Ocaml [HL02].

Si les trois systèmes que nous venons d'évoquer (objets, classes de type et mixins) sont susceptibles de répondre à un certain nombre de problèmes concernant la partie programmation de Foc, ils étudient assez peu les spécifications et les théorèmes. Un article récent [OCRZ03] étudie l'extension des mécanismes d'héritage des objets à des langages de types dépendants, ce qui pourrait ouvrir la voie à des objets mêlant calculs et preuves. De même, les travaux de Judicaël Courant concernant l'implantation d'un système de modules en Coq [Cou01] pourraient permettre de structurer des développements faits en Coq en unités bien distinctes. Toutefois, ce système de module ne permet pas de prendre en compte l'héritage : comme pour Ocaml, il faudrait l'étendre avec des mixins.

5 Plan de la thèse

L'objectif de cette thèse est de fournir un langage – Foc – dans lequel la description de structures telles que celles qui ont été présentées ci-dessus soit aisée, de même que l'écriture d'algorithmes agissant sur ces structures et leurs spécifications. Dans un premier temps, nous examinerons en détail de quels mécanismes on a besoin pour cela (chapitre 1). Ensuite, on examinera le langage Foc tel qu'il se présente actuellement, et on montrera à l'aide d'exemples comment il peut répondre aux besoins précédemment exprimés (chapitre 2).

Dans un second temps, nous nous intéresserons plus particulièrement à la construction du compilateur Foc, qui à partir d'un programme écrit en Foc va engendrer un fichier Ocaml, destiné à réaliser des calculs, et un fichier Coq, qui permettra la vérification des preuves par le système Coq. Tout d'abord, le chapitre 3 présente les différentes analyses effectuées pour s'assurer qu'un programme Foc est correct. Ensuite, on décrit une première traduction possible des structures Foc en Ocaml (chapitre 4). Cette traduction est basée sur les mécanismes orientés-objet de ce langage, et est fortement inspirée des prototypes de bibliothèques écrits directement en Ocaml par Renaud Rioboo. Dans le chapitre 6, on s'intéresse à la traduction

de Foc vers Coq. Contrairement au cas précédent, Coq ne dispose pas de traits objets, et il a donc fallu trouver une représentation adéquate pour un certain nombre de mécanismes nécessaires à Foc. Cette étude a permis de développer une seconde traduction en Ocaml, décrite dans le chapitre 8, qui reprend les grandes idées de la traduction vers Coq.

Enfin, la dernière partie de la thèse est consacrée à la correction du compilateur lui-même. Les chapitres 5 et 7 montrent que les mécanismes retenus pour la traduction vers Ocaml utilisant les objets (respectivement la traduction vers Coq) produisent bien des définitions correctes, c'est à dire bien typées pour les deux langages respectifs. On montre de plus les correspondances qui existent entre le code Foc de départ et chacun des deux résultats de la traduction. Enfin, nous introduirons dans le chapitre 9 la formalisation des structures de Foc faites en Coq par Sylvain Boulmé à l'aide des structures appelées *mixDrecs*. La suite de ce chapitre montre que chacune des structures construites en Foc a un équivalent au niveau des mixDrecs, tandis que le chapitre 10 étudie la façon dont les relations entre les composantes d'une même structure se retrouvent au niveau du mixDrec correspondants.

Chapitre 1

Représenter des structures mathématiques

Pour développer une librairie dans un domaine donné, il convient en premier lieu de définir précisément les objets qui sont manipulés dans ce domaine, et les différentes façons dont ils peuvent interagir. Dans le cas du calcul formel, on peut envisager quatre grands types d'objets, que nous allons détailler par la suite :

1. les éléments des différents ensembles sur lesquels on travaille, comme 0, $X^2 + 2X + 1$, les fonctions sur les entiers, etc. Nous les appelerons *entités*.

2. les structures algébriques génériques telles que les groupes, les anneaux, ou les polynômes à coefficients dans un anneau quelconque. Nous appelerons ces structures des *espèces*. On peut d'ores et déjà noter que ces structures forment une hiérarchie : un corps est un anneau où tout élément non nul possède un inverse. De même, les polynômes forment eux-même un anneau. Un anneau est lui-même un groupe abélien possédant une opération supplémentaire $*$ associative et ayant un élément neutre, etc.

3. les opérations et les propriétés qui définissent ces structures, regroupées sous le nom de *méthodes*.

4. les structures particulières, telles que l'anneau des entiers $\mathbb{Z}$, l'anneau des polynômes à coefficients entiers, $\mathbb{Z}[X]$, l'anneau des polynômes à plusieurs variables et à coefficients entiers, $\mathbb{Z}[X_1, ..., X_n]$, etc. Nous parlerons alors de *collections*. C'est dans les collections qu'on manipule effectivement les entités : les espèces sont des vues plus générales et plus abstraites qui servent à bâtir pas à pas les collections dont on a besoin.

1.1 Quelques structures

Avant de présenter plus en détail les différentes composantes d'une *espèce*, nous allons définir rapidement quelques structures algébriques utilisées en calcul formel, et qui serviront de base aux exemples développés dans la suite du texte.

- un *setoïde* est un ensemble muni d'une relation d'équivalence (une égalité).
- un *semi-groupe* est un *setoïde* muni d'une loi de composition interne (notée généralement $*$ ou $+$, quand elle est commutative, c'est à dire $x*y = y*x$). Cette loi est *associative*, c'est à dire qu'on a $(x*y)*z = x*(y*z)$. On note alors $x^n = \overbrace{x*...*x}^{n\ fois}$.
- un *monoïde* est un semi-groupe muni d'un élément neutre, 1 (ou 0 quand la loi est noté additivement), vérifiant $1*x = x*1 = x$.
- Un *groupe* est un monoïde où tous les éléments ont un inverse, c'est à dire $\forall x, \exists y, x*y = y*x = 1$. Cet y (unique) est noté $x^{(-1)}$ ($-x$ en notation additive).
- Un *anneau* est muni de deux lois de composition, $+$ et $*$ et est un groupe (commutatif) pour $+$ et un monoïde pour $*$. En outre, les deux lois vérifient les propriétés suivantes (distributivité de $*$ sur $+$) : $x*(y+z) = (x*y)+(x*z)$ et $(x+y)*z = (x*z)+(y*z)$.
- Si $\mathbb{A}$ est un anneau et X une variable, l'ensemble des polynômes $\mathbb{A}[X]$ est défini de la manière suivante :

  $$p = a_0 + a_1.X + \ldots + a_n.X^n$$

  où les a_i sont des éléments de $\mathbb{A}$ et n un entier (éventuellement nul). On définit le *degré* de p comme le plus grand entier i tel que a_i soit non nul, ou $-\infty$ si p est lui-même nul. La valeur de p en un élément x de $\mathbb{A}$ est

  $$p(x) = a_0 + a_1 * x + \ldots + a_n * x^n$$

  Cet ensemble peut être muni de deux opérations $+$ et $*$ qui en font un anneau. Avec $p_1 = a_0 + a_1.X + \ldots + a_n.X^n$ et $p_2 = b_0 + b_1.X + \ldots + b_m.X^m$, on a

  $$p_1 + p_2 = \sum_{i=0}^{\max(n,m)} (a_i + b_i).X^i$$

  $$p_1 * p_2 = \sum_{i=0}^{n+m} \left(\sum_{j=0}^{\max(i,m)} a_j * b_{i-j} \right) .X^i$$

- Comme $\mathbb{A}[X]$ est lui même un anneau, on peut construire l'anneau des polynômes dont les coefficients sont dans $\mathbb{A}[X]$, avec comme variable Y,

$(\mathbb{A}[X])[Y]$, et ainsi de suite. Notons que les structures obtenues en permutant l'ordre des variables ($(\mathbb{A}[X])[Y]$ et $(\mathbb{A}[Y])[X]$) sont isomorphes entre elles.

1.2 Entités

Les *entités* sont les objets sur lesquels on veut effectuer des calculs. Toutefois, il ne faut pas confondre l'objet abstrait manipulé en mathématique, et sa représentation concrète sur une machine. En effet, il y a bien souvent plusieurs manières de représenter un même objet mathématique. Par exemple, X^3+2 pourra être vu, en représentation "creuse", comme une liste de paires, `[(1,3);(2,0)]`, le premier élément de chaque paire représentant le coefficient, et le second le degré de chaque monôme. Mais on peut aussi utiliser une représentation "pleine", de la forme `[1,0,0,2]`. Ici, on ne donne pas explicitement les degrés, mais la liste des coefficients. Bien entendu, la forme des algorithmes utilisés variera énormément suivant le choix initial de la représentation.

Par ailleurs, certaines représentations ont à respecter des invariants. Ainsi, dans la représentation creuse, on peut exiger qu'il n'y ait aucun coefficient nul (c'est à dire par exemple refuser d'identifier `[(1,3);(0,2);(2,0)]` avec le polynôme $X^3+0X^2+2=X^3+2$), afin de ne manipuler que des listes les plus petites possible.

Enfin, à l'inverse, plusieurs entités mathématiques peuvent avoir la même représentation en machine : pour tout n inférieur à `MAXINT`, on peut représenter les entiers modulo n par des entiers machine. Malgré tout, on souhaite pouvoir distinguer les éléments de $\mathbb{Z}/_{2\mathbb{Z}}$ et ceux de $\mathbb{Z}/_{6\mathbb{Z}}$. Pour toutes ces raisons, il est nécessaire de considérer ces entiers machine à travers un certain degré d'abstraction. D'une part, il faut s'assurer que les différentes fonctions manipulent bien les entités pour lesquelles elles ont été prévues (ayant le bon type et respectant les invariants de représentation). En effet, $\mathbb{Z}/_{2\mathbb{Z}}$ est un corps, tandis que $\mathbb{Z}/_{6\mathbb{Z}}$ est un anneau : la multiplication, par exemple, n'y a pas les mêmes propriétés. D'autre part, il faut éviter que deux entités abstraites ayant la même représentation soient confondues. Il faut donc considérer un entier machine soit comme un élément de $\mathbb{Z}/_{2\mathbb{Z}}$, soit comme un élément de $\mathbb{Z}/_{6\mathbb{Z}}$.

1.3 Espèces et méthodes

Une espèce peut être vue comme un ensemble de *méthodes*, chacune d'entre elles pouvant être *déclarée*, c'est-à-dire abstraite, ou *définie*, c'est à dire posséder une implantation. Les méthodes sont de trois types différents :

- le *type support*, unique pour chaque espèce, représente l'ensemble où vivent les entités manipulées par les autres méthodes de l'espèce.

- les *fonctions* –quand elles sont définies– et les *signatures* –quand elles sont déclarées– représentent les opérations qu'on peut effectuer sur les éléments du type support.
- Enfin, les *théorèmes* et les *énoncés* permettent d'exprimer des *propriétés* sur les opérations. Un théorème possède une preuve et est donc considéré comme une méthode définie, tandis qu'un énoncé n'est que déclaré. Les énoncés représentent les axiomes de la structure dans laquelle on se trouve. Pour chaque implantation de cette structure, il faudra donc apporter la preuve qu'ils sont bien respectés.

Pour prendre un exemple simple, on peut examiner la spécification d'un groupe additif. Celui-ci est composé d'un certain nombre de méthodes déclarées, correspondant aux axiomes mathématiques des groupes. Cet ensemble de signatures et d'énoncés constitue la spécification proprement dite :

- le type support abstrait
- trois signatures :
  - la loi interne $+$,
  - l'élément neutre 0
  - l'opération d'inversion, $-$
- les propriétés des opérations précédentes[1] :
  - $\forall x,\ x + 0 = x$
  - $\forall x,\ x + (-x) = 0$
  - $\forall x, y, z,\ x + (y + z) = (x + y) + z$
  - $\forall x, y,\ (x + y) = (y + x)$

Par ailleurs, il est possible, à partir des méthodes précédentes, de définir des opérations, et de montrer certaines de leur propriétés. Par exemple, on peut définir la multiplication externe par un entier, $\mathbb{N}$ étant supposé connu (le statut exact de $\mathbb{N}$ n'est pas éclairci).

$$\begin{aligned} 0.x &= 0 \\ (n+1).x &= n.x + x \\ (-n).x &= -(n.x) \end{aligned}$$

et prouver, à l'aide de nos spécifications, le théorème suivant : $\forall x,\ 1.x = x$ (en utilisant la commutativité et la neutralité de 0).

1.4 Héritage

Ainsi qu'on l'a déjà dit, les espèces ne sont pas isolées les unes des autres, mais regroupées en une hiérarchie, qui exprime le fait qu'une structure mathématique est le plus souvent définie à partir de structures préexistantes, en ajoutant des opérations et des propriétés. Ce mécanisme s'apparente ainsi à l'*héritage* des langages orienté-objet. Une espèce s qui hérite d'une espèce

[1] On ne cherche pas ici à minimiser le nombre d'axiomes.

s_1 possède toutes les méthodes, déclarées comme définies de s_1. En outre s peut déclarer de nouvelles méthodes, définir des méthodes qui n'étaient que déclarées dans s_1, déclarer et définir en même temps une nouvelle méthode, ou enfin redéfinir une méthode déjà définie. Dans ce dernier cas, on peut être amené à effacer certaines preuves de théorèmes. En effet, une preuve peut s'appuyer sur la définition d'une fonction : si on change celle-ci, il faut également refaire la preuve. Nous reviendrons plus en détail sur ce point dans la section suivante, consacrée aux dépendances entre les différentes méthodes constituant une espèce.

En outre, il arrive fréquemment qu'on ait besoin d'*héritage multiple*, c'est à dire d'hériter de plusieurs espèces précédemment définies. Dans ce cas, il peut arriver que la même méthode existe dans deux espèces (voire plus). Il convient tout d'abord de vérifier que les déclarations (le type des opérations ou l'énoncé des propriétés) sont cohérentes dans chacune de ces espèces. De plus, s'il y a plus d'une définition, il faut en sélectionner une, ce qui peut, comme dans le cas des redéfinitions, amener à effacer des preuves.

Pour continuer l'exemple précédent, un anneau peut ainsi être défini comme héritant du groupe additif précédemment cité et de l'espèce des monoïdes (multiplicatifs). De plus, il faut ajouter les propriétés (déclarées) de distributivité. On peut par ailleurs donner une définition de 0, comme étant égal à $1 - 1$ (1 étant l'élément neutre pour la multiplication). Cette définition générique pourra être remplacée par des définitions plus spécialisées dans la suite de l'héritage (comme le `0` des entiers machine dans le cas d'une implantation de $\mathbb{Z}$).

1.5 Dépendances

De même que les espèces ont des liens étroits les unes avec les autres, les méthodes d'une espèce donnée peuvent s'appeler les unes les autres. Lorsqu'une méthode m_1 contient un appel à une méthode m_2 de l'espèce courante, on dira que m_1 *dépend* de m_2. De plus, la présence des propriétés conduit à distinguer deux types de dépendances. Lorsqu'on a seulement besoin de connaître le nom et le type de m_2, on parlera de *decl-dépendance*. Si au contraire, on a besoin de connaître aussi la définition de m_2 (ce qui suppose que m_2 est une méthode définie), on est en présence d'une *def-dépendance*.

Dans l'exemple précédent des groupes additifs, on peut ainsi remarquer que tous les énoncés decl-dépendent de $+$ (on a besoin de savoir que $+$ existe et que c'est une opération binaire pour que leur énoncé ait un sens). De même, la multiplication externe $(.)$ decl-dépend de $+$, ainsi que le théorème $\forall x, 1.x = x$. En revanche, ce même théorème def-dépend de $(.)$. En effet, pour pouvoir appliquer les propriétés de $+$, il faut commencer par utiliser la définition de $(.)$, afin de transformer $1.x$ en $(0.x) + x$, puis en $0 + x$.

Lors de l'héritage, les def-dépendances posent un problème particulier.

En effet, si la méthode m_2 dont def-dépend m_1 est redéfinie au cours d'un héritage, la définition de m_1 devient invalide dans le nouvel environnement. De ce fait, il faut effacer cette définition, tout en gardant la déclaration correspondante, dans la nouvelle espèce. Une nouvelle définition de m_1 devra ainsi être fournie dans la suite de l'héritage.

1.6 Paramètres

De nombreuses espèces sont définies par rapport à un ou plusieurs *paramètres*. Ces derniers peuvent appartenir à deux catégories distinctes. En premier lieu, on peut avoir des paramètres *d'entité* : l'espèce s prend pour paramètre un élément x appartenant à une espèce donnée. Les méthodes définies ou déclarées au sein de s peuvent alors faire référence à x. En second lieu, s peut être paramétrée par une espèce c dérivant d'une espèce donnée s'. Dans ce cas, les méthodes de s peuvent faire appel à toute méthode figurant dans s'. Par exemple, une espèce représentant les matrices carrées aura un paramètre d'entité n représentant ses dimensions, et un paramètre d'espèce a, représentant l'anneau des coefficient de la matrice.

Dans le cas d'un paramètre d'espèce c, on ne peut pas faire de supposition, dans s, sur la définition des méthodes de c. En effet, c est construite à partir de s' par une suite d'étapes pouvant comporter des redéfinitions de méthodes. Si une définition de s n'était correcte que pour une définition particulière d'une méthode m de s', on ne pourrait pas instancier c par n'importe quelle espèce dérivant de s'. Ce point oblige à introduire la notion d'*interface* d'une espèce, constituée uniquement des déclarations de l'espèce.

Inversement, comme toutes les méthodes de s' sont susceptibles d'être utilisées, il faut que toutes les méthodes du paramètre c soient définies. Toutes les instantiations de ce paramètre doivent respecter cette contrainte. Cela aboutit à introduire la notion de *collection*, espèce dont toutes les méthodes sont définies.

1.7 Interfaces et Collections

1.7.1 Interfaces

Comme on vient de le voir, une interface est une espèce dont toutes les méthodes ne sont que déclarées. À chaque espèce s doit donc pouvoir être associée une interface, obtenue en effaçant les définitions présentes dans s. Ce n'est toutefois pas toujours possible si on accepte des def-dépendances dans les "types" des méthodes, c'est à dire le type d'une opération ou l'énoncé d'une propriété. Ainsi, si on considère une espèce s possédant les méthodes suivantes :

1. un type support *rep* défini comme le type `int`

2. une fonction *inc*, de type *rep* → *rep*, définie par `fun x -> x+1`
3. un théorème *inc_spec*, montrant la propriété : $\forall x \in rep$,`inc(x)`$\geq$ `x+1`

L'énoncé de *inc_spec* n'a de sens que si on sait que *rep* est un synonyme de `int`. Dans le cas contraire, `inc(x)`, de type *rep*, et `x+1`, de type `int` n'ont pas le même type, et n'ont donc a fortiori aucune raison d'être comparables. Afin d'éviter cela, on impose qu'il n'y ait de def-dépendance que dans la définition d'une méthode (fonction ou théorème), et aucune dans sa déclaration. En pratique, cette restriction ne semble toutefois pas gênante. En effet, la plupart des propriétés qu'on trouve dans une espèce utilisent naturellement la vision abstraite du type support. C'est en particulier le cas pour toutes les propriétés héritées d'espèces dans lesquelles le type support n'est pas défini, ce qui est le cas plupart des propriétés mathématiques classiques. Seuls quelques lemmes "techniques" portant sur des fonctions de très bas niveau pourraient être affectés par ce problème. Mais même dans ce cas, il suffirait d'ajouter à l'espèce des méthodes de conversion, de type *rep* → *t* ou *t* → *rep*, où *t* est la définition concrète du type support. Ainsi, en ajoutant à notre espèce *s* ci-dessus la méthode *to_int*, de type *rep* → *int*, définie par `fun x -> x`, on peut réécrire l'énoncé de *inc_spec* sans def-dépendance : $\forall x \in rep$,`to_int(inc(x))`$\geq$ `to_int(x)+1`.

1.7.2 Collections

Les collections jouent le rôle dual des interfaces : il s'agit d'espèces dont toutes les méthodes sont définies. Elles représentent donc des structures mathématiques bien définies, où on peut mener des calculs effectifs, *et* où toutes les propriétés des opérations ont été prouvées. On impose en outre deux restrictions aux collections :

- On ne peut pas hériter d'une collection. Celle ci représentant une structure mathématique particulière, elle ne peut en effet pas être raffinée.
- Une collection est vue à travers l'interface sous-jacente. On peut appeler des méthodes de la collection, mais on n'en connaît pas la définition –sinon à travers les spécifications correspondantes. C'est cette abstraction, et en particulier l'abstraction du type support, qui permet d'assurer que les méthodes d'une collection manipulent bien les entités pour lesquelles elles sont prévues (voir la section 1.2).

Une collection est donc définie comme l'encapsulation d'une espèce complètement définie, encapsulation qui donne à cette structure la puissance d'abstraction des types abstraits de certains langages, comme les modules, les types abstraits algébriques, etc. L'utilisateur d'une collection ne peut accéder aux définitions de cette collection, et en particulier n'a aucune connaissance de la représentation. Il ne peut manipuler les entités qu'à l'aide des méthodes de la collection, ce qui évite toute rupture des invariants de la représentation et concourt à la sûreté du programme.

Chapitre 2

Le langage Foc

2.1 Syntaxe

On définit dans cette section la syntaxe du langage Foc reconnu par le compilateur `focc`. Les chapitres suivants feront référence à un sous-ensemble de cette syntaxe qui n'en diminue pas le pouvoir expressif (voir le chapitre 3).

2.1.1 Espèces et collections

Définition d'une espèce ou d'une collection

spec	::=	**species** *ident* [(*prm* [{ ,*prm* }*])]
		[**inherits** *spcdef* [{ ,*spcdef* }*]]=
		{ *fields* ; }* **end**
coll	::=	**collection** *ident* **implements** *spcdef* =
		{ *conc_fields* ; }* **end**
prm	::=	*ident* **in** *type*
	|	*ident* **is** *spcdef*
spcdef	::=	*ident*
	|	*ident* (*expr*[{ ,*expr* }*])

La définition d'une espèce peut se décomposer en trois grandes parties :

- La liste de ses paramètres.
- La liste des espèces dont elle hérite.
- La liste des méthodes qu'elle déclare, définit ou redéfinit. On appelera cette dernière partie le *c*orps de l'espèce.

spcdef permet de considérer des espèces paramétrées dans le typage des paramètres ou l'héritage. Cependant tous les paramètres de ces espèces doivent être instantiés.

La définition d'une collection suit le même schéma, mais une collection ne peut avoir de paramètre, et n'implante qu'une unique espèce. De plus, le corps d'une collection se compose uniquement de méthodes définies : il ne peut y avoir de déclaration à ce niveau.

Les champs

```
conc_field ::= let ident [ (typed_id [ { ,typed_id }* ]) ]
                 [ in type ] = expr
             | let rec ident (typed_id [ { ,typed_id }* ])
                 [ in type ] = expr
             | theorem ident = prop
                 proof :
                 [ def : { ident }* ; ] [ decl : { ident }* ;script ; ]
                   proof
             | proof of ident =
                 [ def : { ident }* ; ] [ decl : { ident }* ;script ; ]
             | letprop ident (typed_id [ { ,typed_id }* ]) =
                 prop ; ;
             | rep = type
field  ::= conc_field
         | sig ident in type
         | property ident = prop
         | rep
```

La preuve d'un théorème est un script de preuve Coq, qui n'est pas analysé par le compilateur Foc, mais sera vérifié par Coq dans l'environnement correspondant à la traduction de l'espèce proprement dite. Pour que cet environnement soit correct il faut fournir à Foc les listes des méthodes dont dépend la démonstration, et préciser le type de dépendance (def- ou decl-). **proof of** est un raccourci permettant de ne pas redonner l'énoncé de la propriété héritée d'une espèce parent qu'on veut prouver dans l'espèce courante.

Par ailleurs, il faut faire attention aux types que Foc est capable d'inférer : à l'intérieur d'une collection (ou d'une espèce), il identifie **self** avec sa définition concrète. Ce n'est plus le cas lors des appels de méthodes à l'extérieur de la collection. Toutefois, des problèmes ne sont susceptibles de survenir que lors de la définition de méthodes qui n'ont pas été précédemment déclarées. Par défaut, Foc utilise le type-support "en clair" (i.e. sa version concrète). Ainsi, si on prend l'espèce suivante, où `#int_plus` est l'addition sur les entiers Foc (type **int**) :

```
species s =
  rep = int ;
  let f(x in self) = #int_plus(x, 1) ;
end
```

le type de `f` sera **self** → `int`. En effet, lors du typage, du corps de `f`, `x` a le type **self**. Pour que l'application de `#int_plus` soit correcte, il faut unifier **self** avec `int`. Cette unification est possible dans le contexte de l'espèce *s*

dont le type support est bien `int`. Par contre, comme le type de retour de `f` n'est pas spécifié, on garde le type de retour de `#int_plus`, soit `int`.

2.1.2 Définitions de fonctions et de théorèmes

typed_id ::= *ident* **in** *type*
| *ident* type à inférer

def ::= **let** *ident*[(*typed_ id* [{ ,*typed_ id* }*])][**in** *type*]= *expr* ; ;
| **let rec** *ident*(*typed_ id* [{ ,*typed_ id* }*])[**in** *type*]= *expr* ; ;
| **theorem** *ident* = *prop* **proof :** *script* ; ;
| **attach** *ident ident* **proof :** *script* ; ;
| **letprop** *ident* (*typed_ id* [{ ,*typed_ id* }*]) = *prop* ; ;

La dernière construction permet de définir un prédicat (l'équivalent d'une fonction, mais le type de retour est `Prop`), qui ne sera présent que dans la traduction Coq. **attach** `foc_name coq_name` permet de montrer l'équivalence entre une définition Foc, et une définition préexistante en Coq. Le script de preuve doit permettre à Coq de vérifier que la traduction de la définition de `foc_name` est bien équivalente à la définition de `coq_name`.

2.1.3 Types concrets

typedef ::= **type** *ident* [{ *ident* }*] = { lstcons }*
lstcons ::= **caml_link** *ident*
| **coq_link** *ident*
| *ident* [(*ident*[,*ident*])] **in** *type*

S'il y a plusieurs **caml_link** (ou **coq_link**), associés à un même identificateur Foc, seul le dernier est pris en compte. Ces constructions permettent d'importer des types concrets préexistants en Ocaml (et en Coq), et donc d'avoir accès aux fonctions déjà construites sur ces types.

2.1.4 Types

type ::= *ident* type atomique
| '*ident* variable de type
| *type*→*type*
| (*type***type*)
| *ident*(*type*) type paramétré
| (*type*)
| **Prop** propositions logiques
| **self**

Les types atomiques sont les types supports des collections et les types concrets définis en Foc. **self** ne peut être utilisé qu' à l'intérieur d'une définition d'espèce ou de collection, et représente le type support de la collection courante.

2.1.5 Expressions

expr	::=	*ident*	variable locale
	|	[*ident*]#*ident*	variable globale
	|	*ident* !*ident*	appel de méthode
	|	**self** !*ident*	
	|	**let** *ident* [(*typed_ id* [{ ,*typed_ id* }*])] [**in** *type*] = *expr* **in** *expr*	
	|	**let rec** *ident* (*typed_ id* [{ ,*typed_ id* }*]) [**in** *type*] = *expr* **in** *expr*	
	|	*expr* (*expr*[{ ,*expr* }*])	application
	|	**fun** *ident* [**in** *typ*] → *expr*	fonction anonyme
	|	**if** *expr* **then** *expr* **else** *expr*	
	|	**match** *expr* **with** { |{ *ident* }*→*expr* }* **end**	pattern-matching
	|	**caml** *ident*	
	|	**caml** *ident* **with coqdef** *ident*	
	|	(*expr*)	

On ne peut effectuer que des pattern-matching simples et exhaustifs (c'est à dire sur un terme d'un type concret de Foc, en donnant exactement un cas par constructeur). les deux règles pour **caml** permettent de donner le nom d'une fonction Ocaml (et éventuellement de son équivalent Coq). Ces noms doivent correspondre à une entrée dans un fichier `.fml` (cf 2.1.7) de même nom que le fichier `.foc` en cours d'analyse.

Une variable globale est préfixée par le nom du fichier dans lequel elle est définie. Celui-ci doit avoir été chargé par un `uses` au préalable. Les variables globales du fichier courant ne sont pas préfixées. De même, si on a en plus utilisé `open`, le préfixe est facultatif. Cette directive doit toutefois être utilisée avec précaution. En effet, Foc n'autorise pas la duplication des noms de variables globales. Ainsi, si on fait un `open` sur un fichier qui contient la définition de `f`, il ne sera pas possible de définir une nouvelle fonction `f`.

2.1.6 Propositions logiques

prop	::=	**all** *ident*[,{ *ident* }*] **in** *type*, *prop*
	|	**ex** *ident*[{ *ident* }*] **in** *type*, *prop*
	|	*prop* **and** *prop*
	|	*prop* **or** *prop*
	|	*prop* → *prop*
	|	**not** *prop*
	|	(*prop*)
	|	*expr*

Pour être considérée comme une proposition, une expression doit avoir le type **Prop** ou `bool`. Dans le second cas, on utilise une coercion implicite vers le type **Prop** (voir la section 3.9).

2.1.7 Le fichier fml

deffml	::=	**coq header** {* string *}
	|	**ocaml header** {* string *}
	|	**mli header** {* string *}
	|	**import for** *ident* { *lstfct* }* **end import**
lstfct	::=	*ident* [{ *ident* }*] = {* string *} [**link** {* string *}]

Ce fichier permet d'écrire des fonctions directement en OCAML(et éventuellement en COQ). On distingue les définitions par l'espèce à laquelle elles appartiennent, qui peut être le mot clé `toplevel` pour une définition en dehors d'une espèce ou collection.

2.2 Un exemple complet

Afin de présenter plus concrètement les principales constructions de FOC, on va construire une micro-hiérarchie de structures mathématiques qui permettront de mettre en lumière les différents points qu'on vient d'évoquer dans la syntaxe.

2.2.1 Bref aperçu mathématique

On veut définir en FOC le produit $\mathbb{Z} \times \mathbb{Z}$. Pour cela, on se fixe la hiérarchie suivante :

1. les setoïdes : ensembles avec égalité (`egal`).
2. les monoïdes : des setoïdes munis d'une loi binaire (`multiplie`) et d'un élément `un` neutre pour cette loi.
3. On construit alors $\mathbb{Z}$ comme un monoïde sur les entiers.
4. le produit cartésien de deux setoïdes quelconques, qui est un setoïde.

5. le produit cartésien de deux monoïdes, qui peut être muni d'une structure de monoïde, et qui possède les mêmes propriétés que le produit de deux setoïdes.
6. $\mathbb{Z} \times \mathbb{Z}$ peut alors être défini.

Pour faire cela, on va utiliser toutes les constructions de base de FOC, en particulier :

- définitions d'espèces (correspondant aux setoïdes, monoïdes, ...) et de collections ($\mathbb{Z}$ et $\mathbb{Z} \times \mathbb{Z}$).
- différence entre les méthodes déclarées et celles qui sont définies dans les espèces.
- héritage (de setoïde à monoïde), et héritage multiple (pour le produit cartésien de monoïde).
- espèces paramétrées (pour les produits cartésiens)

2.2.2 Le code

```
uses basics;;
```

Tout d'abord on signale qu'on va utiliser les définitions contenues dans le fichier `basics.foc`. Ce fichier contient des définitions de fonctions de base sur les entiers et les booléens, et l'espèce de base de la hiérarchie, intitulée `basic_object`.

```
open basics;;
```

Normalement, tous les noms de fonctions définies dans un autre fichier doivent être préfixés par le nom du fichier en question : `fichier#f`. La directive **open** permet d'écrire seulement `#f`. Toutefois, il faut la manipuler avec précaution, car l'identificateur `f` doit être unique : si `fichier1` et `fichier2` définissent chacun une fonction `f`, on ne pourra pas ouvrir les deux à la fois.

Nous sommes maintenant prêt à définir les espèces nécessaires. Commençons par `setoide`.

```
species setoide inherits basic_object =
```

L'espèce hérite de la racine de la hiérarchie, `basic_object`, qui contient trois méthodes : le type support, **rep**, et deux "utilitaires" informatiques, `parse` et `print`. `parse` convertit une chaîne de caractère (élément de type `string`) en un élément du type support. Sa définition par défaut est de renvoyer une erreur quel que soit son argument. `print` convertit un élément du type support en `string`. Par défaut, elle renvoit toujours la chaîne `"<abst>"`.

Outre ces méthodes héritées, `setoide` en introduit d'autres qui lui sont propres :

```
  sig egal in self → self → bool ;
  sig element in self ;
  let different (x,y) = basics#not_b( !egal(x,y)) ;

  property refl = all x in self, !egal(x,x) ;
  property symm = all x,y in self, !egal(x,y) → !egal(y,x) ;
end
```

`egal` et `element` sont déclarées : on sait seulement qu'elles existent dans tous les setoïdes, mais on n'en a pas de définition explicite. Celle-ci devra être donnée plus tard au cours de l'héritage.

Au contraire, `different` peut être définie, en fonction de la méthode `egal`. L'appel de méthode est de la forme `collection !meth`. Quand on appelle une méthode de **self**, cela peut être abrégé en `!meth`.

Enfin, on énonce deux propriétés que doit vérifier `egal`.

On définit ensuite les monoïdes :

```
species monoide inherits setoide =
  sig multiplie in self → self → self ;
  sig un in self ;
  let element = !multiplie( !un, !un) ;
end
```

Deux nouvelles méthodes sont déclarées. De plus `element` peut maintenant recevoir une définition. Lors de l'analyse du code, le typeur vérifiera que la définition a bien le type qu'on avait donné dans la déclaration de `element`.

Le produit cartésien de deux `setoides` introduit une nouvelle construction :

```
species setoide_produit(a is setoide, b is setoide)
```

Ici, l'espèce `setoide_produit` prend deux paramètres `a` et `b`, qui sont des collections implantant `setoide`, c'est à dire qu'elles en possèdent toutes les méthodes, avec le type correspondant.

```
  inherits setoide =

  rep = a * b ;
```

Le type support est défini comme le produit des types support de `a` et `b`. Les définitions des autres méthodes font appel aux fonctions Foc sur les paires : `#crp` pour la création, et `#first` et `#scnd` pour les deux projections.

```
  let egal(x,y)=
    #and_b(a !egal(#first(x),#first(y)),
           b !egal(#scnd(x),#scnd(y))) ;
```

De même, `egal` est défini en fonction de l'égalité de `a` et de celle de `b`.

```
let element = self !creer(a !element,b !element) ;

let creer(x,y) in self = #crp(x,y) ;
```

Remarquons ici que `element` utilise `creer`, qui est introduit après. Lors de l'analyse de l'espèce tout sera remis en ordre.

```
let print = fun x →
  #sc("(",
  #sc(a !print(#first(x)),
  #sc(",",
  #sc(b !print(#scnd(x)),
     ")"))))) ;

proof of refl=
  def : egal ;
  assumed
;
proof of symm=
  def : egal ;
  assumed
;
```

end

`#sc` est une fonction de `basics` réalisant la concaténation de deux chaînes de caractères. Le mot-clé **assumed** qui remplace les scripts de preuve signale que les théorèmes sont en fait admis.

Enfin, on crée les produits cartésiens de `monoide`.

```
species monoide_produit(a is monoide, b is monoide)
   inherits monoide, setoide_produit(a,b) =
```

Ici, on trouve un héritage multiple. Des problèmes spécifiques se posent. En particulier, `element` est défini dans chacune des deux espèces parent. Il faut donc choisir celle qui sera retenue pour `monoide_produit`. Pour cela, c'est l'ordre des espèces dans la clause `inherits` qui est pris en compte : on prend la définition venant de l'espèce la plus à droite, soit ici `setoide_produit`.

```
let un = !creer(a !un,b !un) ;

let multiplie(x,y)=
  #crp(a !multiplie(#first(x),#first(y)),
     b !multiplie(#scnd(x),#scnd(y))) ;
```

end

Enfin, on définit les collections. Contrairement aux espèces, une collection n'a pas le droit d'avoir des méthodes déclarées : tout doit être défini.

```
collection entiers implements monoide =
      rep = int;
      let un = 1;
      let multiplie = basics#int_mult;
      let egal = basics#base_eq;
      let print = basics#string_of_int;
      let parse = basics#int_of_string;

      proof of refl=
        assumed;
      proof of symm=
        assumed;
end
```

La définition de la collection correspondant à $\mathbb{Z} \times \mathbb{Z}$ consiste simplement à instancier les paramètres de `monoide_produit` par `entiers`. En effet, l'espèce `monoide_produit` a toutes ses méthodes définies.

```
collection entiers_2
  implements monoide_produit(entiers,entiers)= end
```

On peut maintenant manipuler nos deux collections à travers quelques exemples simples :

```
let cinq = entiers!parse("5");;
let cinq_carre = entiers!multiplie(#cinq,#cinq);;
let foo = entiers_2!creer(#cinq,#cinq_carre);;
let foo2 = entiers_2!multiplie(#foo,#foo);;
#print_string(entiers_2!print(#foo2));;
```

Chapitre 3

Analyse statique d'un programme Foc

3.1 Contraintes à respecter par un programme Foc

Toutes les définitions syntaxiquement correctes écrites en Foc ne sont pas acceptables d'un point de vue sémantique. En effet, il faut s'assurer que les algorithmes définis au sein d'une espèce donnée vérifient les propriétés présentées dans le chapitre 1. Pour ce faire, nous avons défini un certain nombre de contraintes que les espèces de Foc doivent respecter.

Par ailleurs, comme nous voulons être capables de faire des preuves des différents algorithmes écrits en Foc (cf. section 3.9), nous avons été amenés à imposer d'autres contraintes, qui ne sont pas forcément strictement nécessaires pour assurer la correction des programmes, mais qui permettent de simplifier l'expression des propriétés à démontrer (dans les preuves de terminaison en particulier), et donc l'écriture des preuves correspondantes.

Les contraintes qu'un programme Foc doit respecter et que nous détaillerons plus formellement dans le reste de ce chapitre sont les suivantes :

- Typage : toutes les expressions doivent être bien typées. Les arguments donnés aux espèces paramétrées doivent avoir le type (ou l'interface) attendu. Enfin, la redéfinition d'une méthode au cours de l'héritage ne doit pas en changer le type.
- De même, le type support ne peut changer au cours de l'héritage (cette condition est en fait liée à la traduction vers des classes de Ocaml).
- Lorsqu'on crée une collection à partir d'une espèce, toutes les méthodes de l'espèce doivent être définies (et non simplement déclarées)
- Le type support – **rep** – doit être présent dans chaque espèce, qu'il soit seulement déclaré ou lié à un type concret (hérité ou défini de manière explicite dans l'espèce elle-même).
- Il ne doit pas y avoir de récursion mutuelle entre méthodes en dehors d'un champ **let rec**

Héritage et type de méthode La contrainte sur les redéfinitions de méthode permet de s'assurer que toute collection implantant une espèce héritant d'une espèce donnée s pourra être utilisée pour instancier un paramètre devant avoir l'interface de s. En particulier, l'héritage permet dès lors de raffiner les paramètres des espèces, comme dans l'exemple suivant :

species a = **rep** ; ... **end**
species b **inherits** a = ... **end**
species a_1 (`coll_a` **is** a) = ... **end**
species b_1 (`coll_b` **is** b) **inherits** a_1(`coll_b`)

Le paramètre `coll_b` de b_1 ne peut être appliqué à a_1, qui attend une collection d'interface `a`, que si on est sûr que l'héritage préserve les types des méthodes.

Récursion mutuelle Imposer au programmeur de marquer explicitement tous les groupes de méthodes mutuellement récursives – *via* la construction **let rec** – peut sembler un peu curieux dans un contexte de langage orienté-objet. En effet, les méthodes d'un objet peuvent en général s'appeler les unes les autres sans restriction [RV98].

Ici, nous voulons être capables non seulement d'écrire des algorithmes, mais aussi de prouver des propriétés sur le code. En premier lieu, il faut pouvoir montrer que tous les appels de méthode terminent. Or si on suppose que toutes les fonctions de l'espèce peuvent participer à la récursion, on compliquera énormément une telle preuve, le plus souvent inutilement. Au contraire, si le programmeur lui-même indique les groupes de méthodes qui sont impliqués dans une vraie récursion, ces groupes devraient rester aussi petits que possible (et même se réduire la plupart du temps à une seule méthode récursive), ce qui allégera d'autant les preuves de terminaison.

Il est important de noter que cette restriction implique une analyse globale de *toutes* les méthodes d'une espèce donnée, qu'elles soient héritées ou définies au sein de l'espèce. Par exemple, si on considère les espèces a, b, et `c` définies de la manière suivante :

species a = **rep** ; **sig** `x` **in self** ; **let** `y` = **self** !`x` ; **end**
species b = **rep** ; **let** `x` = **self** !`y` ; **sig** `y` **in self** ; **end**
species `c` **inherits** a,b = **rep** = `int` ; **end**

a et b sont manifestement bien définies. Dans l'espèce a, la méthode `y` dépend de la méthode `x`, tandis que c'est le contraire dans b. Aucune de ces deux espèces ne contient de cycle de dépendance. Intéressons nous maintenant à l'espèce `c`. Elle hérite de deux espèces sans cycle de dépendance et définit une seule méthode nouvelle, à savoir son type support. Cette définition n'apporte aucune dépendance nouvelle, et on pourrait donc penser que `c` elle même est bien définie. Toutefois, la définition de la méthode `x` de `c`

est héritée de a, et dépend donc de la méthode `y` de `c`. La définition de cette dernière est héritée de b, et dépend ainsi de la méthode `x`. Ainsi, bien que l'espèce `c` hérite de deux espèces bien définies et n'ajoute aucune dépendance supplémentaire, elle contient, par le jeu d'un héritage multiple, un cycle de dépendance.

Inversement, il paraît difficile d'interdire toute forme de récursion mutuelle. On peut vouloir écrire le code suivant par exemple :

```
species odd_and_even =
 rep = int;
 let rec odd (x in self) =
   if x = 0 then false else self!even(x-1)
 and    even(x in self) =
   if x = 0 then true else self!odd(x-1);
end
```

Ici, la présence d'un **let rec** signale que le programmeur a bien l'intention de créer un cycle de dépendances. Une future version du compilateur devrait en outre demander d'apporter des preuves que chaque appel récursif se fait bien sur un argument "plus petit" (pour un ordre qu'il faudra aussi préciser : voir la section 11.1.1). Une fois que ces preuves auront été faites, il sera tout à fait correct d'utiliser les fonctions ainsi définies.

Analyser les dépendances D'un point de vue strictement calculatoire, l'objet principal de l'analyse de dépendances menée en Foc consiste à rejeter le premier exemple et à accepter le second. Les choses deviendront plus complexes lorsqu'on s'intéressera aux propriétés et aux théorèmes, comme nous le verrons dans la section 3.9.

En résumé, l'analyse d'une espèce s'effectue autour de trois grands axes :

- L'héritage et en particulier la résolution des "conflits" en cas d'héritage multiple.
- L'analyse de dépendance et la détection de cycle.
- Le typage des méthodes définies.

3.2 Définitions de base

Afin d'étudier plus formellement l'analyse de dépendances telle qu'elle est faite en Foc, nous allons étudier un sous-langage de celui qui a été introduit dans le chapitre 2.1. En effet, nous ne considérerons pas le pattern-matching ni le **if-then-else**. Ces deux constructions n'interviennent pas directement dans le calcul des dépendances, et on peut les modéliser en ajoutant les collections adéquates dans l'environnement de typage que nous allons décrire plus loin. On ne perd donc pas en expressivité en n'utilisant pas ces constructions. De plus, on élaborera cette analyse en deux temps. En premier lieu, on

ne considérera que les champs calculatoires (**let rec**, **let** et **sig**), tandis que la section 3.9 étend l'analyse de dépendances aux propriétés et aux preuves. Ce découpage permet une introduction relativement simple des concepts d'espèce bien formée (def.17) et d'espèce en forme normale (def.18) ainsi que des principales propriétés de ces deux notions avant de voir comment il est possible de les élargir dans un contexte de propriétés et de preuves, avec en particulier la distinction entre *def-* et *decl* dépendances dont nous avons déjà parlé précédemment (section 1.5).

Par ailleurs, avant de décrire l'analyse de dépendance proprement dite, nous allons introduire quelques définitions de base à partir de ce langage. Tout d'abord, nous définissons $\mathcal{N}(\phi)$, la liste des noms de méthodes introduites dans un champ d'une espèce, et $\mathcal{D}(\phi) \subseteq \mathcal{N}(\phi)$ les noms introduits par un champ défini (**l**et ou **let rec**). Ces définitions sont ensuite étendues aux espèces, par induction sur le graphe d'héritage. Ce dernier est en effet acyclique (on ne peut hériter que d'une espèce existante), assurant ainsi la correction de la définition.

Définition 1 (Noms introduits par un champ) *Étant donné un champ ϕ, $\mathcal{N}(\phi)$ et $\mathcal{D}(\phi)$ sont définis de la façon suivante :*

$$\begin{array}{rcl} \mathcal{N}(\textbf{\textit{let}}\ x = e) = \mathcal{N}(\textbf{\textit{sig}}\ x\ \textbf{\textit{in}}\ \tau) & = & \{x\} \\ \mathcal{N}(\textbf{\textit{let rec}}\ \{x_1 = e_1; \ldots; x_n = e_n\}) & = & \{x_i\}_{i=1\ldots n} \\ \mathcal{N}(\textbf{\textit{rep}}) = \mathcal{N}(\textbf{\textit{rep}}\ =\ \tau) & = & \textit{rep} \end{array}$$

$$\begin{array}{rcl} \mathcal{D}(\phi) & = & \emptyset \quad si\ \phi = \left\{ \begin{array}{c} \textbf{\textit{sig}}\ s\ \textbf{\textit{in}}\ \tau \\ \textbf{\textit{rep}} \end{array} \right. \\ & = & \mathcal{N}(\phi) \quad sinon \end{array}$$

Avec cette définition, on peut donner une première contrainte sur l'écriture d'une espèce : deux champs différents d'une espèce doivent introduire des noms distincts deux à deux. D'un point de vue pratique, donner deux définitions d'une même méthode à ce niveau est une erreur, puisque seule une d'entre elles sera effectivement utilisée et masquera l'autre. Nous supposerons donc dans la suite de ce chapitre que toutes les espèces considérées sont bien spécifiées.

Définition 2 (Espèce bien spécifiée) *Soit s une espèce définie de la manière suivante :*

species s **inherits** $s_1,\ldots,s_{h_f} = \phi_1,\ldots,\phi_n$ **end**

s est dite bien spécifiée si et seulement si la conditions suivante est vérifiée :

$$\forall i, j \leq n, i \neq j \Rightarrow \mathcal{N}(\phi_i) \cap \mathcal{N}(\phi_j) = \emptyset$$

Notation 1 *Dans la suite, nous adopterons la même notation que ci-dessus pour les champs et les espèce parent d'une espèce bien spécifiée s :*

- *Les champs seront notés Φ_i, avec i variant de 1 à n*
- *Les espèce dont s hérite seront notées s_h, pour h variant de 1 à h_f.*

Dans ce contexte, on définit l'ensemble des noms introduits dans une espèce comme l'union des noms provenant de ses parents et des noms introduits dans les champs propres à l'espèce :

Définition 3 (Noms des méthodes d'une espèce)

$$\mathcal{N}(s) = \left(\bigcup_{h=1}^{h_f} \mathcal{N}(s_h)\right) \cup \left(\bigcup_{i=1}^{n} \mathcal{N}(\phi_i)\right)$$

$$\mathcal{D}(s) = \left(\bigcup_{h=1}^{h_f} \mathcal{D}(s_h)\right) \cup \left(\bigcup_{i=1}^{n} \mathcal{D}(\phi_i)\right)$$

Ensuite, on définit, pour tout x dans $\mathcal{N}(s)$ le "corps" (**B**ody) de la méthode x dans s, qu'on note $\mathcal{B}_s(x)$. On utilisera le symbole $\perp$ pour indiquer que x n'est que déclaré. Si x est définie dans un des champs de s, on retiendra cette définition, comme le veut le principe de la liaison tardive. Dans le cas contraire, si x est héritée d'une espèce s_h, on prendra $\mathcal{B}_s(x) = \mathcal{B}_{s_h}(x)$. S'il y a plusieurs définitions, on prendra celle qui vient en dernier dans l'ordre donné par la clause **inherits**.

Définition 4 (Corps d'une méthode dans une espèce) *Soit $x \in \mathcal{N}(s)$ un nom de méthode d'une espèce s bien spécifiée.*

$$[\textsc{Decl}]\ \frac{x \notin \mathcal{D}(s)}{\mathcal{B}_s(x) = \perp} \qquad [\textsc{RepDef}]\ \frac{\exists i \leq n,\ \phi_i = \textbf{\textit{rep}} = \tau}{\mathcal{B}_s(rep) = \tau} \qquad [\textsc{LetDef}]\ \frac{\exists i \leq n,\ \phi_i = \textbf{\textit{let}}\ x = expr}{\mathcal{B}_s(x) = expr}$$

$$[\textsc{RecDef}]\ \frac{\exists i \leq n,\ \phi_i = \textbf{\textit{let rec}}\ \{x_1 = e_1; \ldots; x_m = e_m\}}{\mathcal{B}_s(x_j) = e_j}\ \forall j \leq m$$

$$[\textsc{Inh}]\ \frac{x \notin \bigcup_{i=1}^{n} \mathcal{D}(\phi_i) \qquad h_0 = \max\{h | x \in \mathcal{D}(s_h)\} \qquad \mathcal{B}_{s_{h_0}}(x) = e}{\mathcal{B}_s(x) = e}$$

Par définition de $\mathcal{D}(s)$ il n'y a pas d'autre cas.

En théorie, il serait possible d'utiliser une définition plus générale pour l'héritage multiple, et en particulier de laisser l'utilisateur choisir l'espèce s_h dont il veut prendre la définition. Pour cela, il suffit de modifier le choix de h_0 dans la définition ci-dessus, pour permettre de sélectionner un indice h quelconque tel que $\mathcal{B}_{s_h}(x) \neq \bot$. Cela n'a pas été retenu dans l'implantation de FOC pour trois raisons pratiques :

- Il n'y a pas encore d'exemple dans la librairie de FOC où ce choix plus large est nécessaire.
- Cela compliquerait la traduction en OCAML, alors que la politique retenue actuellement est exactement la même que celle de OCAML
- Lorsqu'on ajoute les preuves dans le calcul de dépendances (sec. 3.9), des problèmes de cohérence des choix de l'utilisateur pourraient se poser (il ne doit pas pouvoir choisir une fonction f venant de s_1 et une preuve de correction de f venant de s_2 par exemple)

Si on ne tenait pas compte des dépendances, nous pourrions presque arrêter l'analyse de dépendances ici. Il n'y aurait en effet plus qu'à typer $\mathcal{B}_s(x)$, comme dans Objective Foc [Fec01]. Le typage du corps des méthodes est décrit dans la section suivante. Cette analyse seule serait largement suffisante pour produire du code OCAML correct. Toutefois, on ne tient pas ici compte du mot clé avec lequel une méthode est introduite (**let** ou **let rec**). Or comme nous l'avons déjà dit, considérer que toutes les méthodes font implicitement partie d'une grosse structure récursive rendrait les preuves (et en particulier les preuves de terminaison) inutilement compliquées. Dans la section 3.5, on décrit l'analyse qui permet de vérifier que tous les cycles de dépendances sont bien introduits dans un **let rec**. Par ailleurs, le chapitre 8 montre comment la prise en compte des résultats de cette analyse permettent de définir une traduction des espèces FOC en OCAML ne reposant pas sur les objets de OCAML.

3.3 Espèce bien typée

Les méthodes des espèces et des collections ne sont pas polymorphes. Afin d'obtenir de la généricité, nous utiliserons les espèces paramétrées, dont les règles de formation seront données dans la section 3.8). En effet, autoriser un polymorphisme sans restriction à ce niveau amène des problèmes de

consistance, comme le montre l'exemple suivant :

```
species poly = rep ; let id = fun x → x ; end

species id_bool (x is poly) =
  rep = unit ;
  let elt = x!id(true) ;
end

species id_int inherits poly =
  rep = unit ; let id = fun x → x + 1 ;
end

collection coll_int implements id_int

collection error implements id_bool(coll_int)
```

Dans l'espèce `poly`, la méthode `id` a comme type le schéma $\forall\alpha.\alpha \to \alpha$. L'espèce `id_bool`, qui prend une implantation `x` de `poly` en paramètre utilise `x!id` avec le type `bool->bool`. De son côté, `id_int`, qui hérite de `poly` redéfinit `id` en lui donnant le type `int->int`. On a là deux instances parfaitement correctes de $\forall\alpha.\alpha \to \alpha$. Toutefois, il est impossible d'utiliser `coll_int`, la collection bâtie au dessus de `id_int` pour instancier le paramètre de `id_bool` comme tente de le faire la collection `error`. En effet, dans cette collection `error!elt` s'evalue en `true + 1`, ce qui devrait être rejeté.

En revanche, les variables locales du corps d'une méthode peuvent très bien être polymorphes. On définit donc classiquement $\mathcal{F}(\tau)$, l'ensemble des variables de type apparaissant libres dans le type τ ou le schéma de type $\forall\alpha_i.\tau$ de la manière suivante :

Définition 5 (Variables libres dans un type)

$$\mathcal{F}(c) = \emptyset \qquad \mathcal{F}(\alpha) = \alpha \qquad \mathcal{F}(\tau_1 \to \tau_2) = \mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2)$$

$$\mathcal{F}(\tau_1 * \tau_2) = \mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2) \qquad \mathcal{F}(\forall\alpha_i.\tau) = \mathcal{F}(\tau)\backslash\{\alpha_i\}$$

Par extension, si Γ *est un environnement de typage, on notera :*

$$\mathcal{F}(\Gamma) = \bigcup_{x\in\Gamma} \mathcal{F}(\Gamma(x))$$

Définition 6 (Type concret) *Un type* τ *est dit* concret *si et seulement si*
- *ce n'est pas un schéma de type*
- $\mathcal{F}(\tau) = \emptyset$

Notons que τ *peut néammoins contenir des noms de paramètres de collection.*

L'environnement de typage de FOC se compose de quatre fonctions, notées $\mathcal{C}, \mathcal{E}, \Sigma, \Gamma$ qui représentent respectivement les collections, les espèces, les méthodes de **self** et les variables locales du contexte courant. Plus précisément, ces fonctions sont de la forme suivante :

- $\mathcal{C} : c \mapsto \langle x_i : \tau_i \rangle$, associe à une collection c la liste de ses méthodes x_i avec leurs types τ_i, qui sont des types concrets.
- $\mathcal{E} : s \mapsto \{x_i : \tau_i = e_i\}$ associe à une espèce s la liste de ses méthodes x_i avec leurs types τ_i et leur définition (ou $\perp$ si x_i n'est que déclarée dans s). Là encore, les τ_i sont concrets.
- $\Sigma : x \mapsto (\tau, e)$ associe à un nom de méthode son type et sa définition (ou $\perp$) dans l'espèce courante. Le type τ est concret. Par extension, $\Sigma(\mathbf{rep})$ désigne le type support de l'espèce courante, et est égal à $\perp$ si **rep** est déclaré, et à τ si **rep** est défini par le type concret τ.
- $\Gamma : x \mapsto \forall \alpha_i, \tau$ associe à une variable x un schéma de type. Ici τ peut contenir des variables de type.

La présence des définitions des méthodes dans Σ et $\mathcal{E}$ est rendue nécessaire par les def-dépendances. Ces dernières se manifestent au niveau des preuves, comme on le verra plus en détail dans la section 3.9.1.

Notation 2 *Dans la suite, les collections seront habituellement désignées par c, les espèces par s, et les noms de méthodes par x ou y. En ce qui concerne les types, un type concret sera généralement représenté par τ, et un schéma de type par σ.*

Par ailleurs, dans les règles d'inférences utilisées dans la suite, on utilisera le symbole Ω pour faire référence aux fonctions qui ne sont ni utilisées ni modifiées dans la règle.

On se donne de plus une notion de bonne formation au niveau des types. Cette notion signale simplement que les noms de collections qui apparaissent dans un type τ sont des noms appartenant à $\mathcal{C}$. Dans toute la suite, on supposera que tous les types manipulés sont bien formés.

Définition 7 (Bonne formation des types) *Soit $\mathcal{C}$ un environnement de collection et τ un type. On dit que τ est* bien formé dans $\mathcal{C}$, *et on note $\mathcal{C} \vdash \tau$ si et seulement si les conditions suivantes sont remplies :*

$$\mathcal{C} \vdash \alpha \qquad \mathcal{C} \vdash \textbf{\textit{self}} \qquad \frac{\mathcal{C} \vdash \tau_1 \qquad \mathcal{C} \vdash \tau_2}{\mathcal{C} \vdash \tau_1 \to \tau_2} \qquad \frac{\mathcal{C} \vdash \tau_1 \qquad \mathcal{C} \vdash \tau_2}{\mathcal{C} \vdash \tau_1 * \tau_2} \qquad \frac{c \in \mathcal{C}}{\mathcal{C} \vdash c}$$

En ce qui concerne les expressions, les règles de typage sont essentiellement les mêmes que dans l'algorithme d'inférence de Hindley-Milner [Mil78]. Elles sont présentées dans la figure 3.1. Pour cela, nous introduisons les fonctions classiques, Mgu, Gen et Inst.

- Mgu unifie deux types τ_1 et τ_2, et peut instantier des variables de types durant le processus. La seule différence avec l'algorithme d'unification standard provient de l'adjonction des deux règles [SELF1] et [SELF2] qui permettent d'identifier le type support d'une espèce et sa représentation concrète quand elle existe.
- Gen permet de généraliser le type τ inféré pour une variable locale afin d'en faire un schéma de type $\forall\alpha_i, \tau$.
- Inst est la fonction qui introduit des variable de type "fraîches" dans un schéma de type $\forall\alpha_i, \tau$ pour le transformer en un type τ' en évitant d'utiliser des variables de type présentes dans l'environnement.

Introduisons maintenant les définitions de ces fonctions. Pour cela, on peut tout d'abord noter que seule la composante Γ de l'environnement FOC est concernée par ces définitions, puisque c'est la seule où on peut trouver des variables de type.

Définition 8 (Généralisation et instantiation)
Soientt Γ un environnement de typage, et τ un type. On définit $Gen(\tau, \Gamma)$ et $Inst(\tau, \Gamma)$ de la manière suivante :

$$Gen(\tau, \Gamma) = \forall\alpha_i.\tau$$

où $\{\alpha_i\} = \mathcal{F}(\tau)\backslash\mathcal{F}(\Gamma)$

$$Inst(\forall\alpha_i.\tau, \Gamma) = \tau[\alpha_i \leftarrow \alpha'_i]$$

où les α'_i vérifient $\alpha'_i \notin \mathcal{F}(\Gamma)$.

Pour définir Mgu, on commence par se donner une fonction annexe, mgu_{τ_σ} paramétrée par le type support d'une espèce, qui peut être soit un type concret, soit $\perp$ dans le cas où on n'a qu'une déclaration. mgu_{τ_σ} renvoie un type et une substitution permettant d'unifier les deux types s'ils sont unifiables et échoue sinon. $\text{Mgu}(\tau_1, \tau_2)$ est alors défini en instantiant τ_σ par le type support de l'espèce courante, et en appliquant la substitution résultat.

Définition 9 (Unification de types) *Soient t un type concret ou $\perp$, et τ_1 et τ_2 deux types. On définit $Mgu(\tau_1, \tau_2)$, l'unificateur le plus général de τ_1 et τ_2, à partir de la fonction auxiliaire $mg(t, \tau_1, \tau_2)$, qui prend en compte une*

éventuelle définition concrète du type support :

$$[\text{Eq}] \quad mg(t,\tau,\tau) = \tau, id \qquad [\text{Var1}] \quad mg(t,\alpha,\tau) = \tau, [\alpha \leftarrow \tau] \ \alpha \text{ non libre dans } \tau$$

$$[\text{Var2}] \quad mg(t,\tau,\alpha) = \tau, [\alpha \leftarrow \tau] \ \alpha \text{ non libre dans } \tau$$

$$[\text{Arrow}] \quad \frac{mg(t,\tau_1,\tau_1') = \tau_1'', \theta \qquad mg(t,\tau_2\theta,\tau_2'\theta) = \tau_2'', \phi}{mg(t,\tau_1 \to \tau_2, \tau_1' \to \tau_2') = \tau_1'' \to \tau_2'', \phi\theta}$$

$$[\text{Prod}] \quad \frac{mg(t,\tau_1,\tau_1') = \tau_1'', \theta \qquad mg(t,\tau_2\theta,\tau_2'\theta) = \tau_2'', \phi}{mg(t,\tau_1 * \tau_2, \tau_1' * \tau_2') = \tau_1'' * \tau_2'', \phi\theta}$$

$$[\text{Self1}] \quad mg(t, \textbf{\textit{self}}, \tau_\sigma) = \textbf{\textit{self}}, id$$

$$[\text{Self2}] \quad mg(t, \tau_\sigma, \textbf{\textit{self}}) = \textbf{\textit{self}}, id$$

Dans tous les autres cas, $mg(t,\tau_1,\tau_2)$ échoue.

Étant donnés $\mathcal{C}, \mathcal{E}, \Sigma, \Gamma$ et deux types τ_1, τ_2 tels que $mg(\Sigma(\textbf{\textit{rep}}), \tau_1, \tau_2) = \tau_3, \theta$ on définit $Mgu(\tau 1, \tau 2) = \tau_3\theta$.

Remarque 1. On peut noter que les règles [Self1] et [Self2] renvoient systématiquement **self**, c'est à dire abstraient τ_σ. Par ailleurs, il est impossible d'unifier **self** avec un type quelconque si le type support de l'espèce courante n'est pas explicitement défini. Comme dans le cas des méthodes polymorphes, cela permet d'éviter que deux méthodes utilisent des éléments de **self** en avec des hypothèses contradictoires sur sa représentation concrète.

Muni de ces trois fonctions, on peut énoncer les règles de typage pour les expressions de base du langage. Elles sont données dans la figure 3.1.

Nous pouvons maintenant définir la notion d'espèce bien typée : toutes ses méthodes sont bien typées, et les méthodes dont elle herite conservent leur type lors de l'héritage.

Définition 10 (Espèce bien typée) *Soit s une espèce bien spécifiée.*

$$\text{Well-typed-spec} \quad \frac{\begin{array}{c}\forall x \in \mathcal{N}(s),\ \Omega, \{x : \tau = \mathcal{B}_s(x)\}_{x\in\mathcal{N}(s)} \vdash \mathcal{B}_s(x) : \tau \\ \forall i,\ \forall h,\ \text{tel que } x_i \in \mathcal{N}(s_h), \{x_i : \tau_i = \mathcal{B}_{s_h}(x_i)\} \in \mathcal{E}(s_h)\end{array}}{\begin{array}{l}\Omega, \emptyset \vdash \textbf{\textit{species}}\ s \qquad \textbf{\textit{inherits}}\ s_1, ..., s_{h_f} = \qquad \Phi_1 ... \Phi_m : \\ \qquad\qquad \{x_i : \tau_i = \mathcal{B}_s(x_i)\}\end{array}}$$

Si s est bien typée, on définit le type de x dans s *: $\forall x_i \in \mathcal{N}(s)$, $\mathcal{T}_s(x_i) = \tau_i$.*

FIG. 3.1 – Typage des expressions

$$[\text{VAR}]\quad \frac{x : \forall \alpha_i.\tau' \in \Gamma \qquad \tau \leq \text{Inst}(\forall \alpha_i.\tau', \Gamma)}{\Omega, \Gamma \vdash x : \tau} \qquad [\text{ABS}]\quad \frac{\Omega, \Gamma + x : \tau_1 \vdash e : \tau_2}{\Omega, \Gamma \vdash \textbf{fun}\ x \to e : \tau_1 \to \tau_2}$$

$$[\text{LET}]\quad \frac{\Omega, \Gamma \vdash e_1 : \tau_1 \qquad \Omega, \Gamma + x : \text{Gen}(\tau_1, \Gamma) \vdash e_2 : \tau_2}{\Omega, \Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}$$

$$[\text{LET REC}]\quad \frac{\Omega, \Gamma + x : \text{Gen}(\tau_1, \Gamma) \vdash e_1 : \tau_1 \qquad \Omega, \Gamma + x : \text{Gen}(\tau_1, \Gamma) \vdash e_2 : \tau_2}{\Omega, \Gamma \vdash \textbf{let rec}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}$$

$$[\text{APP}]\quad \frac{\Omega \vdash e_0 : \tau_1 \to \ldots \to \tau_n \to \tau \qquad \forall i\ \Omega \vdash e_i : \tau_i}{\Omega \vdash e_0(e_1, \ldots, e_n) : \tau}$$

$$c \neq \textbf{self}\ \frac{[\text{METHCALL}]\quad c \in \mathcal{C} \qquad x : \tau \in \mathcal{C}(c)}{\Omega, \mathcal{C} \vdash c!x : \tau} \qquad \frac{[\text{SELFCALL}]\quad x : \tau = expr \in \Sigma}{\Omega, \Sigma \vdash \textbf{self}\,!x : \tau}$$

Dans la règle WELL-TYPED-SPEC, Ω représente $\mathcal{C}, \mathcal{E}, \Gamma$. Dans le but de la règle, Σ est vide : il n'y a pas de méthode de **self** en dehors d'une espèce. Dans la première prémisse en revanche, on place dans Σ toutes les méthodes de s. La seconde prémisse exprime le fait qu'une méthode doit conserver son type tout au long de l'héritage.

3.4 Collections

Une collection c ne peut être créée qu'à partir d'une espèce complètement définie s (c'est à dire une espèce dont toutes les méthodes possèdent une définition).

Définition 11 (Espèce complètement définie) *Soit s une espèce bien spécifiée. s est dite complètement définie si et seulement si $\mathcal{N}(s) = \mathcal{D}(s)$. Une telle espèce peut par contre comporter des paramètres (cf 3.8).*

La règle de typage d'une collection consiste dès lors à vérifier qu'elle implante bien une espèce complètement définie. Par ailleurs, on abstrait le type support de c ainsi que toutes les méthodes. En effet, contrairement à $\mathcal{E}$, $\mathcal{C}$ ne contient pas de définition. Enfin, comme c'est le souvent le cas en mathématiques, on note de la même manière la collection et le type support

sur lequel elle est bâtie : dans les types des méthodes de l'interface de c, on substitue à **self** le nom de la collection, c.

$$\frac{s : \{x_i : \tau_i = e_i\} \in \mathcal{E} \qquad s \text{ est complètement définie}}{\mathcal{C}, \mathcal{E}, \Sigma, \Gamma \vdash \textbf{collection } c \textbf{ implements } s : \langle x_i : \tau_i[\textbf{self} \leftarrow c]\rangle} \text{[Coll]}$$

Comme on l'a déjà dit, le typage d'une espèce ou d'une collection ne fait donc pas appel aux dépendances. Au contraire, on suit ici un modèle orienté-objet en plaçant dans Σ *toutes* les méthodes de l'espèce s qu'on cherche à typer. Dans ce contexte, une collection n'étant qu'une instance particulière d'une espèce peut également être typée sans qu'on fasse appel aux dépendances.

Toutefois, cette analyse est loin d'offrir toutes les garanties possibles sur un programme Foc. En particulier, elle autorise des dépendances mutuelles entre toutes les méthodes d'une espèce, d'où qu'elles viennent (qu'elles soient héritées d'un parent ou définies dans l'espèce elle-même). Comme on l'a déjà vu précédemment (3.1), cette approche conduit à accepter des programmes qui ne terminent pas, et qu'il est difficile de corriger dans la mesure ou les méthodes mutuellement récursives en cause peuvent se trouver dans des espèces complètement différentes. Afin d'éviter cela, nous allons donc maintenant présenter la seconde étape de l'analyse d'une espèce, la détection de cycle de dépendances entre ses méthodes.

3.5 Les dépendances

Très informellement, une méthode m_1 dépend d'une autre méthode m_2 si le corps de m_1 utilise **self!**$m2$. Pour cela, la fonction $\wr e \int$, définie ci-dessous, renvoie la liste des méthodes de **self** dont dépend une expression e.

Définition 12 (Dépendances d'une expression) *La fonction $\wr\cdot\int$ est définie par induction sur les expressions du langage :*

$$\begin{aligned}
\wr x \int &= \emptyset \\
\wr \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 \int &= \wr e_1 \int \cup \wr e_2 \int \\
\wr \textbf{\textit{let rec}}\ x = e_1\ \textbf{\textit{in}}\ e_2 \int &= \wr e_1 \int \cup \wr e_2 \int \\
\wr \lambda x.e \int &= \wr e \int \\
\wr c!x \int &= \begin{cases} \emptyset & \text{quand } c \neq \textbf{\textit{self}} \\ \{x\} & \text{quand } c = \textbf{\textit{self}} \end{cases} \\
\wr e_0(e_1, \ldots, e_n) \int &= \wr e_0 \int \cup (\textstyle\bigcup_{i=1}^{n} \wr e_i \int)
\end{aligned}$$

A partir de cela, on peut étendre la définition de $\wr\cdot\int$ aux champs, en prenant soin de distinguer **let** et **let rec**. En effet, dans un champ récursif ϕ nous allons effacer les dépendances concernant les méthodes de $\mathcal{N}(\phi)$,

puisqu'on ne cherche à détecter que les cycles de dépendances qui ont lieu en dehors d'un **let rec**.

Notation 3 *On appellera un champ de la forme* ***let*** x=... *un champ* non récursif, *et un champ* ϕ *de la forme* ***let rec*** x=... *un champ* récursif. *Dans ce dernier cas, si* ϕ *introduit au moins deux noms, on parlera de champ* mutuellement récursif

Définition 13 (Dépendances d'un champ)

$$\begin{array}{rcl} \wr \textbf{\textit{let}}\ x = e \int & = & \wr e \int \\ \wr \textbf{\textit{let rec}}\ \{x_1 = e_1\ \ldots x_n = e_n\} \int & = & \cup_{i=1}^{n} \wr e_i \int \setminus \{x_i\}_{i=1\ldots n} \\ \wr \textbf{\textit{sig}}\ x\ \textbf{\textit{in}}\ \tau \int & = & \emptyset \end{array}$$

Enfin, on peut définir $\wr x \int_s$, les dépendances de x dans l'espèce s. Tout comme dans le cas de $\mathcal{B}_s(x)$, dans le cas d'un héritage multiple, on s'intéresse au dernier champ où x apparaît. Toutefois, on tient compte des éventuels autres champs récursifs où x apparaît. En effet, ces champs sont appelés à être fusionnés par la fonction $\circledcirc$ lors de la mise en forme normale (voir la section 3.7). Comme cette fusion peut amener à définir dans le même champ des noms de méthode qui étaient distincts dans les espèces parents, on ne peut se contenter de considérer les dépendances champ par un champ : il faut tenir compte des liaisons induites par la résolution de l'héritage dans le cas des champs mutuellement récursifs. Pour cela, on commence par se donner une relation $\circlearrowright_s$, qui lie les noms de méthodes liés par une récursion mutuelle.

Définition 14 (Noms de Méthode mutuellement liés) *Soit* s *une espèce bien spécifiée, On définit la relation* $\circlearrowright_s$ *sur* $\mathcal{N}(s)$ *de la manière suivante :*

$$\text{NAMES-MUT-REC}\ \frac{\exists i, x \in \mathcal{N}(\phi_i) \wedge y \in \mathcal{N}(\phi_i)}{x \circlearrowright_s y} \qquad \text{NAMES-INH}\ \frac{x \circlearrowright_{s_h} y}{x \circlearrowright_s y} \qquad \text{NAMES-TRANS}\ \frac{x \circlearrowright_s y \qquad y \circlearrowright_s z}{x \circlearrowright_s z}$$

Lemme 1 *Pour toute espèce bien spécifiée* s, $\circlearrowright_s$ *est une relation d'équivalence sur* $\mathcal{N}(s)$.

Preuve. La reflexivité s'obtient par induction sur l'arbre d'héritage : si $x \in \mathcal{N}(\phi_i)$, alors on a trivialement $x \circlearrowright_s x$, par la règles NAMES-MUT-REC. Sinon, par définition de $\mathcal{N}(s)$, $x \in \mathcal{N}(s_h)$ pour une espèce parent s_h de s, et par hypothèse d'induction, on a $x \circlearrowright_{s_h} x$: on conclut avec NAMES-INH.

Pour montrer la symétrie de la relation, on peut procéder par induction sur le graphe d'héritage et sur la hauteur de la dérivation de $x \circlearrowright_s y$, et par cas suivant la dernière règle utilisée.

NAMES-MUT-REC : on a trivialement $y \circlearrowright_s x$, les prémisses étant symétriques.

NAMES-INH : Immédiat par hypothèse de récurrence sur le graphe d'héritage.

NAMES-TRANS : Immédiat par induction sur la dérivation de $x \circlearrowleft_s y$.

Enfin, la transitivité est immédiate avec la règle NAMES-TRANS. □

A l'aide de cette relation, on peut définir l'ensemble des champs impliqués dans la définition de x au sein de l'espèce s : ce sont tous les champs introduisant un nom avec lequel x est en relation.

Définition 15 *Soient s une espèce bien spécifiée et $x \in \mathcal{N}(s)$. On définit les champs d'introduction de x dans s ($Where_s(x)$) de la manière suivante.*

$$\text{FIELD-INTRO}\ \frac{x \circlearrowleft_s y \qquad y \in \mathcal{N}(\phi_i)}{\phi_i \in Where_s(x)} \qquad \text{FIELD-INH}\ \frac{x \circlearrowleft_s y \qquad \phi \in Where_{s_h}(y)}{\phi \in Where_s(x)}$$

Lemme 2 *Soient s une espèce bien spécifiée et $x \in \mathcal{N}(s)$. La classe d'équivalence de x pour $\circlearrowleft_s$ est*

$$\bigcup_{\phi \in Where_s(x)} \mathcal{N}(\phi)$$

Preuve. Soit y tel que $x \circlearrowleft_s y$. Montrons qu'il existe $\phi \in Where_s(x)$ vérifiant $y \in \mathcal{N}(\phi)$. Par induction sur le graphe d'héritage et la dérivation de $x \circlearrowleft_s y$, on a les cas suivants :

NAMES-MUT-REC : par définition, $\phi_i \in Where_s(x)$ et $y \in \mathcal{N}(\phi_i)$.

NAMES-INH : Par induction sur le graphe d'héritage, $\exists \phi \in Where_{s_h}(x)$, tel que $y \in \mathcal{N}(\phi)$. En appliquant FIELD-INH, il vient $\phi \in Where_s(x)$.

NAMES-TRANS : Par induction sur la hauteur de la dérivation, $\exists \phi \in Where_s(z)$, tel que $y \in \mathcal{N}(\phi)$. Comme de plus $x \circlearrowleft_s z$, ϕ est aussi dans $Where_s(x)$.

Inversement, soient $\phi \in Where_s(x)$ et $y \in \mathcal{N}(\phi)$. Montrons que $x \circlearrowleft_s y$. Par induction sur le graphe d'héritage, on a les cas suivants :

FIELD-INTRO $x \circlearrowleft_s z$ et $z \in \mathcal{N}(\phi)$. Il vient $z \circlearrowleft_s y$, et par transitivité, $x \circlearrowleft_s y$.

FIELD-INH $x \circlearrowleft_s z$ et $\phi \in Where_{s_h} z$. Par hypothèse d'induction, $z \circlearrowleft_{s_h} y$, donc avec NAMES-INH, il vient $z \circlearrowleft_s y$, et on conclut par transitivité.

□

Corollaire 1 *Si pour tout ϕ dans $Where_s(x)$, ϕ est non récursif, alors la classe d'équivalence pour $\circlearrowleft_s$ de x est le singleton $\{x\}$.*

Preuve. Immédiat avec le lemme précédent : pour tout $\phi \in Where_s(x)$, $\mathcal{N}(\phi) = \{x\}$ □

Définition 16 (Dépendances au sein d'une espèce) *Soit s une espèce bien spécifiée. On définit $\wr x \int_s$ pour tout x de $\mathcal{D}(s)$:*

- *si* $\forall \phi \in Where_s(x)$, ϕ non récursif, $\wr x \int_s = \wr \mathcal{B}_s(x) \int$
- *sinon,* $\wr x \int_s = \bigcup_{y \circlearrowleft_s x} \wr \mathcal{B}_s(y) \int \backslash \{y | y \circlearrowleft_s x\}$

Lemme 3

$$\forall x, y, z \in \mathcal{N}(s),\ x \in \wr y \int_s \wedge y \circlearrowleft_s z \Rightarrow x \in \wr z \int_s$$

Preuve. Si tous les champs d'introduction de y dans s sont non récursifs, alors on a $z = y$ avec le corollaire 1, et la propriété devient triviale. Sinon, comme $\circlearrowleft_s$ est une relation d'équivalence, on a $u \circlearrowleft_s y \iff u \circlearrowleft_s z$, soit $\wr y \int_s = \wr z \int_s$ par définition de $\wr \cdot \int_s$. □

Cela amène à introduire la notion d'*espèce bien formée*, dans laquelle il n'y a pas de cycle de dépendance en dehors des structures **let rec**. Les seules espèces que le compilateur FOC accepte sont celles qui sont à la fois bien typées et bien formées.

Définition 17 (bonne formation) *Soit s une espèce bien spécifiée. On définit la clôture transitive de la relation de dépendance :*

$$x_1 \blacktriangleleft_s x_2 \iff \exists \{y_i\}_{i=1\ldots n} \text{ tq } y_1 \circlearrowleft_s x_1, y_n \circlearrowleft_s x_2, \forall i < n,\ y_{i+1} \in \wr y_i \int_s$$

s est dite bien formée *si et seulement si* $\forall x \in \mathcal{N}(s)\ \neg (x \blacktriangleleft_a x)$.

La prise en compte de la relation $\circlearrowleft_s$ est liée à la résolution de l'héritage. En effet, une méthode x peut être présente dans plusieurs champs mutuellement récursifs. La fusion de ces champs en un champ mutuellement récursif unique aboutit à introduire tous les noms liés à x dans le même champ. De ce fait, les dépendances de ce champ sont l'union de toutes les dépendances des méthodes liés à x.

Lemme 4 *Soit s une espèce bien spécifiée. La relation* $\blacktriangleleft_s$ *est transitive.*

Preuve. Soient x, y, z tels que $x \blacktriangleleft_s y$ et $y \blacktriangleleft_s z$. Par définition de $\blacktriangleleft_s$, on a $u_1, ..., u_m, u_{m+1}, ...u_n$ verifiant :

1. $\forall i \neq m,\ u_{i+1} \in \wr u_i \int_s$
2. $u_1 \circlearrowleft_s x$
3. $u_m \circlearrowleft_s y$
4. $u_{m+1} \circlearrowleft_s y$
5. $u_n \circlearrowleft_s z$

Par transitivité de $\circlearrowleft_s$, $u_m \circlearrowleft_s u_{m+1}$. Donc, d'après le lemme 3, $u_{m+2} \in \wr u_m \int_s$, et $x \blacktriangleleft_s z$. □

Notons par ailleurs que dans la mesure où s ne peut hériter que d'espèces déjà présentes dans $\mathcal{E}$, toutes les espèces s_h dont hérite s sont bien formées (sinon elles auraient été rejetées).

Enfin, une espèce en *forme normale* est une espèce bien formée n'ayant pas de clause **inherits**. Cette notion est utilisée par le compilateur. L'utilisateur lui-même écrit rarement des espèces en forme normale.

Définition 18 (forme normale d'une espèce) *Soit nf une espèce définie par :*

$$\textbf{\textit{species}}\ nf = \phi_1 \ldots \phi_n\ \textbf{\textit{end}}$$

nf est dite en forme normale *si et seulement si les quatre conditions suivantes sont remplies :*

- *nf ne contient pas de clause* ***inherits****.*
- *nf est bien typée.*
- *nf est bien spécifiée :*

$$\forall i, j,\ i \neq j \Rightarrow \mathcal{N}(\phi_i) \cap \mathcal{N}(\phi_j) = \emptyset$$

- *Une définition ne dépend que des noms introduits dans les champs précédents :*

$$\forall i \leq n, \forall x \in \mathcal{N}(\phi_i),\ \wr x \int_{nf} \subset \bigcup_{j=1}^{i-1} \mathcal{N}(\phi_j)$$

Remarque 2. La dernière condition de la définition précédente implique la bonne formation de l'espèce *nf*.

L'objet des deux sections suivantes est de montrer que, toute espèce s bien formée peut être mise en forme normale, tout en préservant les noms des méthodes de s, ainsi que leurs types et leurs définitions éventuelles. Ces différents points sont formalisés dans le théorème 3. Par ailleurs, nous étudierons un algorithme permettant de déterminer si une espèce bien spécifiée et bien typée est en outre bien formée.

Notation 4 *Si nf est une espèce en forme normale et $x \in \mathcal{N}(nf)$, on notera $Intros_{nf}(x)$ l'unique champ de nf tel que $x \in \mathcal{N}(Intros_x(nf))$*

Lemme 5 *Soit s une espèce bien spécifiée et ne comportant pas de clause* ***inherits****. On a les propriétés suivantes :*

- $x \circlearrowright_s y \iff Intros_s(x) = Intros_s(y)$.
- $Where_s(x) = \{Intros_s(x)\}$.
- $\wr x \int_s = \wr Intros_s(x) \int$.

Preuve. Les deux premières propriétés se déduisent immédiatement du fait que s ne comporte pas de clause **inherits** : les règles NAMES-INH et FIELD-INH ne peuvent être appliquées

Par ailleurs, si $Intros_s(x)$ est non récursif, on a par définition $\wr x \int_s = \wr \mathcal{B}_s(x) \int = \wr Intros_s(x) \int$.

De même, si $Intros_s(x)$ est récursif, $\wr x \int_s = \bigcup_{y \circlearrowright_s x} \mathcal{B}_s(y) \backslash \{y | y \circlearrowright_s x\}$, soit avec ce qui précède,

$$\wr x \int_s = \bigcup_{y \in \mathcal{N}(Intros_s(x))} \mathcal{B}_s(y) \backslash \{y | y \in \mathcal{N}(Intros_s(x))\} = \wr Intros_s(x) \int$$

□

3.6 Résolution de l'héritage et fusion de champs

Dans cette section, nous considérons une espèce bien spécifiée s. Pour montrer qu'elle est bien formée et bien typée, nous allons créer une espèce $norm(s)$ en forme normale qui lui est "équivalente" dans le sens où elle partage les mêmes définitions et les mêmes déclarations. Informellement, s et $norm(s)$ ne peuvent pas être distinguées l'une de l'autre à l'extérieur de l'espèce : elles réagissent de la même manière aux appels de méthodes. Cette construction a lieu par induction sur le graphe d'héritage. Les espèces bien typées et bien formées qui n'héritent d'aucune autre peuvent être mises trivialement en forme normale : il suffit de réordonner leurs champs.

Notation 5 *Dans la suite, nous assimilerons une espèce en forme normale avec la liste de tous ses champs.* $l_1@l_2$ *indiquera la concaténation de deux listes. Enfin, on note* $norm(s_h)$ *la forme normale de* s_h *pour chaque espèce dont* s *hérite.*

Pour construire $norm(s)$, on commence par construire la liste complète, notée $\mathbb{W}_1$ des champs impliqués dans s, c'est à dire les champs des $norm(s_h)$ et ceux du corps de s :

$$\mathbb{W}_1 = norm(s_1)@...@norm(s_{h_f})@[\phi_1, ..., \phi_m]$$

$\mathbb{W}_1$ peut contenir plusieurs occurences du même nom dans des champs différents, à cause de l'héritage multiple ou de la redéfinition de certaines méthodes. Nous allons donc construire une nouvelle liste, $\mathbb{W}_2$ dans telle que chaque nom de $\mathbb{W}_1$ n'y figure qu'une seule fois, à la place requise par la spécification de l'héritage et de la liaison retardée. $\mathbb{W}_2$ sera alors identifiée à l'espèce $\tilde{s}$ définie de la manière suivante :

$$\textbf{species } \tilde{s} = \mathbb{W}_2 \textbf{ end}$$

Nous allons prouver que si s est bien typée et bien formée, c'est aussi le cas pour $\tilde{s}$. La forme normale de s s'obtient alors à partir de $\tilde{s}$ en réordonnant les champs.

Pour construire $\mathbb{W}_2$ à partir de $\mathbb{W}_1$, il faut trouver un moyen de résoudre les "conflits" (définitions ou déclarations multiples d'une même méthode). Pour cela, on peut s'appuyer sur la fonction $\diamond$, définie plus loin, qui effectue la fusion de deux champs ϕ_1 et ϕ_2 qui ont au moins un nom en commun (il peut y en avoir plusieurs dans le cas de champs récursifs). Cette fonction n'est pas totale car un nom de méthode peut être lié à des types différents dans les deux champs. Dans ce cas, la fusion échoue. On montre plus loin (lemme 8) que si l'espèce de départ est bien typée, $\diamond$ n'échoue jamais.

Par ailleurs, deux champs **let rec** peuvent être fusionnés même s'ils n'introduisent pas exactement les mêmes noms. Ainsi, on peut hériter d'un

champ **let rec** et redéfinir seulement une partie des fonctions qui le composent, ou introduire de nouvelles méthodes dans la récursion. Dans ce cas, $\Subset$ prend toutes les méthodes apparaissant dans au moins un des deux champs et renvoie un champ récursif composé de toutes ces méthodes. Bien entendu, cela implique aussi qu'on fasse une nouvelle preuve de terminaison (3.9.2), impliquant *toutes* les méthodes figurant dans ce nouveau champ récursif.

Définition 19 (Fusion de deux champs) *Soient ϕ_1 et ϕ_2 deux champs tels que $\mathcal{N}(\phi_1) \cap \mathcal{N}(\phi_2) \neq \emptyset$.*

$\phi_1 \Subset \phi_2$ est défini par cas de la manière suivante (notons qu'en raison des appels à Mgu, cette fonction est partielle) :

$\textbf{sig}\ x\ \textbf{in}\ \tau$	$\Subset\ \textbf{sig}\ x\ \textbf{in}\ \tau$	$= \textbf{sig}\ x\ \textbf{in}\ \tau$
$\textbf{sig}\ x\ \textbf{in}\ \tau_1$	$\Subset\ \textbf{let}\ x\ in\ \tau_2 = e_2$	$= \textbf{let}\ x\ \textbf{in}\ Mgu(\tau_1, \tau_2) = e_2$
$\textbf{sig}\ x\ \textbf{in}\ \tau_1$	$\Subset \textbf{let rec}$	$= \textbf{let rec}$
	$(x\ in\ \tau_2 = e_2) \cup C$	$(x\ in\ Mgu(\tau_1, \tau_2) = e_2) \cup C$
$\textbf{let rec}$	$\Subset\ \textbf{sig}\ x\ \textbf{in}\ \tau_2$	$= \textbf{let rec}$
$(x\ in\ \tau_1 = e_1) \cup C$		$(x\ in\ Mgu(\tau_1, \tau_2) = e_1) \cup C$
$\textbf{let}\ x\ in\ \tau_1 = e_1$	$\Subset\ \textbf{let}\ x\ in\ \tau_2 = e_2$	$= \textbf{let}\ x\ \textbf{in}\ Mgu(\tau_1, \tau_2) = e_2$
$\textbf{let}\ x\ in\ \tau_1 = e_1$	$\Subset\ \textbf{sig}\ x\ \textbf{in}\ \tau_2$	$= \textbf{let}\ x\ \textbf{in}\ Mgu(\tau_1, \tau_2) = e_1$
$\textbf{let rec}$	$\Subset \textbf{let rec}$	$= \textbf{let rec}$
$\{x_i\ [in\ \tau_i] = ex_i$	$\{x_i\ [in\ \sigma_i] = ex'_i$	$(\{x_i\ in\ Mgu(\tau_i, \sigma_i) = ex'_i\}$
$y_i\ [in\ \rho_i] = ey_i\}$	$z_i\ [in\ \pi_i] = ez_i\}$	$\cup\{y_i\ in\ \rho_i = ey_i\}$
		$\cup\{z_i\ in\ \pi_i = ez_i\})$

Cette opération préserve deux invariants importants. En premier lieu, tous les noms introduits dans ϕ_1 ou ϕ_2 le sont dans leur fusion $\phi_1 \Subset \phi_2$. De même, si une méthode est définie dans un des deux champs, elle l'est aussi dans le résultat. En second lieu, cette opération est compatible avec la *liaison retardée* qui demande qu'un appel de méthode utilise toujours la définition la plus "récente" au sens du graphe d'héritage. Plus formellement, on peut énoncer les deux propriétés suivantes :

Lemme 6 (Préservation des noms) $\forall \phi_1, \phi_2\ st\ \mathcal{N}(\phi_1) \cap \mathcal{N}(\phi_2) \neq \emptyset$

$$\mathcal{N}(\phi_1 \Subset \phi_2) = \mathcal{N}(\phi_1) \cup \mathcal{N}(\phi_2)$$

La même propriété est vraie pour $\mathcal{D}(\cdot)$.

Lemme 7 (Liaison retardée) $\forall \phi_1, \phi_2$ tel que $\mathcal{N}(\phi_1) \cap \mathcal{N}(\phi_2) \neq \emptyset$

$$\begin{cases} \forall x \in \mathcal{D}(\phi_2), & \mathcal{B}_{\phi_1 \Subset \phi_2}(x) = \mathcal{B}_{\phi_2}(x) \\ \forall x \in \mathcal{D}(\phi_1) \setminus \mathcal{D}(\phi_2), & \mathcal{B}_{\phi_1 \Subset \phi_2}(x) = \mathcal{B}_{\phi_1}(x) \end{cases}$$

Preuve. Immédiat par analyse de cas sur la structure de ϕ_1 et ϕ_2. □

Remarque 3. La dernière propriété n'est intéressante que si ϕ_1 et ϕ_2 sont tous les deux des définitions. Sinon, il n'y a pas d'élément dans les $\mathcal{D}(\phi_i)$ et la propriété devient triviale.

3.7 Résolution de l'héritage

3.7.1 Algorithme de mise en forme normale

Nous pouvons construire la liste $\mathbb{W}_2$ à partir de $\mathbb{W}_1$, en analysant les éléments un par un *dans l'ordre* dans lequel ils apparaissent dans la liste. A chaque étape, on examine le premier champ restant dans $\mathbb{W}_1$ et on met à jour $\mathbb{W}_1$ et $\mathbb{W}_2$, suivant qu'il existe ou non un conflit entre ce champ et un (ou plusieurs) des champs de $\mathbb{W}_2$. L'algorithme se compose d'une boucle qui fonctionne de la manière suivante :

ENTRÉE : une liste de champs $\mathbb{W}_1 = \phi_1 \dots \phi_n$.
SORTIE : une liste de champs $\mathbb{W}_2$ introduisant des noms deux à deux distincts.
Initialiser $\mathbb{W}_2 \leftarrow \emptyset$
Tant que $\mathbb{W}_1 \neq \emptyset$ faire :
- $\phi \leftarrow hd(\mathbb{W}_1)$
- $\mathbb{X} \leftarrow tl(\mathbb{W}_1)$
- si $\mathcal{N}(\phi) \cap \mathcal{N}(\mathbb{W}_2) = \emptyset$ faire :
  - $\mathbb{W}_1 \leftarrow \mathbb{X}$
  - $\mathbb{W}_2 \leftarrow \mathbb{W}_2, \phi_1$
- Sinon, soient $\mathbb{W}_2 = [\psi_1, ..., \psi_n]$ et
  i_0 le plus petit indice tel que $\mathcal{N}(\phi) \cap \mathcal{N}(\psi_{i_0}) \neq \emptyset$. faire :
  - $\mathbb{W}_1 \leftarrow (\phi \circledcirc \psi_{i_0}), \mathbb{X})$.
  - $\mathbb{W}_2 \leftarrow [\psi_1 \dots \psi_{i_0-1}, \psi_{i_0+1} \dots \psi_m]$.

A chaque passage dans la boucle, on examine le premier champ ϕ restant dans $\mathbb{W}_1$. Si ϕ n'a aucun nom en commun avec les champs qui ont déjà été analysés, alors on peut l'ajouter à la fin de $\mathbb{W}_2$. Dans le cas contraire, il faut utiliser $\circledcirc$. Toutefois, il ne serait pas correct d'ajouter le résultat directement à la fin de $\mathbb{W}_2$. En effet, si ϕ est un champ définissant plusieurs méthodes mutuellement récursives, il peut avoir des noms en commun avec plus d'un champ ψ_i : il va alors falloir appliquer plusieurs fois de suite $\circledcirc$. Pour cela, on garde $\phi \circledcirc \psi_{i_0}$ dans $\mathbb{W}_1$, afin de poursuivre son analyse.

Pour garantir la terminaison de cet algorithme, on peut prendre l'ordre lexicographique suivant : $(\mathrm{Card}\,\mathbb{W}_1, \mathrm{Card}\,\mathbb{W}_2)$. En effet, en appelant $\widetilde{\mathbb{W}_1}$ et $\widetilde{\mathbb{W}_2}$ les valeurs des deux listes après une étape de la boucle, on est dans un des deux cas suivants :

- S'il n'y a pas de conflit, $\mathrm{Card}\,\widetilde{\mathbb{W}_1} < \mathrm{Card}\,\mathbb{W}_1$
- Sinon $\mathrm{Card}\,\widetilde{\mathbb{W}_1} = \mathrm{Card}\,\mathbb{W}_1$, et on a retiré un champ de $\mathbb{W}_2$, donc $\mathrm{Card}\,\widetilde{\mathbb{W}_2} < \mathrm{Card}\,\mathbb{W}_2$.

Nous allons maintenant établir les principales propriétés de cet algorithme, afin de pouvoir montrer (théorème 3) que $\mathbb{W}_2$ définit bien une espèce bien formée équivalente à s. Dans tout ce qui suit, on utilise les mêmes notations que précédemment pour désigner les champs de $\mathbb{W}_1$ et $\mathbb{W}_2$.

Lemme 8 *Avec les notations précédentes, si s est bien typée, $\psi_{i_0} \circledcirc \phi$ n'échoue jamais.*

Preuve. Conséquence directe de la définition de $\circledcirc$ et la définition 10 : dans une espèce bien typée, tous les types des champs de même nom sont unifiables, donc l'appel à Mgu dans $\circledcirc$ n'échoue jamais. □

Lemme 9 (unicité des noms de $\mathbb{W}_2$)
$\forall \phi_1, \phi_2 \in \mathbb{W}_2,\ \mathcal{N}(\phi_1) \cap \mathcal{N}(\phi_2) = \emptyset.$

Preuve. Par induction sur le nombre d'étapes de l'algorithme $\mathbb{W}_1$. Au départ, $\mathbb{W}_2$ est vide, et l'énoncé est alors trivial. Montrons maintenant que si la propriété est vérifiée avant la n-ième étape, elle l'est aussi après. Il y a deux cas possibles :

- Si $\forall \psi \in \mathbb{W}_2, \mathcal{N}(\phi) \cap \mathcal{N}(\psi) = \emptyset$ alors on ajoute ϕ à $\mathbb{W}_2$ et la propriété est préservée.
- Sinon, on enlève un champ de $\mathbb{W}_2$, et les autres continuent de vérifier la propriété.

□

En particulier, à la fin de l'algorithme, lorsque $\mathbb{W}_1$ est vide, $\mathbb{W}_2$ contient des champs introduisant des noms deux à deux distincts. Nous pouvons donc maintenant définir l'espèce $\tilde{s}$ de la manière suivante :

$$\textbf{species } \tilde{s} = \mathbb{W}_2 \textbf{ end}$$

La propriété principale de $\tilde{s}$ est qu'elle déclare et définit les mêmes méthodes que s :

Théoreme 1 (Équivalence) $\mathcal{N}(s) = \mathcal{N}(\tilde{s}),\ \mathcal{D}(s) = \mathcal{D}(\tilde{s}),\ et$
$\forall x \in \mathcal{D}(\tilde{s}),\ \mathcal{B}_s(x) = \mathcal{B}_{\tilde{s}}(x)$

Preuve. Nous allons montrer qu'à chaque étape de la boucle les propriétés suivantes sont préservées :

1. $\mathcal{N}(s) = \bigcup_{\phi \in \mathbb{W}_1} \mathcal{N}(\phi) \ \cup \ \bigcup_{\psi \in \mathbb{W}_2} \mathcal{N}(\psi)$
2. $\mathcal{D}(s) = \bigcup_{\phi \in \mathbb{W}_1} \mathcal{D}(\phi) \ \cup \ \bigcup_{\psi \in \mathbb{W}_2} \mathcal{D}(\psi)$
3. $\forall x \in \mathcal{N}(s), \exists \phi \in \mathbb{W}_2 \cup \mathbb{W}_1, \mathcal{B}_s(x) \in \phi$

À la première étape, ces propriétés sont trivialement vraies, puisque $\mathbb{W}_1$ contient toutes les définitions et déclarations de s et de ses parents et que $\mathbb{W}_2$ est vide. Supposons maintenant que la propriété est toujours vraie après n étapes. Soit ϕ le premier champ de $\mathbb{W}_1$.

Si $\forall \chi \in \mathbb{W}_2\ \mathcal{N}(\phi) \cap \mathcal{N}(\chi) = \emptyset$, on ne fait que faire passer ϕ de $\mathbb{W}_1$ à $\mathbb{W}_2$, de sorte que ni la liste des noms apparaissant dans une des deux listes, ni la liste des définitions associées ne change.

Sinon, on note ψ_{i_0} le champ de $\mathbb{W}_2$ à fusionner avec ϕ. On enlève ψ_{i_0} de $\mathbb{W}_2$ et ϕ de $\mathbb{W}_1$ et on ajoute $\psi_{i_0} \ominus \phi$ dans $\mathbb{W}_1$. Le lemme 6 nous donne $\mathcal{N}(\psi_{i_0} \ominus \phi) = \mathcal{N}(\psi_{i_0}) \cup \mathcal{N}(\phi)$, Donc, la liste des noms de méthodes reste inchangée. De même, ce lemme permet de conclure pour $\mathcal{D}(\cdot)$: les propriétés 1 et 2 sont donc vérifiées.

Par ailleurs, le lemme 7 montre que les corps de méthodes qui sont effacés sont $\left\{\mathcal{B}_{\psi_{i_0}}(x), x \in \mathcal{D}(\psi_{i_0}) \cap \mathcal{D}(\phi)\right\}$. Or $\mathbb{W}_1$ est ordonné, et, par définition de $\mathcal{B}_s(x)$

$$\forall x \in \mathcal{D}(\psi_{i_0}) \cap \mathcal{D}(\phi),\ \mathcal{B}_s(x) \neq \mathcal{B}_{\psi_{i_0}}(x)$$

La propriété 3 est donc vérifiée elle aussi.

A la fin de la construction, $\mathbb{W}_1 = \emptyset$, donc on a $\mathcal{N}(x) = \bigcup_{\psi \in \mathbb{W}_2} \mathcal{N}(\psi)$. Par ailleurs, le lemme précédent nous montre que chaque méthode définie est introduite une seule fois dans $\mathbb{W}_2$, donc $\mathcal{B}_s(x) = \mathcal{B}_{\tilde{s}}(x)$. □

Notons que, comme on l'a dit précédemment, il serait possible de raffiner, dans le cas de l'héritage multiple, la sélection d'une définition d'un champ x. Plus précisément, il est possible d'hériter la méthode x d'une espèce précise au lieu de toujours sélectionner la dernière définition dans l'ordre de la clause **inherits**. À cette fin, on suppose qu'on a une fonction f (fournie par le programmeur) vérifiant les propriétés suivantes :

- $f : \bigcup_{h=1\ldots h_f} \mathcal{D}(s_h) \to \{s_1 \ldots s_h\}$
- $\forall x, x \in \mathcal{D}(f(x))$
- $\forall x, \forall y \in \mathcal{N}(\mathit{Intros}_{f(x)}(x)), f(y) = f(x)$

À toute méthode x définie dans au moins une des espèce parent, f associe un nom d'espèce s_h tel que x soit défini dans s_h. On suppose en outre que si x est défini par un champ récursif ϕ dans $f(x)$, alors pour toutes les méthodes y définies par ϕ on a $f(x) = f(y)$. Cette restriction permet d'éliminer d'éventuelles ambiguïtés dans la sélection d'un bloc récursif.

Muni de cette fonction, on modifie la définition de $\mathcal{B}_s(x)$ de telle manière que dans le cas d'une méthode héritée on ait $\mathcal{B}_s(x) = \mathcal{B}_{f(x)}(x)$.

Par ailleurs, il faut modifier la définition de la liste d'entrée $\mathbb{W}_1$ de l'algorithme ci-dessus. Pour tout x défini dans les espèce parent de s, on permute alors les champs $\mathit{Intros}_{norm(f(x))}(x)$ avec les autres champs de $norm(s_1)@\ldots@norm(s_n)$ pour qu'ils apparaissent à la fin de la liste des champs hérités dans $\mathbb{W}_1$. La condition sur les champs mutuellement récursifs permet de s'assurer qu'une telle permutation existe.

Une fois, que cette permutation a été faite, on peut dérouler l'algorithme comme précédemment : les étapes de fusion successives vont sélectionner $\mathcal{B}_{f(x)}(x)$ pour les champs x hérités et non redéfinis.

3.7.2 Bonne formation de l'espèce $\tilde{s}$

Il reste maintenant à montrer que l'algorithme préserve la bonne formation de s, c'est à dire que $\tilde{s}$ elle-même est bien formée.

Théoreme 2 (Bonne formation de $\tilde{s}$)
En utilisant les notations de la section précédente, si s est bien formée, alors $\tilde{s}$ l'est aussi.

Preuve. Nous allons prouver ce théorème par l'absurde, en montrant que s'il y a un cycle de dépendance dans $\tilde{s}$, alors il en existe aussi un dans s. Cette preuve repose sur des lemmes techniques qui sont démontrés ci-dessous. Supposons que ce ne soit pas le cas. On va alors montrer qu'on est dans un des deux cas suivants (lemme 10) :

- il existe x tel que $x \in \wr x \int_{\tilde{s}}$ ou
- il existe x_1 et x_2 distincts vérifiant $x_1 \blacktriangleleft_{\tilde{s}} x_2 \blacktriangleleft_{\tilde{s}} x_1$

Dans le premier cas, d'après la remarque 4, $\phi = Intros_{\tilde{s}}(x)$ est un champ non récursif. De plus, par définition de $\circledcirc$, tous les champs introduisant ϕ sont non récursifs (sinon $\circledcirc$ aurait échoué). On a donc $x \in \wr x \int_s (= \wr \mathcal{B}_s(x) \int)$ ce qui permet de conclure immédiatement.

Dans le second cas, on commence par montrer avec le corollaire 2 du lemme 11 que si s est bien formée alors $x_1 \circlearrowright_s x_2$. On peut alors utiliser le lemme 12 pour en déduire que x_1 et x_2 sont définis par le même champ de $\tilde{s}$, ce qui par définition de $\wr \cdot \int$ sur un champ récursif est en contradiction avec $x_1 \blacktriangleleft_{\tilde{s}} x_2$. □

Lemme 10 (mauvaise formation)
*Soit s une espèce bien spécifiée n'ayant pas de clause **inherits**. Si s n'est pas bien formée, alors une des deux propriétés suivante est vérifiée :*

1. $\exists \phi \in s, \phi = \textbf{let}\ x\ = e$ *et* $x \in \wr x \int_s$
2. $\exists x_1, x_2 \in \mathcal{N}(s), (x_1 \blacktriangleleft_s x_2) \wedge (x_2 \blacktriangleleft_s x_1) \wedge Intros_s(x_1) \neq Intros_s(x_2)$

Preuve. Par hypothèse, il existe $x \in \mathcal{N}(s)$ tel que $x \blacktriangleleft_s x$. Soient $\{x_i\}_{i=1..n} \subset \mathcal{N}(s)$ vérifiant

$$\begin{cases} x_1 = x \\ x_n = x \\ \forall i\ x_{i+1} \in \wr x_i \int_s \end{cases}$$

Une telle suite existe par définition de $\blacktriangleleft_s$. Considérons maintenant x_1 et x_2. Si 1 n'est pas vérifiée, alors $x_1 \neq x_2$, car aucune méthode ne dépend directement d'elle même. On a alors les cas suivants : Si $Intros_s(x_1) = Intros_s(x_2)$, alors $Intros_s(x_1) = \textbf{let rec}\{x_1 = expr_1;\ x_2 = expr_2 \ldots\}$, et par définition de $\wr \cdot \int_s$ pour les définitions récursives, $x_2 \in \wr x_1 \int_s$, donc $Intros_s(x_1) \neq Intros_s(x_2)$. De plus, par hypothèse $x_1 \blacktriangleleft_s x_2$ et $x_2 \blacktriangleleft_s x_n = x$. La propriété 2 est bien vérifiée. □

Lemme 11 *Soient s une espèce bien spécifiée et $x_1, ..., x_n \in \mathcal{N}(s)$ $(n > 1)$, tels que $\forall i,\ x_{i+1} \in \wr \mathcal{B}_s(x_i) \int$. Si $\exists i$,* tel que $\neg(x_i \circlearrowright_s x_1)$, *alors $x_1 \blacktriangleleft_s x_n$.*

Preuve. Par induction sur n. Si $n = 2$, la propriété est immédiate par définition de $\wr\cdot\int_s$. Sinon, soit m le plus petit indice vérifiant $\neg(x_m \circlearrowright_s x_1)$. Par définition de m, on a $x_1 \circlearrowright_s x_{m-1}$. De plus, comme $\circlearrowright_s$ est une relation n'équivalence, $\neg(x_m \circlearrowright_s x_{m-1})$: on a $x_m \in \wr x_{m-1}\int_s$, soit par définition de $\blacktriangleleft_s$, $x_1 \blacktriangleleft_s x_m$. De plus, par hypothèse d'induction, on est dans un des deux cas suivant :

1. $x_m \circlearrowright_s x_{m+1} \ldots \circlearrowright_s x_n$.
2. $x_m \blacktriangleleft_s x_n$.

Dans le premier cas, par définition de $\blacktriangleleft_s$, $x_1 \blacktriangleleft_s x_n$. Dans le second, il suffit d'appliquer la transitivité de $\blacktriangleleft_s$ pour conclure de la même manière. □

Corollaire 2 (Récursion mutuelle) *Soient s une espèce bien spécifiée et $x_1, ..., x_n \in \mathcal{N}(s)$ $(n > 1)$, tels que $\forall i,\ x_{i+1} \in \wr\mathcal{B}_s(x_i)\int$ et $x_1 \in \wr\mathcal{B}_s(x_n)\int$. Alors, si s est bien formée, $\forall i, j\ x_i \circlearrowright_s x_j$*

Preuve. Supposons qu'il existe x_i tel que $\neg x_i \circlearrowright_s x_1$. En appliquant le lemme précédent, il vient $x_1 \blacktriangleleft_s x_1$, en contradiction avec l'hypothèse que s est bien formée. □

Remarque 4. En reprenant l'énoncé précédent dans le cas où $n = 1$ et s n'a pas de clause **inherits**, $Intros_s(x_1)$ est de la forme **let rec** $x_1 = \ldots$ (sinon on aurait $x_1 \in \wr x_1\int_s$).

Inversement, si $x_1 \in \wr x_1\int_s$, $Intros_s(x_1)$ est de la forme **let** $x_1 = \ldots$.

Lemme 12 *En utilisant les notations précédentes,*

$$\forall x \in \mathcal{N}(s), \mathcal{N}(Intros_{\tilde{s}}(x)) = \{y | x \circlearrowright_s y\}$$

Preuve.

Par induction sur le graphe d'héritage et la dérivation de $x \circlearrowright_s y$. On a les cas suivants :

NAMES-MUT-REC : x et y sont introduits par un même champ de $\mathbb{W}_1$, et sont donc définis dans le même champ de $\tilde{s}$

NAMES-INH : par hypothèse d'induction, x et y sont introduits par un même champ de $norm(s_h)$, donc de $\mathbb{W}_1$, et sont donc définis par le même champ de $\tilde{s}$.

NAMES-TRANS : $\exists z$, tel que $x \circlearrowright_s z \wedge z \circlearrowright_s y$. Par hypothèse d'induction, on a $Intros_{\tilde{s}}(x) = Intros_{\tilde{s}}(z)$ et $Intros_{\tilde{s}}(z) = Intros_{\tilde{s}}(y)$. Par unicité du champ introduisant z dans $\tilde{s}$, il vient $Intros_{\tilde{s}}(x) = Intros_{\tilde{s}}(y)$.

□

Théoreme 3 (Forme normale d'une espèce)
Pour toute espèce bien formée et bien typée s, il existe une espèce nfs en forme normale et vérifiant les propriétés suivantes :

- *noms : $\mathcal{N}(nfs) = \mathcal{N}(s)$ et $\mathcal{D}(nfs) = \mathcal{D}(s)$*

– *définitions :* $\forall x \in \mathcal{D}(s), \mathcal{B}_s(x) = \mathcal{B}_{nfs}(x)$

Preuve. D'après le théorème 2, $\tilde{s}$ est bien formée, donc $\blacktriangleleft_{\tilde{s}}$ est un ordre strict. Il n'y a donc plus qu'à réordonner les champs de $\tilde{s}$ de manière compatible avec cet ordre pour obtenir une espèce en forme normale. □

3.8 Espèces paramétrées

Nous pouvons maintenant aborder le cas des espèces paramétrées. Comme on va le voir, l'analyse de dépendances n'est pas modifiée en présence de paramètres. Seul l'environnement de typage est affecté. Toutefois, comme on le verra dans le chapitre 8, il est possible d'étendre l'analyse de dépendances aux méthodes des paramètres de collection, afin de prendre en compte plus finement les liens entre une espèce et ses paramètres.

On se donne tout d'abord une fonction $\mathcal{A}$ qui prend en argument une espèce s et un nom de collection c et retourne une interface en abstrayant les méthodes et en remplaçant **self** par c dans les types. En effet, un paramètre de collection de la forme "`c` **is** s" ajoute simplement une collection c d'interface $\mathcal{A}(s, c)$ dans l'environnement $\mathcal{C}$.

Définition 20 (abstraction) *Soient $s = \{x_i : \tau_i = e_i\}_{i=1..n}$ une espèce bien typée et bien formée et c un nom de collection. On définit* l'abstraction de s sur c *de la manière suivante :*

$$\mathcal{A}(s, c) = \langle x_i : \tau_i[self \leftarrow c] \rangle_{i=1..n}$$

Par ailleurs, un paramètre de collection peut bien sûr être instantié par une structure plus riche que celle qu'on attend. Par exemple, les polynômes peuvent être définis au dessus d'un anneau, mais il est possible d'utiliser l'implantation d'un corps pour instantier le paramètre. On va donc définir une relation de *sous-espèce*, $\preccurlyeq$ pour autoriser de telles implantations. Les restrictions mises sur le typage des espèces permettent de montrer une relation simple entre $\preccurlyeq$ et l'héritage.

Définition 21 (sous-espèce) *Soient s_1, s_2 deux espèces atomiques (bien typées et bien formées). On dit que s_1 est une sous-espèce de s_2, qu'on note $s_1 \preccurlyeq s_2$ si et seulement si :*

$$\mathcal{N}(s_2) \subset \mathcal{N}(s_1) \wedge \forall x \in \mathcal{N}(s_2), \mathcal{T}_{s_1}(x) = \mathcal{T}_{s_2}(x)$$

$\mathcal{T}_{s_i}(x)$ étant défini par la def.10

Lemme 13 (Héritage et sous-espèce) *Avec les notations de la définition précédente, si s_1 hérite de s_2, alors $s_1 \preccurlyeq s_2$.*

Preuve. Avec le théorème 3, on a $\mathcal{N}(s_1) \subset \mathcal{N}(s_2)$. De plus, le typage d'une espèce (def.10) implique que toutes les méthodes présentes dans l'espèce parent conservent leur type. □

Remarque 5. La relation $\preccurlyeq$ ne faisant apparaître que des types, elle est aisément étendue aux interfaces $\mathcal{A}(s_1, c)$ et $\mathcal{A}(s_2, c)$, pour tout nom de collection c.

On présente maintenant les règles de typage pour les espèces paramétrées. Pour cela, on enrichit le langage des types d'espèces renvoyé par $\mathcal{E}$ de la manière suivante :

Définition 22 (type d'espèce)

$$t_s ::= \{x_i : \tau_i = e_i\} \mid (x \ \textbf{\textit{in}} \ \tau)t_s \mid (c \ \textbf{\textit{is}} \ < x_i : \tau_i >)t_s$$

Notation 6 *Dans les règles d'inférence ci-dessous, on désignera par a un type d'espèce atomique, c'est à dire de la forme* $\{x_i : \tau_i = e_i\}$

De plus, on emploiera la notation $x\,.\,t$ *pour désigner un paramètre quelconque (d'entité ou de collection) d'une espèce ou d'un type d'espèce. Une espèce paramétrée sera notée*

Enfin, on notera e^s *une expression de la forme* $s(e_1, \ldots, e_{p_f})$*, où* p_f *représente le nombre de paramètres de l'espèce* s*.*

Définition 23 (Bonne formation d'un type d'espèce)
Un type d'espèce t_s *est dit bien formé dans l'environnement* $\mathcal{C}$ *(*$\mathcal{C} \vdash t_s$*) si et seulement si les conditions suivantes sont remplies :*

$$\frac{\forall i, \mathcal{C} \vdash \tau_i}{\mathcal{C} \vdash \{x_i : \tau_i = e_i\}} \qquad \frac{\mathcal{C} \vdash \tau \qquad \mathcal{C} \vdash t_s}{\mathcal{C} \vdash (x \ \textbf{\textit{in}} \ \tau)t_s}$$

$$\frac{\forall i, \mathcal{C} \vdash \tau_i \qquad \mathcal{C} + c :< x_i : \tau_i > \vdash t_s}{\mathcal{C} \vdash (c \ \textbf{\textit{is}} \ < x_i : \tau_i >)t_s}$$

La figure 3.2 présente les règles de typage des espèces paramétrées. Comme précédemment, on suppose que tous les types introduits dans les règles sont bien formés. Les règles ENT-PRM et COLL-PRM permettent de placer les paramètres d'une espèce dans l'environnement correspondant ($\mathcal{C}$ pour les paramètres de collection et Γ pour les paramètres d'entité). On peut aussi remarquer que COLL-PRM exige que e^s ait un type atomique : il n'y a pas de notion d'espèce "d'ordre supérieur" qui prendrait en paramètre une espèce comportant des paramètres "libres". Par contre, e^s peut bien sûr être une espèce paramétrée dont tous les paramètres sont instanciés.

Les règles ENT-INST et COLL-INST vérifient qu'on instantie les paramètres d'une espèce existante avec des arguments du bon type (de la bonne interface dans le cas d'un paramètre de collection). Notons qu'un paramètre de collection ne peut être instantié que par un identificateur $\mathcal{C}$, c'est à dire une

FIG. 3.2 – Règles de typage pour les espèces paramétrées

ENT-PRM

$$\frac{\Omega, \Gamma + x : \tau \vdash \quad \textbf{species } s(prms) \textbf{ inherits } {e^s}_1, ..., {e^s}_{h_f} = \Phi_1 ... \Phi_n : t_s}{\Omega \vdash \textbf{species } s(x \textbf{ in } \tau, prms) \textbf{inherits } {e^s}_1, ..., {e^s}_{h_f} = \Phi_1 ... \Phi_n : (x \textbf{ in } \tau) t_s}$$

COLL-PRM

$$\frac{\begin{array}{c}\mathcal{C}, \Omega \vdash e^s : a \\ \mathcal{C} + c : \mathcal{A}(a, c), \Omega \vdash \\ \textbf{species } s(prms) \textbf{ inherits } {e^s}_1, ..., {e^s}_{h_f} = \Phi_1 ... \Phi_n : t_s\end{array}}{\begin{array}{l}\mathcal{C}, \Omega \vdash \quad \textbf{species } s(c \textbf{ is } e^s, prms) \textbf{ inherits } {e^s}_1, ..., {e^s}_{h_f} = \\ \Phi_1 ... \Phi_n : (c \textbf{ is } \mathcal{A}(a, c)) t_s\end{array}}$$

VAR-SPEC

$$\frac{\mathcal{E}(s) = t_s}{\Omega, \mathcal{E} \vdash s : t_s}$$

ENT-INST

$$\frac{\Omega \vdash e^s : (x \textbf{ in } \tau) t_s \qquad \Omega \vdash e : \tau}{\Omega \vdash e^s(e) : t_s[x \leftarrow e]}$$

COLL-INST

$$\frac{\mathcal{C}, \Omega \vdash e^s : (c_1 \textbf{ is } i_1) t_s \qquad \mathcal{C}(c_2) = i_2 \qquad i_2 \preccurlyeq \mathcal{A}(i_1, c_2)}{\mathcal{C}, \Omega \vdash e^s(c_2) : t_s[c_1 \leftarrow c_2]}$$

PRM-INHERIT

$$\frac{\begin{array}{c}\forall h \leq h_f, \Omega \vdash {e^s}_h : a_h \\ \Omega \vdash \textbf{species } s \textbf{ inherits } a_1, ..., a_{h_f} = \quad \phi_1 ... \phi_n : \quad \{x_i : \tau_i = e_i\}\end{array}}{\Omega \vdash \textbf{species } s \textbf{ inherits } {e^s}_1, ..., {e^s}_{h_f} = \quad \phi_1 ... \phi_n : \quad \{x_i : \tau_i = e_i\}}$$

collection existante (dans l'environnement global ou en tant que paramètre de l'espèce courante). Par ailleurs, comme le reste du type d'espèce t_s peut dépendre du paramètre qu'on est en train d'examiner, il faut substituer à ce paramètre son instantiation.

Enfin, dans la règle PRM-INHERIT, on vérifie deux choses :

- toutes les espèces dont on hérite sont instanciées avec des paramètres admissibles
- Ces espèces sont *complètement* instanciées. Là encore, on ne peut pas hériter d'une espèce comportant des paramètres "libres".

Notation 7 *Soit s une espèce définie par*

species s ... **inherits** ... $s'(e_1, \ldots, e_{p_f})$ ...

telle que $\mathcal{E}(s') = (c_1 \,.\, t_1) ... (c_{p_f} \,.\, t_{p_f}) a$. *on notera* $Inst_s(c_p)$ *l'instantiation de* c_p *dans s, c'est à dire* e_p.

Remarque 6. Les définitions de $\mathcal{B}_s(x)$ et $\mathcal{T}_s(x)$ supposant que les espèces parent sont des listes de méthodes, la substitution des règles *coll-inst* et *ent-inst* s'étendent à la définition du corps et du type d'une méthode héritée.

La mise en forme normale d'une espèce paramétrée s'effectue de la même manière que pour une espèce atomique, dans l'environnement enrichi des paramètres. De même, avec PRM-INHERIT, on commence par instantier les paramètres des espèces parents, de sorte que la résolution de l'héritage s'effectue sur des espèces atomiques.

On va maintenant examiner comment l'héritage se comporte vis-à-vis du sous-typage en présence de paramètre. Plus précisément, on va voir que le lemme 13 peut être étendu au cas où s_1 et s_2 sont des espèces paramétrées, à condition de respecter l'instantiation des paramètres de s_2 donnée dans la clause **inherits** de s_1.

Lemme 14 (Sous-espèce et paramètres) *Soient s_2 une espèce de type*

$$s_2 : (y_1 \,.\, t^2{}_1)...(y_{p_2} \,.\, t^2{}_{p_2})a_2$$

et s_1 une espèce définie par

species s_1 $(x_1$. $t^1{}_1,\ldots,x_{p_1}$. $t^1{}_{p_1})$ **inherits** $s_2(e_1,\ldots,e_{p_2})=$ **end**

Alors si s_1 est bien typée, on a $\forall e'_1,...,e'_{p_1}$ tels que $s_1(e'_1,...,e'_{p_1})$ est bien typée, de type $t^1{}_s$, on a

$$t_1{}^s \preccurlyeq (a_2[y_p \leftarrow e_p]_{p\leq p_2})[x_p \leftarrow e'_p]_{p\leq p_1}$$

Notation 8 *Dans les règles d'inférence ci-dessous, on désignera par def_1 la définition de s_1 ci-dessus, et $\widetilde{def}_1$ la définition de s_1 ou $s_2(e_1,...,e_{p_2})$ a été remplacé par le type d'espèce associé, comme dans la règle* PRM-INHERIT

Preuve. Par induction sur p_1
Si $p_1 = 0$. On a deux cas suivant que y_1 est un paramètre d'entité ou de collection. Dans le premier cas, la dérivation de typage de s_1 est de la forme :

$$\text{PRM-INHERIT}\ \dfrac{\text{ENT-INST}\ \dfrac{\Omega \vdash e_1 : t_1 \quad \Omega \vdash s_2 : (y_1\ \mathbf{in}\ t_1)t_s}{\Omega \vdash s_2(e_1) : t_1[y_1 \leftarrow e_1]} \quad \vdots}{\Omega \vdash s_2(e_1,...,e_{p_2}) : a_2[y_i \leftarrow e_i]} \qquad \dfrac{\vdots}{\Omega \vdash \widetilde{def}_1 : a_1}\ \text{WELL-TYPED-SPEC}}{\Omega \vdash def_1 : a_1}$$

Le second cas est semblable, ENT-INST étant remplacé par COLL-INST. Dans les deux cas, on peut conclure comme dans le cas de deux espèces atomiques (lemme 13), avec les prémisses de WELL-TYPED-SPEC.

Si $p > 0$, la dernière règle de la dérivation de typage de s_1 est ENT-PRM ou COLL-PRM suivant que x_1 est un paramètre d'entité ou de collection. On ne donne ici que le premier cas, le second s'en déduisant immédiatement :

$$ent - prm\ \frac{\begin{array}{c}\vdots\\ \Omega, \Gamma + (x : \tau) \vdash \textbf{species}\ s'_1(x_2 \,.\, t_2, ...x_{p_1} \,.\, t_{p_1}) ... : t_s\end{array}}{\Omega, \Gamma \vdash def_1 : (x\ \textbf{in}\ t_1)t_s}$$

Par hypothèse d'induction, $\forall e'_2, ..., e'_{p_1}$ tels que $\Omega, \Gamma + (x : \tau) \vdash e'_i : t_i$, on a, avec t'_s le type de $s'_1(e'_2, ..., e'_{p_1})$, $t'_s \preccurlyeq (a_1[y_p \leftarrow e_p]_{p \leq p_2})[x_p \leftarrow e'_p]_{2 \leq p \leq p_1}$, Donc, pour toute expression e'_1 de type t_1 dans Ω, Γ, $t'_s[x_1 \leftarrow e'_1] \preccurlyeq (a_1[y_p \leftarrow e_p]_{p \leq p_2})[x_p \leftarrow e'_p]_{p \leq p_1}$. Or avec ENT-INST, $s_1(e'_1, e'_2, ..., e'_{p_1})$ a le même type que $(s'_1(e'_2, ..., e'_{p_1}))s[x_1 \leftarrow e'_1]$, c'est à dire $t'_s[x_1 \leftarrow e'_1]$,
□

Remarque 7. Ce lemme s'étend trivialement au cas où s_1 définit ou déclare des champs supplémentaires (en supposant bien sûr que ces champs sont eux-mêmes bien typés). De même, en cas d'héritage multiple, s_1 est une sous-espèce de toutes les espèces dont elle hérite.

3.9 Ajouter les preuves et les propriétés

Nous allons maintenant examiner les raffinements qu'il faut apporter à l'analyse de dépendances des sections précédentes pour pouvoir l'étendre aux champs **property** et **theorem**. En premier lieu, nous allons voir que le concept de dépendance se divise en deux : les def-dépendances (cf 3.9.1) des théorèmes vis-à-vis des algorithmes dont ils prouvent la correction, et les decl-dépendances, de même nature que celles qui figurent dans les langages de programmation. Ensuite, nous verrons qu'il n'est plus toujours possible, à cause justement des def-dépendances, de garder toutes les définitions héritées d'une espèce parent : certaines doivent être effacées. Nous verrons comment cela se traduit au niveau de l'algorithme d'analyse de dépendance, et comment en particulier raffiner cet algorithme afin qu'il y ait le moins d'effacements possible.

3.9.1 Def-dépendances et decl-dépendances

Lorsqu'on veut introduire des preuves de théorèmes, la notion de dépendance qu'on a introduite dans la section précédente devient rapidement trop limitée. En effet, comme l'a remarqué Sylvain Boulmé [Bou00] dans sa thèse, une preuve peut dépendre de deux manières d'une méthode **self**!f. On peut avoir besoin de connaître le type de f, comme précédemment, mais il est parfois nécessaire de savoir comment f a été définie. Par exemple, dans un

`monoïde`, on peut exprimer que l'opération `plus` est associative, soit en FOC :

```
property assoc =
  all x y z in self,
    self !eq(self !plus(self !plus(x,y),z)
             self !plus(x,self !plus(y,z)))
```

Par la suite, on peut prouver cette propriété pour l'implantation de `plus` sur les entiers de Peano : $0+n = n$ and $succ(m)+n = succ(m+n)$. On le fait par induction sur x, y et z, et on utilise pour cela *la définition exacte* de `plus`. En d'autres termes, la preuve qu'on va obtenir *dépend de la définition* de plus. Nous appelerons donc *def-dépendance* ce nouveau type de dépendances.

Par ailleurs, la preuve de `assoc` n'a *a priori* pas besoin d'utiliser la définition de l'égalité sur les entiers. Seules ses propriétés (et en particulier la transitivité) sont importantes. Par contre, on a besoin de savoir qu'elle a été *déclarée* avec le type **self** $\rightarrow$ **self** $\rightarrow$ *bool*. Sans cela, on ne serait même pas capable de s'assurer que l'énoncé est correct. Nous emploierons donc le terme de *decl-dépendance* pour ce type de dépendances, qui est celui que nous avons déjà rencontré dans les sections précédentes : dans les corps des méthodes calculatoires, il n'y a que des decl-dépendances, comme le montre la règle SELF CALL dans le typage des expressions FOC (fig. 3.1).

En ce qui concerne les cycles de dépendances, il n'y a pas de changement : il faut toujours rejeter les cycles de dépendances, *def* ou *decl*. Par contre, la phase de résolution de l'héritage devient plus compliquée, et en particulier les redéfinitions de méthodes vont impliquer plus de deux champs. En effet, lorsqu'on remplace la définition d'une fonction f héritée de s_1 par une nouvelle définition, héritée ou non, il faut effacer toutes les preuves qui *def-dépendent* de f : elles ne sont plus correctes dans le nouveau contexte. Cela implique donc que la preuve soit refaite pour la nouvelle définition de f (le programmeur FOC avisé évitera donc d'utiliser trop de def-dépendances dans ses preuves). Si on poursuit l'exemple précédent, on peut décider de remplacer la définition de `plus` par un algorithme qui utilise en interne une représentation binaire des entiers (ce qui est nettement plus efficace), mais il faudra alors reprendre la preuve de `assoc` pour cette nouvelle définition.

3.9.2 Obligations de preuves

En dehors des propriétés explicitement écrites par l'utilisateur, certaines obligations de preuves sont requises par le système lui même[1].

Congruences Toutes les méthodes des espèces s construites au dessus de `setoid` (l'espèce de la hiérarchie où est déclarée l'égalité) doivent être

[1] A l'heure actuelle, seules le traitement des preuves de terminaison est partiellement implanté dans le compilateur `focc`.

compatibles avec l'égalité, c'est à dire la méthode `equal` de s. En effet, cette méthode est *a priori* différente de l'égalité structurelle de FOC, qui sera traduite par le prédicat EQ de COQ (et l'égalité structurelle de OCAML).

En d'autre termes, pour toute méthode m de type $c_1 \to c_2$, où c_1 et c_2 sont des collections descendant de `setoid` dans le contexte courant (en particulier si $c_1 =$ **self** quand l'espèce courante descend de `setoid`), on doit montrer la propriété

property `m_cong` : **all** `x y` **in** c_1,
c_1 `!equal(x,y)->c_2 !equal( !m(x), !m(y))` ;

Remarque 8. Si c_2 n'est pas un setoïde, on n'établit pas un telle propriété.

Plus généralement, si m a k arguments de type respectivement $\tau_1, ..., \tau_k$, et de type de retour τ_r la propriété à prouver sera

property `m_cong` :
all x_1 y_1 **in** τ_1, ...
all x_k y_k **in** τ_k,
$egalite_1(x_1,y_1) \to \ldots$
$egalite_k(x_k,y_k) \to$
τ_r `!equal(` $!m(x_1,\ldots,x_k), !m(y_1,\ldots,y_k)$ `)` ;

où $egalite_i$ est
- `tau_i !equal` si `tau_i` est une collection implantant `setoide`
- `#base_eq` sinon

Les énoncés de ce type expriment le fait que `equal` n'est pas une relation d'équivalence quelconque sur **self** mais bien *l'égalité* dont on veut munir l'ensemble sous-jacent. Si on omet ces propriétés, il devient impossible d'utiliser la réécriture dans la plupart des preuves FOC : il est impossible de remplacer une sous-expression e d'un terme par une sous expression e' qui lui est "égale" au sens des setoïdes, puisqu'il n'y a pas de preuve que cette égalité est préservée par des appels de fonction.Par exemple, si on se place dans un monoide, avec la propriété $\forall x, 1 * x = x$ et en omettant l'énoncé de la congruence $\forall x, y, z, t (x = z) \to (y = t) \to (x * y) = (z * t)$, il devient impossible de prouver l'énoncé suivant :

$$\forall x, y, x * (1 * y) = (x * y)$$

En effet, bien qu'on sache que $(1 * y) = y$, on ne peut pas remplacer $(1 * y)$ par y dans l'égalité de l'énoncé.

Preuves de terminaison Comme on l'a dit dans les section précédente, toute définition introduite par un **let rec** doit être accompagnée d'une preuve que les appels récursifs terminent. Deux cas peuvent se présenter :

En premier lieu, la récursion peut être structurelle (au sens de Coq), c'est à dire que les appels récursifs ne mettent en jeu que des arguments structurellement plus petits que le paramètre de départ, à la suite d'un pattern-matching. Ces définitions ne posent pas de problème, puisqu'il suffit de les traduire avec un `Fixpoint` en Coq (voir le chapitre 6) et de vérifier que le terme obtenu est accepté par Coq pour être assuré de la terminaison.

Par contre, si on doit s'appuyer sur un ordre bien fondé pour montrer la terminaison de la méthode, les choses se compliquent nettement : pour chaque appel récursif, il va falloir générer une obligation de preuve, afin que l'utilisateur montre qu'il utilise des arguments plus petits (au sens de l'ordre choisi) que le paramètre initial. De plus, la synthèse de cet énoncé oblige à conserver l'environnement dans lequel se fait l'appel récursif en question. Une fois que toutes les preuves ont été données par l'utilisateur – y compris la preuve que l'ordre utilisé est bien formé – la définition peut être acceptée.

Il convient de noter que cette fois-ci les lemmes ne dépendent pas de la définition récursive : ils restent valables même si on redéfinit la (les) méthode(s) dont ils permettaient de prouver la terminaison. Par contre, ils n'ont en général pas d'intérêt pour la nouvelle preuve de terminaison. Par exemple, dans un monoïde, on peut définir l'exponentielle ($x^n = \underbrace{x * ... * x}_{n fois}$) de la manière suivante :

```
let rec exp (x in self,n in int) =
  if n <= 0 then self !one
  else self !mult(x,self !exp(x,(n−1))) ;
```

L'ordre à utiliser est évidemment l'ordre sur les naturels (il n'y a un appel récursif que dans le cas $n > 0$) et il faut donc prouver la propriété $\forall n > 0, n - 1 < n$. Cette propriété reste vraie si on redéfinit `exp` de manière plus efficace en utilisant la dichotomie :

```
let rec exp(x in self,n in int)=
 if n <= 0 then self !one
 else
 if n mod 2 = 0 then self !exp(self !mult(x,x),n/2)
 else self !mult(x,self !exp(self !mult(x,x),(n−1)/2) ;
```

Par contre, elle n'offre plus d'intérêt pour prouver la terminaison de `exp`, où il faut montrer

$$\begin{cases} \forall n > 0, (n \bmod 2) = 0 \rightarrow \frac{n}{2} < n \\ \forall n > 0, \neg((n \bmod 2) = 0) \rightarrow \frac{n-1}{2} < n \end{cases}$$

Pour l'analyse de dépendance, on peut donc considérer que ces lemmes sont indépendants des fonctions récursives, et que chaque (re)définition de méthode(s) récursive(s) est accompagnée de ses propres lemmes de terminaison. Dans le chapitre 6 concernant la traduction en Coq, nous reviendrons

plus en détail sur la génération et l'utilisation des lemmes de terminaison. Notons que la version actuelle du compilateur `focc` génère les énoncés des lemmes de terminaison, mais ne permet pas encore d'en faire la preuve : ils sont introduits comme hypothèses dans la traduction Coq.

3.9.3 Définitions de base

On étend trivialement les définitions des sections précédentes au cas des propriétés et des théorèmes :

Définition 24 (noms de méthodes)

$$\mathcal{N}(\textbf{\textit{theorem}}\ x = ...) = \{x\} \qquad \mathcal{N}(\textbf{\textit{property}}\ x = ...) = \{x\}$$

$$\mathcal{D}\,(\textbf{\textit{theorem}}\ x = ...) = \{x\} \qquad \mathcal{D}\,(\textbf{\textit{property}}\ x = ...) = \emptyset$$

Définition 25 (Dépendances des propriétés) *Si une proposition p est une simple expression, ses dépendances ont déjà été définies par la def.12.*

Sinon, on les définit par induction sur la structure de p :

$$\lbag p_1\ \textbf{\textit{and}}\ p_2 \rbag = \lbag p_1 \rbag \cup \lbag p_2 \rbag \qquad \lbag p_1\ \textbf{\textit{or}}\ p_2 \rbag = \lbag p_1 \rbag \cup \lbag p_2 \rbag$$

$$\lbag p_1\ \rightarrow\ p_2 \rbag = \lbag p_1 \rbag \cup \lbag p_2 \rbag \qquad \lbag \textbf{\textit{not}}\ p \rbag = \lbag p \rbag \qquad \lbag \textbf{\textit{all}}\ x\ \textbf{\textit{in}}\ \tau, p \rbag = \lbag p \rbag$$

$$\lbag \textbf{\textit{ex}}\ x\ \textbf{\textit{in}}\ \tau, p \rbag = \lbag p \rbag$$

On peut noter que les quantification **all** et **ex** ne changent pas les dépendances de la propriété p : celles-ci prennent en compte les appels de méthodes présentes dans p et non les variables libres ou liées de p.

Définition 26

$$\begin{array}{lcl} \lbag \textbf{\textit{property}}\ x = p \rbag & = & \lbag p \rbag \\ \lbag \textbf{\textit{theorem}}\ x = p\ \textbf{\textit{proof :}} \ldots \textbf{\textit{decl}}\ x_1, \ldots, x_n; \ldots \rbag & = & \lbag p \rbag \cup \{x_i\}_{i=1..n} \end{array}$$

Dans l'attente d'un langage de preuve propre à Foc, Les def-dépendances éventuelles d'un théorème, qu'on notera $\llbracket . \rrbracket$, sont données explicitement par l'utilisateur :

Définition 27 (def-dépendances)

$$\llbracket \textbf{\textit{proof :}} \ldots \textbf{\textit{def :}} x_1 \ldots x_n \ldots \rrbracket = \{x_i\}_{i=1\ldots n}$$

Soit ϕ un champ
– si ϕ est de la forme ***theorem*** $x = \ldots proof$ *alors* $\llbracket \phi \rrbracket = \llbracket proof \rrbracket$
– sinon $\llbracket \phi \rrbracket = \emptyset$

3.9.4 Dépendances vis-à-vis du type support

Dans la section précédente, nous ne nous sommes pas étendus sur la question des dépendances des différentes méthodes vis à vis du type support. En effet, la syntaxe de FOC n'autorise pas à l'heure actuelle le type support à dépendre d'une méthode, et il n'y a donc aucun risque de voir un cycle de dépendance impliquant **rep**.

Toutefois, si on cherche à faire la distinction entre def- et decl-dépendances, les choses changent légèrement : il y a des decl- et def- dépendances vis à vis du type support. Comme on s'interdit une redéfinition du type support au cours de l'héritage (3.1), cette distinction peut apparaître secondaire en pratique, puisque la seule différence entre les deux formes de dépendances a lieu lors des redéfinitions. Toutefois, cette restriction n'est faite que pour des raisons pratiques de traduction en OCAML et il est tout à fait possible de formaliser les dépendances vis à vis du type support comme celles de toutes les autres méthodes. La seule différence réside dans la façon dont on retrouve ces dépendances lors de l'analyse des expressions FOC.

Définition 28 (Dépendances vis-à-vis du type support) *Soit une expression e, si une sous-expression de e a le type **self**, il y a une decl-dépendance vis-à-vis du type support. De plus, si au cours du typage de e une des étapes d'unification Mgu utilise une des deux règles* SELF*, alors il y a une def-dépendance.*

En effet, si on est obligé d'identifier **self** et sa représentation concrète dans s, e n'est plus typable dans un environnement où **self** est lié à un autre type. Par exemple, si on définit l'espèce suivante :

```
species counter =
  rep = int;
  let inc in self → self = fun x → x + 1;
end
```

`inc` def-dépend du type support `int` : pour donner à `inc` le type attendu, **self** → **self** il faut savoir que le type **self** est, dans l'espèce `counter`, lié à `int`

De telles def-dépendances posent un nouveau problème. En effet, elles peuvent apparaître dans l'énoncé – c'est à dire le type – d'un théorème ou d'une propriété. Par exemple, on aurait pu vouloir ajouter à l'espèce `counter` un théorème qui montre que `inc` renvoie un résultat strictement plus grand que l'élément de **self** passé en argument :

```
theorem inc_spec : all x in self, self !inc(x) >= x + 1 proof : ...
```

L'énoncé de `inc_spec` a une def-dépendance vis-à-vis du type support. En d'autres termes, il est impossible de construire une interface correcte de

l'espèce `counter`, puisqu'on ne peut pas abstraire le type support. Comme on veut que toute espèce ait une interface associée (1.7.1), une telle définition doit être rejetée.

Pour cela, les règles de typage des propositions (figure 3.3), n'utilisent pas directement l'environnement Σ tel que défini par l'espèce, mais une restriction,

$$\Sigma^* = \Sigma[\mathcal{B}_s(rep) \leftarrow \bot]$$

qui abstrait une éventuelle définition de **rep**. De cette manière une def-dépendance vis-à-vis de **rep** au niveau des propriétés sera rejetée comme mal typée. En effet, les règles d'unification SELF1 et SELF2 ne pourront être appliquées dans l'environnement Σ^* : on ne pourra pas confondre le type support et sa représentation concrète.

Par ailleurs, la règle EXPR transforme implicitement toute expression booléenne trouvée dans un énoncé en une propriété. Enfin, suivant l'isomorphisme de Curry-Howard on étend $\mathcal{T}_s(x)$ aux méthodes introduites par les champs **theorem** et **property** comme étant l'énoncé de la méthode x dans l'espèce s (le corps de la méthode étant $\bot$ dans le cas de **property** et la preuve dans le cas de **theorem**)

FIG. 3.3 – Règles de typage pour les énoncés

$$[\text{EXPR}]\ \frac{\Omega, \Sigma^* \vdash expr : bool}{\Omega, \Sigma \vdash expr : prop} \qquad [\text{NOT}]\ \frac{\Omega \vdash p : prop}{\Omega \vdash \textbf{not}\ p : prop}$$

$$[\text{AND}]\ \frac{\Omega \vdash p_1 : prop \qquad \Omega \vdash p_2 : prop}{\Omega \vdash p_1\ \textbf{and}\ p_2 : prop} \qquad [\text{OR}]\ \frac{\Omega \vdash p_1 : prop \qquad \Omega \vdash p_2 : prop}{\Omega \vdash p_1\ \textbf{or}\ p_2 : prop}$$

$$[\text{IMP}]\ \frac{\Omega \vdash p_1 : prop \qquad \Omega \vdash p_2 : prop}{\Omega \vdash p_1\ \rightarrow\ p_2 : prop} \qquad [\text{ALL}]\ \frac{\Omega, \Gamma + x : \tau \vdash p : prop}{\Omega, \Gamma \vdash \textbf{all}\ x\ \textbf{in}\ \tau, p : prop}$$

$$[\text{EX}]\ \frac{\Omega, \Gamma + x : \tau \vdash p : prop}{\Omega, \Gamma \vdash \textbf{ex}\ x\ \textbf{in}\ \tau, p : prop}$$

3.9.5 Résolution de l'héritage

Contrairement à la notion de dépendance d'un champ (def.26), la résolution de l'héritage ne peut pas s'étendre directement. En effet, il faut tenir

compte des def-dépendances, ce qui revient à dire qu'une méthode qui était définie dans une espèce s_1 peut redevenir abstraite dans une espèce s qui hérite de s_1. Pour cela, on va définir avec $\mathcal{B}_s(x)$ une fonction auxiliaire, $\mathbb{I}_s(x)$, qui rappelle l'origine de x dans l'espèce s, et servira lorsqu'on devra faire des effacements. Comme dans les sections précédentes, on considérera une espèce s bien spécifiée et définie par :

species s **inherits** $s_1,\ldots,s_{h_f} = \phi_1,\ldots,\phi_m$ **end**

Définition 29 (Corps d'une méthode dans une espèce) *Soient s une espèce bien spécifiée, et $x \in \mathcal{N}(s)$. $\mathcal{B}_s(x)$ et $\mathbb{I}_s(x)$ sont définis récursivement de la manière suivante :*

- *si $\forall h \leq h_f,\ x \notin \mathcal{D}(s_h) \wedge \forall i \leq m,\ x \notin \mathcal{D}(\phi_i)$ alors $\mathcal{B}_s(x) = \bot$.*
- *si $\exists i \leq m,\ \phi_i$ est **let** $x = expr$ alors $\mathcal{B}_s(x) = expr$, et $\mathbb{I}_s(x) = h_f + 1$.*
- *si $\exists i \leq m,\ \phi_i$ est **let rec** $\{x_1 = expr_1 \ldots x_l = expr_l\}$, et $x_j = x$ alors $\mathcal{B}_s(x) = expr_j$ et $\mathbb{I}_s(x) = h_f + 1$*
- *si $\exists i \leq m,\ \phi_i$ est **theorem** x : ...proof alors $\mathcal{B}_s(x) = proof$, et $\mathbb{I}_s(x) = h_f + 1$*
- *Sinon, soit h le plus grand index tel que $x \in \mathcal{D}(norm(s_h))$ alors $\mathcal{B}_s(x) = \mathcal{B}_{s_h}(x)$, et $\mathbb{I}_s(x) = h$*

On conserve la définition de $\mathcal{D}(s)$ de la section 3.2 (def.1).

Nous pouvons également donné la définition de la fonction $x \leftarrow s$, qui donne l'origine de la définition de x dans l'espèce s :

Définition 30 (Origine d'une méthode) *Soit s une espèce définie par*

species s **inherits** $s_1(l_1),\ldots,s_{h_f}(l_{h_f}) = \phi_1 \ldots \phi_m$

Pour toute méthode $x \in \mathcal{D}(s)$ définie dans s, on définit l'origine *de x dans s, notée $x \leftarrow s$, de la manière suivante :*

$$\frac{x \in \mathcal{D}(\phi_i)}{x \leftarrow s = s} \qquad \frac{\forall i, x \notin \mathcal{D}(\phi_i) \qquad x \leftarrow \mathbb{I}_s(x) = s'}{x \leftarrow s = s'}$$

$\mathbb{I}_s(x)$ étant l'espèce d'où provient x parmi les espèces s_h dont s hérite directement *(définition def.29).*

Nous pouvons maintenant définir les dépendances de x dans une espèce s. Comme on l'a vu précédemment, il faut tenir compte non seulement de $\mathcal{B}_s(x)$, comme avec les champs calculatoires, mais aussi des dépendances de $\mathcal{T}_s(x)$, c'est à dire des énoncés des propriétés. En effet, un type pouvant maintenant être une proposition et contenir des expressions, il est susceptible d'avoir des dépendances vis-à-vis des autres méthodes. Notons que cela ne s'applique qu'aux decl-dépendances, puisqu'il ne peut y avoir de def-dépendances dans les types.

Définition 31 (Dépendances dans une espèce)
Soient s une espèce bien spécifiée et $x \in \mathcal{N}(s)$. On définit $\wr x \int_s$ et $\parallel x \rrbracket_s$ de la manière suivante :

- *Si x est une fonction (ou une signature de fonction), ses dépendances sont calculées par la définition 16.*
- *Sinon, $\wr x \int_s = \wr \mathcal{B}_s(x) \int \cup \wr \mathcal{T}_s(x) \int$ et $\parallel x \rrbracket_s = \parallel \mathcal{B}_s(x) \rrbracket$*

Par ailleurs, $\blacktriangleleft_s$ est maintenant définie comme la clôture transitive des def- et des decl-dépendances :

Définition 32

$$x_1 \blacktriangleleft_s x_2 \iff \exists \{y_i\}_{i=1\ldots n} \text{ tel que } \begin{array}{l} y_1 \circlearrowleft_s x_1 \\ y_n \circlearrowleft_s x_2 \\ \forall i < n \; y_{i+1} \in \wr y_i \int_s \cup \parallel y_i \rrbracket_s \end{array}$$

$$x_1 <_s^{def} x_2 \widehat{=} \exists \{y_i\}_{i=1\ldots n} \text{ tel que } y_1 = x_1, y_n = x_2, \forall i < n \; y_{i+1} \in \parallel y_i \rrbracket_s$$

*s est dite **bien formée** si $\forall x \in \mathcal{N}(s) \neg x \blacktriangleleft_s x$.*

3.9.6 Def-dépendances et effacement

On se donne maintenant deux nouvelles définitions, permettant de formaliser l'effacement d'une définition dans une espèce. En particulier, on notera qu'on peut avoir à effacer plusieurs champs les uns après les autres s'il y a une chaîne de dépendance.

Définition 33 (Effacement d'une définition)

$\mathbb{E}(\textbf{\textit{let}}\ x\ \textbf{\textit{in}}\ \tau = e)$	=	$\textbf{\textit{sig}}\ x\ \textbf{\textit{in}}\ \tau$
$\mathbb{E}(\textbf{\textit{theorem}}\ x : prop = demo)$	=	$\textbf{\textit{property}}\ x : prop$
$\mathbb{E}(\textbf{\textit{let rec}}\ x_1\ \textbf{\textit{in}}\ \tau_1 = e_1\ \ldots\ \textbf{\textit{and}}\ x_n\ \textbf{\textit{in}}\ \tau_n = e_n)$	=	$\textbf{\textit{sig}}\ x_1\ \textbf{\textit{in}}\ \tau_1; \ldots;$ $\textbf{\textit{sig}}\ x_n\ \textbf{\textit{in}}\ \tau_n$
$\mathbb{E}(\textbf{\textit{rep}} = \tau)$	=	$\textbf{\textit{rep}}$
$\mathbb{E}(m)$	=	m *sinon*

Définition 34 (Effacement dans un contexte)
Soit $\mathcal{N}$ une liste de noms de méthode. On définit l'effacement dans le contexte de $\mathcal{N}$, $\mathbb{E}_{\mathcal{N}}(.)$, de la manière suivante.

$$\mathbb{E}_{\mathcal{N}}(\emptyset) = \emptyset$$

$$\mathbb{E}_{\mathcal{N}}(m; l) = m; \mathbb{E}_{\mathcal{N}}(l) \text{ si } m \text{ est abstrait}$$

$\mathbb{E}_{\mathcal{N}}(m; l) = \mathbb{E}(m); \mathbb{E}_{\mathcal{N} \cup \mathcal{N}(m)}(l)$ *si* $\parallel m \rrbracket \cap \mathcal{N} \neq \emptyset$
$\mathbb{E}_{\mathcal{N}}(m; l) = m; \mathbb{E}_{\mathcal{N}}(l)$ *si* $\parallel m \rrbracket \cap \mathcal{N} = \emptyset$

Par construction, $\mathbb{E}_{\mathcal{N}}(\cdot)$ transforme une vue d'une espèce en une vue moins définie au sens de $\succcurlyeq$. Cette propriété sera affinée lors de la définition de la traduction en mixDRec (def.88), mais on peut d'ores et déjà en donner une formulation au niveau des espèces.

Lemme 15 *Soient s une espèce en forme normale dont la liste de champs est $\mathbb{W}$, et $\{x_i\}_{i=1..n}$ un ensemble de noms de méthodes. On a les propriétés suivantes :*

$$\mathcal{N}(\mathbb{E}_{\{x_i\}}(l)) = \mathcal{N}(l)$$

$$\mathcal{D}\left(\mathbb{E}_{\{x_i\}}(l)\right) \subset \mathcal{D}(l)$$

Preuve. Immédiat par induction sur la longueur de l. □

Une autre caractéristique importante de $\mathbb{E}_{\{x_i\}}(l)$ est qu'elle efface effectivement les def-dépendances de l vis-à-vis des x_i. Plus précisément, toute méthode x de l qui def-dépend d'un x_i voit sa définition effacée par $\mathbb{E}_{\{x_i\}}(l)$:

Lemme 16 (Effacement et def-dépendance) *Soient $\mathbb{W} = \phi_1, ..., \phi_n$ une liste de champs et $\{x_i\}_{i=1..n}$ un ensemble de noms de méthodes vérifiant les propriétés suivantes :*

1. $\forall i < j \leq n$ tel que $i \neq j$, $\mathcal{N}(\phi_i) \cap \mathcal{N}(\phi_j) = \emptyset$
2. $\{x_i\} \cap \mathcal{N}(\mathbb{W}) = \emptyset$
3. $\forall j \leq n, \llbracket \phi_j \rrbracket \subset \bigcup_{k<j} \mathcal{N}(\phi_k) \cup \{x_i\}$

Alors, $\forall x \in \mathcal{D}(l)$, tel que $\{x_i\} \cap \llbracket x \rrbracket \neq \emptyset$, $x \notin \mathcal{D}\left(\mathbb{E}_{\{x_i\}}(l)\right)$

Remarque 9. Les propriétés 1 et 3 de $\mathbb{W}$ sont vérifiées si on prend la liste des champs d'une espèce en forme normale.

Preuve. Par récurrence sur la position du champ définissant x dans $\mathbb{W}$. Si $\mathbb{W} = \phi; \mathbb{W}_1$, avec $x \in \mathcal{D}(\phi)$, alors le resultat est immédiat. Sinon, il vient $\mathbb{E}_{\{x_i\}}(\mathbb{W}) = \psi, \mathbb{E}_A(\mathbb{W})$, avec $\{x_i\} \subset A$, donc $A \cap \llbracket x \rrbracket \neq \emptyset$ et on peut conclure par récurrence. □

En fait, ce lemme peut être généralisé à la relation $<_{t-dep}$. Cette relation permet d'ailleurs de caractériser les méthodes qui seront effacées, comme le montrent les deux lemmes ci-dessous.

Lemme 17 *Avec les notations du lemme 16, s'il existe i tel que $x_i <_{t-dep} x$, alors $x \notin \mathcal{D}\left(\mathbb{E}_{\{x_i\}}(\mathbb{W})\right)$*

Preuve. D'après la définition de $<_{t-dep}$, $\exists y_1, ..., y_m$ vérifiant $y_j \in \llbracket y_{j+1} \rrbracket$, $y_1 = x_i$ et $y_m = x$. On procède par récurrence sur m.
Si $m = 2$, on applique directement le lemme 16.
Sinon, soit ϕ l'unique champ de $\mathbb{W}$ tel que $y_2 \in \mathcal{D}(\phi)$. On a $\mathbb{W} = \mathbb{W}_1; \phi; \mathbb{W}_2$. De plus, d'après nos hypothèses sur $\mathbb{W}$, $x \in \mathcal{N}(\mathbb{W}_2)$ (puisqu'il def-dépend

de y_2). Donc, $\mathbb{E}_{\{x_i\}}(\mathbb{W}) = \mathbb{W}'_1; \mathbb{E}_A(\phi; \mathbb{W}_2)$, avec $y_1 \in \{x_i\} \subset A$. ϕ est donc effacé, et on a :

$$\mathbb{E}_A(\phi; \mathbb{W}_{\not\in}) = \mathbb{E}(\phi); \mathbb{E}_{A \cup \mathcal{N}(\phi)}(\mathbb{W}_{\not\in})$$

Comme $y_2 \in \mathcal{N}(\phi)$ et $y_n = x \in \mathcal{N}(W_2)$, on peut appliquer l'hypothèse de récurrence sur la liste $y_2, ..., y_n$: x est effacé dans $\mathbb{E}_{A \cup \mathcal{N}(\phi)}(\mathbb{W}_2)$. Comme par ailleurs ce nom n'apparaît pas dans $\mathbb{W}_1$, x est bien effacé dans $\mathbb{E}_{\{x_i\}}(\mathbb{W})$ □

Lemme 18 *Avec les notations du lemme 16, si $x \in \mathcal{D}(\mathbb{W})$ et $\forall i, x_i \not<_{t-dep} x$, alors $x \in \mathcal{D}\left(\mathbb{E}_{\{x_i\}}(\mathbb{W})\right)$*

Preuve. Par induction sur la taille de $\mathbb{W}$: s'il n'y a aucun champ, l'énoncé devient trivial. Sinon, soit $x \in \mathcal{D}(\mathbb{W})$, vérifiant les hypothèses de l'énoncé. On pose $\mathbb{W} = \phi; \mathbb{W}_1$ et on distingue les cas suivants :

- $x \in \mathcal{D}(\phi)$. Comme par hypothèse x ne def-dépend pas des x_i, il vient $\mathbb{E}_{\{x_i\}}(\mathbb{W}) = \phi; \mathbb{E}_{\{x_i\}}(\mathbb{W}_1)$ par définition de l'effacement, et x reste défini après effacement.
- $\{x_i\} \cap \llbracket \mathcal{N}(\phi) \rrbracket = \emptyset$: $\mathbb{E}_{\{x_i\}}(\mathbb{W}) = \phi; \mathbb{E}_{\{x_i\}}(\mathbb{W}_1)$, et on conclut par récurrence sur $\mathbb{W}_1$.
- $\{x_i\} \cap \llbracket \mathcal{N}(\phi) \rrbracket \neq \emptyset$. Alors $\forall y \in \mathcal{N}(\phi), y \not<_{t-dep} x$ (sinon on aurait $x_i <_{t-dep} x$). De plus, on a $\mathbb{E}_{\{x_i\}}(\mathbb{W}) = \mathbb{E}(\phi); \mathbb{E}_{\{x_i\} \cup \mathcal{N}(\phi)}(\mathbb{W}_1)$, et on peut appliquer l'hypothèse de récurrence sur $\{x_i\} \cup \mathcal{N}(\phi)$ et $\mathbb{W}_1$.

□

3.9.7 Mise en forme normale

Il est toujours possible de retrouver une espèce $\tilde{s}$ en forme normale à partir d'une espèce s bien formée et bien typée telle que $\tilde{s}$ et s introduisent les même noms. Toutefois, on ne peut plus garantir que toutes les définitions de s sont conservées dans $\tilde{s}$ à cause des def-dépendances et des effacements de définitions qu'elles induisent. On cherche à minimiser le nombre de ces effacements, puisque chaque preuve effacée devra être refaite dans la suite du graphe d'héritage. Les deux principaux résultats de cette section montrent qu'il est possible de trouver une espèce en forme normale dont les définitions $\mathcal{D}(\tilde{s})$ sont un sous-ensemble des définitions de s, $\mathcal{D}(s)$ d'une part, et d'autre part que $\mathcal{D}(\tilde{s})$ est maximal au sens de l'inclusion ensembliste, c'est à dire qu'on n'a pas procédé à un effacement "inutile".

Pour pouvoir donner un énoncé précis du théorème, il reste à introduire une définition technique, $changed(y, x)$. Il s'agit d'un prédicat qui est vrai si et seulement si la définition de y a changé depuis la dernière définition de x (en suivant l'ordre de la clause **inherits**).

Définition 35 *$changed(y, x)$ est une relation sur $\mathcal{N}(s)$, s étant une espèce définie par defspec.*

$$\begin{array}{rcl} changed(y,x) & \iff & \left(\exists j > \mathbb{I}_s(x),\ y \in \mathcal{D}(s_j) \wedge \mathcal{B}_{s_j}(y) \neq \mathcal{B}_{s_{\mathbb{I}_s(x)}}(y)\right) \\ & \vee & (\exists k,\ y \in \mathcal{D}(\phi_k) \wedge \mathbb{I}_s(x) \neq n+1) \end{array}$$

Par ailleurs, $\circledcirc$ est étendue à la fusion des champs **property** et **theorem**. Dans le cas d'une espèce bien typée, ces derniers ne peuvent être fusionnés avec des champs calculatoires, ce qui rend l'extension relativement simple :

Définition 36 (Fusion des théorèmes et des propriétés)

$$
\begin{array}{lcll}
\textbf{\textit{property}}\ x\ = p & \circledcirc & \textbf{\textit{property}}\ x\ = p & = \textbf{\textit{property}}\ x\ = p \\
\textbf{\textit{property}}\ x\ = p & \circledcirc & \textbf{\textit{theorem}}\ x = p & = \textbf{\textit{theorem}}\ x = p \\
 & & \textbf{\textit{proof}}\ \textit{:prf} & \textbf{\textit{proof}}\ \textit{:prf} \\
\textbf{\textit{theorem}}\ x = p & \circledcirc & \textbf{\textit{property}}\ x\ = p & = \textbf{\textit{theorem}}\ x = p \\
\textbf{\textit{proof}}\ \textit{:prf} & & & \textbf{\textit{proof}}\ \textit{:prf} \\
\textbf{\textit{theorem}}\ x = p & \circledcirc & \textbf{\textit{theorem}}\ x = p & = \textbf{\textit{theorem}}\ x = p \\
\textbf{\textit{proof}}\ \textit{:prf}_1 & & \textbf{\textit{proof}}\ \textit{:prf}_2 & \textbf{\textit{proof}}\ \textit{:prf}_2
\end{array}
$$

Dans tous les autres cas, la fusion échoue.

Remarque 10. Les propriétés de préservation des noms (lemme 6) et de liaison retardée (lemme 7) sont trivialement conservées par cette extension.

Pour construire *nfs*, on reprend le même algorithme qu'à la section précédente, mais en cas de conflit entre deux champs, il faut prendre en compte les def-dépendances et faire les effacements correspondants. Afin que cela soit correct, il faut en outre raffiner la construction de la liste de champs $\mathbb{W}_1$, pour éviter d'effacer des définitions provenant du corps de s lui-même. En effet, si on considère l'exemple suivant :

```
species a inherits basic_object =
sig eq in self->self->bool;
let id(x in self) = x;
theorem foo = all x in self, self!eq(x,self!id(x))
proof : def : id;
 (* preuve utilisant la définition actuelle de id *);
end

species b inherits a =
rep = int;
proof of foo : def : id;
 (* preuve utilisant la nouvelle définition de id,
    donnée en dessous *);
let id(x in self)=int_plus(x,1);
end
```

Lorsqu'on construit la forme normale de b, si on analyse la redéfinition de `foo` *avant* celle de `id`, on va être conduit à effacer la nouvelle preuve, ce qui est manifestement faux. Au contraire, si on analyse `id` *avant* `foo`, on va d'abord effacer l'ancienne preuve, qui def-dépend de la définition de `id` dans `a`, avant de la remplacer quelques étapes plus loin par la nouvelle. La position des

champs dans le corps de l'espèce est donc important, et on va réordonner les ϕ_i pour éviter de se trouver dans le cas de l'exemple ci-dessus. Soient $i_1, \ldots, i_n$ une permutation de $1 \ldots n$ telle que $\forall j < k, \mathcal{N}(\phi_{i_j}) \cap \llbracket \phi_{i_k} \rrbracket = \emptyset$. Par définition de la bonne formation, une telle permutation existe. Sinon comme $\forall x \in \mathcal{N}(\phi_j), \llbracket x \rrbracket_s = \llbracket \phi_j \rrbracket$, il y aurait un cycle de dépendance dans s. On construit alors $\mathbb{W}_1$ à l'aide de cette permutation :

$$\mathbb{W}_1 = norm(s_1)@ \ldots @norm(s_{p_f})@[def_{i_1} \ldots def_{i_n}]$$

Une fois $\mathbb{W}_1$ formée de cette manière, on peut construire $\mathbb{W}_2$ de la même manière qu'auparavant, à condition de tenir compte des effacements en cas de conflit. Dans ce cas là, avec i_0 le plus petit index vérifiant $\mathcal{N}(\phi) \cap \mathcal{N}(\psi_{i_0}) \neq \emptyset$, on effectue les opérations suivantes :

$$\mathbb{W}_1 \leftarrow ((\phi \oslash \psi_{i_0}), \mathbb{X})$$

$$\mathbb{W}_2 \leftarrow [\psi_1; ...; \psi_{i_0 - 1}]@\mathbb{E}_{\mathcal{N}(\psi_{i_0})}(\psi_{i_0+1} \ldots \psi_m)$$

Comme la seule différence par rapport à la version précédente de l'algorithme réside dans l'effacement éventuel de définitions, la propriété d'unicité des noms (lemme 9) se retrouve immédiatement. De même, on a toujours $\mathcal{N}(nfs) = \mathcal{N}(s)$, avec le lemme 15. Il reste à prouver qu'on n'efface pas trop de preuves. Avant cela, il convient de montrer que $\mathbb{W}_2$ vérifie bien les conditions du lemme 16, afin de pouvoir appliquer les lemmes 16, 17 et 18.

Lemme 19 *Soit* $\mathbb{W}_2 = \psi_1, ..., \psi_n.\ \forall i \leq n,\ \llbracket \psi_i \rrbracket \subset \bigcup_{j<i} \mathcal{N}(\psi_j)$

Preuve. On montre cette propriété par récurrence sur le nombre d'étapes de l'algorithme. Au départ, $\mathbb{W}_2$ est vide, et la propriété est triviale. Sinon, en posant $\mathbb{W}_1 = \phi, \mathbb{X}$ on distingue les deux cas suivants :

- si $\mathcal{N}(\phi) \cap \mathcal{N}(\mathbb{W}_2) = \emptyset$, on ajoute ϕ à la fin de $\mathbb{W}_2$. Comme $\mathbb{W}_1$ est formée par les formes normales des espèces héritées, et qu'on a réordonné les champs du corps de s, les champs définissant les méthodes dont ϕ def-dépend sont présents dans $\mathbb{W}_2$. De plus, les noms introduits par ϕ sont de deux sortes :
  - des noms nouveaux n'ayant jamais été introduits par un champ de $\mathbb{W}_2$. Là encore comme les champs sont analysés suivant l'ordre des def-dépendances, aucun champ de $\mathbb{W}_2$ ne peut en def-dépendre.
  - des noms de champs ayant été fusionnés : lorsqu'on fusionne un champ ψ de $\mathbb{W}_2$, on efface toutes les définitions qui def-dépendent de $\mathcal{N}(\psi)$. Il n'y a donc pas non plus de def-dépendance vis-à-vis de ces noms dans $\mathbb{W}_2$.
- sinon, on enlève ψ_{i_0} de $\mathbb{W}_2$, et on efface les définitions des champs suivants de $\mathbb{W}_2$ qui def-dépendent de $\mathcal{N}(\psi_{i_0})$. D'après l'hypothèse de récurrence, on peut appliquer le lemme 17. Il n'y a donc plus de def-dépendance vis-à-vis de $\mathcal{N}(\psi_{i_0})$. Comme les autres champs ne sont pas modifiés, la propriété est préservée.

□

Théoreme 4 (préservation des définitions)
$\mathcal{D}(nfs) \subseteq \mathcal{D}(s)$
$\forall x \in \mathcal{D}(nfs), \mathcal{B}_s(x) = \mathcal{B}_{nfs}(x)$
$\forall x \in \mathcal{D}(s) \setminus \mathcal{D}(nfs), \exists y \in [\![x]\!]_s$ tel que $y \notin \mathcal{D}(nfs) \vee changed(y, x)$

Preuve. Le lemme 15 et le théorème 1 nous permettent de conclure immédiatement que $\mathcal{D}(nfs) \subseteq \mathcal{D}(s)$ (sans les effacements, on aurait l'égalité comme dans les sections précédentes)

Par ailleurs, comme la propriété 7 est toujours valable, on a :

$$\forall x \in \mathcal{D}(nfs), \mathcal{B}_s(x) = \mathcal{B}_{nfs}(x)$$

En effet, l'algorithme sélectionne toujours la dernière définition de x s'il n'a pas besoin de l'effacer. Il reste donc à montrer que les effacements se font bien dans les conditions décrites par l'énoncé.

Supposons qu'il existe $x \in \mathcal{D}(s) \setminus \mathcal{D}(nfs)$. Si $\exists \phi_i$ tel que $x \in \mathcal{D}(\phi_i)$, alors $\exists \phi_j$ tel que $y \in \mathcal{D}(\phi_j)$ tel que $x <_s^{def} y$ et ϕ_i est analysé avant ϕ_j. Sinon, x serait défini dans nfs. Comme on a réordonné la liste des ϕ_i lors de la construction de $\mathbb{W}_1$, ce cas ne peut pas se produire.

Donc, la définition de x a été héritée d'une espèce parent s_h. Supposons que $\forall y,$ tel que $y <_s^{def} x, \neg changed(y, x)$. Alors, d'après le lemme 18, x n'est pas affecté par les effacements ayant lieu après s_h. Comme de plus on utilise la forme normale des espèces parents, une redéfinition d'un tel y au niveau de s_h aurait lieu avant le traitement de x, et x serait donc bien défini dans nfs. Donc, $\exists y, y <_s^{def} x \wedge changed(y, x)$. □

De plus, nfs est bien formée. En effet, les lemmes 10, 2 et 12 s'étendent trivialement aux propriétés et aux théorèmes, la fusion de champs non calculatoires ne mettant en jeu qu'un nom de méthode à la fois. On peut donc conclure de la même manière que dans la section précédente :

Théoreme 5 (Forme normale d'une espèce) *Soit s une espèce bien formée et bien typée définie par defspec. Il existe une espèce $\tilde{s}$ en forme normale et vérifiant les propriétés suivantes :*

- *noms :* $\mathcal{N}(nfs) = \mathcal{N}(s)$ *et* $\mathcal{D}(nfs) \subseteq \mathcal{D}(s)$
- *définitions :* $\forall x \in \mathcal{D}(nfs), \mathcal{B}_s(x) = \mathcal{B}_{nfs}(x)$
- $\forall x \in \mathcal{D}(s) \setminus \mathcal{D}(nfs), \exists y \in [\![x]\!]_s$ tel que
  $(y \notin \mathcal{D}(nfs))$ *ou* $(y \in \mathcal{D}(nfs) \wedge changed(y, x))$.

Chapitre 4

Compilation vers OCaml : des classes et des objets

Le développement initial de Foc [BHMR98, BHR00, Bou00] a été fait en OCaml [LDG+02]. OCaml est un langage fonctionnel, développé à l'INRIA, qui dispose d'un certain nombre de constructions facilitant la description des espèces et des collections de Foc. Par ailleurs, le langage de base de Foc reprend les traits fonctionnels de OCaml. Outre cette parenté, les principaux avantages du choix d'OCaml comme langage-cible de la partie calculatoire de Foc sont les suivants :

- la sémantique de OCaml repose sur des bases solides et bien étudiées, y compris en ce qui concerne les traits objets (voir ci-dessous).
- les traits orienté-objet et le système de modules permettent d'exprimer les principales constructions de Foc tout en garantissant une certaine validation du code généré par `focc` grâce aux analyses du compilateur OCaml (voir le chapitre 5).
- OCaml dispose d'un *garbage collector* très efficace, permettant de gérer la mémoire de façon automatique, ce qui est quasiment indispensable dès qu'on fait tourner les algorithmes de calcul formel sur des données de taille conséquente.
- enfin, le compilateur OCaml est capable de générer du code machine très performant pour une grande variété d'architectures matérielles.

4.1 Description d'OCaml

Le but de cette section n'est pas de faire une description complète du langage OCaml mais d'introduire les deux constructions les plus importantes du point de vue de Foc, qui sont les **classes** et les **modules**. En particulier, on ne décrira pas le langage de base des expressions de OCaml (à des différences syntaxiques près, les expressions de Foc sont incluses dans ce langage).

4.1.1 Modules et abstraction

Un module est un moyen de rassembler différentes définitions de fonctions et/ou de types au sein d'un même bloc de programmation. Il est par ailleurs possible d'abstraire une composante de type d'un module ou de cacher certaines fonctions, qui ne seront alors plus visibles de l'extérieur. Par exemple, on peut définir un module qui effectuera les opérations de base sur des entiers modulo 7, tout en masquant aux fonctions extérieures au module la représentation concrète des entités manipulées :

```
module Entiers_mod_7 :
sig
  type t
  val zero : t
  val un : t
  val opp : t → t
  val plus : t → t → t
  val mult : t → t → t
  val of_t : t → int
end
=
struct
  type t = int
  let zero = 0
  let un = 1
  let opp x = if x = 0 then 0 else 7 − x
  let plus x y = (x + y) mod 7
  let mult x y = (x * y) mod 7
  let of_t x = x
end
```

La première partie de la définition, introduite par **sig** est la *signature* du module : ce sont les seules informations qui seront visibles à l'extérieur du module. Ici, `Entiers_mod_7` définit un type abstrait `t`, deux constantes de ce type, des opérations sur `t`, et une fonction permettant de retrouver un entier à partir d'un élément de type `Entiers_mod_7.t`.

À partir du mot-clé **struct**, on passe à l'*implantation* du module proprement dite : on définit les fonctions exigées par la signature. OCaml va alors vérifier que la signature et l'implantation sont cohérentes, c'est à dire :

- les champs mentionnés dans la signature doivent être présents dans l'implantation (il peut y avoir plus de champs dans l'implantation, mais ceux qui n'apparaissent pas dans l'interface seront inaccessibles de l'extérieur).
- les types inférés pour les définitions de l'implantation doivent être compatibles avec ceux qui sont déclarés dans la signature. Comme pour

les règles SELF de FOC (27), on peut utiliser les définitions de types de l'implantation pour unifier le type inféré et le type déclaré. Par exemple, `of_t` a comme type $\alpha \to \alpha$, qui peut être instantié en `t` $\to$ `int` puisque à l'intérieur de l'implantation `t` = `int`.

Pour en terminer avec cet exemple, on peut revenir un instant sur cette fonction `of_t`. Elle n'a en effet pas un intérêt calculatoire évident, mais est en revanche indispensable d'un point de vue pratique : sans elle, il serait toujours possible de faire toutes les opérations qu'on veut sur des entiers modulo 7, mais il serait impossible d'en voir le résultat. Par exemple, si on essaie d'évaluer zero :

```
# let x = Entiers_mod_7.zero;;
val x : Entiers_mod_7.t = <abstr>
```

on obtient bien un élément du type attendu (`Entiers_mod_7.`t), mais la valeur exact de cet élément n'est pas directement accessible, ce que OCAML signale par `<abstr>`. `of_t` permet de trouver le résultat attendu

```
 # let y = Entiers_mod_7.of_t x;;
val y : int = 0
```

Plus généralement, l'utilisation de types abstraits suppose qu'on se donne des fonctions permettant d'accéder à une représentation des objets de ce type et, à l'inverse, de créer des objets de ce type. On retrouve cette approche en FOC avec les fonction `print` et `parse` de l'espèce de base `basic_object` de la librairie, que l'utilisateur peut redéfinir lorsqu'il décrit une espèce dont le type support est connu.

4.1.2 La composante objet d'OCAML

Parallèlement aux modules, OCAML dispose de traits objets et permet de définir une hiérarchie de **classes**, ainsi que de créer des instances des classes dont toutes les méthodes sont définies. Une classe se compose essentiellement de deux choses. D'une part, on trouve les variables d'instance, qui ne peuvent faire référence à la classe proprement dite et ne sont pas accessible en dehors de cette dernière. D'autre part, on dispose de méthodes, qui peuvent être virtuelles (abstraites) ou posséder une définition. Par ailleurs, les classes peuvent posséder des paramètres de types auxquels les méthodes font référence. Ainsi, on peut définir une classe représentant les fonctions de OCAML :

```
class ['a,'b] func (f :'a → 'b)=
object(self)
  val my_f = f
  method app x =
    print_endline "computing f...";
    my_f x;
end
```

Cette classe a deux paramètres de type, `'a` et `'b`. De plus, elle est abstraite par rapport à une fonction `f` de type `'a → 'b`. Cette abstraction sert à donner la valeur de la variable d'instance `my_f`. Enfin, on définit une méthode `app`, qui consiste à appeler `my_f` (donc `f`), après avoir affiché un message.

À partir de `func`, on peut définir une classe correspondant au cas où `'a` et `'b` sont égaux :

```
class ['c] endo (f1 :'c → 'c) =
object(self)
  inherit ['c,'c] func f1
  method iter n =
    if n <= 0 then fun x → x
    else fun x → self#app (self#iter (n−1) x)
end
```

Cette classe n'a plus qu'un paramètre de type. Elle hérite de `func`, dont on instantie les paramètres de type, et qu'on applique à `f1`. De plus, on peut définir la méthode `iter`, qui consiste à appliquer `n` fois la méthode `app` de l'objet courant, représenté par `self`[1].

Nous définissons maintenant une class *virtuelle* : cette classe contient une méthode déclarée (dont on connaît seulement le type), et on ne peut donc définir d'objet appartenant à cette classe : il faut au cours de l'héritage définir la méthode virtuelle.

```
class virtual ['a] ens_non_vide =
object
  method virtual element : 'a
end
```

À l'aide de cette classe, on peut définir la classe des fonctions dont le codomaine est non vide.

```
class ['a,'b,'cl] func_non_vide (f : 'a → 'b)(e :'cl) =
object(self)
  inherit ['a,'b]func f
  constraint 'cl= 'a #ens_non_vide
  method image_element = self#app(e#element)
end
```

Le deuxième argument de la classe, `e`, a le type `'cl`, qui porte une *contrainte* : ce doit être au moins une implantation de `ens_non_vide`, c'est à dire un objet comportant la méthode `element`. Cette contrainte est nécessaire pour que le corps de la méthode `image_element` soit bien typé.

[1] Contrairement à FOC, `self` n'est pas un mot clé, mais une variable. Son nom est choisi dans la clause **object** de la classe.

Enfin, on peut donner un exemple d'héritage multiple, avec la classe des fonctions opérant sur un ensemble non vide.

```
class['a,'cl1]endo_non_vide (f :'a → 'a) (e :'cl1)=
object(self)
  inherit ['a]endo f
  inherit ['a,'a,'cl]func_non_vide f e
  constraint 'cl1 = 'a #ens_non_vide
  method nieme_image n = self#iter n e#element
  method image_element = self#nieme_image 1
end
```

Cette classe hérite à la fois de `endo` et `func_non_vide`. Dans ce dernier cas, il faut instantier le troisième paramètre de type par un type répondant à la contrainte. C'est le cas ici, puisque `'cl1` lui-même porte la même contrainte.

Notons enfin, qu'à chaque classe correspond un **type de classe**, inférer par OCAML, qui reprend la liste des méthodes de la classe avec leur type (mais pas leur éventuelle définition). Il est aussi possible de définir des types de classe indépendamment d'une classe. Ainsi, on aurait pu représenter les ensembles non vide par un type de classe, de la manière suivante :

```
class type ['a] ens_non_vide =
object
  method element : 'a
end
```

Un type de classe peut hériter d'autres type de classes, et déclarer des méthodes virtuelles.

4.2 Description informelle de la traduction de FOC en OCAML

Comme on a sans doute pu le constater avec la section précédente, les classes de OCAML permettent une représentation relativement simple, du moins au premier abord, des espèces de FOC. Chaque espèce sera représentée par une classe, l'héritage entre espèces est représenté directement par l'héritage entre classes et les paramètres des espèces deviennent des paramètres de classe.

Par ailleurs, le type support devient un paramètre de type, qu'il soit abstrait ou concret. Dans ce dernier cas, une contrainte de type le liera à sa représentation concrète, mais le fait de le présenter explicitement comme un paramètre en permettra l'abstraction lors de la création des collections. De même, les paramètres de collection engendrent chacun deux paramètres de type : un pour leur type support (toujours abstrait par définition) et un pour

la classe à laquelle ils appartiennent. Une contrainte de type supplémentaire lie les deux et signale à OCAML l'interface minimale que le paramètre doit supporter.

Enfin, les collections sont représentées par des objets, c'est à dire par des instances de la classe représentant l'espèce dont elles sont issues. Afin d'obtenir l'abstraction du type support, l'objet en question est encapsulé dans un module, qui comporte essentiellement deux champs, un type abstrait t et un objet de classe $[t]$ `espece_implantee`. L'idée que ce sont les collections qui sont des objets et non pas les entités qui forment ces collections est un des éléments majeurs de la représentation des structures FOC en OCAML (voir en particulier [BHR00] et [Bou00]). De ce fait, les objets que l'on obtient se présentent comme des listes de fonctions qu'on peut appliquer aux éléments du type support. Cette vision permet de s'affranchir des problèmes liés aux méthodes binaires dans les approches orientées objet classiques, et rapproche FOC des mixins (constructions mêlant modules et héritage).

4.2.1 Les espèces vues comme des classes

Nous allons maintenant examiner plus en détail la traduction d'une espèce s de FOC bien formée et bien typée en une classe de OCAML. Pour cela, nous considérerons comme dans le chapitre précédent la définition suivante :

species s **inherits** $s_1,\ldots,s_n$ $=$ ϕ_1 $\ldots$ ϕ_m **end**

Chacune des espèces parents s_i est reflétée par une déclaration **inherit** en OCAML comme en FOC. De cette manière, comme FOC et OCAML partagent la même convention sur l'héritage multiple (on sélectionne toujours la dernière méthode dans l'ordre de l'héritage), on peut se reposer entièrement sur les mécanismes de OCAML pour la résolution de l'héritage. Ainsi, seules les méthodes des champs ϕ_i doivent être traduites dans le corps de la classe s. Une signature sera représentée par une méthode virtuelle, et une définition par une méthode "normale". Comme les méthodes OCAML peuvent s'appeler librement les unes les autres, il n'y a pas lieu ici de faire la distinction entre les méthodes récursives et les autres. De plus, les propriétés et les théorèmes ne sont pas conservés : ces champs ne sont présents que dans la traduction vers COQ que nous verrons dans le chapitre suivant.

Par ailleurs, la traduction des expressions de FOC se fait directement en OCAML, à quelques variantes syntaxiques près (par exemple, l'appel de méthode est noté par un `#` et non un `!`). En revanche, les types ne peuvent être repris tels quels. Le type support de l'espèce est une variable `'t`, de même que le type support de chaque paramètre de collection. Les paramètres eux-mêmes sont donnés comme paramètres de la classe, comme nous le verrons plus loin (4.3.2).

4.2.2 Un exemple de traduction

Afin d'examiner plus en détail la traduction, on peut reprendre l'exemple de la hiérarchie donnée en introduction (section 2.2) et voir comment elle est représentée en OCAML.

Tout d'abord, on définit l'interface correspondant à `setoide`, c'est à dire pour OCAML un type de classe :

```
class type virtual [ 't]setoidetype=
object('self)
  inherit [ 't ]basic_objecttype
  method different : ('t)→ ('t)→ bool
  method virtual element : 't
  method virtual egal : ('t)→ ('t)→ bool
end ;;
```

Comme on l'a déjà dit, le type support de `setoide` devient une variable de type `'t`, tandis qu'on trouve dans le corps de la classe la clause **inherit** qui permet d'importer les méthodes de `basic_object`. `setoide` introduit trois méthodes supplémentaires. Deux d'entre elles, `element` et `egal`, qui ne sont que déclarées dans l'espèce. En OCAML, elles sont marquées comme virtuelles. De ce fait, comme `setoidtype` contient des méthodes virtuelles, la classe entière est marquée **virtual**.

Après la définition de l'interface, on trouve la traduction de l'espèce proprement dite, c'est à dire une classe à part entière.

```
class  virtual [ 't ]setoide =
    object(sobj :'self)
    inherit [ 't ]basic_object
    method virtual egal : ('t)→ ('t)→ bool
    method virtual element : 't
    method different : ('t)→ ('t)→ bool=
       fun (x : 't)→ fun (y : 't)→
          not_b (sobj#egal x y)
end ;;
```

Cette fois-ci, on donne la définition de la méthode `different`, telle qu'elle a été fournie en FOC. Bien que cela ne soit pas obligatoire pour la bonne définition de la classe (OCAML pourrait très bien l'inférer), on donne également explicitement le type que doit avoir `different`, afin de garantir la correspondance entre `setoide` et `setoidetype`.

Une fois la traduction de l'espèce `setoide` effectuée, on trouve celle de `monoide`, qui reprend exactement le même principe (on ne montre que la

classe correspondante, le type de classe étant trivial à inférer).

```
class virtual [ 't ]monoide=
  object(sobj :'self)
  inherit [ 't ]setoide
  method virtual multiplie : ('t)→ ('t)→ 't
  method virtual un : 't
  method element : 't= sobj#multiplie sobj#un sobj#un
end;;
```

L'introduction du produit cartésien de deux setoides permet de voir le traitement réservé aux espèces paramétrées. Commençons par l'interface :

```
class type [ 't, 'a, 'spec_a , 'b, 'spec_b ]setoide_produittype=
object('self)
  constraint 'spec_a= ( 'a)#setoidetype
  constraint 'spec_b= ( 'b)#setoidetype

  inherit [ 't ]setoidetype

  method print : ('t)→ string
  method element : 't
  method creer : ('a)→ ('b)→ 't
  method egal : ('t)→ ('t)→ bool
end;;
```

Comme toutes les méthodes de `setoide_produit` sont définies, le type de classe (et la classe correspondante comme nous le verrons juste au-dessous) n'est plus virtuelle : on peut créer des instances de cette classe, c'est à dire des collections. Par ailleurs, on peut noter la présence de 4 nouvelles variables de type, soit 2 par paramètre de collection dans l'espèce. De plus, deux clauses **constraint** spécifient que `'spec_a` doit être au moins un *setoide* (c'est à dire une classe présentant les méthodes de `setoidetype` et éventuellement d'autres) de type support `'a`. Au niveau du type de la classe, ce sont les deux seuls points qui distinguent une espèce paramétrée d'une espèce atomique. La classe correspondante est par contre un peu différente de celle d'une espèce paramétrée

```
class  [ 't, 'a, 'spec_a , 'b, 'spec_b ]setoide_produit
  ((__p_a, __p_b) :('spec_a)*('spec_b))=

  object(sobj :'self)
  constraint 'spec_a= ( 'a)#setoidetype
  constraint 'spec_b= ( 'b)#setoidetype
```

```
  constraint 't = ('a)*('b)

  inherit [ 't ]setoide

  method egal : ('t)→ ('t)→ bool=
    fun (x : 't)→ fun (y : 't)→
     and_b
      (__p_a#egal (first x) (first y))
      (__p_b#egal (scnd x) (scnd y))

  method creer : ('a)→ ('b)→ 't=
    fun (x : 'a)→ fun (y : 'b)→ crp x y
  method element : 't=
    sobj#creer __p_a#element __p_b#element
  method print : ('t)→ string= fun (x : 't)→
     sc "(" (sc (__p_a#print (first x))
    (sc "," (sc (__p_b#print (scnd x)) ")")))
end;;
```

En premier lieu, on remarque la présence de deux paramètres pour la classes, de type `'spec_a` et `'spec_b`. Comme on utilise des méthodes de ces paramètres (dans le corps de `egal` par exemple), les contraintes de types sont importantes : il faut indiquer quelles sont les méthodes qu'on s'autorise à appeler sur `__p_a` et `__p_b`. Là encore, on pourrait laisser OCAML inférer ces contraintes, mais les mettre explicitement offre une garantie supplémentaire sur la correction de l'analyse d'un programme FOC par `focc`.

Par ailleurs, une troisième contrainte de type intervient. Elle est liée au fait que le type support est défini dans l'espèce `setoide_produit`. `'t` ne peut donc pas avoir une forme quelconque, mais doit être un produit. Cette contrainte aussi pourrait être inférée par OCAML. En effet, la définition de `egal` impose que ses arguments, de type `'self` d'après la déclaration héritée de `setoide`, aient le type `'a * 'b` : l'emploi de `first` et `scnd` implique un produit, et l'appel à `__p_a#egal` et `__p_b#egal` impose que la première composante (respectivement la deuxième) soit le type support de `__p_a` (respectivement `__p_b`). Comme on le verra plus loin, le fait que cette contrainte n'apparaisse pas dans le type de classe correspondant reflète l'idée qu'il n'y a que des decl-dépendances au niveau des types de méthodes. De plus, c'est cette absence de contrainte qui autorise l'abstraction de type lors de la création des collections (cf 4.3.7).

Passons maintenant à la définition du produit de deux monoïdes. Deux points sont importants ici : l'héritage multiple et l'héritage entre espèces paramétrées. Là encore, on ne montre que la classe, les différences avec le

type de classe correspondant étant les mêmes que pour le produit de setoïde.

```
class  [ 't, 'a, 'spec_a , 'b, 'spec_b ]
 monoide_produit ((__p_a, __p_b) :('spec_a*'spec_b))=
   object(sobj :'self)
   constraint 'spec_a= ( 'a)#monoidetype
   constraint 'spec_b= ( 'b)#monoidetype

   constraint 't = 'a*'b
   inherit [ 't ]monoide
   inherit [ 't ,  'a, 'spec_a, 'b, 'spec_b]
               setoide_produit(__p_a , __p_b)

   method un : 't= sobj#creer __p_a#un __p_b#un
   method multiplie : ('t)→ ('t)→ 't=
     fun (x : 't)→ fun (y : 't)→
       crp (__p_a#multiplie (first x) (first y))
           (__p_b#multiplie (scnd x) (scnd y))
end;;
```

L'héritage multiple se traduit par deux clauses **inherit** dans le corps de l'objet, avec bien sûr le même type support (`'t`) en paramètre dans les deux cas. De plus, comme `setoide_produit` impose des contraintes de types à `'a` et `'spec_a`, il faut que celles-ci soient respectées par les mêmes paramètres de `monoide_produit`. C'est le cas puisque tout *monoide* est aussi un *setoide* : la clause `'spec_a= ( 'a)#monoidetype` implique que l'instantiation des paramètres dans l'héritage est correct.

Maintenant que nous avons toutes nos espèces, nous pouvons nous intéresser à la création des collections. Dans le premier cas, `entiers` doit encore définir des méthodes par rapport à l'espèce `monoide` : on crée une classe auxiliaire qui contient ces définitions (et le type de classe correspondant, non représenté ici).

```
class  [ 't ]entiers =
   object(sobj :'self)

   constraint 't = int
   inherit [ 't ]monoide

   method un : 't= (1)
   method multiplie : ('t)→ ('t)→ 't= int_mult
   method egal : ('t)→ ('t)→ bool= base_eq
   method print : ('t)→ string= string_of_int
   method parse : (string)→ 't= int_of_string
end;;
```

Ensuite, on passe à la création de la collection proprement dite. Ainsi qu'on l'a dit, elle est encapsulée dans un module

```
module Entiers :
  sig
    type rep
    and stype= rep monoidetype
    val obj : stype
  end
  =
  struct
    type rep = int
    and stype= rep monoidetype
    let obj = (new entiers :>stype)
  end
```

Ce module a trois composantes :

1. le type `rep`, type support de la collection, qui est abstrait.

2. le type `stype` qui est le type de l'objet représentant la collection, soit en termes Foc l'interface que cette collection implante. On peut remarquer qu'il s'agit bien d'un `monoide` et pas d'un `entiers`.

3. l'objet lui même, converti en un objet de type `stype`, afin de réaliser effectivement l'abstraction de type. On peut dès lors appeler toutes les méthodes de `monoidetype` sur `obj`.

Il convient de remarquer que l'abstraction de `rep` n'est possible que parce que la variable de type `'t` est présente dans toutes les classes, y compris quand le type support est parfaitement défini (dans `entiers`). Si dans cette dernière classe on utilisait directement `int` à la place de `'t`, il serait impossible de réaliser l'abstraction par rapport à `'t`. De plus, les contraintes liant le type support à sa représentation concrète ne doivent effectivement pas apparaître dans les types de classes. En effet dans la signature du module, `rep` est abstrait. Il est impossible de savoir s'il vérifie ces contraintes ou non.

Par ailleurs, le type `stype` n'est pas abstrait : c'est ce type qui nous renseigne sur les méthodes de `Entiers.obj`, c'est à dire qui nous permet d'appeler sur l'objet en question toutes les méthodes de `monoidetype`.

Enfin, le module `Entiers2`, qui représente l'implantation d'une espèce

paramétrée, est défini de la manière suivante :

```
module type Entiers_2 :
  sig
    type a_t =Entiers.rep
    and aspec = Entiers.stype
    and b_t =Entiers.rep
    and bspec = Entiers.stype

    and rep
    and stype= (rep , a_t, aspec , b_t, bspec )
      monoide_produittype
    val obj : stype
  end
  =

  struct
    type a_t =Entiers.rep
    and aspec = Entiers.stype
    and b_t =Entiers.rep
    and bspec = Entiers.stype

    and rep = (a_t*b_t)
    and stype= (rep , a_t, aspec , b_t, bspec )
      monoide_produittype
    let obj =
      (new monoide_produit(Entiers.obj,Entiers.obj) :>stype)
  end
```

Si les trois derniers champs sont à peu près identiques à ceux de `Entiers`, à l'exception du fait qu'il faut bien sûr fournir plus de paramètres de types dans la définition de `stype`, on peut noter la présence de plusieurs autres champs de type, correspondant à l'instantiation des deux paramètres a et b de `monoide_produit`. Plus précisément, on définit deux champs de type par paramètre : un pour le type support et un pour le type de classe représentant le paramètre. Ici, le type support `a_t` doit rester concret si on veut que `Entiers_2` puisse communiquer avec `Entiers` : si ce type était abstrait, il serait impossible d'utiliser `creer`, qui prendrait en argument des éléments d'un type abstrait, pour lequel on ne dispose d'aucun constructeur. Il s'agit donc de partager des contraintes de types, comme avec la construction OCAML

```
with type a_t = Entiers.rep
```

présente en particulier lors qu'on utilise des foncteurs, afin de partager des informations de types entre l'argument du foncteur et le module résultat.

4.3 Formalisation de la traduction

Nous allons maintenant présenter de manière plus systématique la traduction d'une espèce Foc en une classe Ocaml, et d'une collection en un module. Pour cela, on va suivre les différentes composantes d'une classe Ocaml, dans l'ordre où elles apparaissent syntaxiquement :

1. les paramètres de types et les paramètres de la classe
2. les contraintes de type
3. l'héritage
4. les méthodes
5. les modules et collections

Dans le chapitre 5, on verra qu'une espèce bien typée est traduite en une classe typable par Ocaml, ce qui offre une certaine garantie sur le compilateur Foc lui même.

Notation 9 *Nous considérerons dans la suite une espèce de la forme suivante :*

species s $(c_1$ **.** $i_1,\ \ldots\ ,\ c_{p_f}$ **.** $i_{p_f})$
  inherits $s_1,\ldots,s_{h_f}$ $=$
    ϕ_1 ;
    ϕ_m ;
end

avec $\mathbf{.} = \{\boldsymbol{in}, \boldsymbol{is}\}$.
Cette définition sera notée deffoc. Par ailleurs, la traduction des constructions Foc *sera notée* Foc $\twoheadrightarrow$ Ocaml.

4.3.1 Les expressions de base

Comme on l'a déjà dit, les expressions de base de Foc ont une traduction immédiate en Ocaml. Cette traduction est donnée dans la figure 4.1.

4.3.2 Les paramètres des espèces Foc

Paramètres de type des classes Ocaml

En premier lieu, il faut donner la liste $\mathfrak{T}(s)$ des variables de type qui sont utilisées dans le corps de la classe. De plus, on garde une trace des variables associées aux différents paramètres. Pour cela, on définit une fonction $\mathfrak{T}$ qui à tout nom de paramètre a associe une paire de variables de type (ρ, σ), ou ρ est la variable correspondant au type support de a et σ celle correspondant à l'interface de a. Ce rôle sera précisé lorsqu'on s'intéressera aux contraintes de type au sein d'une classe (section 4.3.3).

$$x \twoheadrightarrow x \qquad c!x \twoheadrightarrow c\#x \qquad \frac{e \twoheadrightarrow e'}{\textbf{fun } x \to e \twoheadrightarrow \textbf{fun } x \to e'}$$

$$\frac{e_1 \twoheadrightarrow e_1' \qquad e_2 \twoheadrightarrow e_2'}{\textbf{let } x = e_1 \textbf{ in } e_2 \twoheadrightarrow \textbf{let } x = e_1' \textbf{ in } e_2'}$$

$$\frac{e_1 \twoheadrightarrow e_1' \qquad e_2 \twoheadrightarrow e_2'}{\textbf{let rec } x = e_1 \textbf{ in } e_2 \twoheadrightarrow \textbf{let rec } x = e_1' \textbf{ in } e_2'} \qquad \frac{f \twoheadrightarrow f' \qquad \forall i, e_i \twoheadrightarrow e_i'}{f(e_1, ..., e_n) \twoheadrightarrow (f' e_1' ... e_n')}$$

FIG. 4.1 – Traduction des expressions de base

Définition 37 (Variables de type d'une classe) *Soit une espèce s définie comme ci-dessus, de paramètres $c_1, ..., c_{p_f}$ On définit $\mathfrak{T}(s)$ à partir de la fonction auxiliaire $\mathbb{T}l$ qui donne la liste des variables de type liées aux paramètres :*

$$\frac{\rho, \sigma \vdash \mathbb{T}(c_1 \,.\, i_1, ..., c_{p_f} \,.\, i_{p_f}) = \mathfrak{T}}{\mathfrak{T}(s) = \mathfrak{T} \cup \{self \mapsto (\rho, \sigma)\}} \qquad \mathcal{L} \vdash \mathbb{T}(\emptyset) = \emptyset \qquad \frac{\mathcal{L} \vdash \mathbb{T}(l) = \mathfrak{T}}{\mathcal{L} \vdash \mathbb{T}(c \ \textbf{\textit{in}} \ \tau, l) = \mathfrak{T}}$$

$$\frac{\rho \notin \mathcal{L} \qquad \sigma \notin \mathcal{L} \qquad \mathcal{L}, \rho, \sigma \vdash \mathbb{T}(l) = \mathfrak{T}}{\mathcal{L} \vdash \mathbb{T}(c \ \textbf{\textit{is}} \ i, l) = \mathfrak{T} \cup \{c \mapsto (\rho, \sigma)\}}$$

Pour tout $c \in \mathfrak{T}(s)$, on définit

$$\rho(c) = \text{fst}(\mathfrak{T}(s)(c)) \text{ et } \sigma(c) = \text{snd}(\mathfrak{T}(s)(c))$$

Remarque 1. Avec cette définition, on dispose d'une variable ρ(**self**) représentant le type support de l'espèce courante, et d'une autre, σ(**self**), identifiant l'interface de l'espèce qu'on est en train de traduire. Dans la version de base de FOC cette seconde variable n'est pas utilisée, mais certaines extensions sont amenées à travailler sur **self** en tant que collection à part entière (section 8.4).

Traduction des types

Une fois qu'on dispose de l'environnement $\mathfrak{T}$ on peut proposer une traduction des types des méthodes FOC en OCAML. Pour cela, outre $\mathfrak{T}$ on prend en compte $\mathcal{C}$, la liste des collections existantes au moment où on type l'espèce FOC.

Définition 38 (Traduction des types Foc)

$$\frac{\mathfrak{T},\mathcal{C} \vdash \tau_1 \twoheadrightarrow t_1 \qquad \mathfrak{T},\mathcal{C} \vdash \tau_2 \twoheadrightarrow t_2}{\mathfrak{T},\mathcal{C} \vdash \tau_1 \to \tau_2 \twoheadrightarrow t_1 \to t_2} \qquad \frac{\alpha' \text{ libre dans } \mathfrak{T}}{\mathfrak{T},\mathcal{C} \vdash \alpha \twoheadrightarrow \alpha'} \qquad \frac{c \in \mathfrak{T}}{\mathfrak{T},\mathcal{C} \vdash c \twoheadrightarrow \rho(c)}$$

$$\frac{c \notin \mathfrak{T} \qquad c \in \mathcal{C}}{\mathfrak{T},\mathcal{C} \vdash c \twoheadrightarrow Mod(c).rep}$$

où $Mod(c)$ est le nom du module correspondant à c dans la traduction Ocaml *tel qu'il est défini ci-dessous (def.50)*

Les variables de type Foc doivent être transformées en des variables de type fraîches en Ocaml, c'est à dire n'apparaissant pas dans les paramètres de type de la classe qu'on est en train de construire. Par ailleurs, le type support d'une collection Foc c peut être traduit de deux manières différentes :

- Si c est un paramètre de l'espèce courante, on utilise la variable $\rho(c)$ correspondante.
- Si c est une collection existante, on utilise le champ du module correspondant.

Arguments d'une classe

Outre des variables de types, la classe représentant une espèce paramétrée s prend en argument des valeurs dont le type peut inclure certaines des variables de $\mathfrak{T}$:

Définition 39 (Traduction des paramètres des espèces)

$$\mathfrak{T},\mathcal{C} \vdash \emptyset \twoheadrightarrow \emptyset \qquad \frac{\mathfrak{T},\mathcal{C} \vdash l \twoheadrightarrow l' \qquad \mathfrak{T},\mathcal{C} \vdash \tau \twoheadrightarrow t}{\mathfrak{T},\mathcal{C} \vdash x \ \textbf{\textit{in}}\ \tau, l \twoheadrightarrow x : t, l'}$$

$$\frac{\mathfrak{T},\mathcal{C} \vdash l \twoheadrightarrow l' \qquad c \in \mathfrak{T}}{\mathfrak{T},\mathcal{C} \vdash c \ \textbf{\textit{is}}\ i, l \twoheadrightarrow c : \sigma(c), l'}$$

On peut d'ores et déjà s'assurer que pour une espèce bien typée, la traduction n'échoue jamais :

Lemme 20 (Traduction des paramètres d'entités) *Soit s une espèce définie par deffoc, telle que le paramètre j ($j \leq k$) soit un paramètre d'entité. On note $c_j = x$ et $i_j = \tau$ le nom de ce paramètre et son type. Alors si s est bien typée dans l'environnement $\mathcal{C}, \mathcal{E}, \Gamma, \Sigma$, il existe un type* Ocaml *t tel que*

$$\mathfrak{T}(s),\mathcal{C} \vdash \tau \twoheadrightarrow t$$

Preuve. D'après la définition 23, τ est bien formé dans le contexte $\mathcal{C}'$ composé de $\mathcal{C}$ et des paramètres de collection c_i, avec $i < j$. Donc tous les noms de collection c apparaissant dans τ sont soit des paramètres de collection de s, soit dans $\mathcal{C}$. Dans les deux cas, il existe t tel que $c \twoheadrightarrow t$. □

Lemme 21 (bonne définition de la traduction) *Soit s une espèce définie par deffoc ayant au moins un paramètre ($k \geq 1$) et bien typée dans l'environnement $\mathcal{C}, \mathcal{E}, \Sigma, \Gamma$. Il existe une expression* OCAML *e telle que $\mathfrak{T}(s), \mathcal{C} \vdash c_j \,.\, i_j \twoheadrightarrow e$*

Preuve. Immédiat par induction sur k. On a vu dans le lemme précédent que la traduction de chaque type de paramètre d'entité donnait un type OCAML. Par définition de $\mathfrak{T}$(s), $\mathfrak{T}$(s)(c) existe pour tous les paramètres de collection : dans les deux cas, la traduction donne un résultat. □

4.3.3 Contraintes de type

Après avoir défini les variables de types de la traduction, il faut retrouver les contraintes qui les lient entre elles. Ces contraintes sont de deux sortes : celle qui concerne le type support quand ce dernier est défini dans s et celles qui concernent les paramètres de collection. Dans le premier cas, il suffit de traduire la représentation du type support en OCAML. Dans le second, il faut distinguer deux types d'interfaces : des interfaces atomiques, ou des interfaces instanciées par des paramètres.

Définition 40 (Contraintes sur le type support) *Soit s une espèce définie par deffoc, on définit la contrainte de type sur le type support **rep** de s de la manière suivante :*

$$\frac{\mathcal{B}_s(\textbf{\textit{rep}}) = \bot}{\mathbb{C}(\textbf{\textit{rep}}) = \emptyset} \qquad \frac{\mathcal{B}_s(\textbf{\textit{rep}}) = \tau \qquad \mathfrak{T}(s), \mathcal{C} \vdash \tau \twoheadrightarrow t}{\mathbb{C}(\textbf{\textit{rep}}) = \textbf{\textit{constraint}}\ \rho(\textbf{\textit{self}}) = t}$$

Lemme 22 (Bonne définition de la contrainte du type support) *Soit s définie par deffoc, si s est bien typée, $\mathbb{C}(\textbf{\textit{rep}})$ existe.*

Preuve. Il y a trois cas à examiner :

- **rep** est déclaré dans s : c'est le premier cas de la définition.
- **rep** est défini au niveau de s (et non pas hérité) : on conclut comme dans le lemme 20 que la traduction de τ est correcte
- **rep** est hérité d'une espèce parent s_i : la remarque 6 permet de conclure comme dans le cas précédent. Si s_i est paramétrée, on effectue les substitutions de paramètres, de sorte que $\mathcal{B}_s$(**rep**) est bien formé dans l'environnement de s.

□

Avant d'aborder la traduction des contraintes de type liant $\rho(c)$ et $\sigma(c)$ pour un paramètre de collection c, il faut se donner une définition annexe permettant de gérer le cas où l'interface I du paramètre c n'est pas atomique : la contrainte correspondante ne va pas lier uniquement $\rho(c)$ et $\sigma(c)$, mais inclure d'autres types. Pour cela, on utilise la fonction $\cdot \odot \cdot$ qui à une liste de paramètres et d'instantiations correspondantes associe des types OCAML :

Définition 41 (Instantiation des contraintes de type)

$$\mathfrak{T}, \mathcal{C} \vdash \emptyset \odot \emptyset = \emptyset \qquad \frac{\mathfrak{T}, \mathcal{C} \vdash l_1 \odot l_2 = \gamma}{\mathfrak{T}, \mathcal{C} \vdash (x \ \textbf{\textit{in}} \ \tau, l_1) \odot (e, l_2) = \gamma}$$

$$\frac{\mathfrak{T}, \mathcal{C} \vdash l_1 \odot l_2 = \gamma \qquad c_2 \in \mathfrak{T}}{\mathfrak{T}, \mathcal{C} \vdash (c_1 \ \textbf{\textit{is}} \ I, l_1) \odot (c_2, l_2) = \rho(c_2), \sigma(c_2), \gamma}$$

$$\frac{\mathfrak{T}, \mathcal{C} \vdash l_1 \odot l_2 = \gamma \qquad c_2 \notin \mathfrak{T} \qquad c_2 \in \mathcal{C}}{\mathfrak{T}, \mathcal{C} \vdash (c_1 \ \textbf{\textit{is}} \ I, l_1) \odot (c_2, l_2) = \mathit{Mod}(c_2).\mathit{rep}, \mathit{Mod}(c_2).\mathit{stype}, \gamma}$$

Remarque 2. Comme un paramètre de collection ne peut être instantié que par une collection existante, les deux règles concernant les paramètres de collections sont suffisantes pour définir $\odot$.

Définition 42 (Contraintes de types sur les paramètres) *Soit s une espèce définie par deffoc, on définit les* contraintes des types des paramètres de s *de la manière suivante :*

$$\mathfrak{T}(s), \mathcal{C}, \mathcal{E} \vdash \mathbb{C}(\emptyset) = \emptyset \qquad \frac{\mathfrak{T}(s), \mathcal{C}, \mathcal{E} \vdash \mathbb{C}(l) = \kappa}{\mathfrak{T}, \mathcal{C}, \mathcal{E} \vdash \mathbb{C}(x \ \textbf{\textit{in}} \ \tau, l) = \kappa}$$

$$\frac{\begin{array}{c}\mathfrak{T}(s), \mathcal{C}, \mathcal{E} \vdash \mathbb{C}(l) = \kappa \qquad \mathcal{E}(I) = (c_1 \ . \ t_1, ..., c_k \ . \ t_k)\mathit{defs} \\ \mathfrak{T}(s), \mathcal{C}, \mathcal{E} \vdash c_1 \ . \ t_1, ..., c_k \ . \ t_k \odot a_1 ... a_k = \gamma\end{array}}{\mathfrak{T}, \mathcal{C}, \mathcal{E} \vdash \mathbb{C}(c \ \textbf{\textit{is}} \ I(a_1, ..., a_k), l) = \textbf{\textit{constraint}} \ \sigma(c) = (\rho(c), \phi)\mathcal{C}^t(I), \kappa}$$

où $\mathcal{C}^t(I)$ est l'interface associée à I dans la traduction OCAML *(voir 4.3.6)*

Lemme 23 *Soit s une espèce définie par deffoc et bien typée dans l'environnement $\mathcal{C}, \mathcal{E}, \Sigma, \Gamma$. Alors il existe une suite de contraintes ϕ telle que*

$$\mathfrak{T}(s), \mathcal{C} \vdash \mathbb{C}(c_1 \ . \ i_1, ..., c_k \ . \ i_k) = \phi$$

Preuve. Par récurrence sur k. Les paramètres d'entité ne pose pas de problème. Pour les paramètres de collection, il faut distinguer ceux qui ont une interface atomique de ceux qui ont eux-mêmes des paramètres qu'on instantie. Si le j-ième paramètre est un paramètre de collection de la forme c_j **is** i_j, dans tous les cas, par définition de $\mathfrak{T}$, on a $c_j \in \mathfrak{T}(s)$. Si i_j est atomique, cela suffit pour conclure : $\rho(c_j)$ et $\sigma(c_j)$ sont bien définis. Sinon, on raisonne par récurrence sur le nombre de paramètre de i_j. s étant bien typée, i_j est instantiée par le bon nombre de paramètres (règle COLL-PRM) : les deux arguments de $\odot$ ont la même taille. De plus, les instantiations sont toutes bien typées et, comme dans le lemme 20, les collections apparaissant dans les instantiations des paramètres sont nécessairement dans $\mathfrak{T}(s) \cup \mathcal{C}$, donc $\odot$ renvoie bien une liste d'instantiation de variable de type, et $\mathbb{C}$ une contrainte supplémentaire. □

4.3.4 Héritage

Après les contraintes de type, on trouve les clauses concernant l'héritage. Comme précédemment, il faut procéder à l'instantiation des paramètres de type lorsqu'on hérite d'espèces paramétrées. Par ailleurs, il faut distinguer ici les deux traductions, de l'espèce proprement dite vers une classe, et de l'interface correspondante vers un type de classe. Cette dernière est notée $\twoheadrightarrow^t$.

Définition 43 (Héritage entre types de classe) *Soit s une espèce définie par deffoc. on définit les héritages du type de classe de s de la manière suivante :*

$$\frac{\mathcal{E}(s_h) = (c_1 \ . \ t_1, ..., c_{p_f} \ . \ t_{h_f})defs \qquad \mathfrak{T}(s), \mathcal{C}, \mathcal{E} \vdash c_1 \ . \ t_1, ..., c_{p_f} \ . \ t_{p_f} \odot e_1, ..., e_{p_f} = \gamma}{\mathfrak{T}(s), \mathcal{C}, \mathcal{E} \vdash s_h(e_1, ..., e_{h_f}) \twoheadrightarrow^t \textbf{\textit{inherit}} \ [\rho(\textbf{\textit{rep}}), \gamma]\mathcal{C}^t(s_h)}$$

$\mathcal{C}^t(s_h)$ représentant le nom du type de classe correspondant à s_h dans la traduction (cf section 4.3.6)

Définition 44 (Héritage entre classes)
Soit s une espèce définie par deffoc. on définit les héritages de la classe de s de la manière suivante :

$$\frac{\mathcal{E}(s_h) = (c_1 \ . \ t_1, ..., c_{p_f} \ . \ t_{p_f})defs \qquad \mathfrak{T}(s), \mathcal{C} \vdash c_1 \ . \ t_1, ..., c_{p_f} \ . \ t_{p_f} \odot e_1, ..., e_{p_f} = \gamma \qquad \forall p, e_p \twoheadrightarrow e'_p}{\mathfrak{T}(s), \mathcal{C} \vdash s_h(e_1, ..., e_{p_f}) \twoheadrightarrow \textbf{\textit{inherit}} \ [\rho(\textbf{\textit{rep}}), \gamma]\mathcal{C}(s_h)(e'_1, ..., e'_n)}$$

$\mathcal{C}(s_h)$ représentant le nom de la classe correspondant à s_h dans la traduction (cf section 4.3.6)

Lemme 24 *si s est bien typée, ses héritage de classe et de type de classe sont bien définis.*

Preuve. Immédiat par récurrence sur le nombre d'espèces dont hérite s et le nombre de paramètres de chacune d'entre elles. □

4.3.5 Méthodes définies et virtuelles

Comme on l'a déjà dit, seules les méthodes déclarées ou définies dans l'espèce s elle-même sont traduites dans le corps de $\mathcal{C}(s)$. Les champs **sig** sont traduits par des méthodes virtuelles, et les champs **let** et **let rec** par des méthodes concrètes. Par ailleurs, comme dans la section précédente, on est amené à distinguer les deux cibles de la traduction, c'est à dire le type de classe et la classe.

Définition 45 (Corps d'un type de classe)
Soit s une espèce définie par deffoc. On définit les méthodes du type de classe *de s de la manière suivante :*

$$\mathfrak{T}(s), \mathcal{C} \vdash \emptyset \twoheadrightarrow^t \emptyset \qquad \frac{\mathfrak{T}(s), \mathcal{C} \vdash \Phi \twoheadrightarrow^t \Phi' \qquad \mathfrak{T}(s), \mathcal{C} \vdash \tau \twoheadrightarrow^t t}{\mathfrak{T}(s), \mathcal{C} \vdash \textbf{\textit{sig}}\ x\ \textbf{\textit{in}}\ \tau; \Phi \twoheadrightarrow^t \textbf{\textit{method virtual}}\ x : t; \Phi'}$$

$$\frac{\mathfrak{T}(s), \mathcal{C} \vdash \Phi \twoheadrightarrow^t \Phi' \qquad \mathfrak{T}(s), \mathcal{C} \vdash \tau \twoheadrightarrow^t t}{\mathfrak{T}(s), \mathcal{C} \vdash \textbf{\textit{let}}\ x\ \textbf{\textit{in}}\ \tau = e; \Phi \twoheadrightarrow^t \textbf{\textit{method}}\ x : t; \Phi'}$$

$$\frac{\mathfrak{T}(s), \mathcal{C} \vdash \Phi \twoheadrightarrow^t \Phi' \qquad \forall i, \mathfrak{T}(s), \mathcal{C} \vdash \tau_i \twoheadrightarrow^t t_i}{\mathfrak{T}(s), \mathcal{C} \vdash \textbf{\textit{let rec}}\ x_i\ \textbf{\textit{in}}\ \tau_i = e_i; \Phi \twoheadrightarrow^t \textbf{\textit{method}}\ x_i : t_i; \Phi'}$$

Définition 46 (Corps d'une classe) *Soit s une espèce définie par deffoc. On définit les* méthodes de la classe *de s de la manière suivante :*

$$\mathfrak{T}(s), \mathcal{C} \vdash \emptyset \twoheadrightarrow \emptyset \qquad \frac{\mathfrak{T}(s), \mathcal{C} \vdash \Phi \twoheadrightarrow \Phi' \qquad \mathfrak{T}(s), \mathcal{C} \vdash \tau \twoheadrightarrow t}{\mathfrak{T}(s), \mathcal{C} \vdash \textbf{\textit{sig}}\ x\ \textbf{\textit{in}}\ \tau; \Phi \twoheadrightarrow \textbf{\textit{method virtual}}\ x : t; \Phi'}$$

$$\frac{\mathfrak{T}(s), \mathcal{C} \vdash \Phi \twoheadrightarrow \Phi' \qquad \mathfrak{T}(s), \mathcal{C} \vdash \tau \twoheadrightarrow t \qquad \mathfrak{T}(s), \mathcal{C} \vdash e \twoheadrightarrow e'}{\mathfrak{T}(s), \mathcal{C} \vdash \textbf{\textit{let}}\ x\ \textbf{\textit{in}}\ \tau = e; \Phi \twoheadrightarrow \textbf{\textit{method}}\ x : t = e'; \Phi'}$$

$$\frac{\mathfrak{T}(s), \mathcal{C} \vdash \Phi \twoheadrightarrow \Phi' \qquad \forall i, \mathfrak{T}(s), \mathcal{C} \vdash \tau_i \twoheadrightarrow t_i \qquad \forall i, \mathfrak{T}(s), \mathcal{C} \vdash e_i \twoheadrightarrow e'_i}{\mathfrak{T}(s), \mathcal{C} \vdash \textbf{\textit{let rec}}\ x_i\ \textbf{\textit{in}}\ \tau_i = e_i; \Phi \twoheadrightarrow \textbf{\textit{method}}\ x_i : t_i = e'_i; \Phi'}$$

Lemme 25 *Si s est bien typée, la traduction de ses méthodes est bien définie :*

$$\exists \Phi,\ \mathfrak{T}(s), \mathcal{C}, \mathcal{E} \vdash \phi_1 ... \phi_m \twoheadrightarrow \Phi$$

Preuve. Immédiat par cas sur les champs ϕ_i. □

4.3.6 Traduction complète d'une espèce

Nous avons maintenant toutes les composantes d'une classe OCAML représentant une espèce FOC donnée. Il reste à examiner comment les rassembler pour obtenir une définition complète de classe et de type de classe. Pour cela, on se donne deux fonctions, $\mathcal{C}(.)$ et $\mathcal{C}^t(.)$, qui associent à tout nom d'espèce de FOC un nom d'une classe syntaxique différente de celles de FOC. De plus, il faut marquer comme virtuelles les classes qui ne définissent pas toutes leurs méthodes (héritées ou non).

Définition 47 (Type de classe représentant une interface)
Soit s une espèce définie par

species s(`prms`) **inherits** `inh` = Φ

On définit le type de classe de son interface de la manière suivante :

$$\frac{\mathcal{L}(prm) = \pi \qquad \mathbb{C}(prm) = \kappa \qquad \mathcal{N}(s) = \mathcal{D}(s) \qquad inh \twoheadrightarrow^t \iota \qquad \Phi \twoheadrightarrow^t M \qquad \mathcal{C}^t(s) = s'}{s \twoheadrightarrow^t \textbf{\textit{class type}}\ [\rho(\textbf{\textit{self}}), \pi]s' = \quad \textbf{\textit{object}}\ \kappa\ \iota\ M\ \textbf{\textit{end}}}$$

$$\frac{\mathcal{L}(prm) = \pi \qquad \mathbb{C}(prm) = \kappa \qquad \mathcal{N}(s) \neq \mathcal{D}(s) \qquad inh \twoheadrightarrow^t \iota \qquad \Phi \twoheadrightarrow^t M \qquad \mathcal{C}^t(s) = s'}{s \twoheadrightarrow^t \textbf{\textit{class type virtual}}\ [\rho(\textbf{\textit{self}}), \pi]s' = \quad \textbf{\textit{object}}\ \kappa\ \iota\ M\ \textbf{\textit{end}}}$$

Définition 48 (Classe représentant une espèce) *Soit s une espèce définie par :*

species s(`prms`) **inherits** `inh` $= \Phi$

On définit la classe représentant s de la manière suivante :

$$\frac{\mathcal{N}(s) = \mathcal{D}(s) \qquad prm \twoheadrightarrow f \qquad \mathcal{L}(prm) = \pi \qquad \mathbb{C}(\textbf{\textit{self}}) == \sigma \qquad \mathbb{C}(prm) = \kappa \qquad inh \twoheadrightarrow \iota \qquad \Phi \twoheadrightarrow M \qquad \mathcal{C}(s) = s' \qquad Mod(\textbf{\textit{self}}) = this}{s \twoheadrightarrow \textbf{\textit{class}}\ [\rho(\textbf{\textit{self}}), \pi]s'(arg) = \quad \textbf{\textit{object}}(self)\ \sigma\ \kappa\ \iota\ M\ \textbf{\textit{end}}}$$

$$\frac{\mathcal{N}(s) \neq \mathcal{D}(s) \qquad prm \twoheadrightarrow f \qquad \mathcal{L}(prm) = \pi \qquad \mathbb{C}(\textbf{\textit{self}}) == \sigma \qquad \mathbb{C}(prm) = \kappa \qquad inh \twoheadrightarrow \iota \qquad \Phi \twoheadrightarrow M \qquad \mathcal{C}(s) = s' \qquad Mod(\textbf{\textit{self}}) = this}{s \twoheadrightarrow \textbf{\textit{class virtual}}\ [\rho(\textbf{\textit{self}}), \pi]s'(arg) = \quad \textbf{\textit{object}}(self)\ \sigma\ \kappa\ \iota\ M\ \textbf{\textit{end}}}$$

4.3.7 Modules et traduction d'une collection

Comparées aux espèces, les collections se traduisent beaucoup plus simplement : la seule difficulté consiste à trouver les termes représentant les instantiations de paramètres lorsque la collection implante une espèce paramétrée. Toutefois, on peut remarquer que ce travail a déjà été fait lors de la traduction de l'héritage : la seule différence est qu'ici les instances des paramètres de collections sont forcément des collections déjà existantes (c'est à dire dans $\mathcal{C}$), une collection n'ayant pas de paramètre.

Définition 49 (types utilisés dans les modules) *Soit c une collection définie par :*

collection `c` **implements** $s(e_1,\ldots,e_{p_f})$

On définit les types utilisés dans le module représentant c ($types(c)$) et le type de module représentant c ($types^t(c)$), et la fonction auxiliaire $\boxdot$ de la

manière suivante :

$$\mathcal{C} \vdash \emptyset \boxdot \emptyset = \emptyset \qquad \frac{\mathcal{C} \vdash l_1 \boxdot l_2 = l'}{\mathcal{C} \vdash (x\ \textbf{\textit{in}}\ \tau, l_1) \boxdot (e, l_2) = l'}$$

$$\frac{\mathcal{C} \vdash l_1 \boxdot l_2 = l' \qquad c \in \mathcal{C} \qquad Mod(c) = C \qquad r(x) = x_r \qquad i(x) = x_s}{\mathcal{C} \vdash (x\ \textbf{\textit{is}}\ i, l_1) \boxdot (c, l_2) = x_r = C.rep\ \textbf{\textit{and}}\ x_i = C.stype\ \textbf{\textit{and}}\ l'}$$

$$\frac{\begin{array}{c}\mathcal{E}(s) = (c_1\ .\ t_1, ..., c_{p_f}\ .\ t_{p_f})\,defs \\ \mathcal{C} \vdash c_1\ .\ t_1, ..., c_{p_f}\ .\ t_{p_f} \boxdot e_1, ..., e_{p_f} = \alpha \\ \forall p \text{ tel que } c_p \text{ paramètre de collection}, r(c_p) = r_p \text{ et } i(c_p) = s_p\end{array}}{\mathcal{E}, \mathcal{C} \vdash types^t(c) = \textbf{\textit{type}}\ rep\ \textbf{\textit{and}}\ \alpha\ \textbf{\textit{and}}\ stype = (rep, \{r_p, s_p\})\mathcal{C}^t(s)}$$

$$\frac{\begin{array}{c}\mathcal{E}(s) = (c_1\ .\ t_1, ..., c_{p_f}\ .\ t_{p_f})\,defs \\ \mathcal{C} \vdash c_1\ .\ t_1, ..., c_{p_f}\ .\ t_{p_f} \boxdot e_1, ..., e_{p_f} = \alpha \qquad \mathcal{C} \vdash \mathcal{B}_s(\textbf{\textit{rep}}) \twoheadrightarrow \tau \\ \forall p \text{ tel que } c_p \text{ paramètre de collection}, r(c_p) = r_p \text{ et } i(c_i) = s_i\end{array}}{\mathcal{E}, \mathcal{C} \vdash types(c) = \textbf{\textit{type}}\ rep\ = \tau\ \textbf{\textit{and}}\ \alpha\ \textbf{\textit{and}}\ stype = (rep, r_p, s_p)\mathcal{C}^t(s)}$$

Pour définir le module correspondant, on se donne en outre une fonction *Mod*(.) qui à un nom de collection associe un nom de module de OCAML (c'est à dire commençant par une majuscule).

Définition 50 (Module représentant une collection)
Soit c une collection définie par :

collection `c` **implements** $s(e_1,\ldots,$`e_{p_f}`$)$

On définit le module représentant c de la manière suivante :

$$\frac{\begin{array}{c}Mod(c) = C \\ \mathcal{EC} \vdash types^t(c) = \alpha_1 \qquad \mathcal{E}, \mathcal{C} \vdash types(c) = \alpha_2 \qquad \forall i, a_i \twoheadrightarrow a'_i\end{array}}{\begin{array}{c}c \twoheadrightarrow \textbf{\textit{module}}\ C : \\ \textbf{\textit{sig}}\ \alpha_1\ \textbf{\textit{val}}\ obj : stype\ \textbf{\textit{struct}}\ \alpha_2\ \textbf{\textit{let}}\ obj = (\textbf{\textit{new}}\ \mathcal{C}(s)(a'_1, ..., a'_k) :> i(\textbf{\textit{self}}))\end{array}}$$

4.4 Optimisations : variables d'instance

La traduction proposée ci-dessous ne fournit pas toujours un codage optimum de la hiérarchie mathématique, en raison d'une différence fondamentale entre les objets de OCAML et les collections de FOC. Ces dernières sont statiques : elles représentent une implantation particulière —donc un modèle particulier— d'une certaine structure algébrique. Au contraire, un objet de OCAML [RV98] est une entité complètement dynamique : il possède un état interne, que ses méthodes peuvent manipuler, et qui est donc susceptible d'évoluer au cours de l'exécution d'un programme. De ce fait, aucune méthode d'un objet de OCAML ne peut être considérée comme constante. En

effet, elle prend implicitement en paramètre l'état courant de l'objet. Ainsi, dans le code suivant :

```
class a =
  object
    val y = 2
    method x : int = 40 + y
end
let a_inst = new a
```

Chaque appel à `a_inst#x` provoquera une nouvelle évaluation du corps de la méthode, pour tenir compte d'un possible changement de l'état interne de `a_inst`. Dans la classe issue de la traduction des espèces de FOC, cette réévaluation est inutile, puisque les objets obtenus n'ont pas d'état interne : la valeur des constantes est fixée au moment où on crée une collection, c'est à dire pour OCAML une nouvelle instance de la classe. Or un certain nombre de ces constantes sont le résultat de calculs importants. En particulier, certaines méthodes de FOC créent une nouvelle collection. Par exemple, une implantation récursive des polynômes à plusieurs variables contient une méthode spéciale :

```
species polynomes_multivaries(A is anneau) inherits anneau ... =
  ...
  let updom = polynomes_univaries(self) ;
  ...
```

permettant de passer d'un ensemble $A[X_1, ..., X_n]$ de polynômes à n variables, à un ensemble de polynômes à $n+1$ variables $A[X_1, ..., X_n][X_{n+1}] = A[X_1, ..., X_n, X_{n+1}]$, . Elle est utilisée par les différentes opérations sur les polynômes (addition, multiplication, ...) pour se placer automatiquement dans le "bon" espace de variables —par exemple, pour additionner X et Y, il faut travailler dans $A[X, Y]$. En OCAML, cela aboutit à créer une nouvelle instance de la classe des polynômes. Compte tenu du coût lié à la création d'un nouvel objet, il est important de minimiser le nombre d'instances effectivement créées. Nous allons maintenant examiner différentes possibilités permettant de ne pas réévaluer à chaque appel des corps de méthodes dont la valeur est fixée à la création de l'objet.

4.4.1 Variables d'instance

En OCAML, l'état interne des objets est stocké dans des *variables d'instance*, qui sont initialisées lors de la création d'un objet, et ne peuvent plus être accédées ensuite que par l'intermédiaire des méthodes de l'objet en question. A première vue, elles semblent donc bien répondre à notre problème. Toutefois, on ne peut les utiliser telles quelles. En effet, ces variables ne

peuvent faire référence aux méthodes ou aux variables de la classe qu'on veut créer : leur définition ne peut faire appel qu'à des entités définies auparavant. Ainsi, dans le code FOC suivant :

```
species nat_ring =
  rep = nat;
  let zero in self = 0;
  let succ in self → self = fun x → (S x);
  let one = self !succ(self !zero);
end
```

`one` dépend de `zero` et `succ`, et le code OCAML suivant n'est pas correct, car l'expression définissant la variable d'instance `one` se refère à l'objet `self`, qui ne fait pas partie de son contexte d'évaluation.

```
class nat_ring =
object(self)
  val zero = 0
  val one = self#succ zero (* ERREUR *)
  method get_zero = zero
  method succ = fun x → S x
  method get_one = one
end
```

Une solution consisterait à retarder l'évaluation du corps de `one` jusqu'à ce que l'objet soit effectivement créé et ses méthodes accessibles. Toutefois, OCAML exige que toutes les variables d'instance soient initialisées : il faut donc donner une valeur *temporaire* à `one`, qu'on modifiera *une* fois, durant la phase d'initialisation de l'objet correspondant. Pour cela, nous avons étudié trois possibilités :

Type `option` Le type `option` est un type somme à deux constructeurs : `None`, d'arité 0 et `Some` qui attend un argument. On donne à `one` la valeur `None`, et on modifie le code de la méthode `get_one`, qui effectuera l'initialisation lors du premier appel. Lors des appels suivants, la variable `one` aura sa valeur définitive, et on n'aura plus aucun calcul à faire (hormis le filtrage

de `get_one`). On obtient alors la représentation suivante :

```
class nat =
object(self)
  ...
  val mutable one = None
  ...
  method get_one =
    match one with
      None → (* premier appel : il faut initialiser *)
        let x = self#succ zero in
          one <- Some x; x
    | Some s → s
  ...
end
```

Cette méthode présente toutefois trois inconvénients :

- `get_one` n'est pas un simple appel à la variable `one`, et on effectue un pattern-matching "inutile" à chaque appel à `get_one`. Toutefois, ce coût n'est pas énorme en pratique.
- Il n'est plus possible d'utiliser directement `one` dans les autres méthodes de `nat` : il faut obligatoirement passer par `get_one` (ou déplier le filtrage). Là encore le coût supplémentaire ne semble pas rédhibitoire en pratique.
- Enfin, cette solution s'adapte assez mal au cas où on cherche à définir plusieurs méthodes mutuellement récursives, comme cela peut arriver avec les méthodes renvoyant des collections. L'initialisation de l'une d'entre elles risque de provoquer une boucle infinie de la forme :

```
class foo =
object(self)
...
  method get_x =
    match x with
      None → ... self#get_y ...
    | Some x → x
  method get_y =
    match y with
      None → ... self#get_x ...
    | Some y → y
...
end
```

`initializer` Une seconde possibilité consiste à se passer d'intermédiaire, et à donner directement la valeur de la variable au moment de la création

de l'objet. En effet, on connaît un ordre dans lequel initialiser les variables d'instance ainsi créées. Il suffit pour cela de regarder les résultats de l'analyse de dépendance (cf 3) faite sur le code FOC. Il existe un moyen de passer cette information au compilateur OCAML. En effet, chaque classe est libre de définir un `initializer` [LDG+02], qui est une expression qui sera évaluée à chaque création d'un objet de cette classe, à la fin de l'initialisation, c'est à dire à un moment où `self` est présent dans le contexte d'évaluation.

On définit tout d'abord une classe qui sera la racine de la hiérarchie OCAML. Elle définit une méthode `__init` qui par défaut ne fait rien, et utilise `initializer` pour que `__init` soit appelée à chaque création d'un objet implantant `init`.

```
class init =
  object(self)
    method private __init () = ()
    initializer self#__init ()
end
```

Dès lors, chacune des classes traduites de FOC peut redéfinir `__init` pour initialiser ses variables d'instance suivant l'ordre défini par l'analyse de dépendance pour l'espèce correspondante. Si on reprend l'exemple de `nat`, on va obtenir le code suivant pour la traduction de `one` :

```
class nat =
  object(self)
    inherit init
    ...
    val mutable one = Obj.magic()
    ...
    method get_one = one
    ...
    method private __init () = one <- self#succ zero
end
```

Il faut utiliser la construction `Obj.magic` de OCAML, fonction de type $\alpha \to \beta$. Appliquée à `()` l'unique élément du type `unit`, elle permet d'obtenir un élément de n'importe quel type, et donc de passer la phase de typage d'OCAML. Cependant, on ne peut pas utiliser cette construction n'importe comment. En effet, tenter d'accéder à une variable dont la valeur serait effectivement `Obj.magic ()` provoquerait l'arrêt brutal du programme.

Ici, cette utilisation est légitimée par les analyses faites par le compilateur FOC. La séquence d'initialisation (le corps de la méthode `init`) se fait en respectant l'ordre donné par ces analyses, de sorte qu'aucune variable n'est utilisée alors que sa valeur est encore `Obj.magic ()`.

Plus précisément, considérons une espèce e dont la forme normale est $\{x_i, \tau_i, d_i\}_{i=1..n}$. Soient $k_1 < k_2 < \ldots < k_m$ les indices tels que τ_{k_j} soit une constante et $d_{k_j} \neq \bot$. On définit `__init` de la manière suivante :

method private `__init ()`$= x_{k_1} <- d_{k_1}$; ... ; $x_{k_m} <- d_{k_m}$;

Par hypothèse, l'évaluation de d_{k_j} ne dépend que des x_i, avec $i < k_j$. Ces dernières sont représentées en OCAML soit par des méthodes, soit par des variables d'instance, s'il existe n tel que $i = k_n$.

- Si x_i est traduite par une méthode, l'appel à x_i est correct, dans l'environnement où `__init` est exécuté, c'est à dire au moment de la création d'une nouvelle instance d'une classe héritant de e. En effet, on ne peut définir d'instance d'une classe que si toutes les méthodes de cette dernière ont reçu une définition : les éventuelles méthodes encore virtuelles de e sont alors définies.
- si x_i est traduite par une variable d'instance, on a $i = k_n < k_j$. D'après la définition de `__init`, x_{k_n} a reçu sa définition au moment où on évalue d_{k_j}, et l'appel à x_{k_n} est correct : on ne cherche pas à accéder à une variable dont la valeur est `Obj.magic ()`.

Cette approche est plus efficace que l'utilisation d'un type option : il n'y a plus de pattern matching, et les méthodes de la classe peuvent appeler directement les variables d'instance. Expérimentalement, on constate un gain d'environ 20 à 25 % par appel. Toutefois, cela ne résout pas le problème des méthodes explicitement signalées comme récursives par le programmeur FOC. Nous allons maintenant examiner le moyen de régler ce dernier cas.

Appels paresseux Une dernière possibilité consiste à utiliser le module `Lazy` de OCAML. Le rôle de ce module est de créer des *suspensions*, c'est à dire de retarder l'évaluation d'une expression au moyen du mot-clé **lazy**, jusqu'à ce qu'on demande explicitement à utiliser sa valeur par un appel à `Lazy.force`. Cette fonction teste si son argument est une valeur (dans ce cas elle la renvoie directement) ou une suspension, qu'il reste à évaluer.

```
class nat =
  object(self)
    inherit init
    ...
    val mutable one = Obj.magic()
    ...
    method get_one =
      try Lazy.force one
      with Lazy.Undefined → ...
    ...
    method __init = one <- lazy self#succ zero
end
```

Le principal avantage de cette solution sur l'utilisation d'un type option est que `Lazy` est capable de détecter une éventuelle boucle (un appel à `Lazy.force` sur la suspension qu'on est en déjà en train d'évaluer), levant alors l'exception `Undefined`, qui peut éventuellement être rattrapée. C'est donc la solution qu'on retiendra lorsqu'on se trouve à l'intérieur d'un bloc `let rec` de FOC, pour lequel l'analyse de dépendance ne donne pas d'indication sur l'ordre dans lequel initialiser les variables. Par contre, on n'évite pas les surcoûts liés aux appels de méthodes. De plus, comme dans le cas précédent, on ne peut donner de valeur initiale à `one`, et il faut utiliser `Obj.magic`. Ici, c'est l'utilisation de **lazy** qui garantit qu'on n'accède pas à une variable non initialisée : aucun calcul n'est fait avant la fin de la phase d'initialisation, qui se contente d'affecter des suspensions à chacune des variables d'instance.

4.4.2 Méthodes locales

FOC offre également la possibilité de définir des *méthodes locales*. Ces méthodes ne sont accessibles qu'au sein de l'espèce où elles sont définies, mais n'apparaissent pas dans l'interface correspondante, et ne sont pas prises en compte lors de l'héritage. Elles correspondent donc à des fonctions utilitaires, utilisées par les autres méthodes, mais invisibles en dehors de l'espèce. Deux possibilités de traduction sont offertes :

- En faire des variables dont la portée est le corps de la classe OCAML correspondant à l'espèce. C'est la méthode qui a été utilisée lors du premier développement de la librairie FOC directement en OCAML par Renaud Rioboo.
- définir une variable locale dans chaque méthode qui y fait appel (*i.e.* qui en dépend).

Par exemple, si on part de l'espèce suivante :

```
species a =
  let x = 1;
  local let y = self !x + 1;
  let f(z)=self !y + z;
end
```

On obtiendra dans le premier cas :

```
class a =
  let y x = x + 1 in
object(self)
  val x = 1
  method f z = (y self#x) + z
end
```

Les dépendances des méthodes locales deviennent ainsi des arguments supplémentaires à fournir à chaque appel. En effet, `y` ne peut appeler `self` : ce

nom n'existe pas dans l'environnement où est défini `y`. Cette abstraction n'est pas nécessaire dans le second cas, où `y` va être défini seulement à l'intérieur de `f` :

```
class a =
object(self)
  val x = 1
  method f z = let y = self#x + 1 in y + z
end
```

Cette seconde voie permet de limiter la portée de ces méthodes locales aux endroits où elle sont effectivement utilisées, et réduit le nombre d'appels de fonctions nécessaires, puisqu'on n'abstrait plus `y` par rapport à `x`. En contrepartie, cela aboutit à dupliquer des définitions. Des tests sont nécessaires pour départager les deux approches.

4.4.3 Davantage de variables d'instance

Comme il est remarqué dans [Ler90], il n'y a pas que les constantes qui peuvent être évaluées lors de la création d'un objet. Il peut également être intéressant de forcer l'évaluation partielle du corps d'une fonction, quand cette dernière ne commence pas par une λ-abstraction, mais par une ou plusieurs liaisons locales. Par exemple, dans le code suivant :

```
species a =
rep = int;
...
let f in self → self =
  let x = 40 + 2 in
  fun y →
    x * y;
end
```

Il n'y a pas besoin de connaître l'argument de `f` pour calculer la valeur de `x`. Ainsi, si on traduit `f` en une variable d'instance, `x` ne sera évalué qu'au moment de la création d'un nouvel objet, alors que si on en fait une méthode, il sera réévalué à chaque appel. Notons que ce cas se produit aussi lorsqu'on déplie les méthodes locales comme expliqué dans la section précédente.

De même que dans le cas des constantes, l'analyse de dépendance de FOC permet de donner un ordre d'initialisation dans lequel l'évaluation partielle des fonctions ne fait appel qu'à des méthodes et à des variables d'instance déjà initialisées de l'objet qu'on est en train de créer. Si on reprend les notations de la section 4.4.1, on augmente le nombre d'indices k_j des opérations à traduire par des variables d'instance, mais la même technique reste valable.

Par ailleurs, les fonctions récursives ne sont jamais transformées en variables d'instance. En effet, l'analyse de dépendance telle qu'elle est faite actuellement ne permet pas d'assurer que l'évaluation partielle décrite ci-dessus ne contient pas un appel récursif, comme le montre l'exemple ci-dessous :

```
species a =
rep = int;
...
let rec f in self → self → self =
  let default = self !f(0) in
  fun x → fun y → ...
```

Aucune des méthodes pour transformer `f` en variable d'instance ne marche. En effet, l'initialisation de `f` commence par un appel à `f` elle-même. Ce n'est pas le cas dans cet autre exemple, où la définition de `default` est η-expansée, ce qui retarde l'appel à `self !f` : celui-ci n'est pas fait à l'initialisation, qui dès lors se déroule normalement.

```
species a =
rep = int;
...
let rec f in self → self → self =
  let default(x) = self !f(0,x) in
  fun x → fun y → ...
```

Pour résoudre ce problème, on pourrait raffiner le calcul du graphe de dépendances en suivant les idées développées dans [HL02]. Il s'agit de différencier les cas où la récursion mutuelle est "protégée" ou non par une λ-abstraction. Dans le premier cas, on pourrait s'autoriser à transformer la méthode en une variable d'instance.

Enfin, on peut modifier les corps des fonctions non récursives, de façon à repousser les abstractions le plus loin possible. Pour cela, il faut analyser les différentes sous-expressions pour vérifier qu'elles ne dépendent pas, directement ou indirectement de ces abstractions.

Plus précisément, soient x, y deux variables $x \neq y$. Si $expr_1$ est une expression telle que x n'apparaisse pas libre dans $expr_1$ et $expr_2$ une expression, alors

$$\textbf{fun } x \texttt{ -> } \textbf{ let } y = expr_1 \textbf{ in } expr_2$$

est équivalent à

$$\textbf{let } y = expr_1 \textbf{ in } \textbf{ fun } x \texttt{ -> } expr_2$$

dans la mesure où les expressions de FOC n'ont pas d'effet de bord. Dans le cas contraire, il faudrait vérifier que $expr_1$ ne contient pas de tels effets pour pouvoir faire la transformation.

En itérant cette transformation, on peut faire remonter les liaisons locales d'une fonction qui ne dépendent pas de ses arguments. Cela permet de réduire encore la part des évaluations faites à chaque appel de méthode par rapport à celles qui sont faites une fois lors de l'initialisation de l'objet.

Afin de voir quels sont les gains de performance que l'on peut attendre de ces constructions, ainsi que de la traduction vers des enregistrements OCAML présentée dans le chapitre 8, une série de benchmarks a été réalisée à partir des algorithmes de la librairie standard de FOC. Les résultats sont présentés dans l'annexe A.

Chapitre 5

Correction de la traduction OCaml

Position du problème

Nous avons vu dans le chapitre précédent comment représenter des espèces et des collections à l'aide des traits objets et des modules d'OCaml. Nous allons maintenant nous attacher à montrer que la traduction proposée est correcte. Pour cela nous allons répondre aux deux questions suivantes :

- Le code obtenu par traduction d'une espèce bien typée est-il lui-même bien typé pour OCaml ?
- Quel degré de correspondance y a-t-il entre le code Foc et sa traduction en OCaml ?

Répondre à la première question revient à donner des arguments de *sûreté* du typage de Foc vis-à-vis des transformations qu'on se propose de lui faire subir. Comme nos langages-cibles (OCaml, mais surtout Coq, que nous examinerons dans les deux chapitres suivants) sont eux-mêmes dotés de systèmes de type puissants, nous disposons ainsi d'un deuxième niveau de vérification de la correction du code, après la phase d'analyse du compilateur `focc`.

Toutefois, cette vérification n'aurait qu'un intérêt limité si les traductions ne respectaient pas certains invariants garantissant une forme *d'équivalence* entre les trois mondes. Plus précisément, cette équivalence entre Foc et OCaml, ainsi que la correspondance entre Foc et Coq présentée dans le chapitre 7 permet de rapprocher le code appelé à être exécuté (OCaml) et les preuves de correction (Coq). En particulier, il s'agit de montrer que les appels de méthode conduisent bien à la sélection du code sur lequel on a fait des preuves.

Nous allons maintenant nous intéresser aux liens entre OCaml et Foc. Pour cela, on utilisera la première version de la traduction avec objets, qui n'utilise que des méthodes OCaml pour traduire les méthodes Foc. Nous

examinerons d'abord le cas des espèces atomiques, avant de nous intéresser au sort des paramètres. Enfin, nous verrons comment se comportent les collections.

5.1 Traduction des environnements

En premier lieu, il convient de voir dans quel environnement de typage on va se placer pour vérifier la correction des expressions traduites.

Définition 51 (Traduction d'un environnement de typage)
Soient $\mathcal{C}, \mathcal{E}, \Sigma, \Gamma$ un environnement de typage Foc *et $\mathfrak{T}$ un environnement de variables de type. On définit Γ', l'environnement de typage* Ocaml *correspondant de la manière suivante :*

$$\textsc{Coll1}\quad \frac{c \in \mathcal{C} \qquad c \notin \mathfrak{T} \qquad Mod(c) = C \qquad \mathcal{C}(c) = \{x_i : \tau_i\} \qquad \tau_i \twoheadrightarrow \tau_i'}{\Gamma'(C.obj) = \langle x_i : \tau_i' \rangle}$$

$$\textsc{Coll2}\quad \frac{c \in \mathcal{C} \qquad c \in \mathfrak{T} \qquad \mathcal{C}(c) = \{x_i : \tau_i\} \qquad \tau_i \twoheadrightarrow \tau_i'}{\Gamma'(c) = \langle x_i : \tau_i' \rangle}$$

$$\textsc{Spec}\quad \frac{s \in \mathcal{E} \qquad \mathcal{C}(s) = s' \qquad \mathcal{C}^t(s) = \sigma}{\Gamma'(s') = \sigma}$$

$$\textsc{Self-conc}\quad \frac{\begin{array}{c}\Sigma = \{x_i : \tau_i = e_i\} \\ Mod(self) = this \qquad \rho(self) = \alpha \qquad \Sigma(rep) = \tau \qquad \tau_i \twoheadrightarrow \tau_i'\end{array}}{\Gamma'(this) = \forall \alpha (\alpha = \tau) \langle x_i : \tau_i' \rangle}$$

$$\textsc{Self-abst}\quad \frac{\begin{array}{c}\Sigma = \{x_i : \tau_i = e_i\} \\ Mod(self) = this \qquad \rho(self) = \alpha \qquad \Sigma(rep) = \bot \qquad \tau_i \twoheadrightarrow \tau_i'\end{array}}{\Gamma'(this) = \forall \alpha \langle x_i : \tau_i' \rangle}$$

$$\textsc{Var}\quad \frac{\Gamma(x) = \tau \qquad \tau \twoheadrightarrow \tau'}{\Gamma'(x) = \tau'}$$

On notera $\mathcal{C}, \mathcal{E}, \Sigma, \Gamma \twoheadrightarrow \Gamma'$.

Var ajoute simplement dans Γ' les variables locales de Foc.

La règle Coll1 s'applique à une collection c de Foc. Elle place dans Γ' un module ayant un objet *obj* dont les méthodes sont les mêmes que celle

de c. COLL2 traite le cas des paramètres de collection, qui sont vus comme des variables ayant un type objet reprenant les méthodes de l'interface sous-jacente. De même, SPEC déclare un type objet pour chaque espèce de $\mathcal{E}$.

Par ailleurs, on place dans l'environnement un objet, *this*, correspondant à la classe qu'on est en train de définir. Il faut distinguer deux cas suivant que le type support est défini (SELF-CONC, où on ajoute une contrainte de type) ou non (SELF-ABST).

On peut dès lors s'intéresser à une expression FOC bien typée dans un certain contexte et à sa traduction en OCAML. Le lemme ci-dessous montre qu'on peut lui donner comme type la traduction du type FOC correspondant. Ce lemme servira de base pour la correction de la traduction des espèces atomiques et paramétrées. On s'assurera en effet à cette occasion que la traduction du corps des méthodes a bien comme type la traduction du type inféré par FOC.

Lemme 26 (Typage des expressions) *Soient e une expression et τ un type tels que $\Omega \vdash e : \tau$ pour un certain environnement* FOC. *Alors, avec $e \rightarrowtriangle e'$, $\Omega \rightarrowtriangle \Gamma'$, et $\tau \rightarrowtriangle \tau'$ il vient $\Gamma' \vdash e' : \tau'$*

Preuve. Immédiat par induction sur la dérivation de typage de e (cf figure 3.1) :

- Règle VAR : si e est une variable x telle que $\Gamma(x) = \sigma$, avec $\tau \leq \mathit{Inst}(\sigma)$ $e' = x$, et $\Gamma'(x) = \sigma'$ (avec $\sigma \rightarrowtriangle \sigma'$). Il reste à montrer que τ' est une instance de σ. Or les règles d'unification de def.9 sont identiques aux règles d'unification de OCAML, sauf pour SELF1 et SELF2. Ces deux règles ne peuvent être appliquées que lorsque le type support **self** est défini dans Σ. Dans ce cas, $\Gamma'(\mathbf{self})$ porte une contrainte de type de la forme $\rho(\mathbf{self}) = t$, avec t traduction du type support : l'unification peut avoir lieu.
- Règles ABS, LET, LET REC et APP : ces règles sont identiques à leur équivalent OCAML et le typage est donc possible.
- Règle METHCALL : si e est de la forme $c!x$, par définition de la traduction d'une expression, on a deux cas possibles pour la traduction de c : soit $\mathit{Mod}(c).\mathit{obj}$ si c est une collection, soit c si on est en présence d'un paramètre. Dans les deux cas, l'identificateur en question est, par définition de Γ', une instance d'une classe dont la méthode x a le type τ'.
- Règle SELFCALL : de même que dans le cas précédent, *self* est un objet possédant une méthode x de type τ'.

□

5.2 Les espèces atomiques

Le point le plus important consiste à vérifier que les appels de méthode en OCAML sont corrects. Pour cela, on peut commencer par montrer que toutes les méthodes calculatoires d'une espèce s de FOC sont introduites dans la classe correspondante, et que les méthodes définies sont également définies en OCAML.

Lemme 27 (types de classe OCAML) *Soient s une espèce* FOC*, et $x \in \mathcal{N}(s)$, tel que $\mathcal{T}_s(x) = \tau$ (τ un type* FOC *quelconque). Le type de classe $\mathcal{C}^t(s)$ correspondant est une instance de*

$$< x : \tau; .. >$$

Preuve. Si x est présent dans le corps de s, le résultat est immédiat par définition de $\twoheadrightarrow^t$. Sinon, x est hérité d'une des espèces parent de s, et par induction sur la hauteur de l'arbre d'héritage, il est présent avec le type τ dans une des classes dont hérite $\mathcal{C}^t(s)$. □

Lemme 28 (Méthodes définies et virtuelles)
Soient s une espèce de FOC*, et $x \in \mathcal{D}(s)$. La classe $\mathcal{C}(s)$ correspondante possède une méthode définie x. De plus, si $x \leftarrow s = s'$ (definition 30), alors la définition de x est héritée de la classe $\mathcal{C}(s')$ dans la traduction* OCAML*.*

Preuve. Si x est (re)définie dans le corps de s, le résultat est immédiat par définition de $\twoheadrightarrow$. Sinon, on raisonne par induction sur la longueur du chemin d'héritage entre s et s'. En cas d'héritage multiple, FOC comme OCAML prennent la définition de la dernière espèce (respectivement classe) dont s hérite. Comme cet ordre est préservé lors de la traduction de la clause d'héritage de FOC on sélectionne bien la bonne définition. □

Une fois qu'on a montré que les méthodes FOC et les méthodes OCAML se correspondaient bien, il faut s'intéresser au typage par OCAML des méthodes produites. Pour cela il convient de montrer que le typage du corps des méthodes nouvellement définies est correct. En particulier, il s'agit de voir à quoi correspond au niveau OCAML les règles de typage SELF permettant de convertir la représentation du type support en **self**.

À partir des lemmes précédents, on peut déduire le principal résultat de cette section, c'est à dire que les classes correspondant à des espèces atomiques sont bien typées en OCAML. De plus, on va montrer qu'on peut leur donner comme type le type de classe donné par $\twoheadrightarrow^t$.

Théoreme 6 (Correction de la traduction des espèces atomiques)
Soit s une espèce FOC *sans paramètre bien typée dans le contexte $\mathcal{C}, \mathcal{E}, \Gamma$ et telle que $s \twoheadrightarrow^t \sigma$ et $s \twoheadrightarrow c$. Alors, avec Γ' la traduction du contexte* FOC *en* OCAML*, c est une définition de classe bien typée, et $\Gamma' \vdash c : \sigma$.*

Preuve. D'après la règle WELL-TYPED-SPEC, le typage de chaque corps de méthode définie dans s se fait dans l'environnement $\mathcal{C}, \mathcal{E}, norm(s), \Gamma$. Pour toute méthode $x \in \mathcal{D}(s)$, on a donc, avec $\mathcal{B}_s(x) \twoheadrightarrow e$ et $\mathcal{T}_s(x) \twoheadrightarrow \tau$, $\Gamma' \vdash e : \tau$. Par ailleurs, on a vu dans les lemmes 27 et 28 que $\mathcal{C}^t(s)$ et $\mathcal{C}(s)$ avaient les mêmes noms de méthode. Le type des méthodes déclarées étant dans les deux cas la traduction directe de celui de FOC, $\mathcal{C}^t(s)$ et $\mathcal{C}(s)$ ont bien la même signature. Seule la contrainte sur le type support diffère : elle n'est pas présente dans $\mathcal{C}^t(s)$. Toutefois, cette contrainte n'est nécessaire que pour typer le corps des méthodes, et peut donc être enlevée si on ne s'intéresse qu'aux interfaces. □

Dès lors, le typage des classes correspondant aux espèces atomiques par OCAML permet de vérifier que l'implantation du compilateur `focc` est correcte[1], au moins en ce qui concerne le typage du corps des méthodes. De plus, comme WELL-TYPED-SPEC s'assure que le type d'une méthode est préservé au cours de l'héritage, on peut déduire du théorème précédent que la traduction de l'héritage en OCAML est correcte :

Corollaire 3 (Traduction de l'héritage)
Soit s une espèce atomique bien typée de FOC*, héritant d'une espèce atomique s_1. Alors, $\mathcal{C}(s)$ est un sous-type de $\mathcal{C}(s_1)$: Si x est présent avec le type τ dans s_1, il est présent avec le même type (modulo alpha-équivalence) dans s.*

Preuve. Immédiat avec le théorème précédent et la règle WELL-TYPED-SPEC. □

5.3 Les paramètres

Si on considère maintenant les espèces paramétrées, la principale différence avec la section précédente provient des nouvelles variables et contraintes de type qu'on utilise pour la traduction des paramètres de collection. En effet, les paramètres d'entité ne font qu'ajouter une variable dans l'environnement, tout comme leur traduction OCAML sous forme d'abstraction. En ce qui concerne les paramètres de collection, les variables sont ajoutées dans l'environnement OCAML avec comme type $i(s)$, variable de type portant une contrainte. Il convient donc de vérifier que les contraintes que l'on fournit lors de la traduction sont suffisantes pour que OCAML puisse effectuer le typage du corps des classes.

Toutefois, avant cela, on va vérifier que les contraintes elle-mêmes sont cohérentes entre elles. En effet, un paramètre de collection peut dépendre des précédents. Dans ce cas, la contrainte de type engendrée doit être compatible avec le reste de l'environnement. Informellement, cela revient à vérifier que si un paramètre c_q est utilisé dans le type d'un paramètre c_p d'interface I_p

[1] En admettant la correction du compilateur OCAML lui même.

(avec $q < p$), alors la contrainte engendrée pour c_q est compatible avec les contraintes que la classe $\mathcal{C}(I_p)$ fait peser sur ses propres paramètres.

Lemme 29 (Contraintes de type des paramètres)
*Soit s une espèce bien typée, et $c_1 \cdot I_1, ..., c_{p_f} \cdot I_{p_f}$ la liste de ses paramètres. Soit c_p un paramètre de collection tel que $I_p = s'(e_1, ..., e_n)$. Alors $\mathbb{C}(c_p$ **is** $s'(e_1, ..., e_n))$ est compatible avec les contraintes des paramètres de $\mathcal{C}(s)$ et les contraintes des c_q $(q < p)$.*

De même, si s herite d'une espèce paramétrée $s_1(a_1, ..., a_m)$, alors les contraintes sur les paramètres de $\mathcal{C}(s)$ sont compatibles avec les instantiations des paramètres de s_1.

Preuve. Notons qu'on peut se ramener au cas où tous les e_i et les a_i sont des collections ou des paramètres de collections, les instanciations des paramètres d'entité n'ayant pas d'influence sur les contraintes de type.

Posons

$$\mathbb{C}(c_p \textbf{ is } s'(e_1, ..., e_n)) = \textbf{constraint } \alpha = (\beta, \beta_1, \alpha_1, \ldots, \beta_n, \alpha_n) \# \mathcal{C}^t(s)$$

et $\mathcal{E}(s') = (s_1 \textbf{ is } J_1, ..., s_n \textbf{ is } J_n)\Phi$. Il faut vérifier qu'il y a assez d'information de type pour vérifier les contraintes présentes dans $\mathcal{C}^t(s')$ liant les β_i et les α_i. Pour cela, il faut distinguer deux cas, suivant que e_i est une collection ou un des paramètres c_q précédant c_p.

Si e_i est une collection préexistante, on a $e_i \twoheadrightarrow Mod(e_i).obj$, qui est de type $\langle \mathcal{C}(e_i) \rangle$. De plus, d'après la règle Coll-inst, e_i implante l'interface a_i, donc $\langle \mathcal{C}(e_i) \rangle$ est bien un sous type de $\mathcal{C}^t(b_i)$, et la contrainte est satisfaite.

Si e_i est un des paramètres précédents, c_q, d'après la règle Coll-inst, I_q est une sous-espèce de b_i, et la contrainte engendrée par c_q permet de vérifier la contrainte sur b_i.

Pour montrer que les contraintes sur les paramètres de type de $\mathcal{C}(s_1)$ sont satisfaites, on peut suivre la même démarche. La règle Prm-inherit impose en effet que les espèces dont on hérite aient leurs paramètres complètement instanciés, ce qui implique que tous les a_i soient typés par la règle Prm-inst. □

Une fois montré que les contraintes données par la traduction sont correctes, il reste à s'intéresser au corps de l'espèce lui-même. Pour cela, on va comme dans la section précédente s'appuyer sur le lemme 26 pour montrer que la traduction du corps des méthodes définies est typable en Ocaml. Par rapport au cas précédent, une difficulté supplémentaire vient du fait qu'il faut faire la correspondance entre les variables de type utilisées dans l'espèce et les types de classe correspondant, qui sont ceux qu'on utilise dans la traduction du contexte.

Théoreme 7 (Méthodes d'une espèce paramétrée) *Soit s une espèce* Foc *bien typée dans le contexte $\mathcal{C}, \mathcal{E}, \Gamma$ et $c_1 \cdot I_1, ..., c_k \cdot I_k$ la liste de ses*

paramètres. Posons $s \twoheadrightarrow^t \sigma$, $s \twoheadrightarrow c$ *et* $\mathcal{C}, \mathcal{E}, \Gamma \twoheadrightarrow \Gamma'$. c *est une définition de classe bien typée, et*

$$\Gamma' \vdash c : \tau_1 \to \ldots \to \tau_k \to \sigma$$

τ_i *étant défini de la manière suivante :*

- *Si le ième paramètre de* s *est de la forme* c_i ***in*** I_i *alors* $I_i \twoheadrightarrow \tau_i$
- *Si le ième paramètre de* s *est de la forme* c_i ***is*** I_i *alors* $\tau_i = \sigma(c_i)$

Preuve. Par induction sur le nombre de paramètres de s. Dans le cas où il n'y a pas de paramètre, on applique directement le théorème 6. Sinon, On a deux cas possibles suivant que le premier paramètre est un paramètre d'entité ou un paramètre de collection.

Si le premier paramètre est de la forme c_1 **in** τ_1, d'après la règle ENT-PRM, le reste de l'espèce est typée dans l'environnement $\Gamma + x : \tau$, les autres composantes restant inchangées. Avec la définition de $\twoheadrightarrow$, c a bien un type fonctionnel, et le type de son premier argument est bien τ' traduction de τ. On peut donc conclure avec l'hypothèse d'induction.

Si le premier paramètre de s est de la forme c_1 **is** I_1, d'après la règle COLL-PRM, le reste de l'espèce est typé en ajoutant à l'environnement $\mathcal{C}$ $c : \mathcal{A}(c, inst)$, où *inst* est l'interface de I_1. Dans la traduction de ce nouvel environnement, c va prendre le type $\sigma(c)$, un des paramètres de type de l'espèce. Or on ne sait typer le corps des méthodes OCAML que dans un contexte où c a le type objet $\langle inst \rangle$. Toutefois, $\sigma(c)$ porte une contrainte, par définition de $\twoheadrightarrow$, imposant que $\sigma(c)$ soit de la forme $\langle inst, \rho \rangle$. Cette contrainte permet donc de se placer dans le bon environnement Γ' pour continuer le typage de l'espèce. On peut donc conclure avec l'hypothèse d'induction. □

Comme dans la section précédente, on peut en déduire que les clauses **inherits** de la traduction OCAML sont correctes.

Corollaire 4 (Traduction de l'héritage avec des paramètres)
Soit s *une espèce bien typée de* FOC, *héritant d'une espèce* s_1. *Alors,* $\mathcal{C}(s)$ *est un sous-type de* $\mathcal{C}(s_1)$

Preuve. D'après le lemme 29, les instantiations des éventuels paramètres de s_1 sont corrects dans la traduction de s_2. Pour le types des méthodes, le théorème 7 et la règle WELL-TYPED-SPEC permettent de conclure. □

5.4 Les collections

Il reste à vérifier qu'une collection c implantant une espèce s est bien traduite en un module valide de OCAML, et que la signature qu'on impose à ce module est bien correcte. Pour la signature, il suffit de vérifier que toutes les définitions de type sont correctes, y compris en présence du type *abstrait rep*. En ce qui concerne le module lui-même, le point le plus important consiste

à montrer qu'on peut effectivement créer une nouvelle instance de la classe $\mathcal{C}(s)$, c'est à dire que cette dernière n'est pas virtuelle.

Théoreme 8 (Traduction des collections)
Soit c une collection FOC *implantant l'espèce* $s(e_1, ..., e_n)$. *Posons* I *et* M *une signature et un module tels que* $c \twoheadrightarrow$ ***module*** $Mod(c) : I = M$, *alors* I *est une signature de module bien typée en* OCAML, *et* M *en est une implantation bien typée.*

Preuve. Par induction sur le nombre de paramètres de s. Si s est une espèce atomique, I se compose de trois champs : un type abstrait *rep*, un type objet *stype*, et une valeur de type *stype*. De plus, *stype* est défini comme implantant $\mathcal{C}^t(s)$ en instanciant son premier paramètre par *rep*. Or ce premier paramètre ne porte par définition de $\twoheadrightarrow^t$ pas de contrainte, et peut donc être instancié par n'importe quel type (et en particulier par un type abstrait).

En ce qui concerne M, il implante les champs déclarés de I. De plus, avec la règle COLL, s est complètement défini, donc d'après le lemme 28, $\mathcal{C}(s)$ est complètement définie, et il est possible d'en créer une nouvelle instance. De plus d'après le théorème 6, cette instance peut être coercée vers $\mathcal{C}^t(s)$, dont *stype* est un synonyme : l'implantation M est bien conforme à la spécification I.

Dans le cas où s possède des paramètres, on procède par cas sur le premier paramètre. Si c'est un paramètre d'entité, il n'y a pas de champ supplémentaire, et la traduction de e_1 est bien typée. On peut donc conclure comme ci-dessus. Si on est en présence d'un paramètre de collection, e_1 est un nom de collection préexistante. De plus, on ajoute deux nouveaux types dans I comme dans M. D'après la règle COLL-INST, e_1 est un paramètre admissible pour s, donc sa traduction est typable dans l'environnement du module. Par ailleurs, les nouveaux types étant définis comme synonyme de $Mod(e_1).rep$ et $Mod(e_1).stype$, ils respectent les contraintes des paramètres de type de $\mathcal{C}(s)$. On peut alors conclure avec l'hypothèse d'induction. □

Chapitre 6

Compilation vers Coq : enregistrements et générateurs de méthodes

6.1 Présentation de Coq

Coq est un système d'aide à la preuve, basé sur le calcul des constructions inductives [Coq02]. Celui ci est une extension du noyau purement fonctionnel de Ocaml. Il était donc assez naturel de choisir Coq comme langage cible pour la partie spécification et certification de Foc. Dans cette section, mous allons présenter brièvement les principales constructions de Coq, du moins celles qui servent lors de la traduction de Foc.

6.1.1 *Set* et les calculs

À quelques détails syntaxiques près, les définitions de fonctions en Coq sont assez similaires à celles de Ocaml :

Definition `notb` : `bool` → `bool` :=
  [*a* :`bool`] `if` *a* `then false else true`.

Toutefois, les fonctions polymorphes ne s'expriment pas de la même manière : la variable de type est en fait un argument à part entière de la fonction qu'on veut définir. Le type de cet argument est *Set*, le "type" de tous les types ayant un contenu calculatoire, comme `bool`,`nat` (les entiers de Peano),... Par exemple, on définira l'identité polymorphe de cette manière :

Definition `id` : (`A` :**Set**)`A`->`A` :=[`A` :**Set**][`x` :`A`]`x`.

Le type de cette fonction est un type dépendant : le type du second argument de la fonction ainsi que celui du résultat dépendent du premier argument, `A`. On peut noter que lors de l'application d'une telle fonction, Coq est souvent

capable d'inférer la valeur du premier argument en fonction du second. Dans ce cas, l'utilisateur peut remplacer ce premier argument par un `?`. Ainsi, (`id ? true`) est une expression Coq correcte (qui s'évalue en `true`).

6.1.2 *Prop*, les propriétés, les preuves

Ce qu'on vient de décrire sur la partie calculatoire de Coq se retrouve au niveau des propositions logiques, termes de types `Prop`. Cette correspondance est connue sous le nom d'*isomorphisme de Curry-Howard* [How80]. Dans ce contexte, une preuve d'une proposition n'est autre qu'un terme ayant comme type la proposition. Ainsi, si on définit la propriété suivante :

Definition `tautologie` : **Prop** := (`A` :**Prop**)`A->A`.

dont on peut remarquer qu'elle ressemble beaucoup au type de `id`, on va prouver que cette proposition est vraie en exhibant un terme ayant ce type :

Definition `preuve` : `tautologie` := [`A` :**Prop**][`x` :`A`]`x`.

Ces constructions permettent de définir une partie des constructeurs logiques habituels : l'implication comme on vient de le voir, et la quantification universelle à l'aide d'un produit dépendant. Toutefois, les autres constructeurs nécessitent l'introduction d'une autre catégorie de constructions de Coq, les *types inductifs*, présentés dans la section suivante.

Auparavant, revenons sur *Set* et *Prop* : nous nous en sommes servis jusqu'à présent en guise de types, mais il s'agit aussi de termes Coq et à ce titre ils doivent avoir aussi un type. Il s'agit de $\mathbf{type}_0$, qui lui-même à le type $\mathbf{type}_1$. Par ailleurs, tout élément de type $\mathbf{type}_i$ peut être vu comme étant de type $\mathbf{type}_{i+1}$. L'indice i est le *niveau* du type. Lorsqu'on manipule des termes appartenant à des niveaux différents, on parle de *hiérarchie d'univers*. En pratique, on ne spécifie jamais le niveau où on se trouve dans la hiérarchie d'univers, et c'est le système qui se charge de gérer les contraintes nécessaires à la cohérence de l'ensemble.

6.1.3 Types inductifs

A première vue, les types inductifs de Coq jouent le même rôle que les types concrets de Ocaml, si ce n'est que les constructeurs de ces types peuvent avoir un type dépendant. Ainsi, on peut facilement définir le type des listes sur un type A :

Inductive `list` : (`A` :**Set**)**Set** :=
  `Nil` : (`A` :**Set**)(`list A`)
| `Cons` : (`A` :**Set**)`A->`(`list A`)`→`(`list A`).

Notons que contrairement à Ocaml on précise le type complet des constructeurs, y compris donc leur type de retour. En effet, ce type peut dépendre

des paramètres du constructeur. Ainsi, on peut définir le `et` logique de la manière suivante :

```
Inductive et : Prop → Prop → Prop :=
  intro : (A,B :Prop)A → B → (et A B).
```

Le constructeur `intro` compte 4 arguments : deux propositions A et B, une preuve que A est vraie (c'est à dire, d'après l'isomorphisme de Curry-Howard, un élément de A), et une preuve que B est vraie. Il renvoie un élément –une preuve– de (`et A B`). Lorsqu'on évalue cette définition, on peut remarquer que COQ ne se contente pas de définir *et*, mais qu'il ajoute une fonction, *et_ind*. Cette fonction est un *schéma d'élimination* du type inductif, qui permettent de déstructurer un élément du type *et*, comme le ferait un pattern-matching en OCAML. Si on examine le type de *et_ind* on trouve la forme suivante :

```
et_ind :(Φ :(Prop->Prop->Prop))
        ((A,B :Prop)A->B->(P A B))→(A,B :Prop)(et A B)→(Φ A B)
```

Les arguments de cette fonction sont donc les suivants :
- un prédicat prenant en argument deux propositions
- une preuve que pour toutes les propositions A et B, si A est vrai et B est vrai alors $(\Phi\ A\ B)$ est vraie
- deux propositions A et B
- une preuve de $(et\ A\ B)$

et renvoie alors un élément (une preuve) de $(\Phi\ A\ B)$.

On peut remarquer qu'à partir d'un tel prédicat il est possible de retrouver les règles d'élimination du *et* en déduction naturelle, en utilisant des projections pour instantier Φ (Dans le code ci-dessous, `_` désigne un paramètre qui n'est pas utilisé dans le corps de la fonction).

```
Definition elim1 : (A,B :Prop)(et A B)->A :=
    (et_ind [A;_]A [A;_;pA;_]pA).
Definition elim2 : (A,B :Prop)(et A B)->B :=
    (et_ind [_;B]B [_;B;_;pB]pB).
```

Enfin, on peut définir un prédicat par le biais d'un type inductif. Ainsi l'égalité de COQ est définie sous forme de type inductif, comme le plus petit prédicat réflexif :

```
Inductive eq : (A :Set)A->A->Prop :=
  refl : (A :Set)(x :A)(eq ? x x).
```

"Plus petit" est ici à entendre comme la construction d'un plus petit point fixe. En fait, si on considère les prédicats binaires comme équivalents à un sous-ensemble de `A*A`, `eq` est le plus petit prédicat contenant les couples (`x`,`x`).

Le schéma d'élimination correspondant, `eq_ind` correspond à la définition de l'égalité de Leibniz :

```
eq_ind
     : (A :Set ; x :A ; P :(A->Prop))(P x)→(y :A)x=y->(P y)
```

Si x et y sont égaux, tout prédicat P vérifié par x est vérifié par y.

6.1.4 Enregistrements et coercions

Au delà du noyau que représente le calcul des constructions inductives, COQ fournit un certain nombre d'extensions qui facilitent l'écriture de programmes. Dans cette section et dans la suivante, on détaille les extensions les plus importantes pour FOC : les enregistrements avec champs dépendants, les coercions implicites et le mécanisme de section.

Commençons par décrire les enregistrements. Syntaxiquement, il s'agit d'une liste de noms de champs accompagnés de leur type, comme en OCAML. Toutefois, ici l'ordre des champs est très important, à cause des types dépendants : le type d'un champ peut dépendre des champs précédents, comme dans l'exemple suivant :

```
Record et_enr : Type :=
  mk_enr { A : Prop ; B : Prop ; pA : A ; pB : B }.
```

Ici, `pA` et `pB` sont des preuves respectivement de `A` et de `B`, les deux premiers champs de l'enregistrement.

Techniquement, `et_enr` n'est autre qu'un type inductif muni d'un seul constructeur, et relativement proche de notre type `et` de la section précédente. Toutefois, COQ va définir en plus du type inductif proprement dit, et de son constructeur `mk_enr`, des fonctions de projections correspondant aux différentes composantes de l'enregistrement : `A : et_enr->Prop`, `pA :(E :et_enr)(A E)`, etc. De ce fait, les noms des différents types d'enregistrement doivent être distincts deux à deux, puisqu'ils correspondent à des définitions COQ. Notons enfin que le type de l'enregistrement lui même est de type *Type*, dans la mesure où certaines de ses composantes sont de type *Prop*.

Un autre point important du point de vue de FOC est la possibilité de définir des *coercions* entre les différents types d'enregistrements. Par exemple, si on définit les deux enregistrements permettant de définir des points et des points colorés :

```
Record point : Set := mk_pt { xcoord : Z ; ycoord : Z}.
Record point_colore : Set :=
                  mk_ptc {xcoord1 : Z ; ycoord1 : Z ; color : nat}.
Definition ptc_pt :=
   [x :point_colore] (mk_pt (xcoord1 x) (ycoord1 x)).
Coercion ptc_pt : point_colore >-> point.
```

La fonction `ptc_pt` transforme un point coloré en un point "normal". La dernière ligne la déclare comme coercion, ce qui permet d'utiliser dans la suite un point coloré partout où Coq s'attend à trouver un point. Le système se chargera alors d'insérer un appel à `ptc_pt`.

6.1.5 Chapitres et sections

Il reste à présenter le mécanisme de *section* utilisé en Coq pour rassembler différentes définitions ayant des liens entre elles. À l'intérieur d'une section (ou d'un chapitre, les deux mot-clés **Section** et **Chapter** étant synonymes), il est possible de déclarer des *variables*, de n'importe quel type, et d'y faire référence dans les différentes définitions qu'on donne dans la section. De même, on peut faire des définitions *locales* qui ne sont visibles qu'au sein de la section où elles sont définies. Lorsqu'on ferme la section, Coq fait les abstractions et les expansions de variables locales nécessaires à la bonne formation des définitions globales de la section. Ainsi, on peut définir le *et* logique et ses projections au sein d'une section afin de factoriser une partie du code :

```
Section conjonction.
Variable A,B :Prop.
Inductive et : Prop :=
  intro : A → B → et.

Definition elim1 : et->A := [H :et](et_ind A [pA;_]pA H).
Definition elim2 : et->B := [H :et](et_ind B [_ ;pB]pB H).
End conjonction.
```

Dans la définition de `elim1` et `elim2`, on peut utiliser les deux variables `A` et `B`. Toutefois, si on regarde le type de `elim1` une fois la section refermée, `elim1 : (A,B :Prop)(et A B)->A`,on constate qu'elle prend en paramètre les deux propositions `A` et `B`, qui étaient des variables dans la section `conjonction`.

6.2 Description de la traduction en Coq

Nous allons maintenant voir comment utiliser les différentes constructions de Coq pour modéliser les espèces et collections de Foc. Les interfaces sont facilement représentées par des types d'enregistrement, et les collections par des enregistrements. Un problème se pose par contre pour les espèces. Idéalement, il faudrait pouvoir créer un enregistrement "à trous", où certains champs restent abstraits.

Le mécanisme de section semble à même de fournir cela. En représentant chaque espèce Foc par une section, on peut traduire les champs abstraits

par des variables et les champs concrets par des définitions locales. En fin de section, on pourra alors créer un enregistrement rassemblant tous les champs, et le mécanisme d'abstraction permettra de faire les abstractions correspondant aux méthodes déclarées. Si cette vision est intéressante au premier abord, elle est cependant insuffisante pour capturer toutes les constructions de FOC. En effet, si ce mécanisme permet de définir facilement ce qui est l'équivalent d'une espèce en forme normale, il est impuissant à modéliser la liaison retardée et la redéfinition de méthodes lors de l'héritage : une méthode m_1 qui decl-dépend d'une méthode m_2 définie au sein d'une espèce s utilise la définition locale correspondante de la section représentant s. Si dans une espèce s' héritant de s on redéfinit m_2 sans toucher à m_1, il n'y a pas moyen de réutiliser le code précédent de m_1 : il faut tout réécrire dans une nouvelle section. En effet, le mécanisme d'abstraction ne porte que sur les variables de la section. Or si m_2 est définie, elle sera représentée en COQ par une définition locale. La définition de m_1 dans s fera donc référence à cette définition précise, ce qui va à l'encontre du but recherché. On veut en effet pouvoir réutiliser cette définition de m_1 dans les espèces héritant de s, mais en utilisant la nouvelle définition de m_2.

Pour remédier à cela, nous n'allons pas construire directement les définitions locales, mais passer par l'intermédiaire de définitions globales, appelées *générateurs de méthodes*. Pour construire un tel générateur, on abstrait dans le corps d'une méthode m les noms des méthodes dont m decl-dépend. Si on prend l'exemple de l'espèce `s` ci-dessous,

```
species s =
 sig eq in self-> self->bool;
 let neq = fun x → fun y → notb(self !eq(x,y));
```

Le générateur de méthode correspondant à `neq` sera le suivant (dans lequel `bool__t` représente la traduction du type `bool` de FOC) :

```
[abst_T :Set][abst_eq :abst_T->abst_T->bool__t]
   [x,y :abst_T](notb (abst_eq x y))
```

La première ligne correspond aux abstractions des decl-dépendances, tandis que la seconde ligne est la traduction du corps de la méthode. Ce corps sera évalué *dans l'environnement mis en place par les abstractions précédentes*. En particulier, l'appel de méthode `self !eq` est remplacé par la variable `abst_eq`, second argument de la fonction.

On peut dès lors utiliser ce générateur pour créer la méthode `neq` de `a`, mais aussi de toutes les espèces héritant de `a`, tant que `neq` elle-même n'est pas redéfinie : il suffira de l'appliquer à l'instance courante de la méthode `eq` (et au type support), ce qui correspond bien à la liaison retardée.

Notons qu'on ne peut appliquer le même système aux def-dépendances, puisque par définition si m_1 def-dépend de m_2, on a besoin du corps de m_2 pour typer m_1, et on ne peut donc pas faire d'abstraction.

6.3 Un exemple complet de traduction

Comme dans le chapitre précédent, on s'appuie sur l'exemple des produits cartésiens (voir 2.2 pour le code FOC correspondant) pour présenter plus en détail les différents points de la traduction de FOC vers COQ. Dans la section suivante on présentera une vue plus formelle de cette même traduction. Dans l'exemple, on trouve en premier lieu le **chapter** représentant l'espèce des *setoides*.

Chapter `Setoide`.

Ce **chapter** peut se découper en trois parties :
- Définition du type d'enregistrement des `setoides`.
- Déclaration des méthodes abstraites, définition des générateurs de méthodes et des liaisons locales.
- Définition d'un enregistrement de type setoide.

Nous allons maintenant détailler ces différentes parties. Commençons par le type d'enregistrement, qui correspond à la notion d'interface en FOC :

```
Record setoide : Type :=mk_setoide {
  setoide_T :> Set ;
  setoide_print : (setoide_T)→ string__t ;
  setoide_parse : (string__t)→ setoide_T ;
  setoide_egal : (setoide_T)→ (setoide_T)→ bool__t ;
  setoide_element : setoide_T ;
  setoide_different : (setoide_T)→ (setoide_T)→ bool__t ;
  setoide_refl : (x : setoide_T) (Is_true (setoide_egal x x)) ;
  setoide_symm : (x : setoide_T) (y : setoide_T)
    ((Is_true (setoide_egal x y)))→
        (Is_true (setoide_egal y x))
}.
```

On peut tout d'abord remarquer que ce type vit dans `Type`. En effet, le champ `setoide_T`, qui correspond au type support est un élément de `Set`, et `setoide` doit donc se situer au dessus de `Set` dans la hiérarchie des types de COQ. Par ailleurs, le champ `setoide_T` est défini implicitement comme une coercion de `setoide` dans `Set`, ce qui permet d'alléger les notations lorsqu'on veut parler d'un élément d'un setoide : au lieu de la formule `(S :setoide)(x :(setoide_T S))`...on pourra employer directement `(S :setoide)(x :S)`....

Enfin, il convient de noter que cet enregistrement reprend *tous* les champs de la forme normale de `setoide`, et non pas seulement ceux qui sont déclarés ou définis dans le corps de l'espèce. Par exemple, les champs `parse` et `print` viennent de `basic_object` (voir la section 4.1.1). En effet, `setoide` est pour l'instant complètement indépendant du type d'enregistrement `basic_object`

et doit donc déclarer seul l'ensemble de ses champs. Ce n'est qu'à la fin de la section qu'on définira les liens entre `setoide` et `basic_object` à l'aide d'une coercion (voir ci-dessous).

Après la définition du type d'enregistrement, on trouve une série de déclarations de variables, définitions locales et définitions globales représentant les différentes méthodes de `setoide` :

Variable `self_T` : **Set**.

hérité
Local `self_print` : (`self_T`)→ `string__t` :=
(`basic_object__print self_T`).
Local `self_parse` : (`string__t`)→ `self_T` :=
(`basic_object__parse self_T`).

déclaré
Variable `self_egal` : (`self_T`)→ (`self_T`)→ `bool__t`.
Variable `self_element` : `self_T`.

défini
Definition `setoide__different` :
(`abst_T` : **Set**) (`abst_egal` : (`abst_T`)→ (`abst_T`)→ `bool__t`)
(`abst_T`)→(`abst_T`)->`bool__t` :=
[`abst_T` : **Set**] [`abst_egal` : (`abst_T`)→ (`abst_T`)→ `bool__t`]
[`x` : `abst_T`][`y` : `abst_T`](`not_b` (`abst_egal x y`)).
Local `self_different` : (`self_T`)→ (`self_T`)→ `bool__t` :=
(`setoide__different self_T self_egal`).

déclaré
Hypothesis `self_refl` : (`x` : `self_T`)(`Is_true` (`self_egal x x`)).

Hypothesis `self_symm` : (`x` : `self_T`) (`y` : `self_T`)
((`Is_true` (`self_egal x y`)))→(`Is_true` (`self_egal y x`)).

Là encore, on retrouve toutes les méthodes de la forme normale, qu'elles soient héritées ou non, dans l'ordre défini par les dépendances. Par ailleurs, on peut distinguer trois cas :

- Déclaration de variable (ou d'hypothèse, les deux mots clés étant synonymes en Coq).
- Une définition locale.
- Une définition globale et une définition locale pour la même méthode

Comme on l'a déjà signalé, les variables correspondent aux méthodes de la forme normale de `setoide` qui sont seulement déclarées. Les définitions locales sont les traductions des méthodes définies. Elles sont données comme l'application d'un générateur de méthode aux définitions et/ou déclarations précédentes, suivant les decl-dépendances. Ainsi, `self_print` est définie en

utilisant le générateur pour `print` défini dans le chapitre de `basic_object` (`print` n'est pas redéfinie dans `setoide`), appliqué à `self_T`, type support de `setoide`.

Enfin, les définitions globales correspondent aux générateurs de méthodes. Comme on l'a dit, seules les méthodes définies dans le corps de l'espèce (c'est à dire ici la méthode `different`) donnent lieu à la définition d'un générateur de méthode. Pour les méthodes héritées, comme ici `print` et `parse`, il suffit de réutiliser le générateur correspondant. De ce fait, les générateurs de méthode doivent être des définitions globales et non locales.

La dernière partie de la section correspond à la création d'un setoide "générique" à partir de toutes ses composantes. Celles d'entre elles qui se présentent sous forme de variable (`T`, `egal`, `element`, `refl` et `symm`) seront abstraites lors de la fermeture de la section, faisant de `new_setoid` une fonction qui si on lui fournit une définition pour chacune des méthodes encore abstraites renverra un `setoide`.

```
Definition new_setoide :=
  (mk_setoide self_T self_print self_parse self_egal
              self_element self_different self_refl self_symm).
End Setoide.
```

En dehors de la section, il reste à établir pour Coq le lien entre `setoide` et `basic_object`. Pour cela, on définit la fonction qui extrait les champs présents dans `basic_object` d'un `setoide` quelconque, et on la marque explicitement comme `coercion`.

```
Definition setoide_basic_object :=
[S :setoide](mk_basic_object
 (setoide_T S) (setoide_print S) (setoide_parse S) ).
Coercion setoide_basic_object : setoide>->basic_object.
```

Nous disposons maintenant de tous les éléments permettant de représenter l'espèce `setoide` en Coq. Passons à la traduction de l'espèce `monoide`. Celle-ci s'effectue sur le même modèle que `setoide`, puisqu'il s'agit ici aussi d'un héritage simple. On ne donnera donc pas la traduction complète, mais seulement quelques points sur lesquels il convient d'insister.

```
Chapter Monoide.
  Record monoide : Type :=mk_monoide
  { monoide_T :> Set ;
    (* . . . *)
    monoide_different : (monoide_T)→ (monoide_T)→ bool__t ;
    (* . . . *)
    monoide_multiplie : (monoide_T)→ (monoide_T)→ monoide_T ;
    monoide_un : monoide_T ;
    monoide_element : monoide_T
  }.
```

Comme on peut le voir ici, l'ordre entre les différents champs d'un type d'enregistrement peut varier lors des héritages : `element` qui était placé avant `different` dans `setoide` l'est maintenant après. En effet, c'est l'ordre défini par la mise en forme normale qui sert à définir le type d'enregistrement, et cet ordre peut varier suivant les (re)définitions lors du passage d'une espèce à une autre. Ici, `element` étant défini en fonction de `un`, il doit être placé après le champ correspondant. Notons toutefois que lorsqu'on fait des coercions, ces dernières tiennent bien sûr compte de l'ordre de la forme normale de l'espèce parent, comme on le verra ci-dessous.

Dans les définitions de méthodes, il n'y a cette fois pas besoin de donner le générateur de `different` : on fait appel à celui de `setoide`. Par contre `element`, qui n'était que déclaré dans l'espèce précédente est maintenant défini, et se voit doté du générateur correspondant :

```
...
 Local self_different : (self_T)→ (self_T)→ bool__t :=
 (setoide__different self_T self_egal ).

 Definition monoide__element :
  (abst_T : Set)
  (abst_multiplie : (abst_T)→ (abst_T)→ abst_T)
  (abst_un : abst_T) abst_T :=
  [abst_T : Set]
  [abst_multiplie :(abst_T)→ (abst_T)→ abst_T]
  [ abst_un : abst_T]
    (abst_multiplie abst_un abst_un).

 Local self_element : self_T :=
   (monoide__element self_T self_multiplie self_un ).
...
```

Enfin, on termine comme précédemment en définissant `new_monoide`, et en créant une coercion entre `monoide` et `setoide`.

```
Definition new_monoide :=
  (mk_monoide self_T self_print self_parse self_egal
            self_different self_refl self_symm
            self_multiplie self_un self_element).
End Monoide.

Definition monoide_setoide :=
[S :monoide](mk_setoide
 (monoide_T S) (*. . .*)
 (monoide_element S) (monoide_different S)
 (monoide_symm S)).
Coercion monoide_setoide : monoide>->setoide.
```

Lors de la définition de la coercion, on passe bien à `mk_setoide` les différents champs de `S`, dans l'ordre de la forme normale de `setoide`. De plus, c'est bien la définition de `different` dans `S` qui sera utilisée dans le `setoide` ainsi défini, et non la définition existant dans `setoide`. Les coercions ne modifient pas le mécanisme de liaison tardive qu'on cherche à implanter en Coq.

L'exemple des produits de setoide permet d'introduire la traduction des espèces paramétrées :

```
Chapter Setoide_produit.
  Record setoide_produit [a : setoide ; b : setoide] : Type :=
    mk_setoide_produit {
      setoide_produit_T :> Set ;
      (* . . . *)
      setoide_produit_creer : (a) → (b) → setoide_produit_T ;
      (* . . . *)
    }
```

Les deux paramètres de l'espèce se retrouvent également en paramètre du type d'enregistrement qu'on définit. En effet, les types des différentes méthodes dépendent de a et b. Il faut donc disposer dans l'environnement des deux setoides a et b lorsqu'on déclare le type de `creer`.

Comme précédemment, on poursuit avec les déclarations et définitions des méthodes et des générateurs. Les deux paramètres sont vus comme deux variables (`prod` est le constructeur de produit cartésien en Coq. C'est l'équivalent de `et` pour les éléments de **Set**) :

```
Variable self_a : setoide.
Variable self_b : setoide.

Local self_T : Set := (prod self_a self_b).

Definition setoide_produit__egal :
     (self_T)→ (self_T)→ bool__t :=
 [x :self_T][y : self_T]
    (and_b ((setoide_egal self_a) (first x) (first y))
           ((setoide_egal self_b) (scnd x) (scnd y))).
```

Les deux setoides `self_a` et `self_b` sont utilisés tels quels : contrairement au cas des decl-dépendances, il n'y a pas de lambda-lifting. En effet, les paramètres sont par définition toujours abstraits (c'est à dire présents sous forme de variables dans le chapitre), contrairement aux méthodes, qui peuvent être des variables ou des définitions locales. Pour les paramètres, on peut donc s'appuyer entièrement sur le mécanisme de section de Coq, qui fera de `egal` une fonction de `self_a` et `self_b`. Par ailleurs, on peut remarquer qu'à cause

de la def-dépendance de `egal` vis à vis du type support, ce dernier n'est pas abstrait dans le générateur `setoide_produit__egal`.

Voyons un autre point. Dans l'espèce `setoide_produit` apparaissent des *théorèmes*. Comme on peut le voir ci-dessous avec l'exemple de `refl`, ces méthodes sont traduites dans leur propre section :

```
Section setoide_produit__refl.
  Local abst_T : Set :=
    (self_T       ).

  Local abst_egal : (abst_T)→ (abst_T)→ bool__t :=
    (setoide_produit__egal    ).

  Definition setoide_produit__refl :
    (x : abst_T)(Is_true (abst_egal x x)) :=
      (magic_prove (x : abst_T) (Is_true (abst_egal x x)) __Nil).
End setoide_produit__refl.

Local self_refl : (x : self_T)(Is_true (self_egal x x)) :=
                                        (setoide_produit__refl).
```

Au sein de cette section, on trouve tout d'abord des définitions locales, correspondant au def-dépendances du théorème `refl`. Comme on le verra ci-dessous, d'éventuelles decl-dépendances seraient bien entendu traduites par des variables. On trouve ensuite une définition locale, ici réduite à l'application de `magic_prove` qui trahit l'emploi du mot-clé **assumed** dans le source FOC. `magic_prove` prend en argument, outre l'énoncé de `setoid_produit_refl`, une liste (ici vide) des éventuelles decl-dépendances signalées par l'utilisateur et non reprise dans l'énoncé, afin de s'assurer que COQ fera correctement les abstractions lorsqu'on fermera la section.

Enfin, on rassemble le tout dans une définition globale qui constitue le générateur de méthode du théorème[1]. Une fois sorti de la section, on utilise comme précédemment ce générateur de méthode pour créer la définition locale liée aux définitions courantes de l'espèce.

Le reste du **chapter** consacré aux produits de setoides se poursuit comme les précédents, jusqu'à la définition de l'enregistrement à partir des différentes définitions locales (et des deux variables correspondant aux setoides).

La définition de la coercion avec l'espèce parent `setoide` est toutefois légèrement différente du cas d'une espèce atomique. En effet, cette coercion

[1] Cela est déjà fait pour quelques theorèmes.

est paramétrée par les deux setoides a et b dont on veut faire le produit :

```
Definition setoide_produit_setoide :=
  [a : setoide ; b : setoide] [S : (setoide_produit a b)]
    (mk_setoide (setoide_produit_T ? ? S) (* . . *) ).
Coercion setoide_produit_setoide :setoide_produit >-> setoide.
```

Il reste à regarder la traduction de l'espèce des produits cartésiens de monoides. On va ainsi voir la gestion de l'héritage des espèces paramétrées, ainsi que de l'héritage multiple en Coq :

```
Chapter Monoide_produit.
  Record monoide_produit[a : monoide ; b : monoide] :Type :=
  (* . . . *)

  Variable self_a : monoide.
  Variable self_b : monoide.
  Local self_egal : (self_T)→ (self_T)→ bool__t :=
  (setoide_produit__egal self_a self_b).
  (* . . . *)
```

Cette fois, les deux variables `self_a` et `self_b` sont des monoides et non plus des setoides. Ainsi, lorsqu'on cherche à créer la méthode `egal` à partir du générateur venant de `setoide_produit`, Coq doit insérer la coercion de `monoide` vers `setoide`. En effet, `setoide_produit__egal` attend comme premiers paramètres deux setoides, les paramètres de `setoide_produit`.

On peut également observer qu'un générateur de méthode peut effectivement être réutilisé tout au long de l'arbre d'héritage : ici, `different` est créé à partir du générateur de `setoide`, qui n'est pas un parent direct de `monoide_produit`.

```
  Local self_different : (self_T)→ (self_T)→ bool__t :=
  (setoide__different self_T self_egal).
```

Enfin, les choix de définition en cas de conflit se retrouvent dans le choix du générateur qui produit `element`. En effet, cette méthode est définie dans les deux parents de l'espèce courante, `monoide` et `setoide_produit`. Comme on l'a vu, c'est la définition de l'espèce mentionnée en dernier qui est sélectionnée, c'est à dire ici le générateur de `setoide_produit`.

```
  Local self_element : self_T :=
  (setoide_produit__element self_a self_b self_T self_creer).
  . . .
End Monoide_produit.
```

Après la fermeture du **Chapter**, on définit comme à l'accoutumée les coercions avec les espèces parents. Il y en a cette fois-ci deux, une pour chaque espèce parent.

```
Definition monoide_produit_monoide :=
  [a : monoide ; b : monoide] [S : (monoide_produit a b)]
    (mk_monoide (monoide_produit_T ? ? S) (* . . . *)).
Coercion monoide_produit_monoide : monoide_produit >-> monoide.

Definition monoide_produit_setoide_produit :=
  [a : monoide ; b : monoide] [S : (monoide_produit a b)]
   (mk_setoide_produit a b
     (monoide_produit_T ? ? S) (* . . . *)).
Coercion monoide_produit_setoide_produit :
     monoide_produit >-> setoide_produit.
```

Comme dans le cas de `setoide_produit`, ces deux coercions sont paramétrées par a et b. La première coercion est semblable à celles qu'on a vu précédemment. Pour la seconde, il convient en outre de remarquer qu'il y a de nouveau une coercion implicite à l'œuvre. En effet les monoides a et b sont transformés en `setoide` lorsqu'ils sont appliqués à `mk_setoide_produit`.

Enfin, la collection `entiers` est traduite par un **Chapter**, dans lequel on ne définit pas de type d'enregistrement. La dernière définition du **Chapter** est un enregistrement de type `monoide`, l'espèce qu'implante `entiers` :

```
Chapter Entiers.
...
Definition entiers := (mk_monoide self_T self_print self_parse self_egal
self_different self_refl self_symm self_multiplie self_un self_element).
End Entiers.
```

De même, `entiers_2` se traduit en un chapitre dont les deux premières définitions locales correspondent à l'implantation des paramètres de `monoide_produit`, l'espèce qu'elle implante :

```
Chapter Entiers_2.
  Local self_a :monoide :=entiers .
  Local self_b :monoide :=entiers .

  Definition entiers_2 := (mk_monoide_produit entiers entiers self_T
    self_parse self_egal self_different self_refl self_symm self_creer
    self_element self_print self_un self_multiplie).
End Entiers_2.
```

6.4 Formalisation de la traduction de Foc vers Coq

Le reste de ce chapitre est consacré à la définition formelle des mécanismes de traduction d'une espèce bien formée et bien typée en Coq. Comme dans le cas de la traduction vers Ocaml, nous allons décrire les différentes composantes de la traduction de Foc vers Coq en reprenant l'ordre dans lesquelles on les trouve dans un **Chapter** Coq correspondant à une espèce, c'est à dire

1. le type enregistrement correspondant à l'interface
2. les générateurs de méthodes
3. les définitions de méthodes dans le contexte de l'espèce, et la définition de l'enregistrement correspondant
4. les coercions
5. les paramètres

Enfin, nous reviendrons sur la traduction des méthodes récursives, et en particulier sur les obligations de preuve qu'elles entraînent.

Notation 10 *La traduction des contructions* Foc *en* Coq *sera désignée par* Foc $\rightarrowtail$ Coq

6.4.1 Types et expressions de base

Les types de base de Foc se traduisent assez facilement en Coq. En particulier, les types atomiques c, représentant le type support de la collection c correspondante, se traduisent directement. En effet, il existe une coercion implicite entre chaque type d'enregistrement de Foc et `Set` (voir ci-dessous). Le mécanisme de coercion de Coq peut dès lors transformer tout enregistrement c en l'élément de **Set** représentant le type support.

Par contre, les variables de types posent un problème technique particulier, puisqu'en Coq elles doivent se traduire par des arguments supplémentaires des fonctions (voir la section 6.1). Si cela n'est pas véritablement gênant au niveau des types (il suffit de générer des noms frais pour représenter ces variables), il ne faut pas oublier d'instantier ces arguments supplémentaires lors de la traduction des expressions. Enfin, les propriétés Foc se traduisent trivialement en leur équivalent Coq.

Définition 52 (Traduction des types Foc en Coq)

$$\frac{\tau_1 \rightarrowtail \tau_1' \qquad \tau_2 \rightarrowtail \tau_2'}{\tau_1 \rightarrow \tau_2 \rightarrowtail \tau_1' \rightarrow \tau_2'} \qquad \frac{}{c \rightarrowtail c} \qquad \frac{\mathcal{V}(\alpha) = a}{\alpha \rightarrowtail a} \qquad \frac{\text{Dans une espèce } s}{\textit{self} \rightarrowtail \texttt{rep}}$$

Dans la traduction Coq, le type support d'une espèce s est représenté comme toutes les autres méthodes de s, par une variable de même nom.

La traduction des expressions de base ne pose pas de problème particulier, sauf en ce qui concerne les appels de méthodes. Dans la partie OCAML de la traduction, on peut totalement découpler la traduction de $\mathcal{C}$ et celle de $\mathcal{E}$. En effet, les liens entre objets et classes s'établissent par analyse de la structure des objets : si un objet implante les méthodes déclarées dans une classe avec le type attendu, il peut recevoir comme type cette classe.

Il en va autrement en COQ où le recours à des enregistrements impose qu'on conserve le lien entre chaque collection et l'espèce qu'elle implante, afin de connaître le nom du projecteur approprié.

Définition 53 (Environnement valide)
Soit $\mathcal{C}, \mathcal{E}, \Sigma, \Gamma$ un environnement FOC *bien formé. On suppose qu'il existe une fonction ∇ de* Codom($\mathcal{C}$) *dans* Codom($\mathcal{E}$) *telle que, $\forall c \in \mathcal{C}$, on ait :*

$$\mathcal{C}(c) = \langle x_i : \tau_i \rangle \Rightarrow \mathcal{E}(\nabla(c)) = (c_p \,.\, t_p)\{x_i : \tau_i'\}$$

De plus, il existe une substitution θ des c_p paramètres de collection vers $\mathcal{C}$ telle que $\forall i, \tau_i = \tau_i'\theta$. On dira alors que l'environnement est valide.

Lemme 30 (Ajout d'une collection) *Soient $\mathcal{C}, \mathcal{E}, \Sigma, \Gamma$ un environnement* FOC *valide et c une collection définie par*

collection c **implements** $s(e_1,\ldots,e_{p_f})$

et telle que $\mathcal{C}, \mathcal{E}, \Sigma, \Gamma \vdash c : \langle x_i : \tau_i \rangle$. Alors, $\mathcal{C} + c : \langle x_i : \tau_i \rangle, \mathcal{E}, \Sigma, \Gamma$ est un environnement valide, avec $\nabla(c) = s$ et $\theta(c_p) = e_p$ pour chaque c_p paramètre de collection.

Preuve. Immédiat par induction sur le nombre de paramètres, d'après les règles de typage COLL et COLL-INST □

Notons par ailleurs que deux types enregistrement ne pouvant partager des champs, il faut créer des noms d'étiquette frais pour chaque nouvelle espèce[2]. On supposera pour cela qu'on dispose d'une fonction $\mathbf{E}(s, m)$, qui renvoie un identificateur frais pour chaque couple (s, m).

Définition 54 (Traduction des expressions de FOC en COQ) *On définit la traduction des expressions* FOC *en* COQ *à l'aide de la fonction auxiliaire $\rightarrowtail^e$. $\rightarrowtail$ ajoute devant la traduction $\rightarrowtail^e$ de l'expression proprement dite*

[2]Le nouveau mécanisme de module de COQ pourrait rendre cette contrainte superflue.

les variables de types apparaissant libres dans l'expression.

$$\frac{\Gamma(x) = \forall \alpha_1, ..., \alpha_n, \tau}{\Omega, \Gamma \vdash x \rightarrowtail^e (x \overbrace{?...?}^{n\ fois})} \qquad \frac{\Omega, \Gamma + x : \tau \vdash e \rightarrowtail^e e', \tau \rightarrowtail \tau'}{\Omega, \Gamma \vdash \textbf{\textit{fun}}(x\ \textbf{\textit{in}}\ \tau)e \rightarrowtail^e\ [x : \tau']e'}$$

$$\frac{Gen(\Gamma, \tau) = \sigma \qquad \Omega, \Gamma + x : \sigma \vdash e_2 \rightarrowtail^e e_2' \qquad \Omega, \Gamma \vdash e_1 \rightarrowtail e_1'}{\Omega, \Gamma \vdash \textbf{\textit{let}}\ x\ \textbf{\textit{in}}\ \tau = e_1\ \textbf{\textit{in}}\ e_2 \rightarrowtail^e \qquad [x := e_1']e_2'}$$

$$\frac{\Omega \vdash f \rightarrowtail^e f' \qquad \forall i, \Omega \vdash e_i \rightarrowtail^e e_i'}{\Omega \vdash f(e_1, ..., e_n) \rightarrowtail^e (f'\ e_1' \ldots e_n')} \qquad \begin{array}{l}[\textsc{Self}] \\ self!m \rightarrowtail^e m\end{array}$$

$$[\textsc{Meth}]\quad \frac{\nabla(c) = s \qquad \mathcal{E}(s) = (c_1\ .\ t_1, ..., c_{p_f}\ .\ t_{p_f})\{x_i : \tau_i = e_i\}}{c!m \rightarrowtail^e (\mathbf{E}(s, m)\ \theta(c_1)...\theta(c_{p_f})\ c)}$$

$$\frac{\Omega \vdash e : \tau \qquad Gen(\Gamma, \tau) = \forall \alpha_1, ..., \alpha_n \tau \qquad A_1, ..., A_n \text{ fraîches} \qquad \Omega, \mathcal{V} + \{\alpha_i \mapsto A_i\} \vdash e \rightarrowtail^e e'}{\Gamma \vdash e \rightarrowtail (A_1 : Set)...(A_n : Set)e'}$$

On peut alors définir la traduction des propriétés Foc. Le seul point délicat est qu'il faut faire apparaître explicitement la conversion de `bool` vers `Prop` quand il y a lieu. On le fait à l'aide de la fonction *Is_true* de la bibliothèque standard de Coq. Les fonctions booléennes de Foc représentent en fait des propriétés *calculables*, pouvant être utilisées dans les définitions de fonctions, et destinées à être traduites en Ocaml. Au contraire, les éléments de Foc ayant le type `Prop` n'existent que dans la représentation Coq.

Définition 55 (Traduction des propriétés) *Soit p une propriété* Foc *bien typée. On définit sa traduction p' en* Coq*, notée $p \rightarrowtail p'$ à partir de la fonction intermédiaire $\rightarrowtail^p$, et d'une éventuelle coercion de* `bool` *vers* `Prop` *de la manière suivante :*

$$\frac{\Omega \vdash e : bool \qquad \Omega \vdash e \rightarrowtail e'}{\Omega \vdash e \rightarrowtail^p (Is\_\ true\ e')} \qquad \frac{\Omega \vdash e : Prop \qquad \Omega \vdash e \rightarrowtail e'}{\Omega \vdash e \rightarrowtail^p e'}$$

$$\frac{\Omega \vdash p_1 \rightarrowtail^p p_1' \qquad \Omega \vdash p_2 \rightarrowtail^p p_2'}{\Omega \vdash p_1 \rightarrow p_2 \rightarrowtail^p p_1' \rightarrow p_2'} \qquad \frac{\Omega \vdash p_1 \rightarrowtail^p p_1' \qquad \Omega \vdash p_2 \rightarrowtail^p p_2'}{\Omega \vdash p_1\ \textbf{\textit{and}}\ p_2 \rightarrowtail^p (and\ p_1'\ p_2')}$$

$$\frac{\Omega \vdash p_1 \rightarrowtail^p p_1' \qquad \Omega \vdash p_2 \rightarrowtail^p p_2'}{\Omega \vdash p_1\ \textbf{\textit{or}}\ p_2 \rightarrowtail^p (or\ p_1'\ p_2')} \qquad \frac{\Omega \vdash p \rightarrowtail^p p'}{\Omega \vdash \textbf{\textit{not}}\ p \rightarrowtail^p (not\ p')}$$

$$\frac{\Omega \vdash p \rightarrowtail^p p' \qquad \Omega \vdash \tau \rightarrowtail \tau'}{\Omega \vdash \textbf{\textit{all}}\ x\ \textbf{\textit{in}}\ \tau, p \rightarrowtail^p (x : \tau')p'} \qquad \frac{\Omega \vdash p \rightarrowtail^p p' \qquad \Omega \vdash \tau \rightarrowtail \tau'}{\Omega \vdash \textbf{\textit{ex}}\ x\ \textbf{\textit{in}}\ \tau, p \rightarrowtail^p (EX\ x : \tau' | p')}$$

6.4.2 Enregistrements avec champs dépendants

Comme on l'a déjà fait remarquer, dans un enregistrement COQ, les types des champs peuvent dépendre des champs *précédents* : on va donc être amené à utiliser la forme normale d'une espèce pour créer le type d'enregistrement correspondant. De plus, l'appel à une méthode m de **self** n'est pas transformé, comme dans $\rightarrowtail^e$ en la variable m. En effet, il s'agit de faire référence au champ correspondant, qui a pour étiquette $\mathbf{E}(s, m)$.

Définition 56 (méthodes dans un type d'enregistrement)
Soit s une espèce bien formée et bien typée. On définit la traduction d'un type τ ($\rightarrowtail^r_s$) de la même manière que $\rightarrowtail$, à l'exception de la règle SELF*, qui prend la forme suivante :*

$$\frac{}{\textbf{self} \rightarrowtail^r_s \mathbf{E}(s, \textbf{\textit{rep}})}\ \textsc{Self}$$

*De même, la traduction d'une expression est identique à la définition donnée pour $\rightarrowtail^e$ sauf pour l'appel à une méthode de **self**, qui est traduit par :*

$$\frac{}{\textbf{self}!m \rightarrowtail^r_s \mathbf{E}(s, m)}\ [\textsc{Self}]$$

Définition 57 (Traduction d'une interface FOC) *Soit s une espèce de forme normale $\phi_1, ..\phi_n$. On définit la traduction d'un champ de la manière suivante :*

$$\frac{\Omega \vdash \tau \rightarrowtail^r_s \tau'}{\Omega \vdash \textbf{\textit{sig}}\ x\ \textbf{\textit{in}}\ \tau \rightarrowtail \mathbf{E}(s, x) : \tau'} \qquad \frac{\Omega \vdash \tau \rightarrowtail^r_s \tau'}{\Omega \vdash \textbf{\textit{let}}\ x\ \textbf{\textit{in}}\ \tau = e \rightarrowtail \mathbf{E}(s, x) : \tau'}$$

$$\frac{\forall i, \Omega \vdash \tau_i \rightarrowtail^r_s \tau_i'}{\textbf{\textit{let rec}}\ x_i\ \textbf{\textit{in}}\ \tau_i = e_i \rightarrowtail \mathbf{E}(s, x_i) : \tau_i'} \qquad \frac{\Omega \vdash p \rightarrowtail^r_s p'}{\Omega \vdash \textbf{\textit{property}}\ x :\ p \rightarrowtail \mathbf{E}(s, x) : p'}$$

$$\frac{\Omega \vdash p \rightarrowtail^r_s p'}{\Omega \vdash \textbf{\textit{theorem}}\ x :\ p\ \textbf{\textit{proof}} :... \rightarrowtail \mathbf{E}(s, x) : p'}$$

$$\Omega \vdash \textbf{\textit{rep}}[= \tau] \rightarrowtail \mathbf{E}(s, rep) :> Set$$

Le type d'enregistrement associé à s est alors (avec $\mathbf{E}(s, Build)$ un nom frais dans l'environnement) :

$$\frac{norm(s) = \phi_1 ... \phi_n \qquad \forall i, \phi_i \rightarrowtail \psi_i}{s \rightarrowtail \textbf{\textit{Record}}\ s\ : Type := \mathbf{E}(s, Build)\{\psi_1 ... \psi_n\}}$$

6.4.3 Générateurs de méthodes

Le deuxième point important de la traduction en Coq est la définition des *générateurs de méthodes* pour les méthodes définies –ou redéfinies– dans l'espèce qu'on est en train de traduire. Dans cette section, on se concentrera sur la gestion des dépendances (decl- comme def-), sous forme d'abstraction ou de variable locale. Le cas particulier des méthodes récursives est traité dans la section 6.5. Par ailleurs, on suppose que chaque théorème est accompagné de sa preuve Coq, faite dans un environnement tel qu'il est décrit ci-dessous. Enfin, tout comme dans le cas des étiquettes de la section précédente, on se donne une fonction $\mathbf{G}(s, m)$ qui à une espèce s et un nom de méthode m associe un nom frais dans l'environnement.

Afin de minimiser la taille du générateur correspondant à la méthode m, on souhaite ne garder dans Σ que les méthodes de **self** aux déclarations et définitions qui sont nécessaires pour typer le corps de m. On va donc considérer une restriction de Σ, appelée *l'univers visible* de m. Pour construire cet univers visible, il faut bien sûr distinguer les decl- et les def- dépendances de m, mais aussi, et particulièrement dans le cas des théorèmes, les decl-dépendances suivant qu'elles proviennent du corps ou du type de m. Pour fixer les idées, on peut prendre l'exemple de l'espèce suivante :

```
species s inherits basic_object =
  sig a in self ; sig op in self → self ; let c = !op( !a) ;
  sig p1 in self → Prop ; sig p2 in self → Prop ;
  property p1_p2 : all x in self, !p2(x) → !p1(x) ;
  property op_spec : all x in self, !p1(x)→ !p2( !op(x)) ;
  property a_spec : !p1( !a) ;
  theorem t1 : !p2( !c)
    proof : def c ; decl op_spec a_spec ;{* ... *} ;
  theorem t2 : !p1( !c) proof : decl t1 p1_p2 ; {* ... *} ;
end
```

D'après le calcul des dépendances, `t1` dépend directement de `p2` et `c`, mentionnés dans la propriété. De plus, la preuve nous donne une def-dépendance vis à vis de `c`, et des decl-dépendances vis-à-vis de `op_spec` et `a_spec`. Pour que la définition de `c` soit bien formée, il faut inclure dans l'environnement les déclarations de a et `op` dont elle decl-dépend. De même, l'énoncé de `op_spec` oblige à ajouter `p1` dans l'environnement. Enfin, comme toutes ces fonctions opèrent sur le type support, celui-ci doit être présent. Finalement, l'environnement minimal où `t1` est bien typé est toute l'espèce (à l'exclusion de `t2`). Si on s'était contenté de regarder `t1` et ses def-dépendances, on aurait oublié de rajouter `p1`, et l'énoncé de `op_spec` aurait été mal typé.

Dans le cas de `t2`, par contre, on a des decl-dépendances vis-à-vis de `p1`, `c`, `t1` et `p1_p2`. La prise en compte des énoncés de `t1` et `p1_p2` aboutit à ajouter `p2` dans l'environnement. Par contre, on n'a pas besoin d'analyser les

dépendances portées par le corps de `t1` (c'est à dire la preuve), et `op_spec`, a, et `op` ne sont donc pas conservées.

6.4.4 Dépendances dans les types

En premier lieu, il convient de revenir sur la définition de l'environnement dans lequel on effectue le typage des corps des méthodes définies au sein d'une espèce s. En effet, on place *a priori* dans Σ toutes les méthodes de **self**. Or l'analyse de dépendances permet d'être beaucoup plus économe. En effet, pour typer une expression donnée e, on n'a besoin que des méthodes auxquelles e fait effectivement appel, c'est à dire $\wr e \int$. L'affaire se complique quand les def-dépendances rentrent en jeu, puisqu'il faut alors garder des définitions dans Σ, et s'assurer que ces définitions sont elle-mêmes bien typées. Pour cela, il faut donc conserver dans l'environnement les méthodes qui appartiennent à la clôture transitive de la relation de def-dépendance ainsi que les méthodes dont les précédentes decl-dépendent.

Définition 58 (Univers visible d'une méthode) *Soit s une espèce bien formée et bien typée et $x \in \mathcal{N}(s)$. L'univers visible de x noté $\mid x \mid$ est défini de la manière suivante :*

$$\frac{y \in \wr x \int_s}{y \in \mid x \mid} \qquad \frac{y <_s^{def} x}{y \in \mid x \mid} \qquad \frac{z <_s^{def} x \qquad y \in \wr z \int_s}{y \in \mid x \mid} \qquad \frac{z \in \mid x \mid \qquad y \in \wr \mathcal{T}_s(z) \int_s}{y \in \mid x \mid}$$

La dernière règle est imposée par la présence de type dépendants : il faut tenir compte d'éventuelles decl-dépendances dans les types des méthodes que l'on conserve dans Σ. Cette notion d'univers visible permet de définir la restriction de Σ recherchée, en distinguant trois cas :

1. les méthodes qui ne sont pas dans l'univers visible de x, et qu'on efface complètement.
2. les méthodes qui sont dans l'univers visible de x, dont x ne def-dépend pas, et dont on conserve seulement le type
3. les méthodes dont x def-dépend, dont on conserve le type *et* le corps

Plus formellement, l'environnement minimal de typage d'une méthode peut se définir de la manière suivante :

Définition 59 (Environnement minimal de typage) *Soit s une espèce bien typée et bien formée, et $x \in \mathcal{N}(s)$. En utilisant les notations précédentes, et en posant $norm(s) = \{y_i : \tau_i = e_i\}$, on définit $norm(s) \Cap x$ de la manière*

suivante :

$$\emptyset \Cap x = \emptyset \qquad \frac{y \notin |\ x\ | \qquad \{y_i : \tau_i = e_i\} \Cap x = \Sigma}{\{y : \tau = e; y_i : \tau_i = e_i\} \Cap x = \Sigma}$$

$$\frac{y <_s^{def} x \qquad \{y_i : \tau_i = e_i\} \Cap x = \Sigma}{\{y : \tau = e; y_i : \tau_i = e_i\} \Cap x = \{y : \tau = e; \Sigma\}}$$

$$\frac{y \in |\ x\ | \qquad y \not<_s^{def} x \qquad \{y_i : \tau_i = e_i\} \Cap x = \Sigma}{\{y : \tau = e; y_i : \tau_i = e_i\} \Cap x = \{y : \tau = \bot; \Sigma\}}$$

On peut dès lors définir le générateur de méthode de x proprement dit. Pour cela, on abstrait toutes les méthodes déclarées (*i.e.* dont le corps vaut $\bot$) dans $\Sigma \Cap x$, et on définit des variables locales pour les méthodes définies de ce même environnement. Ces variables locales utilisent elles mêmes des générateurs de méthode, qui proviennent soit de l'espèce s proprement dite, soit de ses parents (dans le cas d'une méthode héritée). Pour obtenir une définition correcte de ces liaisons locales, nous avons donc besoin de $x \upharpoonleft s$ (définition 30), ainsi que d'une autre fonction auxiliaire, pour extraire d'un environnement minimal la liste des noms de méthode abstraits.

Définition 60 (Abstraction du contexte minimal) *Soit Σ un contexte de la forme $\{x_i : \tau_i = e_i\}$. On définit les* abstractions *de ce contexte, $\mathcal{H}(\Sigma)$ de la manière suivante :*

$$\mathcal{H}(\emptyset) = \emptyset \qquad \mathcal{H}(y : \tau = \bot; l) = y; \mathcal{H}(l) \qquad \mathcal{H}(y : \tau = e; l) = \mathcal{H}(l) \text{ si } e \neq \bot$$

Définition 61 (Traduction de l'environnement d'une méthode) *Soit s une espèce bien formée et bien typée, et $x \in \mathcal{D}(s)$ une méthode définie dans le corps de s. On définit le corps du générateur de méthode correspondant à x à l'aide de la traduction de l'univers visible de x, notée $[\![\cdot]\!]_s^x$, de la manière suivante :*

$$[\![\emptyset]\!]_s^x = \emptyset \qquad \frac{[\![l]\!]_s^x = \gamma \qquad \tau \rightarrowtail \tau'}{[\![\{y : \tau = \bot; l\}]\!]_s^x = [y : \tau]\gamma}$$

$$\frac{[\![l]\!]_s^x = \gamma \qquad \tau \rightarrowtail \tau' \qquad y \upharpoonleft s = s' \qquad \mathcal{H}(norm(s') \Cap y) = y_1 ... y_n}{[\![\{y : \tau = \mathcal{B}_s(y); l\}]\!]_s^x = [y := (\mathbf{G}(s', y) y_1 ... y_n)]\gamma}$$

$$\frac{[\![norm(s) \Cap x]\!]_s^x = \gamma \qquad \mathcal{T}_s(x) \rightarrowtail \tau \qquad \mathcal{B}_s(x) \rightarrowtail e}{\mathbf{G}(s, x) \rightarrowtail \textbf{\textit{Definition}}\ \mathbf{G}(s, x)\ \gamma : \tau := e}$$

Remarque 1. Notons que s étant bien formée, les générateurs de méthodes pour les y dont x def-dépend sont forcément définis avant $\mathbf{G}(s, x)$:

- Soit $y \upharpoonleft s \neq s$. Alors $\mathbf{G}(y \upharpoonleft s, y)$ a été défini dans un **Chapter** précédent de la traduction
- Soit y est défini dans s. Comme on suit l'ordre donné par $norm(s)$, $\mathbf{G}(s, y)$ est défini avant $\mathbf{G}(s, x)$.

6.4.5 Méthodes dans le contexte de $norm(s)$ et traduction des espèces

Si on ne s'intéressait qu'à la définition des enregistrements correspondant aux collections de FOC, la traduction d'une espèce pourrait s'arrêter là. En effet, les générateurs de méthodes sont suffisants pour cette tâche : pour définir la traduction de la collection définie en FOC par

collection `c` **implements** s

il suffit de retrouver les générateurs de toutes les méthodes x de s, $\mathbf{G}(x \upharpoonleft s, x)$ et de définir chaque champ en fonction des précédents dans l'ordre donné par $norm(s)$, forme normale de s.

Cependant, il peut être intéressant de définir localement, à l'intérieur du **Chapter** COQ correspondant à l'espèce FOC, les méthodes liées au contexte *courant* de l'espèce. Comme on le verra dans la section 7.6, ces définitions locales permettent de garantir la correction de chaque nœud de l'arbre d'héritage. Si on se contentait de définir les enregistrements correspondant aux collections, on ne ferait vérifier à COQ que la bonne définition de ces dernières, sans avoir de réelles garanties sur la hiérarchie des espèces.

Ces définitions locales se font de la même manière que pour la gestion des def-dépendances de la définition précédente : on retrouve l'espèce où la méthode est définie pour la dernière fois $(x \upharpoonleft s)$ et on applique aux générateurs correspondants les définitions courantes des méthodes dont ils decl-dépendent.

Définition 62 (Méthode sous contexte) *Soit s une espèce bien formée et bien typée, et $x \in \mathcal{N}(s)$. La* définition de x dans le contexte de s *est donnée par les règles suivantes :*

$$\frac{x \notin \mathcal{D}(s) \qquad \mathcal{T}_s(x) \rightarrowtail \tau}{\mathcal{L}_s(x) = \textbf{\textit{Variable}}\ x : \tau}$$

$$\frac{x \in \mathcal{D}(s) \qquad x \upharpoonleft s = s' \qquad \mathcal{H}(norm(s') \Cap x) = y_1 ... y_n}{\mathcal{L}_s(x) = \textbf{\textit{Local}}\ x := (\mathbf{G}(s', x)\ y_1\ ...\ y_n)}$$

Avec $norm(s) = \{x_i : \tau_i = e_i\}_{i=1...n}$, on définit le contexte d'enregistrement de s comme

$$\mathcal{L}_s^* = \mathcal{L}_s(x_1)...\mathcal{L}_s(x_n)$$

Enfin, on peut définir un enregistrement $\mathcal{R}(s)$ (def. 63) de type s à partir des définitions locales et des variables du **Chapter**. Lorsqu'on ferme le **Chapter**, ces variables sont abstraites par Coq. Cela permet d'obtenir une fonction qui associe aux méthodes déclarées de s un enregistrement dont les champs sont les définitions courantes des méthodes de $\mathcal{D}(s)$. En d'autres termes, les arguments de cette fonction sont les méthodes qu'il reste à fournir pour créer une collection implantant s. On peut donc la rapprocher de l'endofoncteur $\mathbb{G}$ – pour Générateur – de la catégorie des générateurs d'enregistrement de la thèse de Sylvain Boulmé ([Bou00], p.184).

Notons enfin que si l'espèce s est complètement définie, cette "fonction" est en fait, dans le cas d'une espèce atomique une constante : définir une collection c implantant l'espèce revient dès lors à donner un nom particulier c à $\mathcal{R}(s)$. Comme on le verra ci-dessous (section 6.4.7), dans le cas d'une espèce paramétrée, $\mathcal{R}(s)$ est abstrait par rapport aux paramètres. Il reste donc à appliquer la fonction obtenue aux instance voulues pour obtenir un enregistrement.

Définition 63 (Enregistrement sous hypothèses)
Soit s une espèce bien formée et bien typée, et $norm(s) = \{x_1 : \tau_1 = e_1, ..., x_n : \tau_n = e_n\}$ sa forme normale. On définit l'enregistrement $\mathcal{R}(s)$ sous les hypothèses $\mathcal{N}(s)\backslash\mathcal{D}(s)$ de la manière suivante :

$$\textbf{\textit{Definition}}\ \mathcal{R}(s)\ := (\mathbf{E}(s, \mathit{Build})\ x_1\ ...\ x_n).$$

De même, si c est une collection implantant s, on définit l'enregistrement c comme

$$\textbf{\textit{Definition}}\ c := \mathcal{R}(s).$$

6.4.6 Coercions

Le dernier point important de la traduction d'une espèce en Coq est la définition des coercions permettant de passer d'une espèce s à chacun de ses parents s_h. Pour cela, il suffit de disposer de la forme normale de s_h afin de savoir dans quel ordre fournir les méthodes au constructeur d'enregistrement $\mathbf{E}(s_h, \mathit{build})$. En effet, il n'y a *a priori* aucune raison pour que cet ordre soit le même que dans la forme normale de s.

Définition 64 (Coercion vers une espèce parent)
Soit s une espèce définie par

species s **inherits** $s_1,\ldots,s_{h_f}$ = ...

Pour tout $h \leq h_f$, on définit la coercion de s vers s_h, $s \Longmapsto s_h$ à partir de la forme normale de s_h, $norm(s_h) = \{x_1 : \tau_1 = e_1 ... x_m : \tau_m = e_m\}$:

$s \Longmapsto s_h =$
$\quad \textbf{\textit{Definition}}\ s\_s_h := [S : s](\mathbf{E}(s_h, \mathit{Build})(\mathbf{E}(s, x_1)\ S)...(\mathbf{E}(s, x_m)\ S)).$

6.4.7 Paramètres

Le traitement des paramètres en COQ ne pose pas de problème particulier : il suffit de placer dans le contexte une variable dont le type correspond au paramètre en question. On tire ici pleinement parti du mécanisme d'abstraction des sections de COQ puisque tous les générateurs dépendant d'un paramètre seront abstrait par rapport à ce paramètre, ainsi que l'enregistrement sous les hypothèses de l'espèce. Il n'y a pas besoin de faire faire le lambda-lifting par le compilateur FOC, contrairement aux dépendances vis-à-vis des méthodes de **self**, car les paramètres sont par définition toujours abstraits. Enfin, les paramètres forment les premiers champs du type d'enregistrement correspondant à l'espèce paramétrée.

En ce qui concerne les types d'enregistement, ainsi qu'on l'a vu, les paramètres FOC sont directement traduits en paramètres COQ pour les types d'enregistrement générés. Toutefois, utiliser des types d'enregistrement paramétrés a un coût : les projecteurs associés prennent des arguments supplémentaires correspondant à ces paramètres (ceux-ci pouvant néammoins être inférés par COQ la plupart du temps). Ainsi, `setoide_produit_creer` a le type (a,b :`setoide` ; s :(`setoide_produit` a b))`a`−>`b`−>`s`. On pourrait donc penser à introduire les paramètres comme des champs de l'enregistrement. De cette manière, les projections garderaient la même forme que l'espèce correspondante soit paramétrée ou non. Avec une telle traduction, le type d'enregistrement aurait la forme suivante :

```
Record setoide_produit : Type :=
  mk_setoide_produit {
  setoide_a : setoide ;
  setoide_b :setoide ;
  setoide_produit_T :> Set ;
  ...
  setoide_produit_creer :
      setoide_a → setoide_b → setoide_produit_T ;
  ...
}.
```

Dans cette traduction, `setoide_produit_creer` ne prend pas `a` et `b` en argument, ainsi que le montre son type :

(s :`setoide_produit`)(`setoide_a` s)→(`setoide_b` s)−>`s`

Les types de ses deux derniers arguments sont bien des champs de l'enregistrement s et n'ont donc pas à apparaître explicitement dans la signature de `setoide_produit_creer`.

Cette autre approche n'est malheureusement pas correcte. En effet, lorsqu'une espèce compte plusieurs paramètres, les derniers paramètres peuvent

dépendre des précédents, comme l'indiquent les règles de typage de la section 3.8. Dans ce cas, la traduction des paramètres en champs de l'enregistrement peut faire perdre des informations et aboutir à des termes Coq mal typés, ainsi que le montre l'exemple suivant :

```
species c (a is basic_object) inherits basic_object =
  rep = a ;
  let toc (x in a ) in self = x ;
end

species foo (a is basic_object, my_c is c(a))
  inherits basic_object =
    rep = my_c ;
    let toc (x in a) in self = my_c !toc(x) ;
end
```

Avec la traduction des paramètres en tant que champs d'un enregistrement,

on obtient le code Coq suivant :

```
Chapter C.
  Record c : Type := mk_c {
     c_a : basic_object;
     c_T :> Set;
     c_print : (c_T)→ string__t;
     c_parse : (string__t)→ c_T;
     c_toc : (c_a)→ c_T
  }.
  ...
End C.

Chapter Foo.
  Record foo : Type := mk_foo {
       foo_a : basic_object;
       foo_my_c : c;
       foo_T :> Set;
       foo_print : (foo_T)→ string__t;
       foo_parse : (string__t)→ foo_T;
       foo_toc : (foo_a)→ foo_T
  }.
  Variable self_a : basic_object.
  Variable self_my_c : c.
  Local self_T : Set := self_my_c.
  (* ERREUR *)
  Definition foo__toc :(self_a)→ self_T :=
           [x : self_a]((c_toc self_my_c ) x).

End Foo.
```

Les deux variables du chapitre `Foo` n'ont a priori aucun lien entre elles (du moins pour Coq). En particulier, on n'a aucun moyen de signaler à Coq que le premier champ de `self_my_c` doit être égal à `self_a`. Or le typage du générateur correspondant à `toc` ne peut se faire que dans un environnement où cette égalité est connue, puisqu'on applique un argument de type `self_a` à une fonction qui attend un élément de type `(c_a my_c)`. La définition ci-dessus est donc rejetée par Coq.

Au contraire, si les types d'enregistrements portent explicitement leurs paramètres, on ne perd pas ce lien entre les paramètres : la traduction a la

forme suivante.

```
Chapter C.
  Record c[a : basic_object] : Type := mk_c { ...
      c_toc : (a)→ c_T }
End C.

Chapter Foo.
  Record foo[a : basic_object ; my_c : (c a)] : Type :=
    mk_foo { (* . . . *) }

  Variable self_a : basic_object.
  Variable self_my_c : (c self_a).

  Definition foo__toc :(self_a)→ self_T :=
    [x : self_a] ((c_toc ? self_my_c ) x).
End Foo.
```

Cette fois-ci, `self_my_c` est un enregistrement de type `(c self_a)`, et le lien entre les deux paramètres apparaît bien. Dans le générateur de la méthode `toc`, on peut donc utiliser le projecteur `c_toc`, qui prend en premier argument le paramètre de l'enregistrement (qu'on peut demander à Coq d'inférer, puisque `self_my_c` contient cette information) et l'enregistrement lui-même. Le résultat est bien une fonction qui attend un argument de type `self_a`, et la définition du générateur est donc dans ce cas bien typée. Bien que plus lourde à gérer en ce qui concerne les appels de méthodes, cette solution doit donc être retenue, puisqu'elle seule permet d'obtenir des définitions bien typées pour toutes les expressions Foc. Nous allons maintenant détailler cette traduction.

Définition 65 (Type d'enregistrement d'une espèce paramétrée) *Soit s une espèce définie par*

species $s(c_1 \text{ . } \tau_1,\ldots,c_k \text{ . } \tau_k)$ ...

telle que sa forme normale soit $norm(s) = \phi_1 ... \phi_n$. On définit alors le type d'enregistrement associé à s de la manière suivante :

$$\frac{\forall j, \tau_j \rightarrowtail \tau'_j \qquad \forall i, \phi_i \rightarrowtail \psi_i}{s \rightarrowtail \textbf{\textit{Record}}\ s\ [c_1 : \tau'_1; ...; c_n : \tau'_n] : \qquad Type := \mathbf{E}(s, Build) \qquad \{\psi_1, ...\psi_n\}}$$

Par ailleurs, l'utilisation des générateurs de méthode correspondant est plus complexe. En effet, il faut substituer au paramètre formel son instantiation dans l'environnement courant. De plus, comme un générateur de méthode peut être utilisé pendant plusieurs "générations" successives d'espèce, on peut être amené à faire récursivement plusieurs instantiations. Par

ailleurs, comme Coq n'abstrait que par rapport aux variables réellement utilisées à l'intérieur de la définition, il faut examiner le corps de la méthode correspondante pour savoir quels sont les paramètres qu'elle utilise.

Définition 66 (Paramètres utilisés par une méthode)
Soient s une espèce définie par

species $s(c_1 \text{ . } \tau_1,\ldots,c_{p_f} \text{ . } \tau_{p_f})$ ...

et $x \in \mathcal{D}(s)$ une méthode telle que $x \upharpoonleft s = s$. On définit les paramètres utilisés dans une expression e, $\mathcal{U}_s(e)$ de la manière suivante :

$$\begin{aligned}
\mathcal{U}_s(y) &= \begin{cases} \{y\} & \text{si } y \in P_e \\ \emptyset & \text{sinon} \end{cases} \\
\mathcal{U}_s(\textbf{let } y = e_1 \textbf{ in } e_2) &= \mathcal{U}_s(e_1) \cup (\mathcal{U}_s(e_2) \backslash \{y\}) \\
\mathcal{U}_s(\textbf{let rec } y = e_1 \textbf{ in } e_2) &= (\mathcal{U}_s(e_1) \cup \mathcal{U}_s(e_2)) \backslash \{y\} \\
\mathcal{U}_s(\lambda y.e) &= \mathcal{U}_s(e) \backslash \{y\} \\
\mathcal{U}_s(c!y) &= \begin{cases} \{c\} & \text{si } c \in P_c \\ \emptyset & \text{sinon} \end{cases}
\end{aligned}$$

Cette définition est étendue aux propriétés de la manière suivante :

$$\begin{aligned}
\mathcal{U}_s(p_1 \textbf{ or } p_2) &= \mathcal{U}_s(p_1) \cup \mathcal{U}_s(p_2) \\
\mathcal{U}_s(p_1 \textbf{ and } p_2) &= \mathcal{U}_s(p_1) \cup \mathcal{U}_s(p_2) \\
\mathcal{U}_s(p_1 \rightarrow p_2) &= \mathcal{U}_s(p_1) \cup \mathcal{U}_s(p_2) \\
\mathcal{U}_s(\textbf{not } p) &= \mathcal{U}_s(p) \\
\mathcal{U}_s(\textbf{all } y \textbf{ in } \tau, p) &= \mathcal{U}_s(p) \\
\mathcal{U}_s(\textbf{ex } y \textbf{ in } \tau, p) &= \mathcal{U}_s(p)
\end{aligned}$$

On définit alors les paramètres utilisés dans la définition de x ($\mathcal{U}_s(x)$) avec les règles suivantes :

Body
$$\frac{c_p \in \mathcal{U}_s(\mathcal{B}_s(x))}{c_p \in \mathcal{U}_s(x)}$$

Type
$$\frac{c_p \in \mathcal{U}_s(\mathcal{T}_s(x))}{c_p \in \mathcal{U}_s(x)}$$

Def-dep
$$\frac{y <_s^{def} x \qquad c_p \in \mathcal{U}_s(\mathcal{B}_s(y))}{c_p \in \mathcal{U}_s(x)}$$

Univers
$$\frac{y \in\mid x \mid \qquad c_p \in \mathcal{U}_s(\mathcal{T}_s(y))}{c_p \in \mathcal{U}_s(x)}$$

Prm
$$\frac{c_{p'} \in \mathcal{U}_s(x) \qquad c_p \in \mathcal{U}_s(\tau_{p'})}{c_p \in \mathcal{U}_s(x)}$$

où P_c (respectivement P_e) est l'ensemble des indices p tels que c_p soit un paramètre de collection (respectivement d'entité).

Comme dans la définition du contexte minimal, on prend en compte les définition des méthodes dont x def-dépend ainsi que les types des méthodes dont x decl-dépend pour vérifier si un paramètre c_p est utilisé ou non. Par ailleurs, la règle PRM vient là encore du fait qu'un paramètre $c_{p'}$ peut dépendre d'un paramètre c_p (avec $p < p'$) : si $c_{p'}$ est utilisé par x, c_p l'est implicitement.

Définition 67 (instantiation des paramètres d'une espèce) *Soit s une espèce, et $x \in \mathcal{D}(s)$. La liste des instantiations des paramètres de x, $Inst_s(x)$ est définie de la manière suivante :*

$$\frac{x \upharpoonleft s = s}{Inst_s(x) = \emptyset}$$

$$\frac{h = \mathbb{I}_s(x) \qquad x \upharpoonleft s = s_h \qquad \mathcal{E}(s_h) = (c_1 \centerdot \tau_1, ..., c_{p_f} \centerdot \tau_{p_f})defs \qquad l_h = e_1...e_{p_f}}{Inst_s(x) = \{Inst_{c_p}(\})_{c_p \in \mathcal{U}_{s_h}(x)\}}$$

$$\frac{h = \mathbb{I}_s(x) \qquad x \upharpoonleft s = s' \qquad \mathcal{E}(s_h) = (c_1 \centerdot \tau_1, ..., c_{p_f} \centerdot \tau_{p_f})defs \qquad l_h = e_1...e_k \qquad Inst_{s_h}(x) = \{a_1...a_k\}}{Inst_s(x) = \{a_i[c_p \leftarrow e_p]\}}$$

Dès lors, il n'y a plus qu'à ajouter $Inst_s(x)$ en tête des arguments passés à chaque générateur de méthode, que ce soit dans les définitions liées aux def-dépendances ou dans les définitions des méthodes sous hypothèse.

Définition 68 (Méthode sous contexte en présence de paramètre) *Soient s et $x \in \mathcal{D}(s)$, telle que $x \upharpoonleft s \neq s$. On modifie la définition def.62 pour tenir compte des paramètres des parents de s :*

$$\frac{x \in \mathcal{D}(s) \qquad x \upharpoonleft s = s' \qquad \mathcal{H}(norm(s') \Cap x) = y_1...y_n}{\mathcal{L}_s(x) = \qquad \textbf{\textit{Local}}\ x := (\mathbf{G}(s', x)\ Inst_s(x)\ y_1...y_n)}$$

En ce qui concerne les collections, il faut désormais tenir compte du fait qu'elles peuvent implanter des espèces paramétrées. Il ne s'agit donc pas d'un simple renommage de `new_spec` comme dans la définition 63, mais d'une application de fonction :

Définition 69 (Création de collection) *Soit c une collection définie par*

collection `c` **implements** $s(e_1,\ldots,e_{p_f})$

On définit l'enregistrement COQ *correspondant de la manière suivante :*

$$\frac{\forall i, e_i \rightarrowtail e'_i \qquad \mathcal{R}(s) = s'}{c \rightarrowtail \textbf{\textit{Definition}}\ c := (s'\ e'_1\ ...\ e'_n)}$$

6.5 Méthodes récursives et preuves de terminaison

Dans ce qui précède, nous avons considéré que les méthodes récursives faisaient l'objet d'un traitement spécial, en dehors de la traduction de l'espèce elle-même. En fait, il s'agit surtout d'adapter le générateur de ces méthodes pour qu'il accepte une définition récursive. En premier lieu, on peut distinguer, suivant la nomenclature de Coq, deux types d'induction :

- l'induction structurelle.
- l'induction bien fondée.

Chacun de ces deux types conduit à une traduction différente. En effet, dans le premier cas, la définition récursive peut être traduite directement à partir du prédicat d'élimination du type inductif concerné. Dans le second, en revanche, l'utilisateur doit donner une preuve explicite de la terminaison de sa fonction. Nous allons maintenant examiner successivement ces deux cas.

Induction structurelle Le premier cas est le plus simple. Il s'applique quand la fonction récursive prend comme argument des éléments d'un type inductif, et que chaque appel récursif se fait avec des arguments structurellement plus petits que les arguments de départ. Ainsi, la définition Foc suivante, qui calcule la longueur d'une liste, est-elle structurellement récursive :

```
let rec length l =
  match l with
      #Nil → 0
    | #Cons(_,tl) → 1 + #length(tl)
;;
```

Intuitivement, on appelle `length` sur la liste `tl` qui compte un constructeur de moins que `l`. Comme toute liste est formée d'un nombre fini de constructeurs `#Cons` (et `#Nil`, celui-ci n'apparaissant qu'à la fin), tout appel à `length` se termine et renvoie un entier.

Les définitions structurellement récursives peuvent être traduites en Coq par l'intermédiaire d'un **Fixpoint**. Cela revient en fait à utiliser le schéma d'élimination du type inductif correspondant.

Induction bien fondée Le second cas est plus difficile à mettre en œuvre dans Coq, mais plus général. Il faut exhiber *un ordre bien fondé* $\prec$ sur les type des arguments de la fonction. Un ordre bien fondé est une relation d'ordre sur un ensemble E telle qu'il n'existe pas de suite infinie de termes strictement décroissante pour cet ordre. On dit alors que E est bien fondé. Ainsi, $\mathbb{N}$ muni de la relation d'ordre usuelle est un ensemble bien fondé. Il reste alors à montrer que chaque appel récursif se fait sur un argument

strictement plus petit au sens de $\prec$. Comme par définition on ne peut avoir une suite infinie d'arguments strictement plus petits, la fonction termine.

Par exemple, la définition de l'exponentielle sur un monoide par dichotomie (donnée p.49) est une récursion bien fondée. En effet en prenant l'ordre usuel sur $\mathbb{N}$, il faut montrer

- n pair $\vee\ n > 0 \Rightarrow \frac{n}{2} < n$
- n impair $\vee\ n > 0 \Rightarrow \frac{n-1}{2} < n$

L'hypothèse $n > 0$ provient du test qui est effectué au début de la fonction : il n'y a pas d'appel récursif dans le cas contraire. Il convient de remarquer que sans cette hypothèse, nos propriétés ne sont pas vraies : Ainsi, $\frac{0}{2} \not< 0$. Les énoncés des lemmes assurant la terminaison doivent donc suivre exactement la définition de la fonction, afin de contenir toutes les hypothèses nécessaires.

La librairie standard de Coq contient un certain nombre de fonctions permettant de manipuler de telles définitions, mais leur maniement est relativement complexe. La traduction d'une telle définition Foc vers Coq se fait au travers d'une **Section** où des variables locales permettent de manipuler l'ordre bien fondé et les différents lemmes de terminaison. Leurs énoncés sont engendrés par l'analyse du code de la fonction elle même. Cette traduction s'inspire largement de la tactique `Program` de Catherine Parent [Par95]. A titre d'exemple, la définition de `exp` sera traduite de la manière suivante :

```
Section Exp.
  Variable abst_T : Set.
  Variable abst_mult : (abst_T)→ (abst_T)→ abst_T.
  Variable abst_one : abst_T.
```

On commence bien sûr par définir les variables Coq correspondant aux decl-dépendances de la définition récursive. De ce point de vue, il n'y a pas de différence avec les définitions non-récursives.

La seconde partie de la **Section** est consacrée à l'ordre qui va assurer la terminaison :

```
Local my_order :((prod abst_T int__t))→
                ((prod abst_T int__t))→ Prop :=
  [x : (prod abst_T int__t)][y : (prod abst_T int__t)] (* ... *)
```

On procède ensuite à la définition d'un prédicat binaire sur l'ensemble des arguments de la fonction, c'est à dire ici le produit `abst_T * int__t` (représentation Coq de **self** * `int`).

```
Local well_founded_my_order :(well_founded ? my_order).
  EApply dontwanttoproveit.
Qed.
```

On montre enfin que `my_order` est un ordre bien fondé. `well_founded` est un prédicat de la librairie standard de Coq.

On peut dès lors énoncer –et prouver– les lemmes intermédiaires qui montrent que chacun des deux appels récursifs s'effectue sur un argument strictement plus petit –au sens de `my_order`– que l'argument initial (`x`,`n`).

```
Local lemma_1 :
  (_aux1 :(prod abst_T int__t))
  [x :=(first _aux1) :abst_T][n :=(scnd _aux1) :int__t]
  (Is_true (not_b (int_leq n `0`)))→
  (Is_true (int_eq (int_mod n `2`) `0`))→
  (my_order
     (crp (abst_mult x x) (int_div n `2`))
     _aux1).
  Proof.
    EApply dontwanttoproveit.
  Qed.

Local lemma_2 :
  (_aux1 :(prod abst_T int__t))
  [x :=(first _aux1) :abst_T][n :=(scnd _aux1) :int__t]
  (Is_true (not_b (int_leq n `0`)))→
  (Is_true (not_b (int_eq (int_mod n `2`) `0`)))→
  (my_order
    (crp (abst_mult x x) (int_div (int_moins n `1`) `2`))
    _aux1).
  Proof.
    EApply dontwanttoproveit.
  Qed.
```

Comme on l'a déjà dit, les énoncés des deux lemmes contiennent des hypothèses représentant les branches des expressions conditionnelles dans lesquelles se trouvent les appels récursifs.

Une fois les lemmes énoncés, on peut passer à la définition de `exp` proprement dite. Cette définition se fait en deux étapes, à l'aide de la fonction

`fix` de la bibliothèque standard de Coq :

```
Local exp_aux :
  (_aux1 :(prod abst_T int__t))
  ((_y :(prod abst_T int__t))(my_order _y _aux1)→ abst_T)→
  abst_T :=
  [_aux1 :(prod abst_T int__t)]
  [exp_aux_rec :
     ((_y :(prod abst_T int__t))(my_order _y _aux1)→ abst_T)]
  [x := (first _aux1)][n := (scnd _aux1)]
  if (int_leq n `0`) then
    abst_one
  else
    if (int_eq (int_mod n `2`) `0`) then
      (exp_aux_rec (crp (abst_mult x x) (int_div n `2`))
         (lemma_1 _aux1 (* . . . *))
    else
     (abst_mult x (exp_aux_rec
                  (crp (abst_mult x x)
                       (int_div (int_moins n `1`) `2`))
                  (lemma_2 _aux1 (* . . . *)))).
```

On commence par définir une fonction auxiliaire `exp_aux` dont on va calculer le point fixe. Cette fonction prend deux arguments :

1. un couple (`x`,`n`).
2. une fonction `exp_aux_rec` qui à tout couple (`x'`,`n'`) strictement plus petit que (`x`,`n`) associe un élément de `abst_T`.

Elle renvoie elle-même un élément de `abst_T`. Le corps de cette fonction est la traduction directe du corps de `exp` dans le source Foc, chaque appel récursif étant remplacé par un appel à `exp_aux_rec`, à laquelle on fournit un argument supplémentaire, la preuve que l'argument est strictement plus petit que (`x`,`n`), sous forme des lemmes intermédiaires.

```
  Local monoid_exp_aux := (fix ? my_order well_founded_my_order
     ([x :(prod abst_T int__t)]abst_T) exp_aux).
  Definition monoid__exp :=[x ; n](monoid_exp_aux (x,n)).
End Exp.
```

Enfin, on utilise `fix` pour définir le générateur de la méthode `exp`. Cette fonction de la bibliothèque standard de Coq construit le point fixe d'une fonction, à partir d'un ordre bien fondé. Plus précisément, les arguments de `fix` sont les suivants :

1. l'ensemble `A` sur lequel est défini la fonction (ici, il est inféré par Coq)
2. un prédicat binaire `R` sur `A` (`my_order`)

3. la preuve que ce prédicat est bien fondé (`well_founded_my_order`)
4. le type de retour de la fonction, Comme `fix` est une fonction très générale, ce type, P peut dépendre de l'argument de départ, et doit donc avoir lui-même le type `A->Set`. Ici, on l'instantie par la fonction constante égale à `abst_T`.
5. une fonction `f` qui prend en argument un élément de `x` de `A` et une fonction qui à tout `y` de `A` strictement plus petit que `x` (vérifiant (`R` y x) `associe un` élément `de` (P y). f `renvoie un` élément `de` (P x). `Ici`, cette fonction est `exp_aux`

`fix` renvoie alors une fonction qui à tout `x` de `A` renvoie un élément de (P `x`), en instantiant le second argument de `f` par [`y` :`A`][(`R y x`)](`fix R wfR f`). En d'autres termes, on bâtit le plus petit point fixe de `f`, qui vérifie l'égalité (`fix f x`)=(`f` (`fix f x`)). Le fait que `f` n'utilise son second argument que sur des éléments strictement plus petits que l'argument de départ, et que l'ordre soit bien fondé garantissent ainsi que `f` ne sera appelée qu'un nombre fini de fois lors de cette construction.

Actuellement, le compilateur FOC génère une **Section** correspondant à ce que nous venons de décrire pour chaque méthode récursive. Toutefois, l'ordre, sa bonne formation et les lemmes intermédiaires ne sont pas démontrés (il n'y a pas de syntaxe en FOC pour le faire). Le seul moyen de finir la preuve consiste donc à éditer le code COQ engendré à la main, pour remplacer les axiomes `dontwanttoproveit` par de vraies preuves. De plus, la gestion des méthodes mutuellement récursives reste à faire (cf. section 11.1.1).

Chapitre 7

Correction de la traduction Coq

Si le typage par Ocaml des classes obtenues par le compilateur Foc offre un certain nombre de garanties quant à la correction du code obtenu, les vérifications de type faites par Coq permettent d'aller bien plus loin dans l'optique de la certification du code.

En premier lieu, la traduction vers Coq inclut les propriétés et théorèmes écrits en Foc, et c'est Coq qui en dernier ressort décide de la véracité des preuves d'un source Foc[1]. Toutefois, même si on ne s'intéresse pas aux théorèmes des espèces Foc en tant que tels, la traduction de Foc vers Coq et le typage par Coq des définitions résultantes permet d'offrir certaines garanties de correction de l'analyse de dépendance telles qu'elle est faite en Foc. Plus précisément, l'analyse par Coq du résultat de la traduction permet de s'assurer des points suivants :

1. tous les cycles de dépendances sont repérés et pris en compte sous un **let rec**
2. l'*univers visible* de chaque méthode est calculé correctement (on n'oublie pas de dépendance dans ce calcul)
3. Les générateurs de méthodes sont toujours utilisés dans un contexte admissible (on n'oublie pas d'effacer une définition à cause de def-dépendances)

Le fait qu'aucune espèce ne puisse contenir de cycle de dépendance en dehors d'un **let rec** est de toute façon un élément essentiel de la mise en forme normale : si ce n'était pas le cas, le compilateur lui même bouclerait, et il n'y aurait pas besoin de la traduction Coq pour voir qu'il y a un problème. Toutefois, l'utilisation de Coq permet de forcer à présenter une preuve de terminaison pour chaque méthode récursive (voir la section 6.5), ce qui est

[1] Rappelons d'ailleurs qu'en pratique à l'heure actuelle les preuves Foc sont des scripts Coq, Foc se "contentant" de mettre en place l'environnement approprié.

une propriété importante pour obtenir du code certifié. Dans le reste de ce chapitre, nous supposerons que toutes les preuves de terminaison nécessaires ont été faites, et que les expressions récursives ont été "rejetées" à l'extérieur des enregistrements et des générateurs de méthodes selon la technique décrite dans la section 6.5.

Par ailleurs, la définition des générateurs de méthodes telles qu'on l'a vue dans la section 6.4.3 n'est acceptable pour COQ que si *l'univers visible* de ces méthodes est correctement calculé : si on oublie des dépendances, le contexte dans lequel COQ essaiera de typer le corps de la fonction ne sera pas correct, et on aura une erreur de typage. La section 7.6 montre qu'on construit bien un environnement correct pour chaque générateur de méthode.

Enfin, une fois qu'on a défini ces générateurs, il reste à les utiliser pour créer les méthodes proprement dites. Comme on l'a dit dans la section 6.4.5, on pourrait effectivement se contenter de n'utiliser ces générateurs qu'au moment de la création des collections, puisque ce sont les seuls endroits où on construit des enregistrements complets (sans méthodes virtuelles). Toutefois, créer des méthodes dans chaque contexte où c'est possible permet de s'assurer que chaque étape de construction de l'arbre d'héritage est correcte, c'est à dire qu'on ne conserve que des définitions qui sont correctes dans le contexte de l'espèce courante. La section 7.6 montre que c'est effectivement le cas.

7.1 Univers visible et générateur de méthode

Le lemme suivant montre que la définition de l'univers visible permet bien de restreindre la vue qu'on a de **self** aux champs nécessaires pour typer $\mathcal{B}_s(x)$ (à condition que $\tilde{s}$ soit bien typée). On y montre en fait deux choses :

1. que l'environnement ainsi restreint est bien typé
2. qu'on peut typer $\mathcal{B}_s(x)$ s'il était typable avec l'espèce $\tilde{s}$ entière.

Lemme 31 (Correction du typage) *En supposant que s est bien formée, et avec les notations de la définition précédente, on a les deux propriétés suivantes :*

$$\forall (x_i : \tau_i = e_i) \in \tilde{s} \Cap x,\ \mathcal{C}, \mathcal{E}, \Gamma, \tilde{s} \Cap x \vdash \mathcal{B}_s(x_i) : \tau_i$$

$$\exists \tau,\ \mathcal{C}, \mathcal{E}, \Gamma, \tilde{s} \Cap x \vdash \mathcal{B}_s(x) : \tau$$

Preuve. Par définition du typage de s (règle WELL-TYPED-SPEC) et de $\Cap$. En effet, s étant bien formée, tous les corps des méthodes sont typables dans l'environnement $\mathcal{C}, \mathcal{E}, \Gamma, \tilde{s}$. Or par définition de $\Cap$, on n'utilise pour ce faire que des méthodes de $\tilde{s} \Cap x$.

Plus précisément, il s'agit de montrer que toutes les applications de la règle SELF-CALL restent correctes dans le nouvel environnement. Nous allons donc examiner les cas où cette règle peut s'appliquer, ces cas étant les sous-expressions pouvant contenir un appel à une méthode de **self**.

Dans $\mathcal{T}_s(x_i)$ Avant de vérifier que les corps des def-dépendances sont typables, il convient de s'assurer que les types proposés sont eux-mêmes bien formés. Des appels à une méthode y de **self** peuvent intervenir sous deux formes : la présence du type support, ou dans le cas de propriétés, des appels à des méthodes calculatoires. Dans tous les cas, d'après la définition de $\mid x \mid$, $y \in \mid x \mid$, et on a donc $y : \mathcal{T}_s(y) \in \tilde{s} \Cap x$.

Dans $\mathcal{B}_s(x_i)$ Par définition de $\Cap$, ce cas ne peut se présenter que si $x_i <_s^{def} x$. Dans ce cas, toutes les dépendances de x_i se trouvent dans $\mid x \mid$. De plus, si y est une def-dépendance de x_i, on a $y <_s^{def} x_i <_s^{def} x$, et par définition de $\Cap$, la définition de y est présente dans l'environnement.

Dans τ De même, les decl-dépendances du type de x lui-même sont présentes par définition dans l'environnement, et τ est bien formé.

Dans $\mathcal{B}_s(x)$ Enfin, pour les appels à une méthode y dans le corps de x, il y a deux possibilités. Soit on a une decl-dépendance, et par définition de $\mid x \mid$, y est bien présente dans l'environnement avec son type, soit on a une def-dépendance, et avec la règle correspondante de $\Cap$, y est présente avec sa définition dans l'environnement. □

Notation 11 *Dans la suite, on notera $\mathcal{C}, \mathcal{E}, \Gamma \vdash \tilde{s} \Cap x$ la première propriété du lemme précédent (bonne formation du contexte de typage de $\mathcal{B}_s(x)$.*

Dans le reste du chapitre, on se concentre essentiellement sur la traduction des générateurs de méthodes en Coq, en supposant en particulier que les preuves de terminaison ont été fournies (et sont correctes) pour toutes les méthodes récursives. Comme pour l'analyse de la traduction en Ocaml, on commence par donner la traduction d'un contexte valide en Coq, avant de s'intéresser au typage de la traduction des expressions de base dans un contexte donné, puis aux générateurs de méthodes eux-mêmes, et enfin à leur utilisation.

7.2 Traduction de l'environnement Foc

Contrairement à Ocaml, un contexte Coq contient à la fois des types et des termes. En particulier, les éléments définis de Σ vont garder leur définition lors de la traduction de l'environnement. On va donc devoir montrer que cette définition est correcte, c'est à dire qu'un environnement Foc bien formé est traduit en un contexte Coq bien formé. Pour cela, il convient de formaliser la notion d'environnement Foc bien formé et de montrer que si une espèce s ou une collection c est bien typée dans un environnement bien formé donné, alors on peut les ajouter à cet environnement.

Définition 70 (Environnement FOC bien formé) *Soit $\mathcal{C}, \mathcal{E}, \Sigma, \Gamma$ un environnement* FOC*. Un tel environnement est dit bien formé, ce qu'on notera* $\mathrm{WF}(\mathcal{C}, \mathcal{E}, \Sigma, \Gamma)$ *si on peut dériver ce jugement à partir des règles suivantes :*

$$\mathrm{WF}(\mathcal{C}, \emptyset, \emptyset, \Gamma) \qquad \frac{\mathrm{WF}(\mathcal{C}, \mathcal{E}, \Sigma, \Gamma) \qquad s \notin \Pi \qquad \mathrm{WF}(\mathcal{C}, \mathcal{E}, \{x_i \mapsto (\tau_i, e_i)\}, \Gamma)}{\mathrm{WF}(\mathcal{C}, \mathcal{E} + s : \{x_i : \tau_i = e_i\}, \Gamma)}$$

$$\frac{\mathrm{WF}(\mathcal{C} + c : I, \mathcal{E} + s : e, \Sigma, \Gamma)}{\mathrm{WF}(\mathcal{C}, \mathcal{E} + s : (c \ \textbf{\textit{is}} \ I)e, \Gamma)} \qquad \frac{\mathrm{WF}(\mathcal{C}, \mathcal{E} + s : e, \Sigma, \Gamma + x : \tau)}{\mathrm{WF}(\mathcal{C}, \mathcal{E} + s : (x \ \textbf{\textit{in}} \ \tau)e, \Gamma)}$$

$$\frac{\forall i, \mathcal{C}, \mathcal{E}, \{x_j : \tau_j = e_j\}, \Gamma \vdash e_i : \tau_i}{\mathrm{WF}(\mathcal{C}, \mathcal{E}, \{x_j : \tau_j = e_j\}, \Gamma)}$$

Lemme 32 (Bonne formation des définitions) *Soit $\mathcal{C}, \mathcal{E}, \Sigma, \Gamma$ un environnement* FOC *bien formé. s une espèce définie par defspec telle que*

$$\mathcal{C}, \mathcal{E}, \Sigma, \Gamma \vdash \textbf{\textit{species}} \ s \ \mathit{defspec} : I$$

Alors $\mathcal{C}, \mathcal{E} + s : I, \Sigma, \Gamma$ est bien formé.

De même, si c est une collection définie par imp, telle que

$$\mathcal{C}, \mathcal{E}, \Sigma, \Gamma \vdash \textbf{\textit{collection}} \ c \ \mathit{imp} : I$$

Alors $\mathcal{C} + c : I, \mathcal{E}, \Sigma, \Gamma$ est bien formé.

Preuve. Trivial par cas à partir des règles de typages du chapitre 3. □

Notation 12 *Dans les règles ci-dessous, la notation $\{x : \tau\}$ est utilisée comme du sucre syntaxique pour signaler qu'on place dans l'environnement non seulement le type d'enregistrement correspondant, mais aussi tous les projecteurs.*

Définition 71 (Traduction de l'environnement FOC en COQ) *Soit $\mathcal{C}, \mathcal{E}, \Sigma, \Gamma$ un environnement* FOC *bien formé. On définit l'environnement* COQ *correspondant Γ' de la manière suivante :*

$$\frac{\mathcal{C}(c) = \{x_i : \tau_i\} \qquad \mathcal{C}(c) = c' \qquad \nabla(c) = s}{(c' : s) \in \Gamma'}$$

$$\frac{\mathcal{E}(s) = (a_j \,.\, t_j)\{x_i : \tau_i = e_i\} \qquad \forall j, \ t_j \rightarrowtail t'_j \qquad \forall i, \ \tau_i \rightarrowtail \tau'_i}{(s := (a_j : t'_j)\{\mathbf{E}(s, x) : \tau'_i\} : \mathit{Type}) \in \Gamma'}$$

$$\frac{\Sigma(x) = (\tau, e) \qquad \tau \rightarrowtail \tau' \qquad e \rightarrowtail^e e'}{(x := e' : \tau) \in \Gamma'} \qquad \frac{\Sigma(x) = (\tau, \bot) \qquad \tau \rightarrowtail \tau'}{(x : \tau') \in \Gamma'}$$

$$\frac{\Gamma(x) = \forall \alpha_1, ..., \alpha_n \tau \qquad \mathcal{V} + \alpha_i \mapsto A_i \vdash \tau \rightarrowtail \tau' \qquad A_i \text{ fraîches}}{x : (A_1, ..., A_n)\tau' \in \Gamma'}$$

Dans la suite, on notera $\Omega \rightarrowtail \Gamma'$ la traduction d'un environnement Foc *en* Coq.

7.3 Typage des expressions de base

Lemme 33 (Bonne formation du contexte) *Soit $\mathcal{C}, \mathcal{E}, \Sigma, \Gamma$ un environnement* Foc *bien formé, et Γ' l'environnement* Coq *correspondant. Alors Γ' est bien formé.*

De plus, si e est une expression Foc *bien typée, telle que $\mathcal{C}, \mathcal{E}, \Sigma, \Gamma \vdash e : \tau$, alors, avec $e \rightarrowtail e'$ et $\tau \rightarrowtail \tau'$, il vient $Gen(\Gamma, \tau) = \forall \alpha_1, ..., \alpha_n \tau$. On a alors $\Gamma' \vdash e' : (A_1, ..., A_n)\tau'$, avec les A_i variables fraîches telles que $\mathcal{V}(\alpha_i) = A_i$.*

De même, si p est une proposition Foc *bien typée dans le contexte Ω, si $p \rightarrowtail^p p'$ et $\Omega \rightarrowtail \Gamma'$ alors $\Gamma' \vdash p' : Prop$*

Preuve. les deux parties du lemme doivent être montrées simultanément par induction sur le nombre d'expressions et de propriétés présentes dans Σ et $\mathcal{E}$. S'il n'y en a aucune, Γ' est bien formé. De plus, si e a le type τ dans l'environnement bien formé $\mathcal{C}, \mathcal{E}, \Sigma, \Gamma$, on va montrer par induction sur la dérivation de typage de e que e', sa traduction en Coq a le type τ' (avec $\tau \rightarrowtail \tau'$) dans l'environnement Γ'. Dans le cas des propriétés, on montrera en outre que lorsqu'on on ajoute une propriété dans Γ', celui-ci reste bien formé.

Variable Si e est une variable x. Alors $\Gamma(x) = \forall \alpha_1, ..., \alpha_m, \tau_1$, et $\Gamma'(x) = (A_1, ..., A_m)\tau'_1$. La traduction est de la forme $(x \overbrace{?...?}^{m\ fois})$. De plus, τ est une instance du schéma de type ci-dessus, et Coq va pouvoir inférer les instances des A_i dans τ'_1 pour retrouver τ'.

Abstraction Si e est de la forme $\mathbf{fun}(x \mathbf{\ in\ } \tau_1)e_2$, τ est de la forme $\tau_1 \rightarrow \tau_2$, avec $e_2 : \tau_2$. En notant $e_2 \rightarrowtail^e e'_2$, $\tau_1 \rightarrowtail \tau'_1$ et $\tau_2 \rightarrowtail \tau'_2$, il vient alors, par hypothèse d'induction, $\Gamma' + x : \tau'_1 \vdash e'_2 : \tau'_2$ et e' a bien le type $\tau'_1 \rightarrow \tau'_2$.

Application Si e est de la forme $f(e_1, ..., e_n)$, avec $f \rightarrowtail^e f'$ et $\forall i, e_i \rightarrowtail^e e'_i$, f' et les e'_i sont, par induction, bien typés dans Γ', et f ayant le type $\tau_1 \rightarrow \ldots \rightarrow \tau_n$, l'application est bien typée en Coq.

Liaison locale Si e est de la forme $\mathbf{let}\ x\ \mathbf{in}\ \tau_1 = e_1\ \mathbf{in}\ e_2$, avec $e_1 \rightarrowtail^e e'_1$ et $e_2 \rightarrowtail^e e'_2$, e'_1 est bien typé avec le type τ'_1, traduction de τ_1, dans Γ'. Donc $\Gamma' + x : \tau'_1$ est un environnement bien formé, dans lequel on peut typer e'_2 avec le type τ'.

Appel de Méthode Si e est de la forme $c!m$, il faut distinguer deux cas, selon que c est **self** ou un nom de collection existant. Si c est une collection existante, alors avec la règle METH-CALL, c possède la méthode m, de type τ, et dans Γ', c est un enregistrement dont un projecteur est m. De plus, ce projecteur attend autant de paramètre que $\nabla(c)$, et $(\mathbf{E}(s,m) \overbrace{?...?}^{nfois} c)$ a bien le type τ.

Si $c =$ **self**, d'après SELF-CALL, la méthode m existe dans Σ, et on a donc $(m : \tau) \in \Gamma'$.

À partir des expressions de base, on peut montrer que la traduction des propriétés est correcte dans Γ'. Soit p une propriété bien typée dans un environnement FOC bien formé, telle que $p \rightarrowtail p'$.

Expressions Si p est une expression e de type `bool`, avec ce qui précède e' a le type `bool` en COQ et $(Is\_true\ e')$ est bien typé de type *Prop*. De même, si e est de type *Prop*, e' est directement de type *Prop* en COQ.

Connecteurs logiques de base Trivial par induction sur la dérivation de typage de la propriété.

Quantificateurs Dans les deux cas (quantificateur universel ou existentiel), on a une propriété p bien typée dans le contexte $\mathcal{C}, \mathcal{E}, \Sigma, \Gamma + x : \tau$. Le contexte $\Gamma' + x : \tau'$ est bien formé, et p' y a bien le type *Prop*.

Une fois que l'on a montré que la traduction des expressions et des propriétés FOC est bien typée dans un environnement Γ' ne comportant pas d'expression ou de propriétés, on peut terminer la preuve par induction sur le nombre d'expressions et de propriétés de Γ' : on ajoute les expressions une par une, en utilisant l'hypothèse de récurrence pour s'assurer que Γ' reste bien formé (si l'environnement FOC de départ est bien formé). □

7.4 Typage des types d'enregistrements

Lorsqu'on traduit une espèce FOC en COQ, on commence par fournir le type d'enregistrement correspondant (voir la def.57). Nous allons montrer que dans le cas des espèces atomiques, ces types d'enregistrements sont bien typés en COQ.

Lemme 34 (Bonne formation des types d'enregistrement)
Soit s une espèce atomique bien typée et bien formée en FOC*, et $\tilde{s}$ sa mise en forme normale. Alors, avec r le type d'enregistrement tel que $s \rightarrowtail$ **Record** $s :$ Type $:= r$, r est un type admissible en* COQ*. De plus, $\forall x \in \mathcal{N}(s)$ tel que $\mathcal{T}_s(x) = \tau$, $\mathbf{E}(s,x)$ est un champ de r de type τ', avec $\tau \rightarrowtail^r_s \tau'$.*

Preuve. Par définition de la forme normale de s, toute méthode x dépend uniquement des méthodes placées avant x dans $\tilde{s}$. De plus, si $\tilde{s} = \{x_i : \tau_i = e_i\}_{1 \leq i \leq n}$, alors pour tout $j \leq n$, τ_j est bien formé dans l'environnement $\mathcal{C}, \mathcal{E}, \tilde{s}, \Gamma$, d'après la règle WELL-TYPED-SPEC. On va montrer par induction sur n que $\{x_i : \tau_i = \bot\}_{1 \leq i < j}$ est un environnement bien formé, et que τ_j est bien formé dans cet environnement. Dans le cas où $j = 1$, l'environnement vide est trivialement bien formé, et τ_j ne dépend d'aucune méthode de $\tilde{s}$ et est donc bien formé.

Sinon, par hypothèse d'induction $\{x_i : \tau_i = \bot\}_{1 \leq i < j-1}$ est bien formé, et τ_{j-1} est bien formé dans cet environnement : $\{x_i : \tau_i = \bot\}_{1 \leq i < j}$ est bien formé. De plus, les dépendances de τ_j sont incluses dans cet environnement. Comme par définition d'une espèce bien formée τ_j ne peut porter de def-dépendance, τ_j est bien formée dans cet environnement.

On obtient donc bien un type d'enregistrement correct à partir de $\tilde{s}$. La définition de $\rightarrowtail^r_s$ permet d'en déduire la condition sur les champs de r. □

Dans le cas des espèces paramétrées, il suffit de montrer que les types des paramètres sont bien formés dans le contexte COQ correspondant à celui de l'espèce FOC. En effet, les paramètres de l'espèce sont directement traduits en paramètres du type d'enregistrement, et le typage de ce dernier se fait donc directement dans l'environnement approprié.

Lemme 35 *Soient s une espèce définie par*

species $s(c_1$ **.** $\tau_1, \ldots, c_k$ **.** $\tau_k)$ ...

et r le type d'enregistrement correspondant. Alors tous les paramètres de r ont des types valides et r est bien formé.

Preuve. Par induction sur le nombre de paramètres k. Si s n'a pas de paramètres, on est dans le cas du lemme précédent. Si s a $k+1$ paramètres, on raisonne par cas sur c_1.

Paramètre d'entité Si le premier paramètre est de la forme c_1 **in** τ_1, d'après la règle ENT-PRM le reste de l'espèce est bien typé dans l'environnement $\Gamma + c_1 : \tau$. Cet environnement se traduit en un environnement COQ bien formé, et on peut donc appliquer l'hypothèse de récurrence.

Paramètre de collection Si le premier paramètre est de la forme c_1 **is** $s(e_1, ..., e_n)$, d'après la règle COLL-PRM, le reste de l'espèce est bien typé dans l'environnement $\mathcal{C} + c_1 : \langle x_i : \tau_i \rangle$. Cet environnement se traduit en un environnement bien formé en COQ, et on peut appliquer l'hypothèse de récurrence. □

7.5 Les Générateurs de méthode

Après la définition de l'interface sous forme de type d'enregistrement, le **Chapter** correspondant à une espèce FOC définit les générateurs de méthodes nécessaires. Nous allons montrer que ces générateurs sont des termes COQ bien typés. Encore une fois, nous distinguerons le cas des espèces atomiques de celui des espèces paramétrées.

7.5.1 Espèces atomiques

La définition du générateur d'une méthode x dans une espèce atomique s conduit à donner à COQ un terme dans lequel les appels de méthode sont remplacés par des variables locales, qu'elles soient des lambda-abstractions ou des liaisons locales. Comme d'après le lemme 31 l'environnement $\tilde{s} \Cap x$ est bien formé, il se traduit en un environnement bien formé en COQ. On peut alors utiliser le lemme 33 pour montrer que la traduction de $\mathcal{B}_s(x)$ elle-même est bien typée dans un tel environnement.

Théoreme 9 (Bonne définition des générateurs de méthode) *Soit s une espèce bien formée et bien typée, et $x \in \mathcal{D}(s)$, telle que $x \upharpoonleft s = s$. Alors $\mathbf{G}(s,x)$ est un terme COQ bien typé. De plus, si on note $\{x_i\}$ les noms des méthodes abstraites dans l'univers visible de x ($\mathcal{H}(s' \Cap x) = \{x_i\}$), et $\mathcal{T}_s(x_i) \rightarrowtail t_i$, on a*

$$\Gamma' \vdash \mathbf{G}(s,x) : (x_1 : t_1) \ldots (x_n : t_n)t$$

Preuve. Ce théorème se montre par induction sur l'arbre d'héritage et le nombre de méthodes dont x def-dépend dans l'espèce s. Plus précisément, la démonstration de ce théorème reprend celle du théorème 10, qui garantit que les générateurs de méthodes sont toujours utilisés avec des arguments corrects. En effet, d'après la définition de $\mathbf{G}$, les def-dépendances de x se traduisent par des applications de générateurs de méthode précédemment définis.

Si $\tilde{s} \Cap x$ ne contient pas de def-dépendance, d'après le lemme 31, l'environnement $\mathcal{C}, \mathcal{E}, \Gamma, \tilde{s} \Cap x$ est bien formé. De plus, on a $\mathcal{C}, \mathcal{E}, \Gamma, \tilde{s} \Cap x \vdash \mathcal{B}_s(x) : \mathcal{T}_s(x)$. Donc, avec $[\![\tilde{s} \Cap x]\!]_s^x = \gamma$, $\mathcal{T}_s(x) \rightarrowtail t$ et $\mathcal{B}_s(x) \rightarrowtail e$, on a

$$\Gamma' + \gamma \vdash e : t$$

d'après le lemme 33. D'après la définition de $\mathbf{G}$, on en déduit que le type de $\mathbf{G}(s,x)$ est de la forme

$$t_1 \to \ldots \to t_n \to t$$

avec $\gamma = x_i : t_i$. □

7.5.2 Espèces paramétrées

Le fait de transformer les paramètres FOC en variables du **Chapter** COQ correspondant permet de s'assurer qu'ils seront présents dans l'environnement dans lequel on type le générateur de méthode. Par ailleurs, on a déjà vu dans le lemme 35 que les types des paramètres sont bien formés dans l'environnement du **Chapter** COQ correspondant. Le théorème précédent s'étend donc trivialement au cas des espèces paramétrées, puisque les paramètres sont présents dans le contexte COQ dans lequel on type les générateurs.

7.6 Contexte d'utilisation des générateurs

Une fois les générateurs créés, il reste à s'assurer qu'on les applique correctement. En particulier, il faut montrer que la définition locale d'une méthode x dans le contexte d'une espèce s (définition 62) fournit au générateur de méthode utilisé les arguments (correspondant aux decl-dépendances de x dans s).

Encore une fois, il faut distinguer les espèces atomiques des espèces paramétrées. Pour ce dernier cas, la situation est un peu plus compliquée que dans la création des générateurs de méthode. Plus précisément, comme on l'a vu dans la définition 68, il faut passer à $\mathbf{G}(s, x)$ les instantiation des paramètres que le générateur utilise. Deux points supplémentaires sont donc à vérifier. Il faut d'une part que $\mathcal{U}_s(x)$ sélectionne bien les paramètres utilisés dans $\mathcal{B}_s(x)$. D'autre part, le calcul de l'instantiation des paramètres $Inst_s(x)$ doit aboutir à une expression ayant le type attendu dans l'environnement correspondant à celui de l'espèce s. Ces deux points sont détaillés dans la section 7.6.2.

7.6.1 Dans une espèce atomique

Considérons tout d'abord le cas d'une espèce atomique. Le résultat important dans ce cas est que si on applique à $\mathbf{G}(s, x)$ les expressions représentant les méthodes dont x decl-dépend, on obtient une expression de type $\mathcal{T}_s(x)$.

Théoreme 10 (Génération d'une méthode)
Soit s une espèce atomique bien typée et bien formée, et x une méthode définie dans s. Alors, avec $\mathcal{T}_s(x) \rightarrowtail \tau'$, $\mathcal{L}_s(x)$ est bien formée et de type τ'.

Preuve. Le cas où x est seulement déclarée dans $\tilde{s}$ est trivial : $\mathcal{L}_s(x)$ est une variable du **Chapter** courant, et on a déjà vu que $\mathcal{T}_s(x)$ était un type admissible.

Si x est défini, on procède par induction sur la place de x dans la forme normale de s. On note s' l'espèce d'origine de x ($x \upharpoonleft s = s'$). D'après le

théorème 9, $\mathbf{G}(s', x)$ est une fonction dont le type de retour est τ'. Il reste à montrer qu'on l'applique aux bons arguments.

Si x ne dépend d'aucune autre méthode, alors $\mathbf{G}(s', x)$ a déjà le type τ', et $\mathcal{L}_s(x)$ aussi. Sinon, par définition de la forme normale, toutes les méthodes x_i dont x dépend sont placées avant x dans $\tilde{s}$. D'après l'hypothèse d'induction, on a donc dans le contexte COQ de définition de x, les variables x_i avec le type τ_i, traduction de $\mathcal{T}_s(x_i)$. Avec le théorème 9, les arguments de $\mathbf{G}(s', x)$ sont un sous-ensemble des x_i. Comme d'après la définition de $\mathcal{L}_s(x)$ on prend ses arguments dans l'ordre dans lequel on le trouve dans $norm(s')$, le terme $\mathcal{L}_s(x)$ est bien typé, de type τ'. □

7.6.2 Les paramètres

Il reste maintenant à étendre le résultat du théorème 10 au cas des espèces paramétrées. Pour cela, on va vérifier deux choses :

1. que les paramètres utilisés dans la traduction COQ sont bien ceux qui sont calculés par $\mathcal{U}_s(x)$.
2. que la traduction des instantiations de paramètre en COQ donne bien des termes du type attendu par $\mathbf{G}(s, x)$.

Lemme 36 *Soient s une espèce définie par*

species $s(c_1 \text{ . } \tau_1, \ldots, c_{p_f} \text{ . } \tau_{p_f})$ ...

et $x \in \mathcal{D}(norm(s))$ une méthode telle que $x \upharpoonleft s = s$. Alors,

$$\forall p \leq p_f,\ c_p \in \mathcal{U}_s(x) \iff c_p \text{ apparaît dans } \mathbf{G}(s, x)$$

Preuve. Pour démontrer la première implication, on procède par induction sur les règles de $\mathcal{U}_s(x)$. Soit $c_p \in \mathcal{U}_s(x)$. D'après la définition 66, il y a quatre cas possibles :

- $c_p \in \mathcal{U}_s(\mathcal{B}_s(x))$. Par définition, c_p apparaît dans le générateur de méthode de x.
- $\exists y \in |\ x\ |$, $c_p \in \mathcal{U}_{\mathcal{T}_s(x)}()$. Dans ce cas, c_p apparaît dans le type d'un des arguments ou d'une des variables locales de $\mathbf{G}(s, x)$.
- $\exists y <_s^{def} x$, $c_p \in \mathcal{U}_s(\mathcal{B}_s(y))$. Si $y \upharpoonleft s = s$, c_p apparaît dans $\mathbf{G}(s, y)$, qui lui même est utilisé dans $\mathbf{G}(s, x)$. Sinon c_p apparaît dans les arguments fournis à $\mathbf{G}(y \upharpoonleft s, y)$.
- $\exists c_{p'} \text{ . } i_{p'}$ un paramètre de s tel que $c_p \in \mathcal{U}_s(i_{p'})$, et $c_{p'} \in \mathcal{U}_s(x)$. Par hypothèse d'induction, $c_{p'}$ apparaît dans $\mathbf{G}(s, x)$, donc son type aussi, et c_p apparaît dans $\mathbf{G}(s, x)$

Réciproquement, si c_p apparaît dans $\mathbf{G}(s, x)$, on a un des cas suivants :

- c_p apparaît dans $\mathcal{B}_s(x)$
- $\exists y \in s \Cap x$, tel que $y \upharpoonleft s = s$ et c_p apparaît dans $\mathcal{B}_s(y)$: on a bien $c_p \in \mathcal{U}_s(x)$.

- $\exists y \in s \Cap x \backslash \mathcal{H}(s \Cap x)$, tel que $y \upharpoonleft s = s' \neq s$, et c_p apparaît dans un des arguments de $\mathbf{G}(s', y)$. D'après la remarque 6, c_p apparaît donc dans $\mathcal{B}_s(y)$. De plus, par définition de $s \Cap x$, $y <_s^{def} x$, donc $c_p \in \mathcal{U}_s(x)$
- $\exists y \in \mathcal{H}(s \Cap x)$ tel que c_p apparaisse dans le type de y : par définition, $y \in \mid x \mid$, et $c_p \in \mathcal{U}_s(x)$.
- c_p est présent dans un type d'un paramètre $c_{p'}$ qui apparaît lui même dans $\mathbf{G}(s, x)$. D'après la règle PRM, $c_p \in \mathcal{U}_s(x)$.

□

Lemme 37 *Soient s une espèce et $x \in \mathcal{D}(norms)$ une méthode telle que $x \upharpoonleft s = s' \neq s$. Alors, si s' est définie par*

species $s'(c_1 \cdot \tau_1, \ldots, c_{p_f} \cdot \tau_{p_f})$ ...

On note $Inst_s(x) = \{e_{p_i}\}$. $\forall p \leq p_f$, tel que $c_p \in \mathcal{U}_{s'}(x)$, e_p a le type $\tau_p[c_p \leftarrow e_p]$.

Preuve. Par induction sur le nombre d'espèces entre s et s' sur le graphe d'héritage. Si s' est un parent direct de s, d'après les règle PRM-INST, e_p a le type c_p. Sinon, soit $h = \mathbb{I}_s(x)$. s_h est définie par

species `s_h`$(c'_1 \cdot \tau'_1, \ldots, c'_{p_f} \cdot \tau'_{p_f})$ ...

Dans s, ses paramètres sont instanciés par $e'_1, ..., e'_{p_f}$. Par hypothèse d'induction, l'expression correspondant à c_p dans $Inst_{s_h}(x) = \{e^h_{p_i}\}$ est de type $\tau_p[c_p \leftarrow e^h_p]$. Or d'après la définition de $Inst$, $e_p = e^h_p[c'_p \leftarrow e'_p]$. e_p a donc le type $\tau_p[c_p \leftarrow e^h_p][c'_p \leftarrow e'_p]$, soit encore

$$\tau_p[c_p \leftarrow e^h_p[c'_p \leftarrow e'_p]] = [c_p \leftarrow e_p]$$

□

Avec ces deux lemmes, on peut montrer que le résultat de la section précédente s'étend au cas des paramètres :

Théoreme 11 (Génération d'une méthode) *Soit s une espèce bien typée, et $x \in \mathcal{D}(s)$. Alors, avec $\mathcal{T}_s(x) \rightarrowtail \tau'$, $\mathcal{L}_s(x)$ est bien formée et de type τ'.*

Preuve. Soient $s' = x \upharpoonleft s$, $\{c_{p_i}\} = \mathcal{U}_s(x)$, et $x_i = \mathcal{H}(x)$. D'après le lemme 36, le type de $\mathbf{G}(s, x)$ est de la forme

$$(c_{p_1} : \tau_{p_1}) ... (c_{p_k} : \tau_{p_k})$$

$$(x_1 : \mathcal{T}_{s'}(x_1)) ... (x_n : \mathcal{T}_{s'}(x_n))\tau'$$

Le lemme 37 indique que les arguments correspondant aux c_{p_i} ont le type attendu (une fois les substitutions des paramètres précédents faites). On peut alors conclure comme dans le théorème 10. □

Chapitre 8

La compilation vers OCaml revisitée : Des enregistrements en OCaml

8.1 La traduction vers des enregistrements

Comme en Coq, il est possible d'envisager une traduction vers OCaml qui utilise des enregistrements et non des classes et des objets. Pour cela, on se repose également sur des *générateurs de méthode* et des *générateurs d'enregistrement*, qui seront utilisés lors de la création des collections. Pour créer ces générateurs, on utilise la forme normale des espèces calculée lors de l'analyse du programme. L'intérêt de cette approche est justifié par le fait qu'un appel de méthode est plus coûteux que la sélection du champ d'un enregistrement : dans un programme Foc, qui contient de nombreux appels de méthodes, on peut espérer un gain d'efficacité. Ainsi, le source ocaml suivant :

```
class obj =
object
  method m = "hello"
end;;

let o = new obj in print_endline o#m;;
```

produit près d'une centaine d'instructions en bytecode OCaml (figure 8.1). En particulier, on constate que la méthode `m` est bien construite comme une clôture prenant en argument l'objet sur lequel elle est appelée, ce qui n'est pas nécessaire dans le cas particulier de Foc.

A l'inverse, si on définit un `record` équivalent,

FIG. 8.1 – un appel de méthode en bytecode

```
   branch L2
L3:envacc 1
   push
   acc 1
   push
   getglobal Camlinternal00!
   getfield 17
   appterm 2, 3
L4:const "hello"
   return 1
L1:closure L4, 0
   push
   envacc 1
   push
   acc 2
   push
   getglobal Camlinternal00!
   getfield 6
   apply 3
   acc 0
   closure L3, 1
   return 1
L2:getglobal Camlinternal00!
   getfield 0
   push
   const "m"
   push
   acc 1
   apply 1
   push
   const 3
   ccall alloc_dummy, 1
   push
   const [0: "m" 0a]
   push
   getglobal Camlinternal00!
   getfield 10
   apply 1
   push
   acc 2
   closure L1, 1
   push
   acc 1
   push
   acc 1
   apply 1
   push
   acc 2
   push
   getglobal Camlinternal00!
   getfield 11
   apply 1
   acc 2
   push
   acc 2
   push
   acc 2
   makeblock 3, 0
   pop 3
   push
   acc 1
   ccall update_dummy, 2
   const 0a
   push
   acc 1
   getfield 0
   apply 1
   push
   acc 0
   push
   acc 3
   getmethod
   apply 1
   push
   getglobal Pervasives!
   getfield 29
   apply 1
   pop 1
   acc 0
   makeblock 1, 0
   pop 3
   setglobal Class!
```

```
type obj = { m : string };;
let f = { m = "hello" } in print_endline f.m
```

---

on obtient un code beaucoup plus concis (figure 8.2), et *a priori* nettement plus rapide. Par ailleurs, cette traduction a le mérite de faire apparaître une correspondance forte entre les composantes "calcul" (traduction en OCAML) et "certification" (traduction en COQ) de FOC.

Comparaison avec les codages antérieurs du projet FOC Lors des premiers développements du projet FOC, différents codage en OCAML des espèces et des collections ont été testés. En particulier, le modèle utilisant les objets d'OCAML a directement inspiré la traduction en OCAML telle qu'elle est décrite dans le chapitre 4. Cependant, une autre approche utilisant les modules et les foncteurs a été tentée [BHMR98, BHR99]. L'idée de base de cette représentation repose sur la distinction entre les méthodes définies et

FIG. 8.2 – sélection d'un champ d'enregistrement en bytecode

```
const [0: "hello"]                getfield 29
push                              apply 1
acc 0                             pop 1
getfield 0                        makeblock 0, 0
push                              setglobal Rec!
getglobal Pervasives!
```

déclarées de chaque espèce. Une espèce va être vue comme un foncteur qui prend un argument un module contenant les méthodes encore déclarées et renvoie un module contenant toutes les méthodes. de l'espèce. Par exemple, un setoide serait représenté par les structures suivantes :

```
module type Setoide_virtuel=
sig
  type t
  val egal : t → t → bool
  val element : t
end

module type Setoide=
sig
  type t
  val egal : t → t → bool
  val element : t
  val different : t → t → bool
end

module Creer_setoide = functor (S : Setoide_virtuel) →
struct
  type t = S.t
  let egal = S.egal
  let element = S.element
  let different x y = not (egal x y)
end
```

Le premier type de module contient la liste des méthodes déclarées dans l'espèce `setoide`. On trouve ensuite le type de module reprenant toutes les méthodes de `setoide`, qu'elles soient définies ou non. Enfin, un foncteur permet de créer un setoide complet à partir de la donnée d'un setoide virtuel, en donnant la définition de `different` à partir de celle de `egal`.

De même, l'héritage se traduit par un foncteur prenant en argument

l'espèce parent (complète). Ainsi, l'espèce des monoides serait représentée de la manière suivante :

```
module type Monoide_virtuel=
sig
  type t
  val un : t
  val mult : t
end

module type Monoide =
sig
  type t
  val egal : t → t → bool
  val element : t
  val different : t → t → bool
  val un : t
  val mult : t
end

module Creer_monoide =
 functor (M : Monoide_virtuel) →
  functor (S : Setoide with type t = M.t) →
   struct
    type t = M.t
    let egal = S.egal
    let different = S.different
    let un = M.un
    let mult = M.mult
    let element = un
end
```

la première signature reprend les méthodes déclarées introduites par l'espèce `monoide`. La seconde, par contre, reprend toutes les méthodes de `monoide`, dont celles qui sont héritées de `setoide`. Enfin, le foncteur `Creer_monoide` prend en argument deux modules. Le premier, `M`, contient les méthodes déclarées de `Monoide_virtuel`, et le second, `S`, est un `Setoide` complet, ayant le même type support que `M`. On crée alors un `Monoide`, en reprenant les fonctions correspondante de `M` ou de `S`, et en définissant `element`.

Héritage multiple L'héritage multiple peut se traduire par autant d'arguments du foncteur qu'il y a d'espèces parents. Cependant, le modèle ne peut traiter le cas de dépendances mutuelles entre deux espèces parents : si on a une espèce s_1 définissant une méthode `m_1` dépendant d'une méthode déclarée `m_2`, et une espèce s_2 définissant `m_2`, et une méthode `m_3` dépendant

de la méthode (déclarée dans s_2), `m_1`, et qu'on veut hériter de s_1 et s_2, il sera impossible de conserver les liens entre les méthodes communes aux deux structures.

Liaison tardive Lorsqu'on "hérite" des fonctions d'un module `M`, celles-ci utilisent les définitions que les méthodes dont elles dépendent avaient dans `M`. On a ici le même problème qu'avec une traduction "naïve" en Coq se reposant uniquement sur le mécanisme d'abstraction des `Sections`. Les méthodes définies d'une espèce donnée ne sont abstraites que par rapport aux méthodes déclarées de cette même espèce. Seuls les générateurs de méthode semblent à même de garantir le choix de la définition la plus récente de chaque méthode dont dépend une définition.

8.1.1 Définitions de méthodes

Dans les espèces de base, les définitions de méthode se font comme en Coq, à l'aide des générateurs de méthodes. On ne donne pas ici la définition de la traduction des générateurs de méthode vers Ocaml, qui est similaire (à ceci près que les corps des méthodes sont traduits en Ocaml et pas en Coq) à celle du chapitre précédent.

8.1.2 Paramètres

Les paramètres d'entités ne posent pas de problème. Une espèce complètement définie et paramétrée par un élément `x` d'un type τ peut être vue comme une fonction associant un enregistrement à tout élément de τ.

On pourrait penser que les paramètres de collection se comportent de la même manière, mais il serait alors difficile dans ce contexte de créer des méthodes engendrant une nouvelle collection, comme `updom` (voir p.82). En effet, cela obligerait à créer l'enregistrement à l'intérieur d'une structure mutuellement récursive, afin de pouvoir le passer en arguments aux constructeurs de collection. Cette approche n'est pas satisfaisante, car elle oblige à réécrire un générateur de méthode pour chaque nouvel enregistrement.

Une solution consiste à prolonger le calcul des dépendances des méthodes de **self** aux méthodes des paramètres de collections. En considérant une espèce de la forme

```
species prm_s(prm is p1) =
  ...
  let x = ... prm!meth ...;
  ...
```

on va analyser la définition `e1` de `x` pour repérer les appels aux méthodes de `prm` qu'elle contient, afin de les abstraire comme on le fait pour les méthodes

de l'espèce `s` elle-même. Ainsi, dans l'exemple ci-dessus, le générateur de méthodes correspondant à `x` sera de la forme

let `s_x` = **fun** `prm_meth` → ... `prm_meth` ...

Dès lors, pour construire une collection à partir d'une espèce paramétrée par `prm`, on n'aura plus besoin de prendre en argument un enregistrement, mais seulement les fonctions correspondant aux méthodes de `prm` effectivement utilisées dans l'espèce. Il n'y a donc plus besoin de créer un enregistrement à l'intérieur de la définition d'un générateur de méthodes.

8.2 Exemple de traduction

Comme dans les chapitres précédents, on peut prendre l'exemple des produits cartésiens pour présenter informellement les différents aspects de cette traduction.

```
module Setoide =
  struct
    type 'self
      stype =
     {
      different : ('self)→ ('self)→ bool;
      element : 'self;
      egal : ('self)→ ('self)→ bool;
      parse : (string)→ 'self;
      print : ('self)→ string;
     };;
    let different =
       fun (abst_egal : ('self)→ ('self)→ bool)→
       fun (x : 'self) → fun (y : 'self)→ not_b (abst_egal x y)
end
```

Ainsi qu'on peut le voir, chaque espèce est encapsulée dans un module OCAML. À l'intérieur de celui-ci, on trouve, comme en COQ, le type d'enregistrement `stype` correspondant à l'interface de l'espèce. Là encore, contrairement à la traduction en classe, on donne l'ensemble des méthodes présentes dans la forme normale de `setoide`. De plus, ce type d'enregistrement comporte une variable de type `'self`, représentant le type support.

Le reste du module est consacré aux générateurs de méthodes. Ici, seule la méthode `different` est définie dans `setoide`. À quelques variantes syntaxiques près, ce générateur est identique à la version COQ (p.106).

La traduction de l'espèce `monoide` s'effectue de la même manière que ci-dessus. Elle n'apporte rien de nouveau : on y définit simplement le type

d'enregistrement correspondant aux monoides, et le générateur de la méthode `element`. On ne s'attardera donc pas sur ce module :

```
module Monoide =
  struct
    type 'self stype = { ... }
    let element =
       fun (abst_multiplie : 'self-> 'self-> 'self)->
      fun (abst_un : 'self)-> abst_multiplie abst_un abst_un
end
```

Avec le module `Setoide_produit` ci-dessous, on aborde la traduction des espèces paramétrées :

```
module Setoide_produit =
  struct
    type ('p_a, 'p_b) self= ('p_a)* ('p_b)
    type ('self,'p_a, 'p_b)
      stype =
     {
      print : ('self)-> string;
      element : 'self;
      creer : ('p_a)-> ('p_b)-> 'self;
      different : ('self)-> ('self)-> bool;
      egal : ('self)-> ('self)-> bool;
      parse : (string)-> 'self;
     };;
```

Le type support est défini (et peut éventuellement être abstrait dans le type de module correspondant), et paramétré par `'p_a` et `'p_b`, types support de a et b. De même `stype` comporte trois paramètres de type.

On passe ensuite à la définition des générateurs de méthode :

```
    let print=
        fun (p_a_print : ('p_a)-> string)->
        fun (p_b_print : ('p_b)-> string)->
        fun (x : 'self)->
          "(" ^ (p_a_print (first x)) ^ "," ^
               (p_b_print (scnd x)) ^ ")"

    let element=
         fun (p_a_element : 'p_a)->
         fun (p_b_element : 'p_b)->
         fun (abst_creer : ('p_a)-> ('p_b)-> 'self)->
             abst_creer p_a_element p_b_element
```

```
let creer= fun (x : 'p_a)→ fun (y : 'p_b)→ crp x y

let egal=fun (p_a_egal : 'p_a-> 'p_a → bool) → (*. . .*)
```

Comme on peut le voir, en particulier dans la définition de `element`, les générateurs sont d'abord abstraits par rapport aux méthodes de a et b dont ils dépendent, puis, comme en Coq, par rapport aux méthodes de **self** dont ils decl-dépendent (il n'y a pas de def-dépendance dans les méthodes calculatoires, qui sont celles qui sont traduites en Ocaml).

Comme `setoide_produit` est une espèce complètement définie, on peut définir un *générateur de collection* :

```
let create =
    fun (p_a_print : ('p_a)→ string) →
    fun (p_a_egal : ('p_a)→ ('p_a)→ bool) →
    fun (p_a_element : 'p_a) →
    fun (p_b_print : ('p_b)→ string) →
    fun (p_b_egal : ('p_b)→ ('p_b)→ bool) →
    fun (p_b_element : 'p_b) →

    let l_parse = Basic_object.parse in
    let l_egal = egal p_a_egal p_b_egal in
    let l_different = Setoide.different l_egal in
    let l_creer = creer in
    let l_element=element p_a_element p_b_element l_creer in
    let l_print = print p_a_print p_b_print in
    {
      parse = l_parse;
      egal = l_egal;
      different = l_different;
      creer = l_creer;
      element = l_element;
      print = l_print;
    }
end
```

Comme on l'a dit précédemment, ce générateur ne prend pas en paramètre les enregistrements correspondant à a et b, mais les différentes méthodes de chacune de ces deux espèces nécessaires pour engendrer les méthodes de `setoide_produit`. Après ces abstractions, on trouve les variables locales correspondant aux différentes méthodes de l'espèce, obtenues à partir des générateurs de méthodes des modules précédents (comme pour `different`, qui est héritée de `setoide`), ou du module courant (comme dans le cas de `element`). Enfin, on crée l'enregistrement proprement dit, à partir des variables précédentes. Le type de cet enregistrement correspond à `stype` (plus précisément à (('a,'b) `self`, 'a, 'b) `stype`).

Le module `Monoide_produit` a une structure très proche de celle de `Setoide_produit`. La seule différence notable se trouve dans le générateur de collection `create` :

```
module Monoide_produit =
  struct
    type 'self= ('p_a)* ('p_b)
    type ('self,'p_a, 'p_b) stype = (* ... *)

    let create =
       fun (p_a_print : ('p_a)→ string) → (* ... *)
       fun (p_b_print : ('p_b)→ string) → (* ... *)
         let l_parse = Basic_object.parse in
         (* ... *)
         let l_element =
          Setoide_produit.element p_a_element p_b_element l_creer
         (* ... *)
         in
          { (* ... *) }
```

La définition de `l_element` permet de voir deux choses. En premier lieu, comme en Coq, la résolution des conflits en cas d'héritage multiple (`element` est défini à la fois dans `monoide` et dans `setoide_produit`) se fait par la sélection du générateur de méthode adéquat –ici `Setoide_produit.element`. Par ailleurs, ce générateur de méthode provient d'une espèce paramétrée : les deux premiers arguments du générateurs sont les méthodes `element` de a et de b dont on a besoin pour construire la méthode `element` de `setoide_produit`. Dans l'héritage, a et b sont instanciés par les paramètres de l'espèce `monoide_produit` elle-même, et les méthodes correspondantes sont donc des arguments de `create`.

La collection `entiers` est traduite en un module, semblable aux précédent, hormis la présence d'une nouvelle définition, `obj`, l'enregistrement qui correspond à la collection à proprement parler :

```
module Entiers =
  struct
  (* ... *)
  let obj = create
end
;;
```

Comme `entiers` implante une espèce atomique, `obj` n'est qu'un synonyme de `create`. Dans le cas de `entiers_2`, qui implante une espèce paramétrée, il faut appliquer à `Monoide_produit.create` les arguments correspondant aux

méthodes des paramètres a et b utilisées dans le corps de l'espèce. Ces paramètres étant instantiés par `entiers`, on utilise les champs de l'enregistrement `Entiers.obj` :

```
module Entiers_2 =
  struct
    type self= (Entiers.self)* (Entiers.self)
    let obj :
      (self,Entiers.self, Entiers.self)
        Monoide_produit.stype =
      Monoide_produit.create
        Entiers.obj.Entiers.print Entiers.obj.Entiers.egal
        Entiers.obj.Entiers.multiplie Entiers.obj.Entiers.un
        Entiers.obj.Entiers.element

        Entiers.obj.Entiers.print
        Entiers.obj.Entiers.egal Entiers.obj.Entiers.multiplie
        Entiers.obj.Entiers.un Entiers.obj.Entiers.element
  end
;;
```

8.3 Dépendances et paramètres

Nous avons déjà vu comment restreindre l'environnement Σ des méthodes d'une espèce dans la section 6.4.4. La section suivante montre comment étendre la notion d'univers visible définie précédemment pour Σ aux paramètres de collections, afin d'utiliser un minimum de méthodes de chacun d'entre eux.

8.3.1 Dépendances vis à vis des méthodes d'un paramètre

On va étudier ici les dépendances d'une méthode x d'une espèce s paramétrée par une collection c_p. Contrairement à ce qu'on avait précédemment, il n'y pas lieu ici de distinguer entre decl- et def- dépendances, puisque les paramètres sont abstraits : il ne peut y avoir que des decl-dépendances. De plus, il ne peut y avoir non plus de cycle de dépendance à ce niveau, les méthodes de c_p ne "connaissant" pas celles de s. Par contre, la notion d'univers visible reste importante : il faut considérer les appels aux méthodes de c_p dans les méthodes dont x def-dépend.

Définition 72 (Dépendances d'une méthode) *Soient $x \in \mathcal{D}(s)$ et c_p un paramètre de collection. On définit les dépendances de x vis à vis de c_p à partir de $\mathcal{B}_s(x)$:*

$$Deps(s, c_p)[y] \quad = \quad \emptyset$$

$$
\begin{array}{rcl}
Deps(s, c_p)\,[\textbf{let } y = e_1 \textbf{ in } e_2] & = & Deps(s, c_p)\,[e_1] \cup Deps(s, c_p)\,[e_2] \\
Deps(s, c_p)\,[\textbf{let rec } y = e_1 \textbf{ in } e_2] & = & Deps(s, c_p)\,[e_1] \cup Deps(s, c_p)\,[e_2] \\
Deps(s, c_p)\,[\lambda x.e] & = & Deps(s, c_p)\,[e] \\
Deps(s, c_p)\,[c!y] & = & \{y\} \\
Deps(s, c_p)\,[c'!y] & = & \emptyset \text{ (si } c' \neq c)
\end{array}
$$

$$
\begin{array}{rcl}
Deps(s, c_p)\,[p_1 \textbf{ or } p_2] & = & Deps(s, c_p)\,[p_1] \cup Deps(s, c_p)\,[p_2] \\
Deps(s, c_p)\,[p_1 \textbf{ and } p_2] & = & Deps(s, c_p)\,[p_1] \cup Deps(s, c_p)\,[p_2] \\
Deps(s, c_p)\,[p_1 \rightarrow p_2] & = & Deps(s, c_p)\,[p_1] \cup Deps(s, c_p)\,[p_2] \\
Deps(s, c_p)\,[\textbf{not } p] & = & Deps(s, c_p)\,[p] \\
Deps(s, c_p)\,[\textbf{all } y \textbf{ in } \tau, p] & = & Deps(s, c_p)\,[p] \\
Deps(s, c_p)\,[\textbf{ex } y \textbf{ in } \tau, p] & = & Deps(s, c_p)\,[p]
\end{array}
$$

On définit alors $Deps(s, c_p)\,[x]$ *comme* $\mathcal{U}_s(x)$*, à partir des règles suivantes :*

$$\textsc{Body}\quad \frac{y \in Deps(s, c_p)\,[\mathcal{B}_s(x)]}{y \in Deps(s, c_p)\,[x]} \qquad \textsc{Type}\quad \frac{y \in Deps(s, c_p)\,[\mathcal{T}_s(x)]}{y \in Deps(s, c_p)\,[x]}$$

$$\textsc{Def-dep}\quad \frac{z <_s^{def} x \qquad y \in Deps(s, c_p)\,[\mathcal{B}_s(z)]}{y \in Deps(s, c_p)\,[x]} \qquad \textsc{Univers}\quad \frac{z \in\mid x \mid \qquad y \in Deps(s, c_p)\,[\mathcal{T}_s(z)]}{y \in Deps(s, c_p)\,[x]}$$

$$\textsc{Prm}\quad \frac{\begin{array}{c} z \in Deps(s, c_{p'})\,[x] \qquad i_{p'} = s'(e_1, ..., e_{k-1}, c_p, e_{k+1}, ...) \\ \mathcal{E}s' = (c'_1 \textbf{ . } i'_1, ..., c'_k \textbf{ is } i'_k, ...) \qquad y \in Deps(s', c'_k)\,[z] \end{array}}{y \in Deps(s, c_p)\,[x]}$$

Si c_p *est un paramètre d'entité, on définit*

$$Deps(s, c_p)\,[x] = \{c_p\} \cap \mathcal{U}_s(x)$$

Cette nouvelle notion de dépendance aboutit à raffiner notre notion d'environnement minimal : on peut maintenant restreindre $\mathcal{C}$, pour que c_p n'apparaisse plus qu'avec les méthodes auxquelles $\mathcal{B}_s(x)$ fait appel.

Définition 73 (Interface minimale des paramètres) *Soient* c_p *un paramètre de l'espèce* s*, d'interface* $I_p = \langle x_i : \tau_i \rangle$*, et* x *une méthode de* s*. L'*interface minimale *de* c_p *vis à vis de* x*,* $\mathcal{I}_x(c_p)$ *est définie de la manière suivante :*

$$\mathcal{I}_x(c_p) = I_p \cap Deps(s, c_p)\,[x]$$

Lemme 38 (Correction de l'environnement de typage) *Avec les notations de la définition précédente, si on a* $\mathcal{C} + c_p : I_h \vdash \tilde{s} \Cap x$, *alors* $\mathcal{C} + c_p : \mathcal{I}_x(c_p) \vdash \tilde{s} \Cap x$

Preuve. Par définition de $\tilde{s} \Cap x$, les seules méthodes de c_p qui sont appelées dans $\tilde{s} \Cap x$ sont dans $\mathcal{I}_x(c_p)$. □

Dès lors, on peut procéder aux deux restrictions d'environnement vues précédemment pour obtenir un environnement de typage minimal dans lequel on peut typer $\mathcal{B}_s(x)$.

Théoreme 12 (Correction du typage avec interface minimale) *Avec les notations de la définition précédente, si on a* $\mathcal{C} + c_p : I_h, \mathcal{E}, \Sigma, \Gamma \vdash \mathcal{B}_s(x) : \tau$, *alors* $\mathcal{C} + c_p : \mathcal{I}_x(c_p), \mathcal{E}, \Sigma \Cap x, \Gamma \vdash \mathcal{B}_s(x) : \tau$

Preuve. Immédiat avec les lemmes 31 et 38 □

Comme on le verra dans la section 8.5, cet environnement minimal permet de définir un générateur de méthodes pour chacune des méthodes x définies dans $\tilde{s}$, et qui pourra être utilisé pour définir une collection implantant $\tilde{s}$ (sauf en cas de redéfinition ou d'effacement dû aux def-dépendances).

8.4 Instantiation d'espèces complètement définies

Étudier les dépendances d'une espèce vis-à-vis de ses paramètres s'avère également intéressant pour l'extension du langage de base de FOC. En particulier, on peut chercher à autoriser en FOC la création de "collections" directement dans des expressions, à partir d'espèces complètement définies. Cette nouvelle construction se rapproche beaucoup du **new** des langages orientés objet, et elle peut donc sembler une étape relativement normale dans l'enrichissement du langage. Par ailleurs, elle s'est averée nécessaire dans le développement de la librairie. En particulier la représentation récursive des polynômes à plusieurs variables, qui offre pour certains types de calcul des performances notablement supérieures à d'autres représentations plus classiques, ne peut être définie en FOC sans ces constructions.

Toutefois, cet ajout ne va pas sans poser des problèmes. En premier lieu, il s'agit de comprendre quelles nouvelles dépendances cette construction est susceptible d'amener. Or, la prise en compte des dépendances vis à vis des paramètres d'une espèce permet de répondre à cette question : si on utilise **self** pour instancier le paramètre c_p de s, les decl-dépendances de l'expression sont les méthodes de l'interface de c_p dont au moins une méthode de s dépend.

Le reste de cette section apporte un début de réponse à ce problème, mais cette nouvelle construction reste encore à intégrer complètement au langage.

8.4.1 Dépendances d'une espèce vis à vis de ses paramètres

Pour cela, on commence par définir les dépendances d'une espèce par rapport à un de ses paramètres comme l'union de toutes les dépendances de ses méthodes vis à vis de ce paramètre :

Définition 74 *Soit s une espèce paramétrée par c_p. On définit les dépendances de s vis-à-vis de c_p de la manière suivante*

$$\mathcal{I}_s(a) = \bigcup_{x \in \mathcal{N}(s)} \mathcal{I}_x(a)$$

8.4.2 Espèce complètement définie et instances d'espèces

On peut dès lors ajouter une nouvelle construction **new** au langage. La syntaxe des expressions est étendue de la manière suivante :

$$expr ::= ... | \textbf{let } c = \textbf{new } s[(expr\{, expr\}*)] \textbf{ in } expr$$

La règle de typage correspondante étant :

NEW

$$\frac{\mathcal{C}(s) = (c_p \,.\, t_p)a \qquad \begin{array}{c} s \in \mathcal{C} \qquad s \text{ complètement définie} \\ \mathcal{C}, \Omega \vdash e_p : t_p \end{array} \qquad \mathcal{C} + c : \mathcal{A}(st, c), \Omega \vdash e : \tau \qquad c \notin \tau}{\mathcal{C}, \Omega \vdash \textbf{let } c = \textbf{new } s(e_p) \textbf{ in } e : \tau}$$

La condition sur l'absence de c dans le type de retour permet d'éviter des problème de portées de la variable c. Notons d'autre part que cette construction interdit l'utilisation d'instances anonymes de l'espèce s. Toutes les instances doivent être liées à un nom de collection c. Cela permet de ne pas interférer avec les autres règles : on ne fait que se donner la possibilité de rajouter localement des collections.

Par ailleurs, il faut vérifier le cas échéant que **self** est un paramètre admissible pour l'espèce s. On se base pour cela sur les espèces dont **self** hérite.

En ce qui concerne les dépendances, on peut également noter qu'il n'y a pas de def-dépendances : tout se passe à un niveau complètement abstrait. Par contre, les decl-dépendances peuvent se manifester de deux façons : soit en passant une méthode (ou plus généralement une expression impliquant une méthode) pour instancier un paramètre d'entité, soit en passant directement **self** en paramètre de collection.

Définition 75
Soit s une espèce paramétrée. On définit $\wr \textbf{let } c = \textbf{new } s(e_i) \textbf{ in } e \int$ par induction sur le nombre de paramètres de s. Pour cela, on définit une fonction auxiliaire $[\![]\!]$ de la manière suivante :

$$\llbracket \emptyset \rrbracket = \emptyset \qquad \llbracket (\textbf{\textit{self}}, c_p \ \textbf{\textit{is}} \ t_p); l \rrbracket = (\mathcal{I}_s(c_p)) \cup \llbracket l \rrbracket$$

$$\llbracket (c, c_p \ \textbf{\textit{is}} \ t_p); l \rrbracket = \llbracket l \rrbracket \ (c \neq \textbf{\textit{self}}) \qquad \llbracket (e, x \ \textbf{\textit{in}} \ \tau) \rrbracket = \wr e \int \cup \llbracket l \rrbracket$$

Si $\mathcal{E}(s)$ *est de la forme* $(c_1 \,.\, t_1)...(c_{p_f} \,.\, t_{p_f})a$*, on a*

$$\wr \textbf{\textit{let}} \ c = \textbf{\textit{new}} \ s(e_p) \ \textbf{\textit{in}} \ e \int = \llbracket \{e_p, c_p \,.\, t_p\} \rrbracket \cup \wr e \int$$

On étend de même la notion de dépendance vis-à vis des paramètres d'une espèce :

Définition 76 *Soient* c_p *un paramètre de l'espèce courante* s *et* s *une espèce paramétrée. On définit* $Deps(s, c_p)\,[\textbf{\textit{let}} \ c = \textbf{\textit{new}} \ s'(e_i) \ \textbf{\textit{in}} \ e]$ *par induction sur le nombre de paramètres de* s*. Pour cela, on définit une fonction auxiliaire* $\mathcal{I}(c_p)[\cdot]$ *de la manière suivante :*

$$\mathcal{I}(c_p)[\emptyset] = \emptyset \qquad \mathcal{I}(c_p)[(c_p, b \ \textbf{\textit{is}} \ t); l] = (\mathcal{I}_{s'}(b)) \cup \mathcal{I}(c_p)[l] \ (c \neq \textbf{\textit{self}})$$

$$\mathcal{I}(c_p)[(\textbf{\textit{self}}, b \ \textbf{\textit{is}} \ i); l] = \mathcal{I}(c_p)[l]$$

$$\mathcal{I}(c_p)[(\textbf{\textit{e}}, x \ \textbf{\textit{in}} \ \tau); l] = Deps(s, c_p)\,[e] \cup \mathcal{I}(c_p)[l]$$

Si $\mathcal{C}(s)$ *est de la forme* $(prm_1 \,.\, t_1)(prm_n \,.\, t_n)\{x_j : \tau_j = e_j\}$*, on a*

$$Deps(s, c_p)\,\left[\textbf{\textit{let}} \ c = \textbf{\textit{new}} \ s'(e_i) \ \textbf{\textit{in}} \ e\right] = \mathcal{I}(c_p)[\{e_i, prm_i] \cup Deps(s, c_p)\,[e]$$

8.5 Traduction en OCAML

Comme on l'a vu précédemment, la notion d'environnement minimal permet de définir pour chaque méthode x nouvellement définie dans une espèce s donnée un générateur de méthode. Ce générateur pourra dès lors être utilisé pour créer une collection implantant s, dans la mesure où x n'a pas été redéfinie dans l'intervalle. Si on prend en compte les dépendances des paramètres, on peut affiner cette notion d'environnement minimal, et faire en sorte que les générateurs de méthodes d'une espèce paramétrée par c d'interface i ne prennent pas en paramètre une collection complète mais seulement les méthodes dont elles ont besoin. De cette manière, on va pouvoir définir, pour une espèce complètement définie s un *générateur* d'espèce, dont les arguments seront les méthodes des paramètres de l'espèce effectivement utilisées dans le corps de s. Comme on va le voir, ce raffinement de l'environnement minimal permet de proposer une traduction OCAML basée, au même titre que celle de COQ, sur des enregistrements, tout en conservant la

possibilité de créer des "collections locales" y compris lorsque **self** est passé en paramètre.

Comme on l'a déjà dit, la seule différence entre les générateurs de méthode du chapitre précédent et ceux qu'on étudie ici concerne la gestion des paramètres de collection[1] : au lieu d'abstraire le corps du générateur de x par rapport à un enregistrement complet, on abstrait par rapport aux seules méthodes strictement nécessaires, données par $Deps(s, c_p)[x]$. De même, on définit le générateur de collection d'une espèce s complètement définie à partir de $Deps(s, c_p)[s]$ et des générateurs de méthodes correspondant, qui restent donnés par $x \leftarrow s$. Par ailleurs, il n'y a pas ici de notion de **section** ni de **variable**. Les abstractions doivent être écrites explicitement. Enfin, comme en COQ, on se donne une fonction $\mathbf{E}(s, x)$ qui à un nom d'espèce ou de collection et de méthode associe un nom de variable OCAML.

8.5.1 Expressions de base et types

Les expressions de base de FOC sont traduites de la même manière que dans la figure 4.1, hormis les appels de méthode, pour lesquels on distingue désormais quatre cas, comme indiqué dans la figure 8.3.

$$\frac{}{\mathbf{self}!x \twoheadrightarrow \mathbf{E}(s, x)} \qquad \frac{c \text{ paramètre de collection}}{c!x \twoheadrightarrow \mathbf{E}(c, x)}$$

$$\frac{c \text{ variable locale} \qquad \nabla(c) = s \qquad Mod(s) = S}{c!x \twoheadrightarrow c.S.\mathbf{E}(s, x)}$$

$$\frac{\nabla(c) = s \qquad Mod(c) = C \qquad Mod(s) = S}{c!x \twoheadrightarrow C.obj.S.\mathbf{E}(s, x)}$$

FIG. 8.3 – traduction des appels de méthode

Par ailleurs, on traduit la définition de collections locales sensiblement de la même manière que les collections globales (voir ci-dessous, section 8.5.3) :

$$\frac{\mathcal{E}(s) = (c_1 \,.\, t_1)...(c_p \,.\, t_p)a \qquad e_1 \twoheadrightarrow e_1' \qquad e_p \twoheadrightarrow e_p' \qquad e \twoheadrightarrow e'}{\begin{array}{c}\mathbf{let}\ c = \mathbf{new}\ s(e_1, ..., e_p)\ \mathbf{in}\ e \twoheadrightarrow \\ \mathbf{let}\ c = S.create(inst_{c_1}^{s}(e_1'))\ldots(inst_{c_p}^{s}(e_p'))\ \mathbf{in}\ e'\end{array}}$$

La traduction des types reste identique à celle du chapitre 4. Toutefois, on ne se servira pas des variable $\sigma(c_h)$ correspondant aux interfaces des paramètres de collection c_h de l'espèce courante. En effet, ces interfaces ne

[1] En fait il y en a une autre, qui réside dans l'absence de def-dépendances à ce niveau.

seront jamais utilisées, puisqu'on abstrait chaque générateur par rapport aux méthodes de c_h qu'il utilise, et non par rapport à tout l'enregistrement.

8.5.2 Traduction d'une espèce

La traduction d'une espèce s'effectue au sein d'un **module** (qu'on peut considérer en première approximation comme l'équivalent du **Chapter** Coq vu précédemment). Ce module, $Mod(s)$, est composé de deux grande parties, auxquelles s'ajoute une troisième quand l'espèce est complètement définie :

1. Le type d'enregistrement correspondant à l'interface de l'espèce
2. La définition des générateurs de méthode
3. Quand l'espèce est complètement définie, la définition d'un générateur de collection.

Comme précédemment, on considérera dans la suite une espèce définie par :

species $s(c_1$ **.** $t_1,\ldots c_{p_f}$ **.** $t_{p_f})$
inherits $s_1(l_1),\ldots,s_{h_f}(l_{h_f})=$
$\phi_1,\ldots,\phi_n$

Définition 77 (Type d'enregistrement) *Soit $\{c_h\}$ les paramètres de collection de s. On définit le type d'enregistrement de s, $recT(s)$ de la manière suivante :*

$$\frac{norm^*(s) = \{x_i : \tau_i = e_i\} \qquad \forall i, \tau_i \twoheadrightarrow \tau_i'}{recT(s) = \quad \textbf{\textit{type}}\ \{\sigma(c_h)\}\ stype = \{\mathbf{E}(s, x_i) : \tau_i'\}}$$

où $norm^*(s)$ est la forme normale norms où on a retiré tous les champs correspondant à des propriétés ou des théorèmes.

Définition 78 (Générateurs de méthodes) *Soit $x \in \mathcal{D}(s)$ une méthode telle que $x \upharpoonleft s = s$. On définit le générateur de méthode de x de la manière suivante :*

$$\frac{\mathcal{B}_s(x) \twoheadrightarrow e \qquad \forall h, \mathcal{I}_x(c_h) = \{x_i^h\} \qquad norms \Cap x = \{x_i\}}{\begin{array}{l}\mathbf{G}^*(s,x) \twoheadrightarrow \quad \textbf{\textit{let}}\ \mathbf{G}^*(s,x) = \textbf{\textit{fun}}\ \mathbf{E}(c_1, x_1^1) \to \ldots \textbf{\textit{fun}}\ \mathbf{E}(c_1, x_{n_1}^1) \to \ldots \\ \textbf{\textit{fun}}\ \mathbf{E}(c_h, x_1^h) \to \ldots \textbf{\textit{fun}}\ \mathbf{E}(c_h, x_{n_h}^h) \to \ldots \qquad \textbf{\textit{fun}}\ x_1 \ \to \ldots\ \textbf{\textit{fun}}\ x_n \ \to e\end{array}}$$

Remarque 1. En présence de méthodes récursives, on utilise un **let rec** et non un **let**.

Si s est complètement définie, on peut en outre définir un générateur de collection. Si s est paramétrée, ce générateur sera une fonction dont les arguments seront les paramètres (ou plutôt leurs méthodes) dont s dépend. Pour cela, on commence par raffiner la définition 67 des instantiations de paramètres, afin que les paramètres de collection soient instantiés par la liste des méthodes nécessaires :

Définition 79 (instantiation de paramètres) *Soit s une espèce, et $x \in \mathcal{D}(s)$. La liste des instantiations des paramètres de x, $Inst_s(x)$ est définie de la manière suivante :*

$$\frac{x \upharpoonleft s = s}{Inst_s(x) = \emptyset}$$

$$\frac{h = \mathbb{I}_s(x) \qquad x \upharpoonleft s = s_h \qquad \mathcal{E}(s_h) = (c_1 \cdot \tau_1, ..., c_{p_f} \cdot \tau_{p_f}) defs \qquad l_h = e_1 ... e_{p_f}}{Inst_s(x) = \{inst_{c_p}^{x}(e_p)\}}$$

$$\frac{h = \mathbb{I}_s(x) \qquad x \upharpoonleft s = s' \qquad \mathcal{E}(s_h) = (c_1 \cdot \tau_1, ..., c_{p_f} \cdot \tau_{p_f}) defs \qquad l_h = e_1 ... e_k \qquad Inst_{s_h}(x) = \{a_1 ... a_k\}}{Inst_s(x) = \{a_i[c_p \leftarrow e_p]\}}$$

où

$$inst_{c_p}^{x}(e) = \begin{cases} \{e!y\}_{y \in Deps(s,c_p)[x]} & c_p \text{ paramètre de collection} \\ \{e\} & c_p \in Deps(s, c_p)[x], \ c_p \text{ paramètre d'entité} \\ \emptyset & \text{sinon} \end{cases}$$

Dès lors, la traduction d'une méthode est identique à la définition 68, en changeant le mot clé **Local** en **let** :

Définition 80 *Soient s et $x \in \mathcal{D}(s)$.*

$$\frac{x \in \mathcal{D}(s) \qquad x \upharpoonleft s = s' \qquad \mathcal{H}(norm(s') \Cap x) = y_1 ... y_n}{\mathcal{L}_s(x) = \quad \textbf{\textit{let}}\ x = (\mathbf{G}(s', x)\ Inst_s(x)\ y_1 ... y_n)}$$

Pour définir le générateur de collection, il suffit d'abstraire tous les paramètres dont nous avons besoin :

Définition 81 (générateur de collection)

$$\frac{\forall p, \mathcal{I}_s(c_p) = \{y_i\} \qquad norm(s) = \{x_i : \tau_i = e_i\}}{\begin{array}{c} \mathcal{G}^*(s) = \textbf{\textit{let}}\ create = \textbf{\textit{fun}}\ \mathbf{E}(c_1, x_1^1) \rightarrow \ldots \textbf{\textit{fun}}\ \mathbf{E}(c_1, x_{n_1}^1) \rightarrow \ldots \\ \textbf{\textit{fun}}\ \mathbf{E}(c_p, x_1^p) \rightarrow \qquad \mathcal{L}_s(x_1)\ \textbf{\textit{in}} \ldots \mathcal{L}_s(x_n)\ \textbf{\textit{in}} \\ \{\mathbf{E}(s, x_1) = x_1; \ldots; \mathbf{E}(s, x_n) = x_n\} \end{array}}$$

8.5.3 Traduction d'une collection

Comme précédemment, la traduction d'une collection est un module, contenant deux champs : le type support, `rep`, et l'enregistrement correspondant à la collection, `obj`. Le type support est traduit comme dans le chapitre 4. L'enregistrement est traduit en appliquant le générateur de collection correspondant à l'espèce s qui est implantée :

Définition 82 (traduction d'une collection) *Soit c une collection définie par*

collection c **implements** $s(e_1,\ldots,e_{p_f})$

On définit l'enregistrement OCAML *correspondant de la manière suivante :*

$$\frac{\forall i, e_i \twoheadrightarrow e_i' \qquad \mathcal{B}_{rep}(c) = \tau \qquad \tau \twoheadrightarrow \tau' \qquad \mathcal{E}s = (c_1 \,.\, t_1)\ldots(c_{p_f} \,.\, t_{p_f} a}{\begin{array}{c} c \twoheadrightarrow \textbf{\textit{module}}\ Mod(c) = \quad \textbf{\textit{struct}} \quad \textbf{\textit{type}}\ rep = \tau' \\ \textbf{\textit{let}}\ obj = Mod(s).create\ (inst_{c_1}^{\ s}(e_1'))\ldots(inst_{c_{p_f}}^{\ s}(e_{p_f}')) \quad \textbf{\textit{end}} \end{array}}$$

Chapitre 9

Des espèces aux mixDrecs

Ce chapitre et le suivant étudient les liens entre le langage Foc tel qu'il a été présenté dans ce qui précède et les structures décrites par Sylvain Boulmé dans sa thèse, les *DRecords* et les *mixDrecs*. Ces structures permettent de formaliser en Coq les étapes d'héritage et les dépendances entre méthodes d'une même espèce. Plus précisément, les *DRecords* jouent le rôle des collections, tandis que les *mixDrecs* représentent les espèces. L'introduction de ces dernières structures a permis à Sylvain Boulmé de spécifier complètement la notion d'espèce en Coq, et en particulier de modéliser les différentes dépendances entre les composantes d'une espèce, en distinguant bien def- et decl-dépendances. À partir de là, il lui a été possible de définir un petit nombre d'opérations de base sur ces mixDrecs, et d'en prouver les principales propriétés dans Coq. L'héritage des espèces est alors codé par la composition de ces opérations de base. Les mixDrecs constituent donc une formalisation complète, en Coq, des structures manipulées par Foc. Il est donc assez naturel de regarder comment l'implantation concrète de ce langage répond à cette formalisation.

Le langage de base sur lequel s'appuyent les mixDrecs dont nous allons parler dans la suite est le calcul des constructions. Le plongement des expressions et des propriétés de base de Foc dans ce calcul est donc le même que pour la traduction en Coq.

Dans toute la suite, on suppose que les noms de méthodes et les identificateurs "normaux" (variables locales et lambda variables) sont pris dans deux ensembles dénombrables disjoints. Cette condition n'est pas restrictive et permet de simplifier notablement certaines des définitions suivantes (en particulier def.87), en supprimant des étapes d'alpha-conversion et/ou la vérification systématique de l'absence de capture de variables lors de la conversion d'un appel de méthode en un appel de variable, comme lors de la traduction en Coq. De plus, on suppose l'existence d'un itérateur $\mathbf{Y} : (\alpha \to \alpha) \to \alpha$ qui étant donné une fonction $f : \alpha \to \alpha$ renvoie son plus petit point fixe s'il existe.

Nous allons tout d'abord rappeler les principales caractéristiques des DRecords et des mixDrecs, telles qu'elles sont présentées dans la thèse de Sylvain Boulmé.

9.1 Drecords

9.1.1 Présentation

Les interfaces et les collections sont facilement représentées en Coq. On peut en effet voir les collections comme des enregistrements, et les interfaces comme des signatures d'enregistrement. Brièvement, un enregistrement est une fonction qui associe un champ à une étiquette, c'est à dire une définition. Une signature d'enregistrement associe un type à une étiquette.

Toutefois, dans le cadre de Foc, on considère des enregistrements avec champs dépendants, appelés ici *Drecords*. En effet, comme on l'a vu précédemment, l'énoncé d'une propriété, comme la commutativité de + dans les groupes abéliens, peut dépendre d'autres méthodes (dans notre exemple, de + elle-même). De ce fait, il existe un ordre naturel entre les champs d'un Drecord ou d'une signature de Drecord : un champ ne peut faire appel qu'aux étiquettes qui le précèdent dans le Drecord.

Plus formellement, on se donne un ensemble (infini) d'étiquette, *Etiq*. Les signatures d'enregistrement sont construites de la manière suivante :

Définition 83 (signature de Drecord)

$$Rec : Etiq \to Type \qquad \frac{}{\{\}_T : Rec_\emptyset}\ \text{Esig} \qquad \frac{T : Type \qquad a \notin l \qquad F : T \to Rec_l}{\{a : T; F\}_T : Rec_{a;l}}\ \text{Csig}$$

La fonction F de la règle Csig permet d'exprimer d'eventuelles dépendances des champs d'une signature de Drecord vis à vis de a. On construit donc une telle signature (et un Drecord correspondant) en commençant à droite par le champ qui "porte le plus de dépendances". Les Drecords eux mêmes sont définis par les règles suivantes :

Définition 84 (Drecord)

$$\frac{}{\{\} : \{\}_T}\ \text{Eimpl} \qquad \frac{\{a : T; F\} : Rec_{a;l} \qquad x : T \qquad i : (Fx)}{\{a = x; i\} : \{a : T; F\}_T}\ \text{Cimpl}$$

9.1.2 Opérations et relations entre signatures de Drecord

Les interfaces de Foc doivent pouvoir être manipulées de trois façons différentes, ce qu'on retrouve au niveau des signatures de Drecord :

1. l'héritage simple correspond à étendre une signature par de nouveaux champs
2. l'héritage multiple correspond à fusionner deux signatures, dont les champs communs ont le même type
3. Afin de vérifier que les paramètres de collections sont instantiés par des collections ayant le bon type, il faut se munir d'une relation de sous-signature.

Sous-signature

La relation de sous-signature entre deux signatures de Drecord s_1 et s_2, notée $s_1{:}\succ s_2$ dit informellement que s_1 contient au moins tous les champs de s_2, avec des types identiques. Elle est définie de la manière suivante :

Définition 85 (sous-signature)

$$\frac{}{s{:}\succ\{\}_T}\ \_\text{E} \qquad \frac{(x:T)(F_1\ x){:}\succ(F_2\ x)}{\{a:T;F_1\}_T{:}\succ\{a:T;F_2\}_T}\ \_\text{C} \qquad \frac{(x:T)(F\ x){:}\succ s}{\{a:T;F\}_T{:}\succ s}\ \_\text{LIFT}$$

$$\frac{s_1{:}\succ s_2 \qquad s_2{:}\succ s_3}{s_1{:}\succ s_3}\ \_\text{TRANS}$$

$$\frac{(x:T_1;y:T_2)(F_1\ x\ y){:}\succ(F_2\ y\ x)}{\{a:T_1;[x:T_1]\{b:T_2;(F_1\ x)\}_T\}_T{:}\succ\{b:T_2;[y:T_2]\{a:T_1;(F_2\ y)\}_T\}_T}\ \_\text{SWAP}$$

La règle _SWAP permet de permuter l'ordre des champs entre s_1 et s_2. Bien entendu, cette règle ne peut pas s'appliquer pour n'importe quel champ d'un mixDrec donné : $:\succ$ ne compare que des signature de DRecord bien formées.

Par ailleurs, tout Drecord i de signature s_1 peut être converti en un Drecord de signature s_2 (avec $s_1{:}\succ s_2$). Le résultat de cette conversion, noté $i_{|s_2}$, est le Drecord obtenu en enlevant les champs n'apparaissant pas dans s_2, et en réordonnant les champs restant de la manière dont il le sont dans s_2. Une définition formelle, par induction sur la preuve de $s_1{:}\succ s_2$ se trouve dans la thèse de S. Boulmé.

Extension de signature

Il est possible d'étendre une signature s de Drecord en ajoutant de nouveaux champs à la fin de la signature. Ces nouveaux champs peuvent bien sûr dépendre des champs existants de s. Le "morceau" de signature qu'on veut ajouter à s doit donc être exprimé dans un contexte contenant les champs de s. De plus, les étiquettes des champs ajoutés doivent être distinctes de celles de s.

Fusion de signatures

Enfin, la fusion de signatures de Drecord agit sur deux signatures s_1 et s_2 contenant éventuellement des noms de champs communs, contrairement au cas précédent. Dans ce cas, il faut vérifier que les types associés à ces étiquettes communes sont égaux. Pour cela, on construit une sous-signature s commune à s_1 et s_2 qui doit contenir *toutes* les étiquettes communes de s_1 et s_2. On construit alors la fusion des signatures comme l'extension de s_2 par les champs de $s_1 \backslash s_2$. Cette dernière notation est définie comme étant l'ensemble des champs de s_1 non présent dans s_2.

9.2 MixDrecs

9.2.1 Présentation

Les MixDrecs servent à représenter les espèces de Foc. Contrairement aux Drecords et à leurs signatures, il s'agit donc de définir des enregistrements mélangeant des champs déclarés et des champs définis. Il faut aussi tenir compte des dépendances, et en particulier des def-dépendances. Pour cela, au lieu de représenter les mixDrecs par une liste ordonnée de champs, on va plutôt les voir comme des arbres. Ces arbres sont composés de trois catégories de nœuds :

- Les nœuds vides, qui forment les feuilles de l'arbre.
- Les nœuds abstraits correspondent aux champs déclarés. Un nœud abstrait contient l'étiquette du champ qu'il représente et son type, et a un seul fils.
- Les nœuds manifestes correspondent aux champs définis. Un tel nœud contient l'étiquette x du champ, son type et sa définition. Le premier fils est un mixDrec où x est considéré comme abstrait (et où on ne peut donc pas trouver de def-dépendance vis-à-vis de x). Le second fils est un mixDrec où on connaît la définition de x, et où on peut donc trouver des def-dépendances. De plus, les deux fils du nœud doivent respecter la même signature : les seules différences qu'on s'autorise sont des effacements de définitions entre le second fils et le premier fils.

Remarque 1. Dans sa thèse, Sylvain Boulmé appelle les def-dépendances *dépendances transparentes* et les decl-dépendances *dépendances opaques*, en référence aux mots clés de Coq `Transparent` et `Opaque` qui signalent à Coq qu'une définition donnée peut être dépliée ou non au cours d'une preuve.

Par exemple, on peut examiner l'espèce des setoïdes, définie informellement de la manière suivante :

- un type support abstrait *rep*
- une signature $eq : rep \to rep \to Prop$
- la propriété $eq\_refl : \forall x \in rep, (eq\ x\ x)$
- la définition $diff := [x, y] \neg (eq\ x\ y)$

– le théorème $diff\_spec : \forall x, y \in rep,\ (diff\ x\ y) \Rightarrow \neg(eq\ x\ y)$
– le théorème $diff\_nrefl : \forall x \in rep,\ \neg(diff\ x\ x)$.

diff _ spec def-dépend de *diff*, tandis que *diff _ nrefl* en decl-dépend. La figure 9.1 représente la forme du mixDrec correspondant ($\mathbb{A}$ représentant les nœuds abstraits et $\mathbb{M}$ les nœuds manifestes. Les nœuds vides qui devraient figurer en bas de la figure ont été omis). Dans cette figure, les trois pre-

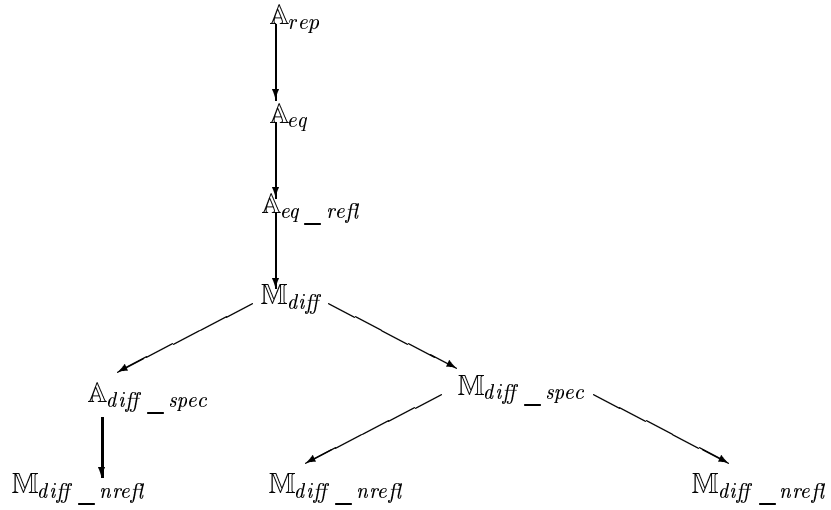


FIG. 9.1 – mixDrec des setoides

miers nœuds sont abstraits. Le quatrième, *diff* porte une définition. Il a donc deux fils. Le premier correspond au cas où on "efface" la définition de *diff*. *diff _ spec* doit alors être considéré comme un nœud abstrait, du fait de la def-dépendance vis-à-vis de *diff*. En revanche, on peut garder la définition de *diff _ nrefl*, qui n'a qu'une decl-dépendance vis-à-vis de *diff* et de *diff _ spec*.

Le second fils de $\mathbb{M}_{diff}$ correspond au cas où on connaît la définition de *diff*. *diff _ spec* peut alors être défini. $\mathbb{M}_{diff\_spec}$ est donc un nœud manifeste, qui a lui même deux fils, qui contiennent tout les deux la définition de *diff _ nrefl*, celle-ci n'étant pas dépendante de la définition de *diff _ spec*.

Plus formellement, on peut représenter un mixDrec par deux grandes composantes :

– Un prédicat Pre qui décrit sa structure arborescente.
– une signature de Drecord, $\{S\}_T$, qui donne la liste des champs du mixDrec.

Cette signature correspond en fait à la branche la plus à gauche du mixDrec, où tous les champs sont considérés comme abstraits. Un mixDrec M de structure p et de signature S sera noté $M : Mix_{S,p}$. De plus, on se donne une relation, $\succcurlyeq$, exprimant qu'un mixDrec est une vue plus définie d'un autre. En particulier, comme on l'a signalé, dans le cas d'un nœud manifeste,

le fils droit doit être une vue plus définie du fils gauche : il reprend les mêmes champs, dans le même ordre, mais peut contenir des nœuds manifestes supplémentaires. Les règles d'inférence pour le typage des mixDrecs et la relation $\succcurlyeq$ sont décrites dans la figure 9.2.

$$\frac{}{\emptyset_p : Pre_\emptyset} \qquad \frac{a \notin l \qquad p : Pre_l}{\mathbb{A}\ p : Pre_{a;l}} \qquad \frac{a \notin l \qquad p_1 : Pre_l \qquad p_2 : Pre_l}{\mathbb{M}\ p_1\ p_2 : Pre_{a;l}}$$

$$\frac{}{\{\!\{\}\!\} : Mix_{\{\}_T,\emptyset_p}} \qquad \frac{f : (x : T)Mix_{(F\ x),p}}{\{\!\{a : T; f\}\!\} : Mix_{\{a:T;F\}_T,\mathbb{A}p}}$$

$$\frac{f : (x : T)Mix_{(F\ x),p_1} \qquad x : T \qquad m : Mix_{(F\ x),p_2} \qquad m \succcurlyeq (f\ x)}{\{\!\{a : T = x; f; m\}\!\} : Mix_{\{a:T;F\}_T,\mathbb{M}p_1p_2}}$$

$$\frac{}{\{\!\{\}\!\} \succcurlyeq \{\!\{\}\!\}} \qquad \frac{(y : T)(f_1\ y) \succcurlyeq (f_2\ y)}{\{\!\{a : T; f_1\}\!\} \succcurlyeq \{\!\{a : T; f_2\}\!\}}$$

$$\frac{m \succcurlyeq (f_1\ x) \qquad (y : T)(f_1\ y) \succcurlyeq (f_2\ y)}{\{\!\{a : T = x; f_1; m\}\!\} \succcurlyeq \{\!\{a : T; f_2\}\!\}}$$

$$\frac{m_1 \succcurlyeq m_2 \qquad (y : T)(f_1\ y) \succcurlyeq (f_2\ y)}{\{\!\{a : T = x; f_1; m_1\}\!\} \succcurlyeq \{\!\{a : T = x; f_2; m_2\}\!\}}$$

FIG. 9.2 – typage des mixDrecs

9.2.2 Opérations sur les mixDrecs

On dispose de deux opérations de base sur les mixDrecs : le plongement dans une sous-signature, où on ajoute les champs supplémentaires sous forme de nœuds abstraits, et la fusion de deux mixDrecs de même signature.

Fusion de deux mixDrecs

On réalise la fusion sur des mixDrecs ayant la même signature. Le principe de base est donc de faire la fusion nœud à nœud. En cas de conflit, c'est à dire si les deux nœuds sont manifestes, on prend toujours la définition du mixDrec de gauche[1]. Toutefois, même si le nœud du mixDrec de gauche est abstrait, on ne peut pas forcément prendre la définition du mixDrec de droite. En effet, il faut tenir compte d'eventuel def-dépendances du nœud courant vis à vis des champs fusionnés précédemment. Par exemple,

[1] Comme on le verra dans la suite, c'est la convention inverse qui a été adoptée en FOC, conformément au mécanisme d'héritage de OCAML.

la fusion des deux mixDrecs suivants :

$$M_1 = \{\!\{a : T_1 = e_1; [a : T_1]\, \{\!\{b : T_2\}\!\}\, ; \{\!\{b : T_2 = e_2\}\!\}\}\!\}$$

$$M_2 = \{\!\{a : T_1 = e_1'; [a : T_1]\, \{\!\{b : T_2 = e_2'\}\!\}\, ; \{\!\{b : T_2 = e_2'\}\!\}\}\!\}$$

donnera le mixDrec

$$M = \{\!\{a : T_1 = e_1; [a : T_1]\, \{\!\{b : T_2\}\!\}\, ; \{\!\{b : T_2 = e_2\}\!\}\}\!\}$$

et non le mixDrec M', obtenu en considérant uniquement les nœuds placés en même position dans M_1 et M_2, sans tenir compte de la structure globale de chacun de ces mixDrecs :

$$M' = \{\!\{a : T_1 = e_1; [a : T_1]\, \{\!\{b : T_2 = e_2'\}\!\}\, ; \{\!\{b : T_2 = e_2\}\!\}\}\!\}$$

En effet M' n'est pas bien formé, car le champ b y existe avec deux définitions différentes.

Plus généralement, on ne peut pas procéder indépendamment à la fusion des différentes branches de deux mixDrecs. Afin de garantir l'obtention de mixDrecs bien formés, on définit la liste de contrôle de la fusion à partir des prédicats Pre des deux mixDrecs M_1 et M_2. Cette liste est composée de 0 et de 1, qui indique si on peut prendre la définition de droite ou non. La définition complète de cette liste de contrôle et de la fusion se trouvent dans la thèse de S. Boulmé.

Plongement d'un mixDrec dans une sous-signature

L'autre opération importante sur les mixDrecs consiste à le plonger dans une sous-signature de sa propre signature. En prenant un mixDrec M et sa signature s, la définition du plongement de M dans s', tel que $s' {:}\succ s$, se fait par induction sur la dérivation de $s' {:}\succ s$. On donne ici informellement les différents cas (la définition formelle en est donnée dans la thèse de S. Boulmé) :

- _E M est le mixDrec vide. On ajoute tous les champs de s comme nœuds abstraits.
- _C On ne touche pas au premier champ de M, et on poursuit le plongement sur le(s) sous-mixDrec(s).
- _LIFT On ajoute le champ de tête de s' en tête de M (en tant que champ abstrait), et on poursuit le plongement.
- _TRANS On compose les deux plongements.
- _SWAP On intervertit les deux champs de M. Dans le cas où le second est un champ défini, on efface cette définition (elle est susceptible de contenir des dépendances vis-à-vis du premier champ).

La suite du chapitre montre qu'on peut faire correspondre à toute interface un Drecord et à toute une espèce s en forme normale un mixDrec bien

formé. De plus, on verra comment l'algorithme de résolution de l'héritage peut se retranscrire à l'aide des opérations de base sur les mixDrecs. Dans le chapitre suivant, on s'intéressera aux générateurs de méthodes. On montrera en particulier que tous les générateurs des méthodes définies d'une espèce donnée s peuvent se retrouver en étudiant la structure du mixDrec associé à s. En un certain sens, l'utilisation de ces générateurs permet de "déplier" la structure arborescente des mixDrecs, et de présenter à Coq des termes plus simple.

9.3 Interface et DRecord

Soit s une espèce de Foc, et $norm(s)$ sa forme normale. On définit $[\![norm(s)]\!]$ le *D-record* associé à $norm(s)$ de la manière suivante :

Définition 86

$$[\![\emptyset]\!] = \{\}_T$$

$$[\![\textbf{\textit{rep}}; l]\!] = \{rep : Set; \lambda rep [\![rep]\!]\}_T$$

$$[\![\textbf{\textit{rep}} = \tau; l]\!] = \{rep : Set; \lambda rep [\![rep]\!]\}_T$$

$$[\![\textbf{\textit{sig}}\ x\ \textbf{\textit{in}}\ \tau; l]\!] = \{x : \tau; \lambda x [\![l]\!]\}_T$$

$$[\![\textbf{\textit{let}}\ x\ \textbf{\textit{in}}\ \tau = e; l]\!] = \{x : \tau; \lambda x [\![l]\!]\}_T$$

$$[\![\textbf{\textit{let rec}}\ x_1\ \textbf{\textit{in}}\ \tau_1 = e_1\ ...\ \textbf{\textit{and}}\ x_n\ \textbf{\textit{in}}\ \tau_n = e_n; l]\!] = \{x_1 : \tau_1; \lambda x_1.(...\{x_n : \tau_n; \lambda x_n [\![l]\!]\}_T)\}_T$$

$$[\![\textbf{\textit{property}}\ x : prop; l]\!] = \{x : prop; \lambda x.[\![l]\!]\}_T$$

$$[\![\textbf{\textit{theorem}}\ x : prop = demo; l]\!] = \{x : prop; \lambda x.[\![l]\!]\}_T$$

Théoreme 13 *Avec s une espèce bien formée, $[\![norm(s)]\!]$ est une signature de Drecord bien formée, et la liste de ses étiquettes est $\mathcal{N}(norm(s))$ $(= \mathcal{N}(s))$:*

$$[\![norm(s)]\!] : Rec_{\mathcal{N}(norm(s))}$$

Preuve. Par induction sur le nombre de champs de la forme normale et par cas sur chaque champ. □

9.4 Espèces et MixDrecs

De même, on définit pour chaque espèce $norm(s)$ en forme normale un mixDrec associé, par induction sur la taille de $norm(s)$. Les deux points délicats sont la traduction d'un champ **let rec** et la creation de la branche abstraite d'un nœud défini du mixDrec (nœud $\mathbb{M}$). Pour cela, on définit une fonction $\mathbb{E}_{\{n\}}$ qui efface les définitions ayant des def-dépendances vis-à-vis d'un nom n donné.

Pour la traduction de **let rec**, on a essentiellement deux solutions : soit refaire une théorie des mixDrec incluant des blocs récursifs, soit donner la définition récursive dans l'environnement global, avec un dépliage des dépendances (def et decl, même si en pratique seul le cas des decl dépendances sera vraiment utilisé) analogue à celui fait lors de la traduction des espèces en Coq (NB : cela suppose qu'on a une preuve de terminaison, et non pas seulement une définition **let rec** "à la ML").

Champs récursifs On peut maintenant définir le mixDrec correspondant à l'espèce. Cette traduction est susceptible de modifier l'environnement global dans le cas de définition mutuellement récursives. On utilise pour cela un nouvel identificateur, *wi*. Comme pour la traduction en Coq, il faut alors faire remonter les dépendances de la définition mutuellement récursive, soit sous forme de λ abstraction (cas des decl-dépendances), soit sous forme de variables globales (def-dépendances).

Définition 87 (Extraction d'un champ récursif) *Soient $norm(s)$ une espèce en forme normale, $\phi =$ **let rec** x_1 **in** $\tau_1 = e_1$ ... **and** x_n **in** $\tau_n = e_n$ un champ récursif de $norm(s)$ et WI un nom de variable frais. On définit le champ extrait de ϕ, $\nwarrow \phi$ de la manière suivante. On note $(y_i : \theta_i = \bot) \in norm(s) \Cap \phi$ les noms des méthodes dont la déclaration est présente dans l'environnement de typage minimal de ϕ, et $(z_i : \kappa_i = d_i) \in norm(s) \Cap \phi$ ceux dont ϕ def-dépend. Alors,*

$$\nwarrow \phi = \mathbf{Y}\,\lambda WI \lambda y_1 \ldots \lambda y_m.\textbf{\textit{let}} z_1 = (d_1) \searrow \textbf{\textit{in}} \ldots \textbf{\textit{let}} z_l = (d_l) \searrow \textbf{\textit{in}}\,(e_1, ..., e_n) \lightning$$

avec

$$(d) \searrow = d[self!y_i \leftarrow y_i; self!z_i \leftarrow z_i]$$

$$(d) \lightning = (d) \searrow [\textbf{\textit{self}}!x_i \leftarrow (proj_i\ (WI y_1 ... y_m))]$$

Pour clarifier cette définition, on peut considérer par exemple une espèce représentant les entiers, et y définir les deux fonctions `is_odd` et `is_even`

par récurrence mutuelle :

```
species int =
  rep;
  sig eq in self → self → bool;
  sig zero in self;
  sig pred in self → self;
  let rec odd (x) =
          if self !eq(x,self !zero) then
            #false
          else
            self !even(self !pred(x))
      and even(x) =
          if self !eq(x,self !zero) then
            #true
          else
            self !odd(self !pred(x))
  ;
end
```

Ici, on a $\wr\texttt{odd}\int_{\texttt{int}} = \wr\texttt{even}\int_{\texttt{int}} = \{eq, zero, pred\}$, et si ϕ est la définition de `odd` et `even`, il vient

$$\nwarrow \phi = \mathbf{Y}(\lambda WI\ \lambda eq\ \lambda zero\ \lambda pred$$

$$(\lambda x.\textbf{if}(eq\ x\ zero)\ \textbf{then}\ \texttt{false}\ \textbf{else}\ ((\texttt{snd}\ (WI\ eq\ zero\ pred))(pred\ x)),$$

$$\lambda x.\textbf{if}(eq\ x\ zero)\ \textbf{then}\ \texttt{true}\ \textbf{else}\ ((\texttt{fst}\ (WI\ eq\ zero\ pred))(pred\ x))))$$

et $\mathbf{Y}$ va construire une fonction qui étant données des implantations de `eq` `zero` et `pred` renvoie les deux fonctions `odd` et `even` correspondantes.

Lemme 39 *Avec les notations précédentes,* $\nwarrow \phi$ *est bien typé, et a le type* $\theta_1 \to .. \to \theta_n \to (\tau_1 * ... * \tau_n)$

Preuve. Par hypothèse, ϕ est bien typé dans l'environnement de l'espèce, de même que les d_i. Par ailleurs, par définition de $\Uparrow$, et de $(,) \searrow$ tous les appels de méthodes sont transformés en variables *présentes dans l'environnement local.* De même, les substitutions des appels récursifs par la i-ème projection de $WI\ y_1\ ...\ y_m$ sont correctes : cette application est bien un n-uplet par construction.

Plus précisément, avec Γ l'environnement dans lequel est typée l'espèce $norm(s)$, on peut obtenir une dérivation de typage pour $\Gamma \vdash \nwarrow \phi$ à partir de la dérivation de ϕ lors du typage de l'espèce :

FIG. 9.3 – dérivation de typage de $\nwarrow \phi$

$$\dfrac{\mathbf{D_1} \qquad \dfrac{\dfrac{\mathbf{\Phi}}{\Gamma^+ + z_1 : \kappa_1 ... z_l : \kappa_l \vdash \quad (e_1, ..., e_n) : (\tau_1 * ... * \tau_n)} \vdots}{\Gamma^+ + z_1 : \kappa_1 \vdash \mathbf{let}\ z_2 = d_2\ \mathbf{in}...\mathbf{in}(e_1, ..., e_n) : (\tau_1 * ... * \tau_n)}}{\Gamma^+ \vdash d_1 : \kappa_1 \qquad \dfrac{\Gamma + WI : \theta_1 \to .. \to \theta_m \to (\tau_1 * ... * \tau_n), \qquad y_1 : \theta_1, y_m : \theta_m \vdash \mathbf{let}\ z_1 = d_1\ \mathbf{in}...\mathbf{in}(e_1, ..., e_n) : (\tau_1 * ... * \tau_n)}{\vdots}}{\Gamma \vdash \nwarrow \phi}$$

les $\mathbf{D_i}$ de la dérivation précédente sont obtenus en remplaçant dans la dérivation de typage de z_i dans l'espèce s toutes les occurences de la règle

$$\dfrac{[\text{SELF CALL}] \qquad x : \tau = expr \in \Sigma}{\Gamma \vdash \mathbf{self}\,!u : \tau}$$

par une simple application de la règle VAR. En effet, par définition de $\Cap$, u est un des y_i ou un des z_i, et a donc été remplacé par un identificateur présent dans l'environnement par $(\cdot) \searrow$.

Dans le cas de $\mathbf{\Phi}$, on peut rencontrer un autre cas, quand $u = x_i$. Dans ce cas, il s'agit de dériver

$$\Gamma \vdash proj_i(WI\ y_1 ... y_m) : \tau_i$$

Or en revenant à la dérivation 9.3, il vient $\Gamma(WI) = \theta_1 \to .. \to \theta_m \to (\tau_1 * ... * \tau_n)$, et $\Gamma(y_i) = \theta_i$ et on peut achever la dérivation. □

Durant la traduction d'une espèce en forme normale en mixDRec, nous allons systématiquement définir chaque bloc mutuellement récursif ϕ à l'extérieur des mixDRecs, et affecter $\nwarrow \phi$ à une variable fraîche WI. Dès lors, la définition de chacune des méthodes du bloc sera simplement un appel à la i-ème composante de WI. Par convention, la branche abstraite issue de x_1, premier nom de méthode apparaissant dans ϕ ne contient que des nœuds abstraits pour les autres méthodes du champs ϕ, bien que l'utilisation de l'itérateur $\mathbf{Y}$ permette de ne pas abstraire tout le bloc. En effet, dans une version où on donne une preuve de terminaison, il est a priori nécessaire de refaire cette preuve à chaque fois que l'on modifie une des méthodes incluse dans le bloc.

On peut maintenant donner une définition du passage d'une espèce en forme normale à un mixDrec comprenant les mêmes méthodes et les mêmes définitions :

Définition 88

$$\ll\emptyset\gg = \{\!\{\}\!\}$$

$$\ll \textbf{\textit{rep}}; l\gg = \{\!\{rep : Set; \lambda rep \ \ \ll l\gg\}\!\}$$

$$\ll \textbf{\textit{rep}} = \tau; l\gg = \{\!\{rep : Set = \tau; \lambda rep.\ \ll \mathbb{E}_{rep}\ (l)\gg; \ll l\gg\}\!\}\ ;$$

$$\ll \textbf{\textit{sig}}\ x\ \textbf{\textit{in}}\ \tau; l\gg = \{\!\{x : \tau; \lambda x.\ \ll l\gg\}\!\}$$

$$\ll \textbf{\textit{let}}\ x\ \textbf{\textit{in}}\ \tau = e; l\gg = \{\!\{x : \tau = e; \lambda x.\ \ll \mathbb{E}_x\ (l)\gg; \ll l\gg\ \}\!\}$$

$$\ll \textbf{\textit{let rec}}\ x_1\ \textbf{\textit{in}}\ \tau_1 = e_1\ ...\ \textbf{\textit{and}}\ x_n\ \textbf{\textit{in}}\ \tau_n = e_n; l\gg =$$

$$\{\!\{\{\!\{x_1 : \tau_1 = fst(WI\ \ y_1 ... y_m); \lambda x_1.\ \{\!\{....\lambda x_n\ \ll \mathbb{E}_{\{x_1,...x_n\}}\ (l)\gg\}\!\}\ ; \ll l\gg\}\!\}\}\!\}$$

$$avec\ WI\ variable\ fraîche$$

$$\ll \textbf{\textit{property}}\ x : prop; l\gg = \{\!\{x :\ \ prop; \lambda x\ \ll l\gg\}\!\}$$

$$\ll \textbf{\textit{theorem}}\ x : prop = demo; l\gg =$$
$$\{\!\{x : prop = demo; \lambda x\ \ll \mathbb{E}_x\ (l)\gg; \ll l\gg\}\!\}$$

On peut dès lors énoncer la spécification de $\mathbb{E}_{\mathcal{N}}\ (l)$ vis à vis de $\gtrdot$:

Lemme 40 *Soit l une liste de champs, et $\{x_i\}_{i=1..n}$ des noms de méthodes, tels que $\{x_i\} \cap \mathcal{N}(l) = \emptyset$. On a*

$$\ll l\gg \gtrdot \ll \mathbb{E}_{\{x_i\}}\ (l)\gg$$

Preuve. Immédiat par induction sur le nombre de champs de l : l'ordre dans lequel se présentent les champs du mixdRec est préservé, et avec le lemme 39, $\mathbb{E}_{\{.\}}\ (l)$ conserve toutes les signatures et un sous-ensemble des définitions de l. □

Remarque 2. La condition sur les noms de méthodes est automatiquement vérifiée quand on part d'une espèce en forme normale, puisque les noms de méthodes introduits le sont dans un unique champ.

De même, on peut définir un arbre d'étiquettes signalant l'absence ou la présence de définitions pour chaque champ de l'espèce.

Définition 89 (information de définition)

$$\mathbb{P}(\emptyset) = \emptyset_p$$

$$\mathbb{P}(\boldsymbol{rep}; l) = \mathbb{A}_{rep}\mathbb{P}(l)$$
$$\mathbb{P}(\boldsymbol{rep} = \tau; l) = \mathbb{M}_{rep}\mathbb{P}(\mathbb{E}_{rep}(l))\mathbb{P}(l);$$
$$\mathbb{P}(\boldsymbol{sig}\ x\ \boldsymbol{in}\ \tau; l) = \mathbb{A}_x\mathbb{P}(l)$$
$$\mathbb{P}(\boldsymbol{let}\ x\ \boldsymbol{in}\ \tau = e; l) = \mathbb{M}_x\mathbb{P}(\mathbb{E}_x(l))\mathbb{P}(l)$$

$$\mathbb{P}(\boldsymbol{let\ rec}\ x_1\ \boldsymbol{in}\ \tau_1 = e_1\ ...\ \boldsymbol{and}\ x_n\ \boldsymbol{in}\ \tau_n = e_n; l) = \mathbb{M}_{x_1}(\mathbb{A}_{x_2}...\mathbb{A}x_n\mathbb{P}(\mathbb{E}_{\{x_1,...x_n\}}(l)))(\mathbb{M}_{x_2}...\mathbb{M}x_n\mathbb{P}(l))$$
$$\mathbb{P}(\boldsymbol{property}\ x : prop; l) = \mathbb{A}_x\mathbb{P}(l)$$
$$\mathbb{P}(\boldsymbol{theorem}\ x : prop = demo; l) = \mathbb{M}_x\mathbb{P}(\mathbb{E}_x(l))\mathbb{P}(l)$$

Théoreme 14 *Avec $norm(s)$ une espece en forme normale, on a*

$$\ll norm(s) \gg : Mix_{[\![norm(s)]\!],\mathbb{P}(norm(s))}$$

Preuve. Par induction sur le nombre de champs de $norm(s)$, et par cas sur chaque champ. La principale difficulté concerne la traduction d'un champ **let rec**. Regardons le cas particulier de deux définitions mutuellement récursives. Il est ensuite facile de généraliser à un bloc de n méthodes. La traduction correspondante est donc :

$$M =$$

$$\left\{\!\!\left\{ \left\{\!\!\left\{ \begin{array}{l} x_1 : \tau_1 = fst\ (WI\ y_1...y_m); \\ \lambda x_1 \left\{\!\!\left\{ x_2 : \tau_2; \ll \mathbb{E}_{\{x_1,x_2\}}(l) \gg \right\}\!\!\right\} \\ \left\{\!\!\left\{ x_2 : \tau_2 = snd\ (WI\ y_1...y_m); \lambda x_2\ \ll \mathbb{E}_{\{x_1,x_2\}}(l) \gg; \ll l \gg \right\}\!\!\right\} \end{array} \right\}\!\!\right\} \right\}\!\!\right\}$$

Par hypothèse, les y_i sont des champs du mixDRec qui précèdent la définition de x_1 et x_2. De plus, avec le lemme 39, il vient

$$WI : \theta_1 \to ... \to \tau_m \to (\tau_1 * ... * \tau_n)$$

La définition de x_1 dans le mixDRec a bien le type τ_1 attendu. On peut donc appliquer la règle de bonne formation des mixDRecs.

$$\frac{\begin{array}{c} \lambda x_1 \left\{\!\!\left\{ x_2 : \tau_2; \ll \mathbb{E}_{\{x_1,x_2\}}(l) \gg \right\}\!\!\right\} : (x_1 : \tau_1) Mix_{[\![\{\mathbf{sig}\ x_2:\tau_2;l\}]\!]\mathbb{A}_{x_2}\mathbb{P}\left(\mathbb{E}_{\{x_1;x_2\}}(l)\right)} \\ (fst\ WI) : \tau_1 \\ \left\{\!\!\left\{ x_2 : \tau_2 = snd\ (WI\ y_1...y_m); \lambda x_2\ \ll \mathbb{E}_{\{x_1,x_2\}}(l) \gg; \ll l \gg \right\}\!\!\right\} : \\ Mix_{[\![\{\mathbf{sig}\ x_2:\tau_2;l\}]\!]\mathbb{M}_{x_2}\mathbb{P}\left(\mathbb{E}_{\{x_1;x_2\}}(l)\right)\mathbb{P}(l)} \\ \left\{\!\!\left\{ x_2 : \tau_2 = snd\ (WI\ y_1...y_m); \lambda x_2\ \ll \mathbb{E}_{\{x_1,x_2\}}(l) \gg; \ll l \gg \right\}\!\!\right\} \gg \\ ((x_1 : \tau_1) Mix_{[\![\{\mathbf{sig}\ x_2:\tau_2;l\}]\!]\mathbb{A}_{x_2}\mathbb{P}\left(\mathbb{E}_{\{x_1;x_2\}}(l)\right)} e_1) \end{array}}{M : Mix_{(x_1)[\![\mathbf{sig}\ x_2\ \mathbf{in}\ \tau_2;\mathbb{E}_{\{x_1;x_2\}}(l)]\!]\mathbb{M}_{x_1}\left(\mathbb{A}_{x_2}\mathbb{P}\left(\mathbb{E}_{\{x_1;x_2\}}(l)\right)\right)(\mathbb{M}_{x_2}\mathbb{P}(l))}}$$

Par hypothèse de récurrence, les mixDrecs sont bien typés. De plus, on vient de voir que le "corps" de x_1 dans le mixDrec avait bien le type τ_1. Enfin, le lemme 40 nous montre que la branche concrète est bien une vue plus définie (au sens de $\succcurlyeq$) de la branche abstraite. □

9.5 Héritage et fusion de mixDrecs

Dans cette section, nous voyons comment les relations d'héritage entre espèces de Foc se retrouvent au niveau des mixDrecs. Nous commençons par décrire un héritage simple, avant de généraliser au cas d'un héritage multiple.

9.5.1 Mise en forme normale et ordre des champs

L'algorithme de mise en forme normale tel qu'il est présenté auparavant nous servira de base pour étudier comment les relations d'héritage entre espèces se retrouvent dans les traductions en signatures de Drecord et en mixDrecs. Toutefois, il est nécessaire d'utiliser un raffinement de cet algorithme, où l'ordre de dépendance est préservé tout au long de la construction de $\mathbb{W}_2$ (en reprenant les notations antérieures), au lieu de réordonner les champs retenus uniquement à la fin de la phase de résolution de l'héritage. Pour cela, il suffit de modifier la façon dont on gère les redéfinitions dans la construction de $\mathbb{W}_2$. Au lieu de se contenter de remettre $(\phi \circledcirc \psi_{i_0})$ dans $\mathbb{W}_1$ pour continuer l'analyse, on enlève également tous les ψ_i vérifiant $\psi_i \blacktriangleleft_{\mathbb{W}_2} \psi_{i_0}$ de $\mathbb{W}_2$. Bien entendu, les éventuels effacements de définitions liés aux def-dépendances sont réalisés sur les ψ_i.

Toutefois, on ne peut remettre directement les ψ_i dans $\mathbb{W}_1$ à la suite du champ fusionné : dans le cas de la fusion de plusieurs blocs mutuellement récursifs, on risquerait de sélectionner la "mauvaise" définition. Par exemple, avec

$$\mathbb{W}_2 = \textbf{let rec } x_1 = e_1 \textbf{ and } x_2 = e_2; \qquad \textbf{let } x_3 = e_3$$

$$\mathbb{W}_1 = \textbf{let rec } x_1 = e_1' \textbf{ and } x_3 = e_3';$$

la première étape fusionne les deux champs récursifs, et donne le résultat suivant :

$$\mathbb{W}_2 = \emptyset$$

$$\mathbb{W}_1 = \textbf{let rec } x_1 = e_1' \textbf{ and } x_2 = e_2 \textbf{ and } x_3 = e_3'; \qquad \textbf{let } x_3 = e_3$$

Et on va donc se retrouver avec un unique champ récursif, où x_3 est défini par e_3 et non e_3'. Pour éviter ce genre de situation, nous utilisons une liste intermédiaire $\mathbb{S}$, qui contient des champs susceptibles d'être fusionnés avec le champ courant de $\mathbb{W}_1$.

Plus formellement, le nouvel algorithme fonctionne de la manière suivante. Au départ, on a $\mathbb{W}_2 = \mathbb{S} = \emptyset$. Ensuite, à chaque étape, avec $\mathbb{W}_1 = \phi, \mathbb{X}$, $\mathbb{W}_2 = \psi_1 \ldots \psi_m$ et $\mathbb{S} = \psi_{m+1} ... \psi_n$. On a deux cas possibles :

- Si $\mathcal{N}(\phi) \cap \mathcal{N}(\mathbb{W}_2) = \emptyset$, on effectue les opérations suivantes :
  - $\mathbb{W}_1 \leftarrow \mathbb{X}$
  - $\mathbb{W}_2 \leftarrow (\psi_1 \ldots \psi_n, \phi_1, \mathbb{S})$
  - $\mathbb{S} \leftarrow \emptyset$

  On ajoute le champ analysé, et les éventuels champs de $\mathbb{S}$, provenant d'une fusion antérieure de ϕ.
- Sinon, on sélectionne i_0 le plus grand index tel que $\mathcal{N}(\phi) \cap \mathcal{N}(\psi_{i_0}) \neq \emptyset$. Soit $\mathbb{D}$,$\mathbb{N}$ la partition de $\mathbb{W}_2$ telle que les champs χ de $\mathbb{D}$ dépendent de $\mathcal{N}(\phi)$ et vérifient $\mathcal{N}(\chi) \cap \mathcal{N}(\phi) = \emptyset$, et les champs de $\mathbb{N}$ n'en dépendent pas. Si $\exists x \in \mathcal{N}(\mathbb{D})$ tel que $x \in \wr\phi\int$, alors la mise en forme normale échoue.
  - $\mathbb{W}_1 \leftarrow (\phi \circledcirc \psi_{i_0}), \mathbb{X}$
  - $\mathbb{W}_2 \leftarrow (\mathbb{N})$
  - $\mathbb{S} \leftarrow \mathbb{E}_{\mathcal{N}(\psi_{i_0})}(\mathbb{D}, \psi_{m+1}, \ldots, \psi_n)$: on efface toutes les définitions qui def-dépendent de $\mathcal{N}(\psi_{i_0})$.

Le même ordre lexicographique que précédemment, $(\mathrm{Card}\mathbb{W}_1, \mathrm{Card}\mathbb{W}_2)$, permet de montrer la terminaison de l'algorithme : Dans le premier cas, on enlève un champ à $\mathbb{W}_1$, et dans le second au moins un champ (ψ_{i_0}) à $\mathbb{W}_2$, tout en conservant le même nombre de champs dans $\mathbb{W}_1$.

Remarque 3. Notons que la condition de non-dépendance de ϕ dans le deuxième cas de l'algorithme aboutit à un calcul plus restrictif que celui qu'on avait auparavant. En effet, dans l'algorithme précédent $\mathbb{W}_2$ pouvait avoir des cycles de dépendances "temporaires", effacés par une redéfinition avant la fin de l'algorithme. On donne ci-dessous un exemple simple d'espèce qui n'est plus acceptée :

```
species s1 =
  rep ;
  sig d in self → self ;

  let rec a(x in self) = ... !b ...
  and b(y in self) = ... !a ... ;

  let c = !a( !d) ;
end


species s2 inherits s1 =
  let rec b(x) = ... !d ...
  and !d (x) = ... !b ... !c ... ;
```

```
  let rec !a(x) = ... !c ...
  and !c = !b( !d) ;
end
```

Dans cet exemple, le premier algorithme aboutit à un unique champ récursif de 4 méthodes. Le second échoue, car au moment d'ajouter le premier champ de s_2, on a $\mathbb{S} = \{c\}$, et un cycle entre c et d. En pratique, les situations où le second algorithme échoue alors que le premier renvoie un résultat semblent assez pathologiques, et il est toujours possible de transformer la définition de l'espèce fille pour retrouver le premier calcul. En effet, il suffit de placer dans un **let rec** les champs incriminés (c'est à dire dans l'exemple de transformer les deux **let rec** de $s2$ en un seul).

Remarque 4. Comme remarqué plus haut, on pourrait raffiner l'effacement de définitions dans le cas de champs mutuellement récursifs, en utilisant $\mathbb{E}_{\mathcal{N}(\phi)\cap\mathcal{N}(\psi_{i_0})}(\cdot)$ au lieu de $\mathbb{E}_{\mathcal{N}(\psi_{i_0})}(\cdot)$. En pratique, cela permet de considérer les blocs récursifs dans leur ensemble lors des calculs de dépendances, et une situation où on aurait besoin de la définition d'une seule fonction parmi un ensemble de fonctions mutuellement récursives semble assez improbable.

De plus, on peut remarquer que cet algorithme respecte les invariants suivants :

Remarque 5. A chaque passage dans la boucle, tous les champs de $\mathbb{S}$ dépendent de ϕ.

Remarque 6. Avec la notation ci dessus, $\wr\mathbb{N}\int \cap \mathcal{N}(\mathbb{D}) = \emptyset$.

Par ailleurs, on peut facilement étendre les propriétés du précédent algorithme à cette nouvelle version : la seule vraie différence est qu'à chaque étape de la boucle, les champs de $\mathbb{W}_2$ et ceux de $\mathbb{S}$ ont un ordre compatible avec leurs dépendances respectives.

Lemme 41 (ordre des champs) *En reprenant les notations précédentes, on a, à chaque passage dans la boucle,*

$$\wr\psi_i\int \subset \mathcal{N}(\psi_1) \cup \ldots \cup \mathcal{N}(\psi_{i-1})$$

pour tout champ ψ_i de $\mathbb{W}_2$ et $\mathbb{S}$.

Preuve. Par récurrence sur la longueur de la boucle. En supposant que la propriété est vraie à l'étape n, on a deux cas pour l'étape $n+1$.

Si on ajoute ϕ et $\mathbb{S}$ à $\mathbb{W}_2$, il suffit de montrer que ϕ ne dépend que des champs de $\mathbb{W}_2$. En effet, l'hypothèse de récurrence permet de conclure dans les autres cas. Or avec la remarque ci-dessus les éléments de $\mathbb{S}$dépendent de ϕ, donc ϕ lui-même ne peut en dépendre sous peine d'avoir un cycle de dépendances. Sinon, comme on place dans $\mathbb{S}$ tous les champs qui dépendent de ϕ, et avec la remarque 6 on ajoute devant $\mathbb{S}$ des champs dont ne dépend aucun autre champ de $\mathbb{W}_2$, ce qui préserve la propriété. □

Notation : Soient une liste de champs $\mathbb{L}$ fermée et dont l'ordre respecte les dépendances, et ϕ tel que $\wr\phi\int \subset \mathcal{N}(\mathbb{L})$. On notera $\mathbb{L}\circledcirc\phi$ l'ajout du champ ϕ à $\mathbb{L}$, c'est à dire le résultat de l'algorithme ci dessus avec $\mathbb{W}_1 = \mathbb{L}; \phi$.

Lemme 42 (Composition des fusions) *Avec les notations précédentes, si lors d'une étape de l'algorithme ci-dessus on a* $\mathcal{N}(\phi) \cap \mathcal{N}(\mathbb{W}_2) \neq \emptyset$*, alors* $\mathbb{W}_2 \circledcirc \phi = \mathbb{N} \circledcirc (\phi \circledcirc \psi_{i_0}); \mathbb{D}$

Preuve. Par induction sur la taille de $\mathcal{N}(\phi) \cap \mathcal{N}(\mathbb{W}_2)$: Si on n'a qu'un nom en commun, l'étape suivante de l'algorithme ajoute le champ fusionné et $\mathbb{S} = \mathbb{D}$ à la suite de $\mathbb{N}$. Sinon, $\mathcal{N}(\phi) \cap \mathcal{N}(\mathbb{W}_2) > \mathcal{N}(\phi) \cap \mathcal{N}(\mathbb{N})$, et on conclut par récurrence. □

Remarque 7. Si ψ_m dépend de ϕ, on a

$$\{\psi_1, \ldots, \psi_m\} \circledcirc \phi = \{(\{\psi_1, \ldots, \psi_{m-1}\} \circledcirc \phi), \psi_m\}$$

9.5.2 Héritage simple

Soit s_1 une espèce bien formée, et $M_1 = \ll norm(s_1) \gg$ le mixDrec associé à sa forme normale. Soit s_2 une espèce définie par héritage à partir de s_1, de la forme suivante :

$$\textbf{species } s_2 \textbf{ inherits } s_1 = \phi_1 \ldots \phi_n \textbf{ end}$$

On suppose bien entendu que s_2 est elle-même bien formée. Soit $norm(s_2)$ la forme normale associée, et $M_2 = \ll norm(s_2) \gg$. En premier lieu, on peut remarquer que la signature de Drecord correspondant à $norm(s_2)$ est bien une sous signature de $[\![norm(s_1)]\!]$:

Lemme 43 *Avec les notations précédentes, on a*

$$[\![norm(s_2)]\!] :\succ [\![norm(s_1)]\!]$$

Preuve. La principale difficulté de la preuve réside dans le fait que les nouveaux champs de s_2 ne sont pas nécessairement ajoutés à la fin de l'espèce, à cause d'éventuelles dépendances d'une méthode présente dans s_1 et (re)définie dans s_2. Cela va se retrouver dans la dérivation de la relation de sous-signature par un certain nombre d'applications de la règle **_swap**. Plus précisément, on raisonne par induction sur la construction de $norm(s_2)$ à partir de $norm(s_1)$, dans le modèle avec def-dépendances c'est-à-dire où on réordonne les définitions du corps de s_2 en accord avec les dépendances avant de démarrer l'algorithme.

Les premières étapes de construction de $norm(s_2)$ à partir de la liste complète des champs permettent de retrouver $norm(s_1)$. Ensuite, il faut distinguer les différents cas d'ajout d'un nouveau champ. On en a 3 :

Ajout d'un nouveau champ Ce cas correspond à $\mathcal{N}(\phi) \cap \mathcal{N}(\mathbb{W}_2) = \emptyset$ et $\mathbb{S} = \emptyset$. Il suffit dès lors d'appliquer la règle **_lift**, puis de propager l'ajout de champ :

$$\_\mathbf{lift}\ \dfrac{\_\mathbf{C}\ \dfrac{\_\mathbf{C}\ \dfrac{\_\mathbf{E}\ \dfrac{}{\mathcal{N}(\mathbb{W}_2) \vdash \phi{:}\succ\{\}_T}}{\vdots}}{x_1 : \tau_1 \vdash [\![\psi_2 ... \psi_m]\!]{:}\succ[\![\psi_2 ... \psi_m]\!]}}{(x_1 : \tau_1)[\![\psi_2 ... \psi_m ; \phi]\!]{:}\succ[\![\psi_2 ... \psi_m]\!]}}{[\![\mathbb{W}_2 ; \phi]\!]{:}\succ[\![\mathbb{W}_2]\!]}$$

A chaque niveau de l'arbre d'inférence, la transformation des listes de champs en signatures de Drecord par $[\![\cdot]\!]$ est légitime. En effet, avec le lemme 41, on considère bien les différents champs – et leur traduction – dans l'ordre donné par les dépendances.

Ajout d'un champ redéfini Ce cas correspond à $\mathcal{N}(\phi) \cap \mathcal{N}(finalseq) = \emptyset$, et $\mathbb{S} = \neq \emptyset$. Là encore, il n'y a plus qu'à rassembler ajouter les nouveaux champs à la fin de $\mathbb{W}_2$. La règle **_E** permet de prendre en compte plusieurs champs à la fin, donc la dérivation reste identique à la précédente, à l'exception de la racine, qui devient

$$\_\mathbf{E}\ \dfrac{}{\mathcal{N}(\mathbb{W}_2) \vdash [\![\phi ; \mathbb{S}]\!]{:}\succ\{\}_T}$$
$$\vdots$$

Par la remarque 6 et le lemme 41 la transformation de cette liste en Drecord est légitime.

Notons que dans les deux cas précédents, on généralise facilement la propriété à l'ajout de n champs avec la règle **_trans** :

$$\_\mathbf{trans}\ \dfrac{\dfrac{\vdots}{norm(s_2){:}\succ[\![\mathbb{W}_2 ; \phi ; \mathbb{S}]\!]} \qquad \dfrac{D}{[\![\mathbb{W}_2 ; \phi ; \mathbb{S}]\!]}}{[\![norm(s_2)]\!]{:}\succ[\![\mathbb{W}_2]\!]}$$

D étant la dérivation vue précédemment. La partie gauche de la dérivation est obtenue en observant les étapes ultérieures de l'algorithme.

Fusion de deux champs Cette fois-ci on a $\mathcal{N}(\phi) \cap \mathcal{N}(\mathbb{W}_2) \neq \emptyset$. Dans ce cas, l'algorithme transforme $\mathbb{W}_2$ en une liste de champ $\mathbb{N}$ qui n'est pas directement une sous-signature du $\mathbb{W}_2$initial (on place les autres champs dans $\mathbb{S}$). On va donc avoir recours explicitement à la règle **_trans** et introduire une liste de champ intermédiaire, $\mathbb{N} \circledcirc \phi$.

$$\dfrac{\dfrac{\dfrac{S}{[\![(\mathbb{N};\mathbb{D}) \circledcirc \phi]\!]:\succ[\![\mathbb{N} \circledcirc \phi]\!]}\ \_\mathbf{swap}}{[\![\mathbb{W}_2 \circledcirc \phi]\!]:\succ[\![\mathbb{N} \circledcirc \phi]\!]}\ \_\mathbf{swap} \qquad \dfrac{R}{[\![\mathbb{N} \circledcirc \phi]\!]:\succ[\![\mathbb{N}]\!]}}{[\![\mathbb{W}_2 \circledcirc \phi]\!]:\succ[\![\mathbb{W}_2]\!]}\ \_\mathbf{trans}$$

Le passage de $\mathbb{W}_2$ à $\mathbb{N};\mathbb{D}$, qui est une partition de $\mathbb{W}_2$ est obtenu par applications répétées de la règle **_swap**. Par définition de $\mathbb{D}$, il est toujours possible de rassembler les champs qui la constituent en fin de liste (en respectant leur ordre relatif). D'après la remarque 7, la fin de la dérivation, S, est une suite de règles **_C** et **_lift** similaire aux dérivations précédentes.

Par ailleurs, en suivant la preuve du lemme 42, la dérivation R est soit une application directe du paragraphe précédent, soit une nouvelle dérivation de fusion, avec un nombre de champs en conflit strictement inférieur, ce qui garantit la terminaison du processus. □

Avant de passer aux mixDrecs proprement dit, il faut examiner ce qui se passe au niveau des arbres d'étiquettes P_1 et P_2 de M_1 et M_2. Plus précisément, on peut retranscrire le fait qu'on efface le moins de définitions possible par une propriété sur les étiquettes de P_2 de M_2 et P_1 de $\Uparrow_{[\![norm(s_2)]\!]} M_1$.

Lemme 44 (Liste de contrôle) *Avec les notations précédentes, on a*

$$[\![P_2; P_1]\!] = 0; ...; 0$$

$[\![P_2; P_1]\!]$ *étant la liste de contrôle de la fusion (voir p.169).*

Preuve. On travaille par induction sur la longueur de la liste d'étiquette. Cette fois-ci tenir compte des effacements de définitions dus aux def-dépendances. D'après la définition de *Pre*, il suffit de montrer qu'on n'a jamais $P_2 = \mathbb{A}_x p_2$ et $P_1 = \mathbb{M}_x p_{11} p_{12}$. Supposons que ce soit le cas, et que x soit la première étiquette abstraite dans P_2 et manifeste dans P_1 : d'après le lemme 5.7 de , $y \in [\![x]\!]_{s_1}$ tel qu'une des deux conditions suivantes soit vérifiée : $y \notin \mathcal{D}(s_2)$ ou $changed(y, x)$.

Pour que x def-dépende de y, il faut que y soit définie, donc le premier cas est en contradiction avec la condition sur x. On a donc $changed(y, x)$. Comme on ne considère qu'un cas d'héritage simple, cette condition devient $\mathcal{B}_{s_2}(y) \neq \mathcal{B}_{s_1}(y)$, et au niveau de y, on a :

$$[\![\mathbb{M}_y p_{21} p_{22}; \mathbb{M}_y p_{11} p_{12}]\!] = 0; [\![p_{22} p_{11}]\!]$$

Pour qu'on ait un '1' dans la liste de contrôle à la position correspondant à x, il faut qu'on ait un nœud manifeste à cette position, soit $\mathbb{M}_x \in p_{11}$. Or, x def-dépend de y dans s_1, ce nœud manifeste est en contradiction avec la définition de $\mathbb{P}(s_1)$: ce cas ne peut se produire. □

Au niveau des espèces, la relation entre s_1 et s_2 se retrouve dans les mixDrecs M_1 et M_2 sous la forme suivante

Théoreme 15 (Correction de l'héritage simple)
Avec les notations précédentes, on a la propriété :

$$M_2 \oplus_{\prod \mathbb{P}(norm(s_2)); \mathbb{P}(norm(s_1)) \prod} \left(\Uparrow_{[\![norm(s_2)]\!]} M_1 \right) = M_2$$

Preuve. Tout d'abord, il convient de remarquer que les opérations faites sur les deux mixDrecs sont légitimes. En premier lieu, examinons le plongement de M_1 dans $[\![norm(s_2)]\!]$. La signature de M_1 est $[\![norm(s_1)]\!]$ (théorème 14), donc avec le lemme 43 et la définition de $\Uparrow$, cette opération est correcte. De même, comme M_2 a pour signature $[\![norm(s_2)]\!]$, la fusion des deux mixDrecs est correcte. La figure 9.4 récapitule tous les cas possibles de fusion des deux mixDrecs. Comme on le voit, il faut en fait prouver que la liste de contrôle n'est composée que de 0, c'est à dire qu'on sélectionne toujours la définition de gauche. Le lemme 44 permet donc de conclure.

FIG. 9.4 – Fusion des deux mixDrecs

			$M_2 \oplus_l \Uparrow_{[\![norm(s_2)]\!]}$
M_2	$\Uparrow_{[\![norm(s_2)]\!]} M_1$	l	M_1
$\{\!\{\}\!\}$	$\{\!\{\}\!\}$	l	$\{\!\{\}\!\}$
$\{\!\{x : \tau; f\}\!\}$	$\{\!\{x : \tau; f_1\}\!\}$	$b; l$	$\{\!\{x : \tau; f \oplus_l f_1\}\!\}$
$\{\!\{x : \tau; f\}\!\}$	$\{\!\{x : \tau = e_1; f_1\}\!\}$	$0; l$	$\{\!\{x : \tau; f \oplus_l f_1\}\!\}$
$\{\!\{x : \tau; f\}\!\}$	$\{\!\{x : \tau = e_1; f_1\}\!\}$	$1; l$	$\{\!\{x : \tau = e_1; f \oplus_l f_1\}\!\}$ Impossible
$\{\!\{x : \tau = e; f\}\!\}$	$\{\!\{x : \tau; f_1\}\!\}$	$b; l$	$\{\!\{x : \tau = e; f \oplus_l f_1\}\!\}$
$\{\!\{x : \tau = e; f\}\!\}$	$\{\!\{x : \tau = e_1; f_1\}\!\}$	$b; l$	$\{\!\{x : \tau = e; f \oplus_{l_1} f_1\}\!\}$

□

9.5.3 Héritage multiple

Les résultats de la section précédente sont aisément étendus au cas de l'héritage multiple, par récurrence sur le nombre d'espèces parents. En effet, soient s_1 et s_2 deux espèces, de forme normale $norm(s_1) = \phi_1 \ldots \phi_m$ et $norm(s_2) = \psi_1 \ldots \psi_n$ respectivement. Alors, les deux espèces suivantes :

species s **inherits** s_1, s_2 = **end**

et

species t **inherits** $s_1 = \psi_1 ; \ldots ; \psi_n$ **end**

ont la même forme normale. En effet, l'algorithme de mise en forme normale démarre dans les deux cas avec la même liste initiale, c'est-à-dire $\mathbb{W}_1 = \phi_1 \ldots \phi_n, \psi_1 \ldots \psi_n$. Dès lors, on va travailler avec l'espèce t pour montrer que la fusion des deux espèces est conforme à celle des mixdRecs correspondants.

Par ailleurs, il convient de remarquer que l'ordre d'héritage n'influe que sur les définitions de méthodes, et pas sur leurs déclarations. Plus précisément,

Lemme 45 (Commutation des signatures)
Soient s_1 et s_2 deux espèces, telles que, $\forall x \in \mathcal{N}(s_1) \cap \mathcal{N}(s_2), \mathcal{T}_{s_1}(x) = \mathcal{T}_{s_2}(x)$ alors les deux espèces suivantes u et v :

species `u` **inherits** s_1, s_2 = **end**
species `v` **inherits** s_2, s_1 = **end**

se traduisent en des signatures de Drecord équivalentes :

$$[\![u]\!] :\succ [\![v]\!] \text{ et } [\![v]\!] :\succ [\![u]\!]$$

Preuve. La condition sur les types des méthodes communes à s_1 et s_2 garantit que u et v sont bien formées, donc admissibles. De plus, on peut remarquer que les deux espèces déclarent les mêmes champs : $\mathcal{N}(u) = \mathcal{N}(v)$. Avec le théorème 14, $[\![u]\!]$ et $[\![v]\!]$ sont définis sur la même liste d'étiquette. De plus, il vient $\forall x \in \mathcal{N}(u), \mathcal{T}_u(x) = \mathcal{T}_v(x)$. Dès lors, on peut passer de $[\![u]\!]$ à $[\![v]\!]$ (et vice-versa) par une suite d'application de la règle **_swap**. □

Notons que dans le cas général, on n'a toutefois pas l'égalité des deux signatures, car l'ordre des champs n'est pas forcément le même dans u et dans v.

Théoreme 16 (Correction de l'héritage multiple)
Soient $\{s_i\}_{i=1..2}$ des espèces bien formées. et s définie par

species s **inherits** s_1, s_2 = **end**

On a l'égalité suivante :

$$\ll s \gg = (\Uparrow_{[\![norm(s)]\!]} \ll s_2 \gg) \oplus_{\Pi \mathbb{P}(norm(s)); \mathbb{P}(norm(s)) \Pi} (\Uparrow_{[\![norm(s)]\!]} \ll s_1 \gg)$$

Preuve. On commence par montrer que la notation $\Uparrow_{[\![norm(s)]\!]} \ll s_2 \gg$ est légitime, c'est à dire que $[\![norm(s)]\!] :\succ [\![norm(s_2)]\!]$. Pour cela, il suffit d'utiliser le lemme 45 pour permuter les deux héritages, et de reprendre la première partie de la preuve du théorème 15.

De plus, il convient de remarquer que $\ll s \gg$ et la fusion des deux espèces précédentes partagent la même signature. Il ne reste plus qu'à montrer que ces deux mixdRecs vérifient le même prédicat Pre, et, dans le cas de nœuds manifestes, donnent la même définition. Pour cela, on raisonne par récurrence sur la taille de la signature de DRecord $[\![s]\!]$: le cas où la signature est vide est trivial. Pour une signature de taille $n+1$, de premier champ x, on a les cas suivants :

x est abstrait dans $norm(s)$ Cette situation se retrouve si x est abstrait dans s_1 et s_2, ou si x est défini dans s_1, et qu'il se produit un effacement. Dans le premier cas, la fusion donnera bien un champ abstrait. Dans le second cas, on se retrouve avec l'opération suivante :

$$\{\!\{x : \tau; f_2\}\!\} \oplus_{b;l} \{\!\{x : \tau = e; f_1\}\!\}$$

Le résultat contient un champ abstrait si et seulement si $b = 0$. Comme dans le lemme 44, on montre que c'est le cas en utilisant une méthode $y \in \llbracket x \rrbracket_{s_1}$, vérifiant le prédicat $changed(y, x)$: lorsqu'on crée la liste de contrôle de la fusion, on se retrouve, au niveau de y dans la situation suivante :

$$\lceil\!\lceil \mathbb{M}_y p_{21} p_{22}; \mathbb{M}_y p_{11} p_{12} \rceil\!\rceil = 0; \lceil\!\lceil p_{22} p_{11} \rceil\!\rceil$$

Par hypothèse, on a $\mathbb{A}_x \in p_{11}$, et on a donc $b = 0$.

x est défini dans $norm(s)$ Cette fois-ci, il y a trois possibilités :

1. $\ll s_2 \gg = \{\!\{x : \tau = e_2; f_2\}\!\}$, $\ll s_1 \gg = \{\!\{x : \tau; f_1\}\!\}$
2. $\ll s_2 \gg = \{\!\{x : \tau; f_2\}\!\}$, $\ll s_1 \gg = \{\!\{x : \tau = e_1; f_1\}\!\}$
3. $\ll s_2 \gg = \{\!\{x : \tau = e_2; f_2\}\!\}$, $\ll s_1 \gg = \{\!\{x : \tau = e_1; f_2\}\!\}$

Dans le premier cas et le troisième cas, la fusion produit un champ manifeste $x : \tau = e_2$, ce qui correspond aussi à la mise en forme normale. Dans le second cas, il faut montrer que la liste de contrôle est de la forme $1; l$. Pour cela, il faut cette fois-ci considérer toutes les méthodes $y \in \llbracket x \rrbracket_{s_1}$. Par hypothèse, aucune de ces méthodes n'est redéfinie dans s_2 (sinon x ne serait pas défini dans $norm(s)$). La liste de contrôle comporte donc des 1 pour chacune des positions y. De plus, on sélectionne à chaque fois la branche manifeste, qui contient donc $\mathbb{M}_x$ pour poursuivre l'établissement de la liste de contrôle. Tous les autres nœuds manifestes intermédiaires contiennent $\mathbb{M}_x$ dans les deux branches. Lorsqu'on se trouve à la profondeur de x, on a donc la situation suivante :

$$\lceil\!\lceil \mathbb{A}_x p; \mathbb{M}_x p_{11} p_{21} \rceil\!\rceil = 1; \lceil\!\lceil p; p_{21} \rceil\!\rceil$$

□

Chapitre 10

Générateurs de méthodes et mixDrecs

10.1 Introduction

Dans ce chapitre, on montre comment extraire d'un mixDrec donné les informations relatives à un champ particulier de ce mixDrec. En particulier, on verra que la notion de chemin à l'interieur d'un mixDrec permet de définir les dépendances d'une définition ou d'une déclaration vis à vis des autres champs. Dans un second temps, on montre que ces informations sont équivalentes à celles données par la mise en forme normale d'une espèce, et sa traduction vers Coq. En particulier, on va montrer que les notions d'environnement minimal et de générateur de méthode, telles qu'elles sont décrites dans le chapitre 6 sont en fait des branches particulières du mixDrec représentant l'espèce courante.

10.2 MixDrecs et générateurs de méthodes

Dans cette section, on considère un mixDrec $\mathcal{M}$ bien formé et un champ d'étiquette x de $\mathcal{M}$, tel qu'il existe au moins un nœud manifeste $\mathbb{M}_x$ dans $\mathcal{M}$.

10.2.1 Chemin de définition

Définition 90 (parcours d'un mixDrec)
On appelle parcours *de $\mathcal{M}$ toute suite de 0 et de 1 de longueur inférieure ou égale à la profondeur de $\mathcal{M}$. Ce parcours est défini sur les informations de présence Pre de $\mathcal{M}$:*

$$\begin{array}{ll}
\not\perp(\emptyset, \mathbb{M}_x p_1 p_2) = \mathbb{M}_x\,\mathbb{E}\,\mathbb{E} & \not\perp(\emptyset, \mathbb{A}_x p) = \mathbb{A}_x\,\mathbb{E} \\
\not\perp(0; l, \mathbb{M}_x p_1 p_2) = \mathbb{A}_x\ \not\perp(l, p_1) & \not\perp(0; l, \mathbb{A}_x p) = \mathbb{A}_x\ \not\perp(l, p) \\
\not\perp(1; l, \mathcal{M}M_x p_1 p_2) = \mathbb{M}_x\ \not\perp(l, p_1)\ \not\perp(l, p_2) & \not\perp(1; l, \mathbb{A}_x p) = \mathbb{A}_x\ \not\perp(l, p)
\end{array}$$

Un parcours aboutissant à un nœud manifeste $\mathbb{M}_x$ *est appelé* chemin de définition *de* x. *Un parcours aboutissant à un nœud abstrait* $\mathbb{A}_x$ *est appelé* chemin de déclaration *de* x. *Enfin, les* chemins aboutissant à x *représentent l'ensemble des* chemins de définition *et* de déclaration *de* x.

Remarque 1. Tous les chemins aboutissant à x dans un mixDrec $\mathcal{M}$ donné ont même longueur, qui sera noté $\| x \|_{\mathcal{M}}$ dans la suite. De plus on peut munir cet ensemble d'une relation d'ordre totale (ordre lexicographique).
Preuve. Immédiat par récurrence sur la profondeur de $\mathcal{M}$ □

Définition 91 (Chemin minimal) *Soit* x *une étiquette telle qu'il exite un champ manifeste* $\mathbb{M}_x$ *dans* $\mathcal{M}$. *L'ensemble* $\mathcal{D}$ *des chemins de définition de* x *admet un minimum pour l'ordre lexicographique. Cet élément, noté* $\rightsquigarrow^{\mathcal{M}}(x)$ *est appelé* chemin minimal de définition de x.

Remarque 2. Les chemins plus grand que $\rightsquigarrow^{M}(x)$ ne sont pas forcément tous des chemins de définition de x.

Lemme 46 (chemin de définition et sous-signature)
Soient $\mathcal{M}_1$ *et* $\mathcal{M}_2$ *deux mixDrecs bien formés tels que* $\mathcal{M}_1 \gtrdot M_2$. *Pour tout* x *et tout* p *chemin de définition de* x *dans* $\mathcal{M}_2$, p *est un chemin de définition de* x *dans* $\mathcal{M}_1$

Preuve. Par induction sur la dérivation de $\mathcal{M}_1 \gtrdot \mathcal{M}_2$. On ne détaille que le cas où le premier nœud de $\mathcal{M}_1$ est manifeste alors que le premier nœud de $\mathcal{M}_2$ est abstrait. Les autres cas sont triviaux avec la définition de $\gtrdot$.

$\mathcal{M}_1 = \{\!\{y : T = e; f_1; m\}\!\}$, $\mathcal{M}_2 = \{\!\{y : T; f_2\}\!\}$. Avec x une étiquette de $\mathcal{M}_2$ et p un chemin de définition de x dans $\mathcal{M}_2$ (donc $x \neq y$, sinon il n'existerait pas de chemin de définition). p est de la forme α, p_1, avec $\alpha \in \{0, 1\}$.

$\alpha = 0$ **:** $\forall z : T$, p_1 est un chemin de définition de x dans $(f_2\ z)$. Comme de plus on a $(f_1\ z) \gtrdot (f_2\ z)$ par définition de $\gtrdot$, on peut conclure avec l'hypothèse de récurrence.

$\alpha = 1$ **:** on a toujours p_1 chemin de définition de x dans $(f_2\ z)$ pour tout z, et en particulier dans $(f_2\ e)$. Par hypothèse, $m \gtrdot (f_1\ e)$ et $(f_1 e) \gtrdot (f_2\ e)$, donc en appliquant deux fois l'hypothèse de récurrence on peut conclure comme précédemment. □
Remarque 3. Le chemin minimal associe toujours un 0 aux nœuds abstraits.

Lemme 47 *Avec* x *un champ défini d'un mixDrec* M, *s'il existe un chemin* p *de définition de* x *tel que pour tout chemin* p' *de définition de* x, $p_i = 1 \Rightarrow p'_i = 1$, *alors* $p = \rightsquigarrow^{M}(x)$

Preuve. Soit p' un chemin de définition de x. Supposons qu'il soit différent de p, et notons i le premier indice tel que les deux chemins diffèrent. On a alors $p_i = 0$ et $p'_i = 1$ (l'inverse est impossible d'après l'hypothèse de l'énoncé sur p). Donc p est strictement plus petit que p'. Comme p est lui-même un chemin de définition de x par hypothèse, c'est bien le chemin minimal. □

Remarque 4. L'existence d'un tel p n'est nullement garantie dans un mixDrec quelconque. On verra ci-dessous que c'est le cas dans les mixDrecs équilibrés (lemme 49).

Définition 92 (tronquage de signature) *Soit $\mathcal{M}$ un mixDrec et p un parcours sur $\mathcal{M}$ de longueur l. On note $s(\mathcal{M}) \mid_p$ le Drecord obtenu en tronquant s à ses l premiers éléments*

10.2.2 MixDrec de définition de x

Il reste à voir comment passer des chemins de définition à des λ-termes correspondant à nos méthodes. Comme S. Boulmé l'a remarqué dans sa thèse, un tel λ-terme ne peut venir seul : il faut l'accompagner d'un certain contexte, mêlant des abstractions et des définitions. En d'autre termes, ce contexte de définition peut-être lui même vu comme un mixDrec.

Dans cette section, on s'intéresse à la définition de x dans un mixDrec $\mathcal{M}$. Pour cela, on va extraire des mixDrecs de $\mathcal{M}$, de dernier champ x, en s'appuyant sur les chemins de définition de x.

Définition 93 (Contexte de définition) *Soient $\mathcal{M}$ un mixDrec, x une étiquette de $\mathcal{M}$ telle qu'il existe un champ manifeste $\mathbb{M}_x$ dans $\mathcal{M}$ et p un chemin de définition de x. Le contexte de définition associé à p, noté $\mathbf{\Gamma}_{\mathcal{M}} \vdash \mathbf{p}$ est défini de la manière suivante :*

$$
\begin{array}{lcl}
\mathbf{\Gamma}_{\mathcal{M}} \vdash \emptyset & = & \{\!\{\}\!\} \\
\mathbf{\Gamma}_{\{\mathbf{x:T=e;f;m}\}} \vdash \mathbf{0;l} & = & \{\!\{x : T; \lambda x.\mathbf{\Gamma_{fx}} \vdash \mathbf{l}\}\!\} \\
\mathbf{\Gamma}_{\{\mathbf{x:T;f}\}} \vdash \mathbf{0;l} & = & \{\!\{\{\!\{x : T; \lambda x.\mathbf{\Gamma_{(fx)}} \vdash \mathbf{l}\}\!\}\!\} \\
\mathbf{\Gamma}_{\{\mathbf{x:T=e;f;m}\}} \vdash \mathbf{1;l} & = & \{\!\{\{\!\{x : T = e; \lambda x.\mathbf{\Gamma_{(fx)}} \vdash \mathbf{l}; \mathbf{\Gamma_m} \vdash \mathbf{l}\}\!\}\!\} \\
\mathbf{\Gamma}_{\{\mathbf{x:T;f}\}} \vdash \mathbf{1;l} & = & \{\!\{x : T; \lambda x.\mathbf{\Gamma_{fx}} \vdash \mathbf{l}\}\!\}
\end{array}
$$

Le contexte minimal de définition, $\mathbf{\Gamma}_{\mathcal{M}} \vdash \rightsquigarrow(x)$ *est noté* $\widetilde{\mathbf{\Gamma_M} \vdash \mathbf{x}}$.

Remarque 5. Tous les contextes de définition de x ont la même profondeur (cf. remarque 1)

Lemme 48 (Bonne formation du contexte) *Avec les notations précédentes, pour tout p chemin de définition de x, on a*

$$\mathbf{\Gamma}_{\mathcal{M}} \vdash \mathbf{p} : Mix_{s(\mathcal{M})|_p, \lambda(p,\mathcal{M})}$$

Preuve. Par induction sur la longueur de p. le cas $p = \emptyset$ est trivial. Sinon, on travaille par cas sur p et le premier nœud de $\mathcal{M}$:

$p = 0; l$ **et** $\mathcal{M} = \{\!\{x : T = e; f; m\}\!\} : Mix_{\{x:T;F\};\mathbb{M}q_1\,q_2}$

$$\frac{\lambda x.\mathbf{\Gamma_{(fx)}} \vdash \mathbf{l} : (x : T) Mix_{s(fx)|_l;\lrcorner(l,q_1)}}{\mathbf{\Gamma_{\mathcal{M}}} \vdash \mathbf{p} = \{\!\{x : T; \lambda x.\mathbf{\Gamma_{(fx)}} \vdash \mathbf{l}\}\!\} : Mix_{s(\mathcal{M})|_p;\lrcorner(p,\mathcal{M})}}$$

$p = 0; l$ **et** $\mathcal{M} = \{\!\{x : T; f\}\!\}$

$$\frac{\lambda x.\mathbf{\Gamma_{(fx)}} \vdash \mathbf{l} : (x : T) Mix_{s(fx)|_l;\lrcorner(l,q_1)}}{\mathbf{\Gamma_{\mathcal{M}}} \vdash \mathbf{p} = \{\!\{x : T; \lambda x.\mathbf{\Gamma_{(fx)}} \vdash \mathbf{l}\}\!\} : Mix_{s(\mathcal{M})|_p;\lrcorner(p,\mathcal{M})}}$$

$p = 1; l$ **et** $\mathcal{M} = \{\!\{x : T = e; f; m\}\!\}$

$$\frac{\begin{array}{c}\lambda x.\mathbf{\Gamma_{(fx)}} \vdash \mathbf{l} : (x : T) Mix_{s(fx)|_l;\lrcorner(l,q_1)} \\ \mathbf{\Gamma_m} \vdash \mathbf{l} : Mix_{s(m)|_l;\lrcorner(l,q_1)} \qquad e : T \qquad \mathbf{\Gamma_m} \vdash \mathbf{l} \geqslant \mathbf{\Gamma_{(fe)}} \vdash \mathbf{l}\end{array}}{\mathbf{\Gamma_{\mathcal{M}}} \vdash \mathbf{p} = \{\!\{x : T; \lambda x.\mathbf{\Gamma_{(fx)}} \vdash \mathbf{l}\}\!\} : Mix_{s(\mathcal{M})|_p;\lrcorner(p,\mathcal{M})}}$$

$p = 0; l$ **et** $\mathcal{M} = \{\!\{x : T; f\}\!\}$

$$\frac{\lambda x.\mathbf{\Gamma_{(fx)}} \vdash \mathbf{l} : (x : T) Mix_{s(fx)|_l;\lrcorner(l,q_1)}}{\mathbf{\Gamma_{\mathcal{M}}} \vdash \mathbf{p} = \{\!\{x : T; \lambda x.\mathbf{\Gamma_{(fx)}} \vdash \mathbf{l}\}\!\} : Mix_{s(\mathcal{M})|_p;\lrcorner(p,\mathcal{M})}}$$

□

On pourrait s'attendre à ce que le chemin minimal de définition de x dans $\mathcal{M}$ se traduise dans un contexte "minimal", en particulier en ce qui concerne les définitions présentes dans $\Gamma_{\rightsquigarrow\mathcal{M}(x)}$. Malheureusement, ce n'est pas le cas en général. Par exemple, si on prend le mixDrec suivant, dont les feuilles ont été enlevées pour plus de lisibilité :

$$F = \left\{\!\!\left\{ \begin{array}{l} a : bool = true; \\ \quad [a] \left\{\!\!\left\{ \begin{array}{l} b : bool = (not\ a);\ \ [b]\ \{\!\{c : bool\}\!\}; \\ \qquad\qquad\qquad\qquad \{\!\{c : bool = a \vee b; [c]\}\!\} \end{array} \right\}\!\!\right\} \\ \quad \left\{\!\!\left\{ \begin{array}{l} b : bool = (not\ a);\ \ [b]\ \{\!\{c : bool = a \vee b\}\!\}; \\ \qquad\qquad\qquad\qquad \{\!\{c : bool = a \vee b\}\!\} \end{array} \right\}\!\!\right\} \end{array} \right\}\!\!\right\}$$

Dans ce mixDrec, le nœud de c le plus à gauche est abstrait, ce qui est correct (il s'agit bien d'une vue moins définie que son voisin de droite), mais pose un problème dans la définition du contexte minimal de définition. Par définition, on a $\rightsquigarrow^F (c) = 0; 1; 1$, soit $\widetilde{\mathbf{\Gamma_F} \vdash \mathbf{c}} = \{\!\{a : bool; \{\!\{b : bool = (not\ a)\}\!\}\}\!\}$. Mais le chemin $p = 1; 0; 1$ est également un chemin de définition de c, et $\mathbf{\Gamma_F} \vdash \mathbf{p} = \{\!\{a : bool = true; \{\!\{b : bool\}\!\}\}\!\} \not> \widetilde{\mathbf{\Gamma_m} \vdash \mathbf{c}}$. Or en pratique, cette notion d'environnement minimal est importante, puisqu'elle garantit qu'on ne va pas effacer trop de définitions (qu'il faudra redonner ensuite) lors d'une

redéfinition. On va donc raffiner la notion de mixDrec pour tenir compte de ce problème. Nous montrerons ensuite que cette nouvelle notion répond bien à notre problème, et que la traduction d'une espèce en forme normale respecte les conditions imposées par ce mixDrec. Nous pourrons dès lors montrer que la notion de contexte minimal introduite lors de la traduction en Coq correspond bien à celle de contexte minimal au sein d'un mixDrec.

10.3 MixDrecs équilibrés

10.3.1 Définition

D'après l'exemple précédent, on peut se faire une idée intuitive des conditions à remplir pour que $\widetilde{\mathbf{\Gamma}_{\mathcal{M}} \vdash \mathbf{x}}$ soit effectivement un contexte minimum. Lorsqu'on est sur un nœud manifeste, la relation $\geqslant$ n'est pas suffisante. Il faut en outre que les effacements de définition qu'on effectue pour passer du mixDrec de droite à celui de gauche soient cohérents entre eux. Ainsi, on ne doit pas trouver un nœud abstrait là où la définition correspondante dans le mixDrec de droite serait bien typée.

Par exemple, dans le mixDrec précédent, dans un environnement où a et b sont deux variables booléennes, la définition de $c = a \vee b$ est parfaitement admissible, et il n'y a aucune raison d'avoir un nœud abstrait à gauche. Cette condition peut se formaliser par une condition sur le chemin minimal.

Définition 94 (MixDrec équilibré) *Un mixDrec bien formé $\mathcal{M}$ est dit équilibré s'il vérifie une des trois conditions suivantes :*

$$\{\!\{\}\!\} \text{ est équilibré} \qquad \frac{\forall x : T,\ (f\ x) \text{ est équilibré}}{\{\!\{x : T; f\}\!\} \text{ est équilibré}}$$

$$\frac{m \text{ est équilibré} \qquad \begin{array}{c}\forall x : T (f\ x) \text{ est équilibré} \\ \forall y \text{ tel que } \rightsquigarrow^{\mathcal{M}} (y) = 0; p,\ \rightsquigarrow^{m} (y) = p\end{array}}{\mathcal{M} = \{\!\{x : T = e; f; m\}\!\} \text{ est équilibré}}$$

Remarque 6. Dans la dernière règle, avec le lemme 48, quel que soit le mixDrec (bien formé), p est un chemin de définition de y dans m, Par contre, ce n'est pas forcément le chemin minimum. (par exemple les deux chemins de l'exemple précédent)

10.3.2 Environnement minimal dans un mixDrec équilibré

On peut maintenant obtenir une caractérisation de la position des 1 dans le chemin minimal de définition de x, avec le lemme suivant :

Lemme 49 (Chemin minimal) *Soit $\mathcal{M}$ un mixDrec équilibré. Avec les notations de la def.91, et i un entier inférieur à la longueur des chemins*

aboutissant à x. Le i-ème élément de $\rightsquigarrow^{\mathcal{M}}(x)$ est un 1 si et seulement si tous *les chemins de définition de x ont un 1 comme i-ème élément.*

Intuitivement ce lemme dit que lorsqu'on parcourt le chemin de définition minimal de x, on ne trouve que les définitions dont x def-dépend.

Preuve. L'implication réciproque est triviale, $\rightsquigarrow^{\mathcal{M}}(x)$ étant lui-même un chemin de définition de x.

Pour l'implication directe, on travaille par récurrence sur i : le cas $i = 1$ est immédiat avec la définition de $\rightsquigarrow^{\mathcal{M}}(x)$: S'il commence par un 1, tous les chemins de définitions de x, plus grand que que $\rightsquigarrow^{\mathcal{M}}(x)$ commencent par 1.

Sinon, on raisonne pas cas sur le premier nœud de $\mathcal{M}$ et le premier élément de $\rightsquigarrow^{\mathcal{M}}(x)$ et de p. Les cas $\mathcal{M} = \{\!\{y : T; f\}\!\}$ et $fst(\rightsquigarrow^{\mathcal{M}}(x)) = fst(p)$ sont triviaux. De plus, $fst(\rightsquigarrow^{\mathcal{M}}(x)) = 1$, $fst(p) = 0$ est impossible par définition de $\rightsquigarrow^{\mathcal{M}}(x)$. Il reste le cas $\mathcal{M} = \{\!\{y : T = e; f; m\}\!\}$, avec $\rightsquigarrow^{\mathcal{M}}(x) = 0; l$, $p = 1; p_1$. Par définition, l est le chemin minimal de définition de x dans $(f\ e)$, donc comme $\mathcal{M}$ est équilibré l est aussi le chemin minimal de définition de x dans m. Comme p_1 est un chemin de définition de x dans m, on peut conclure par récurrence. □

Corollaire 5 (chemin de déclaration) *Dans un mixDrec équilibré, avec x une étiquette définie dans $\mathcal{M}$ et i tel que le i-ème élément de $\rightsquigarrow^{\mathcal{M}}(x)$ soit un 1. Alors, tout chemin de déclaration p aboutissant à x et dont le i-ème élément est un 0 est un chemin de déclaration de x.*

De plus, ce lemme permet de caractériser les mixDrecs bien formés, comme on le montre dans le lemme suivant.

Lemme 50 (Caractérisation des mixDrecs équilibrés)
Réciproquement, si tous les chemins $\rightsquigarrow^{\mathcal{M}}(x)$, pour toutes les méthodes x définies dans $\mathcal{M}$ vérifient la propriété du lemme précédent, alors $\mathcal{M}$ est équilibré

Preuve. Par induction sur la taille de $\mathcal{M}$. Si $\mathcal{M}$ est vide, il est équilibré par définition.

Si $\mathcal{M} = \{\!\{y : T; f\}\!\}$, $\forall x, \rightsquigarrow^{\mathcal{M}}(x) = 0; l$ avec la remarque 1, et l est le chemin minimal de x dans $(f\ z)$ (pour tout $z \in T$). $(f\ z)$ vérifie donc aussi la condition sur les chemins minimaux, et on peut appliquer l'hypothèse de récurrence et la condition d'équilibre.

Sinon, $\mathcal{M} = \{\!\{y : T = e; f; m\}\!\}$. Comme ci-dessus, $(f\ z)$ vérifie la condition sur les chemins minimaux, et est donc un mixDrec équilibré. De même, pour tous les champs x tels que $\rightsquigarrow^{\mathcal{M}}(x) = 1; l$, l est le chemin minimal de x dans m, et pour tout i tel que $l_i = 1$, tout chemin de définition p de x dans m vérifie bien $p_i = 1$ (Sinon cette hypothèse ne serait pas vérifiée pour m).

Soit maintenant x une étiquette telle que $\rightsquigarrow^{\mathcal{M}}(x) = 0; l_1$. l_1 est un chemin de définition de x dans $(f\ e)$ donc dans m. Soit l_2 un chemin de

définition de x dans m. On va montrer $l_{1i} = 1 \Rightarrow l_{2i} = 1$. Cela permettra de montrer à la fois que m est bien formé (avec l'hypothèse de récurrence) et que l_1 est le chemin minimal de définition de x dans m (immédiat en regardant le première position i où l_2 et l_1 diffèrent : on a nécessairement $l_{2i} = 1$ et $l_{1i} = 0$). Or comme l_2 est un chemin de définition de x dans m, $p = 1; l_2$ est un chemin de définition de x dans $\mathcal{M}$. On a dès lors $l_{1i} = \rightsquigarrow^{\mathcal{M}}(x)_{i+1} = 1$, donc par hypothèse sur $\mathcal{M}$, $p_{i+1}(= l_{2i}) = 1$ □

Remarque 7. D'après le lemme 47, il suffit d'exhiber pour tout x un chemin p_x tel que pour tout chemin de définition de x, $p_{xi} = 1 \Rightarrow p_i = 1$.

Lemme 51 (contexte minimal)

Avec les notations précédentes, si $\mathcal{M}$ est un mixDrec équilibré, $\widetilde{\mathbf{\Gamma}_{\mathcal{M}} \vdash \mathbf{x}}$ est minimal pour la relation $\succcurlyeq$. Plus précisément, pour tout p chemin de définition de x

$$\mathbf{\Gamma}_{\mathcal{M}} \vdash \mathbf{p} \succcurlyeq \widetilde{\mathbf{\Gamma}_{\mathcal{M}} \vdash \mathbf{x}}$$

Preuve. Par récurrence sur la longueur l de p (et de $\rightsquigarrow^{\mathcal{M}}(x)$). Le cas de base avec un seul champ est trivial. Si $l > 1$, considérons les différents cas possibles :

$\rightsquigarrow^{\mathcal{M}}(x) = 0; l_1$ **et** $p = 0; l_2$ **ou** $\rightsquigarrow^{\mathcal{M}}(x) = 1; l_1$ **et** $p = 1; l_2$ Immédiat par récurrence (le premier champ des deux contextes est identique)

$\rightsquigarrow^{\mathcal{M}}(x) = 1; l_1$ **et** $p = 0; l_2$ Impossible avec le lemme 49

$\rightsquigarrow^{\mathcal{M}}(x) = 0; l_1$ **et** $p = 1; l_2$ Il vient

$$\frac{m \succcurlyeq (f_1 e) \qquad (y : T)(f_1 y) \succcurlyeq (f_2 y)}{\{\!\{x : T = e; f_1; m\}\!\} \succcurlyeq \{\!\{x : T; f_2\}\!\}}$$

La première prémisse est une conséquence de la bonne formation de $\mathbf{\Gamma}_{\mathcal{M}} \vdash \mathbf{p}$. La seconde se déduit de l'hypothèse de récurrence, et la condition d'équilibre sur $\mathcal{M}$, l_1 étant le chemin minimum de définition de x dans $(f\ e)$ et donc dans m. □

10.4 Forme normale et mixDrecs

Dans cette section on s'attache à montrer que les générateurs de méthodes obtenus à partir de la mise en forme normale et les contextes minimaux des mixDrecs traduits sont équivalents. Un point important de cette section est que la traduction d'une espèce en forme normale produit un mixDrec *équilibré*. En effet, ce résultat permet de déduire deux points importants pour les générateurs de méthodes :

- Ils sont "minimaux" dans le sens où on utilise un minimum de variables locales pour tenir compte des def-dépendances
- Leur éventuel effacement (lors d'opérations d'héritage) ne dépend que du chemin de définition minimum correspondant : le reste du mixDrec n'entre pas en ligne de compte.

Cette dernière remarque montre qu'on peut bien garder une forme "linéaire" pour les espèces en forme normale, dans la mesure où on sait calculer les def- et les decl- dépendances. En effet, on pourra toujours recalculer les opérations d'effacement et de redéfinition nécessaires lors du passage d'une espèce à l'autre, sans avoir à déplier explicitement la structure du mixDrec sous-jacent.

Soit s une espèce bien formée et $\tilde{s}$ sa forme normale. Avant de montrer que $\ll\tilde{s}\gg$ est équilibré, on va travailler sur $\mathbb{P}(\tilde{s})$, afin de donner des propriétés des chemins de définitions de x connaissant $[\![x]\!]_{\tilde{s}}$ (pour une méthode x définie dans $\tilde{s}$). En premier lieu, on définit la *place* d'une méthode x dans l'espèce $\tilde{s}$.

Définition 95 (place d'une méthode) *Avec $x \in \mathcal{N}(norm(s))$, on appelle* place de x dans $norm(s)$ *et on note $\#_{norm(s)}x$ l'entier défini de la manière suivante :*

- $\#_{norm(s)}x = 1$ *si* $norm(s) = \begin{cases} \{\textbf{property}\ x :\ prop; l\} \\ \{\textbf{sig}\ x\ \textbf{in}\ \tau; l\} \\ \{\textbf{let}\ x\ \textbf{in}\ \tau = e; l\} \\ \{\textbf{theorem}\ x :\ prop = demo; l\} \end{cases}$
- $\#_{norm(s)}\textbf{rep} = 1$ *si* $norm(s) = \{\textbf{rep}[= \tau]; l\}$
- $\#_{norm(s)}x_i = i$ *si* $norm(s) = \{\textbf{let rec}\ x_1 = e_1 \ldots x_n = e_n; l\}$, *avec* $i \leq n$.
- $\#_{norm(s)}x = 1 + \#_l x$ *si* $norm(s) = \begin{cases} \{\textbf{property}\ y :\ prop; l\} \\ \{\textbf{sig}\ y\ \textbf{in}\ \tau; l\} \\ \{\textbf{let}\ y\ \textbf{in}\ \tau = e; l\} \\ \{\textbf{theorem}\ y :\ prop = demo; l\} \end{cases}$
  avec $y \neq x$.
- $\#_{norm(s)}x = 1 + \#_l x$ *si* $norm(s) = \{\textbf{rep}[= \tau]; l\}$ *et* $x \neq \textbf{rep}$
- $\#_{norm(s)}x = n + \#_l x$ *si* $norm(s) = \{\textbf{let rec}\ x_1 = e_1 \ldots x_n = e_n; l\}$ *et* $\forall i \leq n,\ x_i \neq x$.

Remarque 8. Dans le mixDrec $\ll norm(s)\gg$, tout chemin aboutissant à x a pour longueur $\#_{norm(s)}x$

Preuve. Immédiat par récurrence sur $\#_{norm(s)}x$ et par cas sur le premier champ de $norm(s)$. □

Lemme 52 (Chemin de définition et effacement)
Avec les notations précédentes, et y une méthode de $norm(s)$, on a $y \in [\![x]\!]_s \Rightarrow \forall p$ chemin de définition de x, $p_{\#_{norm(s)}y} = 1$

Preuve. Par induction sur $\#_{norm(s)}y$. Si y est la première méthode de $norm(s)$, on a par définition, $\mathbb{P}(norm(s)) = \mathbb{M}_y\mathbb{P}\left(\mathbb{E}_{\{y\}}(l)\right);\mathbb{P}(l)$ avec l la suite de la liste des méthodes de $norm(s)$. Avec $y \in \llbracket x \rrbracket_s$, le lemme sur l'effacement des def-dépendances nous donne $x \notin \mathcal{D}\left(\mathbb{E}_{\{y\}}(l)\right)$, et les nœuds correspondants du mixDrec seront donc tous abstraits : tout chemin de définition de x doit commencer par un 1.

Si $\#_{norm(s)}y > 1$. Soit z l'étiquette telle que $\#_{norm(s)}z = 1$. On a plusieurs cas possibles :

- $\mathbb{P}(norm(s)) = \mathbb{A}_z;\mathbb{P}(l) : \#_l y = \#_{norm(s)}y - 1$, et on conclut par récurrence.
- $\mathbb{P}(norm(s)) = \mathbb{M}_z\,\mathbb{P}\left(\mathbb{E}_{\{z\}}(l)\right);\mathbb{P}(l)$.
  - Si $z <_{t-dep} x$, x est abstrait dans $\mathbb{E}_{\{z\}}(l)$. Tous les chemins de définitions aboutissant à x commencent par 1,et on conclut en appliquant l'hypothèse de récurrence sur l.
  - Sinon, y ne def-dépend pas non plus de z (absurde par transitivité de $<_{t-dep}$). y est donc défini dans $\mathbb{P}\left(\mathbb{E}_{\{z\}}(l)\right)$, et on peut appliquer l'hypothèse de récurrence sur les deux fils de l'arbre, c'est à dire que le chemin de définition de x commence par 0 ou par 1.

□

Théoreme 17 *Avec s une espèce bien formée de* Foc *et $norm(s)$ sa forme normale, $\ll norm(s) \gg$ est un mixDrec équilibré.*

Preuve. On utilise pour cela la caractérisation des mixDrec équilibrés du lemme 50. Or on vient de montrer que tout chemin de définition contient des 1 au niveau des nœuds de def-dépendance. Il reste donc à montrer que le chemin π défini par $\pi_{\#_{norm(s)}y} = 1$ si $y <_{t-dep} x$ et $\pi_{\#_{norm(s)}y} = 0$ sinon est bien un chemin de définition de x. En effet, si c'est le cas, on aura $\pi = \rightsquigarrow^{\ll norm(s) \gg}(x)$, et on pourra appliquer le lemme 50.

On raisonne maintenant par induction sur la longueur de π. Si $norm(s)$ est composé uniquement du type support, abstrait ou concret, il y a au plus un chemin de définition (si $x = \mathbf{rep}$) sur $norm(s)$. et l'énoncé devient trivial.

Si π est de longueur $n+1$, on raisonne par cas :

- Si $\pi = 1; l$, ou si $\pi = 0; l$ et le premier champ de $norm(s)$ est abstrait, on conclut trivialement par récurrence.
- Sinon, avec $norm(s) = \phi; s_1$, on a par définition de π et $<_{t-dep}$, $\forall y \in \mathcal{N}(\phi), y \not<_{t-dep} x$. Il vient donc $\pi = \underbrace{0;..;0}_{k fois}; l$, où $k = \mathrm{Card}(\mathcal{N}(\phi))$.

  D'après le lemme de caractérisation de l'effacement, x reste donc défini dans $\mathbb{E}_{\mathcal{N}(\phi)}(s_1)$ et on peut appliquer l'hypothèse de récurrence : l est un chemin de définition de x dans $\mathbb{E}_{\mathcal{N}(phi)}(s_1)$, donc π est un chemin de définition de x dans $\ll norm(s) \gg$.

□

10.5 Générateur et environnement minimal

D'après la section précédente, une espèce $norm(s)$ en forme normale peut se traduire en un mixDrec équilibré, dont on peut donc extraire des contextes de définition minimaux pour chacune des méthodes définies de $norm(s)$. Toutefois, ce n'est pas encore suffisant pour faire la liaison avec les générateurs de méthodes de la traduction en Coq. En effet, le chemin de définition minimal d'une méthode x contient *a priori* "trop" d'abstractions, puisqu'on conserve tous les nœuds précédant x dans la forme normale. Ainsi, si on prend l'exemple de l'espèce s suivante (qui est déjà en forme normale) :

```
species s =
  rep = unit ;
  let id (x in int) in int = x
  let f (x) = self !id(x) ;
  let un = self !f(1) ;
end
```

Le générateur de méthode de `un` est de la forme $\lambda f.(f\,1)$, alors que le contexte minimal contient 3 nœuds abstraits avant le nœud manifeste correspondant à `un`. En d'autres termes, on voudrait pouvoir faire "remonter" les nœuds de `f` et `un` au-dessus de **rep** et `id`. En effet, le contexte minimal de `un` a la forme suivante :

$$\left\{\!\!\left\{ \begin{array}{l} \mathbf{rep} : Set; \\ \quad [rep]\,\{\!\{id : int \to int; [id]\,\{\!\{f : int \to int; [f]\,\{\!\{x : int = (f\ 1)\}\!\}\}\!\}\}\!\} \end{array} \right\}\!\!\right\}$$

Ce contexte tient implicitement compte de la decl-dépendance de `f` par rapport à `id` : dans la forme normale, `id` doit être introduit avant `f` et le contexte minimal suit cet ordre. Or si on s'intéresse uniquement à `un`, un mixDrec beaucoup plus petit suffit à le définir, puisqu'on n'a pas besoin de connaître la définition de `f`, mais seulement son type :

$$\{\!\{f : int \to int; \qquad [f]\,\{\!\{x : int = (f1)\}\!\}\}\!\}$$

Pour cela, on peut utiliser une relation d'équivalence sur les mixDrecs, où on s'autorise à permuter des champs sans effacer de définitions. Bien entendu, toutes les permutations ne sont pas possibles, à cause des dépendances. Intuitivement, cette relation doit identifier tous les mixDrecs ayant les mêmes étiquettes, et les mêmes définitions dans les nœuds manifestes correspondants. Plus précisément, on peut définir un certain nombre d'opérations élémentaires permettant de transformer un mixDrec en un mixDrec équivalent. Ces opérations sont dérivées des règles d'inférence de la relation de sous-signature $:\!\succ$, étendues aux mixDrecs.

Définition 96 (Équivalence entre mixDrecs)
Étant donné deux mixDrecs bien typés $\mathcal{M}_1$ et $\mathcal{M}_2$, on définit la relation $\mathcal{M}_1 \leftrightarrow \mathcal{M}_2$ par les règles données dans la figure 10.1.

Cette relation est en fait l'extension de la relation d'équivalence entre signature de Drecords dérivant de la notion de sous signature. Intuitivement, si $\mathcal{M}_1 \leftrightarrow \mathcal{M}_2$, $\mathcal{M}_1$ et $\mathcal{M}_2$ ont les mêmes champs, et leurs champs définis portent les mêmes définitions. Par contre l'ordre dans lequel les champs sont donnés peut varier. Donnons maintenant les principales propriétés de cette relation :

Lemme 53 *$\leftrightarrow$ est une relation d'équivalence.*

Preuve. La règle TRANS donne directement la transitivité.

La refléxivité se démontre par induction sur le nombre de champs d'un mixDrec $\mathcal{M}$: Dans le cas de base, on applique directement _E. Si $\mathcal{M}$ compte $n+1$ champs, on applique _CABST ou _CCONS suivant que le premier champ de $\mathcal{M}$ est abstrait ou manifeste, et on conclut par récurrence.

Enfin, la symétrie se montre par induction la dérivation de $\mathcal{M}_1 \leftrightarrow \mathcal{M}_2$, et par cas sur la dernière règle utilisée. Les règles _E, _CABST, _CCONS, SWAPAA et SWAPCC sont symétriques, tandis qu'il faut permuter les utilisations de SWAPAC et SWAPCA pour construire une dérivation de $\mathcal{M}_2 \leftrightarrow \mathcal{M}_1$ à partir de celle de $\mathcal{M}_1 \leftrightarrow \mathcal{M}_2$. □

Lemme 54 *Soit $\mathcal{M}_1$ et $\mathcal{M}_2$ deux mixDrecs tels que $\mathcal{M}_1 \leftrightarrow \mathcal{M}_2$. Alors*
- *$\mathcal{M}_1$ et $\mathcal{M}_2$ ont les même étiquettes*
- *Soit x une étiquette. S'il existe un nœud manifeste $\mathbb{M}_x$ dans $\mathcal{M}_1$, un tel nœud existe dans $\mathcal{M}_2$*

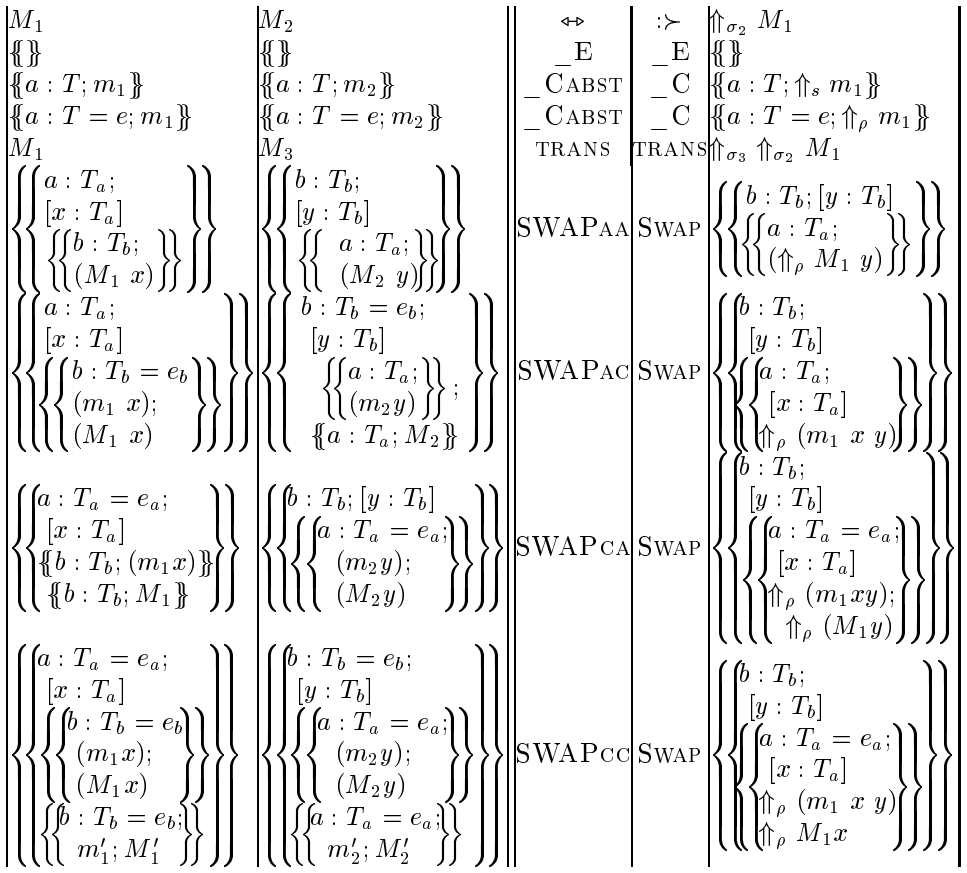
Preuve. Immédiat par induction sur la dérivation de l'équivalence. □

Par ailleurs, cette relation peut s'interpréter à l'aide de $:\succ$ et $\succcurlyeq$. De ce point de vue, deux mixDrecs sont équivalents si chacun est une vue plus définie de l'autre une fois qu'on a fait les permutations nécessaires. En quelque sorte, il s'agit de l'extension aux mixDrec de l'équivalence induite par $:\succ$ sur les signatures de Drecord. Plus précisément, on a la propriété suivante :

Lemme 55 *Soient M_1 et M_2 deux mixDrecs, de signature respectivement σ_1 et σ_2. Si $M_1 \leftrightarrow M_2$, alors*

1. *$\sigma_1 :\succ \sigma_2$ et $\sigma_2 :\succ \sigma_1$*
2. *$M_2 \succcurlyeq \Uparrow_{\sigma_2} M_1$ et $M_1 \succcurlyeq \Uparrow_{\sigma_1} M_2$*

Preuve. Du fait de la symétrie de $\leftrightarrow$, il suffit de montrer $\sigma_1 :\succ \sigma_2$ et $M_2 \succcurlyeq \Uparrow_{\sigma_2} M_1$. On le fait par induction sur la dérivation de $M_1 \leftrightarrow M_2$. À chaque règle de $\leftrightarrow$ correspond une règle de même nom de $:\succ$. Pour la seconde propriété, il faut commencer par examiner le résultat de $\Uparrow$ suivant la règle utilisée. En appelant ρ la liste des champs de σ_2 qui n'interviennent pas dans la règle, les différents cas sont présentés dans le tableau 10.1. Comme on le voit sur le tableau en comparant dans chaque ligne M_2 et $\Uparrow_{\sigma_2} M_1$, cette



TAB. 10.1 – Équivalence entre mixDrecs

transformation donne bien une vue moins définie que M_2, en particulier dans les cas SWAPAC et SWAPCC. □

Une fois établies ces propriétés, il convient de s'intéresser au sort des chemins dans cette relation d'équivalence : on va définir une fonction qui à un chemin de $\mathcal{M}_1$ associe un chemin de $\mathcal{M}_2$ aboutissant à la même étiquette, et tel que tout chemin de définition se transforme en un chemin de définition. Cela va permettre de montrer un autre point important de cette relation, la préservation de la notion d'équilibre. De ce fait, si on part d'un mixDrec équilibré $\mathcal{M}$ définissant un champ x, on va pouvoir étudier les contextes minimaux de définition de x dans tous les mixDrecs m équivalents à $\mathcal{M}$. Dans le cas où cette équivalence est obtenue en permutant x avec des champs précédents, le contexte minimal dans m est plus petit que le contexte minimal dans $\mathcal{M}$. Dès lors, on pourra s'intéresser à trouver les contexte minimaux de définition de x les plus petits possible dans les mixDrecs équivalents à $\mathcal{M}$.

Définition 97 (Transformation des chemins) *Soient $\mathcal{M}_1$ et $\mathcal{M}_2$ deux mixDrecs équivalents, x une étiquette de $\mathcal{M}_1$, et p un chemin de $\mathcal{M}_1$ abou-*

tissant à x. On définit le chemin équivalent *à p dans $\mathcal{M}_2$, qu'on note* $\overset{M_1}{\underset{M_2}{\downarrow}}(p)$ *de la manière suivante :*

Soit n_1 (respectivement n_2) la place de x dans $\mathcal{M}_1$ (respectivement $\mathcal{M}_2$) Soit y le i-ème champ de $\mathcal{M}_2$, ($i \leq n$) et j la place de y dans $\mathcal{M}_1$. Alors

- *Si $j \leq n_1$ alors* $\left(\overset{\mathcal{M}_1}{\underset{\mathcal{M}_2}{\downarrow}}(p)\right)_i = p_j$
- *Si $j > n_1$ alors* $\left(\overset{\mathcal{M}_1}{\underset{\mathcal{M}_2}{\downarrow}}(p)\right)_i = 0$

Lemme 56 (Transformation des chemins de définition) *Soient $\mathcal{M}_1$ et $\mathcal{M}_2$ deux mixDrecs équivalents, et p un chemin de définition de x dans M_1. Alors* $\overset{\mathcal{M}_1}{\underset{\mathcal{M}_2}{\downarrow}}(p)$ *est un chemin de définition de x dans $\mathcal{M}_2$.*

Preuve. Par induction sur la dérivation de l'équivalence entre $\mathcal{M}_1$ et $\mathcal{M}_2$. On conserve pour la preuve les notations des règles ci-dessus.

_E : Trivial (il n'y a pas de champ défini).

_Cabst et _Cconc : Immédiat par induction : les deux chemins ont le même premier élément

trans : Soit $\mathcal{M}_3$ le mixDrec tel que $\mathcal{M}_1 \leftrightarrow \mathcal{M}_2 \leftrightarrow \mathcal{M}_3$. Par hypothèse, $p' = \overset{M_3}{\underset{M_1}{\downarrow}}\left(\overset{M_1}{\underset{M_3}{\downarrow}}(p)\right)$ est un chemin de définition de x dans $\mathcal{M}_2$. On va travailler par récurrence sur la taille de p' pour montrer que $p'' = \overset{\mathcal{M}_1}{\underset{\mathcal{M}_2}{\downarrow}}(p)$ est lui aussi un chemin de définition de x : si x est le premier champ de $\mathcal{M}_2$, le cas est trivial. Sinon, avec y le premier champ de $\mathcal{M}_2$:

- y avant x dans $\mathcal{M}_1$ et $\mathcal{M}_3$: p' et p'' ont le même premier élément et on conclut par récurrence.
- y est avant x dans $\mathcal{M}_1$ et pas dans $\mathcal{M}_3$: $p' = 0; l'$ $p'' = u; l''$. Si $u = 0$, on conclut comme ci-dessus. Sinon, d'après le lemme 46 $1; l'$ est aussi un chemin de définition, et on peut conclure par récurrence.
- y n'est pas dans $\mathcal{M}_1$: p' et p'' commencent par 0 et on conclut par récurrence.

SWAPXX : Si x est un des deux champs de tête de $\mathcal{M}_1$, dans les cas où ils sont définis, le cas est trivial. Sinon, on prend $p = p_a; p_b; l$. $\overset{\mathcal{M}_1}{\downarrow}_{\mathcal{M}_2}(p) = p_b; p_a; l'$. Avec m_1 (respectivement m_2) le sous-mixDrec de $\mathcal{M}_1$ correspondant au chemin $p_a; p_b$ (respectivement celui de $\mathcal{M}_2$ correspondant à $p_b; p_a$), il vient d'après les prémisses des règles $m_1 \leftrightarrow m_2$, et $l' = \overset{m_1}{\downarrow}_{m_2}(l)$: les étiquettes de m_2 sont soit dans m_1 soit après x dans $\mathcal{M}_1$. On peut donc conclure par récurrence. □

La transformation des chemins aboutissant à x de M_1 vers ceux de M_2 n'est ni injective ni surjective, car certains champs y qui étaient avant x dans M_1 peuvent se retrouver après x dans M_2 et vice-versa. Toutefois, pour chaque chemin p_2 de M_2, on peut trouver un chemin p_1 de M_1 tel que $\overset{M_1}{\downarrow}_{M_2}(p_1)$ soit "suffisamment proche" de p_2, comme l'exprime le lemme ci-dessous. Cette propriété sera suffisante pour la suite.

Lemme 57 (Recouvrement) *Soient M_1 et M_2 deux mixDrecs équivalents, et x un champ défini dans les deux mixDrecs. Alors, $\forall p_2$ chemin aboutissant à x dans M_2, $\exists p_1$ chemin aboutissant à x dans M_1, tel que $\forall i$, $\overset{M_1}{\downarrow}_{M_2}(p_1)_i = 1 \Rightarrow p_{2\,i} = 1$*

Preuve. On va montrer que $p = \overset{M_2}{\downarrow}_{M_1}(p_2)$ convient. Examinons pour cela le champ y placé en i-ème position dans M_2. Il y a deux possibilités :

- Soit y est avant x dans M_1 et $\overset{M_1}{\downarrow}_{M_2}(p)_i = p(y) = p_2(y)$, donc dans ce cas, l'implication est vérifiée
- Soit y est après x dans M_1, et $\overset{M_1}{\downarrow}_{M_2}(p)_i = 0 \neq 1$: l'implication est vérifiée.

□

Lemme 58 (Équivalence et équilibre)
Soit M_1 un mixDrec équilibré. Alors, $\forall M_2$, tel que *$M_1 \leftrightarrow M_2$, M_2 est équilibré*

Preuve. Soit x un champ défini dans M_2 (donc dans M_1). On pose $p_x = \underset{M_2}{\overset{M_1}{\downarrow}}(\rightsquigarrow^{M_1}(x))$. D'après le lemme 56, p_x est un chemin de définition de x dans M_2. On va montrer que si la i-ème composante de p_x est un 1, alors tout chemin de définition c de x dans M_2 a un 1 pour i-ème composante. Le lemme 50 et la remarque 7 permettront alors de conclure.

Soit q_1 un autre chemin de définition de x dans M_1, et $q_2 = \underset{M_2}{\overset{M_1}{\downarrow}}(q_1)$. Montrons que $p_{xi} = 1 \Rightarrow q_{2i} = 1$. Soit y l'étiquette du i-ème champ de M_2.

- Si y est avant x dans M_1, alors $p_{xi} = \rightsquigarrow^{M_1}(x)(y)$ et $q_{2i} = q_1(y)$. Or, d'après le lemme 49, M_1 étant équilibré, on a bien $\rightsquigarrow^{M_1}(x)(y) = 1 \Rightarrow q_1(y) = 1$
- sinon, $p_{xi} = q_{2i} = 0$, et l'implication reste valide.

Or d'après le lemme de recouvrement, pour tout chemin c de définition de x dans M_2, il existe c' chemin de définition de x dans M_1 vérifiant $\underset{M_2}{\overset{M_1}{\downarrow}}(c')_i = 1 \Rightarrow c_i = 1$. Comme on vient de montrer que pour un tel c', $p_{xi} = 1 \Rightarrow \underset{M_2}{\overset{M_1}{\downarrow}}(c')_i = 1$, on a bien la conclusion attendue : $p_{xi} = 1 \Rightarrow c_i = 1$. □

10.5.1 MixDrec générateur

Nous disposons maintenant de tous les outils permettant de relier l'environnement minimal de typage d'une méthode définie x d'une espèce s (def 59) au contexte minimal de définition d'un champ d'étiquette x du mixDrec correspondant. En fait, il s'agit de construire un mixDrec m à partir de l'environnement minimal $\Sigma \Cap x$ et du corps de x, et de montrer qu'il existe un mixDrec M, équivalent au contexte minimal de définition de x dans $\ll s \gg$, tel que m soit le contexte minimal de définition de x dans M. De plus, la longueur des chemins aboutissant à x dans M est minimale : tous les mixDrecs équivalents à $\widetilde{\mathbf{\Gamma}_{\ll s \gg}} \vdash \mathbf{x}$ ont des chemins aboutissant à x au moins aussi longs. C'est en ce sens qu'on peut réellement parler d'environnement *minimal* de définition, qu'on peut donc aussi bien extraire d'un mixDrec que d'une espèce en forme normale.

Définition 98 (mixDrec générateur) *Soit $norm(s)$ une espèce en forme normale, et x une méthode définie de $norm(s)$. On définit* le mixDrec générateur de x, *noté $\gamma_{norm(s)}(x)$ comme le mixDrec correspndant à l'environnement minimal de typage de x :*

$$\gamma_{norm(s)}(x) = \ll norm(s) \Cap x; x : \mathcal{T}_s(x) = \mathcal{B}_s(x) \gg$$

Théoreme 18 (mixDrec générateur et contexte minimal)
Soit $norm(s)$ une espèce en forme normale, et x une méthode définie de $norm(s)$. Il existe un mixDrec $\mathcal{M}$ vérifiant les conditions suivantes :

1. $\mathcal{M} \leftrightarrow \widetilde{\mathbf{\Gamma}_{\ll norm(\mathbf{s})\gg}} \vdash \mathbf{x}$
2. $\widetilde{\mathbf{\Gamma}_{\mathcal{M}}} \vdash \mathbf{x} = \gamma_{norm(s)}(x)$
3. $\forall m$, tel que $m \leftrightarrow \widetilde{\mathbf{\Gamma}_{\ll norm(\mathbf{s})\gg}} \vdash \mathbf{x}, \| x \|_{\mathcal{M}} \leq \| x \|_m$

Preuve. On va commencer par construire un mixDrec $\mathcal{M}$ équivalent à $\widetilde{\mathbf{\Gamma}_{\ll norm(\mathbf{s})\gg}} \vdash \mathbf{x}$, par applications successives des règles de $\leftrightarrow$. Nous montrerons ensuite que le mixDrec obtenu répond aux deux autres conditions de l'énoncé.

La construction de $\mathcal{M}$ se fait de la manière suivante. On part de $m = \widetilde{\mathbf{\Gamma}_{\ll norm(\mathbf{s})\gg}} \vdash \mathbf{x}$. On sélectionne alors la paire d'étiquettes consécutives y_1, y_2 vérifiant les conditions suivantes :

- $y_1 \notin (| x | \cup \{x\})$
- $\| y_1 \|_m < \| x \|_m$
- $\| y_1 \|_m$ est maximum

et on échange les deux étiquettes à l'aide des règles ConsXX et SWAPXX. Cet échange est toujours possible, car par définition de l'univers visible d'une méthode, ni le type, ni le corps – s'il existe – de y_2 ne dépendent de y_1 : m est un mixDrec bien formé. Si aucun échange n'est possible, on pose $\mathcal{M} = m$. La terminaison de cette suite d'échanges peut-être montrée en utilisant l'ordre lexicographique sur la paire $(\| x \|_m, \| x \|_m - \| y_1 \|_m)$. En effet, en appelant m_1 et m_2 les deux mixDrecs avant et après une étape de transformation, deux cas de figure se présentent :

- $y_2 = x$, et $\| x \|_{m_2} = \| x \|_{m_1} - 1$
- $y_2 \neq x$. Alors $\| x \|_{m_2} = \| x \|_{m_1}$, mais on a $\| y_1 \|_{m_2} = \| y_1 \|_{m_1} + 1$, et par hypothèse y_1 reste l'étiquette absente de l'univers visible de x telle que $\| y_1 \|_m < \| x \|_m$ soit maximum. Le premier élément de la paire reste constant tandis que le second décroît.

Une fois qu'on a obtenu $\mathcal{M}$, il reste à montrer qu'il vérifie bien les conditions demandées. En premier lieu, toute étiquette de $\mathcal{M}$ située avant x est dans l'univers visible de x. Sinon, un échange serait possible en respectant les conditions données ci-dessus.

On peut ensuite remarquer que deux étiquettes appartenant à $\widetilde{| x |}$ ne sont jamais échangées, et que leur ordre relatif reste donc celui de $\widetilde{\mathbf{\Gamma}_{\ll norm(\mathbf{s})\gg}} \vdash \mathbf{x}$, donc de $norm(s)$. De même, les seuls nœuds manifestes de $\widetilde{\mathbf{\Gamma}_{\ll norm(\mathbf{s})\gg}} \vdash \mathbf{x}$ sont ceux dont x def-dépend, tout comme dans $norm(s) \Cap x$. Les deux mixDrecs sont donc égaux.

Enfin, montrer que la profondeur de x est minimale dans $\mathcal{M}$ revient à prouver que tous les champs d'étiquette y situés avant x dans $\mathcal{M}$ ne peuvent être échangés avec x. Comme ce sont tous les champs de l'univers visible de

x, on peut raisonner par induction sur la preuve que $y \in\mid x \mid$. Il y a dès lors 4 cas possibles :

$y <_s^{def} x$: y doit être présent et défini dans tout environnement de définition de x.

$y \in \wr x \int_s$: y doit être présent dans tout environnement de définition de x

$y \in \wr z \int_s \wedge z <_s^{def} x$: z doit être présent et défini pour qu'on puisse définir x, et on ne peut permuter y et z, donc *a fortiori* y et x.

$y \in \wr \mathcal{T}_s(z) \int_\wedge z \in\mid x \mid$: de même, on ne peut permuter z et x, ni y et z. y ne peut donc se retrouver après x. □

Remarque 9. Dans le cas général, $\mathcal{M}$ n'est pas l'unique mixDrec équivalent à $\mathbf{\Gamma}_{\ll norm(\mathtt{s})\gg} \vdash \mathbf{x}$ vérifiant cette condition de minimalité : des permutations entre les champs situés avant x sont *a priori* possibles. Par exemple, si on prend l'espèce

```
species s =
  rep = int; let un = 1;
  let plus(x,y)=x+y; let succ(x)=!plus(!un,x);
end
```

on peut obtenir un mixDrec équivalent au mixDrec générateur en permutant `un` et `plus`.

10.5.2 Héritage, redéfinition et effacement

il reste à voir comment le mixDrec générateur d'une méthode x se comporte vis-à-vis de l'héritage. Pour cela, considérons une espèce s et une méthode x définie dans $norm(s)$, telle que $x \upharpoonleft s = s' \neq s$. Par hypothèse, toutes les méthodes dont x def-dépend ont conservé la définition qu'elles avaient dans s' (sinon x ne serait pas défini, ou aurait une définition différente). L'univers visible de x dans s est donc le même que celui de s', et il vient :

Théoreme 19 (Héritage et mixDrec générateur) *Soient s une espèce bien formée et bien typée, et x une méthode définie de s, telle que $x \upharpoonleft s = s'$. On a alors :*

$$\gamma_s(x) \leftrightarrow \gamma_{s'}(x)$$

Preuve. Si $s = s'$, le cas est trivial. Sinon, avec les remarques précédentes, les deux environnements minimaux de typages, $x \Cap s'$ et $x \Cap s$ sont composés des mêmes méthodes, déclarées comme définies, et les méthodes définies ont les mêmes corps. Seul l'ordre des méthodes peut varier. Les deux mixDrecs obtenus à partir de ces environnements sont donc équivalents. □

Ainsi, la notion de mixDrec générateur est bien un invariant qui se conserve au cours de l'héritage, tant qu'on ne change pas les méthodes dont x def-dépend.

$$\frac{}{\{\!\{\}\!\} \leftrightarrow \{\!\{\}\!\}}\ \_\text{E} \qquad \frac{(x:T)((m_1 x) \leftrightarrow (m_2 x))}{\{\!\{a:T; m_1\}\!\} \leftrightarrow \{\!\{a:T; m_2\}\!\}}\ \_\text{CABST}$$

$$\frac{(x:T)((m_1 x) \leftrightarrow (m_2 x)) \qquad M_1 \leftrightarrow M_2}{\{\!\{a:T=e; m_1; M_1\}\!\} \leftrightarrow \{\!\{a:T=e; m_2; M_2\}\!\}}\ \_\text{CCONC} \qquad \frac{M_1 \leftrightarrow M_2 \qquad M_2 \leftrightarrow M_3}{M_1 \leftrightarrow M_3}\ \textit{trans}$$

$$\frac{(x:T_a)(y:T_b)((m_1\ x\ y) \leftrightarrow (m_2\ y\ x))}{\{\!\{a:T_a; [x:T_a]\,\{\!\{b:T_b; (m_1\ x)\}\!\}\}\!\} \leftrightarrow \{\!\{b:T_b; [y:T_b]\,\{\!\{a:T_a; (m_2\ y)\}\!\}\}\!\}}\ \text{SWAPAA}$$

$$\frac{(x:T_a)(y:T_b)((m_1\ x\ y) \leftrightarrow (m_2\ y\ x)) \qquad (x:T_a)((M_1 x) \leftrightarrow (M_2\ x))}{\begin{array}{c}\{\!\{a:T_a; [x:T_a]\,\{\!\{b:T_b=e_b; (m_1 x); (M_1 x)\}\!\}\}\!\} \leftrightarrow \\ \{\!\{b:T_b=e_b; [y:T_b]\,\{\!\{a:T_a; (m_2 y)\}\!\}; \{\!\{a:T_a; M_2\}\!\}\}\!\}\end{array}}\ \text{SWAPAC}$$

$$\frac{(x:T_a)(y:T_b)((m_1 xy) \leftrightarrow (m_2 yx)) \qquad (y:T_b)((M_1 y) \leftrightarrow (M_2 y))}{\begin{array}{c}\{\!\{a:T_a=e_a; [x:T_a]\,\{\!\{b:T_b; (m_1 x)\}\!\}; \{\!\{b:T_b; M_1\}\!\}\}\!\} \leftrightarrow \\ \{\!\{b:T_b; [y:T_b]\,\{\!\{a:T_a=e_a; (m_2 y); (M_2 y)\}\!\}\}\!\}\end{array}}\ \text{SWAPCA}$$

$$\frac{\begin{array}{c}(x:T_a)(y:T_b)((m_1 xy) \leftrightarrow (m_2 yx)) \\ (x:T_a)((M_1 x) \leftrightarrow (m_2' x)) \qquad (y:T_b)((m_1' y) \leftrightarrow (M_2 y)) \qquad M_1' \leftrightarrow M_2'\end{array}}{\begin{array}{c}\{\!\{a:T_a=e_a; [x:T_a]\,\{\!\{b:T_b=e_b; (m_1 x); (M_1 x)\}\!\}; \{\!\{b:T_b=e_b; m_1'; M_1'\}\!\}\}\!\} \\ \leftrightarrow \\ \{\!\{b:T_b=e_b; [y:T_b]\,\{\!\{a:T_a=e_a; (m_2 y); (M_2 y)\}\!\}; \{\!\{a:T_a=e_a; m_2'; M_2'\}\!\}\}\!\}\end{array}}\ \text{SWAPCC}$$

FIG. 10.1 – Règles d' inférence pour $\leftrightarrow$

Chapitre 11

Perspectives et Conclusion

11.1 Perspectives

Les principales constructions de FOC telles qu'elles ont été décrites dans les chapitres précédents semblent désormais bien implantées. Concernant la partie programmation proprement dite, le compilateur FOC a pu être testé sur une librairie regroupant une centaine d'espèces et environ 6000 lignes de code. En revanche, il reste à implanter un vrai langage de preuve (différent des scripts COQ, qui sont trop dépendants de la traduction concrète en COQ). De plus, toutes les obligations de preuves ne sont pas générées.

11.1.1 Obligations de preuves et démonstrations

Congruences En premier lieu, il faut établir pour chaque méthode d'une espèce qu'elle est bien compatible avec l'égalité, ainsi qu'on l'a décrit dans la section 3.9.2. Toutefois, en pratique, on souhaite automatiser autant que possible les preuves de tels énoncés. En particulier, quand l'égalité utilisée est l'égalité structurelle sur le type support, le compilateur doit être capable de synthétiser la preuve en même temps que l'égalité. De même, dans notre exemple des produits cartésiens, où l'égalité se fait en comparant chacune des composantes, et en utilisant l'égalité des paramètres a et b, on souhaiterait que les preuves soient inférées automatiquement.

Preuves de terminaison Si les preuves de terminaison des méthodes récursives simples peuvent suivre le schéma décrit dans la section 6.5, le cas de méthodes mutuellement récursives est légèrement plus compliqué en pratique. Très grossièrement, il est a priori possible de se ramener au cas d'une récursion simple, en utilisant un type inductif intermédiaire, muni d'autant de constructeurs qu'il y a de fonctions dans la récursion. Par exemple, avec la définition suivante :

let rec `f_1`(`x` **in** t_1) **in** t'`_1` $= e_1$ **and** `f_2`(`y` **in** t_2) **in** t'`_2` $= e_2$

donnerait lieu aux définitions suivantes :

```
Inductive mon_type : Set :=
  F_1 : t_1 → mon_type
| F_2 : t_2 → mon_type.
(* lemmes de terminaison. *)
Definition rec_aux :=[xy :mon_type]
[f : (xy' :mon_type)(xy'< xy) → T]
<T>Cases xy of
| (F_1 x) => e_1
| (F_2 y) => e_2
end
```

`T` étant le type (`xy` :`A`)**Cases** `xy` **of** (`F_1 x`) => t'`_1` | (`F_2 y`) => t'`_2` **end**. Dans e_1 et e_2, les appels récursifs à `f_1` (respectivement `f_2`) sont remplacés par des appels à `f` avec un argument de la forme (`F_1 x`) (respectivement (`F_2 y`)). Cette méthode permet de se ramener au cas d'une seule fonction, mais elle semble particulièrement lourde à mettre en place, et la définition d'un ordre bien fondé sur `mon_type` et les preuves des lemmes intermédiaires risquent d'être particulièrement difficiles à mener.

Preuves À l'heure actuelle, une "preuve" Foc est un script de preuve Coq, qui sera donné tel quel à ce dernier dans l'environnement correspondant à la traduction Coq du code Foc qui l'entoure. Cette situation permet de réaliser effectivement des preuves, mais est très insuffisante :

- Réaliser une preuve nécessite de connaître à la fois Foc, Coq, et le mécanisme de traduction.
- Cela rend Foc très dépendant de Coq, alors que les autres constructions du langages sont *a priori* représentables dans d'autres systèmes.
- La validité des scripts peut être brisée par les évolutions de Foc lui-même, mais aussi par celles de Coq. Cela rend difficile la maintenance des preuves.
- Les énoncés Foc n'ont pas toujours une forme "naturelle" en Coq, ce qui rend les scripts d'autant plus compliqués, y compris pour des preuves simples.

Il apparaît donc nécessaire à terme de mettre en place un langage de preuve qui soit propre à Foc, et de réaliser un prouveur, plus automatisé et plus spécifique que Coq. Dans ce contexte, Coq servirait exclusivement à vérifier la validité des termes de preuves générés par ce prouveur, et de l'environnement Foc sous-jacent. Une autre possibilité intéressante dans ce domaine serait d'interfacer Foc avec des prouveurs déjà existants.

11.1.2 L'égalité : décidable ou non ?

La librairie Foc actuelle repose sur une espèce des setoïdes où l'égalité est décidable : elle est déclarée comme une fonction à valeur dans `bool`, et doit donc renvoyer `true` ou `false` pour chaque couple d'élément du type support. Or toutes les structures algébriques ne peuvent être munies d'une telle égalité : l'égalité sur $\mathbb{R}$ est indécidable, de même que l'égalité entre fonctions, ... On pourrait imaginer construire une nouvelle hiérarchie de structures basée sur une espèce ou l'égalité serait un prédicat, à valeurs dans **Prop**. `self !equal(x,y)` ne serait plus alors le résultat d'un calcul, mais un énoncé à prouver.

Cependant, on ne souhaite pas dupliquer complètement la hiérarchie existante (ce serait en contradiction avec les principes de réutilisation du code qui ont conduit à la construction de ladite hiérarchie). En effet, toutes les espèces où **self** `!equal` n'apparaît pas dans le corps d'une fonction peuvent très bien s'accommoder d'une égalité non décidable. Deux solutions sont possibles :

- utiliser deux méthodes différentes : `equal` serait déclarée comme un prédicat au début de la hiérarchie. Dans les espèces ayant besoin de faire des calculs utilisant l'égalité, on déclarerait une nouvelle méthode `equal_dec`, renvoyant un booléen, et on définirait `equal` comme étant égale à `equal_dec` –modulo la coercion implicite en Foc entre `bool` et `Prop`
- n'utiliser qu'une seule méthode dont le type de retour serait raffiné de `Prop` en `bool` lorsque c'est nécessaire.

Le premier cas est très simple à implanter (la version actuelle du compilateur suffit), mais n'est pas satisfaisant du point de vue conceptuel : il n'y a qu'une seule égalité, qu'elle soit décidable ou non. Implanter la seconde solution nécessite de modifier légèrement le compilateur pour qu'il accepte un début de sous-typage sur les types de méthode, ce qui nécessite malgré tout quelques précautions.

11.1.3 Invariants de représentation

Comme on l'a déjà dit, dans de nombreux cas les éléments manipulés par une espèce constituent un sous-ensemble du type support concret. Par exemple, les entiers modulo n n'utilisent que les éléments de `int` compris entre `0` et `n−1`. De même, les polynômes en représentation creuse n'ont jamais de coefficient nul : le premier élément de chaque paire de la liste est non nul. L'abstraction du type support dans les collections garantit qu'on n'appelle pas les méthodes d'une collection donnée sur des éléments ne vérifiant pas ces invariants, mais n'assure pas du tout que ces mêmes méthodes respectent ces invariants.

Pour cela, il conviendrait d'ajouter un nouveau type de méthode, per-

mettant d'exprimer précisément ces invariants. Il s'agirait d'un prédicat sur **self**, devant être vérifié par tout élément de **self** (mais pas nécessairement par tout élément de la représentation concrète). Ce prédicat pourra être indécidable dans la mesure où il n'interviendra jamais dans les calculs. Par contre, il engendrera de nouvelles obligations de preuves : toute fonction renvoyant un élément de **self** devra être accompagné de la preuve que cet élément vérifie bien le prédicat.

11.1.4 Collections locales

Les collections locales ont été introduites afin de pouvoir exprimer en FOC certains algorithmes de calcul sur les polynômes. Si leur utilisation dans la partie "programmation" de FOC semble satisfaisante, un certain travail reste à faire pour les introduire dans la composante "preuve" du langage. En particulier, la définition de méthodes renvoyant une collection, construction essentielle de certaines espèces du haut de la hiérarchie FOC, pose un certain nombre de problèmes. Le principal d'entre eux concerne le calcul de dépendance (présenté dans la section 8.4.2), ou plutôt la récursion mutuelle de ces méthodes avec d'autres méthodes de **self**. Outre le fait que les preuves de terminaison de méthodes mutuellement récursives ne sont pas implantées, ce cas particulier lie récursivement une valeur constante (la méthode renvoyant une instance de la collection) et des fonctions. Les preuves de terminaison risquent d'imposer qu'on "ouvre" la collection liée à la méthode pour pouvoir s'intéresser aux récursions entre ses propres méthodes et celles de **self**.

11.1.5 Fonctions Partielles

Enfin, un dernier point à aborder en FOC concerne l'implantation des fonctions partielles. Pour cela, on pourra s'appuyer sur les différentes approches proposées dans [DV98]. On peut bien entendu voir les fonctions partielles à travers un type `option`, mais ce codage est relativement lourd à manipuler, en particulier quand on veut faire des preuves sur ces fonctions. Une autre approche pourrait consister à utiliser des prédicats, comme dans le cas des invariants de représentation, et à imposer de faire la preuve, chaque fois que l'on veut utiliser une telle fonction, qu'on lui donne bien un argument "correct" (pour lequel elle va pouvoir renvoyer un résultat).

En pratique, il est toutefois possible qu'on soit conduit à faire coexister ces deux approches. En effet, il n'est pas toujours facile de formaliser l'ensemble sur lequel une telle fonction est définie, et il est parfois plus aisé de faire les calculs et de regarder si on obtient une valeur ou non.

11.1.6 Liens avec d'autres langages

Enfin, il pourrait être intéressant de dépasser le cadre d'OCAML et de COQ pour étudier une implantation de FOC au dessus d'autres langages. En

effet, les espèces et les collections semblent assez largement indépendantes du langage des expressions utilisé pour les définitions de méthodes. Plus précisément, la donnée de l'algorithme de typage d'une expression ainsi que celle de la fonction permettant de retrouver def- et decl- dépendances d'une expression permettent de définir l'algorithme de mise en forme normale indépendamment de la forme exacte de l'expression en question.

En suivant les idées de [Ler00], on pourrait donc essayer de voir dans quelle mesure il est possible de modulariser l'approche présentée dans cette thèse, afin de pouvoir bâtir des espèces et des collections au-dessus d'un langage de base ayant un certain nombre de propriétés bien identifiées. Cette approche permettrait peut-être d'éloigner FOC de OCAML, en utilisant des langages de base différents. D'autre part, elle constituerait une bonne base pour l'extension de ce même langage par de nouvelles constructions (en particulier des traits impératifs).

11.2 Conclusion

Pour résumer les travaux présentés dans cette thèse, on peut revenir un instant sur le compilateur FOC. Ce dernier reconnaît le langage décrit dans le chapitre 2, ainsi que les extensions permettant de définir des méthodes ou des variables locales renvoyant des collections. Les analyses menées sur des programmes FOC pour déterminer leur correction peuvent être divisées en deux grandes catégories : le typage et l'analyse de dépendances. Cette dernière remplit en fait deux objectifs. D'une part il s'agit de détecter d'éventuels cycles entre différentes méthodes d'une espèce. D'autre part, il faut s'occuper de la résolution de l'héritage (simple comme multiple) au cours de cette analyse, pour tenir compte d'éventuelles redéfinitions. Ainsi qu'on l'a vu, la détection de cycle se fait en effet sur des espèces en forme normale. En ce qui concerne le système de type de FOC, on peut remarquer que si les méthodes "calculatoires" ne nécessitent qu'un système de type polymorphe à la ML, les propriétés, mais surtout les espèces paramétrées entraînent la présence de type dépendants.

La première sortie possible du compilateur est un fichier OCAML représentant la hiérarchie d'espèce présente en FOC. Deux représentations sont possibles, comme on l'a vu. La première utilise la correspondance naturelle qui existe entre les espèces et les classes de OCAML, en particulier en ce qui concerne les mécanismes d'héritage. De plus, le système de modules permet de garantir –par l'intermédiaire du typeur de OCAML– que les abstractions des types supports des collections sont bien réalisées. L'autre méthode de compilation est dérivée de la représentation en COQ de la hiérarchie FOC et repose en particulier sur la simulation de la liaison tardive par l'utilisation de générateurs de méthodes. En effet, la compilation vers COQ ne peut se baser sur des traits objets, qui n'existent pas dans ce dernier langage. Les

générateurs de méthodes sont un moyen d'obtenir les propriétés qui nous intéressent dans l'héritage, c'est à dire la sélection des définitions de méthodes les plus récentes, en évitant au maximum la duplication de code. Il pourrait d'ailleurs être intéressant de comparer les résultats de cette seconde traduction en OCAML avec une extraction du code COQ produit par FOC. Globalement, le compilateur représente environ 17000 lignes de OCAML, qu'on peut répartir de la manière suivante :

parseur	3000
analyses statiques	4500
traduction OCAML 1	2500
traduction OCAML 2	1500
traduction COQ	2000
prototype langage de preuve	2000
front-end et utilitaires	1500

Enfin, les correspondances entre les analyses faites sur les espèces et les relations entre mixDrecs permettent de relier le compilateur FOC à la formalisation des différentes constructions utilisées faite par Sylvain Boulmé. C'est ce lien qui garantit la cohérence du langage FOC dans sa version actuelle avec les spécifications initiales. Cette cohérence est importante dans l'optique de la certification des programmes FOC : le système dans lequel on compte faire la preuve se doit d'être lui-même bien formé.

Par ailleurs, les espèces de base de la hiérarchie pourraient d'un certain point de vue être considérées comme un "cahier des charges", exposant les différentes opérations à réaliser, ainsi que les propriétés qu'elles doivent respecter. À l'autre extrémité de la chaîne, les collections apportent des solutions effectives à ce cahier des charges. Entre les deux, les différentes espèces de la hiérarchie permettent de préciser les spécifications de départ, ou de raffiner certaines définitions génériques pour mieux les adapter à certains cas particuliers (précisés par de nouvelles propriétés). Enfin, l'utilisation des espèces paramétrées offre une certaine modularité : on implante un ensemble de fonctionnalités modulo la donnée d'un certain nombre d'autres éléments, vérifiant des propriétés précisées par l'interface du paramètre.

Pour conclure, si l'implantation concrète du compilateur a été testée par l'implantation d'une librairie de calcul formel dans le langage FOC tel qu'il a été présenté ci-dessus, la compilation vers OCAML et COQ est relativement indépendante de ce domaine particulier. En effet, FOC semble être à même de décrire les connaissances d'un domaine d'application particulier de manière structurée. Une telle construction hiérarchique paraît susceptible de refléter le passage progressif de spécifications abstraites jusqu'à des programmes permettant de manipuler les données du domaine en question. Dans ce contexte, les différentes analyses de FOC permettent de s'assurer que chacune des étapes menant au programme final respecte les spécifications de départ.

Annexe A

Performances de la librairie

Cette annexe compare les résultats, en terme de temps de calcul et d'allocation mémoire, des différents mécanismes de traduction vers OCaml décrits dans les chapitres 4 et 8. Pour ce faire, nous avons étudié la factorisation de polynômes sur un corps fini [Man01], en l'occurence $^{\mathbb{Z}}/_{5\mathbb{Z}}$. Plus précisément, les différentes méthodes de traduction utilisées sont les suivantes :

- la version "naïve" de la traduction avec des objets, dans laquelle seules les méthodes renvoyant une collection sont transformées en variable d'instance
- la version optimisée de la traduction avec des objets (voir 4.4.3)
- la traduction à base d'enregistrements.

On a par ailleurs comparé ces résultats avec la fonction correspondante d'Axiom [WBD+95].

Les figures A.1 et A.2 représentent le temps mis pour factoriser chaque polynôme, en secondes et en proportion du temps mis par Axiom respectivement. Pour les polynômes de degré compris entre 200 et 1000, on peut constater que la traduction objet est environ 2 fois plus rapide qu'Axiom, tandis que la traduction par enregistrement prend seulement 40 % du temps d'Axiom. Pour les degrés supérieurs, ces performances diminuent légèrement, tout en restant nettement supérieures à Axiom. En ce qui concerne les polynômes de degré inférieurs à 200, où Foc est jusqu'à 5 fois plus rapide qu'Axiom, les résultats sont sans doute dus au fait que Axiom précalcule un nombre important de valeurs, ce qui a un coût important lorsque le temps de calcul global est faible.

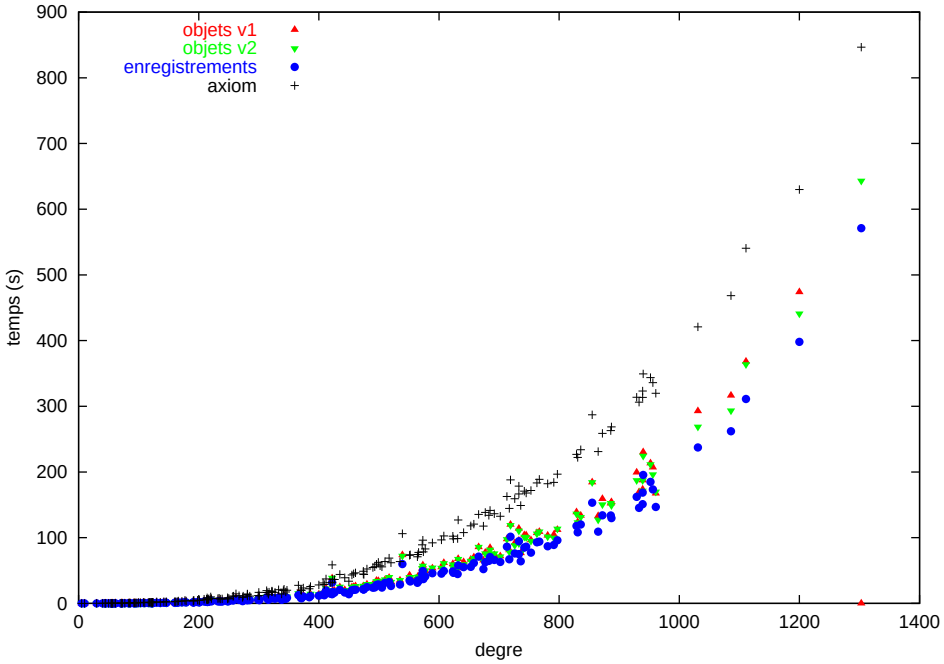


FIG. A.1 – temps de calculs absolus

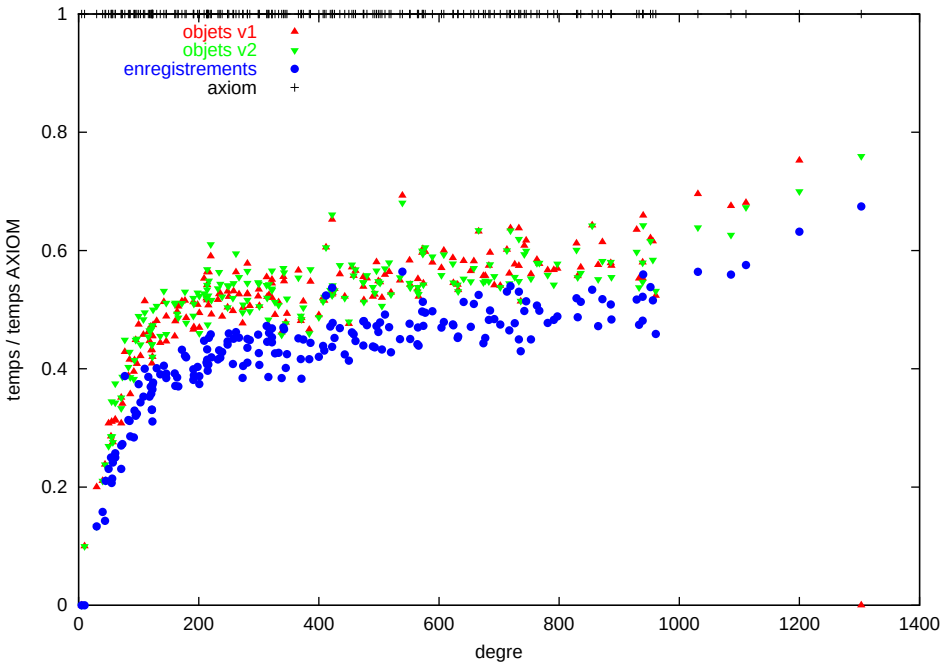


FIG. A.2 – comparaison par rapport à Axiom

Bibliographie

[AC94] Martin Abadi and Luca Cardelli. « A theory of primitive objects : Untyped and first-order systems ». In Springer-Verlag, editor, *Theoretical Aspects of Computer Software*, pages 296–320, avril 1994.

[AL91] Andrea Asperti and Giuseppe Longo. *Categories, Types, and Structures.* The MIT Press, Cambridge, Massachusetts, 1991.

[AZ98] Davide Ancona and Elena Zucca. « An Algebra of Mixin Modules ». In Springer-Verlag, editor, *WADT'97*, number 1376 in LNCS, pages 92–106. Springer-Verlag, 1998.

[B+97] Bruno Buchberger and others. « A survey on the Theorema project ». In W. Kuechlin, editor, *Proceedings of ISSAC'97.* ACM Press, 1997.

[Bet98] Gustavo Betarte. « *Dependent Record Types and Formal Abstract Reasoning : Theory and Practice* ». PhD thesis, University of Göteborg, 1998.

[BHC95] Clemens Ballarin, Karsten Homann, and Jacques Calmet. « Theorems and algorithms : an interface between Isabelle and Maple ». In A. Levelt, editor, *Proceedings of ISSAC*, pages 150–157, Montréal, Canada, 1995. ACM Press.

[BHMR98] Sylvain Boulmé, Thérèse Hardin, Valérie Ménissier, and Renaud Rioboo. « Rapport Foc ». Rapport de recherche, LIP6, 1998.

[BHR99] Sylvain Boulmé, Thérèse Hardin, and Renaud Rioboo. « Modules, Objets et Calcul Formel ». In *JFLA*, 1999.

[BHR00] Sylvain Boulmé, Thérèse Hardin, and Renaud Rioboo. « Polymorphic Data Types, Objects, Modules and Functors : is it too much ? ». Research Report 14, LIP6, 2000. disponible à `http://www.lip6.fr/reports/lip6.2000.014.html`.

[BHR01] Sylvain Boulmé, Thérèse Hardin, and Renaud Rioboo. « Some Hints for polynomials in the Foc project ». In *Calculemus 2001 Proceedings*, juin 2001.

[Bib02] Lazar Bibin. « Implantation de la clôture réelle en FOC ». Rapport de stage de D.E.A, Université Paris 6, septembre 2002.

[BJV98] Bruno Buchberger, Tudor Jebelean, and D. Vasaru. « Theorema : A System for Formal Scientific Training in Natural Language Presentation. ». In I. Tome T. Ottmann, editor, *Proceedings of ED-MEDIA / ED-TELECOM 98*, 1998.

[BKM95] Robert Boyer, Matt Kaufmann, and J Strother Moore. « The Boyer-Moore Theorem Prover and Its Interactive Enhancement », 1995.

[Bou97] Sylvain Boulmé. « Vers la spécification formelle d'une preuve d'un algorithme non trivial de calcul formel ». Stage de D.E.A, Université de Paris 6, 1997.

[Bou00] Sylvain Boulmé. « *Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel* ». Thèse de doctorat, Université Paris 6, 2000.

[BSvH+93] Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill. « Rippling : A Heuristic for Guiding Inductive Proofs ». *Artificial Intelligence*, 62(2) :185–253, 1993.

[Cap02] Venanzio Capretta. « *Abstraction and Computation* ». PhD thesis, Katholieke Univeristeit Nijmegen, avril 2002.

[Car86] John Cartmell. « Generalised Algebraic Theories and Contextual Categories ». *Annals of Pure and Applied Logic*, 32 :209–243, 1986.

[CKL01] Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. « The Rho Cube ». In *Proc. of FOSSACS, Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Sciences, 2001. `http://www.loria.fr/~lliquori/PAPERS/fossacs-00.ps.gz`.

[CMR01] Maura Cerioli, Peter Mosses, and Gianna Reggio, editors. *Proceedings of the 15th International Workshop on Algebraic Development Techniques and the General Workshop of the CoFI WG*, Genova, Italie, avril 2001.

[Coq02] The Coq Development Team. « *The Coq Proof Assistant Reference Manual Version 7* ». INRIA-Rocquencourt, 2002.

[Cou01] Judicaël Courant. « $\mathcal{MC}_2$: A Module Calculus for Pure Type Systems ». Research Report 1292, LRI, septembre 2001.

[Del00] David Delahaye. « A Tactic Language for the System Coq ». In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island*, volume 1955, pages 85–95. Springer-Verlag LNCS/LNAI, novembre 2000.

[DGKM01] Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, and Ursula Martin. « Computer Algebra meets Automated Theorem Proving : A Maple-PVS Interface ». In *Proceedings of the Calculemus Workshop*, 2001.

[DGW97] Stéphane Dalmas, Marc Gaëtano, and Stephen Watt. « An OpenMath 1.0 Implementation ». In W. Kuechlin, editor, *Proceedings of ISSAC 97*. ACM Press, 1997.

[DHK98] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. « Theorem proving modulo ». Research Report 3400, INRIA, 1998.

[DHK01] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. « HOL-$\lambda\sigma$: an intentional first-order expression of higher-order logic ». *Mathematical Structures in Computer Science*, 11(1) :21–45, 2001.

[DHL98] Pietro Di Gianantonio, Furio Honsell, and Luigi Liquori. « A Lambda Calculus of Objects with Self-inflicted Extension ». In *Proc. of ACM-SIGPLAN OOPSLA, International Symposium on Object Oriented, Programming, System, Languages and Applications*, pages 166–178. The ACM Press, 1998. `http://www.loria.fr/~lliquori/PAPERS/oopsla-98.ps.gz`.

[DST93] James Davenport, Yvon Siret, and Evelyne Tournier. *Calcul Formel*. Masson, 1993.

[DV98] Catherine Dubois and Véronique Viguié Donzeau-Gouge. « A step towards the mechanization of partial functions : domains as inductive predicates ». In *Proceedings of CADE-15, Workshop on Mechanization of Partial Functions*, 1998.

[Fec01] Stéphane Fechter. « Une sémantique pour FoC ». Rapport de D.E.A., Université Paris 6, septembre 2001.

[FEV93] Peter Fritzon, Vadim Engelson, and Lars Viklund. « Variant Handling, Inheritance and Composition in the ObjectMath Computer Algebra Environment ». In *Proceedings of DISCO'93 – International Symposium on the Design and Implementation of Symbolic Computation Systems*, volume 722 of *LNCS*, Gmunden, Austria, septembre 1993. Springer Verlag.

[FGFT95] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. « The IMPS User's Manual ». Technical Report M-93B138, The MITRE Corporation, 202 Burlington Road, Bedford, MA 01730-1420, USA, novembre 1995.

[FK00] Andreas Franke and Michael Kohlhase. « System Description : MBASE, an Open Mathematical Knowledge Base ». In *Conference on Automated Deduction*, 2000.

[GPWZ01] Herman Geuvers, Randy Pollack, Freek Wiedijk, and Jan Zwanenburg. « The algebraic hierarchy of the FTA project ». In *Proceedings of the Calculemus Workshop*, 2001.

[GR97] Jacques Garrigue and Didier Rémy. « Extending ML with Semi-Explicit Higher-Order Polymorphism ». In *International Symposium on Theoretical Aspects of Computer Software*, volume 1281

of *Lecture Notes in Computer Science*, pages 20–46. Springer, septembre 1997.

[Gra02] Jérôme Grandguillot. « Réutilisation de preuves : une étude pour le système Foc ». rapport de stage de D.E.A, Université d'Évry Val d'Essone, 2002.

[Har95] John Harrison. « Metatheory and Reflection in Theorem Proving : A Survey and Critique ». Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995.

[Hir00] Tom Hirschowitz. « Modules mixins : Typage et implantation ». rapport de stage de D.E.A, Université Paris VII, 2000.

[HL] Tom Hirschowitz and Xavier Leroy. « Mixin modules in a call-by-value setting ». version longue de [HL02].

[HL94] Robert Harper and Mark Lillibridge. « A Type-Theoretic Approach to Higher-Order Modules with Sharing ». In *21st Symposium on Principle of Programming Languages*, 1994.

[HL02] Tom Hirschowitz and Xavier Leroy. « Mixin modules in a call-by-value setting ». In D. Le Métayer, editor, *Programming Languages and Systems, ESOP'2002*, volume 2305 of *Lecture Notes in Computer Science*, pages 6–20, 2002.

[How80] W.A. Howard. « *To H.B. Curry, Essays on combinatory logics, lambda calculus and formalism* », Chapitre The formulae-as-type notion of construction, pages 479–490. Academic Press, 1980.

[HT98] John Harrison and Laurent Théry. « A Skeptic's Approach to Combining HOL and Maple ». *Journal of Automated Reasoning*, 21 :279–294, 1998.

[Jac94] Paul Jackson. « Exploring Abstract Algebra in Constructive Type Theory ». In *Proceedings of 12th International Conference on Automated Deduction*, juillet 1994.

[Jon96] Mark P. Jones. « Using Parameterized Signatures to Express Modular Structure ». In *Proceedings of 23d Symposium on Principles of Programming Languages*. ACM, janvier 1996.

[JS92] Richard D. Jenks and Robert S. Stutor. *AXIOM, The Scientific Computation System*. Springer-Verlag, 1992.

[Koh03] Michael Kohlhase. « *OMDoc : An Open Markup Format for Mathematical Documents (Version 1.1)* ». `http://www.mathweb.org/omdoc`, 2003.

[Lam93] L. Lamport. « How to Write a Proof ». research report, Digital Equipments Corporation, février 1993.

[LDG+02] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. « *The Objective Caml system, release 3.06* », 2002.

[Ler90] Xavier Leroy. « The Zinc Experiment : an Economical Implementation of the ML Language ». Technical Report 117, INRIA, février 1990.

[Ler00] Xavier Leroy. « A modular module system ». *Journal of Functional Programming*, 10(3) :269–303, 2000.

[LGF00] Mikel Luján, John R. Gurd, and T. L. Freeman. « OoLaLa : an Object Oriented Analysis and Design of Numerical Linear Algebra ». *ACM SIGPLAN Notices*, 35(10) :229–252, 2000.

[Liq96] Luigi Liquori. « An Extended Theory of Primitive Objects ». Technical Report CS-23-96, Computer Science Department, University of Turin, 1996. `http://www.loria.fr/~lliquori/PAPERS/abadi-cardelli-97.ps.gz`.

[Liq98] Luigi Liquori. « On Object Extension ». In *Proc. of ECOOP, European Conference on Object Oriented Programming*, volume 1445 of *Lecture Notes in Computer Sciences*, pages 498–552. Springer Verlag, 1998. `http://www.loria.fr/~lliquori/PAPERS/ecoop-98.ps.gz`.

[LLMS00] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark Shields. « Implicit Parameters : Dynamic Scoping with Static Types ». In *Symposium on Principles of Programming Languages*, pages 108–118, 2000.

[LMT96] Carla Limongelli, Giuseppina Malerba, and Marco Temperini. « Uniform Representation of Basic Algebraic Structures in Computer Algebra ». Technical Report RT-INF-18-96, Università degli Studi di Roma 3, mai 1996.

[Maa01] Manuel Maarek. « Ecriture d'un parser pour Foc en Camlp4 ». Travail d'initiation à la recherche, Université Paris 6, juin 2001.

[Man01] Louis Mandel. « Factorisation de polynômes sur les corps finis ». Rapport de magistère, Université Paris 6, 2001.

[Mec00] Serge D. Mechveliani. « Haskell and Computer Algebra ». available at `http://www.botik.ru/~mechvel/papers.html`, 2000.

[Mec01] Serge D. Mechveliani. « Computer algebra with Haskell : applying functional-categorial-'lazy' programming ». In *Proceedings of CAAP-2001*, Dubna, Russia, 2001. `http ://www.botik.ru/ mechvel/papers.html`.

[Mil78] Robin Milner. « A Theory of Type Polymorphism in Programming ». *Journal of Computer and System Sciences*, 17 :348–375, 1978.

[MP03] Manuel Maarek and Virgile Prevosto. « FoCDoC : the documentation system of FoC ». In *Proceedings of Calculemus*, Rome, Italie, septembre 2003.

[MT94] David B. MacQueen and Mads Tofte. « A Semantics for Higher-order Functors ». In *European Symposium on Programming*, pages 409–423. Springer-Verlag, LNCS 788, 1994.

[OCRZ03] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. « A Nominal Theory of Objects with Dependent Types ». In *Proc. FOOL 10*, janvier 2003. `http://www.cis.upenn.edu/~bcpierce/FOOL/FOOL10.html`.

[Oho96] Atsushi Ohori. « A polymorphic record calculus and its compilation ». *ACM Transactions on Programming Languages and Systems*, 17(6) :844–895, 1996.

[OS01] S. Owre and N. Shankar. « Theory Interpretations in PVS ». Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, Avril 2001.

[Par95] Catherine Parent. « *Synthèse de preuve de programmes dans le calcul des constructions inductives* ». Thèse de doctorat, École normale supérieure de Lyon, janvier 1995.

[PD02] Virgile Prevosto and Damien Doligez. « Inheritance of Algorithms and Proofs in the Computer Algebra Library Foc ». *Journal of Automated Reasoning*, 29(3-4) :337–363, 2002. Special Issue on Mechanising and Automating Mathematics, In Honor of N.G. de Bruijn.

[PDH02] Virgile Prevosto, Damien Doligez, and Thérèse Hardin. « Algebraic Structure and Dependent Records ». In Sofiène Tahar César Muñoz and Víctor Carreño, editors, *Proceedings of TPHOLs 02*. Springer-Verlag, août 2002.

[PJ03] Virgile Prevosto and Mathieu Jaume. « Making proofs in a hierarchy of mathematical structures ». In *Proceedings of Calculemus*, septembre 2003.

[Pol00] Randy Pollack. « Dependently Typed Records for Representing Mathematical Structures ». In *TPHOLs'00*. Springer-Verlag, 2000.

[Pot99] Loïc Pottier. « contrib Algebra pour coq », mars 1999. `http ://pauillac.inria.fr/coq/contribs-eng.html`.

[Pre00] V. Prevosto. « Vers une interface utilisateur pour Foc ». Rapport de D.E.A., Université Paris 6, septembre 2000.

[Pre01] V. Prevosto. « Prototype d'interface utilisateur de la librairie Foc ». In *JFLA*, janvier 2001.

[PT98] Erik Poll and Simon Thompson. « Adding the axioms to Axiom : Towards a system of automated reasoning in Aldor ». In *Proceedings of the Workshop Types-Calculemus*, juillet 1998.

[Ran02] Silvio Ranise. « Combining Generic and Domain Specific Reasoning by Using Contexts ». In Jacques Calmet and others, editors, *Proceedings of AISC-Calculemus*, volume 2385 of *LNAI*, pages 305–318, Marseille, juillet 2002. Springer-Verlag.

[Rém98] Didier Rémy. « *Des enregistrements aux objets* ». Mémoire d'habilitation à diriger des recherches, Université Paris 7, septembre 1998.

[RR96] John H. Reppy and Jon G. Riecke. « Classes in Object. ML via Modules ». Presented at FOOL'3 Workshop, juillet 1996.

[RT99] Chris Ryder and Simon Thompson. « Aldor meets Haskell ». Technical Report 15-99, Computing Laboratory, University of Kent, octobre 1999.

[Rud] Piotr Rudnicki. « A Note on How to write a proof ».

[RV98] Didier Rémy and Jérôme Vouillon. « Objective ML : An effective object-oriented extension to ML ». *Theory and Practice of Object Systems*, 4(1) :p. 27–50, 1998.

[Sal96] Anne Salvensen. « Modules as classes in logical frameworks with explicit substitution ». In *Selected papers from the 8th Nordic Workshop on Programming Theory*, pages 199–214, Oslo, Norway, décembre 1996.

[SP01] Mark Shields and Simon Peyton Jones. « Object-Oriented Style Overloading for Haskell ». In *First Workshop on Multi-language Infrastructure and Interoperability (BABEL'01), Firenze, Italy*, septembre 2001.

[VF92] Lars Viklund and Peter Fritzon. « An object-oriented language for symbolic computation applied to machine element analysis ». In Paul S. Wang, editor, *Papers from the international symposium on Symbolic and algebraic computation*, Series-Proceeding, pages 397–405, Berkeley, California, 1992. SIGSAM, ACM Press.

[Vou98] Jérôme Vouillon. « Using modules as classes ». In *Informal proceedings of the FOOL'5 workshop*, 1998. Disponible à `http://pauillac.inria.fr/~remy/fool`

[Vou00] Jérôme Vouillon. « *Conception et réalisation d'une extension du langage ML avec des objets* ». Thèse, Université Paris 7, octobre 2000.

[WBD+95] S. Watt, P. Broadbery, S. Dooley, P. Iglio, S. Morrison, J. Steinbach, and R. Sutor. « *AXIOM Library Compiler User Guide* ». NAG Ltd, mars 1995.

[Xi01] Hongwei Xi. « Dependent Types for Program Termination Verification ». In *Proceedings of 16th IEEE Symposium on Logic in Computer Science*, Boston, juin 2001.

Index