Thèse de doctorat de l'Université Paris 6

Spécialité :
Infomatique

Présentée

Par :
Sylvain BOULMÉ

Avec le titre :

Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel

Soutenue le
22 décembre 2000

Devant la Commission d'examen composée de :

Mme Thérèse HARDIN	Directrice de thèse
M Gilles KAHN	Rapporteur
M Sergei SOLOVIEV	Rapporteur
M Benjamin WERNER	Rapporteur
Mme Dominique DUVAL	Examinatrice
M Daniel LAZARD	Examinateur
Mme Christine PAULIN-MOHRING	Examinatrice
M Renaud RIOBOO	Examinateur

Remerciements

Je tiens à adresser mes remerciements à Thérèse Hardin, ma directrice de thèse. Créatrice et maîtresse d'œuvre du projet FOC, elle m'a donné la chance de travailler au sein de ce projet ambitieux et stimulant. Cette thèse doit beaucoup à sa célèbre maïeutique. Ses efforts, pour déchiffrer mes hiéroglyphes et pour redresser mes phrases qui se mordent parfois la queue, ont toute ma gratitude.

Je remercie tout particulièrement mes trois rapporteurs Gilles Kahn, Sergei Soloviev et Benjamin Werner d'avoir accepté cette pénible tâche. C'est un immense honneur pour moi que Gilles Kahn ait trouvé de la place dans son emploi du temps chargé pour rapporter cette thèse et assister à la soutenance. Je remercie également Sergei Soloviev de m'avoir accueilli à Toulouse. Ses conseils au moment où je commençais à rédiger ma thèse, en particulier sur les chapitres 7 et 8 m'ont été fort précieux. Je remercie Benjamin Werner de ses réponses à mes questionnements autour de Coq, et de ses encouragements très chaleureux.

Je suis flatté que Dominique Duval, Daniel Lazard et Christine Paulin-Mohring aient accepté d'être dans mon jury. Je remercie Dominique Duval pour la rencontre intéressante et sympathique avec le groupe de travail "Esquisses et Calcul Formel". Je remercie Daniel Lazard d'avoir suivi mon travail avec intérêt, son point de vue m'a souvent permis de remettre ma démarche en perspective. Je remercie Christine Paulin-Mohring de m'avoir éclairé sur de nombreux points à propos de Coq, et bien sûr de sa gentillesse légendaire.

Je suis extrêmement reconnaissant à Renaud Rioboo. Il pourrait être considéré à juste titre comme mon deuxième directeur de thèse. Il m'a fait découvrir un domaine tout à fait passionnant : le Calcul Formel. Sa grande expérience, sa profonde intuition, et sa rigueur extrême en matière de programmation dans ce domaine ont été pour moi à la fois une source d'émerveillement et d'inspiration. J'ai beaucoup appris à son contact.

Je remercie aussi infiniment Loïc Pottier. En particulier, son idée du codage des télescopes en Coq m'a débloqué à un moment où je commençais sérieusement à m'enliser et à me demander comment j'allais arriver à faire quelque chose de ce sujet de thèse : elle m'a permis de réellement démarrer la thèse.

Je remercie tous les membres passés ou présents de FOC que je n'ai point encore cités : Damien Doligez, Catherine Dubois, Daniel Hirschkoff, Valérie Ménissier-Morain (qui a aussi relu ce document), Virgile Prévosto et Véronique Viguié Donzeau-Gouge.

Je remercie tous les membres de l'action CFC (Calcul Formel Certifié), du projet CROAP (maintenant LEMME), du projet COQ (maintenant LOGICAL) et du projet CRISTAL, avec qui les échanges ont toujours été enrichissants. Notamment : Bruno Barras, Yves Bertot, Judicaël Courant, Gilles Dowek, Hugo Herbelin, Tom Hirschowitz, Gérard Huet, Xavier Leroy, Patrick Loiseleur, Micaela Mayero, Olivier Pons, Didier Rémy, Laurence Rideau, Laurent Théry, Jérôme Vouillon.

Je remercie également les autres personnes avec qui j'ai eu des échanges à propos de mon travail : Zhaohui Luo, Randy Pollack et Thierry Coquand.

Je remercie les membres passés ou présents du pôle MFI pour leur compagnie plutôt sympathique. Je remercie tout particulièrement David Massot, notre supersecrétaire et Jean-Michel Frenod, notre administrateur système.

Je remercie enfin les membres de mon entourage, notamment Anne Parreau qui a relu les premiers chapitres de cette thèse avec beaucoup d'attention et de pertinence.

Table des matières

Première partie

Problématiques et outils

Chapitre 1

Introduction

Cette thèse s'inscrit dans le cadre du projet FOC (acronyme de Formel+Ocaml+Coq), dont le but est de développer un environnement de programmation certifiée dédié au Calcul Formel (voir [Bou99b]). Cette thèse s'intéresse donc à deux domaines : les méthodes formelles (domaine principal) et le calcul formel (domaine d'application). On commence donc par donner une présentation des méthodes formelles dans la section 1.1. On décrit ensuite le sujet de la thèse dans la section 1.2, en insistant sur les motivations de ce sujet. On présente ensuite le calcul formel, section 1.3. Enfin, la section 1.4 détaille le cahier des charges initial de cette thèse, et la section 1.5 donne le plan de cette thèse.

1.1 Méthodes formelles et programmation certifiée

1.1.1 Problèmes des bogues informatiques

Les systèmes informatiques ont envahi notre quotidien. Ils contrôlent ascenseurs, machines à laver, systèmes de freinage des voitures... Ils sont au cœur des systèmes de télécommunications. Capables de traiter automatiquement et rapidement une grande masse d'informations, ils peuvent soulager l'humanité de tâches pénibles et répétitives. Mieux, ils accomplissent "facilement" certaines tâches difficiles, voire impossibles pour les humains.

La précision et la rapidité de réaction qu'ils procurent les rendent utiles dans de nombreux domaines critiques : transports, chirurgie, contrôle des usines produisant ou utilisant des produits dangereux, etc. En particulier, leur caractère déterministe (ils réagissent toujours de la même façon, dans des conditions données) et inusable (en ce qui concerne les logiciels) semble leur procurer une fiabilité rassurante. Néanmoins, la complexité de leur fonctionnement rend leur comportement très difficilement prédictible face à une situation pour laquelle ils n'ont pas été prévus : le bogue de l'an 2000 en est sans doute la meilleure illustration.

Pire, cette complexité est telle que parfois leurs concepteurs sont incapables de garantir leur bon fonctionnement en "utilisation normale". Il n'y a qu'à consulter les licences de beaucoup de logiciels grand public pour s'en convaincre : l'informatique est le seul domaine où les marchands s'autorisent à vendre un produit sans en garantir le fonctionnement.

Ainsi, si on prend en compte les bogues, le déterminisme des systèmes informatiques qui les rend tellement rassurants, peut conduire à des catastrophes très graves. On pense à l'incident de janvier 90 sur le réseau téléphonique d'AT&T aux Etats-Unis. Celui-ci s'est retrouvé hors service pendant 9 heures : il a bloqué près de 5 millions d'appels. Dans des conditions de fonctionnement particulières, une portion d'un logiciel installé sur 114 routeurs ne s'est pas comportée comme elle aurait dû : les 114 routeurs ont alors eu le même comportement inattendu, bloquant ainsi tout le réseau.

Des anomalies comme celles-ci ne sont pas isolées : elles se produisent aussi bien dans les produits informatiques "grand public" (bogue algorithmique du processeur Pentium lors de certaines divisions sur les flottants, cf. [Mul95]) que dans des applications critiques (dans les trains, les avions, les fusées comme Ariane 5, ...). Dans le domaine de la chirurgie, on peut citer l'exemple d'un accélérateur de particules destiné à soigner des malades du cancer qui au printemps 86 a donné à certains patients une dose d'irradiations qui leur a été fatale. Cet accélérateur, fabriqué par la US Food and Drug Administration et entièrement piloté par un programme informatique, avait une anomalie lors de l'édition des paramètres du dosage par le technicien chargé de l'opération : si le technicien utilisait "la flèche vers le haut" pour éditer ce menu, cela provoquait une erreur dans le système, signalée par un mystérieux "MALFUNCTION 54", mais dont le résultat était une irradation de 25000 rads, soit 25 fois la dose mortelle, et 250 fois la dose usuelle en séance (anecdotes tirées de [Xan00]).

Si il y a tant de bogues informatiques, c'est que déterminer la correction d'un système informatique n'est pas un problème facile. On sait depuis les débuts de l'informatique qu'il ne peut pas exister de méthode automatique ou d'algorithme permettant de le faire : ce problème est indécidable. En fait, même le problème de la terminaison des programmes informatiques ("est-ce qu'un programme fournit toujours un résultat") est indécidable.

Ainsi éliminer les bogues informatiques est un travail ingrat : cela coûte cher (en temps humain) et malgré tous les efforts investis dans la sûreté informatique[1], on n'aura jamais aucune garantie qu'un système donné est sans bogue. Le zéro-défaut en informatique semble une complète utopie. Pourtant, comme les exemples précédents l'attestent, avoir des systèmes informatiques aussi fiables que possible est un enjeu majeur de l'informatique, voire de l'ensemble de la société. La société pourra-t-elle tolérer de remettre la vie des gens entre les mains d'une technologie non complètement maîtrisée ? L'informatique doit avoir pour ambition de produire des outils aussi proches du zéro-défaut que possible.

1.1.2 Sûreté informatique

Le développement d'un système informatique, et d'un logiciel en particulier, est le résultat d'une succession de reformulations du problème à résoudre dans divers langages, ces reformulations étant effectuées par des humains ou par des machines. On passe ainsi d'une description humaine (langage humain, dessins, ...) et non effective du problème, à une description effective comme une suite de bits, qui se traduisent eux-mêmes en termes de circuits électroniques "ouverts" ou "fermés". A chacune de ces reformulations, on transforme l'information, et on peut introduire des erreurs qui potentiellement pourront entrainer des anomalies de fonctionnement.

Les bogues informatiques vont donc du problème mal posé à la banale erreur d'étourderie, en passant par "des solutions répondant de manière approximative au problème initial". Par exemple, le fameux bogue d'AT&T décrit ci-dessus se trouvait dans un programme écrit en C qui contenait un choix multiple dont le programme était censé sortir par une instruction "break". Or cette instruction étant elle-même située à l'intérieur d'une autre expression conditionnelle, elle n'était pas toujours exécutée comme attendu (cf. [Xan00]).

La sûreté des systèmes concerne donc tout le développement du système informatique, depuis la définition du problème (spécification ou cahier des charges) jusqu'au produit final. Suivant l'adage

[1]Sûreté informatique : ici, fonctionnement correct des systèmes informatiques. A ne pas confondre avec les problèmes dits de sécurité qui concernent la protection des systèmes vis-à-vis d'actes malveillants.

"plus un bogue est détecté tôt, moins il coûte cher à corriger", le génie logiciel cherche aujourd'hui à éliminer les bogues de plus en plus tôt dans le processus de développement : de la chasse aux bogues, on est passé à la prévention des bogues. Mais malgré tout, la validation (c'est-à-dire les tests sur le produit final, tel que le reçoit l'utilisateur, et dans ses conditions d'utilisation) reste indispensable.

Parmi les différentes méthodes de sûreté des systèmes, on appelle *méthodes formelles* celles qui cherchent à démontrer mathématiquement que le système est sans bogue. Dans ces méthodes, on fait d'un côté une description mathématique de ce que doit faire le système (spécification) et d'un autre côté un modèle mathématique du système informatique (implantation) censé répondre à la spécification. Du point de vue des méthodes formelles, le système informatique est sans bogue, si on connaît une démonstration mathématique du fait que la description mathématique de l'implantation correspond à la spécification mathématique. (Voir [Mon96] pour un panorama des méthodes formelles).

Les méthodes formelles ne prétendent pas éliminer tous les bogues. Il faut déjà montrer que la spécification mathématique est cohérente (elle ne fait pas d'hypothèses contradictoire), sinon on peut démontrer n'importe quoi. Mais surtout, il reste deux traductions (une traduction de la spécification, du langage humain vers le langage mathématique, et une traduction du système informatique, du langage mathématique vers le langage machine) au cours desquelles une introduction de bogues est toujours possible.

La vérification de la correction des systèmes informatiques étant souvent relativement pénible et délicate (on ne veut pas faire d'erreur), les méthodes formelles s'utilisent souvent avec un logiciel chargé de vérifier la correction de la preuve. La description mathématique du système informatique dans ce logiciel peut parfois être traduite automatiquement vers du code machine exécutable, ou des circuits électroniques. Cela évite une traduction "manuelle" fort pénible et donc susceptible d'introduire beaucoup d'erreurs d'étourderie. Ceci dit, il reste le problème de savoir quelle confiance on peut avoir dans ce logiciel d'aide à la démonstration et dans ce compilateur (ou même dans la machine sur-laquelle est exécutée le programme) : ce sont typiquement des logiciels dont il est très difficile de faire **entièrement** la preuve.

Les méthodes formelles n'arrivent donc pas en opposition aux méthodes traditionnelles fondées plus sur le test, qui resteront en fait toujours indispensables. De plus, étant donné leur coût de mise en œuvre encore lourd à l'heure actuelle, elles ont surtout un intérêt au moment de la conception du système, c'est-à-dire lorsqu'elles sont intégrées au processus de développement. Le raisonnement mathématique sur le système informatique à construire peut en effet permettre de découvrir des propriétés cachées de celui-ci, et le rendre plus efficace, en particulier en ayant une approche plus offensive (utiliser moins de vérification dynamique, lors de l'exécution, par le système lui-même).

Plus généralement, les méthodes formelles peuvent avoir un autre intérêt que la *"certification"* des systèmes informatiques. Elles peuvent aussi être à la base d'outils d'aide à la conception de systèmes informatiques. Par exemple, le logiciel CENTAUR (voir [Bor89]) est un système générique d'environnement interactif de programmation. Étant données la syntaxe et la sémantique [2] formelles d'un langage de programmation, CENTAUR produit un environnement interactif de programmation adapté à ce langage. Cet environnement comprend un éditeur structuré (par exemple avec un copier/coller sur des expressions syntaxiquement correctes du langage), un interpréteur/déboggueur, etc, le tout dans une interface graphique agréable.

[2] La sémantique formelle d'un langage, c'est la description mathématique de ce que "font" les constructions syntaxiques de ce langage

1.1.3 La programmation certifiée

La programmation certifiée cherche à intégrer le processus de preuve à la programmation des logiciels. Un environnement de programmation certifiée est donc un logiciel qui permet de spécifier, programmer et prouver que les programmes sont corrects vis-à-vis des spécifications. On appelera *spécification formelle* la spécification d'un programme écrite dans cet environnement. Les autres spécifications (en langue naturelle ou même en langage mathématique) seront dites *informelles*. Un *programme certifié* est un programme qui a été prouvé correct vis-à-vis de sa spécification formelle. Pour insister sur ce point, un programme certifié ne va pas forcément donner lieu à un exécutable sans bogue. D'une part, on a en fait juste montré que le **source** du programme, dans le contexte de la sémantique du langage, est "sans bogue" vis-à-vis de sa spécification. Des bogues peuvent être introduits lors de la compilation de ce programme. Mais dans la mesure où le compilateur sera largement utilisé, on peut imaginer que ses bogues les plus fréquents seront détectés et corrigés, et que le code exécutable produit sera donc lui-même quasiment sans bogue. D'autre part, on peut imaginer qu'un tel programme interagit avec d'autres programmes qui eux-mêmes n'ont pas été certifiés. Et a fortiori, la correction de cette interaction n'a pas été prouvée. Comme on l'a signalé ci-dessus, il n'est pas possible de tout prouver entièrement : la machine, le système d'exploitation, le compilateur, l'environnement de programmation certifiée... Il faut donc savoir s'arrêter. Mais cela signifie qu'il faut vérifier (informellement) que les parties certifiées du logiciel font de bonnes hypothèses sur les parties non certifiées, et réciproquement : il ne sert à rien d'avoir un programme certifié si on l'utilise en dehors de son domaine d'application.

A priori, lorsqu'un (bon) programmeur écrit un programme, il a en tête de manière plus ou moins consciente tous les arguments permettant de prouver que son programme est correct. Par exemple, dans les langages impératifs simples, les deux sources d'indécidabilité de la correction d'un programme sont les invariants de boucles (propriétés qui sont préservées à chaque passage de la boucle) et les variants de boucles (propriétés qui garantissent que la boucle va terminer). Ces deux types de propriétés non inférables par un système d'aide à la démonstration peuvent être données directement par le programmeur, car il les a en tête quand il écrit une boucle. D'ailleurs, ce genre de propriétés est souvent mis en commentaire dans le source du programme (sauf quand elles sont complètement triviales), car elles sont indispensables pour comprendre le programme lors d'une relecture par un humain. En fait, le source d'un programme certifié est en quelque sorte un programme dans lequel les commentaires sont écrits dans un langage mathématique, et permettent à un système d'aide à la démonstration d'inférer les propriétés de ce programme.

Malheureusement, le tableau présenté ci-dessus est un peu trop idyllique pour correspondre à la réalité. D'une part, les langages de programmation réels utilisent de nombreux traits assez compliqués à manipuler dans les systèmes d'aide à la démonstration, même si les quelques instructions impératives évoquées ci-dessus suffisent du point de vue théorique de la calculabilité. Par exemple, les pointeurs, les exceptions (l'instruction "break" qui a fait le malheur d'AT&T), les techniques de programmation objet comme la liaison dynamique, sont autant de traits qui ont un comportement difficilement prédictible statiquement (sans exécuter le programme) et par conséquent difficiles à intégrer dans les environnements de programmation certifiée. D'autre part, les propriétés désirées par le programmeur ne sont pas toujours facilement exprimables en termes mathématiques. Cette traduction vers le langage mathématique en rendant les objets plus abstraits, les rend parfois moins intuitifs, ce qui peut provoquer des erreurs d'interprétation. Le passage des spécifications informelles aux spécifications formelles est donc un problème en soi. Pour que ce passage se fasse en douceur, des techniques comme le raffinement ont déjà montré leur efficacité (voir [Abr96]) : il s'agit de décrire les spécifications formelles petit à petit, en partant des spécifications les plus générales, les plus abstraites et les plus importantes, vers celles qui sont les plus particulières, et qui relèvent des choix d'implantation. A chaque étape, on *raffine* donc les spécifications abstraites en spécifications plus effectives, qui se rapprochent du programme final. On montre au cours de ce raffinement que les spécifications effectives vérifient bien les propriétés impliquées par les spécifications abstraites. L'idée est que les spécifications abstraites, plus concises et plus générales sont plus intelligibles et proches des spécifications informelles.

1.2 Sujet et lien avec le calcul formel

Dans le cadre de cette thèse, les programmes auxquels on s'intéresse sont des programmes de calcul formel. Les spécifications des programmes sont déjà connues mathématiquement. On connait les structures mathématiques dans lesquelles s'appliquent ces programmes, et les algorithmes qu'ils doivent implanter. On ne se confronte donc pas complètement au problème de la formalisation des spécifications, car une grande partie de ce travail est déjà réalisé naturellement par la communauté du calcul formel. Cependant, même en calcul formel, le problème de formalisation des spécifications n'est pas complètement trivial. Il reste souvent un écart assez important entre la preuve "informelle" d'un algorithme et la preuve formelle d'un programme. En particulier, les algorithmes laissent au programmeur le choix du codage effectif des données, et toute la gestion de ce codage. On n'abordera pas ce problème ici.

1.2.1 Sujet

On cherche en fait à intégrer les méthodes formelles au processus de développement des bibliothèques de programmes de calcul formel. On veut construire un environnement de programmation certifiée adapté à ce métier, qui permette de spécifier, programmer et prouver des bibliothèques de calcul formel. On se confronte donc principalement à deux problèmes, étroitement liés, qui peuvent se résumer sous le titre suivant : *"modularité des spécifications, et spécifications de la modularité"*. D'une part, il s'agit de trouver comment organiser les spécifications au sein de ces bibliothèques, qui sont souvent imposantes. D'autre part, il s'agit de refléter au niveau du système de preuves les traits de programmation mis en œuvre dans les langages de calcul formel modernes (comme Axiom, voir [Jen92] et [Wat95]) pour structurer ces bibliothèques. Il faudra en effet montrer que l'utilisation de ces traits dans les programmes est bien correcte vis-à-vis de leur sémantique.

Notre démarche sera bien entendu réflexive : nous utiliserons des méthodes formelles pour développer ce logiciel. L'objectif de cette thèse n'est donc pas le logiciel lui-même (qui est en cours de développement), mais une description mathématique détaillée de ce que doit réaliser ce logiciel.

Il s'agit maintenant de justifier un peu l'intérêt de ce sujet, en montrant en particulier ce que le calcul formel peut apporter aux méthodes formelles, et réciproquement quel est l'intérêt d'utiliser les méthodes formelles en calcul formel. Cette discussion est l'occasion de donner une très brève présentation du calcul formel, qui sera complétée dans la section 1.3.

1.2.2 Utilisation du calcul formel dans les méthodes formelles

Le calcul formel (peut-être mieux nommé par le terme anglais de "computer algebra") est l'ensemble des méthodes de calcul symbolique utilisées pour résoudre des problèmes d'algèbre. Ainsi, en calcul formel, on s'intéresse à la simplification des expressions algébriques, à la factorisation de polynômes, à la résolution de systèmes d'équations, etc.

Par exemple, on peut résoudre avec des algorithmes de calcul formel des problèmes géométriques. En effet, il "suffit" d'exprimer le problème sous forme d'un système d'équations polynômiales à plusieurs inconnues. En utilisant des méthodes de calcul formel on peut alors trouver les solutions de ce système qu'il faut ensuite interpréter géométriquement.

Cela peut en particulier s'appliquer à la démonstration automatique en géométrie. Le calcul formel propose en effet un certain nombre de procédures de décision pour prouver des théorèmes géométriques (voir par exemple [Kap97]).

Une autre application typique de ces outils se trouve par exemple dans les problèmes de cinématique inverse. Imaginons une spécification un peu compliquée d'un bras articulé de robot. Le bras est composé de plusieurs barres de longueur variable (via par exemple un système de pistons). Les barres sont reliées entre elles par une articulation : l'angle entre deux barres successives est variable. Une extrémité du bras étant fixe, on désire connaître la position du bras à donner afin que l'autre extrémité soit en un certain point : le robot peut ainsi saisir un objet situé en ce point.

Pour décrire mathématiquement ce problème, on peut utiliser un système d'équations polynômiales à plusieurs inconnues et plusieurs paramètres : les positions des deux extrémités du robot seront des paramètres ; chacune des longueurs des barres est une inconnue ; et les angles sont représentés par 2 inconnues (une pour le cosinus et une pour le sinus) et une équation, etc. Un système de calcul formel permet alors de résoudre ce système d'équations, et d'exprimer les inconnues en fonctions des paramètres.

D'un certain point de vue, le système d'équations polynomiales est donc une spécification du problème consistant à positionner l'extrémité du bras en un point. Et le système de calcul formel produit donc à partir de cette spécification les programmes de déplacement du bras du robot. Un système de calcul formel est donc une sorte de compilateur de spécifications mathématiques.

En conclusion, le calcul formel peut apporter plusieurs types d'outils aux méthodes formelles : des procédures de simplification d'expressions algébriques, des procédures de décision ou de génération automatique de programmes, dans des domaines comme la géométrie qui ont des correspondances avec des théories algébriques.

1.2.3 Intérêt de programmes de calcul formel certifiés

1.2.3.a Sûreté des programmes de calcul formel

L'intérêt du calcul formel par rapport aux méthodes numériques est de donner des résultats exacts, sans erreurs d'arrondis. Les erreurs d'arrondis peuvent parfois être dramatiques, dans la mesure où, par un effet cahotique, leur propagation peut amener des résultats très loin du résultat exact. Par exemple, dans [Mul89], Muller donne une suite qui converge rapidement vers 100 si on utilise des flottants (y compris des flottants de taille arbitrairement grande), alors que la suite converge mathématiquement vers 6. Seule une arithmétique exacte peut calculer avec précision les termes de cette suite.

Mais le calcul formel perd beaucoup de son intérêt, si ces résultats *exacts* ne sont finalement pas si fiables que cela, car parfois faux. Les programmeurs en calcul formel doivent donc prendre un soin particulier pour établir la sûreté des programmes qu'ils diffusent au reste du monde.

Malheureusement, les tests en calcul formel sont difficiles. Comme pour le reste de l'informatique, il y a essentiellement deux sources de difficultés : d'une part le choix des jeux de test et d'autre part la vérification du résultat. De plus, si l'on veut tester des données de l'ordre de grandeur de celles qui vont être manipulées par ces systèmes, les temps de calcul peuvent être très longs.

La difficulté du choix du jeu de test est qu'il faut trouver des données pertinentes pour tester le programme. Ces données dépendent en partie de la spécification du programme. Par exemple, si on veut tester la multiplication sur les entiers, on va chercher des jeux de tests qui permettent de valider les propriétés de la multiplication : associativité, commutativité, considérer le signe du résultat en fonction du signe des entrées... On va surtout considérer les cas limites qui sont statistiquement de grandes sources de bogues. Ici, on a 0 qui est absorbant, 1 qui est neutre, -1 qui change le signe...

Ceci dit, l'implantation peut par ses choix introduire de nouveaux cas critiques. Par exemple si on souhaite avoir des entiers exacts, mais efficaces, on peut utiliser les entiers machine lorsque leur taille le permet (taille d'un mot-machine) et une autre représentation sinon. Un jeu de test qui n'appliquerait des données que sur des entiers "petits" ne testerait finalement pas l'implantation, mais uniquement les entiers machine. Dans le cas du calcul formel, établir un bon jeu de test peut donc demander une analyse mathématique du programme assez fine. On rediscutera de cette question en section 1.3.3.b, après avoir introduit quelques notions de calcul formel.

Une autre difficulté, une fois des jeux de tests établis, est de savoir si le programme retourne le résultat attendu ou pas. Un premier problème est celui de la taille du résultat qui peut être assez gros sur des exemples "réalistes" : et on sera naturellement amené à vérifier la validité du résultat automatiquement, par un autre programme (appelé oracle). Il faudra alors être sûr de cet oracle. Cela peut être une difficulté, car l'oracle peut finalement être aussi difficile à écrire que le programme à tester. Dans le cas de la multiplication, par exemple, on peut soit utiliser un autre programme de multiplication dont on est sûr (car il est par exemple plus simple, car moins optimisé) ou un programme de division, qu'on teste finalement en même temps que le programme de multiplication, en espérant qu'il n'y ait pas de bogues qui se compensent.

1.2.3.b Les calculs par ordinateur dans la démarche scientifique

Parce qu'il permettent de résoudre des problèmes mathématiques, les systèmes de calcul formel sont amenés aussi à jouer un grand rôle dans l'ensemble des sciences, en particulier les mathématiques et la physique. Les scientifiques utilisant ces programmes ont envie de considérer leur résultats comme scientifiquement valides. Mais à l'heure actuelle peut-on considérer le résultat du calcul d'un ordinateur comme une preuve scientifique ? Vu la difficulté que l'on a pour détecter les anomalies des systèmes informatiques, cette question est source de polémiques dans le monde scientifique. Une des difficultés est que même si un programme donne toujours des résultats corrects lorsqu'on sait vérifier ces résultats, ce n'est pas pour cela qu'il ne donnera jamais un résultat incorrect. Essayons néanmoins d'esquisser une solution à ce problème.

Imaginons que le programme qui a produit le résultat, soit écrit dans un langage de haut niveau, indépendant de la machine. Imaginons que vis-à-vis de la sémantique (démontrée cohérente) de ce langage et de sa spécification, on soit sûr que le programme source est sans bogue. Il est alors fort peu probable pour que le même programme source (sans bogue), compilé par plusieurs compilateurs différents sur plusieurs machines différentes reproduise le même bogue. Ainsi, la diversité des machines et des compilateurs donnera une bonne assurance sur le résultat produit : il ne sera pas issu d'un calcul bogué.

Reste à être sûr du programme source. N'importe qui ayant connaissance du langage de programmation est en théorie en mesure de s'assurer que le programme est correct vis-à-vis de sa spécification. Mais étant donné la pénibilité de cette tâche, on peut mettre en doute l'efficacité d'une telle démarche. En fait, on peut reproduire le même raisonnement que précédemment. Imaginons que le programmeur ait donné, en même temps que le programme, un texte informatisé qui est la preuve de correction de ce programme vis-à-vis d'un certain langage logique. Alors la diversité des vérificateurs automatiques de preuves vis-à-vis de cette logique garantira la validité de cette preuve (modulo la cohérence de la logique).

On peut trouver des logiques et des langages de programmation cohérents. Dès lors, même si les vérificateurs de preuves, les compilateurs, les ordinateurs ne sont pas parfaits, tout en restant d'une qualité raisonnable, leur diversité compensera leurs défauts respectifs. Sous cet angle, il ne semble pas hors de portée de pouvoir un jour considérer les résultats de calculs informatiques comme scientifiquement valides.

La difficulté pour cela est en fait de réussir à créer des langages de programmation et des langages logiques, suffisament expressifs pour permettre d'écrire “facilement” des programmes efficaces et leurs preuves, mais suffisament simples pour que l'on puisse établir “facilement” leur cohérence. Remarquons aussi la nécessité d'avoir des “objets preuves” que l'on puisse manipuler comme une donnée informatique.

1.3 Présentation du calcul formel

Nous renvoyons par exemple à [Dav93], [Ged93] et [Zip93] pour une introduction au calcul formel. La présentation est à l'usage des non-spécialistes du domaine et paraîtra sans doute approximative aux praticiens du calcul formel. Il s'agit ici de donner simplement les éléments pour une compréhension intuitive des exemples et de la problématique.

1.3.1 Les polynômes

Nous allons maintenant rappeler les notions fondamentales d'algèbre nécessaires à la compréhension des représentations des polynômes. En effet, l'utilisation des polynômes est fondamentale en calcul formel, car beaucoup d'opérations s'expriment comme des opérations sur les polynômes modulo substitution. Par exemple, le calcul

$$(\cos a - \sin b)(\cos a + \sin b) = \cos^2 a - \sin^2 b$$

correspond au calcul polynômial

$$(x - y)(x + y) = x^2 - y^2$$

dans lequel $\cos a$ à été substitué à x et $\sin b$ à y.

1.3.1.a Anneaux et algèbres

Donnons maintenant les quelques notions de base qui nous serviront d'exemple au cours de cette thèse, car elles forment la base de la hiérarchie des bibliothèques de calcul formel (voir la hiérarchie de la bibliothèque FOC, page 27).

- Un *setoïde* est un type muni d'une relation d'équivalence. Cette terminologie ne vient en fait pas du calcul formel, mais de la théorie des types. Un setoïde est en quelque sorte un ensemble “représentable” sur un ordinateur. La notion de type dépend de la logique que l'on utilise. Intuitivement, un type désigne un ensemble de données sur lesquelles on peut appliquer les mêmes traitements informatiques. L'appartenance d'une donnée à un type doit être décidable. Aussi, les types sont souvent des *“structures syntaxiques librement engendrées”*, comme les listes, les arbres binaires, etc. Donner une relation d'équivalence, revient en quelque sorte à quotienter ces structures libres. On abordera plus largement cette notion de représentation des données dans la section 1.3.2.a.
- Un *semigroupe* est un setoïde muni d'une loi de composition interne associative, compatible avec la relation d'équivalence. Lorsque la loi est commutative, on dit que le semigroupe est *commutatif*.
- Un *monoïde* est un semigroupe muni d'un élément neutre.

- Un *groupe* est un monoïde muni d'une fonction d'inversion (compatible avec la relation d'équivalence).
- Un *anneau* est un setoïde muni de deux lois internes, traditionnellement appelées addition et multiplication. C'est un groupe commutatif vis-à-vis de l'addition et un semigroupe vis-à-vis de la multiplication. La multiplication doit de plus être distributive par rapport à l'addition. Un anneau est dit *commutatif* lorsque son semigroupe multiplicatif l'est. Un anneau est dit *unitaire* lorsque son semigroupe multiplicatif est un monoïde. En accord avec la terminologie anglo-saxonne, les anneaux considérés seront unitaires. De plus, ils seront supposés commutatifs, sauf indication contraire.
- Un *module* est un groupe commutatif G (noté additivement) muni d'une loi de composition externe sur un anneau A. Cette loi, appelée multiplication externe, est associative vis-à-vis de la multiplication de A, distributive par rapport à l'addition de G. L'unité (élément neutre multiplicatif) de A est neutre pour cette multiplication externe, et l'addition de A est distributive par rapport à la multiplication externe, et l'addition de G. Un *espace vectoriel* est un module, dont la multiplication externe est sur un corps.
- Soient deux structures algébriques A et B. Un *morphisme* de A dans B est une fonction de A dans B qui est compatible avec les opérations de A et B. Par exemple, si A est un groupe additif et B un groupe multiplicatif, $f : A \to B$ est un morphisme si $f(a_1+a_2) = f(a_1) \times f(a_2)$. Soient A et B deux anneaux. L'anneau B est une A-*algèbre* s'il existe un morphisme h_B d'anneaux de A dans B. L'anneau B peut alors être considéré comme un A-module dont la multiplication externe est l'opération $(x : A; y : B) \mapsto (h_B(x) \times y)$. Par exemple, une des caractérisations de $\mathbb{Z}$, l'ensemble des entiers, est que pour tout anneau A, il existe un unique morphisme d'anneaux de $\mathbb{Z}$ dans A : c'est un anneau initial. Ainsi, tout anneau A est une $\mathbb{Z}$-algèbre.
- Si A est un anneau, et E un ensemble d'indéterminées, l'anneau de polynômes sur A d'ensemble d'indéterminées E, noté $A[E]$, est l'algèbre sur A librement engendrée par ces indéterminées. Autrement dit, les anneaux de polynômes sur A sont des A-algèbres libres.
  Comme les anneaux de polynômes sur A sont des anneaux, on peut construire des anneaux de polynômes sur les anneaux de polynômes sur A. Mais ces derniers sont en fait isomorphes à des anneaux de polynômes sur A, grâce aux propriétés universelles des anneaux de polynômes. Par exemple, $(A[X])[Y]$ est isomorphe à $A[X, Y]$.

1.3.1.b Ordres sur les polynômes

Pour définir des algorithmes sur les polynômes, on utilise très souvent un ordre, qui permet d'optimiser ces algorithmes. Nous allons définir les ordres les plus couramment utilisés.

Un *monôme*[3] est un produit d'un élément de A (appelé son coefficient) et d'éléments de E. Pour structurer les représentations des polynômes, il est commode d'ordonner les monômes avec un ordre bien fondé. Pour cela, on commence par ordonner les éléments de E, puis on ordonne les degrés en considérant les puissances de chaque indéterminée. L'ordre total sur les degrés induit un ordre seulement partiel sur les monômes, car l'ensemble de leurs coefficients n'est pas toujours muni d'un ordre total. Un polynôme s'écrit comme un ensemble fini (ou une somme) strictement ordonné de monômes. L'égalité de deux polynômes se ramène à l'égalité (sur A) des coefficients de monômes de même degré. On appelle *monôme dominant* d'un polynôme le monôme de coefficient non nul de plus haut degré. Le *reductum* d'un polynôme est ce polynôme privé de son monôme dominant.

Étudions quelques ordres classiques à l'aide du polynôme $P = (x+y)^2(z^2+x+z)$ en ordonnant

[3] Ma définition de *monôme* provient de la terminologie de [Zip93]. Le mot *"terme"* apparaît aussi dans la littérature mais je préfère l'éviter dans ce contexte pour le réserver à la logique.

les indéterminées x, y et z de la façon suivante : $x > y > z$:

- *Ordre lexicographique* Le monôme le plus grand est celui dont l'indéterminée maximale a la puissance la plus élevée ou, en cas d'égalité, la deuxième indéterminée maximale a la puissance la plus élevée, ainsi de suite.

  P est alors ordonné comme suit :

  $$x^3 + 2x^2y + x^2z^2 + x^2z + xy^2 + 2xyz^2 + 2xyz + y^2z^2 + y^2z$$

  Cet ordre permet de regarder un polynôme multivarié comme un polynôme en l'indéterminée principale, ici x. On passe donc facilement à une écriture récursive des polynômes, ici en utilisant l'isomorphisme de $\mathbb{Z}[x, y, z]$ et de $((\mathbb{Z}[z])[y])[x]$. L'écriture récursive de P a la forme suivante :

  $$x^3 + x^2(2y + z^2 + z) + x(y^2 + y(2z^2 + 2z)) + y^2(z^2 + z)$$

  Les représentations récursives sont commodes dès que l'on dispose d'un algorithme traitant de façon suffisamment générale le cas à une indéterminée sur l'anneau de base. Il suffit alors de l'appliquer récursivement pour traiter le cas d'un nombre quelconque d'indéterminées.
- *Ordre sur le degré total du polynôme puis ordre lexicographique* Le degré total d'un monôme est la somme des puissances de ses indéterminées. Pour cet ordre, on commence par comparer les degrés totaux, et en cas d'égalité, on applique l'ordre lexicographique. On ordonne les monômes de P comme suit :

  $$x^2z^2 + 2xyz^2 + y^2z^2 + x^3 + 2x^2y + x^2z + xy^2 + 2xyz + y^2z$$

- *Ordre sur le degré total du polynôme puis ordre lexicographique inverse* Ici, on commence par comparer les degrés totaux, et en cas d'égalité, on applique l'inverse de l'ordre lexicographique. Ce qui donne pour P :

  $$y^2z^2 + 2xyz^2 + x^2z^2 + y^2z + 2xyz + xy^2 + x^2z + 2x^2y + x^3$$

  Cet ordre ne correspond pas à une simple inversion de l'ordre sur les indéterminées. En effet, si on inverse l'ordre sur les indéterminées $x < y < z$, on obtient pour ce dernier ordre :

  $$x^2z^2 + 2xyz^2 + y^2z^2 + x^3 + 2x^2y + xy^2 + x^2z + 2xyz + y^2z$$

  À degré total constant, on regroupe donc ici les termes ordonnés suivant la plus petite puissance de z possible. On retrouve sur cet exemple les monômes dans l'ordre obtenu lorsqu'on développe le produit $(x + y)^2(z^2 + x + z)$ de façon habituelle.

Ces ordres sont les plus utilisés en pratique. Il existe encore d'autres ordres, comme l'ordre lexicographique inverse : il correspond dans le cas à une indéterminée à l'ordre selon les puissances croissantes. Cet ordre n'est pas commode pour la division ou le calcul de pgcd, mais peut s'avérer utile pour des calculs de bases de Gröbner ou bien certains calculs sur des réels algébriques utilisant des séries de Puiseux.

1.3.2 Représentation des données

1.3.2.a Notion de représentation

Pour manipuler des objets mathématiques, on doit les représenter, c'est-à-dire les "coder" dans des structures de données, sur lesquelles on sait calculer. L'efficacité des algorithmes repose en partie sur le choix d'une bonne représentation des données, c'est-à-dire adaptée au problème posé. Toutefois, si une bonne représentation permet d'accélérer certains algorithmes, les changements de représentation trop fréquents peuvent être coûteux. Il convient donc de trouver une représentation adaptée à la plupart des opérations et de s'y tenir la plupart du temps. On ne changera la représentation que si l'opération est particulièrement coûteuse.

Soient E et R des structures mathématiques. R est une **représentation** de E s'il existe un morphisme surjectif de R sur E. Si ce morphisme est de plus injectif, alors R est dite **canonique** : l'égalité de deux éléments de E pourra alors être testée par l'égalité de leurs représentants dans R.

Lorsque E possède une structure de monoïde, *l'égalité à l'élément neutre* joue un rôle particulier. Ainsi, lorsque l'élément neutre n'a qu'un seul représentant, le test d'égalité à l'élément neutre ne demande qu'une seule comparaison. On dit dans ce cas que R est une représentation **normale**. C'est par exemple utile pour tester le deuxième argument d'une division. De plus, si E est un groupe, on peut décider de l'égalité de deux éléments de E, en testant l'égalité à l'élément neutre de la différence de leurs représentants : on a ainsi une méthode générique au niveau des groupes pour tester l'égalité de 2 éléments, qui ne dépend pas de la représentation R.

1.3.2.b Représentation des polynômes

Représentations creuses ou distribuées Parmi les représentations ordonnées des polynômes, on distingue les représentations *creuses* qui ne contiennent que des monômes de coefficients non nuls et les représentations *denses*, qui contiennent au moins tous les monômes inférieurs au monôme dominant. Si les représentations des cœfficients et des degrés sont canoniques, alors les représentations creuses (ordonnées) de polynômes sont canoniques.

Pour bon nombre d'algorithmes (par exemple la division et le pgcd) utilisés en calcul formel, il semble essentiel d'accéder à un coût très faible au degré des polynômes. De ce fait une représentation où seuls les monômes non nuls sont indiqués est meilleure qu'une représentation pleine, qui peut contenir des monômes de coefficients nuls de plus haut degré que le monôme dominant. Ces algorithmes s'appliquent souvent récursivement au monôme dominant et au reductum. Les listes ordonnées par degré décroissant, qui permettent de donner ces deux parties fondamentales en temps constant, sont donc particulièrement adaptées.

Représentations denses Contrairement aux représentations creuses, les représentations denses ne sont pas forcément canoniques : on peut avoir des monômes de coefficients nuls de plus haut degré que le monôme dominant. En fait, les représentations denses ne peuvent être intéressantes que dans le cas des polynômes en une indéterminée. Par exemple, une représentation dense de $a^5b^5c^5d^5e^5$ contient 7776 termes. En général, les polynômes multivariés manipulés dans les applications du calcul formel sont creux.

Mais même dans le cas univarié, les représentations denses peuvent être coûteuses. Il faudra par exemple un million de multiplications pour vérifier que $(x^{1000}+1)(x^{1000}-1) = x^{2000}-1$, alors qu'avec une méthode creuse, 4 multiplications suffisent.

Les représentations denses, si elles utilisent des tableaux, permettent d'accéder en temps constant aux coefficients du polynôme : on n'est pas obligé de parcourir tout le polynôme, pour trouver le coefficient d'un monôme donné. Un calcul de base de Gröbner peut tirer profit d'une représentation dense.

Représentations récursives Comme on l'a suggéré lors de la définition de l'ordre lexicographique, les polynômes peuvent être représentés de manière récursive sur le nombre d'indéterminées.

Cette représentation récursive peut elle même généraliser une représentation donnée des polynômes (creuse, dense, à plusieurs variables, etc).

Plus formellement, on appelle *représentation de polynômes* un morphisme d'anneau P qui prend un anneau X et construit un anneau de polynômes sur X. Soit A un anneau, soit S un ensemble d'indéterminées muni d'un ordre et P une représentation de polynômes. L'ensemble des polynômes récursifs est alors donné par $R(A,S,P) = \cup_{s\in S} R_s$ où R_s est défini par :

$$R_s = A \cup (\cup_{s'<s} P(R_{s'}))$$

Cette représentation permet, lorsque cela a un sens, d'étendre facilement les algorithmes des polynômes en un nombre fixe de variables, à ceux en un nombre quelconque de variables. En fait, la fonction "d'extension" est même générique : elle ne dépend pas de l'opération que l'on souhaite étendre. On montrera donc une seule fois que cette fonction conserve les propriétés des opérations qu'elle étend : associativité, commutativité, etc. Par exemple, cette méthode est utilisée pour passer du calcul générique du pgcd sur des polynômes univariés au calcul générique du pgcd sur des polynômes multivariés.

1.3.3 Enjeux et conséquences des problèmes de complexité

On attend en général des programmes qu'ils retournent un résultat "juste", mais aussi qu'ils retournent ce résultat en un temps "raisonnable". La maîtrise de la complexité est en effet un des principaux enjeux (et savoir-faire) de l'algorithmique. Ce souci doit donc se refléter aussi dans la validation ou la certification des programmes. Cependant en calcul formel, déterminer la complexité d'un programme n'est pas une chose aisée. On se contente souvent de donner des bornes de complexité dans le pire cas sur l'algorithme sous-jacent. L'enjeu est alors de s'assurer que non seulement les programmes donnent bien le résultat attendu, mais aussi qu'ils sont une implantation d'un algorithme pour lequel on a trouvé des bornes de complexité raisonnables. Dans ce contexte, les tests de correction deviennent difficiles à élaborer. C'est ce qui explique que la communauté du calcul formel commence à s'intéresser à la certification de programmes.

1.3.3.a Complexité : importance des arithmétiques

Les nombres en calcul formel sont calculés de façon exacte et sont potentiellement non bornés. Un des enjeux des algorithmes de calcul formel, est d'assurer un contrôle de la taille des nombres, en sortie, mais aussi au cours des calculs. Pour illustrer ce problème, on donne souvent l'exemple du calcul du pgcd des polynômes suivants (donné à l'origine par Brown dans [Bro71]) :

$$\begin{array}{rcl} A & = & x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5 \\ B & = & 3x^6 + 5x^4 - 4x^2 - 9x + 21 \end{array}$$

Si on applique l'algorithme d'Euclide, la suite des restes est :

$$\begin{array}{c} \frac{-5}{9}x^4 + \frac{1}{9}x^2 - \frac{1}{3}, \\ \frac{-117}{25}x^2 - 9x + \frac{441}{25}, \\ \frac{233150}{19773}x - \frac{102500}{6591}, \\ \frac{-1288744821}{543589225} \end{array}$$

Les fractions rationnelles données ici sont irréductibles, et en fait, au cours de ces calculs de restes, on a dû faire des calculs de pgcd sur des entiers assez gros.

Afin d'éviter ces coûteux calculs de pgcd sur les coefficients, on calcule le pgcd via des suites de pseudo-restes : on commence par multiplier A par une certaine puissance (la différence des degrés plus 1) du coefficient dominant de B de façon à ce que le reste existe dans $\mathbb{Z}[X]$. Sur l'exemple, cette suite de pseudo-restes est donc :

$$\begin{gathered} -15x^4 + 3x^2 - 9 \\ 15795x^2 + 30375x - 59535 \\ 1254542875143750x - 1654608338437500 \\ 12593338795500743100931141992187500 \end{gathered}$$

Malheureusement, comme le montre l'exemple, la longueur des coefficients d'une telle suite de pseudo-restes croît en général exponentiellement en fonction des degrés des polynômes donnés. On pourrait penser à diviser à chaque étape le nouveau pseudo-reste par le pgcd de ses coefficients, mais ce serait beaucoup trop coûteux.

Collins et Brown (dans [Col67] et [Bro78]) ont trouvé indépendament une suite de pseudo-restes dont la croissance est seulement linéaire (les coefficients des pseudo-restes sont des déterminants de taille linéaire par rapport aux degrés des polynômes donnés), et qui ne nécessite pas de calculs de pgcd sur les coefficients. Cette suite, appelée suite des **sous-résultants** donne sur l'exemple précédent :

$$\begin{gathered} 15x^4 - 3x^2 + 9 \\ 65x^2 + 125x - 245 \\ 9326x - 12300 \\ 260708 \end{gathered}$$

Les algorithmes sur les suites de pseudo-restes et de pgcd sont en fait toujours un sujet de recherche (cf. [Loo83, Rec90, Mor95, Ho96, Duc96, Lom00], ...).

1.3.3.b Retour sur le problème des tests de correction

Élaborer des tests pour les programmes de calcul formel, n'est pas un problème facile, surtout si on souhaite s'assurer qu'un programme est bien l'implantation d'un algorithme donné. Cela s'illustre assez bien à partir de l'exemple précédent. Si l'on connaît deux polynômes P et Q premiers entre eux, on peut tester l'implantation de l'algorithme de calcul de la suite des sous-résultants appliqué au calcul du pgcd. En effet, en multipliant P et Q par un même polynôme D, on obtient deux polynômes dont on connaît le pgcd : D.

Mais ce genre de tests n'est pas tout à fait satisfaisant. Est-ce que l'algorithme sous-jacent est bien celui de la suite des sous-résultants ? On a pu commettre une erreur de programmation telle que le résultat du programme soit bien un pgcd, mais obtenu à partir d'une autre suite de pseudo-restes (qui lui serait égale modulo multiplication par un rationnel), et dont les calculs risquent d'exploser dans le pire des cas. On pourrait envisager de tester les pseudo-restes intermédiaires à la main (ou à l'aide d'un autre programme). Mais cela n'est pas forcément une vérification fiable. En effet, si on programme mal l'algorithme des sous-résultants, on peut écrire malencontreusement une suite qui va être identique sur la majorité des entrées à celle des sous-résultants, mais qui va exploser sur quelques cas.

Sur cet exemple, on comprend alors la démarche de la programmation certifiée : il ne s'agit pas de faire vérifier les articles scientifiques par un système informatique, il s'agit de s'assurer que les programmes implantent effectivement les algorithmes écrits dans ces articles. Dans ce cadre, le développement de programmes comprend au moins deux phases : une phase algorithmique traditionnelle, où on se persuade "sur le papier" qu'un algorithme a un bon comportement, et une phase de programmation certifiée, où on montre à l'aide d'un assistant à la démonstration que le programme est une implantation de l'algorithme.

1.3.3.c Démarche empirique pour mesurer la complexité

Etant donné la difficulté de donner une complexité théorique qui ait un réel intérêt pratique, les programmeurs sont donc amenés à tester l'efficacité de leur programme de façon empirique. Comme pour les tests de correction, la fiabilité des tests d'efficacité est difficile à mesurer. Un des points délicats est là encore le choix des jeux de tests. Les données manipulées en pratique ne sont pas quelconques. C'est aussi en fait une des difficultés pour donner une complexité réaliste. Par exemple, si dans les calculs intermédiaires, les polynômes peuvent être très gros, ils sont obtenus en général à partir d'opérations sur de "petits" polynômes, ce qui leur donne des propriétés particulières.

Pour prendre conscience des conséquences de ce phénomène, on peut considérer un problème hors du calcul formel, mais bien connu de tous les informaticiens : la compression de données. La théorie de l'information nous dit en effet que si on prend un paquet de bit au hasard, il est fort peu probable qu'un algorithme de compression donné puisse le compresser. Ainsi, si on mesure l'efficacité d'un algorithme de compression sur des suites de bits tirés au hasard, on risque fort d'être déçu. Il en est de même avec les algorithmes de sous-résultants. Ceux-ci vont par exemple être optimisés lorsque les sauts de degré entre les suites de pseudo-restes sont strictement supérieur à 1. Mais typiquement, cette optimisation ne sert à rien sur des polynômes tirés au hasard, car les sauts de degré ont une très forte probabilité d'être de 1. En fait, deux polynômes tirés au hasard sont très probablement premiers entre eux : calculer leur pgcd n'a pas d'intérêt.

Si la démarche empirique est nécessaire, elle n'en est donc pas moins difficile. C'est à prendre en compte lorsqu'on développe un environnement de programmation dédié au calcul formel. Il faut faciliter le développement de prototypes afin que les programmeurs puissent s'assurer de l'intérêt d'une implantation. Cela signifie en particulier que l'on veut pouvoir développer rapidement (c'est-à-dire sans avoir à faire une preuve de correction) du code relativement fiable : pour écrire le prototype, et le tester (en utilisant par exemple des programmes qui simulent l'environnement d'exécution de ce code, c'est-à-dire des *lanceurs* et des *bouchons*, cf. [Xan00]). Pour cela, le code doit être aussi facilement réutilisable et modifiable.

1.3.4 Connaissances du calcul formel

Les connaissances du calcul formel, à l'instar de l'algèbre, sont encyclopédiques : le calcul formel s'intéresse à des centaines de structures mathématiques toutes étroitement reliées les unes aux autres. Pour chacune de ces structures, déterminées par les propriétés (associativité, commutativité, ...) de leurs opérations de base, on recense les notions calculatoires en essayant d'avoir les algorithmes les plus efficaces possible. Plus exactement chacune de ces notions (par exemple, le pgcd) peut habiter dans plusieurs de ces structures, et pour chaque structure l'algorithme "le plus efficace connu" peut être différent. Les anneaux factoriels font l'hypothèse d'une opération de pgcd : il n'y a donc pas d'algorithmes génériques de calcul du pgcd à ce niveau. Dans les anneaux euclidiens, on calculera le pgcd via l'algorithme d'Euclide. Dans les anneaux de polynômes sur un anneau factoriel, on calculera le pgcd via, notamment, des suites de sous-résultants[4].

[4]Ainsi les anneaux de polynômes sur un anneau factoriel sont eux-mêmes factoriels

Pour une même notion calculatoire, il y a donc plusieurs algorithmes possibles : *les algorithmes génériques* qui vont être applicables dans une large gamme de structures, et *les algorithmes spécialisés* sur une structure mathématique ou sur une structure de données qui seront moins généraux mais plus efficaces. Un exemple classique de ce phénomène est la multiplication par un scalaire non nul dans les anneaux de polynômes, avec une représentation creuse des polynômes. En effet, en toute généralité, dans cette opération, il faut tester la non-nullité de chaque produit entre cœfficients, pour garantir le caractère creux du polynôme résultat. Mais sur un anneau intègre, ce test est inutile, car on sait alors que le produit de deux cœfficients non nul est lui-même non nul.

De même, dès qu'on est en présence d'une loi associative (structure de semigroupe), on peut définir l'itération de cette loi sur une suite finie d'éléments. Dans le cas de l'addition des polynômes, on aimera spécialiser cet algorithme : au lieu d'itérer l'addition des polynômes, on préfera itérer l'addition des coefficients, car de cette façon on ne créera pas de polynômes intermédiaires, et on fera donc moins travailler le glaneur de cellule (ou garbage collector).

1.3.5 Programmation du calcul formel

Cette présentation du calcul formel nous permet de dresser maintenant les contraintes d'un environnement de programmation de bibliothèques pour le calcul formel. Ces contraintes seront reprises en section 1.4.1.a :

- On doit pouvoir représenter les algorithmes avec le niveau de généricité qui leur convient : certains sont valables pour n'importe quel type de représentation ; d'autres pour une représentation particulière des données.
- Les algorithmes "habitent" dans des structures mathématiques qui définissent leurs conditions d'existence ou les conditions dans lesquelles on peut les appliquer. On doit donc pouvoir représenter ces structures mathématiques dans le système, ainsi que les liens "mathématiques" entre elles.
- Afin d'avoir la plus grande efficacité possible, dans un contexte donné, on souhaite utiliser l'algorithme applicable dans ce contexte le plus spécialisé (a priori, le plus efficace).
- Le système doit offrir la possibilité de programmer rapidement un algorithme, afin par exemple de tester son efficacité avant de l'optimiser. C'est en particulier important lorsqu'on introduit une nouvelle structure de données. Il faut donc pouvoir utiliser facilement les programmes génériques qui peuvent s'appliquer. Le système doit donc permettre une réutilisation du code déjà existant.
- Les algorithmes de calcul formel sont en constant progrès, il est donc important de pouvoir les mettre à jour dans la bibliothèque facilement.

Pour respecter ces contraintes, les langages dédiés au calcul formel comme Axiom utilisent des techniques de programmation orientée objet. En effet, ces techniques permettent le développement modulaire d'une hiérarchie, en facilitant la maintenance et la réutilisation du code.

1.4 Conception du projet FOC

Le "cahier des charges" de cette thèse découle en fait de celui du projet FOC. Outre l'objectif de créer un environnement de programmation certifiée pour le calcul formel, le projet FOC s'est imposé de réutiliser au maximum des outils déjà existants : Ocaml pour la partie programmation et Coq pour la partie preuve. On présentera ces logiciels dans les chapitres 2 et 3 respectivement.

1.4.1 Cahier des charges du système

On donne maintenant le cahier des charges de ce projet. Il est largement motivé par les sections précédentes.

1.4.1.a Cahier des charges du langage de construction des bibliothèques

1. L'architecture de la librairie doit refléter celle des structures algébriques, et favoriser un développement incrémental du général au particulier. Par exemple les groupes doivent être construits à partir des monoides.
2. Pour une même notion, plusieurs degrés d'abstraction sont proposés si nécessaires. Ainsi, on doit pouvoir utiliser les propriétés de l'opération d'un groupe (son type par exemple) sans avoir à considérer ses implantations.
3. Certaines notions doivent pouvoir être définies "de façon générique" pour toute une famille de structures, et redéfinies si besoin est, pour une structure particulière. Ainsi, la définition de la fonction `est_different_de` comme négation de la fonction `est_egal_a` doit pouvoir être partagée par toutes les structures construites sur les setoïdes. Mais celles-ci peuvent la redéfinir.
4. Cette exigence va de pair avec la possibilité de préciser pas à pas une structure algébrique, pour passer, par exemple, d'une vue abstraite de $\mathbb{Z}/2\mathbb{Z}$ à une implantation ZI utilisant les entiers d'une part, ou à une implantation ZB utilisant les booléens d'autre part. Ainsi ZI et ZB peuvent partager les notions "génériques" de la vue abstraite de $\mathbb{Z}/2\mathbb{Z}$.
5. L'implantation d'une structure algébrique doit masquer le type de sa représentation. D'une part, comme on souhaite laisser la possibilité d'utiliser des unités de la librairie en dehors d'un cadre certifié, il s'agit d'éviter des confusions ou de mauvaises utilisations (par exemple violations d'invariants) de cette structure de données, en forçant l'utilisateur à utiliser les opérations prévues pour manipuler la structure de données. D'autre part, il s'agit de favoriser la maintenance de la bibliothèque : on pourra ainsi changer d'implantation sans perturber l'utilisateur.

1.4.1.b Autres contraintes

1. Produire des exécutables d'une efficacité raisonnable. C'est-à-dire comparables à ceux des langages de programmation dédiés au calcul formel, et ayant le même genre de fonctionnalités : Axiom, Aldor, etc.
2. Réutiliser au maximum les développements existants. En particulier le système est développé au-dessus de OCaml et Coq, qui sont respectivement un langage de programmation et un assistant à la démonstration déjà existants.

1.4.2 Réalisation

Le projet FOC doit ultimement construire une interface au-dessus de OCaml et de Coq. Cette interface offrirait en particulier un langage (proche de OCaml et Coq) pour spécifier, programmer et prouver des unités de bibliothèques de calcul formel. Ces unités seraient ensuite traduites vers du code source OCaml pour aboutir à des programmes exécutables. De même, ces unités seraient traduites vers du code source Coq, afin d'en garantir les propriétés.

Autrement dit, le sujet de cette thèse est de proposer un modèle mathématique du système afin de donner un cadre à ces traductions. Cette thèse présente une spécification informelle du langage à construire (chapitre 4), un prototype les réalisant avec du code source OCaml (chapitre 5), et un prototype permettant d'exprimer ces spécifications en Coq (chapitre 6). Enfin, elle expose dans la partie III une axiomatisation de cette spécification dans un langage catégorique.

La spécification informelle du système et le prototype en OCaml sont le résultat d'un travail collectif au sein du projet FOC. Mes travaux "strictement" personnels sont donc le codage en Coq présenté dans le chapitre 6 et l'axiomatisation en partie III.

Parallèlement à ce travail, Renaud Rioboo a développé une bibliothèque avec une centaine d'unités en OCaml, en suivant la méthodologie présentée dans le chapitre 5 (voir figure page 27). Cette figure représente le graphe d'héritage des unités de bibliothèques, c'est-à-dire la construction incrémentale de ces unités. D'une part, cette figure montre la nécessité de cette construction incrémentale : cette bibliothèque serait impossible à construire sans héritage. D'autre part, elle montre la nécessité d'avoir des outils (comme des *obligations de preuves*) pour éviter de se perdre dans cet univers de dépendances complexes.

Virgile Prevosto et Damien Doligez ont commencé à développer une syntaxe concrète pour le langage FOC et à donner une traduction de cette syntaxe vers OCaml et Coq (cf. [Pre00]). La traduction vers OCaml suit la méthodologie présentée en chapitre 5. Par contre, la traduction vers Coq suit un autre codage que celui présenté au chapitre 6. En effet, si ce dernier codage permet de donner des propriétés méta-théoriques du système, il s'est révélé un peu lourd. La traduction vers Coq utilisée dans [Pre00] est donc un codage plus "superficiel".

1.5 Plan

Cette thèse est en trois parties. La partie actuelle présente le cadre dans lequel se situe le projet FOC, c'est-à-dire le cadre de cette thèse. La deuxième partie donne la description informelle du système de bibliothèques que l'on cherche à construire, et en donne des prototypes d'implantation en OCaml et en Coq. Enfin la dernière partie est consacrée à l'axiomatisation mathématique de ce système.

1.5.1 Première partie

Dans cette première partie, les deux chapitres suivants sont consacrés respectivement à la présentation du langage de programmation OCaml et de l'assistant à la démonstration Coq, les deux systèmes sur lesquels le projet FOC veut s'appuyer pour développer son système de bibliothèque. Ces chapitres ont donc les buts suivants : présenter ces systèmes au lecteur qui ne les connaîtrait pas ; présenter les concepts (par exemple, la liaison retardée en OCaml, ou la notion de dépendances opaque ou transparente en Coq) importantes pour FOC ; enfin, présenter des problèmes à résoudre dans le cadre des implantations en OCaml et en Coq (par exemple, les défauts de la programmation orientée objet, ou les problèmes liés à la tactique d'inversion des familles inductives de types).

Ainsi, le chapitre 2 présente la sémantique "intuitive" de OCaml en détail, et en particulier ses traits objets. La section 2.1 donne un aperçu général du langage afin de justifier le choix originel de FOC d'utiliser OCaml dans le cadre de ce projet. La sous-section 2.2.1 est une présentation du langage de base, qui est aussi une initiation destinée aux lecteurs non familiers avec cette sorte de langage. Les sous-sections 2.2.2.a et 2.2.3 présentent respectivement le système de module et le système objet. Ces deux mécanismes de structuration des programmes en unités de bibliothèques sont complémentaires, et FOC essaye de trouver un "style de programmation" mariant aux mieux ces deux mécanismes pour profiter de leurs avantages respectifs. À cet effet, la sous-section 2.2.5 insiste sur les inconvénients des styles "classiques" de programmation orientée objet (cf. [Bou99a]).

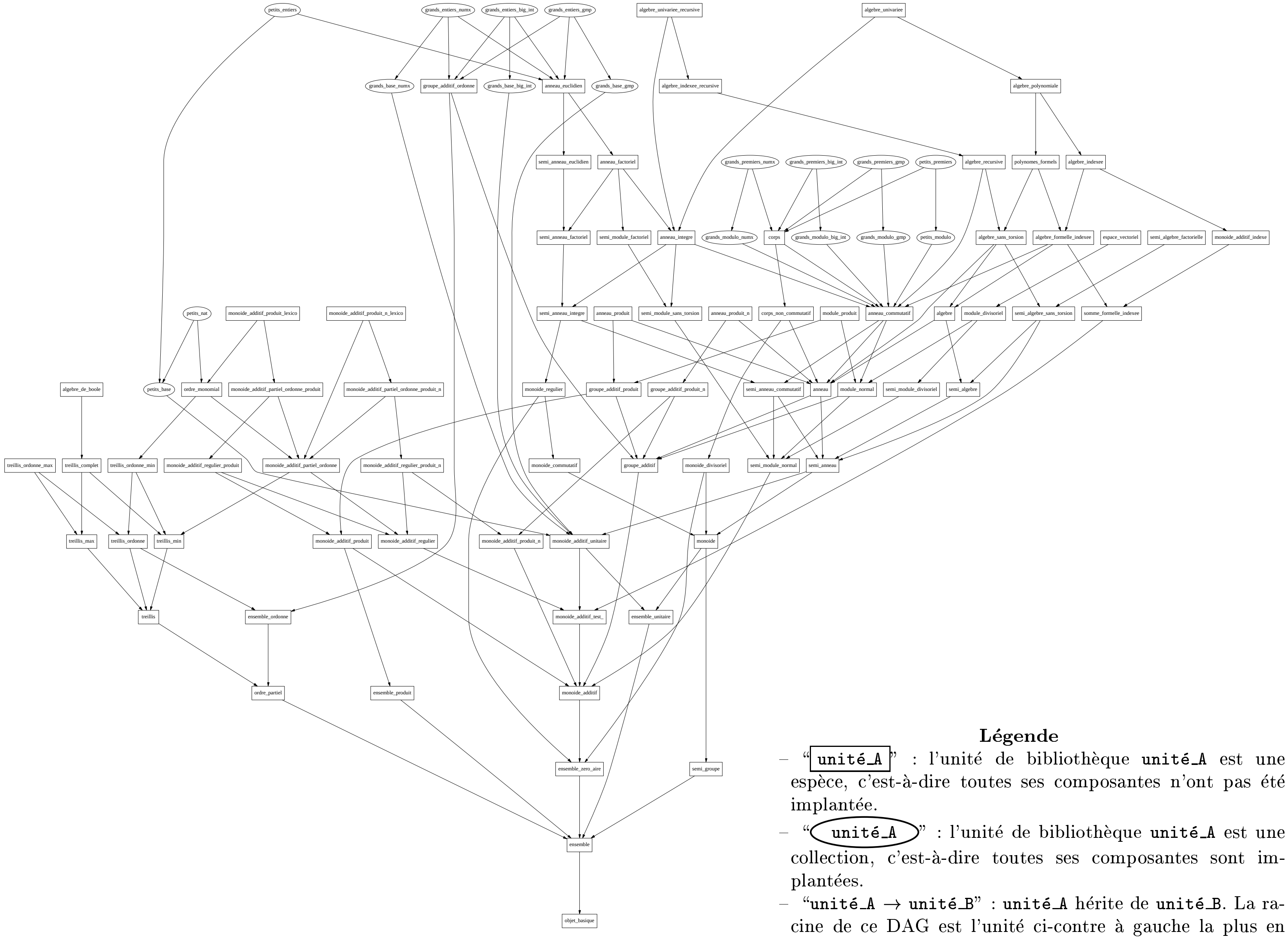


FIG. 1.1 – Bibliothèque FOC en OCaml

Légende

- “$\boxed{\texttt{unité\_A}}$” : l’unité de bibliothèque **unité_A** est une espèce, c’est-à-dire toutes ses composantes n’ont pas été implantée.
- “unité_A” : l’unité de bibliothèque **unité_A** est une collection, c’est-à-dire toutes ses composantes sont implantées.
- “**unité_A** $\rightarrow$ **unité_B**” : **unité_A** hérite de **unité_B**. La racine de ce DAG est l’unité ci-contre à gauche la plus en bas.

Ce chapitre jette donc les bases pour comprendre le style de programmation mis au point dans le cadre du projet FOC, et présenté au chapitre 5.

Le chapitre 3 est le pendant du chapitre précédent, côté Coq. La section 3.1 présente les grands choix de conception sur lesquels repose ce système : elle permet donc d'exposer les motivations de FOC à utiliser ce système. La section 3.2 est une description détaillée de Coq, en insistant sur les aspects nécessaires pour comprendre le codage présenté au chapitre 6.

1.5.2 Deuxième partie

Comme on l'a déjà mentionné, le chapitre 4 donne une description en français du système à construire, indépendamment de tout formalisme mathématique. Ce travail de spécification informelle *en français* est important de notre point de vue, car il permet de prendre du recul par rapport aux diverses formalisations envisageables : cela permet de savoir "vers quoi on va". Un tel document contient par nature beaucoup d'ambiguïtés et d'implicites, que les formalisations doivent lever. Ce document n'est pas figé, il évolue au fur et à mesure de l'avancement de la formalisation, et des questions (ou problèmes) que la formalisation pose à la spécification informelle. Face à un problème (ambiguïté, contradiction, ...) lors de la formalisation, il faut en effet remettre en question le travail réalisé jusqu'alors. La première étape est de déterminer si le problème vient des choix du formalisme ou de la spécification informelle elle-même. Dans ce dernier cas, il faudra alors préciser ou corriger ce cahier des charges. La spécification informelle est donc aussi la mémoire des expériences de formalisation.

Le chapitre 5 est consacré au codage de cette spécification informelle en OCaml : on présente les programmes OCaml qui correspondent à cette spécification. Ces programmes sont définis suivant un "style" mélangeant modules et objets de OCaml. Ce chapitre répond donc aux problèmes posés en sous-section 2.2.5, en tirant parti de l'héritage des objets, mais en conservant un style *"type abstrait de données"*.

Dans le chapitre 6, on retranscrit les structures de cette spécification informelle en Coq. On peut alors à la fois donner des propriétés (méta-théoriques) vérifiées par ces structures, et à la fois les utiliser pour écrire des spécifications FOC en Coq. Les exemples d'utilisation données dans ce chapitre restent cependant très loin d'exemples "réalistes", car ce codage est assez lourd à interpréter en Coq.

1.5.3 Troisième Partie

La description en Coq du chapitre 6 n'est pas suffisante : si les propriétés méta-théoriques montrées dans la chapitre 6 permettent de se convaincre que cette description en Coq correspond à peu près à la spécification informelle, cette description ne permet pas de dire si une autre description correspond à cette spécification informelle. Par exemple, elle ne permet pas de dire si le codage utilisé dans [Pre00] répond bien à la spécification.

La troisième partie propose donc un cadre mathématique plus général et plus abstrait pour donner les propriétés méta-théoriques du système. Ce cadre est inspirée de la notion de catégorie contextuelle, introduite par Cartmell[Car86] et reprise par Streicher dans [Str91] pour donner une sémantique au calcul des constructions (langage sous-jacent à Coq). Il permet d'axiomatiser la spécification informelle, sans syntaxe, uniquement par propriétés : c'est donc une description très déclarative.

Le chapitre 7 donne une description détaillée de chacun des chapitres de cette troisième partie. Il montre aussi comment les notions catégoriques utilisées apparaissent dans les systèmes de types dépendants. Pour cela, il donne en fait une axiomatisation de la notion de systèmes de types dépendants : les D-systèmes. Cette axiomatisation est une étape intermédiaire entre les présentations usuelles des systèmes de types dépendants, et une présentation catégorique à la Cartmell. Elle ne fait pas apparaître la notion de λ-terme, mais se concentre sur les propriétés des contextes de types dépendants et des substitutions. Enfin ce chapitre présente les notions mathématiques (suites finies, arbres et produits fibrés) de bases utilisées dans l'axiomatisation.

Chacun des chapitres suivants présente une "théorie" catégorique particulière. Chacune de ses théories spécialise les précédentes. L'axiomatisation proposée pour la spécification informelle est donnée en section 11.2. On laisse le lecteur se reporter à la section 7.1 pour avoir une description plus détaillée de chacune de ces théories catégoriques.

Le lecteur trouvera en index les références aux pages introduisant les notions principales et certaines notions marginales de cette thèse.

Chapitre 2

Programmation

Le projet FOC a choisi d'utiliser OCaml comme langage de programmation. On va d'abord présenter OCaml en expliquant les raisons de ce choix. On va donner ensuite une description plus détaillée de OCaml, en insistant en particulier sur les objets. On fera ensuite une comparaison des objets de OCaml avec d'autres systèmes objets existants.

2.1 Présentation générale de OCaml

Une des raisons du choix de OCaml est bien sûr qu'il possède un mécanisme objet, dont certaines propriétés semblent bien adaptées pour structurer les bibliothèques de calcul formel. Mais ce n'est pas la seule caractéristique qui justifie le choix d'OCaml. La sémantique rigoureuse qui est à la base de sa conception, ses nombreux traits de programmation (dont les objets ne sont qu'un exemple), le bon compromis qu'il offre entre langage de haut niveau et langage efficace, son interaction avec Coq ou avec le langage C, sont autant de points qui ont été pris en considération.

On donne maintenant une description des principaux traits de ce langage. Chacun des ces traits sera ensuite plus ou moins détaillé dans la section suivante.

Une sémantique rigoureusement étudiée Il n'existe pas à ce jour une sémantique formelle complète de OCaml. Néanmoins chacun des traits du langage a été rigoureusement étudié, en particulier au niveau de l'impact qu'il apportait sur le reste du langage. D'ailleurs, des parties "simplifiées" de ce langage ont été formellement certifiées en Coq (voir [Bou94] ou [Dub99]).

Un langage fonctionnel OCaml est un langage fonctionnel : il manipule les fonctions comme des valeurs du langage. Celles-ci peuvent être utilisées en tant que paramètres d'autres fonctions ou retournées comme résultat d'appel de fonction. Plus généralement, OCaml est un langage de haut niveau qui peut manipuler des structures de données complexes. Il est bien adapté au calcul symbolique, car proche du langage mathématique.

Typage fort, statique et implicite OCaml possède un typage fort : chaque expression possède un unique type principal. Ce type est déterminé *à la compilation* par le système : le typage est statique.

Ainsi, par exemple, la vérification de la compatibilité entre les types des paramètres formels et des paramètres d'appel est effectuée au moment de la compilation. Cela a un double intérêt d'efficacité et de sûreté des programmes. D'une part, c'est efficace, car il n'y a pas besoin de faire ces vérifications à l'exécution, et le typage permet au compilateur de faire des optimisations, en utilisant par exemple la connaissance de la taille des objets qu'il manipule (voir [Ler98]). D'autre part, de nombreuses maladresses ou étourderies sont repérées avant l'exécution du programme. En fait, OCaml possède une propriété importante : toute expression bien typée qui termine, se réduit sur une valeur de même type. Le système ne doit donc jamais se trouver dans un état incohérent (du type "segmentation fault" ou "bus error") : si par malheur, il s'y trouvait, ce ne serait pas un bogue du programme en cours d'exécution, mais un bogue du compilateur.[1]

Il faut ici remarquer que, par défaut[2] le système fait cependant des vérifications dynamiques de manière à garantir le point précédent : il vérifie par exemple le respect des bornes des tableaux et les divisions par zéro. En cas d'erreur, il lève une exception qui interrompt alors le cours de l'exécution (voir ci-dessous).

De plus, le programmeur n'a besoin de donner aucune information de type : le système infère tout seul le type le plus général. Lorsqu'on réutilise des programmes, le typage aide de plus à mieux les comprendre : connaître le type des arguments aide à savoir ce que fait la fonction.

Il faut ici remarquer que l'utilisation des objets impose parfois au programmeur de donner des informations de type. Ces informations peuvent être de 2 natures : obligatoires (le système est incapable de les inférer et rejette l'expression) ou de confort (les types écrits par l'utilisateur sont plus lisibles).

Le typage est polymorphe paramétriquement Lorsqu'une fonction (comme la fonction identité) n'explore pas la totalité de la structure de l'un de ses arguments, le type de celui-ci ne sera pas entièrement déterminé : il est polymorphe. On peut ainsi développer des programmes génériques sur des structures de données différentes, tant qu'elles n'ont pas besoin d'être entièrement connues par les programmes en question. Via l'inférence de type, le programmeur n'a en fait pas à s'en soucier : le système trouve tout seul le type le plus général.

Le glaneur de cellule OCaml est muni d'un glaneur de cellule (*garbage collector*) qui s'occupe tout seul de libérer la mémoire du tas lorsque c'est possible. Ce glaneur de cellule est très efficace, et évite au programmeur de se soucier des problèmes de mémoire. C'est un gage de sécurité. Voir [Dol93] et [Dol95].

Les traits impératifs Entrées-sorties, valeurs mutables (modifiables en place), références (qui jouent le rôle des pointeurs de C, mais sans autoriser "l'arithmétique des pointeurs"), les traits impératifs s'intègrent parfaitement au reste du langage.

Il faut remarquer ici que OCaml est un langage d'appel par valeur strict : tous les arguments des fonctions sont évalués avant d'être appliqués à la fonction. Même en utilisant le style fonctionnel, le programmeur a donc un grand contrôle[3] sur l'ordre d'évaluation des éléments de son programme. De plus, OCaml fait un partage des valeurs structurées : lors d'un passage des arguments à une fonction, seul les pointeurs sur les valeurs structurées sont recopiés ; une valeur ayant des champs modifiables en place pourra donc voir ces champs modifiés après avoir été appliquée à une fonction, comme dans tout langage manipulant de telles données mutables.

[1] Cette seule propriété montre à quel point le typage de OCaml facilite le déboggage.

[2] Il y a une option de compilation pour "débrancher" ces vérifications dynamiques, mais c'est au risque et péril de l'utilisateur

[3] En fait, ce contrôle n'est en théorie pas total, puisque dans une application d'une expression e_1 sur une expression e_2, l'ordre d'évaluation entre e_1 et e_2 est officiellement non spécifié. En particulier, l'ordre d'évaluation des arguments d'une fonction à plusieurs arguments est non spécifié.

Ces stratégies d'évaluation (non négligeables dans la sémantique du langage) jouent un rôle important dans l'efficacité de OCaml. On notera toutefois que si l'utilisateur a besoin d'évaluation paresseuse, il peut toujours la simuler relativement efficacement dans le langage (en fait, cette simulation est déjà fournie par un module de la bibliothèque standard). De même, l'utilisateur a toujours la possibilité de recopier ses valeurs structurées mutables avant de les passer en arguments des fonctions. Le choix de OCaml n'est donc pas trop limitatif.

Création de nouveaux types de données Outre les fonctions, les enregistrements, les tableaux (modifiables en place), les nombres (flottants, entiers machines, entiers exacts, rationnels exacts), les listes, les flots qui sont prédéfinis, et les autres types de base, l'utilisateur peut créer de nouveaux types de données, éventuellement récursifs, à plusieurs constructeurs. Ces types peuvent être eux-mêmes des types paramétrés polymorphes (comme les listes).

Le filtrage Le filtrage permet de raisonner par cas sur le type d'une valeur afin de savoir comment celle-ci est construite. C'est une construction particulièrement concise pour l'écriture des programmes qui évite des "if" en cascade et des fonctions pour déstructurer les données. De plus, elle est compilée très efficacement, en utilisant le partage des valeurs structurées.

Les exceptions Le mécanisme d'exception permet d'interrompre l'exécution normale du programme dans des cas exceptionnels. Il permet de faire par exemple des vérifications dynamiques du type division par zéro, et donc d'écrire des fonctions partielles. Ces exceptions permettent d'encapsuler une valeur lorsqu'elles sont levées. Il est alors possible de raisonner par filtrage sur les exceptions (et les valeurs qu'elles transportent) éventuellement levées au cours de l'exécution d'une expression. Comme pour les types de données, l'utilisateur peut définir ses propres exceptions. Celles-ci sont cependant monomorphes.

Interfaçage avec Coq Coq, le langage d'aide à la preuve que Foc utilise, peut prouver des programmes écrits dans un sous-langage de OCaml. Coq peut en outre extraire d'une preuve un programme OCaml qui réalise la formule ainsi prouvée (voir 3.2.3).

Interfaçage avec C Il est possible d'appeler des fonctions C depuis un programme OCaml, et réciproquement il est possible d'appeler des fonctions OCaml depuis un programme C. En fait c'est surtout le premier point qui intéresse le projet Foc, par exemple pour pouvoir s'interfacer avec la bibliothèque GMP d'arithmétique multi-précision.

Trois modes d'exécution OCaml offre trois modes d'exécution : une boucle d'interaction (pratique pour développer les programmes), la compilation vers du code-octet (bytecode) interprété par une machine vituelle (ce code compilé est donc portable vers n'importe quelle plateforme sur laquelle existe cette machine virtuelle), et enfin la compilation vers du code machine, moins portable, mais plus efficace.

Le système de modules paramétrés Les modules de OCaml sont un moyen de structurer les programmes en unités indépendantes, compilables séparément. Les modules communiquent entre eux via les interfaces de modules qui peuvent masquer la représentation des données. Le mécanisme d'abstraction de type permet ainsi de changer la représentation d'une structure de données sans risque pour les autres modules qui n'ont pas pu accéder à cette représentation. Les modules peuvent être paramétrés par d'autres modules (eux-mêmes paramétrés). Plus qu'un outil puissant de programmation séparée, les modules renforcent le contrôle du programmeur sur le système de type de OCaml.

Les mécanismes objets Les objets de OCaml sont une alternative aux modules, car ces derniers ne sont pas des citoyens de première classe : ils ne peuvent donc pas être passés en argument des fonctions ou retournés par elles. De plus, ils n'offrent pas l'héritage possible sur les objets.

Au contraire, les objets sont des valeurs qui "contiennent" à la fois des données, et les traitements sur ces données. Les données sont stockées dans des *variables d'instance*. Les traitements s'appellent des *méthodes*. Deux objets de même type auront des méthodes de mêmes noms et de mêmes types, mais pas de même implantation. Les types des objets spécifient donc uniquement le nom des méthodes et leur type.

Les objets sont en fait créés à partir de classes. Les *classes* permettent de définir une famille d'objets qui partageront en partie les mêmes méthodes, et les mêmes types de données. Une classe peut hériter de une ou plusieurs autres classes, ce qui signifie intuitivement qu'elle étend toutes ces classes. Au cours de l'héritage, on peut redéfinir des méthodes. Au moment où l'on crée l'objet, c'est la dernière définition de chaque méthode qui est choisie.

Ainsi, le projet Foc s'appuie à la fois sur les objets et les modules pour représenter les structures du calcul formel.

En conclusion, OCaml est donc un langage de programmation puissant et efficace, qui offre un grand confort d'utilisation. En outre, même dans le cadre d'une programmation non certifiée, il est relativement sûr. Comme Coq s'interface avec lui, il est donc idéal pour mélanger des bouts de programmes certifiés et des bouts de programmes qui ne le sont pas. Comme il s'interface avec C, il permet de réutiliser des bibliothèques de calcul formel déjà existantes. De plus, les nombreux traits de programmation qu'il offre laissent beaucoup de choix quant au style de programmation que l'on souhaite adopter.

2.2 Présentation détaillée de Ocaml

Au cours du projet FOC, nous avons été amenés à étudier particulièrement le système de modules et le système objet de OCaml. Comme les présentations de ces systèmes (en particulier le système objet) restent relativement rares, nous tenons à montrer ici le fruit de cette étude. Mais ces systèmes ont tellement d'interconnexions avec le reste du langage, qu'il est difficile de les présenter indépendamment de l'ensemble de OCaml. Pour des raisons de place, on ne fera qu'une présentation succinte des autres aspects de OCaml. Le lecteur qui aimerait en avoir une présentation plus approfondie est invité à se reporter à [Cha00] et [Ler99]. De plus, on ne s'attarde pas sur la syntaxe de OCaml, ni même sur sa sémantique formelle. On cherche en fait plutôt à donner au lecteur une intuition de sa sémantique. Le chapitre 5 s'appuiera en effet sur cette intuition pour justifier du style d'implantation des bibliothèques FOC en OCaml.

La section 2.2.1 présente le langage de base, la section 2.2.2.a présente les modules et la section 2.2.3 présente les objets. Le lecteur ayant une bonne connaissance de CAML, mais connaissant peu les objets peut directement passer à la section 2.2.3.

2.2.1 Langage de base

2.2.1.a Types et valeurs en OCaml

Comme dans tout langage fonctionnel, c'est la notion de calcul qui prime en OCaml. Exécuter un "programme OCaml", c'est en fait *évaluer une expression*, c'est-à-dire "simplifier" cette expression, jusqu'à obtenir une forme irréductible. Parmi les expressions irréductibles du langage, on distingue celles qu'on appelle *valeur*. Les valeurs correspondent à la notion de données informatiques.

Par exemple, si on tape "`2 + 3;;`" dans la boucle interactive, OCaml commence par typer l'expression `2 + 3`, puis exécute cette expression. Il affiche alors son type `int` et sa valeur `5`.

En OCaml, le typage est un mécanisme complètement statique (sans évaluation de l'expression). Toutes les expressions du langage n'ont pas forcément un type : elles seront rejetées par OCaml qui refusera de les évaluer. Le typage a en outre les propriétés suivantes :

- lorsqu'on simplifie une expression bien typée (qui admet un type), le type de l'expression simplifiée est le même que celui de l'expression de départ.
- une expression bien typée est soit "simplifiable" infiniment, soit finit par se simplifier sur une valeur. Autrement dit, le typage rejette les expressions qui ne se simplifieraient pas sur une valeur, et mettraient le système dans un état incohérent.

A l'instar de la vérification de la cohérence des unités de mesure dans les calculs en physique, le typage permet en fait d'éviter un grand nombre d'incohérences et d'erreurs grossières avant l'exécution des programmes. Il ne les évite cependant pas toutes, puisque ce problème est indécidable. Intuitivement, un type désigne un ensemble de valeurs (ou de données) sur lesquelles on peut appliquer les mêmes traitements informatiques.

Les types (ou les valeurs) manipulés par OCaml sont comme dans de nombreux langages de programmation : les entiers, les flottants, les booléens, les listes, les fonctions, les flots, les tableaux, les objets, ... Cette énumération est extensible à l'infini, car l'utilisateur a la possibilité de définir de nouveaux types de données.

Parmi les types qu'on utilisera ici, et qui sont prédéfinis dans OCaml, notons :

- Le type `int` des entiers machines (de taille bornée) : `0`, `1`, `2` ...
- Le type `bool` des booléens : `true` et `false`.
- Le type `string` des chaînes de caractères : `"Renaud"`, `"Damien"`, ...
- Le type `unit` qui n'a qu'une valeur `()`. Ce type est en fait utilisé pour les expressions qui ne produisent pas de valeur intéressante, comme les instructions impératives qui ne font que des effets de bord. Par exemple `print_string`, la fonction pour afficher à l'écran une chaîne de caractères est à valeur dans `unit`. Ainsi l'expression `print_string "bonjour"` a le type `unit` : attention, cette expression n'est pas elle-même une valeur, seule `()` est une valeur.
- Le type `list`. A la différence des types précédents, ce type est paramétré par un autre type. Par exemple `int list` est le type des listes d'entiers : `[0;1;5]`, `[2;7;6;8]`. De même `string list` est le type des listes de chaînes de caractères : `["Thérèse";"Valérie"]`. Remarquons par contre que `["Thérèse";2;3]` n'est pas une expression bien typée. Tous les éléments d'une liste doivent avoir le même type. Il faut noter aussi le cas de la liste vide, notée `[]`, qui a le type `'a list`. Ici `'a` représente un type quelconque. Le symbole apostrophe `'` introduit en fait un identificateur qui est une variable de type. Dans une expression de type, les variables de type sont implicitement quantifiées universellement (ce qui signifie qu'elles sont substituables par n'importe quoi). Un type contenant une telle variable de type est dit *polymorphe*. [4]

[4] On verra page 44 que c'est un peu plus compliqué : il existe des variables qui ne sont pas quantifiées universellement, car instanciables une seule fois. Elles sont notées "`'_a`".

- Les types fonctionnels. Ils sont dénotés comme usuellement en mathématique avec le symbole `->`. Ainsi `int -> int` est le type des fonctions des entiers dans les entiers. Le symbole `->` est associatif à droite. Ainsi, `int -> bool -> string` est égal à `int -> (bool -> string)`. Ce type dénote les fonctions prenant un entier et retournant une fonction des booléens dans les chaînes de caractères : plus simplement, on peut voir cela comme le type des fonctions prenant un entier et un booléen et retournant une chaîne de caractères.
  Les valeurs fonctionnnelles sont introduites à partir du mot-clef `fun` et du symbole `->`[5]. Ainsi `fun x y z -> x*(y+z)` est une valeur fonctionnelle dont les paramètres formels sont `x`, `y` et `z` (des variables liées). Cette valeur a le type `int -> int -> int -> int`, car `*` et `+` sont des opérateurs arithmétiques réservés aux entiers machine.
  Il y a aussi des valeurs fonctionnelles polymorphes comme `fun b x y -> if b then x else y`. Cette valeur a le type `bool -> 'a -> 'a -> 'a` car la condition dans un `if ... then ... else ...` doit être un booléen, et que les deux branches d'exécution possibles doivent avoir le même type. C'est ce type qui sera le type de retour de cette opération conditionnelle.
  Un autre exemple de valeur fonctionnelle est l'application des fonctions elle-même, qui s'écrit `fun f x -> f x`. Cette fonction prend une fonction `f` et un élément `x` et retourne le résultat de l'application de `f` à `x`. Elle a le type `('a -> 'b) -> 'a -> 'b`.
- Les types de n-uplets. À l'instar du symbole fonctionnel `->`, le symbole `*` permet de construire des types de n-uplets. Par exemple, `((1,true),[],(fun x -> x))` est une valeur n-uplets qui a le type `(int * bool) * 'a list * ('b -> 'b)`. Notons que les types `(int*bool) * string`, `int * (bool * string)` et `int * bool * string` sont 3 types isomorphes, mais 2 à 2 distincts. Il en est de même des 2 types `(int * bool) -> string` et `int -> bool -> string`.

Comme on l'a mentionné il y a des opérations spécifiques sur chacun de ces types. On ne les détaillera pas ici.

2.2.1.b Nommer une valeur

En fait, l'évaluation de chaque expression a lieu dans un certain *environnement* : cet environnement associe à chaque identificateur une certaine valeur. Plus précisément il y a deux environnements associés à chaque expression : l'environnement global qui est commun à toutes les expressions, et l'environnement local qui ne sert que pour évaluer cette expression.

Deux constructions permettent de nommer une valeur et de modifier ainsi les environnements. Premièrement, la construction `let` v `=` e qui associe la valeur produite par l'évaluation de l'expression e au nom v dans l'environnement global : c'est une *définition globale*. Et deuxièment, la construction `let` v `=` e_1 `in` e_2 qui associe la valeur produite par l'expression e_1 au nom v dans l'environnement local qui servira à l'évaluation de l'expression e_2 : c'est une *définition locale*. Il faut noter que ces deux constructions ne sont pas elles-mêmes des valeurs (sinon, elles ne "s'exécuteraient" pas). La première construction retourne la valeur produite par e, la deuxième retourne la valeur produite par l'évaluation de e_2 (dans son environnement local).

Notons aussi que si la deuxième construction est une expression du langage, la première construction n'en est pas une. C'est une construction syntaxique à part. Cette restriction syntaxique permet de limiter la possibilité de modifier l'environnement global. On approfondira ce point après les quelques exemples ci-dessous. À ce stade de la présentation, on peut en fait considérer qu'un programme OCaml est une suite d'expressions et de définitions globales.

[5]Il faut donc remarquer ici que le symbole `->` a un sens différent suivant qu'il apparaît dans une expression de type ou une expression du langage.

Ainsi, la définition globale ci-contre associe la valeur `3` au nom `x` dans l'environnement global.

```
let x = 1+2
```

Si on évalue ensuite l'expression ci-contre, on obtient la valeur `12`. Mais l'environnement global n'est pas modifié. De l'extérieur de cette constante `y` n'existe pas.

```
let y = 3*x in y+x
```

À l'exécution, l'expression précédente est en faite équivalente à celle ci-contre, dans laquelle on applique `3*x` à la fonction qui prend `y` et l'ajoute à `x`.

```
(fun y -> y+x) (3*x)
```

Dans une expression, un nom ainsi défini fait toujours référence à sa dernière définition : il masque éventuellement les précédentes définitions du même nom. Ainsi, la définition globale ci-contre s'évalue en `[3;6;3]`.

```
let z =
  let y =
    let x = x+x in x
  in [x;y;x]
```

Ce mécanisme de masquage est aussi valable pour les variables liées des valeurs fonctionnelles. La définition globale ci-contre produit la même valeur que la définition précédente.

```
let z =
 ( (fun y -> [x;y;x])
      ((fun x -> x) (x+x)) )
```

On peut aussi définir des valeurs fonctionnelles. Par exemple, voilà une fonction qui prend un entier `y` et qui teste si `y` est supérieur à `x`.

```
let supx = (fun y -> y > x)
```

Pour alléger la syntaxe, OCaml autorise de placer les arguments avant le signe `=` du `let`. Voilà une définition complétement équivalente à la précédente.

```
let supx y = y > x
```

L'évaluation de `supx 2` rend la valeur `false`, et celle de `supx 4` produit `true`. Si alors, on redéfinit le `x` de l'environnement global, le `x` dans le corps de `supx` ne désigne pas cette nouvelle valeur, mais l'ancienne.

Ainsi, la déclaration ci-contre renvoie la valeur `true`.

```
let x = 5 in supx 4
```

Chaque valeur fonctionnelle porte donc en elle l'environnement dans laquelle elle a été définie : c'est ce qu'on appelle *une fermeture*. La portée des identificateurs en OCaml est donc *lexicale* : indépendamment de l'exécution du code, on sait trouver le *"lieu"* dans le source qui définit à quelle expression font référence les identificateurs. C'est un gage de simplicité et de compréhensibilité du code. On dit aussi que *la liaison des identificateurs est statique*. Pour résumer, cette propriété est garantie par les trois points suivants :

1) Dans une définition locale `let` v `=` e_1 `in` e_2, l'association de v à l'expression e_1 est limitée à e_2.

2) Une définition globale ne peut pas avoir lieu "à l'intérieur d'une expression" : l'environnement global ne peut pas être modifié *au cours* d'un calcul, mais seulement *après* un calcul.

3) Les identificateurs à l'intérieur d'une valeur fonctionnelle sont liés à l'environnement dans lequel cette fonction a été définie, et non pas à l'environnement d'exécution de la fonction.

2.2.1.c Polymorphisme et inférence

Dans les exemples de code qui précèdent, on remarque que le programmeur n'est en général pas obligé de donner d'informations de type : c'est le compilateur OCaml qui infère les types. Cela a pour effet d'allèger particulièrement l'écriture et la lecture du code. L'utilisateur peut néanmoins accéder à cette information. Par exemple, OCaml affiche les types de chaque définition globale. Le typage instaure donc un "dialogue" entre le programmeur et le compilateur qui évite de longues séances de déboggage.

Revenons maintenant un peu sur le polymorphisme. La fonction polymorphe la plus simple est l'identité.

```
let id x = x
```

Pour cette définition, OCaml donne le type : `'a -> 'a`. C'est en fait le type le plus général de cette fonction. Les types `int -> int` ou `bool -> bool` pourraient eux aussi convenir, mais pas le type `int -> bool`. En fait, chaque expression typable de OCaml, a un unique type principal, c'est-à-dire duquel tous les autres types sont instances. C'est ce type principal qui est inféré par le compilateur. Ainsi, sous l'hypothèse de l'environnement global précédent : l'expression `(id x)` a le type `int`, et `(id z)` a le type `int list`.

Le polymorphisme permet d'exprimer certaines contraintes sur les arguments. Reprenons l'exemple de la conditionnelle.

```
let cond b x y = if b then x else y
```

La fonction `cond` a le type `bool -> 'a -> 'a -> 'a`. OCaml impose donc que les deux branches de la conditionnelle soient de même type. En effet, sans cette restriction, on ne saurait pas quel type donner au résultat de cette construction. Ainsi, l'expression `cond 5 4 3` est mal typée : le premier argument devrait être un booléen, et est ici un entier. Cette expression est rejetée par OCaml et ne peut pas être évaluée. De même l'expression `cond true true 1` est rejetée car le deuxième et le troisième argument devraient être de même type, mais ne le sont pas. Une raison pour distinguer ainsi les booléens et les entiers est que le compilateur ne manipule pas ces données de la même façon : par exemple, un booléen tient sur 1 bit, alors qu'un entier machine sera codé en général sur 32 bits. Une autre raison, plus profonde, est que du point de vue du programmeur (du point de vue sémantique), ces types sont différents. Et si le programmeur a mis un entier à la place d'un booléen, c'est probablement qu'il s'est trompé.

Ces deux exemples montrent en quelque sorte le but des types polymorphes : être suffisamment généraux pour être instanciables dans de nombreux contextes, mais être suffisamment contraignants pour rejeter les expressions dépourvues de sens, ou dangereuses. Précisons un peu les limites de ce polymorphisme. On a dit précédemment que les variables de type étaient implicitement quantifiées universellement. Il faut préciser maintenant que cette quantification a lieu "en tête" des types : les quantifications à l'intérieur des types ne sont pas possibles. Cette restriction est dûe au fait qu'on souhaite garder les informations de types implicites : autoriser les quantificateurs partout rendrait l'inférence de type indécidable (cf. [Pfe93]).

Dans la pratique, cela signifie par exemple que les arguments formels d'une fonction ne peuvent pas eux-même être polymorphes (on dit qu'ils sont monomorphes) : seule cette dernière peut l'être. Reprenons l'exemple de `id` et `cond`. L'expression `(cond (id true) (id 5) (id 6))` est bien typée (et a le type `int`).

Mais l'expression suivante ne l'est pas. En effet, pour que `bad_function` soit bien typée, il faudrait que `f` soit polymorphe, ce qui est interdit.

```
let bad_function f g =
  (g (f true) (f 5) (f 6))
```

Notons maintenant que la généralisation des variables de type (c'est-à-dire la mise sous forme polymorphe) n'a lieu en fait qu'au moment des définitions (c'est-à-dire dans un `let`). Par exemple si on applique à une autre fonction la valeur fonctionelle `(fun x -> x)` cette dernière sera monomorphe.

Ainsi l'expression ci-contre provoquera une erreur de typage.

```
(fun f -> (f true, f 0)) (fun x->x)
```

Par contre, celle-ci a le type `bool * int`

```
let f = fun x -> x in (f true, f 0)
```

2.2.1.d Types récursifs

En OCaml, l'utilisateur peut introduire de nouveaux types de données. Il donne pour cela les constructeurs qui permettent de fabriquer des valeurs de ce type. Il faut bien distinguer les constructeurs et les fonctions : l'expression qui applique une fonction à un argument n'est pas une valeur, par contre "l'application" d'un constructeur à ses arguments est une valeur. Les constructeurs ne sont d'ailleurs eux-mêmes pas des expressions du langage s'ils ne sont pas appliqués à tous leurs arguments. Pour bien marquer cette différence, OCaml impose que les identificateurs commencent par une majuscule, alors que les identificateurs vus jusqu'à présent doivent commencer par une minuscule !

Ces types peuvent être récursifs. Par exemple, on peut définir le type des entiers naturels à la Peano, avec le constructeur constant `O` (comme zéro), et le constructeur `S` (comme successeur) qui prend un entier naturel en argument.

Une telle définition de type est introduite par le mot-clef `type`, les constructeurs sont précédés du symbole `|`, et le type des éventuels arguments des constructeurs est précédé de `of`.

```
type nat = | O | S of nat
```

Ce type `nat` ci-contre est habité par les valeurs suivantes : `O`, `(S O)`, `(S (S O))`, `(S (S (S O)))`, etc.

Un tel type peut être paramétré par un autre type : on dira qu'il est polymorphe. Par exemple, le type `list`, que nous avons déjà évoqué, est construit en suivant ce procédé. Construisons maintenant le type des arbres binaires.

Ce type `bin_tree` est paramétré par le type de ses éléments `'a`. Il a deux constructeurs `Leaf`, qui construit une feuille, et `Node` qui construit un nœud. Le constructeur `Node` prend 3 arguments : un élément de type `'a`, un arbre fils gauche et un arbre fils droit.

```
type 'a bin_tree =
| Leaf
| Node of 'a*('a bin_tree)*('a bin_tree)
```

Une valeur du type `int bin_tree` est par exemple `Node(1, Node(2, Leaf, Leaf), Leaf)`. Notons que `Leaf` est une valeur qui a le type polymorphe `'a bin_tree`.

Nous allons montrer comment utiliser un tel type pour représenter une structure de données bien connue : les tas. Étant donné un ordre total sur les éléments, un tas est un arbre binaire, tel que pour tout nœud, les éléments des fils de ce nœud sont supérieurs à l'élément de ce nœud. Les tas ont une propriété intéressante : chaque fils d'un nœud est lui-même est un tas. Les tas sont donc des arbres binaires bien particuliers, qui permettent de coder efficacement la notion de liste triée. Le minimum de la liste est en effet trouvé en temps constant, l'insertion d'un élément et la suppression du minimum sont logarithmiques en fonction du nombre d'éléments dans le pire cas.[6]

[6] Pour des raisons pédagogiques, les tas que nous implantons ici ne sont pas optimaux : ils peuvent être deséquilibrés.

Commençons par écrire la fonction qui insère un élément dans un tas. Cette fonction sera récursive : on compare l'élément à introduire au premier du tas.

Le minimum de ces deux éléments deviendra le sommet du tas, et le maximum sera récursivement inséré dans un des fils du tas. Pour introduire une fonction récursive en OCaml, on utilise le mot-clef `let rec` au lieu de `let`. Cette fonction `heap_insert` est donnée ci-contre. Elle est paramétrée par un ordre `o` dont le type est `'a -> 'a -> bool`, un élément `x` de type `'a` et un tas `h` de type `'a bin_tree`.

```
let rec heap_insert o x h =
  match h with
   | Node (y,h1,h2) ->
      if (o x y)
      then
        Node(x,(heap_insert o y h2),h1)
      else
        Node(y,(heap_insert o x h2),h1)
   | Leaf -> Node(x,h,h)
```

La construction `match` e `with ..` est un filtrage qui s'applique sur le résultat v de l'évaluation de e. C'est une sorte de traitement du résultat par cas : chacun des cas est ici précédé du symbole `|`. Les cas sont donc de la forme `|` f `->` e, où f est le filtre et e une expression. Les identificateurs autres que les constructeurs qui apparaissent dans f sont des variables liées. Tester ce filtre revient à essayer d'instancier ces variables de manière à retrouver v. Lors de l'exécution d'un filtrage, chacun des cas est testé successivement, jusqu'au premier qui filtre v. L'expression associée à ce cas est alors exécutée, en substituant les variables du filtre par les termes qui leur correspondent dans v. OCaml affiche à la compilation un message de mise en garde lorsqu'un filtrage n'est pas exhaustif ; l'utilisateur qui ignore ce message prend le risque de voir son programme interrompu par une exception à l'exécution, lorsqu'aucun des filtres ne correspond à v. Le filtrage est donc un moyen concis et efficace de raisonner sur la forme d'une valeur d'un certain type.

La fonction `heap_insert` a le type `('a -> 'a -> bool) -> 'a -> 'a bin_tree -> 'a bin_tree`. Remarquons que lors de la construction du nouveau tas, on permute le fils gauche et le fils droit, de manière à ne pas trop deséquilibrer l'arbre lors d'une succession d'insertions.

On va maintenant se servir de cette fonction pour construire un tas. Plutôt que d'insérer un à un les éléments dans le tas, ce qui serait fort pénible, on va construire une fonction qui transforme une liste en tas. On va pour cela utiliser une fonction prédéfinie appelée `List.fold_left` qui a le type `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` et telle que Les expressions de la forme `(List.fold_left f a [b1; ...; bn])` s'évaluent en `(f (... (f (f a b1) b2) ...) bn)`. Intuitivement, `List.fold_left` est une fonction d'itération générique des listes.[7] Pour notre problème, la fonction `f` est une fonction qui insère un élément dans un tas. Le type `'b` est celui des éléments du tas, et le type `'a` est celui des tas soit `'b bin_tree`. L'argument `a` sert à initialiser le processus : c'est le tas vide. Notre fonction `list2heap` qui convertit des listes en tas, prend donc un argument un ordre `o` et une liste `l` et retourne un tas :

```
let list2heap o l =
  List.fold_left (fun h x -> (heap_insert o x h)) Leaf l
```

Elle est de type `('a->'a->bool) -> 'a list -> 'a bin_tree`. On peut maintenant l'essayer en prenant l'ordre croissant sur les entiers machines :

```
let ex_heap = list2heap (<) [10;1;15;12;4;5;3;9;11]
```

OCaml donne pour `ex_heap` la valeur suivante, de type `int bin_tree` :

Ainsi nos opérations d'insertion et de suppression ne sont pas logarithmiques dans le pire des cas, mais linéaires. Consulter [Oka98] et [Mot99] pour une implantation logarithmique dans le pire des cas.

[7] Mathématiquement, cette fonction traduit que la structure de listes est une structure de monoïde libre.

```
Node
 (1,
  Node (3, Node (11, Node (15, Leaf, Leaf), Leaf), Node (4, Leaf, Leaf)),
  Node (5, Node (9, Node (12, Leaf, Leaf), Leaf), Node (10, Leaf, Leaf)))
```

On aimerait maintenant écrire la fonction `heap2list` qui transforme un tas en une liste triée. Comme pour `list2heap`, on veut en fait passer par l'intermédiaire d'une fonction `heap_foldl` qui se comporte comme `List.fold_left`. Ainsi, `heap_foldl` prend une fonction `f`, un élément `a` et un tas `h`. Lorsque `h` est une feuille, elle renvoie `a`. Sinon elle applique `f` d'une part sur le premier élément de `h`, et d'autre part sur le résultat obtenu en s'appliquant récursivement sur le "reste" de `h`.

Pour implémenter cette idée de "reste" de `h`, on utilise en fait la fusion de deux tas : le "reste" de `h` est la fusion des fils de `h`. La fusion de deux tas est donnée par la fonction récursive `heap_merge` ci-contre. La fusion de deux tas non triviaux est un nœud, dont l'élément est le minimum de ces deux tas, dont un fils est la fusion des deux fils du tas qui portait ce minimum, et dont l'autre fils est le tas qui ne portait pas ce minimum. Dans l'expression de filtrage ci-contre, le symbole `_` est le filtre universel : il filtre n'importe quel terme.

```
let rec heap_merge o h1 h2 =
  match (h1,h2) with
   | (Node(x,ha,hb),Node(y,hc,hd)) ->
      if (o x y)
      then
        Node(x,(heap_merge o ha hb),h2)
      else
        Node(y,h1,(heap_merge o hc hd))
   | (Leaf, _) -> h2
   | _ -> h1
```

Cette fonction `heap_merge` est de type `('a -> 'a -> bool) -> 'a bin_tree -> 'a bin_tree -> 'a bin_tree`. On peut maintenant définir `heap_foldl` :

```
let rec heap_foldl o f a h =
  match h with
   | Node (y, h1, h2) -> (f (heap_foldl o f a (heap_merge o h1 h2)) y)
   | Leaf -> a
```

On vérifie que OCaml donne le type attendu : `('a -> 'a -> bool) -> ('b -> 'a -> 'b) -> 'b -> 'a bin_tree -> 'b`. Finalement, on définit `heap2list` d'une manière très similaire à `list2heap`.

On utilise ici le constructeur de liste, qui a une syntaxe spéciale en OCaml : c'est un symbole infixe "`::`" qui prend à gauche un élément, et à droite une liste.

```
let heap2list o h =
 heap_foldl o (fun l x -> x::l) [] h
```

Cette fonction a bien le type `('a -> 'a -> bool) -> 'a bin_tree -> 'a list`

On va maintenant la tester sur `ex_heap`, le tas calculé précédemment.

```
let ex_list = heap2list (<) ex_heap
```

OCaml donne pour `ex_list` la valeur suivante : `[1;3;4;5;9;10;11;12;15]`. C'est bien la liste triée correspondant à celle donnée pour construire `ex_heap`. On peut donc ainsi faire une fonction de tri sur les listes :

```
let heap_sort o l = (heap2list o (list2heap o l))
```

Grâce au polymorphisme de OCaml, cette fonction de type `('a -> 'a -> bool) -> 'a list -> 'a list` ne s'applique pas que sur l'ordre croissant des entiers. Par exemple, on peut l'appliquer sur l'ordre décroissant des chaînes de caractères :

```
heap_sort (>) ["Thérèse"; "Renaud"; "Valérie"; "Damien"; "Virgile"; "Daniel"; "Sylvain"]
```

Cette expression produit la liste :

```
["Virgile"; "Valérie"; "Thérèse"; "Sylvain"; "Renaud"; "Daniel"; "Damien"]
```

2.2.1.e Les exceptions

Les fonctions vues jusqu'à maintenant étaient toujours totales. Quels que soient les arguments qu'on leur fournissait, elles retournaient toujours une valeur. Mais on a envie aussi de pouvoir coder des fonctions partielles : par exemple, une fonction qui prend un tas et retourne son premier élément. Cette fonction est partielle, car si le tas est une feuille, il n'a pas de premier élément.

OCaml introduit pour cela un mécanisme appelé *exception* (voir [Ler00b]). Lorsque on donne à une fonction f, un argument qui n'est pas dans son domaine de définition, celle-ci peut *lever une exception*. L'exécution du programme est ainsi interrompue en affichant une erreur. Si f était en fait invoquée par une autre fonction g, alors g peut *rattraper cette exception*. Ainsi g peut s'apercevoir que f a été mal appliquée, et réagir en conséquence. Si g ne la rattrape pas, cette exception continue de se propager, et peut être rattrapée par une fonction invoquant g, et ainsi de suite...

Les exceptions sont en fait des constructeurs d'un type particulier prédéfini extensible, nommé `exn`. Comme tous les constructeurs, elle commence par une majuscule. L'utilisateur peut définir de nouvelles exceptions, grâce au mot-clef `exception`.

Ainsi, l'expression ci-contre déclare l'exception sans argument `Empty`.

```
exception Empty
```

Comme pour les autres constructeurs, on peut leur donner des arguments (monomorphes), mais on ne détaillera pas ça ici.

On peut alors lever une exception grâce à la fonction `raise`. Cette fonction a le type `exn -> 'a`. On remarque en particulier le type de retour de cette fonction, qui est quelconque : la valeur de retour de cette fonction a donc n'importe quel type. Mais cette valeur est surtout formelle, puisque l'application de `raise` sur une exception provoque l'interruption du programme. Ici, il faut distinguer deux valeurs : celle de l'exception, qui est de type `exn`, et celle provoquée par la levée de l'exception, qui est du type de la valeur normalement attendue.

On peut donc maintenant écrire la fonction `heap_head` qui retourne le premier élément d'un tas. Elle lève l'exception `Empty` lorsque son argument est une feuille.

```
let heap_head h =
  match h with
    | Node (x,h1,h2) -> x
    | Leaf -> raise Empty
```

Comme attendu, le type de cette fonction est `'a bin_tree -> 'a`. Ainsi `heap_head ex_heap` s'évalue sur `1`, et `heap_heap Leaf` déclenche une erreur à l'évaluation.

Pour rattraper une exception, on dispose de la construction `try` e `with ...` où e est une expression. Les `...` derrière le `with` désigne un filtrage sur le type `exn`. Lors de l'évaluation d'une expression de cette forme, on commence par évaluer e. Si e ne lève pas d'exception, alors la valeur de retour de cette expression est celle de e. Si l'évaluation de e provoque une interruption en levant une exception `x` (de type `exn`), le système passe alors à l'évaluation du filtrage derrière le `with` sur la valeur `x`. Ce filtrage est exactement comme n'importe quel filtrage, sauf 2 points. Premièrement, OCaml ne vérifie pas l'exhaustivité du filtrage : à l'exécution, si l'exception `x` n'est pas rattrapée, alors elle continue de se propager. Deuxièmement, le type de ce filtrage doit être le même que celui de e.

Par exemple, on définit la fonction ci-contre, de type `'a bin_tree -> bool` qui indique si un tas est vide, en utilisant `heap_head`. Elle teste en fait si `heap_head` lève l'exception `Empty`.

```
let is_empty h =
 try let r=heap_head h in false
 with | Empty -> true
```

L'expression `is_empty ex_heap` renvoie la valeur `false`, et l'expression `is_empty Leaf` renvoie la valeur `true`.

2.2.1.f Traits impératifs

En programmation, on a souvent besoin de fonctions qui effectuent une tâche parallèlement aux calculs, par exemple afficher un texte à l'écran. On dit qu'elles produisent *des effets de bord.* Par rapport aux fonctions précédemment vues, un nouveau souci s'impose alors au programmeur : l'ordre dans lequel les instructions sont exécutées. OCaml est un langage d'appel par valeur strict : tous les arguments des fonctions sont évalués avant d'être appliqués. Il faut noter cependant le cas des constructions conditionnelles ou du filtrage, où seules les branches dont l'évaluation est nécessaire sont exécutées. Pour ordonner l'évaluation des instructions, OCaml utilise comme de nombreux langages le symbole point-virgule (symbole `;`). Ainsi, dans une expression de la forme "e_1 `;` e_2", e_1 est exécutée avant e_2. La valeur d'une telle expression est celle de e_2. La valeur de e_1 est perdue (OCaml affiche un avertissement à la compilation lorsque e_1 n'est pas de type `unit`). Il faut remarquer que cette opérateur de séquence n'est pas vraiment nécessaire, puisque l'expression précédente est équivalente à `let _ =` e_1 `in` e_2.

Parmi ces fonctions à effet de bord, notons les fonctions d'affichage : `print_string` et `print_int` qui sont de types respectifs `string -> unit` et `int -> unit` (voir la signification de `unit` en section 2.2.1.a).

Il y a aussi les fonctions qui gèrent explicitement la mémoire : elles permettent de manipuler plus efficacement les données. Par exemple, une structure comme les files (schéma premier entré/premier sorti) est plus efficace quand elle est codée en manipulant explicitement la mémoire. L'exemple typique de ces structures de données est la notion de référence. Une réference correspond en fait à une case mémoire, dont le contenu peut être lu et modifié. C'est une valeur de type t `ref` où t est le type de son contenu. Il existe une fonction aussi appelée `ref` et dont le type est `'a -> 'a ref`. Cette fonction crée (ou alloue) une case mémoire, en initialisant son contenu. Si e est une expression qui s'évalue sur une référence, `!`e permet d'accéder à la valeur contenue dans cet référence, et e `:=`e' permet de stocker la valeur obtenue après évaluation de e' dans cette référence. En fait, `!` est une fonction de type `'a ref -> 'a`, et `:=` est de type `'a ref -> 'a -> unit`.

Par exemple, on crée une référence `x` en l'initialisant à `2`.

```
let x = ref 2
```

OCaml dit que `x` est de type `int ref`. L'expression `!x` s'évalue sur 2. L'expression "`x:=!x+1 ; !x`" s'évalue sur `3`. Reprenons l'exemple de la fonction `supx` précédent.

On redéfinit ci-contre cette fonction.

```
let supx y = y > !x
```

L'expression `let a = supx 4 in x:=5*(!x); (a, supx 4)` s'évalue sur `(true,false)`.

L'introduction de ces valeurs physiquement modifiables a cependant une influence sur le typage. Supposons en effet la définition ci-contre.

```
let idr = ref (fun x -> x)
```

Si cette fonction `idr` a le type polymorphe `('a -> 'a) ref`, alors l'expression ci-contre est bien typée.

```
idr := (fun x -> x+2) ;
!idr true
```

Or l'exécution de cette instruction serait dramatique, car elle ajouterait un booléen et un entier...

Pour éviter ce problème (voir [Ler92]), il existe en OCaml une autre forme de variables de type : *les variables de type faibles*. Contrairement aux variables de type usuelles, elles ne sont instanciables qu'une seule fois. Leur quantification implicite n'est donc pas universelle, mais plutôt existentielle (mais toujours en forme prénexe). Au lieu d'être préfixées d'un simple "`'`", ces variables sont préfixées par un "`'_`".

Par exemple, pour la constante `idr` précédente, OCaml donne en fait le type `('_a -> '_a) ref`. Si on évalue maintenant `idr := (fun x -> x+2)`, le type de `idr` devient `(int -> int) ref`.

OCaml distingue donc les expressions dites "expansives", qui présentent le risque de modifier physiquement une valeur, et les autres dites "non expansives". Les expressions "expansives" sont les valeurs physiquement modifiables et les expressions qui ne sont pas des valeurs (applications de fonction, conditionnelle, filtrage, etc ...). Les expressions "non expansives" sont principalement les valeurs non physiquement modifiables (listes, valeurs fonctionnelles, etc...). Seules les expressions "non expansives" ont des variables qui sont quantifiées universellement. Après une généralisation (ou quantification), les expressions "expansives" ont des variables de type faibles. Cet aspect concerne aussi des programmes fonctionnels purs. Par exemple, sous l'hypothèse de la fonction nommée `id` définie page 38.

La constante `foo` ci-contre a le type `('_a -> '_a) * ('_b -> '_b)`.

```
let foo = (id,id id)
```

Mais, la constante `foo` ci-contre a le type `('a -> 'a) * ('b -> 'b)`

```
let foo = (id,id)
```

2.2.2 Présentation du système de modules

Une présentation détaillée du système de modules est donnée dans [Ler94], [Ler95] et [Ler00a]. Reconsidérons l'exemple précédent sur les tas pour implanter les listes triées. Plusieurs questions viennent à l'esprit :

- Remarquant que les tas prennent un peu plus de place mémoire que les listes, on souhaite plutôt utiliser une implantation des listes triées utilisant directement des listes triées, lorsqu'on sait que l'on manipule des petites listes et des petites données. Comment faire cohabiter ces deux implantations relativement simplement, sans changer le code utilisant ces implantations pour manipuler des listes triées ?
- On trouve que dans les fonctions implantant les tas, il est un peu lourd de reporter partout un paramètre `o` pour l'ordre : comment fixer ce paramètre pour toute l'écriture des fonctions sur les tas ?
- Les tas reposent sur une structure de données plus générale : les arbres binaires. Avec les fonctions précédentes sur les tas, il est possible de donner n'importe quel arbre binaire (qui n'est pas un tas) comme argument à une fonction, alors que celle-ci attend un tas. En particulier, il est possible de passer à une fonction attendant un tas pour un certain ordre, un tas produit par un autre ordre. Comment renforcer le typage afin de ne pas risquer de confondre un tas pour un certain ordre, avec un arbre binaire quelconque ?

Les modules permettent de répondre à ces différentes questions. Ils forment une catégorie syntaxique à part dans le langage : ce ne sont pas des citoyens de première classe. Ils ne peuvent pas être passés en argument de fonctions ou comme valeur de retour. Mais ils sont bâtis sur le même modèle que le reste du langage : ils ont un système de types appelés *signatures*, ils ont un mécanisme de "fonctions" appelés *foncteurs*, qui peuvent être mêmes passés comme argument à d'autres foncteurs. Enfin, les modules peuvent être compilés séparement.

2.2.2.a Modules de base

Une signature est une suite de déclarations/définitions de type et de déclarations de valeurs. Les déclarations/définitions de type sont préfixés d'un mot-clef `type` qui introduit leur nom et éventuellement leur définition. Les déclarations de valeur sont précédées du mot-clefs `val` : elles introduisent le nom de la valeur, et son type. L'intérêt des définitions de type, est d'exporter ce type. Au contraire une déclaration de type, cache l'implantation de ce type.

Par exemple, on définit la signature `Order` qui déclare un type `t` et une valeur `order`. Le texte entre `(*` et `*)` est un commentaire. Le type de module est défini par le texte entre les mots-clefs `sig` et `end`. Le mot-clef `module type` permet d'introduire le nom `Order` pour désigner ce type de module.

```
module type Order =
  sig
    type t (* type support de l'ordre *)
    val order: t -> t -> bool
  end
```

On peut maintenant définir un module qui admet une telle signature comme type : il doit définir tous les types déclarés ou définis dans cette signature, ainsi que toutes les valeurs déclarées, mais il n'est ni obligé de tenir compte de l'ordre de ces déclarations/définitions, ni de se limiter à celles-ci.

Par exemple, le module `IntOrder` ci-contre admet `Order` comme signature. Le module est défini par le texte entre les mots-clefs `struct` et `end`. Le mot-clef `module` permet d'introduire un nom pour ce module.

```
module IntOrder =
  struct
    let minus x y = x - y
    let order x y = (minus y x) < 0
    type t = int
  end
```

On peut utiliser les valeurs de `IntOrder` dans les expressions OCaml. L'accès aux valeurs est effectué via la notation point (symbole `.`) : l'expression `IntOrder.order` désigne la valeur `order` du module `IntOrder`. On a ainsi sans le savoir déjà utilisé plusieurs fois la valeur `fold_left` du module `List` de la distribution d'OCaml en invoquant `List.fold_left`. Ici `(IntOrder.order 3 4)` renvoie la valeur `false`.

En fait, la signature inférée par OCaml pour `IntOrder` n'est pas celle nommée `Order`, mais celle donnée ci-contre. Ceci dit, il y a en fait une relation de sous-typage entre ces deux signatures : cette dernière est un sous-type de `Order`.

```
sig
  val minus : int -> int -> int
  val order : int -> int -> bool
  type t = int
end
```

On ne donne pas la définition formelle du sous-typage des modules. Intuitivement, le sous-typage permet de considérer les modules, modulo les transformations suivantes : permutation de déclarations et de définitions ; oubli de déclarations ou de définitions ; substitution d'un nom de type apparaissant dans une définition par sa valeur associée dans une partie de la signature et inversement. Par exemple, la signature précédente est sous-type de `Order`, par permutation de `order` et `t`, puis par substitution de `int` par `t`, puis par oubli de la définition de `t`.

Pour chaque expression de module, il existe un type principal : tous les autres types acceptables pour cette expression de module sont des sur-types du type principal. OCaml infère pour chaque expression de module, son type principal.

On peut contraindre un module à avoir une certaine signature. Ainsi, `AbsIntOrder` a `Order` comme type. En particulier, le type `AbsIntOrder.t` est *abstrait* : son implantation est masquée. La valeur `AbsIntOrder.order` a le type `AbsIntOrder.t -> AbsIntOrder.t -> bool`.

```
module AbsIntOrder: Order =
  struct
    let minus x y = x - y
    let order x y = (minus y x) < 0
    type t = int
  end
```

Et `AbsIntOrder.order 3 4` provoque une erreur de type, car comme le type `AbsIntOrder.t` est abstrait, il est incompatible avec `int`. Ce module `AbsIntOrder` est en fait inutilisable, car on n'a donné aucun moyen de construire des valeurs de type `AbsIntOrder.t`

Le choix de rendre abstrait ou pas un type dépend de ce qu'on souhaite offrir aux utilisateurs. Parmi les bibliothèques de la distribution d'OCaml, la signature du module `Num` (voir [MM94]) exporte ainsi le type `num` : [8]

```
type num = Int of int | Big_int of Big_int.big_int | Ratio of Ratio.ratio
```

Par contre, les types `Big_int.big_int` et `Ratio.ratio` sont abstraits. Le module `num` offre en fait la possibilité de faire des calculs arithmétiques exacts avec la représentation adaptée : entiers machine, entiers non bornés ou rationnels exacts. La représentation du résultat des calculs est en fait choisie par la fonction invoquée pour les effectuer. Comme le type `num` est exporté, l'utilisateur peut ainsi voir quelle représentation a été choisie. Par contre, l'utilisateur n'accède pas à ces représentations directement (à part pour `int`) : elles sont ainsi protégées contre une utilisation abusive, tout en rendant le code de l'utilisateur indépendant de ces représentations.

Considérons maintenant la construction `with` sur les signatures, qui permet de définir un type qui n'était que déclaré dans une signature. Par exemple, `IntOrder2` ci-contre a le type `sig type t=int val order:t->t->bool end`.

```
module IntOrder2:
 (Order with type t = int) =
  struct
    let minus x y = x - y
    let order x y = (minus y x) < 0
    type t = int
  end
```

L'expression dans le `with` est évaluée dans le contexte où est écrit ce `with`, et non pas dans le contexte de définition du module ou de la déclaration de ce type. Ainsi, dans l'expression `with type b=a` ci-contre, le nom `a` fait référence au type `string`.

```
type a = bool
module type essai =
  sig  type a=int  type b  val s:b end
type a = string
module Foo: (essai with type b=a) =
  struct
    type b=a    type a=int
    let s="coucou"
  end
```

2.2.2.b Les foncteurs

Les modules peuvent eux-mêmes déclarer ou définir d'autres modules, à la manière de la déclaration ou définition de type. Par exemple, la signature des listes ordonnées `OrderedLists` ci-dessous déclare un module `O` qui est l'ordre sur les éléments des listes. Dans la signature ci-dessous, il faut donc bien distinguer `O.t` qui est le type des éléments de la liste, et `t` qui est le type des listes triées.

[8] Attention à ne pas confondre ici les constructeurs de type des modules

```
module type OrderedLists =
  sig
    module O:Order (* éléments de la liste    *)
    type t         (* type support des listes *)

    (* construction *)
    val nil: t
    val cons: O.t -> t -> t

    (* destruction *)
    exception Empty
    val hd: t -> O.t (* lève Empty en cas de liste vide *)
    val tl: t -> t   (* lève Empty en cas de liste vide *)

    val merge: t -> t -> t (* fusion triée *)
    val fold: ('a -> O.t -> 'a) -> 'a -> t -> 'a

    val from_list : O.t list -> t
    val to_list : t -> O.t list
  end
```

Notons aussi dans cette signature la déclaration de l'exception `Empty`, afin que cette exception puisse être manipulée par les utilisateurs des modules implantant cette signature.

On peut maintenant définir le foncteur `ListByHeap` qui prend un argument un module de signature `Order` et construit un module implantant les listes triées. On se place pour cette définition dans le contexte des définitions précédentes sur les tas.

```
module ListByHeap (X: Order) : (OrderedLists with module O=X) =
  struct
    module O=X
    type t = O.t bin_tree
    let nil = Leaf
    let cons x l = heap_insert O.order x l
    let merge l m =  heap_merge O.order l m
    let fold f x l = heap_foldl O.order f x l
    let from_list l = list2heap O.order l
    let to_list l = heap2list O.order l

    exception Empty
    let hd l =
      match l with
        | Node (x,h1,h2) -> x
        | Leaf -> raise Empty
    let tl l =
      match l with
        | Node (x,h1,h2) -> (merge h1 h2)
        | Leaf -> raise Empty
  end
```

Ici, on force le module résultat de ce foncteur à avoir le type `OrderedLists with module O=X`. Autrement dit, le type `t` est abstrait, mais par contre le module `O` ne l'est pas : on dit qu'il est *manifeste*. Ainsi les utilisateurs de ce foncteur ne pourront pas accéder à la représentation des listes triées (comme des tas), mais ils pourront accéder à l'ordre sur les éléments des listes. Notons ici que le fait d'exporter `Empty` dans `OrderedLists` oblige à le redéfinir localement, à l'intérieur du module, ainsi que toutes les fonctions utilisant cette exception.

Pour appliquer un foncteur à un argument, OCaml impose de nommer le résultat de cette application.

Par exemple, ci-contre, on nomme `A` le module obtenu par application de `IntOrder` sur `ListByHeap`.

```
module A = ListByHeap (IntOrder)
```

On peut alors utiliser `A` pour créer une liste triée :

```
let l1 = (A.merge (A.cons 10 (A.cons 7 A.nil)) (A.from_list [4;12;9;2]))
```

Le type de `l1` est `A.t`. Sa valeur est inaccessible pour l'utilisateur car ce type est abstrait. En particulier le type `A.t` est incompatible avec le type `int bin_tree` : les expressions `(A.cons 3 ex_heap)` et `(A.merge l1 ex_heap)` provoquent deux erreurs de typage. Par contre, le type des éléments est lui accessible : `A.hd l1` est de type `A.O.t` qui vaut `int`. Ainsi `(A.hd l1)+1` est de type `int` et vaut `13`.

L'intérêt du mécanisme de type abstrait est donc de protéger la représentation des données contre des utilisations abusives. Ici, pour chaque instanciation de ce foncteur avec un ordre différent sur les éléments, il y aura un type abstrait des listes triées spécifique pour cet ordre.

Notons cependant que deux mêmes ordres peuvent induire des types abstraits de tas incompatibles. Par exemple, on définit le module `Foo` ci-contre, et on l'applique ensuite sur `IntOrder`. On nomme `l2` (resp. `l3`) un tas issu de `BarX` (resp. `BarY`). Dans ce contexte, `B.BarX.t`, le type de `l2` est égal à `A.t`, mais pas à `B.BarY.t`. Ainsi `(A.merge l1 l2)` est bien typé, mais `(A.merge l1 l3)` et `(B.BarX.merge l2 l3)` sont mal typés. Le simple “renommage” d'un module peut donc entraîner des incompatibilités de type.

```
module Foo (X:Order) =
 struct
   module BarX = ListByHeap (X)
   module Y = X
   module BarY = ListByHeap (Y)
 end

module B = Foo(IntOrder)

let l2 = B.BarX.from_list [1;5;4;3]
let l3 = B.BarY.from_list [1;5;4;3]
```

Plus généralement, au cours du typage, le système n'effectue pas de calculs sur les modules. Il se contente en fait de propager des égalités de types lors des instanciations de foncteurs et des renommages de modules via l'opérateur `with`. Ce n'est donc pas un *système de types dépendants*, contrairement à ce qu'il peut laisser sembler.

2.2.2.c Polymorphisme et modules

Les modules donnent une certaine forme de contrôle du typage, via le mécanisme d'abstraction de type. Ils permettent en fait aussi de contrôler le polymorphisme, et d'étendre ainsi l'expressivité du polymorphisme. On peut par exemple forcer les utilisateurs d'un foncteur à écrire des fonctions polymorphes. En reprenant l'exemple de la fonction `bad_function` précédente (page 38) qui ne typait pas, on peut désormais écrire une fonction `good_function` qui lui est presque[9] équivalente et qui type :

```
module GoodFunction (M: sig  val f:'a->'a  end) =
```

[9]Du fait que `Good_function` est un module, on ne pourra instancier `f` que par des fonctions créées statiquement, alors que les fonctions peuvent prendre en argument des fonctions créées dynamiquement

```
 struct
   let good_function g = (g (M.f true) (M.f 5) (M.f 6))
 end
```

C'est bien une définition de foncteur correcte, que l'on peut instancier par `(struct  let f x = x  end)`. Mais comme on est exigeant, on aimerait aussi pouvoir instancier `M.f` dans la fonction `good_function` par `(fun x->[x])`. On serait tenter de définir alors le module :

```
module NotSoGoodFunction (M: sig  val f:'a->'b  end) =
 struct
   let good_function g = (g (M.f true) (M.f 5) (M.f 6))
 end
```

Mais, le type donné à la fonction `f` est alors trop général, et on ne peut pas instancier `NotSoGoodFunction`, hormis par des modules comme `(struct let f x = raise Empty end)` ou `(struct let rec f x = f x end)`. En particulier on ne peut instancier `f` ni par `(fun x->x)` de type `'a -> 'a` ou par `(fun x->[x])` de type `'a -> ('a list)`

Une solution à ce problème est d'utiliser un type paramétré comme l'autorise OCaml :

```
module BetterFunction (M: sig  type 'a t  val f:'a->('a t)  end) =
 struct
   let good_function g = (g (M.f true) (M.f 5) (M.f 6))
 end
```

La fonction `good_function` ci-dessus a ainsi le type `(bool M.t -> int M.t -> int M.t -> 'a) -> 'a`. On peut alors instancier ce module par

```
struct
  type 'a t = 'a
  let f x = x
end
```

ou par

```
struct
  type 'a t = 'a list
  let f x = [x]
end
```

En conclusion, on espère avoir montré que, plus qu'un simple moyen de découpage du code en morceaux, les modules de OCaml offrent des mécanismes puissants pour contrôler et renforcer le typage de OCaml.

2.2.3 Description générale des objets

Alternativement au système de modules, OCaml offre un autre mécanisme de découpage du code : les objets. Si ces deux systèmes ont comme buts communs de rendre le code plus facilement maintenable et réutilisable, ils sont à utiliser chacun dans un esprit très différent. Les modules sont des boites noires et fermées, qui ne sont utilisables que dans le contexte prévu par le programmeur : ils sont donc l'archétype de la programmation “séparée”, les modules peuvent être implantés complètement indépendamment les uns des autres, seules comptent leurs signatures. Au contraire, grâce à l'héritage et la liaison retardée, les objets offrent beaucoup plus de souplesse au niveau de la réutilisation du code. Par contre, dans un gros développement, il peut devenir assez difficile de comprendre leur comportement. Voir [Vou98] pour une étude sur les rapports entre modules et objets. Les objets de OCaml sont décrits de manière approfondie dans [Rém98] ou [Rém97].

Intuitivement un objet est une sorte de module qui serait une valeur : par exemple, passable comme argument ou comme valeur de retour de fonctions. Une différence syntaxique entre les objets et les modules est que les identificateurs d'objets sont en minuscules, et que on accède aux champs d'un objet en utilisant le symbole dièse, plutôt que le symbole point.

Par exemple, la fonction `foo` ci-contre prend un objet `o` et évalue le champ `m` de `o`.

```
let foo o = o#m
```

En fait, `m` fait référence à une *méthode* de l'objet, c'est-à-dire un certain traitement informatique que connaît cet objet. On dira que l'*on invoque la méthode* `m` *de l'objet* `o`[10].

Plus précisement, les objets sont des valeurs structurées, qui contiennent des données, stockées dans des *variables d'instance*, et des traitements sur ces données, stockés dans les *méthodes*. En OCaml, les variables d'instance peuvent être traitées de façon fonctionnelle (pour changer ces valeurs, on crée un nouvel objet, par recopie de l'objet courant) ou de façon impérative (on change ces valeurs en place, dans l'objet).

Ce style fonctionnel n'est pas très classique dans les langages orientés objet, c'est néanmoins celui qu'on retiendra ici pour présenter les objets. Ce style est en effet plus simple (et donc plus sûr) pour raisonner sur les objets, car ceux-ci ne risquent pas d'être modifiés physiquement au cours de l'exécution.

L'originalité essentielle des objets par rapport aux modules vient en fait de la façon dont on les construit, via des *classes*. On va donner un panorama des objets, en utilisant un exemple autour de la notion de liste. Pour distinguer les listes standard de OCaml des listes de cet exemple, on appelera ces dernières “objets-listes”. Cet exemple s'effectue dans le contexte des fonctions précédentes sur les tas, et en particulier de l'exception `Empty`.

2.2.3.a Exemple de la classe des listes abstraites

Les classes permettent de décrire une famille d'objets. Elles en décrivent les *variables d'instance* et les *méthodes*. Les classes peuvent être virtuelles : certaines méthodes peuvent être uniquement déclarées, ce qui permet de les utiliser à l'intérieur de la classe, sans connaître leur implantation réelle. L'implantation réelle des méthodes (y compris de celles qui ne sont pas virtuelles) ne s'effectue pas au niveau des classes, mais au moment de la création des objets. Le caractère virtuel des méthodes ou des classes est indiqué par le mot-clef `virtual`.

Les classes peuvent être polymorphes (mais pas les méthodes qui sont monomorphes, on reviendra sur ce point en section 2.2.4.b). Les paramètres de types doivent être indiqués explicitement comme dans une définition de type. Par exemple, on peut créer la classe `abslist` des objets-listes abstraits, c'est-à-dire dont la représentation n'est pas encore connue (cf. page 51).

Cette classe est paramétrée par le type des éléments de l'objet-liste : ici la variable de type `'a`. Elle décrit de façon générique un objet, dont les composants sont donnés entre les mots-clefs `object` et `end`. Cette description est générique au sens où plusieurs objets distincts, pourront être créés à partir de cette description. L'objet générique que cette classe décrit, est désigné par la variable `s`. La variable de type `'s` est le type de cet objet. Ces deux variables sont des variables liées qui seront partiellement instanciées au cours de l'héritage, et complètement instanciées lors de la création d'objets. Le sens précis de ces variables va se dégager peu à peu, au cours de cette présentation.

Cette classe contient 3 méthodes virtuelles : `hd`, `tl` et `cons`. La méthode `hd` retourne le premier élément, de type `'a`, de l'objet-liste lorsqu'il est non vide, et lève l'exception `Empty` dans le cas contraire. La méthode `tl` retourne l'objet-liste privé du premier élément, lorsqu'il est non vide, et lève l'exception `Empty` dans le cas contraire. Cet objet-liste est de même type que l'objet `s`.

[10] Usuellement, en terminologie objet, on aussi dit que *le message* `m` *est envoyé à l'objet* `o`.

La méthode cons prend un élément, et retourne un objet-liste qui est la copie de l'objet-liste courant auquel on a ajouté cet élement. Le type de cette méthode cons nous indique ici qu'on a choisi un style fonctionnel. En effet, dans un style impératif, cette méthode modifierait l'objet courant pour lui ajouter l'élément. Le style impératif n'est en fait pas beaucoup plus efficace que le style fonctionnel, puisqu'on ne va pas recopier ici l'ensemble de l'objet-liste, mais seulement un pointeur sur cet objet-liste : on fait du partage (on profite justement du fait que les objets-listes qu'on partage ne peuvent pas être modifiés). De plus, dans le style impératif, lorsqu'une recopie de l'objet-liste est nécessaire, cela s'avère particulièrement coûteux.

```
class virtual ['a] abslist =
  object (s:'s)
    (* hd et tl doivent lever Empty  *)
    (*         en cas d'echec        *)
    method virtual hd: 'a
    method virtual tl: 's
    method virtual cons : 'a -> 's
    method is_empty =
      try
        s#hd; false
      with
       | Empty -> true
    method iter f =
      try
        s#tl#iter f;
        f s#hd
      with
       | Empty -> ()
    method merge (l:'a abslist) =
     let aux = ref s
     in
       begin
         l#iter (fun x ->
                    aux:=(!aux)#cons x);
         !aux
       end
  end
```

A partir de ces méthodes virtuelles, on peut définir (au moins) trois méthodes : is_empty qui indique si l'objet courant est l'objet-liste vide, iter qui permet d'itérer une fonction f sur tous les éléments de l'objet-liste et merge qui fait la fusion d'un objet-liste l avec l'objet-liste courant.

OCaml donne à la classe abslist le type ci-contre. On remarque que OCaml a renommé tout seul 's en 'b, mais ce n'est pas bien grave, car comme on a l'a vu tout à l'heure, c'est une variable de type liée.

```
class virtual ['a] abslist :
  object ('b)
    method virtual cons : 'a -> 'b
    method virtual hd : 'a
    method is_empty : bool
    method iter : ('a -> unit) -> unit
    method merge : 'a abslist -> 'b
    method virtual tl : 'b
  end
```

La méthode is_empty est de type bool. Lors de l'invocation de cette méthode, on commence donc par exécuter la méthode hd de s. L'objet s désigne ici l'objet sur lequel on va invoquer la méthode is_empty. Autrement dit, si o est un objet qui possède cette méthode is_empty, lorsqu'on invoque is_empty sur l'objet o, on va substituer s par o dans le code ci-dessus. Lors de cette invocation, on commencera donc par invoquer hd sur o. On remarque donc le caractère récursif de ce mécanisme. Cette liaison sur s s'appelle liaison *self* : c'est une auto-référence.

Poursuivons maintenant l'étude du corps de is_empty. Si cet objet s représente une liste non vide, cette expression va finalement valoir false. Si cet objet représente la liste vide, l'exécution de hd va lever l'exception Empty. Comme cette exception est rattrapée derrière le with, l'expression va finalement valoir true.

La méthode iter a le type ('a -> unit) -> unit[11]. Le paramètre formel de iter est donc une fonction f à valeur dans unit et d'argument de type 'a (le type des éléments de la liste). Cette

[11]Le type de cette fonction peut sembler un peu restrictif, on verra en 2.2.4.b les causes de cette restriction.

méthode commence par exécuter `s#tl#iter f`. Ainsi, on invoque `iter` sur `s#tl`, qui est le résultat de l'appel de `tl` sur `s`. Si `s` est l'objet-liste vide, l'exception `Empty` sera levée lors de l'évaluation de `tl`, puis rattrapée dans le `with`, et `iter` retournera finalement la valeur `()`. Sinon, `tl` retournera la queue de `s`, sur laquelle on appliquera donc `iter` avec la fonction `f`. Ensuite, l'expression `f s#hd` sera évaluée : comme `s` est non vide, `hd` retournera son premier élément qui sera passé à `f`. Ainsi, intuitivement, `iter` applique `f` successivement à tous les éléments de `s`, en commençant par la fin. On utilise donc ici la récursivité qu'autorise la référence à `s`.

La méthode `merge` a le type `'a abslist -> 's` (où `'s` est le type de `s`). La classe `abslist` définit en fait aussi une abréviation de type : elle donne son nom a un type objet. Comme la classe, ce type objet est paramétré par le type des éléments. C'est le type des objets, instances de `abslist`, mais qui n'ont pas d'autres méthodes que celles définies dans `abslist`.

Ce type est donné in extenso ci-contre. Comme l'indique le mot-clef `as`, ce type est cyclique : la variable de type `'b` est liée et désigne ce type objet. La variable de type `'a` est un paramètre de ce type qui correspond au type des éléments des objets-listes.

```
< cons : 'a -> 'b ; hd : 'a ;
   is_empty : bool ;
   iter : ('a -> unit) -> unit;
   merge : 'b -> 'b;
   tl : 'b
> as 'b
```

La méthode `merge` fait donc la “fusion” d'un objet de type `'a abslist` avec l'objet courant, pour retourner un objet de type `'s`. Dans le corps de la méthode `merge`, on commence par définir la référence `aux` que l'on initialise à `s`, l'objet courant. Le contenu de `aux` sera donc de type `'s`. Ensuite, on crée la fonction `fun x -> aux:=(!aux)#cons x` qui prend un élément de `x` de type `'a`, qui l'insère (via la méthode `cons`) à la liste contenue dans `aux`, et stocke ce résultat dans `aux`. Cette fonction est donc de type `'a -> unit`. On l'applique alors à `l#iter`. Finalement, on renvoie la valeur contenue dans `aux`. Cette valeur sera donc le résultat de l'application de la fonction précedente sur tous les éléments de `l`. Autrement dit, la concaténation de `l` avec `s`, car `iter` va insérer les éléments de `l` dans le contenu de `aux` en commençant par les derniers éléments de `l`.

2.2.3.b Classe encapsulant les listes du langage

Pour implanter ces objets-listes, une première approche consiste à encapsuler les listes déjà définies via un type récursif.

On crée donc la classe `syslist` encapsulant les listes du langage.

```
class ['a] syslist l =
  object
    val rep = l
    inherit ['a] abslist
    method cons x = {< rep = x::rep >}
    method hd =
     match rep with
       | x::l -> x
       | [] -> raise Empty
    method tl =
     match rep with
       | x::l -> {< rep = l >}
       | [] -> raise Empty
  end
```

Le type que le système donne pour cette classe est le suivant :

```
class ['a] syslist :
  'a list ->
  object ('b)
    val rep : 'a list
    method cons : 'a -> 'b
    method hd : 'a
    method is_empty : bool
    method iter : ('a -> unit) -> unit
    method merge : 'a abslist -> 'b
    method tl : 'b
  end
```

Cette classe n'est pas déclarée virtuelle, elle doit donc donner une implantation pour chacune de ses méthodes. Elle est paramétrée par le type `'a` des éléments. Elle a aussi un argument `l`, une liste d'éléments qui va servir à initialiser la variable d'instance introduite par le mot-clef `val` et nommée `rep`.

La classe `syslist` commence par hériter de `abslist` (mot-clef `inherit`) : elle récupère donc toutes les méthodes définies ou déclarées dans `abslist`. Elle pourrait éventuellement redéfinir ces méthodes, mais ici, elle se contente d'implanter les méthodes virtuelles : `hd`, `tl` et `cons`.

La variable d'instance `rep` contient la représentation de l'objet-liste sous forme d'une liste du langage.

Dans la méthode `cons`, la construction `{<` v_1 `=` e_1 `;` ... `;` v_n `=` e_n `>}` désigne la recopie de l'objet courant dans lequel on a assigné aux variables d'instances $v_1 \ldots v_n$ les résultats respectifs de l'évaluation des expressions $e_1 \ldots e_n$. Cette expression est donc un objet du même type que l'objet courant. Ainsi `cons` prend un élément `x` et retourne un nouvel objet, dans lequel `rep` est le résultat de l'insertion de `x` dans la représentation de l'objet courant.

Pour créer un objet, on utilise le mot-clef `new` suivi du nom de la classe et de ses différents paramètres. En fait si *id* est un nom de classe, `new` *id* est une fonction qui a les mêmes paramètres que ceux de la classe, et qui crée un objet instance de cette classe. Les classes sont *ainsi des générateurs d'objets*. Si les objets sont des citoyens de première classe (et même des valeurs), les classes forment une couche syntaxique à part.

Par exemple, on définit un objet-liste `ex1` dont la représentation est la liste "`[4 ;3 ;2 ;1 ;0]`". Il est de type `int syslist` ou encore `int abslist`.

```
let ex1 = new syslist [4;3;2;1;0]
```

Les objets étant des citoyens de première classe, on peut les passer en argument des fonctions.

On peut ainsi définir une fonction `print_iter` qui prend une fonction `p` d'affichage des éléments et un objet-liste `l`, et qui affiche ses éléments en utilisant `iter`, et en les séparant par des espaces, puis en terminant par un retour chariot.

```
let print_iter p l =
 begin
   l#iter (fun x -> p x;
           print_string " ");
   print_newline()
 end
```

Ainsi, en utilisant la fonction `print_int` d'affichage des entiers, on peut afficher les éléments de `ex1`, par l'expression ci-dessous

L'expression ci-contre provoque l'affichage à l'écran du texte suivant "`0 1 2 3 4`"

```
print_iter print_int ex1
```

Avec, l'expression ci-contre, on obtient le texte "`5 6 7 0 1 2 3 4`"

```
print_iter
  print_int
  ((new syslist [7;6;5])#merge ex1)
```

2.2.3.c Types objet et polymorphisme d'inclusion

La fonction `print_iter` a le type `('a -> 'b) -> < iter : ('a -> unit) -> 'c; .. > -> unit` Le type `< iter : ('a -> unit) -> 'c; .. >` est, à cause des "`..`", ce que l'on appelle *un type objet ouvert*. Le symbole "`..`" est ici une sorte de variables de types, que l'on appelle encore *variable de rangée*. Dans ce type, cette variable peut être instanciée par n'importe quelle suite de déclarations de méthodes dont les noms de méthodes ne figurent pas déjà dans le type. Ainsi, ici, ce type pourra être instancié par n'importe quel type objet ayant une méthode `iter`, dont le type instancie `('a -> unit) -> 'c`. La fonction `print_iter` est donc beaucoup plus générale que ce que l'on a annoncé précédemment, car l'objet `l` n'est pas forcément un objet-liste, et `iter` n'a pas forcément l'implantation donnée dans la classe `abslist`. Cependant, comme on l'a vu ci-dessus, si `l` est un objet-liste dont l'implantation de la méthode `iter` est celle donnée en `abslist`, alors `print_iter` affichera les éléments de `l` en mettant ceux de la fin tout à gauche.

Cette forme de polymorphisme s'appelle le *polymorphisme d'inclusion*. On a vu que chaque classe engendre une abbréviation de type objet fermé : ce type correspond à l'ensemble des objets qui sont des instances de cette classe, sans méthode supplémentaire.

De même chaque classe engendre une abbréviation de type objet ouvert. Par exemple la classe `abslist` engendre l'abbréviation de type objet ouvert `#abslist`. Ainsi `'a #abslist` est égal au type ci-contre.

```
< cons : 'a -> 'b ; hd : 'a ;
   is_empty : bool ;
   iter : ('a -> unit) -> unit;
   merge : 'a abslist -> 'b;
   tl : 'b ; .. > as 'b
```

N'importe quel objet instance d'une classe qui étend `abslist` aura donc un type qui est lui-même une instance ce type polymorphe.

Pour être tout à fait exact, `'a #abslist` n'est pas une abbréviation de type, mais de schéma de type : dans une expression de type, chaque occurrence de `'a #abslist` représente une instance différente de ce schéma de type. Pour "identifier" 2 instances, il faut donc nommer une de ces instances avec le mot-clef `as`, comme dans les expressions cycliques de type. Par exemple, si on définit la fonction `misc` ci-dessous

```
let misc (f: 'a #abslist -> unit) (x: 'a #abslist) (y:'a #abslist) =
   (y#zoo); (f (y#merge x))
```

OCaml va lui donner le type ci-contre (au renommage près des variables). A cause du type de `merge`, l'argument `x` a le type `'a abslist` et le type de l'argument de `f` (nommé `'b` ci-dessus) est le même que celui de `y`. Comme on invoque une méthode `zoo` sur `y`, ce type `'b` est une instance de `'a #abslist` qui contient une méthode `zoo`.

```
((< cons : 'a -> 'b; hd : 'a;
      is_empty : bool;
      iter : ('a -> unit) -> unit;
      merge : 'a abslist -> 'b;
      tl : 'b;
      zoo : 'c; .. > as 'b)
   -> unit)
 -> 'a abslist -> 'b -> unit
```

Ainsi, la définition ci-dessous provoque une erreur de typage, car `x` est appliqué à `f` sans avoir la méthode `zoo`.

```
let misc2 (f: 'a #abslist -> unit) (x: 'a #abslist) (y:'a #abslist) =
   (y#zoo); (f (y#merge x)); (f x)
```

2.2.3.d Héritage multiple

Pour éviter d'avoir à reporter partout en paramètre la fonction d'affichage des éléments `print_int`, on peut mettre `print_iter` comme une méthode de l'objet. La fonction d'affichage des éléments sera alors donnée comme un paramètre de classe.

On décide en fait de créer un nouvelle classe `foo_print` pour définir cette méthode d'affichage, appelée `print`. La fonction d'affichage des éléments est le paramètre `p`.

```
class virtual ['a] foo_print p =
 object (s)
   method virtual iter : ('a -> unit) -> unit
   method print =
     begin
      s#iter (fun x -> p x; print_string " ");
      print_newline()
     end
 end
```

On peut alors définir la classe `syslist_print` qui étend à la fois `syslist` et `foo_print` :

```
class ['a] syslist_print p l =
 object
   inherit ['a] foo_print p
   inherit ['a] syslist l
 end
```

Les méthodes de mêmes noms entre `foo_print` et `syslist` doivent avoir des types unifiables : c'est-à-dire une instance commune.

Il faut noter que si une même méthode avait été implantée dans `foo_print` et `syslist`, c'est l'implantation de `syslist` qui aurait été choisie. L'héritage multiple est cohérent avec le mécanisme de redéfinition décrit en section 2.2.3.e.
On pose maintenant

```
let ex2 = new syslist_print print_int [9;8;7;6;5]
```

Cet objet a le type `int syslist_print` qui est distinct du type objet fermé `int abslist`. En effet, ce premier type objet contient la méthode `print` qui n'apparaît pas dans ce dernier.

L'expression `ex2#print` provoque l'affichage du texte "`5 6 7 8 9`". De même, l'expression `print_iter print_int ex2` est typable et affiche ce même texte. L'expression `(ex2#merge ex1)#print` est typable car `ex1` est de type `int abslist`. Elle provoque donc du texte "`5 6 7 8 9 0 1 2 3 4`".

Par contre, l'expression `ex1#merge ex2` provoque une erreur de typage, car `ex2` n'est pas de type `int abslist` : il a une méthode `print` que `int abslist` ne possède pas. Grâce au sous-typage explicite de OCaml, on peut cependant convertir `ex2` en un `int abslist`. On écrit pour cela `(ex2 :> int abslist)`. Cette conversion est une simple conversion de type qui n'a pas de coût à l'exécution. L'expression `(print_iter print_int (ex1#merge (ex2 :> int abslist)))` affiche "`0 1 2 3 4 5 6 7 8 9`".

2.2.3.e Redéfinitions de méthodes et de variables d'instances

Au cours de l'héritage, on peut redéfinir des méthodes. Cette redéfinition peut-être motivée par des soucis d'efficacité (on spécialise une méthode), mais elle peut être une simple réutilisation de code : on change (ou on précise) la sémantique de cette méthode.

Par exemple, pour définir la classe `osyslist` des listes ordonnées, on peut étendre la classe `syslist_print` et redéfinir la méthode `cons`. Dans cette classe la méthode `cons` insèrera le nouvel élément de manière à conserver le caractère trié de la liste. Pour écrire cette méthode `cons`, on va utiliser une fonction intermédiaire `oinsert`. Cette fonction permet de raisonner directement sur la représentation de l'objet.

Cette classe `osyslist` admet donc finalement trois paramètres. Le paramètre `o` est un ordre, `p` une fonction d'affichage, et `l` une liste d'initialisation. On remarque dans la méthode `cons` la possibilité d'utiliser la variable d'instance `rep` définie dans `syslist` dont hérite `syslist_print`. L'ordre entre le `inherit` et la méthode `cons` est ici important, puisque c'est la dernière implantation de `cons` qui sera conservée : si on avait inversé la ligne `inherit` avec la ligne `method`, ce serait l'implémantation de `syslist` qui aurait été conservée.

```
class ['a] osyslist o p l =
 object
   inherit ['a] syslist_print p l
   method cons x =
     let rec oinsert l =
       match l with
         | y::m when (not (o x y))
               -> y::(oinsert m)
         | _ -> x::l
     in {< rep = oinsert rep >}
 end;;
```

Dans le premier filtre de la fonction `oinsert` on utilise la construction `when` afin de n'appliquer ce filtre que lorsque la condition derrière le `when` est respectée.

On peut facilement vérifier que si la liste contenue dans `rep` est triée, toutes les méthodes de `osyslist` vont maintenir cette liste triée. Attention, il faut donc que la liste d'initialisation `l` soit triée. On définit l'objet-liste trié `ex3` suivant :

```
let ex3 = new osyslist (<) print_int [0; 2; 5; 7; 9; 12; 14]
```

La fonction d'ordre choisie ici est `(<)` une fonction prédéfinie, qui correspond à l'ordre croissant usuel sur les entiers. On teste ainsi que `(ex3#cons 8)#print` affiche "`14 12 9 8 7 5 2 0`". On peut aussi tester la méthode `merge` de `ex3` qui doit nous donner un objet-liste trié : `(ex3#merge ex1)#print` affiche "`14 12 9 7 5 4 3 2 2 1 0 0`". On observe donc bien ainsi que dans la méthode `merge`, c'est bien la dernière implantation de `cons` qui a été choisie : le code associé aux méthodes d'un objet n'est fixé qu'au moment de la création de l'objet. C'est ce qu'on appelle en OCaml la liaison tardive.

Plus exactement, le code associé à une méthode dans un objet ne dépend que de la classe à partir de laquelle il a été créé. On dit de cet objet qu'il est une instance de cette classe. Toutes les instances d'une classe partagent le même code pour chaque méthode.

Pour ne pas imposer la contrainte que la liste d'initialisation de la classe `osyslist` soit triée, on aurait donc envie de rédéfinir aussi `rep` afin de trier `l` au moment de l'initialisation. En fait, en OCaml lorsqu'on "redéfinit" une variable d'instance, c'est plutôt une **nouvelle** variable d'instance qui est créée. L'ancienne variable d'instance n'est plus accessible pour les nouvelles méthodes, mais sa valeur reste inchangée pour les anciennes méthodes.

Pour s'en persuader, on peut introduire la classe `foo` suivante, dans laquelle on redéfinit une variable d'instance `a`. Ici, `(new foo)#a1` vaut `1` mais `(new foo)#a2` vaut `2`. Contrairement aux méthodes, les variables d'instances ont *une portée statique à l'intérieur de la classe.*

```
class foo =
 object
   val a = 1
   method a1 = a
   val a = 2
   method a2 = a
 end
```

Pour résoudre ce problème d'initialisation, il faut en fait changer l'initialisation au moment du `inherit`. Au lieu d'hériter de `syslist_print` avec `l` comme liste d'initialisation, on va hériter de `syslist_print` sur le résultat du tri de `l`. Pour effectuer ce tri, on va justement utiliser la fonction `oinsert` : on fait un tri par insertion. On utilise la même méthode que pour transformer une liste en un tas, via la fonction `List.fold_left` déjà vue précédemment. Pour que la fonction `oinsert` soit visible depuis le `inherit` on la définit dans l'environnement local de la classe, en utilisant la construction `let ... in`, qui est aussi valable dans le langage de classe. La méthode `cons` sera alors elle-même plus efficace, car on évite ainsi la création de la fermeture correspondant à `oinsert` à chaque invocation.

```
class ['a] osyslist =
 let rec oinsert o x l =
   match l with
    | y::m when (not (o x y)) -> y::(oinsert o x m)
    | _ -> x::l
 in
   fun o p l ->
    object
      inherit ['a] syslist_print p (List.fold_left (fun m x -> oinsert o x m) [] l)
      method cons x = {< rep = oinsert o x rep >}
    end;;
```

On peut maintenant créer un objet-liste trié à partir d'une liste non triée :

```
let ex3 = new osyslist (>) print_int [12; 5; 14; 2; 9; 7; 0];;
```

Remarquons que par rapport à l'exemple précédent, on a changé l'ordre : le symbole `>` est ici l'ordre décroissant. Ainsi, `ex3#print` affiche "`0 2 5 7 9 12 14`".

2.2.3.f Liaison *super*

Lorsqu'on redéfinit une méthode, on perd donc l'implantation qu'en donnait la classe parente. Parfois, on peut avoir cependant envie de conserver malgré tout cette méthode. Au cours de l'héritage, OCaml donne un moyen de conserver cette implantation. Ce mécanisme, classique dans les langages orientés objet, est connu sous le nom de liaison *super* (ou ancêtre).

En OCaml, on commence par associer à cette classe un nom introduit par le mot-clef `as`. On accède alors au code des méthodes **concrètes** de la classe parente, par invocation des méthodes sur ce nom.

Dans la classe `sysdouble` ci-contre, on associe le nom `father` à la classe parente. On peut alors renommer la méthode `cons` de `syslist_print` en `old_cons`, et on s'en sert pour redéfinir `cons` : on insère en fait chaque élement en double dans la liste.

```
class ['a] sysdouble p l =
 object (s)
   inherit ['a] syslist_print p l
                        as father
   method old_cons = father#cons
   method cons x =
     (s#old_cons x)#old_cons x
 end
```

Mais attention, contrairement à ce qu'il se passe pour la liaison self, ici **`father` n'est pas un objet**, ni une valeur. Aucun type ne lui est associé. Il n'a en fait qu'une existence syntaxique : on ne peut utiliser ce nom que lors d'une invocation avec des méthodes non virtuelles de la classe parente. Ainsi `father#cons` désigne le code associé à la dernière définition de `cons` dans la classe `syslist_print`. C'est donc juste un mécanisme de renommage, utile notamment quand on désire préserver du code qui serait écrasé par héritage multiple.
On peut tester la classe `sysdouble`. L'expression ci-dessous affiche le texte "`7 0 1 1 6 6 3 8 8`".

```
(((((new sysdouble print_int [0;7])#cons 1)#cons 6)#old_cons 3)#cons 8)#print
```

Il faut toutefois se méfier de l'interaction entre ce mécanisme avec la liaison self. Par exemple, changeons la méthode `iter` de `syslist_print` pour parcourir les éléments de la liste, dans le sens de la liste, et non plus dans le sens inverse. On renomme cependant l'ancienne version de `iter` en `old_iter`.

```
class ['a] syslist_niter p l =
 object (s)
   inherit ['a] syslist_print p l as father
    method iter f =
      try
        f s#hd;
        s#tl#iter f
      with
       | Empty -> ()
    method old_iter = father#iter
 end

let foo = new syslist_niter print_int [9;8;7;6;5]
```

Comme attendu, `foo#print` affiche la liste dans l'ordre des éléments de la liste, car `print` utilise la nouvelle version de `iter` : il affiche "`9 8 7 6 5`". Si on utilise l'ancienne version de `iter` pour afficher, via

```
foo#old_iter (fun x -> print_int x; print_string " ")
```

on obtient maintenant "`8 7 6 5 9`". Le comportement de `old_iter` n'est donc pas d'afficher la liste dans l'ordre inverse, comme le faisait le `iter` des objets instances de `syslist_print`. En effet dans l'ancien code de `iter`, on avait codé la récursivité en invoquant `iter` sur `s#tl`. Donc dans le code désigné par `old_iter`, l'occurrence de `iter` désigne la nouvelle version de `iter`. Le comportement de `old_iter` est donc de parcourir les éléments de la liste dans l'ordre, mais en partant du deuxième élément, puis de finir sur le premier élément de la liste.

Ceci illustre en fait les confusions qui peuvent arriver via le mécanisme de liaison tardive : on croit utiliser une méthode qui a une certaine sémantique intuitive, mais celle-ci a "changé", car elle fait référence à une méthode dont la sémantique intuitive a elle-même "changé"... D'un côté la force des objets, c'est qu'une toute petite modification peut changer le comportement d'un grand nombre de méthodes : on a une forte réutilisation de code. C'est aussi leur faiblesse, car cela implique qu'il faut connaître les dépendances entre toutes les méthodes de toutes les classes de la hiérarchie...

2.2.4 Propriétés du typage des objets

Une des grandes originalité du système objet de OCaml est son système de type, qui découle directement de celui de ML. En particulier, il est rare de trouver un langage objet fortement typé, et statiquement typé. Cette propriété confère à ce langage une grande sûreté : tout en ayant un mécanisme dynamique d'invocation de méthodes sur les objets, on n'aura pas d'interruption du genre "method not found" à l'exécution. Le typage assure donc que tout message envoyé à un objet sera bien reçu...

En outre, les classes peuvent être polymorphes, et il existe une notion de polymorphisme d'inclusion sur les types objets.

2.2.4.a Le sous-typage et polymorphisme d'inclusion

OCaml a deux mécanismes d'oubli de méthodes au niveau du système de type : le sous-typage et le polymorphisme d'inclusion. Plusieurs questions se posent à ce stade : quel est leur lien avec l'héritage, et quelle est leur différence.

En fait, l'héritage est un mécanisme syntaxique pour construire de nouvelles classes à partir d'anciennes. Au contraire, le sous-typage et le polymorphisme d'inclusion sont des mécanismes de typage liés à la sémantique des objets dans le langage.

Ceci dit, il a un lien entre héritage et polymorphisme d'inclusion : si une classe A hérite d'une classe B, alors le type fermé et le type ouvert associé à A sont des instances du type objet ouvert associé à B. Mais, il se peut que le type fermé associé à une classe A soit une instance du type ouvert associé à une classe B, sans que A hérite de B : il suffit par exemple que pour chaque méthode de B, A ait une méthode de même nom et de même type.

En revanche, héritage et sous-typage n'ont aucun lien. En particulier, héritage n'implique pas le sous-typage. Ce point est du à la contra-variance du sous-typage. En effet, soient deux types fonctionnels $C_1 = A_1 \to B_1$ et $C_2 = A_2 \to B_2$. Le type C_2 est un sous-type de C_1 si et seulement si B_2 est un sous-type de B_1 et A_1 est un sous-type de A_2. Intuitivement, si f est une fonction de type C_2, et qu'on considère que f est de type C_1, on va pouvoir passer à f des arguments de type A_1 : il faut donc que A_1 soit un sous-type de A_2. Donnons maintenant un exemple de ce point en OCaml. On suppose ainsi que dans la classe `abslist`, on veuille donner à `merge` le type `'s -> 's` au lieu de `'a abslist -> 's`.[12]

Pour cela, on crée en fait une classe `abslist2` qui étend `abslist` en lui rajoutant une méthode `merge2` qui simule `merge` en lui donnant le type `'s -> 's`.

```
class virtual ['a] abslist2 =
 object (s:'s)
   inherit ['a] abslist
   method merge2 (x:'s) =
    x#merge (s:> 'a abslist)
 end
```

On peut alors étendre `syslist_print` en lui rajoutant aussi cette méthode `merge2`. On définit en réalité une classe `syslist_print2` qui hérite à la fois de `abslist2` et `syslist_print`. On crée alors un objet de cette classe, appelé `ex4`. L'expression `(ex4 :> int abslist2)` provoque alors une erreur de type, car `int syslist_print2` n'est pas un sous-type de `int abslist2`.

```
class ['a] syslist_print2 p l =
 object (s:'s)
   inherit ['a] abslist2
   inherit ['a] syslist_print p l
 end

let ex4 =
  new syslist_print2 print_int [1;2;3]
```

Le problème vient en fait de la méthode `merge2` qui est dans le premier cas de type `int syslist_print2 -> int syslist_print2`, et dans le deuxième cas de type `int abslist2 -> int abslist2`. Ce premier type fonctionnel devrait être un sous-type de ce dernier. Mais `int abslist2` n'est certainement pas un sous-type de `syslist_print2`, car ce dernier contient une méthode `print` que ce premier ne contient pas. En fait, du point de vue du système de type, la méthode `merge2` de type `int syslist_print2` peut dans son implantation utiliser la méthode `print` de son argument, ce que n'a pas le droit de faire le `merge2` du type `int abslist2`. Or la cœrcion d'un objet sur un sur-type ne change pas le code associé à ses méthodes. Les objets de type `int syslist_print2` ne peuvent donc pas être cœrcés sur `int abslist2`.

Une profonde différence entre le sous-typage et le polymorphisme d'inclusion tient au fait que le sous-typage permet d'appliquer un objet qui a des "méthodes en trop" sur une méthode monomorphe. Le sous-typage est donc surtout indispensable dans les contextes où on ne peut pas utiliser de fonctions polymorphes (voir par exemple 2.2.4.b).

[12]Une telle méthode est utile par exemple, lorsqu'on représente les listes triées par des tas (cf. 2.2.5.c)

2.2.4.b Monomorphismes des méthodes

Si le lecteur a été subjugué par la fonction `List.fold_left` vue précédemment, il peut se demander pourquoi on a utilisé le style impératif de la méthode `iter` dans la classe `abslist`, au lieu d'employer une jolie méthode polymorphe, ce qui aurait permis de coder ensuite `merge` bien plus simplement, en utilisant un style fonctionnel ! C'est que malheureusement, OCaml interdit les méthodes polymorphes : seules les classes peuvent être polymorphes.

Cela vient du fait que pour conserver les types implicites en ayant une inférence de type calculable, OCaml n'autorise qu'un polymorphisme sous forme prénexe : la quantification universelle n'est autorisée qu'en tête des types. Or autoriser des méthodes polymorphes reviendrait à autoriser des quantificateurs à l'intérieur des types objets, ce qui n'est pas possible.

Les variables de type apparaissant dans les méthodes d'une classe doivent donc toutes être mises en paramètre de cette classe. Cela signifie que pour un objet instance de cette classe, ces variables seront instanciées. Ainsi, il n'y a pas d'objet "polymorphe".

En pratique, cela pose deux types de problèmes. D'une part, les fonctions polymorphes du type `List.fold_left` ne peuvent pas être des méthodes. D'autre part, les méthodes prenant en paramètre un objet (comme la méthode `merge` de la classe `abslist`) ne peuvent pas avoir un type objet ouvert sur ce paramètre, ce genre de type étant polymorphe. Pour coder ce genre de méthode, on a donc intérêt à utiliser le type objet fermé le plus grand possible au sens du sous-typage, afin d'être le plus général possible.

2.2.4.c Variables de rangée faibles

On a vu en section 2.2.1.f qu'introduire des structures de données physiquement modifiables a conduit à introduire la notion de variables de type faible pour rendre ces structures de données compatibles avec le polymorphisme. Cette notion s'étend aux variables de rangée. Par exemple, l'application partielle de `(<)` sur `print_iter` fait apparaître une telle variable de rangée faible : la fonction `(<)` étant polymorphe, `print_iter (<)` a le type `< iter : ('_a->unit) -> '_b ; _..>`

2.2.5 Problèmes de la programmation avec encapsulation des données dans les objets

Classiquement, en programmation orientée objet, les objets permettent de masquer la représentation des données : cette représentation est en effet encapsulée dans l'objet, et seules les méthodes de l'objet ont accès à cette représentation. De l'extérieur, tous les traitements sur les données passent par l'intermédiaire des méthodes. Nous avons ici suivi ce schéma. On va en montrer ici les limites. On essayera de convaincre le lecteur que ces limites ne sont pas dues aux choix de conception de OCaml, mais inhérentes à ce style lui-même. Ces points ont déjà été évoqués dans [Bou99a].

2.2.5.a Une certaine faiblesse du typage

Un premier problème survient lorsque les méthodes sont surchargées : plusieurs sortes d'instances de classe peuvent être extérieurement statiquement indistinguables, mais avoir des comportements dynamiques très différents. En particulier, deux objets peuvent avoir des méthodes de même nom et de même type (donc extérieurement statiquement indistinguables), mais dont les implantations diffèrent. Il est alors facile de faire des confusions, et de ne plus savoir prédire le comportement de ces objets. Ce problème arrive aussi en OCaml.

Par exemple reprenons l'exemple de `syslist_print2` et `abslist2` précédent. On crée maintenant la classe `osyslist2` qui étend la classe `osyslist` des objets-listes triés, en lui rajoutant la méthode `merge2`

```
class ['a] osyslist2 o p l =
 object (s:'s)
   inherit ['a] abslist2
   inherit ['a] osyslist o p l
 end
```

Cette classe a le type ci-contre.

```
class ['a] osyslist2 :
  ('a -> 'a -> bool) ->
  ('a -> 'c) -> 'a list ->
  object ('b)
    val rep : 'a list
    method cons : 'a -> 'b
    method hd : 'a
    method is_empty : bool
    method iter : ('a -> unit) -> unit
    method merge : 'a abslist -> 'b
    method merge2 : 'b -> 'b
    method print : unit
    method tl : 'b
  end
```

Pour changer un peu d'exemple (et profiter du polymorphisme de OCaml), on va maintenant considérer des chaînes de caractères au lieu des entiers :

```
let exa = new osyslist2 (>) print_string ["Thérèse"; "Renaud"; "Valérie"; "Damien"]
let exb = new osyslist2 (>) print_string ["Virgile"; "Daniel"; "Sylvain"]
```

Les objets `exa` et `exb` sont deux objets-listes triés. Ils ont le type : `string osyslist2`. L'objet `exa#merge2 exb` a aussi ce type là, et d'ailleurs l'évaluation de `(exa#merge2 exb)#print` provoque l'affichage du texte "`Damien Daniel Renaud Sylvain Thérèse Valérie Virgile`". On peut aussi définir :

```
let exc = exb#merge2 (new syslist_print2 print_string [])
```

Cet objet est du type `string osyslist2` et sa représentation est une liste triée : l'expression `exc#print` provoque l'affichage de "`Daniel Sylvain Virgile`". L'objet `exa#merge2 exc` est aussi de type `string osyslist2`, mais par contre sa représentation n'est pas une liste triée : l'expression `(exa#merge2 exc)#print` provoque l'affichage de "`Daniel Sylvain Virgile Damien Renaud Thérèse Valérie`". La bizarrerie vient ici du fait que malgré les apparences `exc` n'est pas un objet-liste trié ! Cet objet n'est pas une instance de la classe `osyslist2`, mais de la classe `syslist_print2`. En effet, `exc` a été produit en utilisant la méthode `merge2` de l'objet `exb` sur une instance de `syslist_print2`. Or, si on observe le corps de la méthode `merge2`, elle applique la méthode `merge` de son argument. Comme `merge` crée un objet instance de la même classe que l'objet qui la porte, `merge2` crée une instance de la même classe que son argument.

Ce problème vient du fait que l'on a aucun signe extérieur que des instances de `syslist_print2` et de `osyslist2` n'ont pas la même sémantique. Il faut lire pas-à-pas le programme en connaissant pour chaque objet, la classe dont il est issue. Or a priori, c'est justement ce genre de tâche que l'on cherche à éviter en encapsulant les données dans les objets... Le typage au niveau des objets ne procure donc pas la sécurité offerte par le mécanisme d'abstraction de type des modules.

Ce défaut n'est à notre avis pas lié au système objet de OCaml, mais plutôt inhérent à ce style de programmation. Avec ce style de programmation, on n'a plus de trace que deux modes de représentations issus de deux classes différentes sont éventuellement distincts. Comme on souhaite malgré tout manipuler ces données de la même manière de temps en temps, on est obligé de les assimiler complètement, ce qui est assez dangereux.

2.2.5.b Codage des “constantes”

Un second problème, est le codage des “constantes”. Reprenons l'exemple de la fonction `heap2list`. On souhaite écrire une méthode générique qui transforme un tas en un objet-liste (triée) indépendamment de la représentation de cet objet-liste. On s'aperçoit ici que la classe `abslist` a un profond défaut : il lui manque une méthode pour construire un objet-liste vide. On crée donc une classe `abslist_nil` qui rajoute la méthode `nil` pour créer l'objet liste vide `nil`, et la méthode `from_heap` (de type `('a -> 'a -> bool) -> 'a bin_tree -> 's` qui crée une liste à partir d'un tas.

```
class virtual ['a] abslist_nil =
 object (s:'s)
   inherit ['a] abslist
   method virtual nil: 's
   method from_heap o h = heap_foldl o (fun l x -> l#cons x) s#nil h
 end
```

Ces deux méthodes sont assez étranges : elles sont censées servir à créer des objets-listes, mais pour les invoquer il faut déjà posséder un objet-liste... Au niveau du typage, `nil` a le même type que `tl`, or ces deux méthodes sont très différentes, puisque `tl` représente une fonction unaire, alors que `nil` est une constante. Dans ce style, où les méthodes sont des fonctions implicitement appliquées à l'objet qui les porte, le codage des constantes est donc un peu lourd : on les “applique” à un objet inutile.

2.2.5.c Codage des *“lois binaires de composition interne”*

Si les “constantes” sont un peu lourdes à coder, pour d'autres raisons, il en est de même avec les fonctions qui ont strictement plus d'un argument : seul les fonctions unaires semblent ne poser aucun problème.

Ainsi, on a déjà vu que les fonctions binaires comme `merge2` étaient incompatibles avec le sous-typage. Mais il se peut aussi que ce genre de fonction ait besoin de connaître la représentation de son argument : dans ce cas, cela signifie que l'objet doit exporter sa représentation interne. Il en est ainsi fini du masquage de la représentation. En effet, on n'a pas de moyen de forcer l'argument de `merge2` a être instance de la même classe que l'objet courant, et partager ainsi avec lui une méthode privée. L'argument doit donc exporter sa représentation dans son type objet.

C'est le cas par exemple, si on implante les listes ordonnées par les tas. Pour implanter la fusion de tas (méthode `merge2`), il va falloir utiliser une méthode `rep` qui exporte la représentation de l'argument. Sans cette méthode `rep`, il est tout simplement impossible d'implanter cet algorithme :

```
class ['a] heaplist o p l =
 object (s:'s)
   inherit ['a] abslist2
   inherit ['a] foo_print p
   val h = list2heap o l
   method cons x =
     {< h= heap_insert o x h >}
   method hd = heap_head h
   method rep = h
   method merge2 x =
     {< h = heap_merge o h x#rep >}
   method tl =
     match h with
      | Node (x,h1,h2) ->
          {< h = h1 >}#merge2 {< h = h2 >}
      | Leaf -> raise Empty
 end
```

Cette classe a le type :

```
class ['a] heaplist :
  ('a -> 'a -> bool) ->
  ('a -> 'c) -> 'a list ->
  object ('b)
    val h : 'a bin_tree
    method cons : 'a -> 'b
    method hd : 'a
    method is_empty : bool
    method iter : ('a -> unit) -> unit
    method merge : 'a abslist -> 'b
    method merge2 : 'b -> 'b
    method print : unit
    method rep : 'a bin_tree
    method tl : 'b
  end
```

Notons ici, l'utilisation de `merge2` dans la méthode `tl` afin de bénéficier d'une éventuelle redéfinition de `merge2` au cours de l'héritage.

2.2.5.d Abstraction de type versus encapsulation des données

Finalement, les trois problèmes évoqués ici ont une origine commune : dans ce style de programmation, on assimile abusivement la structure "algébrique" des données (c'est-à-dire l'ensemble des opérations et des propriétés sur ces données) avec ces données. On perd alors la possibilité de parler des données indépendamment des traitements qui s'appliquent dessus : il n'y a pas de dénotation du type des données à l'extérieur de l'objet. Au contraire, le mécanisme d'abstraction de type des modules permet d'introduire une dénotation pour le type de données, en laissant éventuellement cette dénotation "abstraite", c'est-à-dire, sans représentation associée.

De ce point de vue, le style de programmation avec *encapsulation des données dans les objets* est moins général que celui avec *des types abstraits de donnée*. Comme le montre les exemples précédents, il est aussi moins sûr, moins compréhensible et moins efficace : il faut sans cesse décaspuler/encapsuler la représentation des données, alors que le mécanisme d'abstraction de type n'a aucun coût à l'exécution (c'est uniquement un mécanisme de typage).

Le style de programmation des bibliothèques FOC en OCaml (cf. chapitre 5) est un exemple de style qui marie abstraction de type et programmation orientée objet.

2.2.6 Autre style fonctionnel, avec encapsulation des données

Dans le style présenté précédemment, on peut trouver désagréable d'avoir une représentation des données externe au système des objets. Dans ce style, on définit en effet le traitement des données en dehors des objets, avant de les encapsuler dans des méthodes. C'est par exemple ce qui a été fait avec `oinsert` la fonction d'insertion dans une liste triée.

On aimerait donc plutôt définir directement les notions au niveau des objets.

Dans l'exemple des objets-listes, on voudrait ainsi une classe `nilc` créant l'objet-liste vide, et une classe `consc` créant les objets listes non vide. Ces deux classes étant elles-mêmes des sous-classes de `abslist`. Par exemple, `consc` peut être donnée par la classe ci-contre. Cette classe prend en argument un élément `x` et un objet-liste `l`. Ces deux arguments servent à initialiser les deux variables d'instance de l'objet : `h` qui contient la tête, et `t` qui contient la queue.

```
class ['a] consc x l =
  object (s)
    val h=x
    val t=l
    inherit ['a] abslist
    method hd=h
    method tl=t
    method cons y = {< h=y; t=s>}
  end
```

La classe `nilc` est plus difficile à écrire. On aimerait écrire quelque chose comme la classe ci-dessous. **Mais une telle définition provoque une erreur de typage.** En effet, l'objet retourné par `cons` doit être du même type que `s`. Or ici, il est de type `'a consc`. Donc cela impose que `s` soit de type `'a consc`. Autrement dit, lorsqu'on étendra `nilc` en ajoutant une méthode (par exemple `print`), `s` aura une méthode de plus que dans le type `consc` mais sera de type `consc` : ce n'est pas cohérent.

```
class ['a] nilc =
  object (s)
    inherit ['a] abslist
    method hd=raise Empty
    method tl=raise Empty
    method cons y = new consc y s
  end
```

Une solution serait de simplifier `abslist` en une classe `badl` ci-contre. Mais une telle classe ne serait pas vraiment pratique, puisque on ne pourrait pas bénéficier dans l'objet retourné par `cons` des méthodes ajoutées par héritage.

```
class virtual ['a] badl =
  object (s:'s)
    method virtual hd: 'a
    method virtual tl: 's
    method virtual cons : 'a -> 'a badl
  end
```

Une autre solution pour écrire `nilc` est en fait de paramétrer cette classe par la méthode `cons`. Dans l'exemple ci-dessous ce paramètre est donné par `mcons`. On simule ainsi une sorte d'inter-référence entre les objets instances de `nilc` et les objets instances de `consc` : c'est une généralisation de la liaison self. On parlera ici de liaison self mutuelle (à l'image de la récursivité mutuelle).

```
class ['a] nilc mcons =
  object (s)
    inherit ['a] abslist
    method hd=raise Empty
    method tl=raise Empty
    method cons = mcons
  end
```

La création des objets de cette classe est maintenant un peu délicate. On passe par l'intermédiaire d'une fonction récursive `nil` qui permet d'implanter la liaison self mutuelle :

```
let rec nil () = new nilc (fun x -> new consc x (nil ()))
```

On peut alors créer l'objet-liste suivant :

```
let ex5 = (((nil())#cons 1)#cons 2)#cons 4
```

On peut appliquer `print_iter` sur cet objet. Ainsi `print_iter print_int ex5` affiche comme attendu "`1 2 4`".

L'héritage se passe alors sans grande difficulté : on rajoute ici les méthodes `print` et `merge2`, en héritant des classes `foo_print` et `abslist2`.

```
class ['a] consc2 p x l =
  object (s)
    inherit ['a] abslist2
    inherit ['a] foo_print p
    inherit ['a] consc x l
  end
```

```
class ['a] nilc2 p mcons =
  object (s)
    inherit ['a] abslist2
    inherit ['a] foo_print p
    inherit ['a] nilc mcons
  end
```

```
let rec nil2 p =  new nilc2 p (fun x -> new consc2 p x (nil2 p))
```

Pour finir cet exemple, on crée l'objet suivant :

```
let exd = (((nil2 print_string)#cons "Blah")#cons "Foo")#cons "Bar"
```

Comme attendu, l'expression `(exa#merge (exd :> string abslist))#print` affiche "`Bar Blah Damien Foo Renaud Thérèse Valérie`".
De même, l'expression `(exd#merge (exa :> string abslist))#print` affiche "`Blah Foo Bar Damien Renaud Thérèse Valérie`".

Les problèmes évoqués en section 2.2.5 restent inchangés ici. Par exemple avec `exe` au lieu de `exc`, on reproduit le problème de la faiblesse du typage :

```
let exe = exb#merge2 (nil2 print_string)
```

Le principal défaut de ce style est donc cette gestion "à la main" de la notion de liaison self mutuelle, qui complique un peu le code.

2.3 Comparaison du système objet de OCaml avec d'autres systèmes

Voir [Cas97] et [Aba96].

2.3.1 Typage dynamique et surcharge

Une particularité de OCaml est son typage statique. D'autres systèmes orientés objets ont en effet un typage dynamique. Cela rend le langage plus expressif en autorisant plus de surcharge au niveau des méthodes. D'un autre côté, on a vu en section 2.2.5.a que la surcharge peut facilement amener des confusions.

Ainsi, le code exécuté lors de l'envoi d'un message à un objet, ne dépend plus seulement de cet objet mais aussi des arguments passés lors de cet envoi de message. Cela permet par exemple de contourner la contradiction entre sous-typage et héritage vu en section 2.2.4.a. En reprenant cet exemple, supposons qu'on définisse une classe `B` qui étende la classe `abslist2` en rajoutant une méthode `print` et que la méthode `merge2` soit redéfinie, en utilisant la méthode `print` sur son argument. Cœrcer un objet `o` sur le type `abslist2` signifie que la méthode `merge2` utilisée sur cet objet coercé n'est pas le `merge2` de `B`, mais celui de `abslist2`. On surmonte ainsi la contradiction, mais un objet peut alors avoir un comportement différent suivant le type avec lequel il est utilisé !

Contrairement à celle de OCaml, cette forme de liaison tardive n'est donc pas faite au moment de la création de l'objet, mais au moment de l'envoi de message, en fonction du type dynamique des arguments (voir [Cas97]).

Un langage utilisant un typage dynamique est par exemple Axiom. L'instruction "Has" d'Axiom teste le type dynamique d'un "domaine" (unité de bibliothèque d'Axiom) à l'exécution (voir [Jen92] et [Wat95]).

2.3.2 Des "modules" avec de l'héritage

Voir [Apo96], [Anc98] et [Hir00].

Chapitre 3

Preuves

On va d'abord donner une description générale des concepts importants en Coq. Ensuite, on fera une présentation un peu plus détaillée.

3.1 Concepts importants en Coq

On va donner ici les considérations logiques, informatiques et méthodologiques principales qui ont donné naissance à Coq. Cet assistant à la démonstration est directement inspiré de la théorie des types (voir notamment [Dow99b] et [Luo94]).

3.1.1 Les langages de preuves comme langages de programmation certifiée

La notion de preuve est au cœur de l'activité mathématique. Les mathématiciens sont en effet d'accord sur un certain nombre (fini) de règles logiques. Ces règles expriment des transformations syntaxiques sur les phrases mathématiques. Si on peut trouver une suite (finie) de ces transformations syntaxiques qui permettent de passer des phrases mathématiques $A_1, \ldots, A_n$ à la phrase B, alors on dit que de $A_1, \ldots, A_n$ on peut *déduire* B. Une *preuve* de cette déduction est tout simplement la suite de transformations syntaxiques en question. Comme ces transformations sont de nature syntaxique, n'importe qui de sincère peut vérifier qu'elles sont correctement appliquées : il n'y a pas à réfléchir, les transformations s'effectuent *"automatiquement"*. Par exemple, si je sais que "$1 \neq 0$", je peux en déduire *"il existe un entier n tel que $n \neq 0$"*. Ces transformations ne font que reformuler les hypothèses. En mathématiques, on appelle théorème une phrase dont on peut donner une preuve.

En pratique, les connaissances mathématiques n'auraient pas l'étendue qu'elles ont aujourd'hui, si on s'était limité à une telle vision *formelle* des preuves mathématiques. D'une part, établir une telle preuve *formelle* sur un théorème un peu compliqué nécessite beaucoup de temps et d'efforts. D'autre part, une preuve trop détaillée, qui atteint le niveau de "granularité logique" des transformations syntaxiques, est en fait peu intelligible et donc peu convaincante. Les preuves mathématiques réelles ne sont donc pas aussi détaillées que les preuves formelles. Elles sont de "plus haut niveau" (avec un plus gros grain logique) et utilisent plutôt des arguments "sémantiques" et intuitifs que des pures règles syntaxiques. Mais les mathématiciens n'acceptent une preuve que s'ils sont convaincus qu'ils pourraient écrire cette preuve formelle.

En informatique, au contraire, les preuves formelles sont intéressantes d'un point de vue pratique, car on peut justement les faire vérifier par un programme. Mieux, les preuves ont en général un contenu algorithmique, et on peut alors *en extraire des programmes qui réalisent* ce qu'elles démontrent. Par exemple, pour prouver que tous les polynômes de degré 2 sur $\mathbb{C}$ peuvent être factorisés, on va construire une factorisation systématique des polynômes de degré 2. Le fait que la correction de la preuve soit elle-même vérifiée automatiquement garantit que le programme sera lui-même correct. De telles preuves peuvent donc être vues comme des programmes certifiés qui contiennent les arguments permettant de les certifier. Les preuves formelles sont certes pénibles à écrire, mais on peut espérer confier une partie du caractère pénible de cette tâche à un système informatique, qui aidera en quelque sorte un humain à bâtir une preuve complètement formelle.

Malheureusement, dans la logique classique des mathématiciens, la règle du tiers exclus (pour toute proposition A, la proposition *"A ou non A"* est vraie), permet des preuves qui n'ont pas de contenu algorithmique. Par exemple, en logique classique, on peut démontrer que toute fonction des entiers dans les entiers admet un minimum. En effet, si on suppose l'existence d'une f fonction des entiers dans les entiers n'ayant pas de minimum, on peut construire une suite $(S_i)_i$ d'entiers infinie et strictement décroissante. On prend $S_0 = f(0)$. Comme S_0 n'est pas le minimum de f, il existe k tel que $f(k) < S_0$. On pose $S_1 = f(k)$. La suite $(S_i)_i$ est construite par récurrence sur $\mathbb{N}$ en suivant ce procédé. On conclut alors à une contradiction, car une telle suite ne peut exister sur les entiers. Une telle fonction n'existe donc pas. En utilisant le tiers exclus, on en conclut que toute fonction des entiers dans les entiers admet un minimum (de *"non non A"*, on déduit "A"). Maintenant, étant donné une fonction des entiers dans les entiers, cette preuve ne permet pas de savoir quel est son minimum. D'ailleurs, aucune preuve ne le permet, car ce problème est indécidable.

En fait, le tiers exclus privilégie la *validité* sur la *prouvabilité*. La validité est une notion "abstraite" qui s'intéresse à la vérité d'une proposition à l'intérieur des *modèles* du système logique, c'est-à-dire les "théories mathématiques" dans lesquelles les axiomes du système sont vrais. Au contraire, la prouvabilité est une notion "concrète", qui correspond à la possibilité d'établir la preuve d'une proposition. On exige que les systèmes logiques soient *corrects* : toute proposition prouvable doit être valide dans tous les modèles. En général, les systèmes logiques sont aussi *complets* : une propriété valide dans tous les modèles est prouvable. Par contre, Gödel a montré (cf. [Nag89]) que si on s'intéresse à un modèle particulier suffisamment compliqué (contenant l'arithmétique) alors tout système logique est nécessairement *incomplet* vis-à-vis de ce modèle : il y a des propositions vraies dans le modèle, qui ne sont pas prouvables dans le système logique (et leur négation encore moins, puisqu'elles sont vraies). En particulier, la cohérence [1] d'un système logique cohérent n'est pas démontrable à l'intérieur de ce système logique.

Ainsi une proposition de la forme *"A ou non A"* est valide dans tous les modèles, mais il existe a priori des modèles dans lequel "A" est valide, et d'autres dans lequel *"non A"* est valide. Utiliser le tiers exclus ne crée pas d'absurdité, mais c'est utiliser un argument "transcendant". Cette "transcendance" a un gros inconvénient : elle brise la correspondance entre les opérateurs logiques du langage et ceux du méta-langage. En effet, le fait que *"A ou B"* soit valide (dans tous les modèles) n'implique pas que "A" soit valide ou que "B" soit valide. Dans un système logique utilisant le tiers exclus, prouver *"A ou B"* ne signifie pas que l'on sait prouver "A" ou que l'on sait prouver "B".

Au contraire, en enlevant le tiers exclus de la logique, on se concentre uniquement sur la notion de prouvabilité, et on retrouve une correspondance entre les opérateurs du langage et ceux du méta-langage. Ainsi prouver *"A ou B"* signifie que l'on sait prouver "A" ou que l'on sait prouver "B". De plus, une preuve de *"A ou B"* indique laquelle de ces deux propositions elle démontre. De même, prouver une proposition de la forme *"il existe a tel que..."* signifie que l'on sait construire

[1] Un système est cohérent s'il n'existe pas de proposition qui soit valide (dans tous les modèles), et dont la négation soit aussi valide. Dans un système non cohérent, la notion de validité n'a pas de sens, car toutes les propositions (et leur négation) sont valides.

un tel a.

La logique ainsi définie s'appelle *logique intuitionniste* ou encore *logique constructive*. Du point de vue de la logique classique, la logique intuitionniste lui est équivalente : pour toute proposition démontrable en logique classique, il existe[2] une proposition intuitionniste démontrable qui lui est classiquement équivalente. Par exemple, sur le problème du minimum des fonctions d'entiers, cette proposition intuitioniste serait *il n'existe pas de fonction des entiers dans les entiers qui n'admette pas de minimum*. La logique intuitionniste est donc plus fine que la logique classique : elle distingue des propositions que la logique classique considère comme équivalentes.

La logique constructive est finalement un bon candidat en tant que langage de programmation certifiée. Les spécifications des programmes seront des propositions de cette logique, et les programmes seront des preuves. Cette analogie entre preuves et programmes est ce qu'on appelle *l'isomorphisme de Curry-Howard*. Un avantage de raisonner en logique constructive est, en outre, que les spécifications des programmes sont elles-mêmes des théorèmes de la logique : associer un programme à une telle spécification, c'est prouver cette spécification. Ainsi, on n'a pas besoin de montrer la cohérence de ces spécifications, car cette cohérence est assurée par la cohérence du système logique. Ceci ne reste bien sûr valable que si l'utilisateur n'ajoute pas d'axiomes à la logique. Si l'utilisateur ajoute un axiome, c'est à lui de vérifier que celui-ci n'est pas incohérent avec le système logique.

3.1.2 Les preuves comme données informatiques

Un aspect fondamental et fondateur de l'informatique est que les programmes informatiques sont eux-mêmes des données informatiques. Ainsi, un système d'exploitation ou un compilateur sont des programmes qui manipulent d'autres programmes. Une des conséquences de l'adage *"les preuves sont des programmes certifiés"* est que les preuves doivent elles-mêmes devenir des données : des preuves vont raisonner sur d'autres preuves. D'un point de vue logique, cela signifie que les preuves deviennent un objet du langage. En mathématique, ce n'est pas le cas jusqu'ici : les preuves appartiennent au métalangage des mathématiciens. Cela implique que la programmation certifée doit inventer vraiment sa propre logique, car celle usuelle en mathématique n'est pas suffisante.

Avoir les preuves comme données informatiques présente en fait de nombreux avantages. Par exemple, la communicabilité informatique des preuves : un individu A qui a écrit une preuve pourra la donner sur un support électronique, à un individu B qui n'aura alors qu'à passer cette preuve à son vérificateur de preuves favori. L'individu B sera alors sûr de la preuve que A lui a donné, et pourra en extraire un programme certifié. La preuve peut en fait être vue comme une compilation partielle de la spécification qui contient les informations permettant de garantir qu'elle vérifie cette spécification. L'étape de compilation suivante est d'éliminer les informations inutiles lors de l'exécution... Autre exemple, un programme pas forcément certifié pourra en même temps qu'il calcule un résultat, produire une preuve de la correction de ce résultat. Cette preuve sera alors donnée à un vérificateur automatique qui pourra garantir ce résultat. Ce genre de programme a surtout un intérêt si la vérification du résultat est un calcul suffisament rapide.

Coq utilise une autre application de cette idée. La distribution de Coq fournit en effet des programmes, appelés tactiques, qui construisent des preuves automatiquement pour certains types de problèmes spécialisés. Beaucoup de ces programmes, écrits en OCaml, ne sont pas eux-mêmes certifiés, mais cela n'est pas très grave, car les preuves qu'ils construisent seront vérifiées par Coq avant d'être acceptées. En fait, Coq laisse ainsi l'utilisateur écrire ses propres tactiques en OCaml : c'est sans danger pour l'utilisateur, même si les tactiques écrites ont des bogues et retournent des

[2] C'est un "il existe" constructif : il y a un algorithme qui trouve cette proposition.

preuves non correctes, celles-ci seront rejetées par Coq. La partie vitale de Coq, qui doit être sans bogue pour garantir le caractère certifié des programmes, n'est donc en fait qu'un tout petit noyau, qui comprend le langage logique, et le vérificateur de preuve. Dans le reste de Coq, qui comprend en particulier l'aide à la construction de preuves, les bogues éventuels ne mettent pas en péril la cohérence logique du système et des spécifications de l'utilisateur.

3.1.3 Calcul et déduction

Dès que l'on fait des preuves formelles, on doit écrire chaque étape de raisonnement. On a donc envie de profiter, à l'intérieur de la logique, de la notion de calcul associé aux preuves. Par exemple, ayant défini l'addition sur les entiers, et prouvé ses propriétés (associativité, commutativité, etc), on n'a pas envie d'avoir à fournir une démonstration pour dire que $2 + 3 = 5$. On aimerait que ce fait soit directement obtenu par calcul en utilisant la définition de l'addition.

Bien que cela puisse sembler évident, cette idée n'est pas courante dans les preuves mathématiques usuelles, où on a tendance à confondre étapes de calcul et étapes de déduction (ou raisonnement). Cela vient du fait que l'on veut faciliter les étapes de calcul pour les lecteurs, et ne leur laisser que vérifier ces calculs. Mais a priori, dans une preuve mathématique on pourrait se contenter de donner le résultat des calculs, car étant donner leur caractère *automatisable*, n'importe quel lecteur peut les reproduire (à condition que ces calculs ne soient pas trop longs).

Dans le cadre de preuves formelles sur ordinateur, le point de vue est différent : on veut justement profiter de la puissance de calcul des ordinateurs pour ne pas avoir à se préoccuper de calcul. On va ainsi chercher à remplacer ce qui s'apparente usuellement à de la déduction, par du calcul (voir [Dow99a] et [Dow98]). Par exemple, on va donc préférer donner une règle de calcul, plutôt qu'un axiome. C'est un premier pas vers une semi-automatisation des démonstrations. C'est aussi une sorte de mécanisme de compression des preuves : les preuves deviennent plus compactes, mais un peu plus longues à vérifier par l'ordinateur, car il doit faire des calculs.

On peut ainsi utiliser Coq pour formaliser une procédure de décision, puis utiliser ensuite cette procédure de décision pour démontrer des théorèmes en Coq. Cette technique est appelée *réflexion* : on *réfléchit* la procédure de décision dans la logique (voir [Bou97]).

3.1.4 Problème de la correction du vérificateur automatique de preuves

Un des problèmes des méthodes formelles est de garantir les outils informatiques qu'elles proposent. Ici, il faut certifier le noyau vital dont on parlait précédemment, et garantir en particulier le programme chargé de la vérification des preuves. Cependant, une des conséquences du théorème de Gödel est qu'on ne pourra pas prouver la cohérence de la logique de l'assistant à la démonstration à l'intérieur même de cet assistant. Cette preuve, ou une partie de cette preuve, sera donc métathéorique. Ceci dit, ce n'est pas forcément grave, car la cohérence de la logique est seulement nécessaire pour garantir la cohérence *de la spécification de l'assistant*. Cela n'empêche pas forcément de démontrer la correction de *l'implantation de l'assistant*, vis-à-vis de sa spécification, avec cet assistant. En outre, il est raisonnable d'espérer que s'il y a peu de bogues dans le premier assistant (non certifié), le deuxième assistant décrit dans ce premier assistant n'aura pas ces bogues. En effet, il est peu probable que la personne écrivant le deuxième assistant dans le premier, utilise les bogues du premier pour commettre des bogues dans le second. Par exemple, lorsque Russel a trouvé une contradiction dans la théorie des ensembles formulée jusqu'alors, les mathématiques ne se sont pas écroulées : peu de théorèmes utilisaient vraiment cette contradiction.

Ces constats amènent à penser que ce noyau devra être aussi simple que possible, tout en restant suffisamment expressif. Il faut en effet d'une part que ce noyau puisse être exprimé dans l'assistant, d'autre part que l'on puisse être convaincu par la preuve de cohérence du système logique. L'avantage d'avoir un noyau simple est aussi que "chacun" peut le programmer chez soi, pour vérifier les preuves du voisin (y compris la preuve de correction de son assistant...).

3.2 Présentation de Coq

Coq est construit suivant les principes ci-dessus. Le cœur du système est un langage fonctionnel typé, avec peu de constructions, mais extrêment puissantes : le calcul des constructions inductives (voir [Coq85], [Coq88], [PM89] et [Coq90]).

Coq a de plus une propriété très forte : tous les programmes qu'on peut exprimer en Coq terminent. Par un résultat célèbre, on sait donc que toutes les fonctions *calculables* ne sont pas exprimables en Coq. Mais ces fonctions sont plutôt rares en pratique, et on peut considérer que Coq est suffisamment expressif, au moins d'un point de vue théorique.

Cette restriction est importante du point de vue de la cohérence de la logique. En effet, on a dit que les calculs effectués par les programmes pouvaient avoir lieu lors de la vérification des preuves. Avoir des programmes qui ne terminent pas signifierait donc que l'on pourrait écrire des preuves impossibles à vérifier, car prenant un temps de vérification infini.

Autour de ce noyau, Coq procure des bibliothèques de définitions pour construire les notions de base du monde mathématique ou informatique (les entiers, les listes, etc) et des bibliothèques de tactiques qui aident l'utilisateur à construire des preuves ou lui fournissent du sucre syntaxique. Coq peut aussi extraire des preuves vers du code source OCaml. Inversement des tactiques Coq permettent de prendre des programmes écrits dans un sous-langage de OCaml et engendrent des obligations de preuves, afin de transformer ces programmes en preuves Coq.

Le noyau de Coq, écrit en OCaml, est sur le point d'être certifié en Coq (voir [Bar99]).

3.2.1 Le calcul des constructions inductives

Comme pour OCaml, on ne va pas donner une description formelle de ce langage, mais plutôt intuitive. On recommande au lecteur de se reporter sur [Bar00] et [Wer94] pour avoir une présentation plus approfondie. Le lecteur trouvera une présentation intuitive de Coq dans [Hue99].

Suivant le point de vue qu'on adopte, mathématique ou informatique, les types correspondent en fait respectivement aux formules mathématiques ou aux spécifications des programmes. De même, les termes correspondent respectivement à des preuves ou des programmes. La notion de base est la fonction, dont la syntaxe est $[x : A]b$. Ce qui correspond à `(fun (`x` :`A`) ->` b`)` en OCaml. Il faut noter que les types des variables doivent être explicités par l'utilisateur contrairement à OCaml : il n'y a pas d'inférence de type sur les termes non annotés (car elle n'est pas décidable pour Coq). L'application se note comme en OCaml, ainsi `([x :nat]x O)` dénote l'application de la fonction identité sur les entiers à `O`.

3.2.1.a Isomorphisme de Curry-Howard

Le tableau ci-dessous montre l'isomorphisme de Curry-Howard, avec de la syntaxe Coq :

Type	Intuition Logique	Termes associés
$A \to B$	A implique B	$[x : A]b$ avec $x : A \vdash b : B$
$(x : A)P$	$\forall x \in A, P$	$[x : A]b$ avec $x : A \vdash b : P$
$A \wedge B$	A et B	$(conj\ a\ b)$ avec $a : A$ et $b : B$
$A \vee B$	A ou B	$(or\_introl\ a)$ avec $a : A$, ou de $(or\_intror\ b)$ avec $b : B$
$Ex\ [x : A]P$	$\exists x \in A, P$	$(ex\_intro\ x\ b)$ avec $x : A$ et $x : A \vdash b : P$

En fait, la seule construction primitive au niveau des types ci-dessus est le type fonctionnel $(x : A)P$.[3] Toutes les autres constructions sont définies à l'intérieur du système, via des constructions primitives : les types inductifs.

Avant de considérer ces types inductifs, on va préciser un peu les particularités des types en Coq. En particulier, les types Coq sont eux-mêmes des termes. Ceci procure une grande expressivité au langage : les fonctions peuvent être paramétrées par des types (c'est du polymorphisme) ou retourner des types.

3.2.1.b Hiérarchie d'univers

Ainsi, les types ont eux-mêmes des types. Les types de types (appelés les sortes) sont désignés par le mot-clef `Type`. Mais d'un point de vue logique `Type` ne représente pas un type, mais des types. Chaque occurence de `Type` contient un indice implicite i géré par Coq : il faut la voir comme un $\texttt{Type}_i$. Cet indice, un entier naturel, est ce qu'on appelle l'univers de ce type. Il y a un nombre infini (dénombrable) d'univers. Et $\texttt{Type}_i$ est de type $\texttt{Type}_j$ si $j > i$. Coq stocke ces contraintes sur les univers pour vérifier qu'elles restent cohérentes. Ces contraintes assurent en effet qu'on ne puisse pas créer de contradiction de la forme *"l'ensemble de tous les ensembles"*. Comme ces contraintes sont implicites, l'utilisateur n'a pas trop a s'en soucier, et il peut avoir l'impression que `Type` est une sorte de type `Type`. Mais, parfois, un message d'inconsistance d'univers (qui s'apparente à une erreur de typage) le ramènera à la dure réalité.

En fait, les sortes désignées par `Type` ne sont pas les seules sortes. Il y a deux autres sortes : `Set` et `Prop`. Ces deux sortes ont un rapport avec le mécanisme d'extraction. En effet, extraire un programme d'une preuve, signifie en fait, éliminer toutes les informations de la preuve qui ne sont pas calculatoires. Mais comment reconnaître ces informations ? En Coq, c'est l'utilisateur qui met cette information, via les sortes qu'ils donnent à ses types. Les types de sorte `Set` seront considérés comme des *types informatifs*, dont il faut garder les termes à l'extraction. Les types de sorte `Prop` sont considérés comme des propositions, c'est-à-dire des types *non informatifs*, dont il faut éliminer les termes à l'extraction.

Par exemple, le type des entiers naturels (type `nat`) est considéré comme un type informatif. Ici `O` est zéro, et `S` est successeur. Voilà leur type en Coq.

$$\texttt{O :nat} \quad \texttt{S : nat -> nat} \quad \texttt{(S (S O)) : nat}$$
$$\texttt{nat : Set} \quad \texttt{nat -> nat : Set}$$
$$\texttt{Set : Type}_0 \quad \texttt{Type}_0\texttt{ : Type}_1$$

On a ici mis les indices aux sortes `Type` afin d'éclairer le lecteur, mais on rappelle qu'en syntaxe Coq, ces indices sont implicites.

De la même manière, `True` "la proposition vraie", est un type non informatif. Ce type a en fait une seule valeur : `I` la preuve du vrai. Le type `False` est, lui, un type vide : il n'en existe aucune preuve.

[3] $A \to B$ est une juste une abréviation de $(x : A)B$ quand x n'est pas une variable libre de B.

`I :True   True : Prop   False : Prop   Prop :` $\texttt{Type}_0$

La conjonction (symbole `/\`) est un opérateur binaire sur les propositions. Son équivalent dans l'isomorphisme de Curry-Howard est le produit cartésien :

```
/\: Prop -> Prop -> Prop
```

Remarquons que $\wedge$ est un symbole infixe. Ainsi, le type `(A :Prop)(True /\ A) -> A` a la sorte `Prop`. Ce type est celui des fonctions qui prennent une proposition `A`, et une preuve de `True /\ A` et retournent une preuve de `A`.

3.2.1.c Polymorphisme et types dépendants

Le type précédent est en fait polymorphe sur le type `A`. Il correspond[4] au type OCaml `(unit * 'a) -> 'a`, où `unit` est le type avec une seule valeur. Le polymorphisme de Coq est cependant strictement plus général celui de OCaml, car en Coq, on peut quantifier à l'intérieur des types. Par exemple le type `(A,B :Prop)((C :Prop)A -> C)-> A -> B` n'a pas d'équivalent en OCaml. Une fonction de ce type est par exemple `[A,B :Prop ; H :((C :Prop)A-> C) ; H' :A](H B H')`.

Une autre forme de supériorité du typage de Coq sur OCaml se trouve dans les types dépendants. Par exemple, étant donnée la fonction polymorphe d'égalité `eq` de type `(A :Set)A -> A -> Prop`. On peut construire le type qui dit que tout successeur d'un entier quelconque n'est pas zéro. C'est le type `(x :nat)(eq nat O (S x))-> False`. Ce type est celui des fonctions qui prennent un entier `x` et une preuve que `O` égale `(S x)`, et retourne un terme de type `False` (type vide) : autrement dit, l'absurde. Le terme `x` n'est ici pas un type, mais le type du deuxième argument de ces fonctions dépend de cet argument `x`. C'est ce qu'on appelle *un type dépendant*.

Avec une même syntaxe, on réunit donc ici deux notions historiquement disjointes : les types dépendants (quantification sur des termes) et polymorphisme (quantification sur des types).

3.2.1.d Types inductifs

Les types inductifs de Coq (voir [PM93a]) ressemblent en fait aux types récursifs de OCaml. Ils sont définis via la déclaration d'une suite finie de constructeurs. Les *valeurs* de ces types sont les termes librement engendrés par ces constructeurs. Autrement dit, ces types sont *"les plus petits types engendrés par ces constructeurs"*.

Présentation intuitive On définit un type inductif, par le mot-clef `Inductive`, on donne ensuite son nom, puis sa sorte. On donne ensuite la suite de déclarations des constructeurs, séparées par des "`|`". A la différence de OCaml où le type de retour des constructeurs était implicite, il est ici explicite. Ainsi le type `nat` est défini par :

```
Inductive nat: Set := | O: nat | S: nat->nat.
```

Ce type déclare deux constructeurs : `O` et `S`.

Lors d'une telle définition le système définit des fonctions permettant de manipuler les types inductifs. Ces fonctions, appelées *schémas d'élimination* déterminent la sémantique de *"type librement engendré"*. Pour `nat`, le schéma `nat_rec` a le type ci-contre

```
(P:(nat->Set))
  (P O)
    ->((n:nat)(P n)->(P (S n)))
      ->(n:nat)(P n)
```

[4]Cette correspondance n'a ici rien avoir avec le mécanisme d'extraction.

Autrement dit, c'est une fonction qui prend quatre arguments : un prédicat (c'est-à-dire une fonction à valeur dans les types) `P` sur les entiers ; un terme de type `(P O)` ; une fonction qui à tout entier `n` et tout terme de `(P n)` renvoie un terme de type `(P (S n))` ; un entier `n`. La fonction `nat_rec` retourne alors un élément de `(P n)`. On reconnaît ici le classique schéma de récurrence.

Ce schéma est en fait un schéma calculatoire. Par exemple, on peut définir la fonction `plus` en utilisant `nat_rec` :

```
Definition plus: nat->nat->nat :=
  (nat_rec [n:nat]nat->nat [y:nat]y [m,p][y:nat](S (p y))).
```

Le mot-clef `Definition` dit que l'on veut définir `plus` dans l'environnement global. Le type `nat->nat->nat` est celui qu'on donne à `plus`. L'expression qui suit le symbole " `:=`" est l'expression à laquelle `plus` fera référence. Intuitivement, pour définir la fonction `plus`, on calcule par récurrence sur l'entier `n` la fonction "`(plus n)`".

On applique ainsi `nat_rec` avec le prédicat `[n :nat]nat->nat` comme instance de `P`. Pour le cas de `O`, on renvoie la fonction identité. Pour le cas `(S m)`, on renvoie la fonction qui à `y` associe `(S (p y))` où `p` est la fonction "`(plus m)`", calculée par récurrence. Ces lignes de Coq correspondent donc exactement aux lignes OCaml ci-contre.

```
type nat = | O | S of nat

let rec plus n =
  match n with
  | O -> (fun y -> y)
  | (S m) -> (fun y -> (S (plus m y)))
```

Si cette définition de `nat` est agréable pour raisonner par récurrence, elle n'est pas vraiment convaincante en tant que type de données de programmation. Regardons donc les listes. Elles sont données par le type Coq suivant :

```
Inductive list[A:Set]: Set := | nil: (list A) | cons: A->(list A)->(list A).
```

Le schéma `list_rec` a le type :

```
(A:Set; P:((list A)->Set))
  (P (nil A))
    ->((y:A; l:(list A))(P l)->(P (cons A y l)))
    ->(l:(list A))(P l)
```

Si on simplifie ce type pour en donner une version sans types dépendants, on obtient :

```
(A:Set; P:Set)P->(A->(list A)->P->P)->(list A)->P
```

Ce qui donne en OCaml : `'p -> ('a -> ('a list) -> 'p -> 'p) -> ('a list) -> 'p`.

C'est une fonction qui généralise la fonction `List.fold_left` du chapitre précédent. En OCaml, ce schéma `list_rec` s'écrit comme ci-contre. Il permet d'écrire des fonctions sur les listes en toute généralité.

```
let rec list_rec a f l =
  match l with
  | [] -> a
  | x::m -> (f x m (list_rec a f m))
```

En fait pour `nat` (resp. `list`), Coq donne deux autres schémas d'élimination : `nat_ind` (resp. `list_ind`) et `nat_rect` (resp. `list_rect`). Les schémas suffixés en `ind` portent en fait sur `Prop` : ils changent la sorte de retour de l'argument `P` de `Set` en `Prop`. De même les schémas suffixés en `rect` portent en fait sur `Type` : ils changent la sorte de retour de l'argument `P` de `Set` en `Type`.

Intuitivement, les schémas suffixés en `ind` permettent donc de faire les raisonnements par récurrence usuels en mathématiques. Les schémas suffixés en `rec` permettent d'écrire les fonctions récursives usuelles en programmation fonctionnelle. Les schémas suffixés en `rect` donnent la sémantique de *"plus petit type librement engendré"*. Par exemple, si on veut démontrer que le successeur d'un entier n'est pas zéro, on commence par construire la fonction `discr` :

```
Definition discr: nat->Prop :=
  (nat_rect [_:nat]Prop True [_;_]False).
```

Le symbole "_" signifie ici comme en OCaml, qu'on ne souhaite pas donner de noms aux arguments (car on ne les utilise pas). Cette fonction `discr` prend un entier, et retourne la proposition `True` si cet entier est zéro, et `False` sinon. Elle permet donc de discriminer l'entier zéro des autres entiers. La fonction `eq` est elle-même définie par un type inductif qu'on donnera ultérieurement. Son schéma d'élimination `eq_ind` est le suivant :

```
(A:Set; x:A; P:(A->Prop))(P x)->(y:A)(eq A x y)->(P y)
```

On reconnaît ici la propriété de l'égalité de Leibniz : si deux termes sont égaux, alors ils vérifient les mêmes propriétés. Ce schéma `eq_ind` permet d'écrire maintenant le lemme désiré :

```
Definition discr_O_Sn: (n:nat)(eq nat O (S n))->False
 :=[n:nat](eq_ind nat O discr I (S n)).
```

En effet, il est facile de voir que `I` étant un terme de type `True`, le terme `[n :nat](eq_ind nat O discr I (S n))` est de type `(n :nat)O=(S n)->(discr (S n))`. On remarque au passage les deux calculs qu'il faut faire pour s'assurer que le terme associé à `discr_O_Sn` a bien le type qu'on veut lui donner : il faut d'une part calculer `(discr O)` pour vérifier que `I` est de ce type, et d'autre part calculer `(discr (S n))` à `n` fixé, pour vérifier qu'il est égal à `False`.

Le schéma `nat_rect` est ce qu'on appelle une *élimination forte* : il permet de construire par récurrence (ou par cas sur les entiers), des termes vivant dans un univers au-dessus de celui des entiers. Par exemple, le type `nat` est de type `Set`, et dans `discr` on construit des propositions, c'est-à-dire des termes de type `Prop` où `Prop` est de type $\texttt{Type}_0$. On va voir que ce type de schéma n'est pas toujours autorisé, pour diverses raisons.

Remarquons maintenant que certains types inductifs Coq n'ont pas leur équivalent en OCaml, en particulier en utilisant les types dépendants. Par exemple, le type `finiteSet` suivant, paramétré par un entier `n`, construit une famille de types telle que `(finiteSet n)` est un type ayant exactement `n` valeurs distinctes.

```
Inductive finiteSet: nat -> Set :=
  | base: (n:nat)(finiteSet (S n))
  | lift: (n:nat)(finiteSet n)->(finiteSet (S n)).
```

Pour s'en convaincre, on peut faire un raisonnement par récurrence (dans la métathéorie). Pour `(finiteSet O)`, il n'y a aucun constructeur qui permette de construire une valeur de ce type. Le type `(finiteSet O)` est donc vide. Si `(finiteSet n)` contient exactement `n` valeurs distinctes, on peut encapsuler ces `n` valeurs distinctes via le constructeur `lift` et donc construire `n` valeurs distinctes dans `(finiteSet (S n))`. On peut en construire une de plus dans `(finiteSet (S n))`, en utilisant le constructeur `base`. On conclut alors en remarquant qu'il n'est pas possible de construire d'autres valeurs dans `(finiteSet (S n))`.

Remarquons que l'argument dans la définition de type `finiteSet`, n'a pas le même comportement que celui de `list`. En fait la définition inductive de `list` n'a pas lieu sur `list`, mais sur `(list A)` : on définit inductivement un type constant. Dans `finiteSet`, on définit au contraire inductivement une fonction à valeur dans les types.

Concluons cette présentation "intuitive" des types inductifs, en insistant sur la force qu'ils donnent à ce calcul. En effet, les types inductifs permettent à l'utilisateur de définir une infinité de nouveaux types, et leurs schémas de récurrence associés. Ces schémas sont montrés (dans la métathéorie de Coq) cohérents avec le reste du langage, et l'utilisateur peut donc introduire ces types sans soucis. Dans la pratique, étant donné l'expressivité de ces types, l'utilisateur n'a en général pas réellement besoin d'introduire des axiomes pour construire ses théories. Plus effectifs que des axiomes, ces schémas d'élimination sont des fonctions génériques de calcul sur les types de l'utilisateur.

Quelques restrictions Il faut noter ici que si les types inductifs généralisent certains aspects des types récursifs de OCaml, ils en restreignent la récursivité. En effet, la pleine récursivité dans ces types permettraient d'écrire des fonctions qui ne terminent pas. Par exemple, le type OCaml :

```
type loop = Zouh of (loop -> unit)
```

permet d'écrire la fonction `build` de type `loop -> unit` :

```
let build x = match x with
  | Zouh f -> f (Zouh f)
```

Cette fonction ne termine pas quand on lui passe l'argument `(Zouh build)`.

Les restrictions sur la récursivité en Coq se basent uniquement sur des critères syntaxiques : les occurrences du type inductif dans ses constructeurs doivent être *positives*. Cela signifie que d'une part, les constructeurs ne peuvent pas avoir des arguments fonctionnels dont les arguments soient du type qu'on est en train de définir. D'autre part, le type qu'on est en train de construire ne peut pas être passé lui-même comme argument à un de ses paramètres.

Pour éviter des paradoxes, il y a des aussi des restrictions portant sur les schémas d'élimination (cf. [Wer94] et [Coq86]). Pour garantir la correction de l'extraction, il faut interdire la construction de termes informatifs par un raisonnement par cas sur un terme non-informatif. Comme à l'extraction le terme non-informatif disparaît, on ne saurait pas choisir entre les différentes branches d'exécution d'un raisonnement par cas. Ainsi le seul schéma d'élimination autorisé pour les types inductifs à plusieurs constructeurs définis dans `Prop` est celui suffixé par `ind`. Par exemple, l'opérateur `or` du "ou" logique est défini par :

```
Inductive or[A,B:Prop]:Prop :=
  | or_introl : A->(or A B) | or_intror : B->(or A B).
```

Le seul schéma d'élimination est `or_ind`. On ne peut donc pas faire un raisonnement par cas sur un terme de type `(or A B)` pour construire un terme de type `nat`. Il faut remarquer aussi que l'élimination pour les types inductifs définis dans `Prop` est non-dépendante. Cela traduit que usuellement on ne distingue pas les termes non-informatifs de même type (c'est-à-dire les "preuves" d'un même théorème) : on considère tous ces termes comme équivalents. Ce n'est bien sûr pas le cas sur un type informatif comme `nat`. On ne construit donc en général pas de proposition dépendant d'un terme non-informatif. Ainsi `or_ind` est de type :

```
(A,B,P:Prop)(A->P)->(B->P)->(or A B)->P
```

Une version avec types dépendants de ce schéma aurait été :

```
(A,B:Prop; P:(or A B)->Prop)
        ((y:A)(P (or_introl A B y)))
        ->((y:B)(P (or_intror A B y)))
        ->(o:(or A B))(P o)
```

Une deuxième contrainte sur les schémas d'élimination forte est qu'on ne doit pas pouvoir construire un type U de sorte S, tel qu'il existe *encap* de type S->U et *decap* de type U->S tels que (*decap* (*encap* x)) se réduit sur x. Intuitivement, ce genre de type permet de violer la hiérarchie d'univers, car on peut encapsuler x de type S, dans un terme de type U (U étant de sorte S) puis de récupérer ensuite ce x. Le problème se pose uniquement pour `Set` et `Type` puisque il n'y a pas d'élimination forte sur `Prop`.

Sur les types inductifs définis dans `Set`, on interdit les schémas d'élimination forte dès qu'il y a un constructeur dont un des arguments est de type `Type`. Par exemple, le type `FooS` suivant :

```
Inductive FooS: Set := | encapS: Prop -> FooS | bar: nat -> FooS.
```

n'a pas de schéma `Foo_rect`, il a seulement le schéma `FooS_ind` et `FooS_rec`. On ne peut donc écrire la projection correspondant à `encapS`. Par contre, on peut construire celle correspondant à `bar`.

Pour `Type`, au contraire, on autorise librement le schéma d'élimination. Par contre, on impose des contraintes d'univers sur les constructeurs afin d'éviter une encapsulation abusive.

```
Inductive FooT: Type := | encapT: Type -> FooT.
```

Ci-dessus, l'univers de l'argument de `encapT` est strictement inférieur à celui de `FooT`. Ainsi `(encapT FooT)` produit une erreur de consistance d'univers et est rejeté par Coq.

Les constructions `Fixpoint` et `Cases` Pour faciliter la tâche du développeur en Coq et avoir du code syntaxiquement plus proche de OCaml, les schémas d'élimination sont en fait codés via un opérateur de point fixe `Fixpoint` et un opérateur de filtrage `Cases`. Par exemple la fonction `plus` de tout à l'heure peut s'écrire :

```
Fixpoint plus[x:nat]:nat->nat :=
  Cases x of
   | O => [y:nat]y
   | (S n) => [y:nat](S (plus n y))
  end.
```

L'argument du point fixe sur lequel va se faire la récursion doit être indiqué entre crochet derrière le nom de la fonction. Les opérateurs `Fixpoint` et `Cases` ne sont pas l'équivalent du `let rec` et du `match` de OCaml. Ils ne permettent que d'écrire les fonctions qu'on aurait pu définir avec les schémas d'élimination. Remarquons que, comme l'unification d'ordre supérieure est indécidable, il faut parfois indiquer dans une construction `Cases` le prédicat `P` sur lequel on applique l'élimination, car le système n'est pas capable de l'inférer tout seul. Par exemple, essayons de construire la fonction `discr_O_Sn` de toute à l'heure avec un `Cases`. L'égalité est en fait définie par le type inductif :

```
Inductive eq [A:Set; x:A] : A->Prop :=  | refl_equal : (eq A x x)
```

Son schéma d'élimination (non dépendant) est `eq_ind` dont on a déjà donné le type ci-dessus. On reviendra ultérieurement sur cette définition. Pour l'intant, remarquons simplement que l'égalité n'a qu'un seul constructeur : `refl_equal`. On peut donc avoir envie de définir `discr_O_Sn` par :

```
Definition discr_O_Sn: (n:nat)(eq nat O (S n))->False
 :=[n:nat; H:O=(S n)]Cases H of | refl_equal => I end.
```

Mais cette définition est rejetée par Coq, car le système n'arrive pas à inférer le prédicat sur lequel on applique l'élimination. Trouver ce prédicat est en effet non trivial, puisque c'est `discr`. Il faut donc ici l'indiquer au système. Cet indication se donne entre `<` et `>` devant `Cases` :

```
Definition discr_O_Sn: (n:nat)(eq nat O (S n))->False
 :=[n:nat; H:O=(S n)]<discr>Cases H of | refl_equal => I end.
```

3.2.2 Utilisation de Coq

Le noyau de Coq n'est formé que de quelques constructions précédentes : abstractions (c'est-à-dire termes de la forme "$[x : A]b$"), applications, quantifications (c'est-à-dire termes de la forme "$(x : A)b$"), les types inductifs, et les sortes `Set`, `Prop` et `Type`. A partir de ces constructions, Coq définit les concepts de base en logique et de programmation, comme nous l'avons déjà suggéré ci-dessus. Quelques-uns de ces concepts sont présentés en 3.2.2.a. On passe ensuite à d'autres aspects de Coq : simplification de la syntaxe,

3.2.2.a Bibliothèque de base

On va surtout s'intéresser aux quelques notions logiques de base.

Les connecteurs de base

Le type `False` est un type inductif sans constructeur. Son schéma d'élimination dit que si on a un terme de type `False`, on peut en déduire n'importe quoi : `(P:Prop)False->P`. Remarquons que dans le cadre de Curry-Howard, ce schéma d'induction correspond à une instruction de levée d'exception (qui ne devrait jamais être exécutée).

```
Inductive False: Prop:=.
```

La négation est simplement donnée par la définition ci-contre. Ainsi une preuve de `(not A)` est une preuve que `A` conduit à une absurdité.

```
Definition not:=[A:Prop]A->False.
```

Le type `True` est un type inductif avec un constructeur constant `I`.

```
Inductive True : Prop := | I : True.
```

Le `and` logique est donné par un type inductif à un seul constructeur.

```
Inductive and[A:Prop; B:Prop]: Prop :=
  | conj : A->B->(and A B).
```

L'existentiel est aussi un type à un seul constructeur. Ainsi une preuve de l'existence d'un terme x qui vérifie $(P\ x)$ est un couple qui contient un témoin t et une preuve que t vérifie $(P\ t)$.

```
Inductive ex[A:Set; P:A->Prop]: Prop :=
      ex_intro : (x:A)(P x)->(ex A P)
```

L'égalité Quand on débute en Coq, le type le plus mystérieux reste sans doute celui de l'égalité. Comme on l'a vu tout-à-l'heure, la notion d'égalité n'est pas primitive en Coq, mais définie à l'aide d'un type inductif, et *correspond à l'égalité de Leibniz*. En un certain sens, elle *réflechit* en Coqla méta-égalité de Coq. Cette méta-égalité, qu'on appellera *interconvertibilité*, dit que 2 termes sont égaux si et seulement si ils se réduisent sur le même terme irréductible. Dans le cas de termes clos (sans variables libres), cela signifie que 2 termes sont interconvertibles s'ils représentent la même valeur. L'interconvertibilité est en fait décidable : l'algorithme de typage va ainsi pouvoir substituer un terme par un terme qui lui est interconvertible. On a en fait déjà observé ce phénomène lors du typage `discr_O_Sn`.

Ainsi, l'égalité de Leibniz en Coq est engendrée par l'axiome de réflexivité (voir ci-contre). Au lieu de `(eq A x y)`, Coq autorise la notation infixe `x=y` qui laisse `A` implicite.

```
Inductive eq [A:Set; x:A] : A->Prop :=
  | refl_equal : (eq A x x)
```

Sur les termes clos, l'égalité de Leibniz correspond exactement à l'interconvertibilité. De plus interconvertibilité implique égalité de Leibniz : er, si t_1 est interconvertible avec t_2, alors ils ont alors le même type T et `(refl_equal` T t_1`)` est de type `(eq` T t_1 t_2`)` (car ce dernier type est alors interconvertible avec `(eq` T t_1 t_1`)`).

Pour rester décidable, l'interconvertibilité est une égalité non-extensionnelle : deux fonctions produisant les mêmes résultats sur les mêmes arguments *ne sont pas forcément interconvertibles.* On dit de l'interconvertibilité qu'elle est intensionnelle : deux fonctions sont interconvertibles si et seulement si elles correspondent fondamentalement au *même* calcul. Cela a pour effet par exemple de distinguer une fonction calculant le tri d'une liste via un algorithme de tri par tas, d'une fonction implantant un algorithme de tri par insertion. Cette propriété d'intensionalité de l'interconvertibilité est réfléchie au niveau de l'égalité de Leibniz. On ne pourra donc pas prouver en toute généralité que deux fonctions calculant les mêmes valeurs sont égales. Remarquons que cela ne veut pas dire qu'on saura prouver le contraire ! De même, l'inégalité de Leibniz sur les termes clos (c'est-à-dire la possibilité de prouver que 2 termes clos ne sont pas égaux) ne correspond pas à la non-interconvertibilité. En effet, il existe des termes clos non interconvertibles dont on ne peut pas prouver (en Coq) qu'ils sont non-égaux pour Leibniz. Par exemple les termes `[A :Set ; f :(A->A)]f` et `[A :Set][f :A->A][x :A](f x)` ne sont pas interconvertibles car Coq n'a pas la règle de η-conversion (forme faible d'extensionalité) qui réduirait $[x : A](f\ x)$ sur f ou inversement. Mais on ne peut pas démontrer en Coq que ces deux termes ne sont pas égaux vis-à-vis de Leibniz, car aucune propriété ne permet de les distinguer (comme `discr` permet de distinguer `O` des successeurs). C'est une forme d'incomplétude.

Intuitivement, étant donné la fonction `plus` de tout-à-l'heure, le lemme "`(n :nat)(plus n O)=n`" fournit donc une preuve que pour toute instance close de `t` de `n`, `(plus t O)` se réduit sur la valeur que représente `t`. On remarque que si `(plus O n)` est interconvertible avec `n`, le terme `(plus n O)` ne l'est pas. Le lemme précédent est donc obtenu en effectuant une preuve par récurrence sur l'entier `n`.

Cette égalité de Leibniz n'est en faite pas toujours suffisante, dans le sens, où on peut avoir des termes dont les instances closes soient interconvertibles, mais dont on ne pas exprimer l'égalité (via Leibniz) car leur types ne sont eux-mêmes pas interconvertibles. Par exemple, en reprenant l'exemple de `finiteSet` précédent, on peut construire un terme `liftm` qui généralise `lift` en plongeant `(finiteSet n)` dans `(finiteSet (plus n m))` :

```
Fixpoint liftm[n:nat; f:(finiteSet n)]:(m:nat)(finiteSet (plus n m)):=
  <[n:nat; _:(finiteSet n)](m:nat)(finiteSet (plus n m))>Cases f of
   | (base x) => [m](base (plus x m))
```

```
  | (lift x f') => [m](lift (plus x m) (liftm x f' m))
  end.
```

On aimerait maintenant montrer que : “(n :nat ; f :(finitSet n))f=(liftm n f O)”. Mais cette expression est mal typée car (liftm n f O) est de type (finiteSet (plus n O)) qui est incompatible avec (finiteSet n).

On peut alors généraliser l'égalité précédente, par exemple avec :

```
Inductive depeq[A:Set;x:A]: (B:Set)B->Prop:= | refl_depeq: (depeq A x A x).
```

On peut alors démontrer “(n :nat ; f :(finiteSet n)) (depeq (finiteSet n) f (finiteSet (plus n O)) (liftm n f O))”. On appelera finiteSet_n_O ce lemme.

Toutefois, il faut remarquer que si on peut facilement montrer “(A :Set ;x,y :A)x=y->(depeq A x A y)”, la réciproque n'est pas démontrable.

Décidabilité En utilisant le caractère intuitionniste de la logique, la décidabilité est une notion très facile à formaliser en Coq : par exemple un prédicat P de type nat->Prop est décidable si pour tout entier x, on sait démontrer (P x) ou (not (P x)). Ceci s'exprime donc dans la formule : (x :nat)(or (P x) (not (P x))).

En fait, en programmation, on va souvent vouloir utiliser le contenu calculatoire d'une preuve de décidabilité. Par exemple sur la preuve précédente, ce contenu serait une fonction qui prendrait un entier x et retournerait un booléen indiquant si x vérifie P ou pas. En fait, pour utiliser ce contenu calculatoire à l'extraction, on n'exprime pas la décidabilité via or. On utilise plutôt sumbool qui est un type inductif défini comme or, mais à valeur dans Set :

```
Inductive sumbool [A:Prop; B:Prop] : Set :=
    | Left : A->(sumbool A B) | Right : B->(sumbool A B).
```

A l'extraction, l'information non-calculatoire contenue dans ces constructeurs disparaît. La version “extraite” de sumbool en OCaml, donnée ci-contre, est un type isomorphe au type bool de OCaml.

```
type sumbool = | Left | Right
```

3.2.2.b Simplification de la syntaxe

Pour faciliter l'écriture du code, et le rendre plus lisible, Coq fournit un certain nombre d'outils à l'utilisateur pour simplifier la syntaxe. On ne veut pas expliquer ici comment utiliser ces mécanismes, on veut juste mentionner leur existence, afin que le lecteur puisse lire les textes Coq donnés dans les chapitres suivants. Voir [Bar00] pour plus de détails.

Extension de la syntaxe L'utilisateur a par exemple la possibilité d'étendre la syntaxe Coq pour utiliser des opérateurs usuels : = pour eq, /\ pour and, \/ pour or, ~ pour not, { . }+{ . } pour sumbool, ... La décidabilité de l'égalité sur les entiers s'écrit ainsi (x,y :nat){x=y}+{~x=y}.

Exemple de la macro Structure Un autre exemple de ces possibilités d'extension de la syntaxe est la macro prédéfinie Structure qui simule des enregistrements en Coq. Cette macro permet de définir un *“type enregistrement”* S en déclarant les différents champs de ce type. En fait, le type S est défini comme un type inductif à un constructeur, appelé Build_S. Les champs des termes de type S sont accédées via des fonctions de projection. Ainsi, la définition :

```
Structure pair[A,B:Set]: Set := { fst: A; snd: B}.
```

équivaut à

```
Inductive pair [A:Set; B:Set] : Set :=  | Build_pair : A->B->(pair A B).

Definition fst := [A,B:Set; x:(pair A B)]Cases x of (Build_pair a _) => a end.

Definition snd := [A,B:Set; x:(pair A B)]Cases x of (Build_pair _ b) => b end.
```

Les enregistrements simulés ainsi ont cependant un gros défaut : deux types enregistrements distincts ne peuvent pas partager un même nom de champ.

Arguments implicites Les arguments implicites permettent d'éviter d'écrire les arguments redondants des fonctions ayant des types dépendants. Ces arguments sont en fait inférés, un peu comme pour le polymorphisme implicite de ML, lors de la vérification de type. L'utilisateur indique ces arguments implicites sous forme d'un `?` au vérificateur de type. Par exemple, le lemme `finiteSet_n_O` s'écrit alors : `(n:nat; f:(finiteSet n)) (depeq ? f ? (liftm ? f O))`.

Mieux, l'utilisateur a la possibilité de définir certaines fonctions en *mode "arguments implicites"*. Lors de l'application de ces fonctions, il n'est plus nécessaire de remplacer les arguments redondants par un point d'interrogation. Par exemple, reprenons l'exemple de `liftm` et `depeq` en mode argument implicite.

```
Fixpoint liftm[n:nat; f:(finiteSet n)]:(m:nat)(finiteSet (plus n m)):=
  <[n:nat; _:(finiteSet n)](m:nat)(finiteSet (plus n m))>Cases f of
   | (base x) => [m](base (plus x m))
   | (lift x f') => [m](lift (plus x m) (liftm f' m))
  end.

Inductive depeq[A:Set;x:A]: (B:Set)B->Prop:= refl_depeq: (depeq x x).
```

Le lemme `finiteSet_n_O` s'écrit alors : `(n :nat ; f :(finiteSet n))(depeq f (liftm f O))`.

Étant donné l'indécidabilité de l'unification en Coq, l'algorithme d'inférence des arguments implicites est incomplet : il faut parfois aider le vérificateur de type, en lui explicitant certains arguments "implicites". Ceci dit, le mécanisme d'arguments implicites reste très intéressant puisqu'il va permettre de raccourcir certaines expressions de façon logarithmique.

Coercive Subtyping Il existe en Coq un mécanisme de cœrcions implicites, qui permet de simuler une sorte de sous-typage. Ces fonctions de cœrcions sont très expressives, puisque quasiment n'importe quelle fonction entre "classes" de Coq peut être une fonction de cœrcion. Les classes de Coq sont grosso-modo les fonctions à valeurs dans les types (ie. le type codomaine est une sorte). En fait, comme pour le mécanisme d'arguments implicites, ces cœrcions implicites ne sont que du sucre syntaxique, et sont explicitées par le système lors de la vérification de type. En particulier, contrairement aux cœrcions usuellement employées dans les langages de programmation, ces cœrcions ont un coût à l'exécution. Elles permettent donc simplement d'alléger la syntaxe. Combinées au mode arguments implicites et à la macro `Structure`, ces cœrcions sont très agréables : elles ont permis une formalisation conséquente de la théorie des catégories ([Saï99]) et de l'algèbre de base ([Pot98]). Des généralisations de ce mécanisme sont à l'étude dans [Sol99b] et [Sol99a].

Plus formellement, une classe de Coq est n'importe quel terme C dont le type est de la forme $(x_1 : A_1)\dots(x_n : A_n)s$ où s est une sorte. Un objet de cette classe sera alors n'importe quel terme dont le type est de la forme $(C\ t_1 \ldots t_n)$. Si C et D sont deux classes, une fonction de cœrcion de C vers D est alors n'importe quelle fonction f dont le type est de la forme

$$(x_1 : A_1) \ldots (x_n : A_n)(y : (C\ x_1 \ldots x_n))(D\ u_1 \ldots u_m)$$

On notera $f : C$`>->`D. Dans un contexte où le typeur attend un objet de classe D, il transformera un terme t de type $(C\ t_1 \ldots t_n)$ en $(f\ t_1 \ldots t_n\ t)$.

Par exemple, on peut transformer le constructeur `base` de type `(n :nat)(finiteSet (S n))`, comme une cœrcion. Cela s'écrit par la commande suivante :

```
Coercion base: nat >-> finiteSet.
```

Sous l'hypothèse que `liftm` a été définie dans le mode argument implicite, alors `(liftm (S O) (S (S (S O))))` est typable et a le type `(finiteSet (plus (S (S O)) (S (S (S O)))))`. Ainsi ce terme a été transformé en `(liftm (base (S O)) (S (S (S O))))`

En combinant ce mécanisme avec la macro `Structure`, on a alors une sorte d'héritage au niveau de ces enregistrements.

```
Structure triple[A,B,C:Set]: Set := { triple2pair:> (pair A B); thd: C }.
```

Ci-dessus le `:>` déclare la projection `triple2pair` comme une cœrcion. Autrement dit, `triple2pair` de type `(A,B,C :Set)(triple A B C)->(pair A B)` est une cœrcion de `triple >-> pair`. Sous l'hypothèse que les définitions de `pair` et `triple` ont été faite dans le mode arguments implicites, `(snd (Build_triple (Build_pair O (base O)) (S O)))` est typable, et de type `(finiteSet (S O))`. On a ainsi l'impression que `snd` peut aussi être appliqué sur les enregistrements de type `triple`. Cette forme d'héritage est donc un moyen de faire du partage entre des champs de types enregistrement distincts. Ce partage peut être assez fin, en utilisant le polymorphisme de Coq. Cette méthode de partage des champs des enregistrements peut cependant vite devenir assez lourde.

On remarque qu'il y a une condition (appelée condition d'héritage uniforme) qui restreint les possibilités des fonctions de cœrcions : une fonction de type

$$(x_1 : A_1) \ldots (x_n : A_n)(y : (C\ u_1 \ldots u_n))(D\ v_1 \ldots v_m)$$

ne peut pas être directement une fonction de cœrcion. Il faudra en fait avoir recours à une classe C' définie par $[x_1 : A_1] \ldots [x_n : A_n](C\ u_1 \ldots u_n)$ et une fonction de cœrcion identité de C' vers C. f sera alors considérée comme une cœrcion de C' vers D. Ceci est largement détaillé dans [Bar00] et [Saï99].

Il y a aussi deux autres classes en Coq, dites abstraites, car on ne peut les utiliser que comme codomaine d'une cœrcion. Ces classes sont `SORTCLASS` et `FUNCLASS` qui permettent respectivement de considérer une classe comme une sorte, ou comme un type de fonction. Par exemple,

```
Structure foo: Set := { C:> nat; misc:> (n:nat)(finiteSet (plus n C))}.
```

La définition ci-dessus déclare `C` de type `foo->nat` ci-dessus comme une cœrcion de `foo>->nat`, et `misc` de type `(x :foo ; n :nat)(finiteSet (plus n (C x)))` comme une cœrcion de `foo>->FUNCLASS`. Ainsi, si `t` est de type `foo`, `t` peut-être utilisé à la place de la fonction `(misc t)`. Ainsi, le terme `[x :foo](x x)` est typable, et est de type `(x :foo)(finiteSet (plus x x))`. Il signifie en fait `[x :foo](misc x (C x))`.

Ce mécanisme de cœrcion est donc un moyen puissant de faire des “abus de notations”, mais qui doit être utilisé avec bon sens, sinon il obscurcit le code plus qu'il ne le simplifie.

3.2.2.c Constructions de preuves

Outre le calcul des constructions, Coq procure un mode interactif de construction de preuves. Ce mode fournit un langage de tactiques qui aident l'utilisateur à construire des preuves. Sans ces tactiques, la construction des preuves serait vraiment très difficile et très pénible. L'utilisateur a d'ailleurs la possibilité d'écrire lui-même de nouvelles tactiques en OCaml. Le principe est que ces tactiques construisent un terme Coq qui est vérifié par le typeur de Coq. Même si les tactiques de l'utilisateurs ont des bogues, ceux-ci ne risquent donc pas d'introduire d'incohérence dans le système. Au pire, ils font perdre du temps aux utilisateurs qui croient avoir construit une preuve, qui est finalement rejetée par le système.

On va présenter rapidement ce langage, en s'attardant sur la tactique dite *d'inversion*, car cette tactique aide à faire des raisonnement par cas sur les types inductifs, mais est parfois délicate à manier en présence de type dépendants. Or le code présenté dans les chapitres suivants utilisent beaucoup de types dépendants.

Présentation générale Ce mode interactif de construction de preuves fonctionne suivant un principe *de chaînage arrière*. On part de la formule à prouver (ou du type dont il faut construire un terme). Puis via des invocations de tactiques, on "simplifie" cette formule jusqu'à trouver des formules triviales, c'est-à-dire pour lequel l'utilisateur ou le système sait trouver une preuve. Chaque étape de simplification peut faire apparaître plusieurs nouvelles formules à prouver : typiquement, des hypothèses à décharger pour que l'étape de simplification soit valide. Chaque étape de simplification peut amener aussi de nouvelles hypothèses dans le contexte.

En fait, une telle preuve peut être vue comme un arbre, dont la racine est la formule à prouver, et les feuilles, les hypothèses auxquelles on s'est ramenées, et qu'on a finalement prouvées. Dans cet arbre, les nœuds s'appellent aussi des *buts*, et les fils d'un nœud s'appellent des *sous-buts*.

Par exemple, supposons le but suivant à prouver :

```
P : nat->nat->Prop
z : nat
H1 : z=0
H2 : (x,y:nat)~x=(S 0)->(~z=0->x=z)->(P x y)
============================
 (P (S (S 0)) (S 0))
```

Coq indique ici les hypothèses au-dessus du double trait, et le but à prouver en-dessous. Pour prouver ce but, l'idée est d'appliquer l'hypothèse H2, ce qui se dit en Coq "`Apply H2`". Cela donne 2 sous-buts qui, sous les mêmes hypothèses sont : `~(S (S 0))=(S 0)` et `~z=0->(S (S 0))=z`. Pour prouver ce dernier sous-but, il faudra alors *introduire* l'hypothèse `~z=0` (c'est-à-dire placer `~z=0` en hypothèse), et montrer la contradiction entre `H1` et cette dernière hypothèse.

Remarquons que les tactiques ne sont souvent que des heuristiques : le problème qu'elles tentent de résoudre est souvent indécidable. Elles peuvent donc ne pas réussir la simplification demandée par l'utilisateur. Lorsqu'une tactique échoue, elle rend en général la main à l'utilisateur sans changer le but,[5] en affichant un message d'erreur expliquant la cause de son échec. L'utilisateur doit alors trouver d'autres moyens d'arriver à ses fins. Bien sûr, il se peut que la cause profonde de l'échec d'une tactique, vienne de la non-prouvabilité du but de l'utilisateur, ou d'une erreur dans ce but.

[5] C'est au concepteur de la tactique d'en décider

Descriptions de quelques tactiques La tactique `Intros` permet d'introduire les hypothèses d'un but en les nommant éventuellement : elle construit donc une abstraction.

On a vu la tactique `Apply` qui construit une application. Cette tactique essaye d'unifier le type de retour du terme que l'on cherche à appliquer sur le sous-but. Comme l'unification d'ordre supérieure est indécidable, cette tactique peut échouer parfois. De nombreuses tactiques sont en fait des spécialisations de `Apply` qui rendent cette unification plus efficace dans le contexte de leur spécialisation. Par exemple, la tactique `Elim H` va appliquer le schéma d'élimination de `H` (lorsque le type de `H` est un type inductif). Le prédicat du schéma d'élimination est inféré à partir du but. Autre exemple, la tactique `Rewrite H` va appliquer le schéma d'élimination de l'égalité (lorsque le type de `H` est une égalité), et remplacer le premier terme de cette égalité par le deuxième dans le but.

D'autres tactiques appliquent une règle de conversion (ie remplacement d'un sous-terme par un sous-terme qui lui est interconvertible). Par exemple, `Unfold id` remplace la constante `id` par sa définition.

La tactique `Auto` va chercher à prouver le but complètement, à la Prolog, en essayant toutes les hypothèses, et tous les lemmes présents dans ses bases de données. L'utilisateur a la possibilité d'indiquer les bases de données à utiliser ainsi que la profondeur d'arbre de preuve à explorer, de créer ses propres bases de données de lemmes, etc.

Il y a des procédures de décisions spécialisées sur des problèmes précis : `Tauto` pour résoudre les formules propositionnelles intuitionnistes, `Ring` pour décider de l'égalité sur la structure libre d'anneau, `Omega` pour résoudre des problèmes sur l'arithmétique de Prestburger, etc.

Dans ce langage de tactiques, il faut noter la présence de `tacticals` qui permettent de combiner ces différentes tactiques. Par exemple l'opérateur de séquence `;` permet d'appliquer une suite de tactiques sur un ensemble de sous-buts ; l'opérateur `Orelse` qui permet d'essayer une première tactique, puis d'en appliquer une autre si cette première échoue ; un opérateur permet la composition de tactiques en parallèle sur plusieurs sous-but, etc.

Lorsqu'une preuve est finie, il reste à faire vérifier cette preuve par le typeur : c'est la commande `Qed` qui demande au système de faire ce travail.

Le langage de tactiques est en fait un langage de script, qui permet à l'utilisateur de s'abstraire de CCI : les scripts de preuves vont être plus facile à écrire, plus réutilisables, et plus robustes aux changements que les termes Coq eux-mêmes. Ceci dit, l'utilisateur a moins de contrôle sur la preuve produite : du point de vue programmation, cela peut être une source d'inefficacité. De plus, les scripts ne sont intelligibles que dans le cadre d'une utilisation interactive du système, et ils ne permettent pas de savoir quels arguments ont été utilisés dans la preuve. De nombreux outils pour gérer preuves et scripts sont proposés dans [Pon99].

Le cas de l'inversion On a vu que les schémas d'élimination permettent de raisonner par cas sur un type inductif : par exemple `nat`. Comme on manipule aussi des *familles inductives de types* (types inductifs dépendants), le schéma d'élimination simple n'est parfois pas suffisant, car il ne permet que de raisonner sur la totalité d'une famille inductive, mais pas sur une de ses *sous-familles*. Par exemple, le schéma d'élimination sur l'égalité permet de raisonner par cas de manière générale sur `x=y` en substituant `x` par `y`, mais pas sur `O=(S x)` ou `(S x)=(S y)`. Comme on l'a vu pour `discr_O_Sn`, il faut faire des étapes supplémentaires de raisonnement pour conclure que "aucun `x` ne vérifie `O=(S x)`". La tactique `Inversion` (voir [Cor97]) construit un tel raisonnement par cas sur une sous-famille. Bien sûr, à cause de l'indécidabilité de l'unification d'ordre supérieur, cette tactique ne réussit pas toujours, mais elle est bien utile en pratique. Comme on s'est beaucoup servi de familles inductives dans le chapitre 6, on va s'attarder quelques instants sur cette tactique.

En fait, le lemme `discr_O_Sn` est très facilement démontrable en Coq en utilisant cette tactique (voir ci-dessous).

Du point de vue Coq, `Lemma` est un synonyme de `Definition`. Dans le script ci-contre on commence donc par introduire `n` et l'hypothèse "`O=(S n)`" qu'on nomme `H` dans le contexte. Puis on fait une inversion sur cette hypothèse, ce qui achève la preuve car il n'y a aucun `n` qui vérifie `O=(S n)`.

```
Lemma discr_O_Sn: (n:nat)O=(S n)->False.
  Intros n H; Inversion H.
Qed.
```

Voilà ci-contre un autre exemple. On introduit ici `n`, `m` et "`(S m)=(S n)`" en hypothèses. On inverse cette dernière hypothèse. Il n'y a qu'un cas : "`m=n`", donc la tactique d'inversion va remplacer `m` par `n` dans le but. Il reste alors à prouver `n=n`, ce qu'on fait en utilisant le constructeur `refl_equal`.

```
Lemma S_injective:
 (n,m:nat)(S m)=(S n)->n=m.
   Intros n m H;
   Inversion H;
   Apply refl_equal.
Qed.
```

En fait, la tactique inversion produit un lemme intermédiaire appelé *lemme d'inversion* auquel il est possible de donner un nom lorsqu'on applique souvent l'inversion sur la même hypothèse (ce qui peut prendre un certain temps). Le lemme d'inversion `eq_Sm_Sn` fournit donc un schéma de raisonnement par cas (pour la sorte `Prop`) sur l'hypothèse "`(S m)=(S n)`" :

```
Derive Inversion eq_Sm_Sn with (n,m:nat)(S m)=(S n) Sort Prop.
```

Le terme `eq_Sm_Sn` ainsi engendré a le type suivant (modulo une petite simplification de ma part) ci-contre.

```
(P:nat->nat->Prop; n,m:nat)
   (P n n)->(S m)=(S n)->(P n m)
```

On est en fait confronté à l'incomplétude de cette tactique, dès qu'on manipule des types inductifs dépendants, comme `finiteSet`. On va montrer ce phénomène sur une autre version de `finiteSet` appelée `finiteSet1` telle que `(finiteSet1 n)` est un ensemble fini de $n+1$ éléments :

```
Inductive finiteSet1: nat -> Set :=
  | base1: (n:nat)(finiteSet1 n)
  | lift1: (n:nat)(finiteSet1 n)->(finiteSet1 (S n)).
```

On peut alors prouver le lemme qui dit que `(base1 O)` est le seul élément de `(finiteSet1 O)` :

```
Lemma O_one_elt: (x:(finiteSet1 O))x=(base1 O).
 Intros x; Dependent Inversion x; Apply refl_equal.
Qed.
```

Ici on a du utiliser une variante de `Inversion`, car le but dépend de l'hypothèse `x` sur laquelle on applique l'inversion. L'inversion de `(finiteSet1 O)` a donc réussi. Par contre si on essaye de prouver :

```
Lemma SO_two_elt: (x:(finiteSet1 (S O)))x=(base1 (S O))\/x=(lift1 O (base1 O)).
```

L'inversion sur `(finiteSet1 (S O))` échoue. Pour comprendre ce qu'il se passe, commençons par engendrer le lemme d'inversion de `(finiteSet1 (S O))` :

```
Derive Dependent Inversion finiteSet1_SO_inv with (finiteSet1 (S O)) Sort Prop.
```

Le type de ce lemme est le suivant :

```
(P:((a:nat)(finiteSet1 a)->Prop))
        (P (S O) (base1 (S O)))
        ->((f:(finiteSet1 O))(P (S O) (lift1 O f)))
        ->(x:(finiteSet1 (S O)))(P (S O) x)
```

Le problème dans la preuve de `SO_two_elt` est en fait que la tactique n'arrive pas à trouver une propriété `P` qui généralise le but. En effet, si on voit le but comme une fonction de `x`, la généralisation "naturelle" du but est de type `(finiteSet1 (S O))->Prop`. Le système n'arrive pas à généraliser ce dernier type en `(a :nat)(finiteSet1 a)->Prop`. On doit donc le faire à la main :

```
Lemma SO_two_elt: (x:(finiteSet1 (S O)))x=(base1 (S O))\/x=(lift1 O (base1 O)).
  Intros x.
  Apply finiteSet1_SO_inv with P:=
    [n:nat]<[n:nat](finiteSet1 n)->Prop>Cases n of
      | (S O) => [x:(finiteSet1 (S O))]x=(base1 (S O))\/x=(lift1 O (base1 O))
      | _ => [_]True
      end.
  (* Cas x=(base1 (S O)) *)  Auto.
  (* Cas x=(lift1 O f)   *)  Intros f; Rewrite (O_one_elt f); Auto.
Qed.
```

Dans cette preuve, on commence par introduire `x`, puis on applique le lemme d'inversion `finiteSet1_SO_inv`. On indique pour cela le prédicat `P` sur lequel doit s'appliquer ce lemme. Cette application engendre 2 cas indiqué ci-dessus en commentaire, entre "`(*`" et "`*)`". Le premier est résolu automatiquement. Le second but est le suivant :

```
(f:(finiteSet1 O))(lift1 O f)=(base1 (S O))\/(lift1 O f)=(lift1 O (base1 O))
```

On commence par introduire `f`, puis on remplace `f` par `(base1 O)`, comme l'autorise `O_one_elt`. Ce dernier sous-but est résolu automatiquement.

Si il est toujours possible d'aider ainsi la tactique inversion lorsqu'elle échoue, cela peut être relativement pénible. Pour éviter ce problème, on cherche à ne pas employer de type inductif qui mélange récursivité et types dépendants. On cherche donc à exprimer la récursivité de `finiteSet1` via un `Fixpoint`. On utilise des types inductifs pour exprimer les différents constructeurs de chaque cas : cas de bases, et cas récursifs. Pour les cas récursifs, les appels récursifs sont passés en argument (comme pour les continuations) du type inductif correspondant. Par exemple, ci-dessous `finiteSet2` est défini récursivement sur les entiers. Pour le cas de base, on a un seul constructeur possible (type `Unit`), tandis que pour le cas récursif, on en a deux (type `Add`).

```
Inductive Unit: Set := one: Unit.
Inductive Add[T:Set] : Set :=
  | baseA: (Add T)
  | liftA: T->(Add T).

Fixpoint finiteSet2 [n:nat] : Set :=
  Cases n of
    | O => Unit
    | (S m) => (Add (finiteSet2 m))
  end.
```

Le type `finiteSet2` est complètement équivalent au type `finiteSet1` précédent. Pour s'en convaincre, on peut simuler les 2 constructeurs de `finiteSet1` :

```
Definition base2:= [n:nat]<[n:nat](finiteSet2 n)> Cases n of
       | O => one
       | (S m) => (baseA (finiteSet2 m))
  end.

Definition lift2:(n:nat)(finiteSet2 n)->(finiteSet2 (S n)):= [n;x](liftA ? x).
```

On peut alors simuler les lemmes d'élimination de `finiteSet1` :

```
Lemma finiteSet2_rec:
  (P:((n:nat)(finiteSet2 n)->Prop))
        ((n:nat)(P n (base2 n)))
        ->((n:nat; f:(finiteSet2 n))(P n f)->(P (S n) (lift2 n f)))
        ->(n:nat; x:(finiteSet2 n))(P n x).
```

Ce lemme s'écrit facilement, en trois lignes de script.

On a finalement gagné que les lemmes précédants sont beaucoup plus faciles à écrire :

```
Lemma O_one_elt_2: (x:(finiteSet2 O))x=(base2 O).
  Unfold base2; Simpl; Intros x; Case x; Auto.
Qed.

Lemma SO_two_elt_2: (x:(finiteSet2 (S O)))x=(base2 (S O))\/x=(lift2 O (base2 O)).
  Unfold base2 lift2; Simpl; Intros x; Case x.
    (* Cas x=(base2 (S O)) *) Auto.
    (* Cas x=(lift2 O f)   *) Intros f; Case f; Auto.
Qed.
```

En effet, on n'a plus besoin d'utiliser la tactique d'inversion. Celle-ci est simulée par des réductions (tactique `Simpl`) et une application du schéma d'élimination sur des termes de type `Unit` et `Add`.

Ceci montre en fait que les types récursifs (définis par point fixe) sont plus effectifs que les types inductifs. Les types inductifs sont en fait strictement plus généraux : ils permettent d'exprimer une forme d'indéterminisme, qui doit être au contraire levé lorsqu'on utilise un point fixe. Pour cette raison, les types définis par point fixe sont un peu plus difficiles à établir et à comprendre. On remarque cependant qu'il est aussi très facile d'écrire `finiteSet` sur le modèle de `finiteSet2` : il suffit d'utiliser un type vide `Empty` au lieu de `Unit`.

3.2.2.d Découpage du code et compilation des preuves

Comme pour n'importe quel langage de programmation, il devient nécessaire de modulariser le code, dès que celui-ci devient important. Le découpage du code en Coq a lieu au travers de 2 mécanisme : le découpage en fichiers séparés et le découpage en section. Une section est en fait une unité à l'intérieur d'un fichier.

Mécanisme de section Une section est en fait un ensemble de définitions de l'utilisateur. Sur cet ensemble de définitions, l'utilisateur peut factoriser un certain nombre d'hypothèses ou de paramètres. Les sections permettent aussi de masquer certaines définitions. A la fin de la section, les paramètres et les hypothèses seront reinjectés dans dans chaque constantes exportées qui les utilisent.

Les sections ont surtout un intérêt pour celui qui en définit les constantes. En effet, lorsqu'on utilise les constantes exportées d'une section, il faut instancier chaque constante de manière autonome, car chacune peut avoir un jeu d'hypothèses différent, déterminé en fonction des hypothèses de la section qu'elle utilise réellement.

On peut cependant éviter en partie la tâche pénible d'instancier ces hypothèses, en ayant défini les constantes avec le mode arguments implicites.

Compilation des fichiers Un des avantages des objets preuves est que ces objets peuvent être compilés et stockés sur disque, de façon compressée. L'utilisateur a alors la possibilité de charger ces objets preuves en mémoire vive quand il en a besoin, sans les faire revérifier par le typeur.

Pour compiler une succession de commandes Coq, il suffit de les placer dans un fichier. Lors de la compilation du fichier, Coq va exécuter ces commandes comme au cours d'une session interactive. S'il ne rencontre pas d'erreurs, il va alors compiler l'ensemble des définitions dans un fichier binaire. Sinon, il va afficher un message d'erreurs.

Il est alors possible au cours d'une session Coq de recharger ces définitions en mémoire. Il faut noter que lors de ce rechargement, Coq effectue tout de même quelques vérifications pour préserver la cohérence. Par exemple, il faut vérifier la compatibilité des contraintes d'univers apportées par ces définitions, puisque d'entres elles pourraient être incompatibles celles de l'environnement courant.

Gestion de la δ-réduction En fait, lors du chargement en mémoire d'un fichier, il serait très coûteux de recharger entièrement toutes les définitions. En effet, en général en mathématiques, lorsqu'on prouve un théorème, on peut ensuite oublier la preuve de ce théorème, car on ne se sert que de son énoncé. En pratique, on ne sert donc pas en Coq de toutes les définitions, mais que d'une petite partie, et ce sont de plus en général les définitions de plus petite taille.

Lorsqu'il définit une constante en Coq, l'utilisateur a en fait le choix de faire de cette constante un terme *opaque* ou *transparent*. La constante est opaque si on ne souhaite pas exporter son terme, mais seulement son type. Elle est transparente si on exporte son type et son terme. Lorsqu'on importe une constante opaque, on ne va en fait importer que son type. L'utilisateur garde cependant toujours la possibilité de rendre cette constante transparente, et d'importer son terme, lorsqu'il en a besoin.

Ce mécanisme de termes opaques ou transparents a une influence sur le comportement de la réduction, en particulier la δ-réduction qui remplace une constante par sa définition. Un terme non typable peut le devenir (et inversement) suivant la nature opaque ou transparente d'une constante. L'utilisateur a cependant intérêt à garder le plus de constantes opaques possibles, car cela améliore l'efficacité de Coq : le typage est plus rapide (surtout dans les cas d'échec), les preuves créées automatiquement sont moins grosses, etc.

3.2.3 Coq comme langage de programmation

Voir [PM93b] et [Fil99].

Deuxième partie

Description du système de bibliothèques

Chapitre 4

Description informelle du système

A partir des exigences données dans l'introduction, et au cours des phases de prototypages en OCaml et en Coq décrites dans les chapitres 5 et 6, on a pu décrire plus précisement ce que devait être le langage de construction de bibliothèque du système Foc. On appelle FocKer ce langage, pour Foc Kernel.

4.1 Description informelle du système

On suppose disposer d'un langage de base qui permet d'exprimer types, programmes, propositions et preuves. Dans un esprit Curry-Howard, on assimile types et propositions : on les désignera sous le nom de *types*. De même, on désignera programmes et preuves, par le mot *terme*. Le langage de base possède en outre une notion de *contexte* qui correspond à une suite de déclarations et définitions. Le langage de base est muni d'un jugement de typage, qui permet de savoir si *un type est valide* dans un certain contexte (de déclarations et définitions) et si *un terme a un certain type*, dans un contexte donné. On dira alors qu'une déclaration est *valide* dans un certain contexte, si elle déclare un type valide dans ce contexte. De même, une définition est *valide* dans un certain contexte, si le terme qu'elle définit a le type que cette définition lui donne dans ce contexte. Un contexte sera dit valide si chacune de ses déclarations et définitions est valide dans le contexte des déclarations et définitions qui les précèdent. Cette notion de validité induit donc des dépendances entre déclarations et définitions : les déclarations et les définitions dans un contexte peuvent dépendre de celles qui précèdent dans ce contexte. Les propriétés du langage de base seront données plus formellement dans la partie III. Ce langage pourrait être Coq.

On suppose aussi donné un langage d'identificateurs, qui ne sont pas forcément des identificateurs du langage de base. On appellera ces identificateurs des *étiquettes*. Sur ce langage, on suppose l'existence d'une égalité décidable.

4.1.1 Interfaces et Collections

Intuitivement, les interfaces permettent de décrire les structures mathématiques abstraites telles les anneaux, les algèbres libres, etc... en enumérant les propriétés que vérifient ces structures. Les collections permettent de coder des domaines mathématiques tels les entiers, les anneaux de polynômes, etc... Les interfaces permettent donc de spécifier une famille de collections.

Interfaces sans paramètre Une interface est un ensemble fini et ordonné de couples étiquette/type. On écrira ici une interface sous la forme d'une suite $(< e_i, T_i >)_{i\in[1,n]}$ où e_i est une étiquette et T_i un type. Les étiquettes d'une interface donnée sont distinctes 2 à 2. L'ensemble ordonné $(T_i)_{i\in[1,n]}$ doit définir un contexte valide de déclarations.

Collections sans paramètre Une collection est un ensemble fini et ordonné de triplets étiquette/type/terme, notés $(< e_i, T_i, t_i >)_{i\in[1,n]}$. Les étiquettes d'une collection doivent être distinctes 2 à 2. L'ensemble ordonné $(< T_i, t_i >)_{i\in[1,n]}$ doit définir un contexte valide de définitions. On impose aussi que $(T_i)_{i\in[1,n]}$ définisse un contexte valide de déclarations : les types ne peuvent donc pas dépendre des termes qui les précèdent (vis-à-vis de l'ordre).

Cette contrainte permet de faire correspondre à chaque collection $(< e_i, T_i, t_i >)_{i\in[1,n]}$ une interface $(< e_i, T_i >)_{i\in[1,n]}$. Cela correspond aussi à la condition 5 du cahier des charges donné en section 1.4.1.a.

Collections paramétrées Les collections peuvent prendre en paramètre plusieurs autres collections. Ces collections en paramètre sont spécifiées par leurs interfaces. On définit une opération d'instanciation sur les collections paramétrées qui permet de construire une collection en substituant chaque paramètre par une autre collection.

Sélection des méthodes des collections On appelera méthode les éléments des collections (et des interfaces). Pour une collection non paramétrée, ou pour une collection complètement instanciée, on peut sélectionner une des ses méthodes en donnant son étiquette. Cela permet d'utiliser la définition de cette méthode. Le langage des interfaces/collections est donc un langage d'enregistrements à champs dépendants.

Conversion sur les collections Entre deux interfaces, on peut définir une relation d'inclusion : on dit qu'un interface A est incluse dans B ssi on obtient A à partir de B en oubliant des couples étiquette/type. De même, si A est incluse dans B, on peut définir une fonction de conversion qui transforme les collections d'interface B en collection d'interface A : il suffit d'oublier au niveau des collections, les éléments correspondants à ceux oubliés pour passer de B vers A.

Renommage Le renommage est une opération explictement manipulée par l'utilisateur. Elle s'applique à une interface $(< e_i, T_i >)_{i\in[1,n]}$ (resp. une collection $(< e_i, T_i, t_i >)_{i\in[1,n]}$) pour créer une nouvelle interface $(< e'_i, T_i >)_{i\in[1,n]}$ (resp. une collection $(< e'_i, T_i, t_i >)_{i\in[1,n]}$) en changeant uniquement les étiquettes de e_i en e'_i. Les étiquettes e'_i doivent bien sûr être distinctes 2 à 2.

Cet opération de renommage se combine donc à la notion d'inclusion et de conversion décrites ci-dessus : on parlera de la relation d'inclusion après un certain renommage, et de même, d'une conversion après un certain renommage. Ces renommages devront être, là encore, explicités par l'utilisateur.

Type support et entités Intuitivement, les interfaces et les collections décrivent des opérations et des propriétés d'un ensemble concret d'objets mathématiques (par exemple les polynômes). On appelera *entités* ces objets. Ces entités étant des termes de notre langage de base, ils ont eux-mêmes un type. Ce type est désigné sous le nom de *type support*. Ce type est en fait lui-même un terme du langage. Il sera défini comme un élément distingué d'une interface ou d'une collection. Le type support d'une interface est donc juste déclaré, mais dans une collection, il est défini. Ceci dit, une collection n'exporte pas la définition de son type support, qui reste abstrait à l'extérieur de la collection. Cette restriction découle de la condition 5 donnée en section 1.4.1.a.

4.1.2 Espèces

Les espèces fournissent un moyen de construire des collections. Elles permettent surtout de partager des définitions génériques entre collections.

4.1.2.a Définition

Une espèce est un ensemble fini et ordonné avec deux sortes d'éléments : des couples étiquette/type et des triplets étiquette/type/terme. Pour l'homogénéité de définition, on ajoute un terme $\perp$ qui représente le non-défini. Ainsi, une espèce devient un ensemble de triplets étiquette/type/terme noté $(< e_i, T_i, t_i >)_{i \in [1,n]}$, vérifiant les mêmes conditions que les collections. Les espèces peuvent prendre en paramètre des collections.

On désigne aussi les éléments des espèces sous le mot *méthode*. Une méthode sera dite *définie* si son terme est différent de $\perp$, et *déclarée* sinon.

Si les interfaces servent à spécifier les collections, les espèces servent à construire les collections. A chaque espèce dont toutes méthodes sont définies, on peut en effet associer une unique collection $(< e_i, T_i, t_i >)_{i \in [1,n]}$. On va donc maintenant donner un certain nombre d'opérations sur les espèces pour construire incrémentalement des espèces à partir d'autres espèces.

4.1.2.b Opérations sur les espèces

On désigne par le mot *héritage* cette notion de "construction incrémentale". L'héritage dans les espèces a 4 primitives : l'enrichissement, la spécialisation, la redéfinition, et la fusion.

Enrichissement Enrichir une espèce consiste à lui ajouter des méthodes déclarées. Il faut vérifier que ces déclarations sont valides dans le contexte des déclarations précédentes.

Spécialisation Spécialiser une espèce consiste à définir des méthodes qui étaient simplement déclarées. Lors de la définition d'une méthode (simplement déclarée), on vérifie que la définition correspond au type de la méthode dans le contexte des déclarations et définitions précédentes.

Redéfinition La redéfinition consiste à redéfinir une méthode (déjà définie). Cela nécessite de considérer les dépendances entre méthodes. On dira que la méthode m_2 dépend de manière *opaque* de m_1, si pour donner le type ou la définition de m_2, on a besoin du type de m_1, mais pas de sa définition. De même, on dira que la méthode m_2 dépend de manière *transparente* de m_1, si pour donner la définition de m_2, on a besoin du type et de la définition de m_1.

Lorsqu'on redéfinit une méthode m_1, on rend non définies toutes les méthodes qui dépendent de manière transparente de m_1 : on conserve leur type (car les types ne dépendent pas de manière transparente des définitions), mais on efface leur définition.

Fusion La fusion est une opération binaire sur les espèces. Ce n'est pas une opération symétrique : si A et B sont deux espèces, la fusion de A avec B n'est pas égale à la fusion de B avec A. La fusion de A avec B contient en fait forcément toutes les définitions de A mais pas forcément toutes celles de B.

Pour faire la fusion d'une espèce A avec une espèce B, il faut que les méthodes de même étiquette aient des types compatibles : cela signifie qu'il existe une interface contenant toutes les étiquettes communes à A et B, et qui soit incluse dans les interfaces associées à A et B.

Si cette condition est respectée, la fusion consiste donc à enrichir une des espèces, par exemple A, avec toutes les méthodes de B n'étant pas dans A. Ensuite, on spécialise l'espèce ainsi obtenue en utilisant les définitions de B pour les méthodes qui étaient seulement déclarées dans A, lorsque c'est possible. Remarquons en effet que cette spécialisation n'a pas toujours un sens. D'une part, une de ces définitions de B peut dépendre (dans B) de façon transparente d'une méthode qui est aussi définie dans A, mais avec une autre définition : cette définition venant de B n'a donc plus de sens dans la fusion de A et B. D'autre part, une de ces définitions sur une méthode m peut dépendre d'une méthode p dans B, et la méthode p peut dépendre de m dans A : l'ajout de cette définition de m créerait donc un cycle dans les dépendances, et elle n'a donc pas de sens.

Remarquons finalement que l'utilisateur sait donc facilement de quelle espèce est issue chacune des définitions de la fusion : les seules définitions qui sont issues de B sont sur les méthodes de la fusion qui n'étaient pas définies dans A.

Renommage La fusion impose de donner aux méthodes de même étiquette la même "sémantique" à l'intérieur de l'espèce : les méthodes de même étiquettes auront non seulement le même type, mais elles devront vérifier les mêmes propriétés. On définit donc une opération de renommage sur les étiquettes, qui permet de distinguer des méthodes de même nom avant de faire une fusion. C'est l'utilisateur qui doit indiquer explicitement les renommages.

4.1.2.c Utilisation de l'héritage

Si A et A' sont des espèces et si A' hérite de A, alors l'interface de A est incluse dans celle de A' après éventuellement un renommage de méthodes. Ainsi "tout ce qu'on peut faire avec A, on peut le faire avec A'". Mais, comme précisé ci-dessus, cette notion d'héritage n'est pas une relation entre spécifications, mais le résultat d'opérations sur les spécifications. Ainsi le fait que A' hérite de A fait partie de la spécification de A' : cela sert à la construire. Cela rend FocKer extensible car toute opération dérivée ajoutée à A est automatiquement propagée dans A'. On gagne aussi en concision, car la définition de A' n'a à fournir que les méthodes déclarées et les méthodes définies à ajouter à A ou à redéfinir.

Pour gagner encore en concision, on peut associer à chaque espèce A, une interface de même nom, obtenue en oubliant toutes les définitions de A. Ainsi à chaque fois que l'utilisateur définit une espèce, il définit aussi une interface qui pourra lui servir à spécifier les collections obtenues à partir de cette espèce.

Comme on l'a déjà dit, lorsque toutes les méthodes d'une espèce sont définies, on peut en faire une collection. Mais ce processus n'est pas réversible : une collection ne peut pas redevenir une espèce. Une collection est en effet une unité de bibliothèque complètement fermée, et en particulier la représentation de son type support est masquée. Pour souligner que toutes les méthodes d'une espèce sont définies, on introduit donc la notion *d'espèce concrète*. On parlera d'espèce abstraite sinon. Ainsi une espèce concrète est presque une collection, mais à la différence des collections, on peut en hériter, utiliser la représentation de son type support, etc...

4.1.2.d Exemples d'héritage

Exemple de combinaison enrichissement/spécialisation L'espèce des anneaux factoriels déclare une fonction totale de calcul du pgcd. L'espèce des anneaux euclidiens possède une division euclidienne et le pgcd y est obtenu par l'algorithme d'Euclide. L'espèce des anneaux euclidiens est en fait construite par enrichissement de l'espèce des anneaux factoriels en ajoutant la division euclidienne et par spécialisation à partir de cette même espèce en définissant le pgcd.

Lorsqu'on code une collection A (par exemple, la collection des entiers relatifs) implantant l'espèce abstraite des anneaux euclidiens, on n'aura pas à expliciter le calcul du pgcd dans cette collection, puisqu'il existe un algorithme général, au niveau des anneaux euclidiens. De plus, on a une conversion qui transforme A en anneau factoriel.

Exemple de fusion L'espèce concrète des anneaux de polynômes sur un corps peut être obtenue par fusion de l'espèce concrète des anneaux de polynômes sur un anneau factoriel, et l'espèce des anneaux euclidiens. Ces deux dernières espèces contiennent une méthode implantant un algorithme de calcul de pgcd : Euclide pour la dernière, et un algorithme par calcul des sous-résultants pour la première. Suivant l'ordre dans lequel on effectue la fusion, on peut obtenir au niveau de anneaux de polynômes sur un corps l'une ou l'autre des implantations du calcul du pgcd. On préférera sans doute l'algorithme des sous-résultants qui est plus efficace.

Exemple de fusion avec renommage Un anneau est la combinaison d'un groupe et d'un monoïde. L'espèce des groupes hérite elle-même de l'espèce des monoïdes. L'espèce des monoïdes hérite de l'espèce des setoïdes, qui définit un type muni d'une relation d'équivalence, appelée égalité.

Pour construire l'espèce des anneaux, on fera donc une fusion de l'espèce des groupes et de celle des monoïdes, après renommage de l'opération de monoïde (et d'élement neutre, etc...), mais en assimilant (ie sans renommer) les 2 setoïdes sous-jacents. Ainsi dans un anneau le groupe sera noté additivement, et le monoïde multiplicativement, mais le groupe et le monoïde partageront la même égalité.

4.2 Exemples dans une pseudo-syntaxe concrète

Pour rendre plus concret ce système au lecteur, on donne quelques exemples dans une syntaxe virtuelle, inspirée de celle de Coq. Les preuves sont donc des λ-termes. On utilise le fait que $\neg P$, où P est une proposition, est défini comme $P \rightarrow \texttt{False}$.

Exemple d'espèce L'espèce des setoïdes peut être décrite comme suit (pour la concision de l'exemple, on suppose juste la relation `eq` réflexive) :

$$setoid := \left\{\begin{array}{ll}
\texttt{decl} & T : \texttt{Type} \\
\texttt{decl} & eq : T \rightarrow T \rightarrow \texttt{Prop} \\
\texttt{decl} & eq_refl : \forall (x : T)(eq\ x\ x) \\
\texttt{def} & neq : T \rightarrow T \rightarrow \texttt{Prop} \\
 & := \lambda[x, y : T]\neg(eq\ x\ y) \\
\texttt{def} & neq_spec : \forall (x, y : T)([\![neq]\!]\ x\ y) \rightarrow \neg(eq\ x\ y) \\
 & := \lambda[x, y : T; H : \neg(eq\ x\ y)]H \\
\texttt{def} & neq_nrefl : \forall (x : T)\neg(neq\ x\ x) \\
 & := \lambda[x : T; H : (neq\ x\ x)](neq_spec\ x\ x\ H\ (eq_refl\ x))
\end{array}\right\}$$

Le mot-clef `decl` introduit une déclaration, et le mot-clef `def` une définition. Dans cette syntaxe, on demande à l'utilisateur d'indiquer les dépendances transparentes et opaques. Par défaut les dépendances seront considérées comme opaques. Pour indiquer une dépendance transparente, il faut indiquer entre $[\![.]\!]$ *une* occurrence de l'identificateur dont on a besoin de connaître la définition. Par exemple, ci-dessus, neq_spec dépend de neq de manière transparente. En effet, le paramètre H de son terme est de type $\neg(eq\ x\ y)$: il faut donc connaître la définition de neq pour pouvoir considérer que H est de type $(neq\ x\ y)$.

Exemple d'espèce paramétrée A partir de 2 setoïdes, on peut construire l'espèce des morphismes de setoïdes. Dans l'exemple suivant, on suppose que si A désigne une collection, on accède à une méthode T de A, via l'expression $A.T$:

$$map[A, B : setoid] :=$$

$$\left\{\begin{array}{ll}
\texttt{decl} & ap : A.T \rightarrow B.T \\
\texttt{decl} & ap_eq : \forall (x, y : A.T)(A.eq\ x\ y) \rightarrow (B.eq\ (ap\ x)\ (ap\ y)) \\
\texttt{def} & prop : \forall (x, y : A.T)(B.neq\ (ap\ x)\ (ap\ y)) \rightarrow \neg(A.eq\ x\ y) \\
 & := \lambda[x, y : A.T; H : (B.neq\ (ap\ x)\ (ap\ y)); H' : (A.eq\ x\ y)] \\
 & \qquad (B.neq_spec\ (ap\ x)\ (ap\ y)\ H\ (ap_eq\ x\ y\ H'))
\end{array}\right\}$$

Exemple d'une espèce concrète paramétrée avec spécialisation et redéfinition On va construire l'espèce concrète du setoïde de morphismes de setoïdes.

$$map\_setoid[A, B : setoid] :=$$

$$\left\{\begin{array}{lll} \multicolumn{3}{l}{\texttt{inherit}\ setoid} \\ \texttt{def} & T : & \texttt{Type} \\ & := & (map\ A\ B) \\ \texttt{def} & eq : & [\![T]\!] \to [\![T]\!] \to \texttt{Prop} \\ & := & \lambda[p, q]\forall(x : A.T)(B.eq\ (p.ap\ x)\ (q.ap\ x)) \\ \texttt{def} & eq\_refl : & \forall(p : [\![T]\!])([\![eq]\!]\ p\ p) \\ & := & \lambda[p; x : A.T](B.eq\_refl\ (p.ap\ x)) \\ \texttt{def} & neq : & [\![T]\!] \to [\![T]\!] \to \texttt{Prop} \\ & := & \lambda[p, q]\exists[x : A.T](B.neq\ (p.ap\ x)\ (q.ap\ x)) \\ \texttt{def} & neq\_spec : & \forall(p, q : [\![T]\!])([\![neq]\!]\ p\ q) \to \neg([\![eq]\!]\ x\ y) \\ & := & \lambda[p, q; H]... \end{array}\right\}$$

Dans cet exemple, on a choisi de redéfinir `neq`. Comme `neq_spec` dépendait de manière de transparente de `neq`, pour avoir une espèce concrète, il a fallu aussi redéfinir `neq_spec`. Par contre, on a récupéré gratuitement la méthode `neq_nrefl` qui dépendait de `neq` et `neq_spec` de manière opaque.

Chapitre 5

Implantation en OCaml

Ce chapitre présente le prototype en OCaml qui a permis de mettre au point la description FocKer du chapitre précédent, en respectant le cahier des charges donné en introduction. On illustre ce prototype en donnant le code OCaml associé à quelques espèces et collections. Ce code OCaml est donné à titre indicatif, mais n'est pas le code de FOC. En fait ce n'est même pas du code réellement exécutable.

5.1 Espèces et collections de base

Côté OCaml, on veut profiter de l'héritage au niveau des objets pour représenter l'héritage de FocKer. Ceci dit, on n'a pas envie de tomber dans les problèmes évoqués en section 2.2.5. En particulier, on a vu que le typage des objets ne suffit pas pour protéger la représentation des données. On souhaite donc pallier ce problème en utilisant un style de programmation "type abstrait de donnée", dans lequel structure et type support sont deux notions distinctes. Pour protéger la représentation, on rendra en fait le type support abstrait en utilisant le mécanisme d'abstraction de type des modules.

5.1.1 Description générale

Les espèces vont correspondre à des classes OCaml. Les méthodes déclarées dans les espèces vont se traduire en méthodes virtuelles dans les classes OCaml. De même, les méthodes définies vont se traduire en méthodes concrètes. On utilise l'héritage dans les classes pour représenter l'héritage sur les espèces. Le mécanisme de redéfinition avec liaison tardive en OCaml permet de simuler le mécanisme de redéfinition de FocKer.

Pour représenter les collections de FocKer, on va utiliser les objets de OCaml. Ainsi le fait qu'une espèce dont toutes les méthodes sont définies engendre une collection sera représenté par un `new` en OCaml. Les objets ne contiendront en fait que les opérations sur un certain type support. Une collection sera donc représentée par un couple `(t,meth)` où `t` est le type support, et `meth` est un objet qui contient les méthodes de la collection. On utilise le mécanisme d'abstraction de type des modules pour rendre `t` abstrait, et respecter ainsi le point 5 du cahier des charges. Ce couple sera donc implanté par un module à deux champs.

Les interfaces seront représentées elles-aussi par un couple variable de (type/type objet) où la variable de type représente le type support, et le type objet porte les méthodes de l'interface. Ce type objet sera en fait un type ouvert : ainsi la relation d'inclusion sur les interfaces va correspondre à de l'instanciation de variables de rangée dans les types objets ouverts. La conversion sous-jacente est donc en fait l'identité.

Les espèces seront représentées par un couple (type du support/classe). Si le type support est simplement déclaré, celui-ci sera représenté par une variable de type, qui sera un paramètre de la classe. Sinon, ce sera un type OCaml quelconque.

5.1.2 Exemple

On illustre ici l'exemple des anneaux factoriels et euclidiens donné en 4.1.2.d. On introduit tout d'abord deux classes virtuelles paramétrées par leur type support, correspondant à l'espèce des anneaux factoriels et à l'espèce des anneaux euclidiens.

On suppose que la classe `anneau_integre` a été définie précédemment, que cette classe déclare une méthode `eq_zero`.

```
class virtual ['a] anneau_factoriel =
object
  inherit ['a] anneau_integre
  method virtual pgcd: 'a->'a->'a
  ...
end

class virtual ['a] anneau_euclidien =
object (self)
  inherit ['a] anneau_factoriel
  method virtual reste_eucl: 'a->'a->'a
  method pgcd =
    let rec pgcd_rec x y =
      if (self#eq_zero y)
      then x
      else (pgcd_rec y (self#reste_eucl x y))
    in pgcd_rec
  ...
end
```

Les classes précédentes définissent aussi l'interface des anneaux factoriels (et resp. l'interface des anneaux euclidiens). Ainsi l'interface des anneaux factoriels est représentée par le couple `('a, 'a #anneau_factoriel)`.

On peut alors définir une espèce dont toutes les méthodes sont définies. Cette étape se passe en 2 temps. D'abord on définit une classe virtuelle qui prend le support en paramètre et permet de déclarer les méthodes qu'introduit cette espèce vis-à-vis des espèces dont elle hérite. Ensuite, on définit une classe concrète qui implante les méthodes virtuelles de cette classe virtuelle. Par exemple, dans l'espèce des entiers relatifs ci-dessous, on introduit une nouvelle méthode `parse_int` qui permet de construire un élément du type support en fonction d'un entier "machine". Cette méthode `parse_int` est d'abord déclarée dans `entiers_relatifs_sig`, avant d'être implantée dans `entiers_relatifs`. La classe `entiers_relatifs` implante aussi les méthodes restées virtuelles dans `entiers_relatifs_sig`, comme `reste_eucl`.

```
class virtual ['a] entiers_relatifs_sig =
object
  inherit ['a] anneau_euclidien
  virtual method parse_int: int -> 'a
end
```

```
class entiers_relatifs =
object
  inherit [int] entiers_relatifs_sig
  method reste_eucl x y = x mod y
  ...
  method parse_int x = x
end
```

L'intérêt de la classe `entiers_relatifs_sig` surgit lorsqu'on veut créer la collection des entiers relatifs : elle sert à abstraire le type support dans les objets créés à partir de `entiers_relatifs`. On définit une signature de module qui abstrait le type support de l'implantation de module `EntiersRelatifs`.

```
module type EntiersRelatifs_sig =
  sig
    type t
    val meth: t entiers_relatifs_sig
  end
end
```

```
module EntiersRelatifs: EntiersRelatifs_sig =
  struct
    type t=int
    let meth = new entiers_relatifs
  end
end
```

Le type de `EntiersRelatifs.meth` est donc `EntiersRelatifs.t entiers_relatifs_sig`, qui est une abréviation pour le type objet fermé :

$$< \texttt{parse\_int} : \texttt{int} \rightarrow \texttt{EntiersRelatifs.t}\,;\ \texttt{pgcd} : \texttt{EntiersRelatifs.t} \rightarrow \texttt{EntiersRelatifs.t} \rightarrow \texttt{EntiersRelatifs.t}\,;\ \textit{etc} >$$

Le type `EntiersRelatifs.t` est ici un type abstrait, dont la représentation est masquée. Chaque collection de base aura donc un type support incompatible avec celui de toutes les autres. Ce type support ne sera utilisable qu'à travers les méthodes de la collection qui auront été prévues pour, comme `parse_int` dans l'exemple ci-dessus. L'utilisateur ne risque donc pas de mal utiliser la représentation des données : il n'y a pas accès. L'implanteur de la collection peut alors changer à son gré la représentation des données, sans que l'utilisateur ait à en souffrir. L'utilisateur pourra cependant écrire des fonctions qui s'appliquent sur différentes collections issues des mêmes espèces : ces fonctions devront être elles-même polymorphes sur le type support.

Avec ce mélange original d'objets et de modules, on obtient finalement un style pour construire des bibliothèques qui a la souplesse de l'héritage offert par les objets, tout en ayant le contrôle du typage offert par les modules. La mise au point de ce style de programmation a nécessité le développement de plusieurs prototypes afin d'analyser d'un point de vue *pratique* les différents traits du langage, et en particulier les modules et les objets. C'est une synthèse de ce travail d'analyse, dégagée du contexte du calcul formel, qui a été présentée au chapitre 2.

5.2 Espèces et collections paramétrées

On s'intéresse maintenant aux espèces et aux collections paramétrées. En fait, on considère ici des espèces et collections paramétrées du premier ordre : les collections prises en paramètres sont des collections de base. Dans la section 5.4.1, on reviendra sur le problème de l'implantation des espèces et des collections paramétrées d'ordre supérieur, c'est-à-dire prenant en paramètre des collections elles-mêmes paramétrées.

5.2.1 Description générale

Une espèce paramétrée par des collections est une classe comportant deux paramètres pour chacune des collections : un paramètre de type pour le type support, et un objet contenant les méthodes de la collection.

On pourrait choisir de représenter les collections paramétrées par des foncteurs, mais cela est un peu lourd. On préfère en fait redescendre les paramètres sur les 2 champs `t` et `meth`. Le type `t` est ainsi paramétré par les types support des collections prises en arguments, et `meth` est une fonction prenant des objets en argument et retournant un objet.

5.2.2 Exemple

On introduit maintenant l'espèce des algèbres polynomiales en une variable (à degré dans les entiers).[1] On suppose défini sur le même modèle que précédemment une classe `anneau` de laquelle hérite `anneau_integre`. Cette classe `anneau` déclare en particulier une méthode `mult` de multiplication (interne). La classe `anneau` définit la méthode `eq` de test d'égalité, via la technique décrite en 1.3 (application de `eq_zero` sur la différence). Une algèbre est une espèce qui prend en paramètre une collection qui est un anneau. L'interface de cette collection est ici représentée par le couple `('c,'c #anneau)`. On définit alors une algèbre conformément à la présentation donnée en section 1.3 : `lift` est un morphisme d'anneau. On définit la multiplication externe `mult_extern` de l'algèbre à partir de ce morphisme et de la multiplication interne. L'espèce `alg_poly` a aussi une méthode `lc` qui retourne le coefficient dominant d'un polynôme. Ce coefficient est non nul sauf si le polynôme pris en argument est lui-même nul. Cela permet d'implanter la méthode `eq_zero` en utilisant l'égalité à zéro sur les cœfficients. Cela permet aussi de tester si un polynôme est isomorphe à un cœfficient (le polynôme nul est isomorphe à zéro).

```
class virtual ['c,'a] alg_poly (coef:'c #anneau) = object (self)
  inherit ['a] anneau
  method virtual lift: 'c -> 'a
  method mult_extern c p =
     self#mult (self#lift c) p
  ...
  method virtual lc: 'a -> 'c
  method eq_zero p =
    coef#eq_zero (self#lc p)
  method is_coef p =
    self#eq p (self#lift (self#lc p))
end
```

On peut donner l'espèce concrète des polynômes creux en une variable, avec un ordre par degré décroissant sur les monômes. (voir section 1.3)

```
class ['c] poly_creux (coef:'c #anneau) =
object
  inherit ['c,('c*int) list] alg_poly coef
  method lift c =
    if (coef#eq_zero c) then [] else [(c,0)]
  method lc p = match p with
    | [] -> coef#zero
```

[1] Cet exemple pourrait sans difficulté être généralisé à plusieurs variables, mais ce n'est pas le propos ici.

```
    | (x,d)::l -> x
  ...
end
```

La collection correspondant à cette espèce concrète est donnée par :

```
module type Poly_sig =
  sig
    type 'c t
    val meth:
      ('c #anneau) -> ('c,'c t) alg_poly
  end
```

```
module PolyCreux: Poly_sig =
  struct
    type 'c t = ('c*int) list
    let meth = new poly_creux
  end
```

Le type de `PolyCreux.meth` est `('c #anneau) -> ('c,'c PolyCreux.t) alg_poly`. Instancier cette collection par la collection des entiers relatifs, revient donc à construire le couple :

`(EntiersRelatifs.t PolyCreux.t, PolyCreux.meth EntiersRelatifs.meth)`

La construction de `PolyCreux.meth EntiersRelatifs.meth` est autorisée par le vérificateur de types, car `EntiersRelatifs.t entiers_relatifs_sig` est une instance du type polymorphe `'c #anneau`.

Ainsi, deux polynômes bâtis sur le même anneau de cœfficients auront donc le même type. Mais deux polynômes bâtis sur deux anneaux de cœfficients distincts auront des types incompatibles. On a en effet vu que deux collections de bases distinctes ont des types supports incompatibles.

5.3 Extensions possibles vis-à-vis de FocKer

La puissance d'expression de OCaml permet des constructions qui n'ont pas été intégrées dans FocKer. Ces constructions sont pourtant intéressantes, mais on n'en maîtrise pas encore suffisamment tous les aspects côtés preuves. On présente ici deux de ces constructions : l'extension de domaine des algorithmes génériques, et la possibilité d'écrire des collections récursives.

5.3.1 Extension de domaine des algorithmes génériques

Dans FocKer, on s'est surtout focalisé sur la possibilité de spécialiser un algorithme générique au cours de l'héritage : ajouter "plus de structure" (ajouter des propriétés et des opérations, ou préciser le support) permet d'écrire des algorithmes plus efficaces, plus offensifs. Mais en fait, ajouter "plus de structure" permet aussi parfois d'augmenter le domaine d'application d'un algorithme générique (vu comme fonction partielle). En utilisant la liaison *super* (voir 2.2.3.f), on peut simuler un tel mécanisme : cette liaison permet de faire référence à l'implantation de la méthode dans la classe parente.

Pour illustrer ce point, on prend l'exemple de la fonction `iter` qui, sur un semigroupe (structure avec une loi interne associative), prend un élément x du semigroupe et un entier n, et qui itère n fois la loi du semigroupe sur x. Cet algorithme est partiel au niveau des semigroupes : n doit être strictement positif. Sur les monoïdes, on peut étendre son domaine d'application par 0 (le résultat de `iter` étant alors l'élement neutre). Sur les groupes, cette fonction devient totale.

Evidemment, plus l'algorithme est compliqué, plus cette possibilité d'extension, qui évite de refaire le travail est intéressante. Pour la simplicité de l'exemple, on présente ici un algorithme naïf (simple récursion terminale), mais on pourrait imaginer un algorithme moins trivial, comme un algorithme dichotomique. La seule partie à modifier serait alors la fonction `iter_rec`.

En OCaml, on a donc :

```
class virtual ['a] semigroupe =
object (self)
  inherit ['a] ensemble
  method virtual plus : 'a->'a->'a
  method iter =
    (** fonction partielle n ≥ 1 **)
    let rec iter_rec x n acc =
      if n<=1
      then acc
      else (iter_rec x (n-1) (self#plus x acc))
    in (iter_rec x n x)
end
```

```
class virtual ['a] monoide =
object (self)
  inherit ['a] semigroup as super
  method virtual zero: 'a
  method iter x n =
    (** fonction partielle n ≥ 0 **)
    if n>0
    then (super#iter x n)
    else self#zero
end
```

```
class virtual ['a] groupe =
object (self)
  inherit ['a] monoide as super
  method virtual opp: 'a->'a
  method iter x n =
    if n<0
    then (self#opp (super#iter x (-n)))
    else (super#iter x n)
end
```

5.3.2 Exemple des polynômes récursifs

Cet exemple utilise à la fois la possibilité d'écrire des foncteurs et la récursivité de la liaison tardive. Les polynômes récursifs sont une représentation des polynômes en un nombre potentiellement non borné de variables. Ils sont obtenus en itérant une *représentation donnée* des polynômes sur elle-même (voir 1.3). Par exemple, on peut créer des polynômes récursifs, en itérant sur une représentation des polynômes à deux variables :

$$A[X_1, X_2, X_3, X_4, \ldots, X_{2n}, X_{2n+1}, \ldots] = ((\ldots(A[X_1, X_2])[X_3, X_4]\ldots)[X_{2n}, X_{2n+1}])\ldots$$

Ce genre de représentation s'avère utile pour implanter des algorithmes comme ceux de [Rio92].

On ne se place pas ici au niveau de généralité de la définition donnée en section 1.3 (page 21), car on ne prend pas S, l'ensemble des indéterminées, en paramètre. On choisit de prendre les chaines de caractères muni de l'ordre lexicographique pour S. Mais il n'est pas très compliqué de généraliser cet exemple afin de prendre S quelconque. La réelle difficulté est plutôt de prendre P, la “représentation de polynômes”, en paramètre. On utilise ici la signature de module `Poly_sig` précédente pour capturer cette notion de représentation de polynômes. L'espèce des polynômes récursifs sera en faite décrite par un foncteur qui prend en argument un module `P` de signature `Poly_sig`. La récursivité de ces polynômes se traduit sur deux niveaux : le type support est récursif, mais l'espèce est elle aussi récursive, via le mécanisme de liaison tardive.

On garde ici les polynômes récursifs sous forme canonique, c'est-à-dire que les polynômes construits avec le constructeur `Comp` ne sont pas isomorphes à des cœfficients (le polynôme nul étant isomorphe au cœfficient zéro).

```
module PolRecEspece (P:Poly_sig) =
 struct
```

```
(*** type support ***)
type 'c sup =
| Base of 'c
| Comp of string*(('c sup) P.t)
```

Les polynômes récursifs sont indexés par des noms de variables (type `string`), et leurs cœfficients sont des polynômes récursifs avec des noms de variables strictement plus petits (ordre lexicographique).

```
(*** méthodes ***)
class ['c] espece (coef:'c #anneau) =
object (self)
  inherit ['c,'c sup] alg_poly coef
  method lift a = (Base a)

  method lc p = match p with
    | (Base a) -> a
    | (Comp (v,l)) -> self#lc ((P.meth self)#lc l)

  method mult =
     let compose v l =
       (*** mise en forme canonique ***)
       if ((P.meth self)#is_coef l)
       then (P.meth self)#lc l
       else Comp (v,p)
     in fun x y -> match (x,y) with
        | (Base a, Base b) -> Base (coef#mult a b)
        | ((Comp (v1,l1)) as x1,(Comp (v2,l2) as x2))->
            if (v1<v2)
            then compose v2 ((P.meth self)#mult_extern x1 l2)
            else if (v1=v2)
              then compose v1 ((P.meth self)#mult l1 l2)
              else compose v1 ((P.meth self)#mult_extern x2 l1)
        | (Comp (v,l), x) -> compose v ((P.meth self)#mult_extern x l)
        | (x, Comp (v,l)) -> compose v ((P.meth self)#mult_extern x l)
...
  end
end
```

L'objet *self* sera de type `'c #anneau` → `('c,'c sup) alg_poly`

On cherche le polynôme récursif canonique de cœfficient `l` (de type `('c sup) P.t`) et de variable `v`. En supposant que `l` est lui-même sous forme canonique, il suffit de regarder si `l` est un cœfficient (y compris zéro) auquel cas, le polynôme récursif sera lui-même ce cœfficient.

Dans la classe `espece` ci-dessus, on commence par hériter de `alg_poly`, il faut donc implanter les méthodes encore virtuelles de cette classe `lift`, `lc` et `mult`. Dans ces implantations, on utilise les opérations définies dans `P`, via des appels sur `(P.meth` *self*`)`. Ces appels sont récursifs puisque dans `(P.meth` *self*`)`, on utilise les méthodes de `self` qu'on est en train de définir. Mais les méthodes utilisées lors de ces appels récursifs, sont invoquées sur des polynômes de variables strictement plus petites : la terminaison est assurée par le fait que chaque polynôme récursif est bâti sur un nombre fini de variables.

Dans la méthode `mult`, on commence par définir une fonction `compose` qui prend un polynôme `l` de type `('c` *sup*`) P.t` et une variable `v`, et renvoie un polynôme récursif sous forme canonique. On suppose ici, bien sûr, que les cœfficients de `l` sont eux-mêmes sous forme canonique. La méthode `mult` raisonne ensuite sur ses arguments pour savoir quelle multiplication utiliser (la multiplication de l'anneau de base, la multiplication externe ou interne de `(P.meth` *self*`)`) avant de plonger le résultat dans les polynômes récursifs via `Base` ou `compose`.

En fait, cette méthode `mult` elle-même pourrait être généralisée, l'ensemble des méthodes binaires internes de `P` s'étendent aux polynômes récursifs suivant le même schéma.

On peut alors fabriquer la collection des polynômes récursifs. Cette collection est elle-aussi paramétrée par une collection (paramétrée) de polynômes :

```
module PolRecCollec (P:Poly_sig) : Poly_sig =
  struct
    module Aux = PolRecEspece(P)
    type 'c t = 'c Aux.t
    let meth = new Aux.espece
  end
```

On peut ensuite construire la collection paramétrée `PolRecCollec(PolyCreux)`.

5.4 Limites du prototype vis-à-vis de FocKer

On présente ici les limites du prototype OCaml par rapport aux exigences formulées dans FocKer. Grâce à une syntaxe concrète, on peut cependant espérer masquer ces problèmes à l'utilisateur et les traiter pour lui.

5.4.1 Ordre supérieur et relation d'inclusion entre interface

Dans l'exemple 5.3.2, la collection des polynômes récursifs est une collection paramétrée du second ordre : elle prend en argument une collection elle-même paramétrée par une collection de base. On suppose qu'on a :

```
module P= PolRecCollec(PolyCreux)
```

On peut construire `PolRecCollec(P)`, car `P` est de signature `Poly_sig`. Mais supposons maintenant qu'en fait `PolRecCollec (P :Poly_sig)` soit de type `Poly_sig2` où

```
module type Poly_sig2=
 sig
   type 'c t
   val meth: ('c #anneau) -> ('c,'c t) alg_poly2
 end
```

avec `alg_poly2` est une classe qui hérite de `alg_poly` en ajoutant de nouvelles méthodes, de telle sorte que le type `alg_poly2` soit une instance de `#alg_poly` distincte de `alg_poly`. Sous ces nouvelles hypothèses `P` est de type `Poly_sig2`. Si on peut construire `(PolyCreux.meth P.meth)`, par contre, comme `alg_poly2` et `alg_poly` sont distincts, on ne peut plus construire `PolRecCollec(P)`. On doit en fait construire :

```
module P2=
 struct
   type 'c t = 'c P.t
   let meth a = (P.meth :> alg_poly)
 end
```

On est donc obligé d'utiliser le mécanisme de sous-typage explicite des objets. Le mécanisme (polymorphisme de variables de rangée) utilisé jusqu'ici pour représenter la relation d'inclusion entre interfaces montre ici ces limites : pour pouvoir utiliser un mécanisme de variables de rangée sur cet exemple, il faudrait avoir une quantification existentielle sur les variables de rangée. [2]

[2] Ce problème vient du fait que les quantificateurs implicites sur les composantes d'un module passé en argument d'un foncteur apparaissent elles-même à l'intérieur d'une quantification, comme décrit en section 2.2.2.c, page 48.

Comme suggéré ici, on peut s'en sortir avec le sous-typage sur les objets. Mais c'est un peu lourd. En particulier, si on veut l'ordre supérieur, cela nécessite d'écrire complètement le sous-typage à la main, avec création d'un module à chaque conversion.

5.4.2 Renommage

OCaml ne permet pas de faire du renommage de méthodes. Il faut donc écrire à la main le renommage. Il s'écrit à l'aide de 3 classes : la classe dans laquelle les méthodes ont été renommées et deux classes pour les deux fonctions de conversion.

Pour l'exemple, supposons qu'on souhaite renommer la méthode `parse_int` en `lit_entier` de l'espèce `entiers_relatifs` vu précédemment. Il faut ici nommer l'espèce résultat : appelons la `foo`.

La classe `foo` s'écrit par recopie de la classe `entiers_relatifs`, en effectuant le renommage souhaité :

```
class foo =
object
  inherit [int] anneau_euclidien
  method reste_eucl x y = x mod y
  ...
  method lit_entier x =x
end
```

On peut alors créer la fonction de conversion entre `entier_relatifs` et `foo`. Cette fonction de conversion (appelée `entiers_relatifs2foo` est de type `'a #entier_relatifs_sig -> 'a foo_sig`). On passe par l'intermédiaire d'une classe du même nom qui effectue le renommage, en recopiant toutes les méthodes de l'objet passé en argument :

```
class ['a] entiers_relatifs2foo (o: 'a #entier_relatifs_sig) =
object
  method reste_eucl = o#reste_eucl
  method pgcd = o#pgcd
  ...
  method lit_entier = o#parse_int
end

let entiers_relatifs2foo = new entiers_relatifs2foo
```

Cette fonction peut en particulier s'appliquer à `EntiersRelatifs.meth`. Il faudra aussi créer la réciproque de cette fonction.

En pratique, le renommage est donc là-encore une opération un peu lourde à simuler en OCaml.

5.5 Conclusion

La mise au point du style de programmation présenté ici a été l'œuvre d'un travail collectif au sein du projet Foc (voir [Bou99a]).

En suivant les principes de développement présentés ici, Renaud Rioboo a implanté une bibliothèque assez conséquente. Les tests d'efficacité qu'il a réalisés montrent que OCaml est au moins aussi efficace qu'Axiom. Du point de vue de l'efficacité, il n'est donc pas déraisonnable d'utiliser OCaml pour effectuer ce genre de développement. Du point de vue de la sûreté de fonctionnement (rigueur de la sémantique, fiabilité de l'implantation ...), OCaml est bien supérieur à Axiom, mais c'est pour cela que OCaml a été choisi. Le principal défaut de ce prototype se situe donc plutôt au niveau de la simplicité du langage et de la syntaxe. Mais, Virgile Prevosto et Damien Doligez ont commencé à développer une syntaxe concrète pour FOC (cf. [Pre00]) qui tente de corriger ce défaut. Ils ont fournit un traducteur de cette syntaxe vers du code source OCaml en suivant le style présenté ici.

Il est intéressant de noter que, hormis l'exemple des polynômes récursifs, chaque collection (y compris les collections paramétrées) correspond à un unique `new` sur une classe. Autrement dit, dans la bibliothèque suggérée ici, toutes les collections peuvent être engendrées à la compilation. Cela signifie en particulier que cette bibliothèque pourrait être écrite dans un système de module (à la OCaml) avec de l'héritage.

Chapitre 6

Implantation en Coq

On représente maintenant FocKer en Coq. On va se donner un type des espèces, un type des collections et un type des interfaces. On pourra donc étudier en Coq les propriétés de ce codage. De ce point de vue, ce codage est profond (un “deep embedding”). En même temps, on ne souhaite pas recoder Coq à l'intérieur de lui-même. Les types et les termes associés aux méthodes de FocKer sont donc en fait des termes Coq, manipulés avec la logique de Coq. De ce point de vue, le codage est superficiel (“shallow embedding”) puisqu'on ne recode pas entièrement un système logique.

L'avantage de ce codage est qu'on peut à la fois l'étudier d'un point de vue méta-théorique en Coq, et qu'il est utilisable pour représenter des espèces et des collections en Coq. Ce codage peut donc être vu comme une “extension” de Coq restant à l'intérieur de Coq. On obtient ainsi assez simplement un système de spécification, dont la cohérence et l'implantation sont garanties par Coq. L'inconvénient de cette technique est que les calculs inhérents à notre système passent toujours par l'intermédiaire de Coq, et coûtent donc relativement cher.

Comme on l'a déjà fait remarquer, le système interface/collection correspond à un système d'enregistrements avec types de champs dépendants. La première étape consiste donc à décrire un tel système en Coq. On introduit donc pour cela deux types : le type `sign` des signatures d'enregistrements, correspondant donc aux interfaces, et le type `Drecord` des enregistrements, correspondant aux collections. Ensuite on créera un nouveau type appelé `mixDrec` afin de représenter les espèces. Ce codage a été présenté dans [Bou00].

Le lecteur trouvera au début de l'index les références aux pages introduisant les principaux types de ce codage en Coq.

6.1 Des enregistrements pour Coq codés en Coq

6.1.1 Présentation informelle des Drecords

Un *Drecord* peut-être vu comme une fonction d'un ensemble fini d'étiquettes vers un ensemble de champs. *Sélectionner un champ*, c'est appliquer cette fonction à une étiquette donnée.

6.1.1.a Notations

L'ensemble de la théorie des *Drecord* est paramétrée par un type `'A` (de sorte `Set`) qui est le type des étiquettes, et dont on suppose l'égalité décidable. On note `'A`* le type des listes d'étiquettes, et on note $\emptyset$ la liste vide. Soient l et l' des listes d'étiquettes et a une étiquette donnée. On suppose que les définitions suivantes ont été préalablement formalisées :

- $\in$: $(a \ \in \ l)$ est vrai ssi a est un élément de l.
- $\supset$: $(l \ \supset \ l')$ est vrai ssi l contient l'.
- $\not\pitchfork$: $(l \ \not\pitchfork l')$ est vrai ssi l et l' n'ont aucun élément commun.
- $\widehat{.}$: $\widehat{l}$ est vrai ssi l ne contient pas 2 occurrences d'une même étiquette.
- $\setminus$: $(l \setminus l')$ est une fonction qui retourne une liste des étiquettes de l qui n'appartiennent pas à l'. L'ordre des éléments de $(l \setminus l')$ sera précisé dans 6.2.4.b.

Le code Coq présenté ici est en "mode arguments implicites" : certains arguments des fonctions sont laissés implicites (ils sont inférés par le vérificateur de type, voir 3.2.2.b). Lorsque Coq ne peut les inférer, ils peuvent être donnés explictement avec la position n! où n est la position de l'argument.

6.1.1.b Signatures de Drecords et Drecords

Dans cette présentation informelle, les Drecords sont des termes et les signatures de Drecords sont des types de Drecords. On commence donc par introduire la sorte des signatures de Drecords, appelé $Sign$. C'est une sorte paramétrée par une liste d'étiquettes.

Définition 6.1 (Enregistrements associatifs à droite) *On appelle* signatures d'enregistrement *les termes entre* $\{.\}$*, contruits à partir des règles suivantes :*

$$Sign : {}^{\mathtt{e}}\!\mathtt{A}^* \to Type$$

$$\frac{}{\{\} : Sign_{\emptyset}}\ \mathbf{Esig} \qquad \frac{T : Type \quad a \notin l \quad F : T \to Sign_l}{\{a : T\ ;\ F\} : Sign_{a;l}}\mathbf{Csig}$$

De même, on appelle enregistrements *les termes entre* $<.>$*, contruits à partir des règles suivantes :*

$$\frac{}{<> : \{\}}\ \mathbf{Eimpl} \qquad \frac{\{a : T\ ;\ F\} : Sign_{a;l} \quad x : T \quad i : (F\ x)}{<a = x\ ;\ i> : \{a : T\ ;\ F\}}\mathbf{Cimpl}$$

Ce système d'enregistrements est dit associatif à droite (terminologie de [Pol00]), car on construit ces enregistrements en ajoutant des champs à gauche (champ étiqueté a de la règle `Csig`). Ceci n'est pas très naturel, car les champs de droite peuvent dépendre de ceux de gauche. Cette dépendance est ici exprimée par la fonction `F` de la règle `Csig`. Parallèlement à notre travail, Pollack a étudié le codage de la notion d'enregistrements en théorie des types (cf. [Pol00]). Il présente un système plus naturel que le notre, mais qui malheureusement ne se code pas directement en Coq (voir la discussion qui suit en section 6.5.1).

Dans la définition 6.1, les étiquettes sont attachées au champs des Drecords et peuvent être considérées comme faisant partie de la définition des Drecords. Par exemple, ils ne sont pas touchés par α-conversion. En effet, ce sont des noms donnés par l'utilisateur pour sélectionner les champs d'un enregistrement depuis l'extérieur. Pour cette raison, comme expliqué dans [Har94] et [Cou98], les étiquettes ne peuvent pas être dans la même classe syntaxique que les autres identificateurs, car cela causerait des *"captures d'étiquettes"*. Si on assimile les étiquettes à des variables, la β-réduction sur $([x : Type]<y : Type; z : x>\ y)$ crée une capture : le remplacement de x par y dans cet enregistrement ajoute une dépendance du deuxième champ sur le premier, ce qui est incorrect.

Les dépendances entre champs doivent donc être exprimées par des variables liées et non directement par des étiquettes. Ici, elles sont exprimées par l'abstraction "$F : T \to Sign_l$" de la règle `Csig`.

Une version naïve de cette définition n'aurait pas paramétré *Sign* par la liste des étiquettes. Une telle définition souffrirait de ce qu'on peut appeler "*une capture de structure*". Ceci signifie qu'avec une telle définition, on n'aurait pas eu les propriétés structurelles attendues pour les Drecords, parce que ces propriétés auraient été "capturées" (au même sens que pour les captures de variables) par l'abstraction dans la règle `Csig`. Par exemple, il n'y aurait pas eu de notion de *nombre de champs* d'une signature. En effet, $F : T \rightarrow Sign$ signifierait que pour chaque x de type T, le nombre de champs de $(F\ x)$ peut être différent.

Dans notre définition, le fait que *Sign* soit paramétrée par une liste d'étiquettes oblige chaque $(F\ x)$ à avoir le même ensemble d'étiquettes, dans le même ordre, pour chaque x. En particulier, le nombre de champs correspond à la taille de la liste en paramètre de *Sign*.

Dans un Drecord, toutes les étiquettes sont distinctes 2 à 2. C'est en fait seulement une propriété sémantique qui garantit que la sélection d'un champ est non ambigüe. Toutes les opérations sur les Drecords peuvent en fait être définies indépendament de cette condition. C'est le choix suivi dans l'implantation présentée ici. Lors d'un premier essai, on avait codé les Drecords en considérant à la fois les aspects opérationnels et cet aspect sémantique. Mais les preuves sont rapidement devenues pénibles à faire, car on devait montrer à chaque pas de raisonnement l'unicité des étiquettes. En séparant ces aspects, on montre la conservation de l'unicité des étiquettes seulement là où c'est nécessaire.

On va donc définir une notion plus générale que celle de Drecords, dans laquelle les étiquettes ne seront pas supposées distinctes. Nous appelons *télescope* cette notion plus générale, car cette notion ressemble à celle introduite dans [dB91] pour exprimer des dépendances entre contextes. Les opérations sur les Drecords sont construites en restreignant les opérations des télescopes grâce à des gardes. Ces gardes correspondent en fait aux prédicats notés $\in$, $\supset$, $\not\pitchfork$ et $\frown$. Elles sont décidables (sous l'hypothèse de la décidabilité de l'égalité des étiquettes) : elles peuvent donc être déchargées par le vérificateur de type de Coq.

6.1.1.c Codage des gardes sur les listes

Pour exprimer le fait que ces gardes expriment seulement des propriétés sémantiques des Drecords, on les a placées dans `Prop`, mais on aurait aussi bien pu les placer dans `Set` ou `Type`. Comme elles sont décidables, elles peuvent renvoyer 2 valeurs : le type `True` ou le type `False`. Elles sont implantées par des instanciations de l'un des 2 prédicats `AllD` et `ExD` ci-dessous. Ces deux prédicats prennent en paramètres une propriété `P` décidable et une liste `l`. Le prédicat `AllD` indique si la propriété `P` est vérifiée par tous les éléments de `l`. Et le prédicat `ExD` indique si la propriété est vérifiée par un élément de `l`.

```
Variable P:A->(list A)->Prop.
Hypothesis P_dec: (x:A; l:(list A)){(P x l)}+{~(P x l)}.

Fixpoint AllD[l:(list A)]: Prop :=
   Cases l of
      nil => True
    | (cons a m) => if (P_dec a m) then [_](AllD m) else [_]False end.

Fixpoint ExD[l:(list A)]: Prop :=
   Cases l of
      nil => False
    | (cons a m) => if (P_dec a m) then [_]True else [_](ExD m) end.
```

Le fait que les gardes soient des types (et non pas des termes), permet de les utiliser pour exprimer des fonctions partielles. Comme les seules valeurs possibles de ces gardes sont `True` et `False`, le seul terme possible de ces gardes est `I` (l'unique constructeur du type `True`). La construction d'une preuve pour décharger ces gardes est donc triviale.[1]

6.1.1.d Sélection des champs

En suivant le style informel, on donne maintenant la description de la sélection des champs.

$$\frac{r : \{a : T \ ; \ F\}}{r.a : T}\,\mathbf{Fte} \qquad \frac{b \neq a \qquad i.b : U}{<a = x \ ; \ i>.b : U}\,\mathbf{Ftn}$$

Sélection des champs (règles de typage)

$$\frac{}{<a = x \ ; \ i>.a \overset{def}{=} x}\,\mathbf{Fce} \qquad \frac{b \neq a}{<a = x \ ; \ i>.b \overset{def}{=} i.b}\,\mathbf{Fcn}$$

Sélection des champs (règles de calcul)

6.1.1.e Drecords en Coq

On définira formellement en section 6.1.2.e le type `sign` de sorte `Type`, et le type `Drecord` de type `sign`→`Type`.

Une *signature de Drecord* est alors par définition un terme de type `sign`. De même, un *Drecord* est par définition un terme de type `(Drecord s)`, où `s` est un terme de type `sign`. La fonction `sign_l` retournant la liste des étiquettes d'une signature de Drecord sera aussi donnée en 6.1.2.e.

La sélection des champs est définie par deux fonctions, `field` et `fieldT`, qui prennent un Drecord `D` et une étiquette `a` comme arguments. Le terme `(field D a)` est le champ associé à `a` dans `D` et `(fieldT D a)` est le type de `(field D a)`. Ci-dessous, on note $\mathfrak{L}_{\mathtt{s}}$ au lieu de `(sign_l s)`.

```
fieldT: (s:sign) (Drecord s)->A->Type
field:(s:sign ; i:(Drecord s) ; a:A ; H:(a∈𝔏s))(fieldT i a)
```

Remarquons que l'expression `(fieldT i a)` a un sens seulement si `a` est une étiquette du Drecord `i`. Elle n'est pas censée être directement employée par l'utilisateur de cette théorie.

On considère maintenant le codage de `sign` et `Drecord`. Comme mentionné ci-dessous, on traitera le problème de l'unicité des étiquettes dans un deuxième temps. On introduit tout d'abord la notion de télescope.

6.1.2 Télescopes

L'idée d'utiliser le concept de *télescope* pour exprimer de enregistrements avec types des champs dépendants est apparu dans [Pot99]. La notion de télescope elle-même a été introduite par [dB91].

Nous présentons ici des télescopes avec étiquettes. Leur implantation diffère de celle de [Pot99] (parce que de notre point de vue, celle-ci souffrait d'une *capture de structure*). Mais nous conservons le nom de télescopes.

[1]En utilisant les grammaires extensibles de Coq, on peut même "masquer" (voir 6.3.1) la partialité des fonctions ainsi créées : le vérificateur de type déclenchera une erreur de typage lorsque ces fonctions seront appliquées en dehors de leur domaine.

6.1.2.a Paire avec champs dépendants

Une paire avec champs dépendants est une sorte de produit cartésien : elle est codée par une somme dépendante et deux projections canoniques.

```
Structure dpair[T1:Type; T2:T1->Type] : Type:= {x1: T1; x2: (T2 x1)}.
```

Remarquons que les différentes occurences de `Type` dans cette définition dénotent différents *Type_n* où n est un niveau implicite d'univers (voir 3.2.1.b). En notant explicitement les niveaux d'univers, la définition précédente devient :

```
Structure dpair[T1:Type_i; T2:T1->Type_j]: Type_k:={x1: T1; x2: (T2 x2)}.
```

où $i < k$ et $j < k$. Ce mécanisme (invisible à l'utilisateur) empêche la construction de paradoxes (cf. [Wer94] et [Coq86]).

Les télescopes sont définis par une itération de ce type `dpair` sur lui-même. Formellement, les télescopes sont définis via deux types : `sigtel` le types des "signatures de télescopes" et `impltel` le type des "implémentations de télescopes". La sorte de `sigtel` et `impltel` est `Type`, car les champs d'un télescope peuvent être dans `Set` ou dans `Prop` ou même dans `Type`. Mais à cause des contraintes d'univers, les champs de télescopes doivent être dans un univers strictement inférieur à celui des télescopes. Cela peut être génant dans certaines formes d'imbrication des enregistrements (cf. 6.3.2).

6.1.2.b Signatures de télescopes

Le type des signatures de télescopes, `sigtel`, est défini par récurrence sur les listes. Il utilise deux types auxiliaires : `EsigT` un type avec un unique élement qui représente la signature vide, et `(FunIn a X)` le type étiqueté par `a` des fonctions à valeur dans `X`.

```
Inductive EsigT: Type := Esig_: EsigT.

Structure FunIn[a:A; X:Type]: Type:=
   {dom: Type; fun:> (dom->X)}.
```

```
Fixpoint sigtel[l:(list A)]:Type :=
  Cases l of
    nil => EsigT
  | (cons a m) => (FunIn a (sigtel m))
  end.
```

On rappelle que le symbole ">" derrière `fun` dans `FunIn` déclare que la projection `fun` est une cœrcion Coq vers `FUNCLASS` (voir 3.2.2.b). Sans cette cœrcion, si `s` est un terme de type `(FunIn a X)` et `x` un terme de type `(dom s)`, le terme `(s x)` est mal typé, car `(FunIn a X)` n'est pas un type de fonction. Avec cette cœrcion, ce terme est typable est signifie en fait `((fun s) x)`.

Le type `FunIn` généralise en fait la règle `Csig` de la définition 6.1. Ainsi, les règles de cette définition s'écrivent de la façon suivante :

```
Definition Esig: (sigtel (nil A)) := Esig_.
Definition Csig: (a:A; l:(list A); T:Type)(T->(sigtel l))->(sigtel (cons a l))
  :=[a,l,T,f](Build_FunIn a f).
```

6.1.2.c Implémentations de télescopes

Les télescopes sont définis de la même manière, par itération sur la notion de paire dépendante :

```
Inductive EimplT: Type :=
  Eimpl_: EimplT.

Structure dpairT
 [T:Type,f:T->Type;a:A]: Type :=
   {dprojT1: T; dprojT2: (f dprojT1) }.
```

```
Fixpoint impltel[l:(list A)]:(sigtel l)->Type:=
 <[l:(list A)](sigtel l)->Type>Cases l of
       nil => [_]EimplT
     | (cons a m) =>
         [s](dpairT [x:(dom s)](impltel (s x)) a)
 end.
```

Ces deux types récursifs `sigtel` et `impltel` auraient pu être définis autrement en utilisant les types inductifs de Coq, ce qui aurait donné une définition plus proche de la présentation informelle. Par exemple, `sigtel` aurait pu être donné par :

```
Inductive sigtel: (list A)->Type :=
   Esig: (sigtel (nil A))
 | Csig: (a:A; l:(list A); dom:Type)(dom->(sigtel l))->(sigtel (cons a l)).
```

Mais, pour les raisons invoqués en section 3.2.2.c, cette définition uniquement inductive est moins facile à utiliser. De plus, définir `impltel` par un type inductif exige de le placer dans un univers au-dessus de `sigtel`, ce qui est contraire à l'intuition (et pose quelques difficultés). En définissant `impltel` de manière récursive, celui-ci se trouve dans un univers de moins que `sigtel`.

Ainsi, dans toute cette implémentation, nous n'utilisons que des types inductifs qui ne mélangent pas "récursivité" et types dépendants (excepté pour `cont0` et `subsig`, ci-dessous). Pour cela, on utilise la technique présentée page 86 en section 3.2.2.c.

6.1.2.d Sélection des champs

La sélection des champs est réalisée grâce à deux fonctions, `fieldsig` qui retourne le type du champ, et `fiedimpl` qui retourne ce champ. Ces deux fonctions ne sont pas totales. On introduit le type `Dummy` a un seul constructeur `foo` pour exprimer cette partialité.

```
Inductive Dummy: Type := foo: Dummy.

Hypothesis eqA_dec:(x, y:A){x=y}+{~x=y}.

Fixpoint fieldsig[a:A; l:(list A)]:(s:(sigtel l))(impltel s)->Type :=
  <[l:(list A)](s:(sigtel l))(impltel s)->Type>Cases l of
      nil => [_;_]Dummy
    | (cons b m) => [s;i]
         if (eqA_dec a b) then
           [_](dom s)
         else
           [_](fieldsig a (dprojT2 i))
  end.

Fixpoint fieldimpl[a:A; l:(list A)]:(s:(sigtel l); i:(impltel s))(fieldsig a i):=
  <[l:(list A)](s:(sigtel l); i:(impltel s))(fieldsig a i)>Cases l of
      nil => [_;_]foo
    | (cons b m) =>
        [s;i]<[H:a=b+~a=b]if H then [_](dom s) else [_](fieldsig a (dprojT2 i))>
         if (eqA_dec a b) then
            [_](dprojT1 i)
         else
           [_](fieldimpl a (dprojT2 i))
  end.
```

6.1.2.e Des télescopes aux Drecords

Définition 6.2 *Une signature de Drecord est un triplet : une liste d'étiquettes* `sign_l`*, une preuve* `sign_l_df` *que cette preuve est sans-doublon, et un sigtel* `sign_p` *bâti sur* `sign_l`*.*

```
Structure sign: Type := {
    sign_l: (list A);
    sign_l_df: sign_l;
    sign_p:> (sigtel sign_l) }.
```

Remarquons que `sign_p` est une cœrcion Coq, de `sign` vers `sigtel`.

Définition 6.3 *Les Drecords sont des télescopes, bâtis sur une signature de Drecord :*

```
Definition Drecord[s:sign]:=(impltel s).
```

La transformation d'un télescope en Drecord n'est ainsi qu'une conversion de type.

Définition 6.4 *Les fonctions de sélection des champs sont :*

```
Definition fieldT:(s:sign)(Drecord s)->A->Type
  :=[s;i;a](fieldsig eqA_dec a i).

Definition field:(s:sign; i:(Drecord s); a:A; H:(a∈ℒs))(fieldT i a)
  :=[s;i;a;H](fieldimpl eqA_dec a i).
```

6.2 Opérations et relations entre signatures de Drecord

On présente ici les opérations et les propriétés de Drecords. On les introduit d'abord de manière informelle, avant d'en donner l'implantation au niveau des télescopes.

6.2.1 Relation de sous-signature et conversions entre Drecords

On encode ici la notion d'inclusion entre interfaces, et de conversions au niveau des collections.

6.2.1.a Présentation informelle

Entre signatures de Drecord, il y a une relation naturelle appelée relation de sous-signature : s_1 est une sous-signature de s_2 (on notera informellement $s_1{:}{\succ}s_2$), si s_1 peut-être transformé en s_2 en oubliant ou permutant certains champs. Dans l'implantation en Coq, cette relation est appelée `subsign`.

Cette relation est donnée par les règles suivantes :

$$:\succ\ : (l_1, l_2 : \mathtt{A}^*) Sign_{l_1} \times Sign_{l_2} \rightarrow Type$$

$$\frac{}{s{:\succ}\{\}}\ \mathbf{\_E} \qquad \frac{(x:T)(F_1\ x){:\succ}(F_2\ x)}{\{a:T\ ;\ F_1\}{:\succ}\{a:T\ ;\ F_2\}}\ \mathbf{\_C}$$

$$\frac{(x:T)(F\ x){:\succ}s}{\{a:T\ ;\ F\}{:\succ}s}\ \mathbf{\_lift} \qquad \frac{s_1{:\succ}s_2 \quad s_2{:\succ}s_3}{s_1{:\succ}s_3}\ \mathbf{\_trans}$$

$$\frac{(x:T_1; y:T_2)(F_1\ x\ y){:\succ}(F_2\ y\ x)}{\{a:T_1\ ;\ [x:T_1]\{b:T_2; (F_1\ x)\}\}{:\succ}\{b:T_2\ ;\ [y:T_2]\{a:T_1; (F_2\ y)\}\}}\ \mathbf{\_swap}$$

Relation de sous-signature

En fait, cette présentation informelle souffre d'une capture de structure : on ne peut pas considérer que la relation de sous-signature est une suite de transformations élémentaires sur les champs, indépendante des quantifications à l'intérieur des règles précédentes... Cette notion de suite de transformations est cependant nécessaire pour définir la fusion de signature (cf. 6.2.4), parce que l'utilisation des règles _swap et _trans va avoir une influence sur l'ordre des champs dans la signature résultat.

Mais cette présentation est largement suffisante pour comprendre intuitivement quelles sont ces transformations élémentaires (voir `subsig` pour les détails de l'implantation).

Associée à cette relation de sous-signature, il y a une fonction de conversions entre Drecords. Si $s_1{:\succ}s_2$ et $i : s_1$, on note informellement $i_{|\succ s_2}$ le Drecord résultant de cette conversion de i sur s_2. Informellement, cette conversion est définie par induction sur une preuve de $s_1{:\succ}s_2$, dans la table suivante :

$s_1 : Sign_{l_1}$	$s_2 : Sign_{l_2}$	$s_1{:\succ}s_2$	$i : s_1$	$i_{\vert\succ s_2} : s_2$
s	$\{\}$	_E	i	$<>$
$\{a:T\ ;\ F_1\}$	$\{a:T\ ;\ F_2\}$	_C	$<a=x\ ;\ i>$	$<a=x\ ; i_{\vert\succ(F_2\ x)}>$
$\{a:T\ ;\ F\}$	s	_lift	$<a=x\ ; i>$	$i_{\vert\succ s}$
s_1	s_3	_trans	i	$(i_{\vert\succ s_2})_{\vert\succ s_3}$
$\{a:T_1\ ;\ [x:T_1]\{b:T_2; (F_1\ x)\}\}$	$\{b:T_2\ ;\ [y:T_2]\{a:T_1; (F_2\ y)\}\}$	_swap	$\{a=x\ ;\ <b=y\ ;\ i>\}$	$\{b=y\ ;\ <a=x\ ;\ i_{\vert\succ(F_2\ y\ x)}>\}$

6.2.1.b Propriétés

La relation de sous-signature est implémentée par `subsign` (définie plus loin), dont le type est `sign->sign->Type`. Et la fonction de conversion est implémentée par `coerce` de type
`(s1,s2:(sign))(subsign s1 s2)->(Drecord s1)->(Drecord s2)`

Théorème 6.1 *La relation* `subsign` *est un préordre, dont la relation d'équivalence associée peut être facilement redéfinie : elle correspond à* `subsign` *sur les signatures de longueur égale.*

Preuve Les propositions suivantes ont été démontrées en Coq :

```
(* Propriétés de préordre*)
Lemma subsign_refl:(s:sign)(subsign s s).

Lemma subsign_trans:(s1,s2,s3:sign)(subsign s1 s2)->(subsign s2 s3)->(subsign s1 s3).

(* Propriétés de la relation d'équivalence associée  *)
Lemma subsign_antisym:(s1,s2:sign)(subsign s1 s2)->(subsign s2 s1)
      ->(length 𝔏s1)=(length 𝔏s2).

Lemma subsign_sym:(s1,s2:sign)(subsign s1 s2)
      ->(length 𝔏s1)=(length 𝔏s2)->(subsign s2 s1).
```

□

La fonction de conversion préserve le comportement extensionnel des Drecords (égalité des champs sélectionnés via la même étiquette).

Théorème 6.2 *Soient s_1 et s_2 2 signatures telles que $s_1 :\succ s_2$. Alors la fonction* `coerce` *a la propriété suivante (appelée* `subsign_ext`*) :*
Soit i un (`Drecord` s_1)*, alors pour toute étiquette a de $\mathfrak{L}_{s_2}$, la sélection (fontion* `field`*) du champ a sur i est égale à celle sur* (`coerce` i)*.*

$$\begin{tikzcd}
s_1 \arrow[rr, "\texttt{subsign}"] & & s_2 \\
i \arrow[rr, "\texttt{coerce}"] \arrow[dr, dashed, "\texttt{field}"'] & & i|_{\succ s_2} \arrow[dl, dashed, "\texttt{field}"] \\
& {} &
\end{tikzcd}$$

Preuve Supposons définie la relation `eqT_dep`[2], de l'égalité dépendante sur les types. Le théorème `subsign_ext` a été prouvée en Coq :

```
Lemma subsign_cont:(s1,s2:sign)(subsign s1 s2)->𝔏s1⊃𝔏s2.

Lemma subsign_ext_T: (s1,s2:sign; H:(subsign s1 s2); i:(Drecord s1); a:A)(a∈𝔏s)
       ->(fieldT i a)==(fieldT (coerce H i) a).

Theorem subsign_ext: (s1,s2:sign; H1:(subsign s1 s2); i:(Drecord s1); a:A; H2:(a∈𝔏s2))
      (eqT_dep 2![T:Type]T (field i (cont_incl (subsign_cont H1) H2)³)
                           (field (coerce H1 i) H2)).
```

□

[2]En mode “arguments implicites”, c'est la définition suivante :

```
Inductive eqT_dep[U :Type ; P :U->Type ; p :U ; x :(P p)] : (q :U)(P q)->Prop :=
          eqT_dep_intro : (eqT_dep x x).
```

[3]preuve de $a \in \mathfrak{L}_{s1}$

Cette dernière propriété (prouvée en Coq) garantie la correction de l'implantation de la notion de sous-signature vis-à-vis de sa sémantique. En particulier, quelles que soient les "preuves" que 2 signatures données sont en relation de sous-signature, les conversions associées au niveau des Drecords sont équivalentes (vis-à-vis de la sélection de champs).

6.2.2 Implémentation

Pour exprimer le fait que la relation "`s1` est sous-signature de `s2`" est définie comme une succesion élémentaire de transformations sur les signatures (oubli de champs, permutation de champs, etc...) on doit d'abord définir ces transformations élémentaires sur des listes d'étiquettes. Ainsi `subsign_cont0` donne le squelette de ces transformations élémentaires pour passer de $\mathfrak{L}_{\mathtt{s1}}$ à $\mathfrak{L}_{\mathtt{s2}}$. Et cette succession de transformations est exprimé au niveau des sigtels par `subsign_p` :

```
Structure subsign[s1,s2:sign]: Type :=
  { subsign_cont0: (cont0 𝔏s1 𝔏s2);
    subsign_p:> (subsig subsign_cont0 s1 s2)}.
```

Où, `cont0` est défini par :

```
Inductive cont0:(list A)->(list A)->Set :=
   cont0_nil: (l:(list A))(cont0 l (nil A))
 | cont0_cons: (a:A)(l1,l2:(list A))(cont0 l1 l2)->
                      (cont0 (cons a l1) (cons a l2))
 | cont0_lift:(a:A; l,m:(list A))(cont0 l m)->(cont0 (cons a l) m)
 | cont0_swap:(a1,a2:A; l1,l2:(list A))
     (cont0 l1 l2)->(cont0 (cons a1 (cons a2 l1)) (cons a2 (cons a1 l2)))
 | cont0_trans: (l1,l2,l3:(list A))(cont0 l1 l2)->(cont0 l2 l3)->(cont0 l1 l3).
```

Et, `subsig` est défini par :

```
Inductive subsig: (l1, l2:(list A))(cont0 l1 l2)->(sigtel l1)->(sigtel l2)->Type :=
   subsig_E: (l:(list A); s:(sigtel l))(subsig (cont0_nil l) s Esig)
 | subsig_C: (a:A; l1, l2:(list A); T:Type; f:T ->(sigtel l1); g:T ->(sigtel l2);
     H:(cont0 l1 l2))
      ((x:T)(subsig H (f x) (g x)))->(subsig (cont0_cons a H) (Csig a f) (Csig a g))
 | subsig_lift: (a:A; l1, l2:(list A); T:Type; f:T ->(sigtel l1); s:(sigtel l2);
     H:(cont0 l1 l2))
       ((x:T)(subsig H (f x) s))->(subsig (cont0_lift a H) (Csig a f) s)
 | subsig_swap:
     (a1, a2:A; l1, l2:(list A); T1, T2:Type;
       f:T1 -> T2 ->(sigtel l1); g:T2 -> T1 ->(sigtel l2))
       (H:(cont0 l1 l2))
       ((x:T1; y:T2)(subsig H (f x y) (g y x))) ->
       (subsig (cont0_swap a1 a2 H)
           (Csig a1 [x:T1](Csig a2 [y:T2](f x y)))
           (Csig a2 [y:T2](Csig a1 [x:T1](g y x))))
 | subsig_trans:
     (l1, l2, l3:(list A); s1:(sigtel l1); s2:(sigtel l2); s3:(sigtel l3);
          H1:(cont0 l1 l2); H2:(cont0 l2 l3))
          (subsig H1 s1 s2) -> (subsig H2 s2 s3) -> (subsig (cont0_trans H1 H2) s1 s3).
```

La fonction de conversions entre impltels, `coerce_impltel`, est alors une triviale induction sur une preuve de subsig. Voici son type :

```
Definition coerce_impltel: (l1, l2:(list A); s1:(sigtel l1); s2:(sigtel l2);
 H:(cont0 (eq A) l1 l2))(subsig H s1 s2) -> (impltel s1) -> (impltel s2).
```

On peut alors définir la conversion sur les Drecords :

```
Definition coerce: (s1,s2:(sign))(subsign s1 s2)->(Drecord s1)->(Drecord s2)
  :=[s1,s2;H;i](coerce_impltel H i).
```

6.2.3 Etendre une signature de Drecord en lui ajoutant des champs

Cette opération correspond à l'enrichissement aux niveaux des espèces.

6.2.3.a Description

Etendre une signature s, c'est ajouter de **nouveaux** champs à la fin (à droite) de s. Ces nouveaux champs peuvent éventuellement dépendre des champs de s. Donc ce morceau de signature (les nouveaux champs) doit être exprimé dans un contexte contenant les champs de s, c'est-à-dire un Drecord de signature de s[4]. Ce morceau de signature sera donc exprimé au travers d'une fonction de type $s \to Sign_l$ où `l` est une liste sans doublon de nouvelles étiquettes.

Informellement, cela s'écrit :

$$Flat : (l_1, l_2 : {}^{\mathfrak{c}}\mathtt{A}^*)(l_1 \not\cap l_2) \to (s : Sign_{l_1})(r : s \to Sign_{l_2}) \to Sign_{l_1 \wedge l_2}$$

$s : Sign_{l_1}$	$G : s \to Sign_{l_2}$	$(Flat\ G) : Sign_{l_1 \wedge l_2}$
$\{\}$	G	$(G <>)$
$\{a : T; F\}$	G	$\{a : T; [x : T](Flat\ [i : (F\ x)](G <a = x; i>))\}$

Flat est en réalité implantée par `flatsig`.

Théorème 6.3 *L'extension d'une signature* `s` *est une sous-signature de* `s`.

Preuve Ceci a été prouvé en Coq :

```
Variable s:sign.
Variable l:(list A)).
Hypothesis H1:l̂.
Hypothesis H2:(l∩̸𝔏s).
```

```
Definition flatsign: ((Drecord s)->(sigtel l))->sign.

Theorem flatsign_subsign:
  (f:(Drecord s)->(sigtel l))(subsign (flatsign f) s).
```

□

Par construction $\mathfrak{L}_{\mathtt{flatsign}}$ est la concaténation de $\mathfrak{L}_{\mathtt{s}}$ et de `l`. Comme cas particulier d'extension de signature, on peut prendre la concaténation de signature :

```
Variables s1,s2:sign.
Hypothesis H:(𝔏s2∩̸𝔏s1).
```

```
Definition concatsign:=
 (flatsign (sign_l_df s2)[5] H [_:(Drecord s1)](sign_p s2)).

Lemma subsign_concatsign_1: (subsign concatsign s1).
Lemma subsign_concatsign_2: (subsign concatsign s2).
```

[4]Les télescopes ont justement été introduit pour formaliser le concept de contexte.

[5]preuve de $\widehat{\mathfrak{L}_{\mathtt{s2}}}$.

6.2.3.b Implémentation

L'opération `flatsign` est implantée par `flatsig` au niveau des télescopes. Pour des raisons de lisibilité, on dénote `(impltel s)` par $\overline{s}$ et `(coerce_impltel H i)` par $i_{|\succ H}$. `flatsig` est une récurrence triviale sur `l1`, dont le type est :

```
flatsig: (l1,l2:(list A))(s1:(sigtel l1))s1->(sigtel l2))->(sigtel (l1^l2)).
```

Voici quelques propriétés importantes de `flatsig` utilisées pour établir la fusion de signatures. Les termes "`cont0`" sont cachées dans la notation "`...`".

- Une extension de la signature `s1` par une signature `s2` qui ne dépend pas de `s1` est sous-signature de `s2`.
  ```
  Lemma flatsig_constant: (l1,l2:(list A); s1:(sigtel l1); s2:(sigtel l2))
    (subsig ... (flatsig [_:s1]s2) s2).
  ```
- L'extension est compatible avec la relation de sous-signature.
  ```
  Lemma subsig_flatsig_coerce: (l1,l2:(list A); s1:(sigtel l1); s2:(sigtel l2); ...;
   H:(subsig ... s2 s1); l3:(list A); f:s1->(sigtel l3))
     (subsig ... (flatsig [i](f i|>H)) (flatsig [i](f i))).

  Lemma subsig_flatsig: (l1,l2,l3:(list A); s:(sigtel l1); f:s->(sigtel l2);
   g:s->(sigtel l3); ...)
    ((i:s)(subsig ... (f i) (g i)))->
      (subsig ... (flatsig f) (flatsig g)).
  ```
- La permutation d'un paquet de champs est compatible avec la relation de sous-signature :
  ```
  Lemma subtel_Csig_flatsig :
   (l1:(list A); s:(sigtel l1); a:A; l2,l3:(list A); T:Type;
     f:T->s->(sigtel l2); g:T->(sigtel l3); ...)
     ((x:T)(subsig ... (flatsig [i](f x i)) (g x)))
       ->(subsig ... (flatsig [i](Csig a [x](f x i))) (Csig a g)).
  ```

6.2.4 Fusion de signatures

Cette opération est nécessaire pour implanter la fusion d'espèces. Comme dans la fusion d'espèces, il s'agit de fusionner deux signatures en assimilant les champs de même étiquette. Cela impose que ces étiquettes communes correspondent à des champs de même type. Ainsi, la fusion de signatures est plus générale que l'opération de concaténation précédemment définie qui ne peut être appliquée qu'à des signatures n'ayant pas d'étiquettes en commun.

6.2.4.a Description

Cette contrainte de compatibilité sur les étiquettes communes impose l'existence d'une sous-signature (possiblement vide) commune à ces deux signatures, et contenant toutes les étiquettes communes aux 2 signatures. Cette contrainte n'étant pas décidable (car il faudrait décider de l'égalité entre type), on demande à l'utilisateur de la décharger complètement.

Pour implanter cette opération, on a d'abord écrit une fonction inverse de $Flat$, c'est-à-dire, une fonction Ap telle que pour toutes signatures s_1 et s_2 avec $s_1{:}{\succ}s_2$, Ap factorise s_2 dans s_1 en exprimant les champs de s_1 qui ne sont pas dans s_2, en fonction de ceux de s_2.

Très informellement, Ap est définie par la table ci-dessous. On commence par définir la suppression du premier champ d'un enregistrement

$$\frac{}{<>_{|} \stackrel{def}{=} <> : \{\}} \qquad \frac{<a = x\ ;\ i> : \{a : T\ ;\ F\}}{<a = x\ ;\ i>_{|} \stackrel{def}{=} i\ :\ (F\ x)}$$

Pour définir Ap on raisonne par récurrence sur l'arbre de preuve de $s_1 :\succ s_2$. Chacun des cas ci-dessous correspond donc à une règle de formation d'une preuve de $s_1 :\succ s_2$.

$s_1 : Sign_{l_1}$	$s_2 : Sign_{l_2}$	$s_1 :\succ s_2$	$(Ap\ s_1) : s_2 \to Sign_{l_1 \backslash l_2}$
s	$\{\}$	_E	$[i : \{\}]s$
$\{a : T\ ;\ F_1\}$	$\{a : T\ ;\ F_2\}$	_C	$[i : \{a : T\ ;\ F_2\}](Ap\ (F_1\ i.a)\ i_{\mid})$
$\{a : T\ ;\ F\}$	s	_lift	$[i : s]\{a : T\ ;\ [x : T](Ap\ (F\ x)\ i)\}$
s_1	s_3	_trans	$(Ap\ (Flat\ [j : (Flat\ [k : s_3](Ap\ s_2\ k))](Ap\ s_1\ j_{\mid \succ s_2})))$
$\{a : T_1\ ;\ [x : T_1]\{b : T_2; (F_1\ x)\}\}$	$\{b : T_2\ ;\ [y : T_2]\{a : T_1; (F_2\ y)\}\}$	_swap	$[i : s](Ap\ (F_1\ i.a\ i.b)\ (i_{\mid})_{\mid})$

Formellement `Ap` est implantée par `apsign`, et on a les propriétés suivantes :

```
apsign: (s1,s2:sign)(subsign s1 s2)->(Drecord s2)->sign.

apsign_subsign_l:(s1,s2:sign; H:(subsign s1 s2))
    (subsign (flatsign ... [i:(Drecord s2)](apsign H i)) s1).

apsign_subsign_r:(s1,s2:sign; H:(subsign s1 s2))
    (subsign s1 (flatsign ... [i:(Drecord s2)](apsign H i))).
```

Une fois `apsign` defini, il est assez simple de définir `mergesign` :

```
Variables s,s1,s2:sign.
Hypothesis H1: (subsign s1 s).
Hypothesis H2: (subsign s2 s).
Hypothesis H3: (𝔏s2\𝔏s)∩̸(𝔏s1\𝔏s).

Definition mergesign :=
    (flatsign ... [i:(Drecord s)](concatsign 1!(apsign H1 i) 2!(apsign H2 i) H3)).
```

Théorème 6.4 *La fusion de 2 signatures est sous-signature de ces deux signatures.*

Preuve En Coq, on a prouvé les lemmes suivants :

```
Lemma subsign_mergesign_1: (subsign mergesign s1).
Lemma subsign_mergesign_2: (subsign mergesign s2).
```

$\square$

Par construction $\mathfrak{L}_{\texttt{mergesign}}$ est la concaténation de 3 listes : $\mathfrak{L}_{\texttt{s}}$, $(\mathfrak{L}_{\texttt{s1}} \backslash \mathfrak{L}_{\texttt{s}})$ et $(\mathfrak{L}_{\texttt{s2}} \backslash \mathfrak{L}_{\texttt{s}})$.

6.2.4.b Implémentation

La fonction **apsign** est définie au niveau des télescopes, par induction structurelle sur la preuve que $s_1 {:}{\succ} s_2$. La principale difficulté de cette définition (et même de tout ce développement) vient du cas **subsig_trans**.

Informellement, on définit **apsig** de type

$$(s_1 {:}{\succ} s_2) \rightarrow (\overline{s_2} \rightarrow (\texttt{sigtel}\ \mathfrak{L}_{s_1} \setminus \mathfrak{L}_{s_2}))$$

avec s_1 et s_2 sigtels, par induction sur une preuve que $s_1 {:}{\succ} s_2$. Ainsi le cas **subsig_trans** correspond à :

$$\begin{array}{|ll}
H_1 : & s_1 {:}{\succ} s_2 \\
f : & \overline{s_2} \rightarrow (\texttt{sigtel}\ \mathfrak{L}_{s_1} \setminus \mathfrak{L}_{s_2}) \\
H_2 : & s_2 {:}{\succ} s_3 \\
g : & \overline{s_3} \rightarrow (\texttt{sigtel}\ \mathfrak{L}_{s_2} \setminus \mathfrak{L}_{s_3}) \\
\hline
 & \overline{s_3} \rightarrow (\texttt{sigtel}\ \mathfrak{L}_{s_1} \setminus \mathfrak{L}_{s_3})
\end{array}$$

L'idée est de donner le terme suivant pour ce but :

$$[i : \overline{s_3}](\texttt{apsig}\ (\texttt{flatsig}\ [j : \overline{(\texttt{flatsig}\ g)}](f\ j_{|\succ s_2}))\ i)$$

Afin de pouvoir écrire "$[j : \overline{(\texttt{flatsig}\ g)}](f\ j_{|\succ s_2})$" (expression qu'on notera h désormais), **apsign_subsign_l** doit être prouvé en même temps que l'on construit **apsig** .

De plus, à cet instant **apsig** n'est pas applicable à `flatsig` h, car on est en train de définir **apsig** par induction structurelle. Heureusement, on peut en fait démontrer que `flatsig` h est une signature dont s_3 est un préfixe (au même sens que la notion de préfixe sur les mots). Ainsi, on commence en fait par définir cette relation préfixe entre signatures de télescopes (en montrant par exemple que si s_2 est un préfixe de s_1 alors $s_1 {:}{\succ} s_2$). Ensuite, on définit la fonction **apsig_pre** qui correspond à la fonction **apsig** restreinte à la relation préfixe : avec cette restriction du domaine d'application, **apsig_pre** est facilement définissable.

Enfin, on définit **apsig** comme la première projection de **apsig_dpair** dont le type est :

```
Definition apsig_dpair:(l1,l2:(list A); s1:(sigtel l1); s2:(sigtel l2);
 H:(cont0 l1 l2); H1:(subsig H s1 s2))
   (dpair [f:(impltel s2)->(sigtel (extract0 H))]
          (subsig (cont0_app_extract0 H) (flatsig [i](f i)) s1)).
```

où **extract0** est une fonction qui prend en argument une preuve `H` de **(cont0 l1 l2)** et qui retourne une liste contenant les éléments **l1** qui ne sont pas dans **l2**. L'ordre des éléments dans cette liste dépend de `H` (et en particulier des permutations dans `H`). En fait, ce que nous avons jusque là noté abusivement $l1 \setminus l2$ est **(extract0 H)**.

La fonction **apsig_dpair** est définie par induction structurelle, comme suggéré ci-dessus. Le terme correspondant au cas **subsig_trans** est finalement :

$$[i : \overline{s_3}](\texttt{apsig\_pre}\ (\texttt{flatsig}\ h)\ i)$$

6.2.5 Renommage

Le renommage décrit dans FocKer n'a pas été implanté. Mais à priori, il ne devrait pas poser de grandes difficultés, car les étiquettes ne servent pas à exprimer la dépendance entre champs.

6.3 Exemples d'utilisation de cette théorie des Drecords

6.3.1 Exemple de base

Dans cet exemple, on commence par définir le type des étiquettes, comme un type énuméré (fini), appelé `Label`, qui contient toutes les étiquettes dont nous aurons besoins. On aurait pu bien sûr définir un type d'étiquettes infini. Mais pour la lisibilité de l'exemple, c'est plus simple ainsi.

```
Inductive Label: Set := T: Label | EQ : Label | EQ_refl: Label | EQ_sym: Label
  | EQ_trans: Label | OP: Label | OP_EQ_l:Label | AP: Label | AP_EQ: Label.

Lemma eqLabel_dec : (x,y:Label){x=y}+{~x=y}.
  Intros x y; Case x; Case y; Auto; Right; Discriminate.
Defined.
```

On défini `setoid_sig` la signature de setoid, et le type `setoid` sera le type des Drecords sur cette signature. On utilise ici les grammaires extensibles de Coq pour masquer le mécanisme de gardes sur les listes : le système vérifie tout seul que la liste d'étiquettes est sans-doublon (l'utilisateur n'a pas de preuve à faire), et déclenche une erreur de typage en cas de violation de cette condition :

```
Definition setoid_sig:=(<eqLabel_dec>Sign
 T:Type;
 EQ: T->T->Prop;
 EQ_refl: (x:T)(EQ x x);
 EQ_sym: (x,y:T)(EQ x y)->(EQ y x);
 EQ_trans: (x,y,z:T)(EQ x y)->(EQ y z)->(EQ x z)).

Definition setoid:=(Drecord setoid_sig).
```

Grâce à une fonction `Build` de la théorie des Drecords, on crée une fonction pour construire un setoid à partir de l'ensemble de ses champs, donnés dans l'ordre où ils apparaissent dans la signature. Par exemple, les types eux-mêmes forment un setoid :

```
Definition Build_setoid:=(Build setoid_sig).

Definition Type_setoid: setoid :=
  (Build_setoid (refl_eqT Type) (sym_eqT Type) (trans_eqT Type)).
```

Les champs d'un Drecord sont sélectionnés via l'opérateur binaire `@`. Là encore, on a masqué la vérification que l'étiquette sélectionnée appartient bien au Drecord :

```
Definition carrier: setoid->Type:=[s](s @ T).
```

Le type `semigroup` est construit comme une extension de signature des setoids, en ajoutant une loi binaire, associative et compatible avec la relation d'équivalence :

```
Definition semigroup_on:=[s:setoid](Sigtel
     OP:((s @ T)->(s @ T)->(s @ T)) ;
     OP_EQ_l: (x,y,z:(s @ T))((s @ EQ) x y)->((s @ EQ) (OP x z) (OP y z)) ;
     OP_EQ_r: (x,y,z:(s @ T)) ((s @ EQ) x y)->((s @ EQ) (OP z x) (OP z y)) ;
     OP_ass : (x,y,z:(s @ T))((s @ EQ) (OP (OP x y) z) (OP x (OP y z)))).

Definition semigroup_sig:=(Flat [s](semigroup_on s)).
Definition semigroup:=(Drecord semigroup_sig).
```

On peut alors bénéficier des conversions de la théorie pour les transformer en cœrcions Coq.

```
Definition sgp_setoid: semigroup -> setoid := (coerce (FlatSub [s](semigroup_on s))).
Coercion sgp_setoid: semigroup >-> setoid.
```

De même, on construit le type `order` des setoids ordonnés. En fusionnant `semigroup_sig` et `order_sig` (avec `setoid_sig` en tant que sous-signature commune), on peut créer un type de Drecord `ordsg` qui peut ensuite être étendu afin de décrire les semigroupes ordonnés (cette dernière extension est nécessaire pour ajouter les hypothèses de compatibilité de la loi interne sur la relation d'ordre). Malheureusement, ces calculs sur les types sont trop gros pour l'implémentation actuelle de Coq. Ainsi en pratique, sélectionner un champ semble un problème exponentiel en fonction de la taille du Drecord.

6.3.2 Enregistrements imbriqués & Problèmes d'univers

On peut tout à fait créer des signatures dont les champs sont de types `(Drecord s)` où `s` est une signature. En effet, comme expliqué en 6.1.2, les Drecords sont dans un univers en-dessous de ceux des signatures, grâce au choix du codage de `impltel` par point fixe. Ce ne serait pas le cas avec un codage de `impltel` via un type inductif. Ceci permet donc de faire des enregistrements imbriqués. Par contre, si on reprend l'exemple des setoids précédent, on ne peut pas créer de Drecords de type `setoid` dont le support (champ d'étiquette `T`) serait lui-même le type `setoid`. Le type de `setoid` est dans le même univers que `setoid_sig`. Or le type associé à l'étiquette `T` dans `setoid_sig` est dans un univers de moins que `setoid_sig`. On ne peut donc pas "instancier" `T` par `setoid` sans créer une inconsistance d'univers.

Ainsi l'exemple de la section 4.2 des morphismes de setoids n'est pas directement transposable dans les Drecords. Etudions néanmoins les moyens d'exprimer cet exemple.

On commence par créer la signature des morphismes de setoids, puis son type de Drecord associé.

```
Section MapSetoid.
Variables A,B: setoid.

Definition Map_sig:= (<eqLabel_dec>Sign
  AP: (A->B) ;
  AP_EQ: (x,y:A)((A @ EQ) x y)->((B @ EQ) (AP x) (AP y))
  ).

Definition Map:=(Drecord Map_sig).
Identity Coercion Map2Drecord: Map >-> Drecord.

Definition Map_ap: Map -> A -> B := [m:Map](m @ AP).
Coercion Map_ap: Map >-> FUNCLASS.
```

Finalement, on fait de `Map_ap` la projection sur le champ `AP`, une cœrcion vers la classe des fonctions. Ainsi, les termes de type `Map` peuvent être considérés comme des fonctions. On définit alors l'extensionalité :

```
Definition ext:=[f,g:Map](x:A)((B @ EQ) (f x) (g x)).
```

On montre alors que `ext` est réflexive (lemme `ext_refl`) et antisymétrique (lemme `ext_sym`) et transitive (lemme `ext_trans`). Si on écrit *(Build_setoid ext_refl ext_sym ext_trans)*, on provoque une erreur de consistance d'univers.

Pour répondre néanmoins (partiellement) à ce problème, on a deux solutions. Premièrement, redéfinir une notion spéciale de "setoides" avec des types de la forme `(Drecord s)` comme support implicite (cf. `setoid'` ci-dessous). Deuxièmement "renommer à la main" toute la théorie des Drecords pour décrire les morphismes de setoids dans cette nouvelle théorie : cette théorie sera la théorie des Drecords d'un univers inférieur à ceux vus jusqu'à maintenant. Les types `sigtel`, `impltel`, `sign`, `Drecord` ... seront renommés en `sigtel'`, `impltel'`, `sign'`, `Drecord'`... Cette dernière solution est celle choisie par A. Saïbi pour sa catégorie des catégories. (cf. [Saï99]).

Dans la première solution on utilise le `Map` précedemment défini, et on construit le `setoid'` des `Map`.

```
Section Osetoid.

Variable s:(sign eqLabel_dec).

Definition setoid'_sig:=(<eqLabel_dec>Sign
   EQ: ((Drecord s)->(Drecord s)->Prop) ;
   EQ_refl : (x:(Drecord s))(EQ x x) ;
   EQ_sym : ((x,y:(Drecord s))(EQ x y)->(EQ y x)) ;
   EQ_trans : ((x,y,z:(Drecord s))(EQ x y)->(EQ y z)->(EQ x z))
   ).

Definition Build_setoid':=(Build setoid'_sig).

End Osetoid.

Definition Mapset: (Drecord (setoid'_sig Map_sig)):=
  (Build_setoid' ext_refl ext_sym ext_trans).

End MapSetoid.
```

Dans la deuxième solution, on oublie le `Map` précedemment défini. On en redéfinit un autre `Map'`, exactement de la même manière, basé sur une autre (mais identique) définition des Drecords. On aura donc défini `sign'`, `Drecord'`, mais aussi `field'` : on redéfinit l'ensemble de la théorie des Drecords. La nouvelle définition de `Map` est la suivante :

```
Variables A,B: setoid.

Definition Map'_sig := (<eqLabel_dec>Sign'
  AP: (A->B) ;
  AP_EQ: (x,y:A)((A @ EQ) x y)->((B @ EQ) (AP x) (AP y))
  ).

Definition Map':=(Drecord' Map'_sig).
```

On peut alors sans problème écrire :

```
Definition Map'set:setoid:=(Build_setoid ext'_refl ext'_sym ext'_trans).
```

Cet exemple montre que si on veut ajouter des enregistrements pour Coq, dont les signatures sont des termes du langage, il faut étendre la notion d'univers flottant aux signatures elles-mêmes.

6.4 MixDrecs

Pour implanter la notion d'espèce de FocKer, on introduit un nouveau type de données : les mixDrecs.

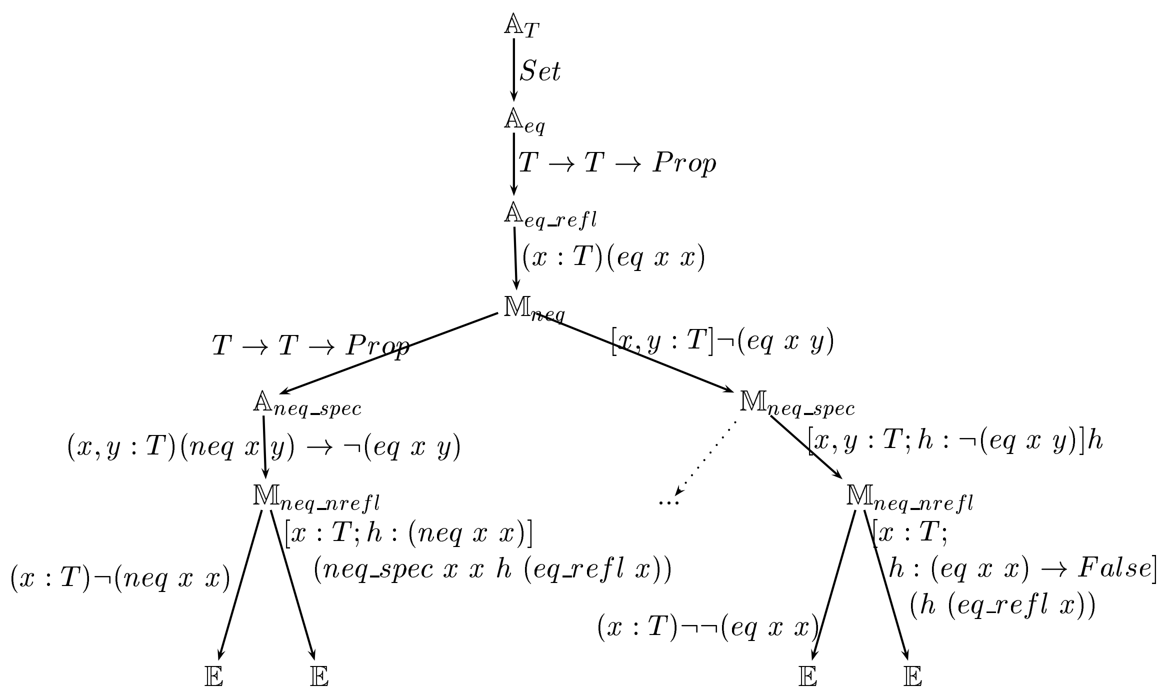
6.4.1 Description informelle

Les mixDrecs contiennent à la fois des champs définis et des champs déclarés. Il y a ainsi une correspondance entre les Drecords et les mixDrecs dont tous les champs sont définis. De façon duale, il y a une correspondance entre les signatures de Drecords et les mixDrecs dont tous les champs sont déclarés.

Comme pour les Drecords, on défini en fait un type `mixtel` plus général, et on définira les mixDrecs comme des mixtels vérifiant certaines propriétés (comme d'avoir une liste d'étiquettes sans doublon).

Contrairement aux télescopes qui ont une structure linéaire, les mixtels ont une structure arborescente avec 2 sortes de nœuds internes : les nœuds "abstraits" qui représentent grosso-modo des champs déclarés, et les nœuds "manifestes" qui représentent grosso-modo des champs définis. Les nœuds abstraits correspondent aux "nœuds internes" des signatures de télescopes : ils contiennent la même information, en particulier, ils n'ont qu'un seul fils. Chaque nœud manifeste contient plusieurs informations : le nom du champ qu'il représente, son type, son implémentation, et deux fils : un mixtel qui dépend de manière opaque de ce champ (comme dans les signatures de télescopes) et un mixtel qui dépend de manière transparente de ce champ (comme dans les implantations de télescopes).

Mixtel correspondant à l'exemple *setoid* de FocKer



$\mathbb{A}$: nœud abstrait ; $\mathbb{M}$: nœud manifeste ; $\mathbb{E}$: feuilles (mixtel vide).

Dans un nœud manifeste, les deux fils représentent le même mixtel : celui de gauche (opaque) le représente si on oublie (ou change) l'implémentation associée au nœud ; celui de droite (transparent) représente ce mixtel quand l'implémentation du nœud est préservée. Un *mixtel* bien formé est tel que le fils droit de chaque nœud manifeste est une vue plus définie du fils de gauche.

Un mixDrec est finalement un mixtel bien formé dont la liste d'étiquette est bien formée. Pour finir, précisons davantage les concepts de champs déclarés ou définis dans un mixDrec. Un champ d'un mixDrec sera dit défini si, sur la branche la plus à droite, le nœud qui porte son étiquette est manifeste. Au contraire, un champ sera dit déclaré si sur cette branche, le nœud qui porte son étiquette est abstrait. Dans ce cas, tous les nœuds qui portent son étiquette sont abstraits. Quand tous les champs d'un mixDrec sont définis, on peut facilement en extraire un Drecord : c'est la branche la plus à droite.

On peut maintenant donner une description plus mathématique des mixDrecs. Afin d'éviter une capture de structure, on doit d'abord décrire la structure arborescente des mixDrecs sur une liste d'étiquettes. C'est ce que fait le prédicat Pre. Ensuite $\{\!\{\}\!\}$ est un prédicat sur les signatures de Drecords. Pour construire des mixDrecs bien formés on utilise la relation $\gtrdot$ telle que $m_1 \gtrdot m_2$ signifie que m_1 est une vue plus définie de m_2.

$$Pre : \mathtt{`A}^* \to Type \qquad \frac{}{\emptyset_p : Pre_\emptyset} \qquad \frac{a \in \mathtt{`A} \quad p : Pre_l}{\mathbb{A}\ p : Pre_{a;l}} \qquad \frac{a \in \mathtt{`A} \quad p_1 : Pre_l \quad p_2 : Pre_l}{\mathbb{M}\ p1\ p2 : Pre_{a;l}}$$

$$Mix : (l : \mathtt{`A}^*) Sign_l \times Pre_l \to Type \qquad \frac{}{\{\!\{\}\!\} : Mix_{\{\},\emptyset_p}}$$

$$\frac{f : (x:T) Mix_{(F\ x),p}}{\{\!\{a : T\ ;\ f\}\!\} : Mix_{\{a:T\ ;\ F\},(\mathbb{A}\ p)}} \qquad \frac{f : (x:T) Mix_{(F\ x),p_1} \quad x : T \quad m : Mix_{(F\ x),p_2} \quad m \gtrdot (f\ x)}{\{\!\{a : T = x\ ;\ f;\ m\}\!\} : Mix_{\{a:T\ ;\ F\},(\mathbb{M}\ p_1\ p_2)}}$$

$$\frac{}{\{\!\{\}\!\} \gtrdot \{\!\{\}\!\}} \qquad \frac{(y:T)(f_1\ y) \gtrdot (f_2\ y)}{\{\!\{a : T\ ;\ f_1\}\!\} \gtrdot \{\!\{a : T;\ f_2\}\!\}} \qquad \frac{m \gtrdot (f_1\ x) \qquad (y:T)(f_1\ y) \gtrdot (f_2\ y)}{\{\!\{a : T = x\ ;\ f_1\ ;\ m\}\!\} \gtrdot \{\!\{a : T;\ f_2\}\!\}}$$

$$\frac{m_1 \gtrdot (f_1\ x) \quad m_2 \gtrdot (f_2\ x) \quad m_1 \gtrdot m_2 \quad (y:T)(f_1\ y) \gtrdot (f_2\ y)}{\{\!\{a : T = x\ ;\ f_1\ ;\ m_1\}\!\} \gtrdot \{\!\{a : T = x;\ f_2;\ m_2\}\!\}}$$

En fait, $\gtrdot$ ne sera pas implantée tout-à-fait comme suggéré ci-dessus, car cette présentation souffre d'une capture de structure.

Les opérations de l'héritage sur les espèces vont être ici codées à partir de deux opérations : la fusion de 2 mixDrecs de même signature, et le plongement d'un mixDrec par une sous-signature (de sa signature). On ne suit donc pas exactement l'approche suggérée dans FocKer qui permettait par exemple d'ajouter une définition sur un champ d'une espèce donnée. Il ne semble en effet pas très commode d'introduire une notion de "définition" : il faudrait pouvoir coder la notion de *contexte d'une définition*, sachant qu'un tel contexte peut contenir lui-même des définitions, à cause des dépendances transparentes. La notion de *contexte* de définition correspond finalement à la notion de mixDrec. Pour exprimer une "définition", on écrira donc un certain *mixDrec*.

Pour définir (ou redéfinir) un champ d'un mixDrec M de signature s. On commencera donc par exprimer cette définition par un mixDrec M', de signature s' telle que $s{:}\succ s'$. On *plongera* alors M' dans s ce qui donnera un mixDrec $\Uparrow_s M'$ dont les champs de s non présents dans M' auront été ajoutés comme non-définis. Enfin on fera la fusion de $\Uparrow_s M'$ avec M, ces deux mixDrecs étant de même signature.

On définit maintenant ces deux opérations : la fusion de mixDrecs de même signature, et le plongement d'un mixDrec dans une sous-signature.

6.4.1.a Fusion de mixDrecs de même signature

Le principe de cette opération est de faire la fusion nœud à nœud. On prend le paradigme suivant : on essaie d'avoir le mixDrec le plus défini possible, mais en cas de conflit (les deux nœuds sont manifestes) le nœud de gauche gagne toujours. Cela a pour conséquence que la fusion de 2 mixDrecs M_1 et M_2 est toujours une vue plus définie que celle de M_1 (mais pas celle de M_2). L'utilisateur sait ainsi facilement quelles définitions se retrouvent dans le mixDrec résultant. La définition de la fusion n'est cependant pas si simple qu'il n'y parait, car il faut préserver la bonne-formation des mixtels. Si M et M' sont deux mixDrecs de la forme : $M = \{\!\{a : T_1 = t_1; [a : T_1]\{\!\{b : T_2\}\!\}; \{\!\{b : T_2 = t_2\}\!\}\}\!\}$ et $M' = \{\!\{a : T_1 = t'_1; [a : T_1]\{\!\{b : T_2 = t'_2\}\!\}; \{\!\{b : T_2 = t'_2\}\!\}\}\!\}$. On attend que leur fusion, notée $M \oplus M'$ soit égale à M, car les deux champs de M sont déjà définis. Ainsi, sur les arbres, on a l'opération suivante :

$$
\begin{tikzcd}[column sep=small]
& \mathbb{M}\ t_1 \arrow[dl] \arrow[dr] & & & \mathbb{M}\ t'_1 \arrow[dl] \arrow[dr] & & & \mathbb{M}\ t_1 \arrow[dl] \arrow[dr] & \\
\mathbb{A} & & \mathbb{M}\ t_2 \arrow[r, phantom, "\oplus" above, pos=0.9] & \mathbb{M}\ t'_2 & & \mathbb{M}\ t'_2 \arrow[r, phantom, "=" above] & \mathbb{A} & & \mathbb{M}\ t_2
\end{tikzcd}
$$

En suivant bêtement le paradigme *"on essaie d'avoir le mixDrec le plus défini possible"*, on aurait obtenu le mixDrec ci-contre, qui n'aurait pas été bien formé.

$$
\begin{tikzcd}
& \mathbb{M}\ t_1 \arrow[dl] \arrow[dr] & \\
\mathbb{M}\ t'_2 & & \mathbb{M}\ t_2
\end{tikzcd}
$$

Cet exemple indique que l'on ne peut pas traiter les branches des mixDrecs indépendament, si on veut conserver la propriété d'être bien formé. Si M_1 et M_2 ont respectivement un Pre noté p_1 et p_2, on commence donc par définir *la liste de contrôle* de la fusion de p_1 et p_2 : c'est-à-dire une liste de booléens (notée $[\![p_1, p_2]\!]$) qui indique pour chaque étiquette si c'est la droite qui gagne. Pour créer cette liste, on compare les branches les plus à droites de chacun des mixDrec en éliminant les branches droites de M_2 dont les définitions seront effacées.

$p : Pre_l$	$p' : Pre_l$	$[\![p, p']\!] : (0\|1)^*$
$\emptyset_p$	$\emptyset_p$	$\emptyset$
$\mathbb{A}\ p_1$	$\mathbb{A}\ p_2$	$0; [\![p_1, p_2]\!]$
$\mathbb{A}\ p_1$	$\mathbb{M}\ p_{2_1}\ p_{2_2}$	$1; [\![p_1, p_{2_2}]\!]$
$\mathbb{M}\ p_{1_1}\ p_{1_2}$	$\mathbb{A}\ p_2$	$0; [\![p_{1_2}, p_2]\!]$
$\mathbb{M}\ p_{1_1}\ p_{1_2}$	$\mathbb{M}\ p_{2_1}\ p_{2_2}$	$0; [\![p_{1_2}, p_{2_1}]\!]$

Alors la fusion de M_1 et M_2 est $M_1 \oplus_{[\![p_1,p_2]\!]} M_2$, où $\oplus_l$ est défini figure 6.1 (page 127) pour l quelconque. La dernière règle de la table figure 6.1 permet seulement de faire de $\oplus$ une fonction totale sur les listes de booléens en ajoutant arbitrairement des "0". Mais, à l'exécution de $M \oplus_{[\![p_1,p_2]\!]} M'$, cette règle ne sera pas utilisée, car la liste de contrôle a la même longueur que M et M'.

En fait, pour définir proprement cette opération (en évitant une capture de structure) et pouvoir la typer, il faut commencer par la définir sur les "Pre" : il faut définir la fusion de p_1 et p_2 sur l, notée ici $p_1 \oplus_l^\dagger p_2$. La définition de cette opération sur les Pre suit en fait exactement les mêmes règles.

6.4.1.b Plongement d'un mixDrec dans une sous-signature

On définit figure 6.2 (page 128) l'opération $\Uparrow_s M$ qui plonge un mixDrec M dans une sous-signature s de sa signature s'. Cette opération est définie par induction sur une preuve que $s{:}{\succ}s'$. L'idée est d'ajouter les champs de s qui ne sont pas dans s' comme des champs abstraits. En fait, lors des permutations, on est aussi amené à effacer des définitions, car les permutations brisent

$m : Mix_{s,p}$	$m' : Mix_{s,p'}$	$l : (0\|1)^*$	$m \oplus_l m' : Mix_{s,p\oplus_l^\dagger p'}$
$\{\!\{\}\!\}$	$\{\!\{\}\!\}$	l	$\{\!\{\}\!\}$
$\{\!\{a : T\ ;\ f_1\}\!\}$	$\{\!\{a : T\ ;\ f_2\}\!\}$	$b;l$	$\{\!\{a : T\ ;$ $[x : T]((f_1\ x) \oplus_l (f_2\ x))\}\!\}$
$\{\!\{a : T\ ;\ f_1\}\!\}$	$\{\!\{a : T = x\ ;\ \ f_2;\ m\}\!\}$	$0;l$	$\{\!\{a : T\ ;$ $[x : T]((f_1\ x) \oplus_l (f_2\ x))\}\!\}$
$\{\!\{a : T\ ;\ f_1\}\!\}$	$\{\!\{a : T = x\ ;\ \ f_2;\ m\}\!\}$	$1;l$	$\{\!\{a : T = x\ ;$ $[y : T]((f_1\ y) \oplus_l (f_2\ y))\ ;$ $(f_1\ x) \oplus_l m\}\!\}$
$\{\!\{a : T = x\ ;\ f_1;\ m\}\!\}$	$\{\!\{a : T\ ;\ f_2\}\!\}$	$b;l$	$\{\!\{a : T = x\ ;$ $[y : T]((f_1\ y) \oplus_l (f_2\ y));$ $m \oplus_l (f_2\ x)\}\!\}$
$\{\!\{a : T = x\ ;\ f_1;\ m_1\}\!\}$	$\{\!\{a : T = y;\ f_2; m_2\}\!\}$	$b;l$	$\{\!\{a : T = x\ ;$ $[z : T]((f_1\ z) \oplus_l (f_2\ z));$ $m_1 \oplus_l (f_2\ x)\}\!\}$
m	m'	$\emptyset$	$m \oplus_0 m'$

Fig. 6.1 – Calcul de la fusion de mixDrecs

éventuellement les dépendances. Ainsi pour le cas _swap, lorsque le deuxième champ de M est défini, on oubli systématiquement cette définition, car celle-ci peut éventuellement dépendre du premier champ.

Comme pour la fusion, il faudrait commencer par définir cette opération au niveau des Pre en suivant exactement les cas de la figure 6.2. Si $s : Sign_l$ et $p : Pre_l$, on a noté abusivement cette opération par $\Uparrow_s p$.

6.4.2 Implantation

Comme expliqué précédemment, les mixDrecs sont en fait implantés via les mixtels. Le type `mixtel` est défini figure 6.3 (page 129) en 2 parties. Premièrement, la structure d'arbre est exprimée seulement au niveau des listes d'étiquettes, via le prédicat `pre`. Deuxièment, la valeur associée aux nœuds est exprimée dans `mixtel`. Comme pour les télescopes, `pre` et `mixtel` sont construits comme des types récursifs au lieu d'être construits comme des types inductifs. Ainsi, on définit respectivement `flag` et `mix` qui contiennent la "partie inductive" de ces définitions.

6.4.3 Propriétés et opérations

Le prédicat `is_all_def` teste si toutes les étiquettes dans un `pre` sont définies. La fonction `new` engendre un impltel d'un mixtel dont toutes les étiquettes sont définies.

```
Definition new:(l:(list A); s:(sigtel l); p:(pre l))(is_all_def p)
  ->(mixtel s p)->(impltel s).
```

$s : Sign_l$	$s' : Sign_{l'}$	$s{:}\succ s'$	$M : Mix_{s',p}$	$\Uparrow_s M : Mix_{s,\Uparrow_s p}$
s	$\{\}$	_E	$\{\!\!\{\}\!\!\}$	$\widetilde{s}$
$\{a : T\ ;\ F_1\}$	$\{a : T\ ;\ F_2\}$	_C	$\{\!\!\{a : T\ ;\ f\}\!\!\}$	$\{\!\!\{a : T\ ;$ $[x : T] \Uparrow_{(F\ x)} (f\ x)\}\!\!\}$
idem	idem	idem	$\{\!\!\{a : T = x\ ;\ f;\ m\}\!\!\}$	$\{\!\!\{a : T = x\ ;$ $[y : T] \Uparrow_{(F\ y)} (f\ x);$ $\Uparrow_{(F\ x)} m\}\!\!\}$
$\{a : T\ ;\ F\}$	s	_lift	m	$\{\!\!\{a : T\ ;$ $[x : T] \Uparrow_{(F\ x)} m\}\!\!\}$
s_1	s_3	_trans	m	$\Uparrow_{s_1} (\Uparrow_{s_2} m)$
$\{a : T_1\ ;$ $[x : T_1]$ $\{b : T_2;$ $(F_1\ x)\}$ $\}$	$\{b : T_2\ ;$ $[y : T_2]$ $\{a : T_1;$ $(F_2\ y)\}$ $\}$	_swap	$\{\!\!\{b : T_2\ ;$ $[y : T_2]\{\!\!\{a : T_1\ ; (f\ y)\}\!\!\}\}\!\!\}$	$\{\!\!\{a : T_1\ ;$ $[x : T_1]$ $\{\!\!\{b : T_2\ ;$ $[y : T_2]$ $\Uparrow_{(F_1\ x\ y)} (f\ y\ x)\}\!\!\}\}\!\!\}$
idem	idem	idem	$\{\!\!\{b : T_2\ ;$ $[y : T_2]\{\!\!\{a : T_1 = x\ ;\ (f\ y);\ \_\}\!\!\}\}\!\!\}$	idem
idem	idem	idem	$\{\!\!\{b : T_2 = y\ ;$ $[z : T_2]\{\!\!\{a : T_1\ ;\ (f\ z)\}\!\!\};$ $\{\!\!\{a : T_1;\ g\}\!\!\}\}\!\!\}$	$\{\!\!\{a : T_1\ ;$ $[x : T_1]$ $\{\!\!\{b : T_2 = y\ ;$ $[z : T_2]$ $\Uparrow_{(F_1\ x\ z)} (f\ z\ x);$ $\Uparrow_{(F_1\ x\ y)} (g\ x)\}\!\!\}\}\!\!\}$
idem	idem	idem	$\{\!\!\{b : T_2 = y\ ;$ $[z : T_2]\{\!\!\{a : T_1\ ;\ (f\ z)\}\!\!\};$ $\{\!\!\{a : T_1 = \_\ ;\ g\ ;\ \_\}\!\!\}\}\!\!\}$	idem
idem	idem	idem	$\{\!\!\{b : T_2 = y\ ;$ $[z : T_2]\{\!\!\{a : T_1 = \_\ ;\ (f\ z);\ \_\}\!\!\};$ $\{\!\!\{a : T_1 = \_\ ;\ g\ ;\ \_\}\!\!\}\}\!\!\}$	idem

Légende : Dans cette table le mot "idem" signifie que la valeur de la case est la même que celle de la case "juste au-dessus" (dans la même colonne). Le symbole "_" est le filtre universel (variable qui n'est pas utilisée dans la suite).

FIG. 6.2 – Calcul du plongement d'un mixDrec dans une sous-signature

L'implémentation de $\gg$ est fait sur les mixtels via `match_mix`.

Définition 6.5 *Etant donné une signature* `s`*, et 2 mixtels* `m1` *et* `m2`*,* `(match_mix m1 m2)` *signifie que* m_1 *est une vue plus définie que* m_2 *(*`match_pre` *est la notion correspondante au niveau de* `pre`*).*

```
l:(list A)                                 Definition match_mix:
s:(sigtel l)                                 (match_pre p1 p2)
p1,p2:(pre l)                                  ->(mixtel s p1)->(mixtel s p2)->Type.
```

`match_mix` est une relation transitive, mais ni réflexive (ni anti-reflexive).

Théorème 6.5 *Sous les hypothèses précédentes, si toutes les étiquette de* `m2` *sont définies, et que toutes les étiquettes de* `m1` *le sont (propriété* `is_all_def_match`*), alors l'implantation de télescope*

```
Inductive UnitS:Set:= unitS:UnitS.

Inductive flag[U:Set]:Set :=
   Abs: U->(flag U)
 | Man: U->U->(flag U).

Fixpoint pre[l:(list A)]: Set :=
  Cases l of
       nil => UnitS
  | (cons a m) => (flag (pre m))
  end.

Definition abs_of:=[U:Set;x:(flag U)]
  Cases x of (Abs p) => p
           | (Man p _) => p
  end.
```

```
Variable T:Type.
Variable U:Set.
Variable F:T->U->Type.

Structure dpair[f:T->Type]:Type :=
  {dproj1:T;dproj2:(f dproj1)}.

Definition option:(flag U)->Type:=
 [p]Cases p  of
   (Abs _)=>UnitT
 | (Man _ p2)=>(dpair [x:T](F x p2))
 end.

Structure mix[p:(flag U);a:A]:Type:=
 {mix_spec:(x:T)(F x (abs_of p));
  mix_val:(option p) }.
```

```
Fixpoint mixtel[l:(list A)]:(sigtel l)->(pre l)->Type :=
  <[l:(list A)](sigtel l)->(pre l)->Type>Cases l of
        nil => [_;_]EsigT
    | (cons a m) => [s;p](mix [x:(dom s)](mixtel (s x)) p a)
  end.
```

FIG. 6.3 – Définition Coq des mixtels

engendrée (via `new`*) par* `m2` *est égale à celle engendrée par* `m1`*.*

Preuve En Coq, on a prouvé les propriétés suivantes :

```
m1:(mixtel s p1)
m2:(mixtel s p2)
```

```
Theorem match_mix_new:(H1:(is_all_def p2);
  H2:(match_pre p1 p2))(match_mix H2 m1 m2)->
    (new H1 m2)===(new (is_all_def_match H1 H2) m1).
```

□

Définition 6.6 *Un mixtel bien formé est un mixtel qui vérifie la réflexivité de* `match_mix`*.*

```
Definition wellmixed:=
  [p:(pre l);H:(wellpre p);m:(mixtel s p)](match_mix H m m).
```

On définit alors les deux opérations suivantes sur les mixtels, en suivant la description informelle précédentes : `merge_mix` pour la fusion de 2 mixtels de même signatures et `lift_mix` pour plonger un mixtel dans une sous-signature.

```
Definition merge_mix:(l:(list A); s:(sigtel l); p1,p2:(pre l))
    (mixtel s p1)->(mixtel s p2)->(mixtel s (merge_pre p1 p2)).

Definition lift_mix:(l1,l2:(list A); H:(l1 ⊃ l2);
 s1:(sigtel l1); s2:(sigtel l2))(subsig H s1 s2)->(p:(pre l2))
      (mixtel s2 p)->(mixtel s1 (lift_pre H p)).
```

Les propriétés suivantes sont toujours en cours de démonstration : il y a quelques difficultés à manipuler les gros termes qu'elles engendrent. La propriété `wellmix_merge` assure que `merge_mix` transforme les mixtels bien formés en mixtels bien formés. Et `match_mix_merge_1` assure que `merge_mix m1 m2` est une vue plus définie de `m1`.

```
Lemma wellmix_merge: (wellmixed H1 m1) -> (wellmixed H2 m2)->
    (wellmixed (wellpre_merge H1 H2) (merge_mix m1 m2)).

Lemma match_mix_merge_1: (wellmixed H1 m1) -> (wellmixed H2 m2)->
    (match_mix (match_pre_merge_1 H1 H2) (merge_mix m1 m2)  m1).
```

C'est le point où s'arrête l'implantation. Il faudrait en fait écrire quelques lemmes de plus pour vérifier la correction de `lift_mix`.

6.4.4 Utilisation des mixtels

La correction des opérations `merge_mix` et `lift_mix` n'est pas complètement prouvée. Mais on peut déjà commencer à utiliser ces opérations. Nous avons testé les exemples présentés ici. Nous ne donnons pas les sources, car elles sont tout simplement illisibles (il n'y a pas de syntaxe concrète pour les mixtels). Pour expliquer ces exemples, nous utilisons la syntaxe virtuelle de FocKer.

D'abord, on a codé le mixtel *setoid* mentionné ci-dessus. Ensuite, on a défini *Bool_base* comme suit, où *bool* est un type énuméré à 2 constructeurs, *negb* la négation sur les booléens, et *Is_true* la conversion des booléens vers `Prop` :

$$Bool\_base := \left\{ \begin{array}{ll} \texttt{def} & T : \texttt{Type} \\ & := bool \\ \texttt{def} & eq : [\![T]\!] \rightarrow [\![T]\!] \rightarrow \texttt{Prop} \\ & := \lambda[b_1, b_2 : bool](Is\_true\ (if\ b_1\ then\ b_2\ else\ (negb\ b_2))) \\ \texttt{def} & eq\_refl : \forall(x : [\![T]\!])([\![eq]\!]\ x\ x) \\ & := \lambda[b : bool]if\ b\ then\ I\ else\ I. \\ \texttt{def} & neq : [\![T]\!] \rightarrow [\![T]\!] \rightarrow \texttt{Prop} \\ & := \lambda[b_1, b_2 : bool](Is\_true\ (if\ (negb\ b_1)\ then\ (negb\ b_2)\ else\ b_2)) \\ \texttt{def} & neq\_spec : \forall(x, y : [\![T]\!])([\![neq]\!]\ x\ y) \rightarrow \neg([\![eq]\!]\ x\ y) \\ & := \lambda[b1, b2 : bool]if\ b1\ \ then\ if\ b2\ then\ \lambda[H, \_]H\ else\ \lambda[\_, H]H \\ & \qquad\qquad\qquad\qquad\quad else\ if\ b2\ then\ \lambda[\_, H]H\ else\ \lambda[H, \_]H \end{array} \right\}$$

Le but de l'exemple est de faire la fusion de *setoid* et *Bool_base*. Comme *neq_nrefl* est un champ de *setoid*, mais pas de *Bool_base*, on doit d'abord plonger *Bool_base* dans la signature de *setoid*. Cela construit un mixtel qui enrichit *Bool_base* avec le champ *neq_nrefl*. Ensuite, on peut faire la fusion. Mais la fusion n'est pas commutative, car *neq* n'a pas la même implantation dans les deux mixtels. On a essayé les deux possibilités. Les résultats ont été calculés avec une efficacité raisonnable.

6.4.5 Des mixtels aux mixDrecs

Comme l'a déjà dit, les mixDrecs sont des mixtels bien formés sur une signature de Drecords. La bonne formation peut être déchargée via une tactique qui échoue lorsque le mixtel n'est pas bien formé. Ainsi on pourrait imaginer utiliser la syntaxe concréte précédemment définie. Pour l'instant, il n'y a pas de syntaxe concrète, et l'utilisateur doit décrire toute la structure arborescente décrite ci-dessus.

6.5 Conclusion

L'implantation en Coq n'est donc pas totalement achevée, mais elle est suffisament avancée pour qu'on puisse déjà en tirer certaines conclusions. Parallèlement à ce travail, Pollack a proposé un codage des enregistrements en théorie des types, qu'il a implanté en Lego. Il est donc intéressant de comparer notre approche à celle de Pollack.

6.5.1 Comparaison avec les records de Pollack

Les records donnés par Pollack dans [Pol00] sont définis par les règles d'introduction suivantes[6] :

$$Sign : {}^{\mathtt{e}}\mathtt{A}^* \rightarrow Type$$

$$\frac{}{\{\} : Sign_{\emptyset}}\ \textbf{EForm} \qquad \frac{A : Sign_l \quad a \notin l \quad U : A \rightarrow Type}{\{A\ ;\ a : U\} : Sign_{l;a}}\ \textbf{CForm}$$

$$\frac{}{<> : \{\}}\ \textbf{EIntro} \qquad \frac{\{A\ ;\ a : U\} : Sign_{l;a} \quad X : A \quad x : (U\ X)}{<X\ ;\ a = x> : \{A\ ;\ a : U\}}\ \textbf{CIntro}$$

On appelle *signature d'enregistrement* les termes entre {.}, et *enregistrement* les termes entre <.>. Ce système d'enregistrements est dit associatif à gauche, car on étend ces enregistrements "en rajoutant des champs à droite" (règle `CForm`). Et les dépendances entre champs sont exprimées par une abstraction (le type `U` de la règle `CForm`). Les champs de droite dépendent donc des champs de gauche.

Remarquons cependant cette définition des enregistrements associatifs à gauche ne s'exprime pas directement en Coq : dans la règle `CForm` il y a une sorte d'induction mutuelle entre la notion de signature et la notion d'enregistrement : le terme U est une fonction qui prend en argument un enregistrement. De plus, ce terme U introduit une occurrence négative du point de vue des types inductifs (cf. 3.2.1.d). Pour coder ses enregistrements en Coq, Pollack utilise une astuce qui consiste à passer le type enregistrement en argument de <.>. Si `UnitT` et `dpair` représentent respectivement le type à un élément, et le type des paires dépendantes, les signatures se définissent de la manière suivantes :

$$Sign : Type \times {}^{\mathtt{e}}\mathtt{A}^* \rightarrow Type$$

$$\frac{}{\{\} : Sign_{\mathtt{UnitT},\emptyset}}\ \textbf{EForm'} \qquad \frac{AT : Type \quad AS : Sign_{AT,l} \quad a \notin l \quad U : AT \rightarrow Type}{\{AS\ ;\ a : U\} : Sign_{(\mathtt{dpair}\ U),(l;a)}}\ \textbf{CForm'}$$

Alors, pour toute liste d'étiquettes l, pour tout type T, et pour toute signature s de type $Sign_{T,l}$, un enregistrement est un terme de type T. Cela signifie que les constructeur des enregistrements sont `IT`, constructeur de `UnitT`, et `Build_dpair`, constructeur de `dpair`. Si on ne se préoccupe pas ici des étiquettes (ce qui n'est pas important ici, car il n'y a pas de capture de structure) cela conduit aux définitions Coq suivantes :

[6] En réalité, Pollack a donné une version simplifiée de ces règles. Mais nous n'avons pas souhaité garder cette version pour les raisons invoquées en section 8.1.1.a

```
Inductive sign: Type -> Type :=
  | EForm': (sign UnitT)
  | CForm': (AT:Type)(sign AT)->(U:AT -> Type)(sign (dpair U)).

Definition Record:= [T:Type; s:(sign T)]T.
```

Les constructeurs des records sont alors :

```
Definition EIntro: (Record EForm') := IT.

Definition CIntro: (T:Type; s:(sign T); U: (Record s)->Type)
       (X:(Record s))(U X)->(Record (CForm' s U))
           := [T;s;U;X;x](Build_dpair x).
```

Le système associatif associatif à gauche est plus naturel que celui associatif à droite, car l'opération d'extension des signatures (opération *Flat* des Drecords) est une règle primitive. Il semble aussi plus efficace, car on peut ainsi avoir plus de partage entre une signature et ses sur-signatures. De plus, Pollack propose une extension simple de ce système (voir le chapitre 10 et [Pol00]) afin d'avoir une notion de signatures avec champs manifestes. Cependant, les signatures avec champs manifestes de Pollack ne correspondent pas aux mixDrecs. D'une part, les mixDrecs contiennent une notion de dépendance opaque entre champs que les signatures avec champs manifestes ne contiennent pas. D'autre part, certaines signatures avec champs manifestes ne sont pas directement exprimables dans les mixDrecs. En fait, sur ces signatures, la redéfinition des champs n'est pas toujours possible (elle n'a pas toujours un sens). Cette discussion est détaillée dans la section 10.4. Et finalement, le codage des mixDrecs ne semble pas évident à transposer pour des signatures associatives à gauche (mais inversement, il n'est pas simple d'étendre le système associatif à droite pour avoir des signatures manifestes).

6.5.2 Intérêts et limites de ce codage

Ce codage a permis de dégager plus précisement quelles étaient les propriétés et les opérations exigées dans FocKer. Le fait que l'on puisse étudier les propriétés de ce système en Coq a permis d'approfondir notre intuition sur ses objets, et de donner confiance dans cette intuition. Coq garantit en effet que cette construction a un sens : aucun axiome n'a été utilisé dans ce codage. En particulier les axiomes parfois nécessaires lorsqu'on manipule l'égalité avec des types dépendants ont pu être évités. Cela montre aussi que notre système n'étend pas l'expressivité théorique du calcul des constructions inductives, puisqu'il peut se coder dedans.

Le principal défaut de ce système est donc son caractère inutilisable en pratique. Mais cette lourdeur d'utilisation était prévisible : on demande ici au système de faire beaucoup de calculs sur les types, ce qui n'est en général pas le cas dans une utilisation courante de Coq. En général, on n'utilise pas vraiment en Coq des programmes écrits en Coq : on les extrait vers OCaml, puis on les compile avant de les exécuter.

Néanmoins, comme on sait maintenant que FocKer peut être exprimé en Coq, il est raisonnable de tenter un codage plus superficiel. Ainsi, dans [Pre00], Virgile Prevosto a traduit la syntaxe concrète FOC vers du Coq en effectuant une bonne part des calculs lors de la traduction vers Coq, c'est-à-dire en dehors de Coq. Cela lui a permis d'écrire une partie conséquente de la librairie.

Il reste que cette formalisation de FocKer est complètement dépendante de Coq, et qu'elle ne semble pas suffisament abstraite pour donner un cadre théorique à un codage comme celui de [Pre00]. On va donc s'appuyer les intuitions apportées par cette expérience pour donner une axiomatisation abstraite de FocKer.

Troisième partie

Axiomatisation catégorique du système

Chapitre 7

Introduction

7.1 Présentation générale

La description en Coq précédente n'est pas complètement satisfaisante. En particulier, cette description est très dépendante de Coq. Si on peut se convaincre qu'elle correspond à peu près à FocKer, elle ne permet pas de dire si une autre description correspond à FocKer. On aimerait donc maintenant une description plus générale, plus abstraite, en définitive plus axiomatique. Cette description permettrait de donner de manière plus claire les propriétés méta-théoriques de FocKer.

On propose ici une telle axiomatisation, en théorie des catégories. La première étape de l'axiomatisation, chapitre 8, est la notion de *catégorie de signatures d'enregistrements*. Cette notion, très fortement inspirée des catégories contextuelles de Cartmell[Car86] a pour but d'axiomatiser la notion d'enregistrements avec champs dépendants. On introduit ensuite, chapitre 9, la notion de *catégorie de générateurs d'enregistrements* qui ajoute des définitions de champs à l'intérieur des signatures. Comme leur nom l'indique, les générateurs d'enregistrements correspondent en fait aux classes dans les langages à objets : à chaque générateur, on associe une fonction qui prend en argument un enregistrement, dont les champs correspondent aux champs non définis du générateur, et qui étend cet enregistrement en lui ajoutant les champs définis dans le générateur.

Bien que les générateurs d'enregistrements soient des signatures avec des champs qui comportent éventuellement des définitions, ils ne correspondent pas forcément aux signatures avec champs manifestes au sens usuel (par exemple [Pol00] ou [Cou98]). On montre néanmoins que les signatures avec champs manifestes de Pollack peuvent être exprimées par la notion de *catégorie de signatures avec champs manifestes*, présentée au chapitre 10. Cette dernière notion de catégorie est en fait un cas particulier de catégories de générateurs d'enregistrements.

Finalement, au chapitre 11, on étend la notion de catégorie de générateurs d'enregistrements en *catégorie de générateurs d'enregistrements avec effacement de définition*. Dans cette dernière notion de catégorie, il est ainsi possible d'effacer une définition : on peut alors simuler la redéfinition, comme un effacement de définition suivi d'une définition. Bien sûr, comme dans FocKer, l'effacement d'une définition conduit aussi à effacer toutes les définitions qui en dépendaient de manière transparente.

Dans chacune de ces notions catégoriques, on va introduire des opérations qui servent à décrire les opérations de FocKer. De plus, on essayera de construire des modèles de ces notions catégoriques afin de montrer la cohérence de leurs axiomes. Toutefois, on n'a pas réussi à donner un modèle complet des CGED. Ce point sera évoqué en section 11.6.

Les notations utilisées dans cette partie peuvent parfois paraître très lourdes, et nous nous en excusons auprès du lecteur. Nous avons tenté de les alléger au maximum sans pour autant nuire à la précision nécessaire dans les preuves. En effet, dans ces constructions catégoriques, l'équilibre entre les "abus de notation" qui facilitent la lecture, et la précision des notations qui facilite la compréhension en ne laissant pas trop d'implicites, est parfois dure à trouver. Le lecteur trouvera en début d'Index les numéros des pages qui introduisent les principales notations des chapitres de cette partie.

7.2 Description par propriétés des systèmes de types dépendants

Pour motiver l'axiomatisation catégorique, on introduit la notion de *D-système*. Cette notion est elle-même une axiomatisation des systèmes de types dépendants. C'est une vision très déclarative des systèmes de types dépendants : on ne donne pas de syntaxe pour les termes (et pour les variables en particulier). Cette notion est très rudimentaire puisqu'on ne suppose pas qu'elle contient un λ-calcul typé. Elle permet juste d'exprimer la notion de contexte dans un système de types dépendants.

Cette axiomatisation est fortement inspirée par la notion de *"théorie algébrique généralisée"* de Cartmell (cf. [Car86]). L'originalité de l'approche est ici de ne pas avoir de noms de variables et d'utiliser un opérateur (noté $[\![.]\!]$) pour appliquer les substitutions. On est ainsi très proche d'une description catégorique.

7.2.1 Notations sur les suites

Soit E un ensemble, on note E^* l'ensemble des suites finies sur E. Le symbole $\emptyset$ désignera la suite vide. Si s est une suite, et x un élément de E, on notera $s\,;x$ la suite qui étend s en lui ajoutant x comme dernier élément. On notera aussi $x\,;s$ la suite qui contient x comme premier élément et les élements de s comme reste. Enfin, si s et s' sont deux suites, $s\,;s'$ désignera leur concaténation.

Si s est une suite qui contient au moins k éléments, on notera $s(k)$ ce k-ième élément. L'entier $|s|$ désignera la longueur de s.

Si n et m sont deux entiers, $[n, m]$ dénote l'intervalle des entiers compris entre n et m (inclus).

7.2.2 Définition des D-systèmes

On appelle *D-système* un uplet :

$$< \mathcal{L}_{Type}, \mathcal{L}_{term}, . \equiv_{Type} ., . \equiv_{term} ., . \vdash . = ., . \vdash . = . : ., [\![\cdot]\!]_{Type}(\cdot), [\![\cdot]\!]_{term}(\cdot), id, \circ >$$

qui vérifie :

- "$\mathcal{L}_{Type}$" est un ensemble appelé *langage de pré-types*. Les suites finies de $\mathcal{L}_{Type}$ s'appellent les *contextes*. L'ensemble des contextes s'écrit donc $\mathcal{L}^*_{Type}$. Remarquons que les contextes ne contiennent pas de noms de variables, mais, intuitivement, on peut considérer que les variables sont désignées par leur position dans le contexte, dans une approche à la De Bruijn.
- "$\mathcal{L}_{term}$" est un ensemble appelé *le langage de pré-termes*. Les suites finies de $\mathcal{L}_{term}$ s'appellent les *substitutions*. L'ensemble des substitutions s'écrit donc $\mathcal{L}^*_{term}$.
- "$. \equiv_{Type} .$" est une relation binaire sur $\mathcal{L}_{Type}$ appelée *égalité syntaxique sur les types*.
- "$. \equiv_{term} .$" est une relation binaire sur $\mathcal{L}_{term}$ appelée *égalité syntaxique sur les termes*.
- "$. \vdash . = .$" est une relation de $\mathcal{L}^*_{Type} \times \mathcal{L}_{Type} \times \mathcal{L}_{Type}$ qui s'appelle *jugement d'égalité sémantique sur les types*
- "$. \vdash . = . : .$" est une relation de $\mathcal{L}^*_{Type} \times \mathcal{L}_{term} \times \mathcal{L}_{term} \times \mathcal{L}_{Type}$ qui s'appelle *jugement d'égalité sémantique sur les termes*

- "$[\![.]\!]_{Type}(.)$" est une fonction de $\mathcal{L}_{Type} \times \mathcal{L}^*_{term}$ dans $\mathcal{L}_{Type}$, appelée *application des substitutions sur les types*
- "$[\![.]\!]_{term}(.)$" est une fonction de $\mathcal{L}_{term} \times \mathcal{L}^*_{term}$ dans $\mathcal{L}_{term}$, appelée *application des substitutions sur les termes*
- "$id(.)$" est une fonction de $\mathcal{L}^*_{Type}$ dans $\mathcal{L}^*_{term}$, appelée *substitution identité.*
- "$. \circ .$" est une fonction de $\mathcal{L}^*_{term} \times \mathcal{L}^*_{term}$ dans $\mathcal{L}^*_{term}$, appelée *composition des substitutions*
- Un D-système vérifie en outre les 20 axiomes ci-dessous.

Pour faciliter la lecture de ces 20 axiomes, on donne quelques notations et définitions. Ainsi, les contextes seront dénotés Γ, Δ, Σ, etc... Les types (ou pré-types) seront dénotés T_1, T_2, ... ou T, T', T'', ... Les termes (ou pré-termes) seront dénotés t, t', t'', ... Les substitutions seront dénotées par s_1, s_2, ... ou s, s', s'', ... On ne distinguera pas pré-types et types, ou pré-termes et termes.

Lorsque cela ne crée pas d'ambiguïté, on notera les égalités syntaxiques sur les types et les termes par le même symbole $\equiv$. On étend de plus cette notion d'égalité syntaxique aux contextes, et aux substitutions. De même, on notera les applications de substitution sur les types et les termes par $[\![]\!](.)$.

Axiome 1 (Egalités syntaxiques) On suppose que les égalités syntaxiques sont des relations d'équivalence, et que toutes les fonctions et relations précédemment introduites sont compatibles avec ces égalités syntaxiques.

Définition 7.1 *A partir des jugements d'égalité sémantique, on définit 2 nouveaux jugements.*
- *Le* jugement de bonne formation des contextes *est un prédicat unaire sur les contextes, noté* $. \vdash$*. Il est défini par cas sur la longueur de* Γ *:*
$$\emptyset \vdash \qquad \text{et} \qquad \Gamma\,;T \vdash \ \text{ équivaut à } \Gamma \vdash T = T$$
- *Le* jugement de typage d'un terme *est une relation de* $\mathcal{L}^*_{Type} \times \mathcal{L}_{term} \times \mathcal{L}_{Type}$*, notée* $. \vdash . : .$*, et telle que* $\Gamma \vdash t : T$ *équivaut à* $\Gamma \vdash t = t : T$*.*

On exprime ci-dessous les axiomes vérifiés par les égalités sémantiques. Remarquons que ces égalités sémantiques ne sont pas forcément réflexives dans la mesure où elles peuvent s'appliquer à des pré-types (resp. des pré-termes), c'est-à-dire pas forcément bien formés (resp. bien typés).

Axiomes (Jugements d'égalité) Les jugements d'égalité sémantique vérifient les axiomes suivants :

2) $\Gamma \vdash T = T'$ implique $\Gamma\,; T \vdash$ et $\Gamma\,; T' \vdash$.

3) $\Gamma\,; T\,; T' \vdash$ implique $\Gamma\,; T \vdash$.

4) $\Gamma \vdash t = t' : T$ implique $\Gamma\,; T \vdash$, $\Gamma \vdash t : T$ et $\Gamma \vdash t' : T$.

5) À Γ fixé, la relation $(T, T') \mapsto \Gamma \vdash T = T'$ est une relation symétrique et transitive.

6) De même, à Γ et T fixé, la relation $(t, t') \mapsto \Gamma \vdash t = t' : T$ est une relation symétrique et transitive.

7) Si $\Gamma \vdash P$ et $\Gamma \vdash T''$ alors $\Gamma\,; T'' \vdash P$ (où P est $T = T'$ où $t = t' : T$).

8) Si $\Gamma\,; T_1\,; \Delta \vdash P$ et $\Gamma \vdash T_1 = T_2$ alors $\Gamma\,; T_2\,; \Delta \vdash P$ (où P est $T = T'$ ou $t = t' : T$).

Intuitivement, si T (resp. t) est un pré-type (resp. un pré-terme) dans un calcul avec des variables à la de Bruijn, $[\![T]\!]$ (resp. $[\![t]\!]$) interprète T (resp. t) comme une fonction du méta-langage prenant en argument une suite finie de pré-termes $t_1 \ldots t_n$, et substituant respectivement ces n termes sur les variables libres de numéros (avant relocation) 1 à n de T (resp. de t). Remarquons que si la variable numéro k ne figure pas dans T (resp. t), alors le type (resp. terme) retourné par $[\![T]\!]$ (resp. $[\![t]\!]$) est indépendant du k-ième élément de la liste qu'on lui passe en argument.

En particulier, si Γ est un contexte, tel que $\Gamma\,;T \vdash$ (resp. $\Gamma \vdash t : T$), alors toutes les variables libres de T (resp. t) sont dans Γ. Donc seuls les $|\Gamma|$ premiers termes des suites en arguments de $[\![T]\!]$ (resp. $[\![t]\!]$) sont utilisés. C'est ce qu'expriment les deux axiomes suivants.

Axiomes (Applications des substitutions) Les applications des substitutions sont indépendantes de la notion de contexte. Néanmoins, on sait que un pré-type bien formé dans un contexte Γ (resp. un pré-terme bien typé dans un contexte) *est clos dans* Γ vis-à-vis de l'application des substitutions. Autrement dit, lorsqu'on applique une substitution s sur ce type (resp. ce terme), seuls les $|\Gamma|$ premiers termes de s ont un rôle.

9) Pour tout Γ tel que $\Gamma\,;T \vdash$ et pour toute substitution s de longueur n avec $n \geq |\Gamma|$

$$[\![T]\!](s(1)\dots s(n)) \equiv [\![T]\!](s(1)\dots s(|\Gamma|))$$

10) Pour tout Γ et tout T tel que $\Gamma \vdash t : T$ et pour toute substitution s de longueur n avec $n \geq |\Gamma|$, alors

$$[\![t]\!](s(1)\dots s(n)) \equiv [\![t]\!](s(1)\dots s(|\Gamma|))$$

Définition 7.2 (Typage des substitutions) *On définit le jugement de typage sur les substitutions, noté "$. : . \to .$" comme une relation de $\mathcal{L}^*_{Type} \times \mathcal{L}^*_{term} \times \mathcal{L}^*_{Type}$. On pose $s : \Gamma \to \Delta$ ssi : $\Gamma \vdash$ et $\Delta \vdash$ et $|s| = |\Delta|$ et pour tout k de $[1, |\Gamma|]$,*

$$\Gamma \vdash s(k) : [\![\Delta(k)]\!](s(1)\dots s(k-1))$$

Si s est une substitution telle que $s : \Gamma \to \Delta$, on dira que s est *une substitution de* $\Gamma \to \Delta$.

Définition 7.3 (Jugement d'égalité sur les substitutions) *On définit maintenant le jugement d'égalité sémantique des substitutions, noté $. = . : . \to .$ comme une relation de $\mathcal{L}^*_{term} \times \mathcal{L}^*_{term} \times \mathcal{L}^*_{Type} \times \mathcal{L}^*_{Type}$. On pose que $s_1 = s_2 : \Gamma \to \Delta$ ssi s_1 et s_2 sont des substitutions de $\Gamma \to \Delta$ et pour tout k de $[1, |\Delta|]$: $\Gamma \vdash s_1(k) = s_2(k) : [\![\Delta(k)]\!](s_1(1)\dots s_1(k-1))$.*

Remarque 7.1 *A Γ et Δ fixés, la relation $(s_1, s_2) \mapsto s_1 = s_2 : \Gamma \to \Delta$ est symétrique et transitive. De plus, pour toute substitution s de $\Gamma \to \Delta$, on a $s = s : \Gamma \to \Delta$.*

Axiomes (Egalité sémantique des substitutions) Si s_1 et s_2 sont deux substitutions telles que $s_1 = s_2 : \Gamma \to \Delta$.

11) $\Delta \vdash T = T'$ implique $\Gamma \vdash [\![T]\!](s_1) = [\![T']\!](s_2)$.

12) $\Delta \vdash t = t' : T$ et $\Delta \vdash T = T'$ impliquent $\Gamma \vdash [\![t]\!](s_1) = [\![t']\!](s_2) : [\![T']\!](s_1)$.

Remarque 7.2 *On déduit de ces 2 axiomes que si s est une substitution de $\Gamma \to \Delta$:*
- *$\Delta\,;T \vdash$ implique $\Gamma\,;[\![T]\!](s) \vdash$.*
- *$\Delta \vdash t : T$ implique $\Gamma \vdash [\![t]\!](s) : [\![T]\!](s)$.*

Intuitivement, avec des variables à la de Bruijn, si $\Gamma \equiv T_1 \dots T_n$, alors id_Γ est la suite des variables de numéro 1 à n.

Axiomes (Substitution identité) Pour tout $\Gamma \equiv T_1 \dots T_n$, la substitution identité sur Γ, notée id_Γ vérifie :

13) $id_\Gamma : \Gamma \to \Gamma$.

14) pour tout T, si $\Gamma\,;T \vdash$ alors $[\![T]\!](id_\Gamma) \equiv T$.

15) pour tout t et tout T, si $\Gamma \vdash t : T$ alors $[\![t]\!](id_\Gamma) \equiv t$.
16) pour tout k et tout s de $\Delta \to \Gamma$, $[\![id_\Gamma(k)]\!](s) \equiv s(k)$.
17) pour tout k, $id_\Gamma(k) \equiv id_{T_1 \ldots T_k}(k)$

Remarque 7.3 *On déduit de l'axiome 13 que $id_\emptyset$ est la suite vide. Donc l'axiome 14 implique que pour tout T tel que $T \vdash$, alors $[\![T]\!](id_\emptyset) \equiv T$. Et l'axiome 15 implique que pour tous t et T tels que $\vdash t : T$ alors $[\![t]\!](id_\emptyset) \equiv t$.*

Remarque 7.4 *En utilisant l'axiome 12 sur id_Γ, on montre que $\Gamma \vdash t = t' : T$ et $\Gamma \vdash T = T'$ impliquent $\Gamma \vdash t = t' : T'$.*

Axiomes (Composition des substitutions) Si s_1 est une substitution de $\Gamma \to \Delta$ et s_2 une substitution de $\Delta \to \Sigma$, alors $s_2 \circ s_1$ est une substitution de $\Gamma \to \Sigma$ telle que :
18) pour tout k de $[1, |\Sigma|]$, $(s_2 \circ s_1)(k) \equiv [\![s_2(k)]\!](s_1)$
19) pour tout T, si $\Sigma\,;\,T \vdash$ alors $[\![T]\!](s_2 \circ s_1) \equiv [\![[\![T]\!](s_2)]\!](s_1)$
20) pour tout t et tout T, si $\Sigma \vdash t : T$ alors $[\![t]\!](s_2 \circ s_1) \equiv [\![[\![t]\!](s_2)]\!](s_1)$

7.2.3 Propriétés fondamentales d'un tel système

On commence par donner 2 propriétés catégoriques vis-à-vis de l'égalité syntaxique.

Lemme 7.1 (élément neutre) *Soit s de $\Gamma \to \Delta$, alors*
$$id_\Delta \circ s \equiv s \qquad et \qquad s \circ id_\Gamma \equiv s$$

Preuve Immédiat en utilisant les axiomes 18, 16 et 15.

□

Lemme 7.2 (associativité de la composition) *Si s_1 de $\Gamma_1 \to \Gamma_2$, s_2 de $\Gamma_2 \to \Gamma_3$ et s_3 de $\Gamma_3 \to \Gamma_4$ alors*
$$((s_3 \circ s_2) \circ s_1) \equiv (s_3 \circ (s_2 \circ s_1))$$

Preuve Immédiat en utilisant les axiomes 18 et 20.

□

On définit l'égalité sémantique sur les contextes, et on montre que les propriétés des substitutions (typage, égalité sémantique, identité et composition) sont compatibles avec cette égalité sémantique.

Définition 7.4 (Jugement d'égalité sur les contextes) *Soient $\Gamma \equiv T_1 \ldots T_n$ et $\Delta \equiv T'_1 \ldots T'_m$ deux contextes, on dit que $\Gamma = \Delta$ ssi $\Gamma \vdash$ et $\Delta \vdash$ et $n = m$ et pour tout k, $T_1 \ldots T_{k-1} \vdash T_k = T'_k$.*

Lemme 7.3 *La relation $. = .$ est une relation d'équivalence sur les contextes bien formés. De plus, $\Gamma \vdash P$ (où P est $T = T'$ ou $t = t' : T$), alors $\Gamma = \Delta$ implique $\Delta \vdash P$.*

Preuve Par itération de l'axiome 8.

□

Lemme 7.4 *Si $\Gamma = \Gamma'$ et $\Delta = \Delta'$ alors $s_1 = s_2 : \Gamma \to \Delta$ implique $s_1 = s_2 : \Gamma' \to \Delta'$*

Preuve Supposons $s_1 = s_2 : \Gamma \to \Delta$ et $|\Delta| = |\Delta'| = n$.
Par le lemme précédent, par la propriété que $\Delta(1)\dots\Delta(k-1) \vdash \Delta(k) = \Delta'(k)$, et l'axiome 12, on montre que pour tout k de $[1,n]$, $\Gamma' \vdash s_1(k) : [\![\Delta'(k)]\!](s_1(1)\dots s_1(k-1))$.
On montre alors de même que pour tout k de $[1,n]$,

$$\Gamma' \vdash s_1(k) = s_2(k) : [\![\Delta'(k)]\!](s_1(1)\dots s_1(k-1))$$

On montre ensuite en utilisant l'axiome 12 que pour tout k,

$$\Gamma' \vdash s_2(k) : [\![\Delta'(k)]\!](s_2(1)\dots s_2(k-1))$$

On en déduit finalement que $s_1 = s_2 : \Gamma' \to \Delta'$

□

Remarque 7.5 *Dans le lemme précédent, on a en particulier montré que si $\Gamma = \Gamma'$, $\Delta = \Delta'$ et $s : \Gamma \to \Delta$ alors $s : \Gamma' \to \Delta'$.*

On en déduit que si $s_1 : \Gamma \to \Delta$ et $s_2 : \Delta' \to \Sigma$ avec $\Delta = \Delta'$, alors $s_2 \circ s_1$ est une substitution de $\Gamma \to \Delta$ qui vérifie les axiomes 18, 19 et 20.

Lemme 7.5 *Si $\Gamma = \Gamma'$ alors $id_\Gamma \equiv id_{\Gamma'}$.*

Preuve En utilisant le lemme 7.4, on montre que $id_{\Gamma'} : \Gamma \to \Gamma$ et $id_\Gamma : \Gamma' \to \Gamma'$. En utilisant le lemme 7.1 on montre alors que $id_\Gamma \circ id_{\Gamma'} \equiv id_{\Gamma'} \equiv id_\Gamma$.

□

Lemme 7.6 *Si $s_1 = s_1' : \Gamma \to \Delta$ et $s_2 = s_2' : \Delta \to \Sigma$ alors $s_2 \circ s_1 = s_2' \circ s_1' : \Gamma \to \Sigma$.*

Preuve En utilisant l'axiome 12, on montre que pour tout k de $[1, |\Sigma|]$ que

$$\Gamma \vdash [\![s_2(k)]\!](s_1) = [\![s_2'(k)]\!](s_1') : [\![\Sigma(k)]\!]([\![s_2(1)]\!](s_1)\dots[\![s_2(k-1)]\!](s_1))$$

□

Remarque 7.6 *Les propriétés qui précèdent montrent que l'on peut construire des catégories ayant les contextes (bien formés) comme objets, et les substitutions (bien typées) comme morphismes. Dans ces constructions catégoriques, on peut considérer au choix l'égalité syntaxique ou l'égalité sémantique.*

Le reste de cette présentation des D-systèmes est maintenant consacrée à la présentation (au niveau des D-systèmes) des différentes notions catégoriques que nous allons introduire et utiliser par la suite.

7.2.4 Notion de section

Les deux définitions suivantes montrent que même si on ne manipule pas explicitement une notion de "termes bien typés" au niveau catégorique, on peut néanmoins exprimer cette notion.

Définition 7.5 (substitution projection) *Soient Γ et T tels que $\Gamma ; T \vdash$, on pose $p_{\Gamma;T} \equiv (id_{\Gamma;T}(k))_{1\le k\le|\Gamma|}$. $p_{\Gamma;T}$ est une substitution de $\Gamma ; T \to \Gamma$ appelée* projection *de $\Gamma ; T$.*

Remarque 7.7 *Intuitivement, le nom de cette substitution vient du fait que $p_{\Gamma\,;T}$ est la suite des variables de Γ : on a donc construit cette substitution en oubliant la dernière variable de $\Gamma\,;T$. Mais paradoxalement, si $\Gamma\,;T' \vdash$, alors $[\![T']\!](p_{\Gamma\,;T}) \equiv T'$: cette substitution ne fait rien!!*

Remarque 7.8 *Avec les mêmes hypothèses, si t est un terme tel que $\Gamma \vdash t : T$, alors on peut définir une substitution s de $\Gamma \to \Gamma\,;T$, en posant $s = id_\Gamma\,;t$. La substitution s vérifie en outre la propriété suivante :*

$$p_{\Gamma\,;T} \circ s \equiv id_\Gamma$$

Cette propriété motive la définition ci-dessous.

Définition 7.6 (section) *Soient Γ et T tels que $\Gamma\,;T \vdash$. Soit s une substitution. On dit que s* est une section de *$\Gamma\,;T$ si et seulement si $s : \Gamma \to \Gamma\,;T$ et*

$$p_{\Gamma\,;T} \circ s \equiv id_\Gamma$$

Remarque 7.9 *On montre facilement que pour tout k de $[1, |\Gamma|]$, $s(k) \equiv id_\Gamma(k)$. On en déduit, que d'après la définition des substitutions, $\Gamma \vdash s(|\Gamma| + 1) : T$.*

Cette dernière remarque montre donc qu'il y a une correspondance bi-univoque entre la notion de section et la notion de termes bien typés, au sens où chaque terme bien typé est représenté par une unique section et réciproquement, chaque section représente un unique terme bien typé.

En particulier, si T est tel que $T \vdash$, T est un type clos (sans variables libres), et toute substitution s de $\emptyset \to T$ est une section de T. Une telle section s est un terme clos.

7.2.5 Construction d'un produit fibré

Etant donné deux contextes Γ et Δ, une substitution s de $\Gamma \to \Delta$ et T tel que $\Delta\,;T \vdash$, on construit maintenant le contexte qui étend Γ par $[\![T]\!](s)$.

Définition 7.7 (Opération d'extension d'un contexte) *Soit $\Delta' \equiv \Delta\,;T$ tel que $\Delta' \vdash$. Soit s de $\Gamma \to \Delta$. On pose*

$$s * \Delta' \equiv \Gamma\,;[\![T]\!](s)$$

On a $s * \Delta' \vdash$.

On définit maintenant la substitution canonique qui étend s.

Définition 7.8 (Opération d'extension d'une substitution) *Avec les mêmes hypothèses que précédemment, on pose*

$$q(s, \Delta') \equiv s\,;id_{s*\Delta'}(|s * \Delta'|)$$

La substitution $q(s, \Delta')$ est la suite qui étend s en lui ajoutant le dernier terme de $id_{s*\Delta'}$.

On exprime maintenant au niveau catégorique que $q(s, \Delta')$ est une extension de s.

Lemme 7.7 *Avec les mêmes hypothèses, $q(s, \Delta')$ est une substitution de $s * \Delta' \to \Delta'$, qui vérifie*

$$s \circ p_{s*\Delta'} \equiv p_{\Delta'} \circ q(s, \Delta') \qquad \begin{tikzcd} s * \Delta' \arrow[r, "q(s,\Delta')"] \arrow[d, "p_{s*\Delta'}"'] & \Delta' \arrow[d, "p_{\Delta'}"] \\ \Gamma \arrow[r, "s"'] & \Delta \end{tikzcd}$$

Preuve On suppose $\Delta \equiv T_1 \dots T_n$.
Par définition du type des substitutions, pour tout $1 \leq k \leq n$, on a
$\Gamma \vdash s(k) : [\![T_k]\!](s(1) \dots s(k-1))$.
Par l'axiome 7, $\Gamma\,;\, [\![T]\!](s) \vdash s(k) : [\![T_k]\!](s(1) \dots s(k-1))$.
De plus, par définition de l'identité (axiome 13), $\Gamma\,;\, [\![T]\!](s) \vdash id_{s*\Delta'}(|s * \Delta'|) : [\![T]\!](s)$.
Donc $q(s, \Delta')$ est bien une substitution de $s * \Delta' \to \Delta'$.
On pose $\phi \equiv s \circ p_{s*\Delta'}$ De plus, pour tout k de $[1, |\Delta|]$,

$$\begin{aligned}
\phi(k) &\equiv [\![s(k)]\!](id_{s*\Delta'}(1) \dots id_{s*\Delta'}(|\Gamma|)) \\
&\equiv [\![s(k)]\!](id_\Gamma(1) \dots id_\Gamma(|\Gamma|)) \\
&\equiv [\![s(k)]\!](id_\Gamma)
\end{aligned}$$

Comme $\Gamma \vdash s(k) : [\![\Delta(k)]\!](s(1) \dots s(k-1))$, on en déduit que $\phi(k) = s(k)$. De plus, comme $k \leq |\Delta|$, $id_\Delta(k) = id_{\Delta'}(k)$:

$$\begin{aligned}
\phi(k) &\equiv s(k) \\
&\equiv [\![id_\Delta(k)]\!](s) \\
&\equiv [\![id_{\Delta'}(k)]\!](s) \\
&\equiv [\![id_{\Delta'}(k)]\!](s\,;\, id_{s*\Delta'}(|s * \Delta'|)) \\
&\equiv [\![p_{\Delta'}(k)]\!](q(s, \Delta')) \\
&\equiv (p_{\Delta'} \circ q(s, \Delta'))(k)
\end{aligned}$$

On vérifie donc que $s \circ p_{s*\Delta'} \equiv p_{\Delta'} \circ q(s, \Delta')$.

□

On va maintenant montrer que cette extension de contexte et de substitution est bien *"canonique"*. En particulier, la définition et le lemme qui suivent montrent que "$*$" est une contruction catégorique bien connue, appelée produit fibré, qui sera présentée plus loin. Ci-dessous $\curlywedge$ dénote la notion d'unique morphisme associé au produit fibré.

Définition 7.9 *Sous les mêmes hypothèses, soient Σ un contexte, s_1 une substitution de $\Sigma \to \Gamma$ et s_2 de $\Sigma \to \Delta'$ tels que $s \circ s_1 = p_{\Delta'} \circ s_2 : \Sigma \to \Delta$. On pose*

$$s_1 \curlywedge_{s*\Delta'} s_2 \equiv s_1\,;\, s_2(|\Delta'|)$$

Lemme 7.8 *Avec les mêmes hypothèses, $s_1 \curlywedge_{s*\Delta'} s_2$ est l'unique substitution (modulo égalité sémantique) ϕ de $\Sigma \to s * \Delta'$ qui vérifie*

$$p_{s*\Delta'} \circ \phi = s_1 : \Sigma \to \Gamma$$

et

$$q(s, \Delta') \circ \phi = s_2 : \Sigma \to \Delta'$$

$$\begin{tikzcd}
\Sigma \arrow[drr, bend left, "s_2"] \arrow[dr, dotted, "s_1 \curlywedge_{s*\Delta'} s_2"] \arrow[ddr, bend right, "s_1"'] & & \\
& s * \Delta' \arrow[r, "q(s,\Delta')"] \arrow[d, "p_{s*\Delta'}"'] & \Delta' \arrow[d, "p_{\Delta'}"] \\
& \Gamma \arrow[r, "s"'] & \Delta
\end{tikzcd}$$

Preuve On pose ici $\phi \equiv s_1 \curlywedge_{s*\Delta'} s_2$. On suppose $\Gamma \equiv T'_1 \dots T'_m$ et $\Delta \equiv T_1 \dots T_n$.
Pour tout k avec $1 \leq k \leq m$, on a $\Sigma \vdash s_1(k) : [\![T'_k]\!](s_1(1) \dots s_1(k-1))$.
De plus $\Sigma \vdash s_2(n+1) : [\![T]\!](s_2(1) \dots s_2(n))$.
Or, $\Sigma \vdash s \circ s_1 = p_{\Delta'} \circ s_2 : \Delta$. Donc pour tout k avec $1 \leq k \leq n$,
$\Sigma \vdash s_2(k) = [\![s(k)]\!](s_1) : [\![T_k]\!](s_2(1) \dots s_2(k-1))$.
Donc, en appliquant les axiomes de composition et les axiomes des substitutions bien typées, on trouve que $\Sigma \vdash [\![T]\!](s_2(1) \dots s_2(n)) = [\![[\![T]\!](s)]\!](s_1)$. On en déduit $\Sigma \vdash s_2(n+1) : [\![[\![T]\!](s)]\!](s_1)$.
Donc, que ϕ est une substitution de $\Sigma \to s * \Delta'$.
On vérifie aisément que $p_{s*\Delta'} \circ \phi \equiv s_1$ et $q(s, \Delta') \circ \phi \equiv s_2$.
Supposons maintenant une substitution ϕ' de $\Sigma \to s * \Delta'$ qui vérifie ces mêmes égalités. On veut montrer $\phi' = \phi : \Sigma \to s * \Delta'$.
Comme $p_{s*\Delta'} \circ \phi' = s_1 : \Sigma \to \Gamma$, pour tout k avec $1 \leq k \leq m$,
$\Sigma \vdash \phi'(k) = s_1(k) : [\![T'_k]\!](s_1(1) \dots s_1(k-1))$. C'est-à-dire pour tout k avec $1 \leq k \leq m$,
$\Sigma \vdash \phi'(k) = \phi(k) : [\![T'_k]\!](s_1(1) \dots s_1(k-1))$.
De même $q(s, \Delta') \circ \phi' = s_2 : \Sigma \to \Delta'$ implique $\Sigma \vdash \phi'(m+1) = s_2(n+1) : [\![T]\!](s_2(1) \dots s_2(n))$.
Donc $\Sigma \vdash \phi'(m+1) = \phi(m+1) : [\![[\![T]\!](s)]\!](s_1)$. On en déduit l'égalité recherchée.

□

Lemme 7.9 *Pour tout Γ et tout T, tels que $\Gamma\,;T \vdash$*

$$id_\Gamma * (\Gamma\,; T) \equiv \Gamma\,; T \qquad \textit{et} \qquad q(id_\Gamma, (\Gamma\,; T)) \equiv id_{\Gamma\,;T}$$

Preuve Immédiat.

□

Lemme 7.10 *Pour tout Γ, Δ, Σ, T, s_1 de $\Gamma \to \Delta$, s_2 de $\Delta \to \Sigma$ et $\Sigma\,; \vdash T$*

$$s_1 * (s_2 * (\Sigma\,; T)) \equiv (s_2 \circ s_1) * (\Sigma\,; T) \quad \textit{et} \quad q(s_2 \circ s_1, (\Sigma\,; T)) \equiv q(s_2, (\Sigma\,; T)) \circ q(s_1, s_2 * (\Sigma\,; T))$$

$$\begin{tikzcd}
s_1 * (s_2 * (\Sigma\,; T)) \arrow[r, "q(s_1, s_2 * (\Sigma\,; T))"'] \arrow[rr, bend left, "q(s_2 \circ s_1, (\Sigma\,; T))"] \arrow[d, "p_{s_1 * (s_2 * (\Sigma\,; T))}"'] & s_2 * (\Sigma\,; T) \arrow[r, "q(s_2, (\Sigma\,; T))"'] \arrow[d, "p_{s_2 * (\Sigma\,; T)}"'] & \Sigma\,; T \arrow[d, "p_{\Sigma\,; T}"] \\
\Gamma \arrow[r, "s_1"'] & \Delta \arrow[r, "s_2"'] & \Sigma
\end{tikzcd}$$

Preuve

1) On a : $s_1 * (s_2 * (\Sigma\,; T)) \equiv \Gamma\,; [\![[\![T]\!](s_2)]\!](s_1)$ et $(s_2 \circ s_1) * (\Sigma\,; T) \equiv \Gamma\,; [\![T]\!](s_2 \circ s_1)$.
L'égalité de ces contextes découle donc des axiomes de la composition.

2) On a $q(s_2 \circ s_1, \Sigma\,; T) \equiv s_2 \circ s_1\,; id_{(s_2 \circ s_1)*(\Sigma\,;T)}(|\Gamma|+1)$.
Donc pour $k \in [1, |\Gamma|]$, en utilisant le fait que $\Delta \vdash s_2(k) : [\![\Sigma(k)]\!](s_2(1) \dots s_2(k-1))$

$$\begin{aligned}
(q(s_2, \Sigma\,; T) \circ q(s_1, s_2 * (\Sigma\,; T)))(k) &\equiv [\![s_2(k)]\!](q(s_1, s_2 * (\Sigma\,; T))) \\
&\equiv [\![s_2(k)]\!](s_1\,; id_{s_1 * (s_2 * (\Sigma\,;T))}(|\Gamma|+1)) \\
&\equiv [\![s_2(k)]\!](s_1) \\
&\equiv (s_2 \circ s_1)(k)
\end{aligned}$$

Pour $k = |\Gamma| + 1$, on a :

$$\begin{aligned}(q(s_2, \Sigma\,;\,T) \circ q(s_1, s_2 * (\Sigma\,;\,T)))(k) &\equiv [\![id_{s_2*(\Sigma\,;\,T)}(k)]\!](q(s_1, s_2 * (\Sigma\,;\,T))) \\ &\equiv q(s_1, s_2 * (\Sigma\,;\,T))(k) \\ &\equiv (s_1\,;\, id_{s_1*(s_2*(\Sigma\,;\,T))}(k))(k) \\ &\equiv id_{s_1*(s_2*(\Sigma\,;\,T))}(k)\end{aligned}$$

On peut donc conclure.

□

Remarque 7.10 *Comme énoncé dans la remarque 7.6, on peut étendre les constructions "$*$", "q" et "$\curlywedge$." avec leurs propriétés, en utilisant l'égalité sémantique sur les contextes et les substitutions, au lieu de la simple égalité syntaxique.*

Si on considère la catégorie ayant les contextes comme objets, et les substitutions comme morphismes, l'opération "$*$" correspond à un certain produit fibré. Cette remarque a conduit Cartmell ([Car86]) à définir la notion de catégorie contextuelle. Cette théorie permet de modéliser les systèmes de types dépendant en laissant implicite la notion de types, de termes et de leurs jugements d'égalité, en s'attachant uniquement aux propriétés algébriques des contextes et des substitutions.

7.2.6 Propriétés de régularité

Les propriétés de régularité, introduites ci-dessous, jouent un rôle important dans notre axiomatisation catégorique. En effet, ces propriétés expriment le fait que la substitution projection de la section précédente est surjective et que les extensions successives de la projection le sont aussi. Il est donc intéressant de montrer à quoi elles correspondent dans les D-systèmes.

Supposons $\Gamma \vdash t : T$ et $\Gamma \vdash t' : T$. On peut donc construire deux substitutions de $\Gamma \to \Gamma\,;\,T$ données par $s_1 = id_\Gamma\,;\,t$ et $s_2 = id_\Gamma\,;\,t'$. Si $\Gamma\,;\,T' \vdash$, alors la surjectivité de $p_{\Gamma\,;\,T'}$ signifie que $s_1 \circ p_{\Gamma\,;\,T'} = s_2 \circ p_{\Gamma\,;\,T'} : \Gamma\,;\,T' \to (\Gamma\,;\,T)$ implique $s_1 = s_2 : \Gamma \to \Gamma\,;\,T$.

Cette dernière implication signifie que $\Gamma\,;\,T' \vdash [\![t]\!](p_{\Gamma\,;\,T'}) = [\![t']\!](p_{\Gamma\,;\,T'}) : [\![T]\!](p_{\Gamma\,;\,T'})$ implique $\Gamma \vdash t = t' : T$. Autrement dit, $\Gamma\,;\,T' \vdash t = t' : T$ implique $\Gamma \vdash t = t' : T$.

Plus généralement, la régularité affirme que pour tout Γ, Δ, T, t et t' tels que $\Gamma\,;\,\Delta \vdash t : T$ et $\Gamma\,;\,\Delta \vdash t' : T$, pour tout T', si dans le contexte formé de $\Gamma\,;\,T'$ suivi du contexte "Δ dont on a relogé les variables", on sait que le jugement "$t = t' : T$ après relocation" est vérifié, alors on a $\Gamma\,;\,\Delta \vdash t = t' : T$. Autrement dit, l'égalité sémantique n'utilise que les informations "minimales" nécessaires au typage des termes.

Les définitions suivantes ont pour but d'exprimer cette relocation des variables. La relocation correspond en fait justement à appliquer les substitutions qui sont des extensions successives de la projection. Par exemple, supposons $\Gamma\,;\,T \vdash t : T'$ et $\Gamma\,;\,T'' \vdash$. On note s la substitution $q(p_{\Gamma\,;\,T''}, (\Gamma\,;\,T))$. On sait que s est une substitution de $\Gamma\,;\,T''\,;\,T \to \Gamma\,;\,T$. On en déduit que $\Gamma\,;\,T''\,;\,T \vdash [\![t]\!](s) : [\![T']\!](s)$. Donc, intuitivement, $[\![t]\!](s)$ est le résultat de la relocation de la variable correspondant à T dans t. Pour cette raison on notera ci-dessous s sous la forme $\uparrow_T^{\Gamma\,;\,T''}$. Le contexte $\Gamma\,;\,T''\,;\,T\,;\,([\![T']\!](s))$ sera dénoté $\Gamma\,;\,T''\,;\,(R_{\Gamma\,;\,T''}(T\,;\,T'))$.

Définition 7.10 *Soient Γ, Δ tels que $\Gamma\,;\,\Delta \vdash$. Soit T tel que $\Gamma\,;\,T \vdash$. On définit de façon mutuellement récursive : $R_{\Gamma\,;\,T}\Delta$ et $\uparrow_\Delta^{\Gamma\,;\,T} : (\Gamma\,;\,T\,;\,R_{\Gamma\,;\,T}\Delta) \to (\Gamma\,;\,\Delta)$.*

1) *si $\Delta \equiv \emptyset$, on pose*

$$R_{\Gamma\,;\,T}\Delta \equiv \emptyset \qquad \textit{et} \qquad \uparrow_\Delta^{\Gamma\,;\,T} \equiv p_{\Gamma\,;\,T}$$

2) *si* $\Delta \equiv \Delta' \,;\, T'$, *on pose*
$R_{\Gamma\,;T}\Delta \equiv (R_{\Gamma\,;T}\Delta')\,;\, [\![T']\!](\uparrow^{\Gamma\,;T}_{\Delta'})$ *et* $\uparrow^{\Gamma\,;T}_{\Delta} \equiv q(\uparrow^{\Gamma\,;T}_{\Delta'}, (\Gamma\,;\,\Delta))$
$\equiv \uparrow^{\Gamma\,;T}_{\Delta'}\,;\,(id_{\Gamma\,;T\,;R_{\Gamma\,;T}\Delta}(|\Gamma\,;\,T\,;\,R_{\Gamma\,;T}\Delta|))$

Au lieu de noter $[\![T]\!](\uparrow^{\Gamma\,;T'}_{\Delta})$, on écrirera plus simplement $\uparrow^{\Gamma\,;T'}_{\Delta} T$ (on prend la même notation sur les termes).

Définition 7.11 *On dit qu'un D-système est régulier sur les types, lorsque pour tout* Γ, Δ, T, T', T'', *si* $\Gamma\,;\,\Delta\,;\,T \vdash$ *et* $\Gamma\,;\,\Delta\,;\,T' \vdash$, *alors* $\Gamma\,;\,T''\,;\,R_{\Gamma\,;T''}\Delta \vdash \uparrow^{\Gamma\,;T''}_{\Delta} T = \uparrow^{\Gamma\,;T''}_{\Delta} T'$ *implique* $\Gamma\,;\,\Delta \vdash T = T'$.

Remarque 7.11 *Un D-système comme le* $\lambda\Pi$*-calcul possède la propriété de régularité sur les types.*

Définition 7.12 *On dit qu'un D-système est régulier sur les contextes, lorsque pour tout* Γ, Δ_1, Δ_2, T, *si* $\Gamma\,;\,\Delta_1 \vdash$ *et* $\Gamma\,;\,\Delta_2 \vdash$ *alors* $\Gamma\,;\,T\,;\,R_{\Gamma\,;T}\Delta_1 = \Gamma\,;\,T\,;\,R_{\Gamma\,;T}\Delta_2$ *implique* $\Gamma\,;\,\Delta_1 = \Gamma\,;\,\Delta_2$.

Remarque 7.12 *La régularité sur les contextes est une simple itération de la régularité sur les types. Et la régularité sur les types est un cas particulier de la régularité sur les contextes.*

Définition 7.13 *On dit qu'un tel système est régulier sur les termes, lorsque pour tout* Γ, Δ, T, t, y, *si* $\Gamma\,;\,\Delta \vdash t : T$ *et* $\Gamma\,;\,\Delta \vdash t' : T$ *alors* $\Gamma\,;\,T''\,;\,R_{\Gamma\,;T''}\Delta \vdash \uparrow^{\Gamma\,;T}_{\Delta} t = \uparrow^{\Gamma\,;T}_{\Delta} t' : \uparrow^{\Gamma\,;T}_{\Delta} T$ *implique* $\Gamma\,;\,\Delta \vdash t = t' : T$.

Remarque 7.13 *Un D-système comme le* $\lambda\Pi$*-calcul possède la propriété de régularité sur les termes.*

Définition 7.14 *On dit qu'un tel système est régulier sur les substitutions, lorsque pour tout* Γ, Δ, T, s_1, s_2, *si* $s_1 : \Gamma\,;\,\Delta \to \Delta'$ *et* $s_2 : \Gamma\,;\,\Delta \to \Delta'$, *alors* $s_1\circ \uparrow^{\Gamma\,;T''}_{\Delta} = s_2\circ \uparrow^{\Gamma\,;T''}_{\Delta} : \Gamma\,;\,T''\,;\,R_{\Gamma\,;T''}\Delta \to \Delta'$ *implique* $s_1 = s_2 : \Gamma\,;\,\Delta \to \Delta'$.

Remarque 7.14 *La régularité sur les substitutions est une simple itération de la régularité sur les termes, et la régularité sur les termes est un cas particulier de la régularité sur les substitutions.*

7.2.7 D-système et FocKer

La définition de D-système permet déjà d'exprimer ce que l'on appelera d'interface dans notre librairie et de collection. Ces notions d'interface et de collection correspondent en fait respectivement aux notions de signatures et d'enregistrements.

Soit $\mathcal{A}$ un ensemble, dit ensemble d'étiquettes. Une signature (ou interface) est une suite finie de couples $(a_1, T_1) \ldots (a_n, T_n)$ tels que :

1) $a_i \in \mathcal{A}$ et pour tout i et j, $a_i \neq a_j$.

2) $T_i \in \mathcal{L}_{Type}$, et $T_1 \ldots T_n \vdash$.

Les "a_i" sont des noms propres à notre système de librairie, qui n'ont rien à voir avec les noms du D-système.

De même, un enregistrement (ou collection) est une suite finie de triplets $(a_1, t_1, T_1) \ldots (a_n, t_n, T_n)$ tels que pour tout k,

1) $a_i \in \mathcal{A}$ et pour tout i et j, $a_i \neq a_j$.

2) Pour tout k, $T_1 \dots T_{k-1} \vdash t_k : T_k$.

Un tel enregistrement a donc une unique signature sous-jacente : $(a_1, T_1) \dots (a_n, T_n)$.

Soit $S_1 = (a_1, T_1) \dots (a_n, T_n)$ et $S_2 = (b_1, T'_1) \dots (b_m, T'_m)$ deux signatures. Une fonction des enregistrements de signature S_1 vers les enregistrements de signature S_2 correspond à une (unique) substitution de $T_1 \dots T_n \rightarrow T'_1 \dots T'_m$.

En s'inspirant de la théorie des catégories contextuelles de Cartmell, on va décrire les catégories de signatures d'enregistrements, en laissant implicite la notion de D-système. L'avantage d'une telle approche est que la modélisation ainsi proposée recouvre aussi la formalisation déjà effectuée en Coq via des types inductifs. En effet, en Coq, on ne peut avoir qu'un accès implicite aux jugements d'interconvertibilité, et aux substitutions de Coq.

En fait, la définition des D-systèmes n'est pas tout-à-fait assez forte, par rapport à nos besoins de FocKer : elle ne fait pas apparaître la possibilité de mettre des définitions dans un contexte. Mais plutôt que d'ajouter cette possibilité aux D-systèmes, on va directement l'ajouter au niveau des catégories de signatures d'enregistrements. On va ainsi étendre progressivement cette théorie catégorique de manière à décrire complètement les unités de notre librairie.

7.3 Préliminaires mathématiques à la présentation du modèle

Dans cette section, nous présentons quelques rappels sur la notion de produit fibré (ou pullback) et sur les arbres à branchement infini, mais dont la longueur des branches est finie.

7.3.1 Notion de pullback (ou produit fibré)

Pour les définitions suivantes on se place sur une catégorie $\mathcal{C}$. On notera $Ob(C)$ la collection des objets de $\mathcal{C}$. On écrira parfois abusivement $\mathcal{C}$ au lieu de $Ob(\mathcal{C})$. Ainsi, si A et B sont deux objets de $\mathcal{C}$, on note abusivement $\mathcal{C}[A, B]$ la collection des morphismes de $\mathcal{C}$ (cf. [Asp91] pour une introduction à la théorie des catégories).

Définition 7.15 *Soient A, I, B trois objets de $\mathcal{C}$, et f de $\mathcal{C}[A, I]$ et g de $\mathcal{C}[B, I]$. Soit P un objet de $\mathcal{C}$, i de $\mathcal{C}[P, A]$ et j de $\mathcal{C}[P, B]$, on dit que $< i, j >$* est un cône de pullback dans $\mathcal{C}$ du diagramme *$< f, g >$, ssi*

1) *$f \circ i = g \circ j$, c'est-à-dire, le diagramme suivant commute*

$$\begin{tikzcd} P \arrow[r, "j"] \arrow[d, "i"'] & B \arrow[d, "g"] \\ A \arrow[r, "f"'] & I \end{tikzcd}$$

2) *Pour tous C de $\mathcal{C}$, s de $\mathcal{C}[C, A]$, t de $\mathcal{C}[C, B]$ tel que $f \circ s = g \circ t$, il existe un unique morphisme m de $\mathcal{C}[C, P]$ tel que*

$$i \circ m = s$$

et

$$j \circ m = t$$

On note $s \curlywedge_P t$ cet unique morphisme

$$\begin{tikzcd} C \arrow[dr, dotted, "m"] \arrow[drr, bend left, "t"] \arrow[ddr, bend right, "s"'] & & \\ & P \arrow[r, "j"] \arrow[d, "i"'] & B \arrow[d, "g"] \\ & A \arrow[r, "f"'] & I \end{tikzcd}$$

Remarque 7.15 *On suppose maintenant que sous les hypothèses précédentes* $< i, j >$ *est un cône de pullback de* $< f, g >$*. Alors par symétrie,* $< j, i >$ *est un cône de pullback de* $< g, f >$*.*

Remarque 7.16 *Avec les mêmes hypothèses, pour tous* C *de* $\mathcal{C}$*,* s *de* $\mathcal{C}[C, A]$*,* t *de* $\mathcal{C}[C, B]$*, et* m *de* $\mathcal{C}[C, P]$*, si* $i \circ m = s$ *et* $j \circ m = t$ *alors* $m = s \curlywedge_P t$*. Autrement dit, il suffit de montrer que* m *vérifie ces deux égalités pour savoir que* $s \curlywedge_P t$ *existe.*

En effet, dans ce cas, on a :

$$\begin{aligned} & f \circ s \\ & \quad = f \circ i \circ m \\ & \quad = g \circ j \circ m \\ & \quad = g \circ t \end{aligned}$$

On en déduit que $s \curlywedge_P t$ *existe et étant donné les équations vérifiées par* m*, alors* $m = s \curlywedge_P t$*.*

On déduit par exemple de cette remarque que sous les hypothèses précédentes, comme $i \circ id_P = i$ et $j \circ id_P = j$ alors $i \curlywedge_P j = id_P$.

Lemme 7.11 *Avec les mêmes hypothèses, soient* C *de* $\mathcal{C}$*,* s *de* $\mathcal{C}[C, A]$*,* t *de* $\mathcal{C}[C, B]$ *tels que* $f \circ s = g \circ t$*. Soit* D *de* $\mathcal{C}$ *et* h *de* $\mathcal{C}[D, C]$*,*
on a

$$(s \curlywedge_P t) \circ h = (s \circ h) \curlywedge_P (t \circ h)$$

$$\begin{tikzcd}
D \arrow[dr, "h"] & & & \\
& C \arrow[dr, dotted, "s \curlywedge_P t"] \arrow[drr, bend left, "t"] \arrow[ddr, bend right, "s"'] & & \\
& & P \arrow[r, "j"] \arrow[d, "i"'] & B \arrow[d, "g"] \\
& & A \arrow[r, "f"'] & I
\end{tikzcd}$$

Preuve En utilisant la remarque précédente, il suffit de montrer que

$$i \circ ((s \curlywedge_P t) \circ h) = s \circ h \qquad \text{et} \qquad j \circ ((s \curlywedge_P t) \circ h) = t \circ h$$

Ces deux égalités découlent immédiatement de la définition de $s \curlywedge_P t$.

□

Lemme 7.12 (Unicité des pullbacks modulo isomorphisme) *Soient* A, I, B *trois objets de* $\mathcal{C}$*, et* f *de* $\mathcal{C}[A, I]$ *et* g *de* $\mathcal{C}[B, I]$*. Pour* $k = 1$ *ou* $k = 2$*, soient* P_k *de* $Ob(\mathcal{C})$*,* i_k *de* $\mathcal{C}[P_k, A]$ *et* j_k *de* $\mathcal{C}[P_k, B]$ *tels que* $< i_k, j_k >$ *soit un cône de pullback de* $< f, g >$*. Les objets* P_1 *et* P_2 *sont isomorphes.*

Preuve On pose $\phi = i_2 \curlywedge_{P_1} j_2$ (resp. $\psi = i_1 \curlywedge_{P_2} j_1$), c'est un morphisme de $\mathcal{C}[P_2, P_1]$ (resp. $\mathcal{C}[P_1, P_2]$).
D'après le lemme 7.11,

$$\begin{aligned} & \phi \circ \psi \\ & \quad = (i_2 \circ \psi) \curlywedge_{P_1} (j_2 \circ \psi) \\ & \quad = i_1 \curlywedge_{P_1} j_1 \\ & \quad = id_{P_1} \end{aligned}$$

Par symétrie du raisonnement, on en déduit que $\psi \circ \phi = id_{P_2}$. Donc P_1 et P_2 sont isomorphes.

□

Lemme 7.13 *Soient A et B de $Ob(\mathcal{C})$ et f de $\mathcal{C}[A,B]$. Alors, $< f, id_B >$ est un cône de pullback de $< id_A, f >$.*

$$\begin{tikzcd}
B \arrow[r, "id_B"] \arrow[d, "f"'] & B \arrow[d, "f"] \\
A \arrow[r, "id_A"'] & A
\end{tikzcd}$$

Preuve Supposons C de $Ob(\mathcal{C})$, s de $\mathcal{C}[C,A]$ et t de $\mathcal{C}[C,B]$ tels que $s = f \circ t$. Il est trivial de vérifier que t est l'unique morphisme de $\mathcal{C}[C,B]$ qui fait commuter le diagramme.

□

Lemme 7.14 *La composition de 2 pullbacks est un pullback.*

Preuve On considère le diagramme ci-dessous. Supposons que $< i_1, j_1 >$ (resp. $< i_2, j_2 >$) soit un cône de pullback de $< f_1, g >$ (resp. $< f_2, i_1 >$).

$$\begin{tikzcd}
P_2 \arrow[r, "j_2"] \arrow[d, "i_2"'] & P_1 \arrow[r, "j_1"] \arrow[d, "i_1"'] & B \arrow[d, "g"] \\
A_2 \arrow[r, "f_2"'] & A_1 \arrow[r, "f_1"'] & I
\end{tikzcd}$$

Alors $< i_2, j_1 \circ j_2 >$ est un cône de pullback de $< f_1 \circ f_2, g >$.

En effet, soient C de $\mathcal{C}$, s de $\mathcal{C}[C, A_2]$ et t de $\mathcal{C}[C,B]$, tels que $f_1 \circ f_2 \circ s = g \circ t$.
On pose $\phi = (f_2 \circ s) \curlywedge_{P_1} t$.
Par définition de ϕ, on a : $s \circ f_2 = i_1 \circ \phi$. On pose $\psi = s \curlywedge_{P_2} \phi$ (sur le cône $< i_2, j_2 >$).
Par définition de ψ, on a : $i_2 \circ \psi = s$ et

$$\begin{aligned}
&j_1 \circ j_2 \circ \psi \\
&\quad = j_1 \circ \phi \\
&\quad = t
\end{aligned}$$

Vérifions que ψ est l'unique morphisme à vérifier ces 2 égalités. Supposons un morphisme m qui vérifie $i_2 \circ m = s$ et $j_1 \circ j_2 \circ m = t$. On a :

$$\begin{aligned}
&i_1 \circ j_2 \circ m \\
&\quad = f_2 \circ i_2 \circ m \\
&\quad = f_2 \circ s
\end{aligned}$$

Donc par unicité de ϕ, on en déduit que $\phi = j_2 \circ m$. Par unicité de ψ sur le cône $< i_2, j_2 >$ on en déduit que $m = \psi$.

□

7.3.2 Notion d'arbre

On s'intéresse ici à des arbres, dont la longueur des branches est finie, mais le branchement potentiellement infini. En effet, on peut les signatures vues en 7.2.7 comme des nœuds d'un arbre, en considérant que le nœud père d'une signature A est la signature obtenue en oubliant le dernier champs de A. La racine de cet arbre est alors la signature vide. Comme on ne manipule que des signatures finies, la longueur des branches est finie. Mais, à partir d'une signature A, on peut potentiellement imaginer une infinité de signature qui ajoute un champs à A : le branchement est donc potentiellement infini. Cette présentation des arbres est tirée de [Str91].

Définition 7.16 *Soit $\mathcal{C}$ un ensemble non vide, et $\mathcal{F}$ une fonction de $\mathcal{C}$ dans $\mathcal{C}$. $< \mathcal{C}, \mathcal{F} >$ est* un arbre *ssi :*

1) *pour tout A de $\mathcal{C}$, il existe un entier n tel que $\mathcal{F}^{n+1}(A) = \mathcal{F}^n(A)$*

2) *pour tout A et B de $\mathcal{C}$, $\mathcal{F}(A) = A$ et $\mathcal{F}(B) = B$ implique $A = B$.*

$\mathcal{F}$ s'appelle la fonction *père* de $\mathcal{C}$. Pour tout A, on note $|A|$ le plus petit n tel que $\mathcal{F}^{n+1}(A) = \mathcal{F}^n(A)$. $|.|$ définit une fonction, appelée *profondeur*.

Comme $\mathcal{C}$ est non vide, d'après le point 1 de la définition, il existe E de $\mathcal{C}$ tel que $\mathcal{F}(E) = E$, et d'après le point 2, E est l'unique élément de $\mathcal{C}$ à vérifier cette propriété. E s'appelle *racine* de l'arbre. On a $|E| = 0$.

Chapitre 8

Catégories de signatures d'enregistrements (ou CS)

La première étape de notre modélisation est de donner une définition déclarative (par propriétés) de la notion d'enregistrement avec types de champs dépendants. La section précédente a permis de montrer comment simuler un tel système à partir d'un système de types dépendants. Elle a aussi permis de dégager un certain nombre de propriétés vérifiées par ces systèmes. C'est justement de ces propriétés qu'est issue la définition de catégories de signatures d'enregistrements.

Ces catégories modélisent la notion de système interfaces/collections de FocKer. Les interfaces sont représentées par les objets de ces catégories. Les morphismes de ces catégories représentent des fonctions entre collections. Ces catégories ont un objet terminal $\mathbf{1}$ (l'interface vide) : les collections sont donc les morphismes de domaine $\mathbf{1}$.

Dans ces catégories, on va chercher à exprimer les notions de renommage, d'oubli, de conversion. On va aussi chercher à exprimer la notion de déclaration de méthodes et de définition de méthodes pour faire apparaître progressivement la notion d'espèces.

8.1 Définition et exemples

La définition suivante est très fortement inspirée des catégories contextuelles de Cartmell. On a ajouté une notion d'étiquette, qui permet d'exprimer plus simplement les propriétés de l'oubli de champ.

Les champs d'une signature portent ainsi chacun une étiquette appartenant à un certain ensemble ${}^{\mathtt{e}}\mathtt{A}$. Ces étiquettes sont supposées deux à deux distinctes. Les signatures ont de plus une structure d'arbre : le père d'une signature A étant A privé de son dernier champ. La profondeur de A (notée $|A|$) correspond donc au nombre de champs de A. Chaque étiquette d'une signature est numérotée par la profondeur de champs auquel elle correspond. Enfin, on a une opération d'extension des signatures qui est axiomatisée par un produit fibré.

Définition 8.1 *Une* catégorie de signatures d'enregistrements *(ou CS en abrégé)* $\mathfrak{C}$ *est un 8-uplet* $< \mathcal{C}, {}^{\mathtt{e}}\mathtt{A}, \mathcal{F}, \mathfrak{L}, \mathbf{1}, p, *, q >$ *tel que :*

1) *$\mathcal{C}$ est une catégorie munie d'un objet terminal* $\mathbf{1}$.

2) *${}^{\mathtt{e}}\mathtt{A}$ est un ensemble (dit ensemble d'étiquettes)*

3) *$< Ob(\mathcal{C}), \mathcal{F} >$ est un arbre, dont $\mathbf{1}$ est la racine (ie. tel que $\mathcal{F}(\mathbf{1}) = \mathbf{1}$). Intuitivement, $\mathcal{F}$ représente l'oubli du dernier champ d'une signature. Donc l'ensemble des sous-nœuds d'une* *signature représente l'ensemble de ses champs.*

4) *Pour tout objet A de $\mathcal{C}$, il existe une fonction injective $\mathfrak{L}_A$ de $[1, |A|]$ dans ${}^{\mathfrak{e}}\mathtt{A}$. Autrement dit, chaque champ porte une étiquette unique.*

5) *Pour tout A, et $k \in [1, |\mathcal{F}(A)|]$, $\mathfrak{L}_{\mathcal{F}(A)}(k) = \mathfrak{L}_A(k)$. L'étiquetage est cohérent par oubli du dernier champ.*

6) *Pour tout objet A de $\mathcal{C}$, différent de **1**, il existe un morphisme, appelé projection de A et noté p_A, de $\mathcal{C}[A, \mathcal{F}(A)]$.*

7) *Pour tous A et B de $Ob(\mathcal{C})$, pour tout a tel que $a \notin Im(\mathfrak{L}_A)$, pour tout f de $\mathcal{C}[A, \mathcal{F}(B)]$, il existe un objet $(f *^a B)$ de $\mathcal{C}$ et un morphisme $q^a(f, B)$ de $\mathcal{C}[f *^a B, B]$, tel que*

$$\mathcal{F}(f *^a B) = A \qquad et \qquad \mathfrak{L}_{f*^a B}(|A| + 1) = a$$

et tel que $< p_{f*^a B}, q^a(f, B) >$ est un cône de pullback dans $\mathcal{C}$ du diagramme $< f, p_B >$:

$$\begin{tikzcd}
f *^a B \arrow[r, "q^a(f,B)"] \arrow[d, "p_{f*^a B}"'] & B \arrow[d, "p_B"] \\
A \arrow[r, "f"'] & \mathcal{F}(B)
\end{tikzcd}$$

Autrement dit, $f *^a B$ représente la signature A qu'on a étendue en lui ajoutant un champ d'étiquette a (qui ne figure pas dans A). Ce champ correspond, via f, au champ ajouté à $\mathcal{F}(B)$ pour obtenir B. Le morphisme $q^a(f, B)$ exprime cette correspondance.

8) *Pour tout B de $Ob(\mathcal{C})$, $id_{\mathcal{F}(B)} *^{\mathfrak{L}_B(|B|)} B = B$ et $q^{\mathfrak{L}_B(|B|)}(id_{\mathcal{F}(B)}, B) = id_B$*

$$\begin{tikzcd}
id_{\mathcal{F}(B)} *^{\mathfrak{L}_B(|B|)} B = B \arrow[r, "q^{\mathfrak{L}_B(|B|)}(id_{\mathcal{F}(B)}, B) = id_B"] \arrow[d, "p_B"'] & B \arrow[d, "p_B"] \\
\mathcal{F}(B) \arrow[r, "id_{\mathcal{F}(B)}"'] & \mathcal{F}(B)
\end{tikzcd}$$

9) *Pout tous A, B, C de $Ob(\mathcal{C})$, pour tous a et b tels que $a \notin Im(\mathfrak{L}_A)$ et $b \notin Im(\mathfrak{L}_B)$, pour tous f de $\mathcal{C}[A, B]$ et g de $\mathcal{C}[B, \mathcal{F}(C)]$,*

$$(g \circ f) *^a C = f *^a (g *^b C)$$

et

$$q^a(g \circ f, C) = q^b(g, C) \circ q^a(f, g *^b C)$$

$$\begin{tikzcd}
f *^a (g *^b C) \arrow[r, "q^a(f, g *^b C)"'] \arrow[rr, bend left, "q^a(g \circ f, C)"] \arrow[d, "p_{f*^a(g*^b C)}"'] & g *^b C \arrow[r, "q^b(g, C)"'] \arrow[d, "p_{g*^b C}"'] & C \arrow[d, "p_C"] \\
A \arrow[r, "f"'] & B \arrow[r, "g"'] & \mathcal{F}(C)
\end{tikzcd}$$

Remarque 8.1 *Remarquons que les axiomes 8 et 9 renforcent l'opération de pullback. Sans ces axiomes, on n'aurait en effet que des relations d'isomorphisme au lieu des relations d'égalité.*

On commence par montrer 2 petits lemmes techniques de composition du pullback.

Lemme 8.1 *Pout tous A, B, C de $Ob(\mathcal{C})$, a et b de ${}^{\mathfrak{e}}\mathtt{A}$ tels que $a \notin Im(\mathfrak{L}_A)$ et $b \notin Im(\mathfrak{L}_B)$, et f de $\mathcal{C}[A, B]$ et g de $\mathcal{C}[B, \mathcal{F}(C)]$, on a :*

$$q^a(f, g *^b C) = (f \circ p_{f*^a(g*^b C)}) \curlywedge_{g*^b C} q^a(g \circ f, C)$$

Preuve C'est une autre formulation de la deuxième égalité de l'axiome 9 :

$$\begin{tikzcd}
f *^a (g *^b C) \arrow[d, "p_{f*^a(g*^b C)}"'] \arrow[dr, dotted, "q^a(f, g *^b C)"] \arrow[drr, bend left, "q^a(g \circ f, C)"] & & \\
A \arrow[dr, bend right, "f"'] & g *^b C \arrow[r, "q^b(g,C)"'] \arrow[d, "p_{g*^b C}"'] & C \arrow[d, "p_C"] \\
& B \arrow[r, "g"'] & \mathcal{F}(C)
\end{tikzcd}$$

□

Lemme 8.2 *Si $a \notin Im(\mathfrak{L}_A)$ et $b \notin Im(\mathfrak{L}_C)$, pour tous f de $\mathcal{C}[A, \mathcal{F}(B)]$, g de $\mathcal{C}[C, \mathcal{F}(D)]$, i de $\mathcal{C}[E, A]$, j de $\mathcal{C}[E, B]$ tels que $p_B \circ j = f \circ i$, k de $\mathcal{C}[B, C]$ et l de $\mathcal{C}[A, D]$ tels que $p_D \circ l \circ p_{f*^a B} = g \circ k \circ q^a(f, B)$, on a :*

$$((k \circ q^a(f,B)) \curlywedge_{g*^b D} (l \circ p_{f*B})) \circ (i \curlywedge_{f*B} j) = (k \circ j) \curlywedge_{g*^b D} (l \circ i)$$

$$\begin{tikzcd}
E \arrow[dr, dotted, "i \curlywedge_{f*^a B} j"'] \arrow[drr, bend left, "i"] \arrow[ddr, bend right, "j"'] & & & & \\
& f *^a B \arrow[r, "p_{f*^a B}"] \arrow[d, "q^a(f,B)"'] \arrow[ddrr, dotted, "(k \circ q^a(f,B)) \curlywedge_{g*^b D} (l \circ p_{f*B})"'] & A \arrow[d, "f"] \arrow[ddrr, bend left, "l"] & & \\
& B \arrow[r, "p_B"] \arrow[ddrr, bend right, "k"'] & \mathcal{F}(B) & & \\
& & & g *^b D \arrow[r, "q^b(g,D)"] \arrow[d, "p_{g*^b D}"'] & D \arrow[d, "p_D"] \\
& & & C \arrow[r, "g"'] & \mathcal{F}(D)
\end{tikzcd}$$

Preuve On pose $\phi = i \curlywedge_{f*^a B} j$ et $\psi = (k \circ q^a(f, B)) \curlywedge_{g*^b D} (l \circ p_{f*^a B})$. On a :

$$\begin{aligned}
&p_{g*^b C} \circ \psi \circ \phi \\
&\quad = k \circ p_{f*^a B} \circ \phi \\
&\quad = k \circ j
\end{aligned}
\qquad\qquad
\begin{aligned}
&q^b(g, D) \circ \psi \circ \phi \\
&\quad = l \circ q^a(f, B) \circ \phi \\
&\quad = l \circ i
\end{aligned}$$

On conclut que $(k \circ j) \curlywedge_{g*^b D} (l \circ i)$ existe et vaut $\psi \circ \phi$.

□

8.1.1 Exemple des enregistrements associatifs à gauche

Un modèle naturel de cette axiomatisation catégorique des enregistrements avec type de champs dépendants est un système d'enregistrement associatif à gauche. En effet, si on considère la notion d'enregistrement bâtie sur les D-systèmes et présentée en section 7.2.7, le pullback correspond à une opération d'extension des signatures par la droite.

On aimerait donc montrer que le système de Pollack (cf. section 6.5.1) vérifie ces axiomes. Le système de Pollack étant exprimé sous forme de règles d'inférence *en théorie des types*, on cherche donc à montrer les propriétés catégoriques de ce système en théorie des types. Malheureusement, en toute rigueur, cette tentative semble vouée à l'échec. En effet, il semble que de telles propriétés catégoriques nécessitent des η-règles (cf. section 8.1.1.b ci-dessous). Or en présence de types dépendants et de la règle de conversion, qui permet de substituer un type par un autre qui lui est interconvertible lors de la vérification de type, les η-règles empêchent le système de vérifier la propriété de Church-Rosser (cf. [Luo94]).

Malgré tout, on sait qu'il existe des modèles extensionnels de la théorie des types, et en particulier des modèles où ces η-règles peuvent être vérifiées. Les constructions qui suivent n'auront donc une validité que dans de tels modèles.

La présentation de cette exemple n'est pas très rigoureuse, et se veut surtout intuitive. Cet exemple permet en outre d'illustrer assez simplement les extensions des catégories de signatures avec des champs manifestes (cf. chapitre 10).

8.1.1.a Rappel sur le système d'enregistrements à gauche

Plus que le rappel de la section 6.5.1, cette section est l'occasion de justifier pourquoi nous considérons le système suivant, qui n'est pas exactement celui présenté par Pollack :

$$Sign : {}^{\mathfrak{e}}\mathtt{A}^* \to Type$$

$$\frac{}{\{\} : Sign_{\emptyset}}\ \textbf{EForm} \qquad \frac{A : Sign_l \quad a \notin l \quad U : A \to Type}{\{A\ ;\ a : U\} : Sign_{l;a}}\ \textbf{CForm}$$

$$\frac{}{<> : \{\}}\ \textbf{EIntro} \qquad \frac{\{A\ ;\ a : U\} : Sign_{l;a} \quad X : A \quad x : (U\ X)}{<X\ ;\ a = x> : \{A\ ;\ a : U\}}\ \textbf{CIntro}$$

En fait, dans [Pol00], Pollack donne une version simplifiée du système où $Sign_{.}$ n'apparaît pas, et est remplacé par $Type$. La signature de l'enregistrement vide est alors le type `Unit`, c'est-à-dire le type avec un unique constructeur constant. Mais cette présentation ne convient pas tout-à-fait ici. D'une part, elle ne garantit pas l'unicité des étiquettes. D'autre part, dans la règle `CForm`, A peut être de n'importe quel type, les signatures d'enregistrements de cette présentation ne forment donc pas un arbre, mais plutôt une forêt. En effet, il n'y a pas une unique racine puisque tout type est potentiellement racine.

8.1.1.b Codage de la description catégorique en théorie des types

Intuitivement, pour écrire le système de Pollack, comme une CS *en théorie des types*, on a envie de prendre les signatures des enregistrements (les termes entre "{.}") comme objets, et les fonctions entre enregistrements comme morphismes. Mais ceci exige une règle de η-conversion. En effet, dans cette approche, l'identité est la fonction $[X]X$ et la composition $G \circ F$ de 2 morphismes F et G est le morphisme $[X](G\ (F\ X))$. Vérifier l'axiome catégorique $F \circ id = F$ revient à vérifier $[X](F\ X) = F$.

On a vu que dans les D-systèmes (cf. 7.2.2), on peut exprimer ces propriétés catégoriques en prenant les contextes comme objet et les substitutions comme morphismes. Cette approche n'est pas possible ici, car il faudrait être capable de parler à l'intérieur de la théorie des pré-termes (termes "non typés") et du mécanisme de substitution (opérateur $[\![]\!]$).

Pour traiter cet exemple des systèmes d'enregistrements en toute rigueur, il faudrait donc définir une notion de substitution sur les enregistrements : on prendrait alors les pré-enregistrements (enregistrements pas forcément bien typés) comme morphismes, et on décrirait la composition à partir de l'opération de substitution. En fait, cela revient à dire que dans le texte qui suit, on peut interpréter tout λ-terme de la forme "$[x]b$" par le pré-terme "b", et toute application "$([x]b\ u)$" par la substitution $b{<}u/x{>}$. Avec cette interprétation les η-règles deviennent des propriétés syntaxiques des substitutions.

8.1.1.c Construction catégorique

Comme on l'a dit précédemment, cette construction catégorique n'est pas valable en théorie des types, mais seulement dans les modèles de la théorie des types qui possède des η-règles (η-conversion et surjective pairing).

- L'objet terminal de cette catégorie sera donc $\{\}$. L'unicité du morphisme de codomaine $\{\}$ est due à la η-conversion. La fonction père est définie dans ce système avec la règle suivante :

  $$\frac{}{\mathcal{F}(\{\}) \overset{def}{=} \{\}}\ \mathcal{F}\textbf{-DefE} \qquad \frac{\{A\ ;\ a:U\} : Sign_{l;a}}{\mathcal{F}(\{A\ ;\ a:U\}) \overset{def}{=} A}\ \mathcal{F}\textbf{-DefC}$$

  On se convaint facilement que les types enregistrements forment un arbre. On montre en effet par récurrence que pour tout A de type $Sign_l$, si n est la longueur de l alors $\mathcal{F}^n(A) = \{\}$. De plus $\mathcal{F}(A) = A$ implique $A = \{\}$.
- De même le morphisme projection s'écrit :

  $$\frac{<X\ ;\ a = x> : \{A\ ;\ a:U\}}{p_{\{A\ ;\ a:U\}}(<X\ ;\ a = x>) \overset{def}{=} X}\ p\textbf{-Def}$$

  Dans la suite, on définit aussi une autre projection (mais qui n'est pas un "morphisme"), appelée $L_{\{A\ ;\ a:U\}}$, et qui sélectionne le dernier champ d'un enregistrement. $L_{\{A\ ;\ a:U\}}$ de type $(X : \{A\ ;\ a:U\})(U\ (p_{\{A\ ;\ a:U\}}\ X))$ est défini par :

  $$\frac{\{A\ ;\ a:U\} : Sign_{l;a}}{L_{\{A\ ;\ a:U\}}(<X\ ;\ a = t>) \overset{def}{=} t}\ L\textbf{-Def}$$

  On suppose maintenant la propriété de "*surjective pairing*", c'est-à-dire pour tout $X : \{A\ ;\ a:U\}$, $X = <(p_{\{A\ ;\ a:U\}}\ X); a = (L_{\{A\ ;\ a:U\}}\ X)>$.
- L'opération "$*$" est définie par la règle suivante :

  $$\frac{A_1 : Sign_{l_1} \quad b \notin l_1 \quad F : A_1 \to A_2 \quad \{A_2\ ;\ a:U\} : Sign_{l_2;a}}{F *^b \{A_2\ ;\ a:U\} \overset{def}{=} \{A_1\ ;\ b : [X : A_1](U\ (F\ X))\}}\ *\textbf{-Def}$$

  Cette règle est correcte, car étant donné son contexte, $\{A_1\ ;\ b : [X : A_1](U\ (F\ X))\}$ est bien typé de type $Sign_{l_1;b}$.

  Il est immédiat de voir que $\mathcal{F}(F *^b \{A_2\ ;\ a:U\}) = A_1$. Par contre, pour montrer que $[X : A_2]X *^a \{A_2\ ;\ a:U\} = \{A_2\ ;\ a:U\}$, il faut avoir que $U = [X : A_2](U\ X)$. Autrement dit, on doit là-encore utiliser la η-conversion.

– Si $B = \{A_2\ ;\ a : U\}$, alors sous les hypothèses, $A_1 : Sign_{l_1}$, $b \notin l_1$, $F : A_1 \to A_2$ et $B : Sign_{l_2;a}$, le morphisme "q" est défini par :

$$q^b(F,B) \stackrel{def}{=} [X : F *^b B]<(F\ (p_{F*^bB}\ X)); a = (L_{F*^bB}\ X)> \qquad \textbf{q-Def}$$

Sous l'hypothèse de la validité de la η-conversion et de surjective pairing, on vérifie que $*$ et q ainsi définis vérifient bien les propriétés exigées. Il reste à exhiber la construction de l'unique morphisme de pullback.

– Sous les hypothèses du point précédant, l'unique morphisme de pullback est alors défini par la règle suivante :

$$\frac{A_3 : Sign_{l_3} \quad G : A_3 \to A_1 \quad H : A_3 \to B \quad p_B \circ H = F \circ G}{(G \curlywedge_{F*^bB} H) \stackrel{def}{=} [X : A_3]<(G\ X); b = (L_B\ (H\ X))>} \curlywedge \textbf{-Def}$$

Il est intéressant de remarquer que si F est une fonction de cœrcion au sens de P. Aczel (dont les cœrcions Coq sont une implantation, voir 3.2.2.b), alors $q^b(F, \{A_2\ ;\ a : U\})$ est un prolongement de cette cœrcion au niveau de $F *^b \{A_2\ ;\ a : U\}$. C'est le prolongement proposé par Z. Luo [Luo99] au niveau des Σ-types, et repris par Pollack dans [Pol00].

8.1.2 Exemple des Drecords

On va maintenant esquisser le lien entre les enregistrements codés en Coq, et cette présentation catégorique. On n'a pas complètement démontré ce lien formellement en Coq, car cela semble assez difficile. D'une part, Coq n'ayant pas la règle de η-conversion (cf. discussion en section 8.1.1.b), on ne peut pas prendre les signatures de Drecords comme objets : il faudrait les considérer modulo une relation d'équivalence. D'autre part, étant donné les problèmes posés par l'égalité avec les types dépendants[1] il semble très technique de montrer des propriétés équivalentes aux axiomes 8 et 9 précédents. On va donc juste esquisser ici la construction du pullback en Coq (axiomes 1 à 7).

Pour simplifier le problème, on va se placer sur les télescopes (cf. section 6.1.2) : on a déjà vu que l'unicité des étiquettes n'est pas vraiment la source de difficulté. Une des difficultés pour établir le lien entre le codage Coq et cette axiomatisation catégorique des enregistrements, vient du fait que le modèle naturel de cette axiomatisation est un système d'enregistrement associatif à gauche : l'opérateur "$*$" est un opérateur de fusion/extension de signature à droite. Or, les télescopes que nous avons codés sont associatifs à droite : ils s'étendent par la gauche.

8.1.2.a Simulation du caractère associatif à gauche

Il faut donc se donner 2 constructeurs et 2 destructeurs sur les sigtels et les impltels pour simuler le caractère associatif à gauche. Au niveau des sigtels, les constructeurs sont le sigtel vide, et l'extension par un champ à droite. Les destructeurs sont la projection (la signature privée du dernier champ) et la factorisation (c'est-à-dire, le dernier champ donné dans le contexte des autres champs).

Niveau des sigtels Pour les sigtels, le travail est donc déjà presque fait. Le sigtel vide est `Esig`. L'extension d'un champ à droite est un cas particulier de `flatsig` : on ajoute 1 champ au lieu de n. De même, la factorisation est un cas particulier de `apsig`. Avant de donner ces constructions, on définit `Eimpl` et `Cimpl` les deux constructeurs correspondant au niveau des télescopes à ceux de la définition 6.1.

[1] Par exemple, si `==` est l'égalité sur les propositions, alors la proposition `(A :Set ;x :A ;H :x=x)H==(refl_equal A x)` n'est pas démontrable en Coq

```
Definition Eimpl: (impltel Esig) := IT.

Definition Cimpl: (a:A; l:(list A); T:Type; f:T ->(sigtel l); x:T)(impltel (f x)) ->
  (impltel (Csig a f)) :=
   [a;l;T;f;x;m](Build_dpairT 2![x:T](impltel (f x)) a m).
```

On définit ensuite `flat1` le cas particulier de `flatsig` qui ajoute seulement 1 champ au lieu de n.

```
Fixpoint flat1[l:(list A)]:(a:A; s:(sigtel l))((impltel s)->Type)->(sigtel (cons a l)):=
 <[l:(list A)](a:A; s:(sigtel l))((impltel s)->Type)->(sigtel (cons a l))>Cases l of
   nil => [a;s;F](Csig a [_:(F Eimpl)]Esig)
 | (cons b m) => [a; s;F](Csig a [x:(dom s)]
            (flat1 b [i:(impltel (s x))](F (Cimpl b 4![x:(dom s)](s x) i))))
 end.
```

On définit maintenant la fonction `father` qui correspond à la fonction $\mathcal{F}$ des catégories de signatures d'enregistrements.

```
Fixpoint father[l:(list A)]:(a:A)(sigtel (cons a l))->(sigtel l) :=
  <[l:(list A)](a:A)(sigtel (cons a l))->(sigtel l)>Cases l of
      nil => [a; s]Esig
    | (cons b m) => [_; s](Csig b [x:(dom s)](father (s x)))
  end.
```

On définit `app1` le cas particulier de `apsig`, lorsqu'on factorise `(father s)` sur une signature `s`.

```
Fixpoint app1[l:(list A)]:(a:A; s:(sigtel (cons a l)))(impltel (father s))->Type:=
<[l:(list A)](a:A; s:(sigtel (cons a l)))(impltel (father s))->Type>Cases l of
   nil => [_;s;d](dom s)
 | (cons a l) => [_;s;d](app1 (dprojT2 d))
end.
```

Mais là commencent les premières difficultés. Sous les hypothèses, `l :(list A)`, `a :A`, `s :(sigtel l)`, `f :(impltel s)`$\rightarrow$ `Type`, le terme `(father (flat1 a f))` n'est pas égal à `s`. En effet, comme il n'y a pas la η-conversion en Coq, `Csig [x :T](f x)` n'est pas égal à `Csig f`. (Ici, l'expression "n'est pas égal" signifie qu'on ne peut pas démontrer l'égalité. La différence n'est pas non plus démontrable).

On devrait donc considérer les sigtels modulo une relation d'équivalence, qui est la relation d'équivalence associée au préordre de la relation préfixe (voir 6.2.4.b). Cela signifie que l'on ne peut pas réellement exprimer les propriétés catégoriques des CS pour les Drecords.

On montre alors que les sigtels forment un arbre avec `father` pour fonction père, et `Esig` pour racine. On peut en fait démontrer la propriété suivante qui correspond au principe de récurrence associé à `father`.

```
Lemma sigtel_father_rect:
  (P:(l:(list A))(sigtel l)->Type)
    (P ? Esig)
   ->((a:A; l:(list A); s:(sigtel (cons a l)))(P ? (father s))->(P ? s))
      ->(l:(list A); s:(sigtel l))(P ? s).
```

Niveau des impltels Pour les impltels, on imite les constructions sur les sigtels. Ainsi les constructeurs sont le impltel vide (`Eimpl`), et la fonction `box` ci-dessous qui ajoute un champ à droite d'un impltel `d`.

```
Definition box:(l:(list A); a:A; s:(sigtel l);F:(impltel s)->Type; d:(impltel s))
         (F d)->(impltel (flat1 a F)).
```

Ensuite, l'équivalent du morphisme catégorique p est donné par la fonction `proj` ci-dessous.

```
Definition proj[l:(list A)]:(a:A; s:(sigtel (cons a l)))(impltel s)->(impltel (father s)).
```

Enfin, l'équivalent de la fonction L définie ci-dessus dans l'exemple des enregistrements de Pollack est la fonction `last` ci-dessous.

```
Definition last[l:(list A)]:(a:A; s:(sigtel (cons a l)))(d:(impltel s))(app1 (proj d))
```

8.1.2.b Construction du produit fibré

On note `flat_app2id` la conversion de `(impltel (flat1 a [i :(impltel (father s))](app1 i)))` vers `(impltel s)`. De même, on note `father_flat2id` la conversion de `(impltel (father (flat1 a f)))` vers `(impltel s)`. Si on supposait la η-conversion, ces deux conversions se réduiraient sur l'identité. On a par exemple le lemme,

```
Definition flat_last: (l:(list A); a:A; s:(sigtel l); f:(impltel s)->Type;
  d:(impltel (father (flat1 a f))))
   (app1 d)===(f (father_flat2id d)).
```

Construction de "$*$" et "q" Sous les hypothèses suivantes, on peut maintenant construire les différentes composantes du pullback :

```
Variables l1,l2:(list A).
Variables a,b:A.
Variable s1:(sigtel l1).
Variable s2:(sigtel (cons a l2)).
Variable f: (impltel s1)->(impltel (father s2)).

Definition star:(sigtel (cons b l1)):=
  (flat1 b [x:(impltel s1)](app1 (f x))).

Definition q:(impltel star)->(impltel s2):=
 [d](flat_app2id (box a 4![d':(impltel (father s2))](app1 d')
        (identityT_rect ? ? [T;_]T (last d) ? (flat_last (proj d))))).
```

Sous les mêmes hypothèses, on peut montrer :

```
Lemma cone_commut: (d:(impltel (star b f)))
     (f (father_flat2id (proj d)))===(proj (q d)).
```

Construction de " $\curlywedge$ " Avec les hypothèses supplémentaires :

```
Variable l3:(list A).
Variable s3:(sigtel l3).
Variable g:(impltel s3)->(impltel s1).
Variable h_l:(d:(impltel s3))(app1 (f (g d))).
```

On peut aussi construire le morphisme " $\curlywedge$ " :

```
Definition pull: (impltel s3)->(impltel star):=
  [d](box b 4![x:(impltel s1)](app1 (f x)) (h_l d)).
```

Et on a montré en Coq, les propriétés de commutation du morphisme " $\curlywedge$ " quand il est composé avec "p" et "q" :

```
Lemma proj_circ_pull: (d:(impltel s3))(father_flat2id (proj (pull b h_l d)))===(g d).

Lemma q_circ_pull: (d:(impltel s3))(q (pull b 7!f h_l d))===
  (flat_app2id (box a 4![x:(impltel (father s2))](app1 x) (h_l d))).
```

On n'a pas démontré en Coq de propriétés correspondant à l'unicité.

Conclusion On a donc montré ici des propriétés qui correspondent approximativement aux 7 premiers axiomes des catégories de signatures d'enregistrement : les propriétés démontrées sont plus faibles que celles exigées par l'axiomatisation catégorique. Néanmoins, d'un point de vue intuitif, cela confirme l'idée que cette notion catégorique généralise les Drecords.

8.2 Propriétés de base

8.2.1 Généralisation du pullback

La notion de pullback correspond donc ici à l'ajout d'un champ dans une signature. On étend cette notion de manière à ajouter n champs.

Définition 8.2 *Si $|A| \geq n$, on notera p_A^n ou $p(A, \mathcal{F}^n(A))$ la composition des projections allant de A vers $\mathcal{F}^n(A)$.*

Plus formellement, on définit p_A^n par récurrence sur n en posant

1) pour 0 :

$$p_A^0 = id_A$$

2) pour $n+1$:

$$p_A^{n+1} = p_{\mathcal{F}^n(A)} \circ p_A^n$$

Soient deux objets A et B, et f une fonction de $\mathcal{C}[A, \mathcal{F}^n(B)]$, et S un ensemble de n étiquettes qui ne sont pas dans A. Comme pour le pullback simple, $f *^S B$ désigne la signature obtenue en ajoutant les n derniers champs de B à A, et étiquetés avec S.

Définition 8.3 *Soient A et B de $Ob(\mathcal{C})$, soit S une suite d'éléments de $\mathfrak{A}$ distincts 2 à 2 telle que $S \cap Im(\mathfrak{L}_A) = \emptyset$ et $|S| \leq B$. On pose $n = |S|$. Pour tout f de $\mathcal{C}[A, \mathcal{F}^n(B)]$, on construit un objet $f *^S B$ de $\mathcal{C}$ et un morphisme $q^S(f, B)$ de $\mathcal{C}[f *^S B, B]$ définis par une récurrence mutuelle sur n :*

1) *pour* $n = 0$

On pose

$$f *^{\emptyset} B = A \qquad et \qquad q^{\emptyset}(f, B) = f$$

$$\begin{tikzcd} f *^{\emptyset} B = A \arrow[r, "q^{\emptyset}(f,B)", "= f"'] & B \end{tikzcd}$$

2) *Sinon posons* $S = S'; a$. *Par récurrence* $q^{S'}(f, \mathcal{F}(B))$ *est un morphisme de* $\mathcal{C}[f*^{S'}\mathcal{F}(B), \mathcal{F}(B)]$,

On pose

$$f *^{S} B = q^{S'}(f, \mathcal{F}(B)) *^{a} B$$

et

$$q^{S}(f, B) = q^{a}(q^{S'}(f, \mathcal{F}(B)), B)$$

$$\begin{tikzcd} f *^{S';a} B \arrow[r, "q^{S';a}(f,B)"] \arrow[d, "p_{f*^{S';a}B}"'] & B \arrow[d, "p_B"] \\ f *^{S'} \mathcal{F}(B) \arrow[r, "q^{S'}(f,\mathcal{F}(B))"'] & \mathcal{F}(B) \end{tikzcd}$$

Lemme 8.3 *Avec les hypothèses de la définition précédente, on a que,* $\mathcal{F}^n(f *^S B) = A$ *et* $< p^n_{f*^S B}, q^S(f, B) >$ *est un cône de pullback dans* $\mathcal{C}$ *du diagramme* $< f, p^n_B >$ *:*

$$\begin{tikzcd} f *^{S} B \arrow[r, "q^{S}(f,B)"] \arrow[d, "p^n_{f*^S B}"'] & B \arrow[d, "p^n_B"] \\ A \arrow[r, "f"'] & \mathcal{F}^n(B) \end{tikzcd}$$

Preuve Par récurrence en utilisant le fait que la composition de 2 pullbacks est un pullback (voir le lemme 7.14).

$$\begin{tikzcd} f *^{S';a} B \arrow[r, "q^{S';a}(f,B)"] \arrow[d, "p_{f*^{S';a}B}"'] & B \arrow[d, "p_B"] \\ f *^{S'} B \arrow[r, "q^{S'}(f,B)"] \arrow[d, "p^{n-1}_{f*^{S'}B}"'] & \mathcal{F}(B) \arrow[d, "p^{n-1}_B"] \\ A \arrow[r, "f"'] & \mathcal{F}^{n-1}(B) \end{tikzcd}$$

□

Le lemme suivant montre que les diagrammes de pullbacks de cette opération se composent "verticalement".

Lemme 8.4 *Avec les hypothèses de la définition précédente,* $S = S_1; S_2$ *et* $m = |S|$

$$f *^{S} B = q^{S_2}(f, \mathcal{F}^m(B)) *^{S_1} B \qquad et \qquad q^{S}(f, B) = q^{S_1}(q^{S_2}(f, \mathcal{F}^m(B)), B)$$

Preuve Trivial.

□

On définit une notation dans le cas particulier où cette opération n'introduit pas de nouvelles étiquettes, mais utilise les n dernières étiquettes de B.

Définition 8.4 *Pour tout entier n, pour tout A et B de $Ob(\mathcal{C})$, tels que $n \leq |B|$ et $Im(\mathfrak{L}_A) \cap Im(\mathfrak{L}_B) = Im(\mathfrak{L}_A) \cap Im(\mathfrak{L}_{\mathcal{F}^n(B)})$, on note*

$$f *^n B = f *^{\mathfrak{L}_B(|B|-n)\ldots\mathfrak{L}_B(|B|)} B \qquad et \qquad q^n(f,B) = q^{\mathfrak{L}_B(|B|-n)\ldots\mathfrak{L}_B(|B|)}(f,B)$$

Si $n = 1$, on notera $f * B = f *^1 B$ et $q(f,B) = q^1(f,B)$.

Les deux propriétés fondamentales du pullback "simple" s'étendent au pullback généralisé. Les preuves sont sans difficultés.

Lemme 8.5 *Pour tout A et n tels que $n \leq |A|$.*

$$id_{\mathcal{F}^n(A)} *^n A = A \qquad et \qquad q^n(id_{\mathcal{F}^n(A)}, A) = id_A$$

Lemme 8.6 *Soient A, B, C, de $Ob(\mathcal{C})$. Soient S_1, S_2 deux suites de longueur $n \leq |C|$ d'éléments distincts 2 à 2 et telle que $S_1 \cap A = \emptyset$ et $S_2 \cap B = \emptyset$. Pour tout f de $\mathcal{C}[A,B]$ et g de $\mathcal{C}[B,\mathcal{F}^n(C)]$,*

$$(g \circ f) *^{S_1} C = f *^{S_1} (g *^{S_2} C) \qquad et \qquad q^{S_1}(g \circ f, C) = q^{S_2}(g,C) \circ q^{S_1}(f, g *^{S_2} C)$$

8.2.2 Foncteurs de catégories de signatures d'enregistrements

On définit maintenant la notion de foncteur de catégories de signatures d'enregistrements. Cette notion est utile pour caractériser les renommages de champs.

Définition 8.5 *Pour $i \in [1,2]$, soit $\mathfrak{C}_i =< \mathcal{C}_i, \mathtt{\mathfrak{A}}_i, \mathcal{F}_i, \mathfrak{L}_i, \mathbf{1}_i, p_i, *_i, q_i >$. Un foncteur de catégories de signatures d'enregistrements de $\mathfrak{C}_1$ vers $\mathfrak{C}_2$ est un couple $< F, L >$ où F est un foncteur de $\mathcal{C}_1$ vers $\mathcal{C}_2$ préservant la structure de signatures d'enregistrements, et L une fonction injective de $\mathtt{\mathfrak{A}}_1$ dans $\mathtt{\mathfrak{A}}_2$:*

- **1)** $F(\mathcal{F}_1(A)) = \mathcal{F}_2(F(A))$
- **2)** $F(p_1(A)) = p_2(F(A))$
- **3)** $F(f *_1^a B) = F(f) *_2^{L(a)} F(B)$
- **4)** $F(q_1^a(f,B)) = q_2^{L(a)}(F(f), F(B))$

Preuve Les conditions sur les étiquettes dans 3 et 4 sont respectées par l'injectivité de L.

□

Lemme 8.7 *Si $< F, L >$ est un tel foncteur, on a :*

- **1)** $F(\mathbf{1}_1) = \mathbf{1}_2$.
- **2)** $\mathfrak{L}_{2F(A)} = L \circ \mathfrak{L}_{1A}$

Preuve

- **1)** $\mathcal{F}(F(\mathbf{1}_1)) = F(\mathcal{F}(\mathbf{1}_1)) = F(\mathbf{1}_1)$. Par unicité de la racine, $F(\mathbf{1}_1) = \mathbf{1}_2$.
- **2)** Pour tout A de $Ob(\mathcal{C}_1)$, $A = id_{\mathbf{1}_1} *^{|A|} A$. Donc $F(A) = id_{\mathbf{1}_2} *^{L(\mathfrak{L}_{1A}(1))\ldots L(\mathfrak{L}_{1A}(|A|))} F(A)$. On en déduit que $\mathfrak{L}_{2F(A)} = L \circ \mathfrak{L}_{1A}$.

□

8.2.3 Renommage de champs

Si A est une signature et κ une bijection de l'ensemble des étiquettes, on définit ci-dessous $\mathcal{R}_\kappa(A)$ la signature obtenue après renommage des champs de A via κ. On définit aussi $\mathbb{R}_\kappa$ le morphisme associé qui convertit les enregistrements de $\mathcal{R}_\kappa(A)$ en ceux de A.

Définition 8.6 *Soit κ une bijection de $\mathfrak{A}$ dans $\mathfrak{A}$. Soit A de $Ob(\mathcal{C})$.*
On pose

$$\mathcal{R}_\kappa(A) = id_{\mathbf{1}} *^{\kappa(\mathfrak{L}_A(1))\ldots\kappa(\mathfrak{L}_A(|A|))} A$$

et

$$\mathbb{R}_\kappa(A) = q^{\kappa(\mathfrak{L}_A(1))\ldots\kappa(\mathfrak{L}_A(|A|))}(id_{\mathbf{1}}, A)$$

$$\begin{tikzcd}
\mathcal{R}_\kappa(A) \arrow[r, "\mathbb{R}_\kappa(A)"] \arrow[d, "p^{|A|}_{\mathcal{R}_\kappa(A)}"'] & A \arrow[d, "p^{|A|}_A"] \\
\mathbf{1} \arrow[r, "id_{\mathbf{1}}"'] & \mathbf{1}
\end{tikzcd}$$

Remarque 8.2 *Cette définition est correcte car pour tous les ensembles $\mathtt{E}$ et $\mathtt{F}$ inclus dans $\mathfrak{A}$ et toute bijection k de $\mathfrak{A}$ dans $\mathfrak{A}$ alors $\kappa(\mathtt{E} \cap \mathtt{F}) = \kappa(\mathtt{E}) \cap \kappa(\mathtt{F})$. En particulier, si $\mathtt{E} \cap \mathtt{F} = \emptyset$, alors $\kappa(\mathtt{E}) \cap \kappa(\mathtt{F}) = \emptyset$.*

Lemme 8.8 *Pour tout A :*

$$\mathcal{R}_{id_{\mathfrak{A}}}(A) = A \qquad et \qquad \mathbb{R}_{id_{\mathfrak{A}}}(A) = id_A$$

Lemme 8.9 *Pour tous κ_1 et κ_2 de $\mathfrak{A}$ dans $\mathfrak{A}$,*
on a

$$\mathcal{R}_{\kappa_2 \circ \kappa_1}(A) = \mathcal{R}_{\kappa_2}(\mathcal{R}_{\kappa_1}(A))$$

et

$$\mathbb{R}_{\kappa_2 \circ \kappa_1}(A) = \mathbb{R}_{\kappa_1}(A) \circ \mathbb{R}_{\kappa_2}(\mathcal{R}_{\kappa_1}(A))$$

$$\begin{tikzcd}
\mathcal{R}_{\kappa_2}(\mathcal{R}_{\kappa_1}(A)) = \mathcal{R}_{\kappa_2 \circ \kappa_1}(A) \arrow[r, "\mathbb{R}_{\kappa_2}(\mathcal{R}_{\kappa_1}(A))"] \arrow[rr, bend right, "\mathbb{R}_{\kappa_2 \circ \kappa_1}(A)"'] & \mathcal{R}_{\kappa_1}(A) \arrow[r, "\mathbb{R}_{\kappa_1}(A)"] & A
\end{tikzcd}$$

Preuve Cela découle directement des propriétés de composition du pullback.

□

Remarque 8.3 *On en déduit en particulier que :*

$$A = \mathcal{R}_{\kappa^{-1}}(\mathcal{R}_\kappa(A)) \qquad et \qquad \mathbb{R}_{\kappa^{-1}}(\mathcal{R}_\kappa(A)) = (\mathbb{R}_\kappa(A))^{-1}$$

Définition 8.7 *Soit κ bijective de $\mathfrak{A}$ dans $\mathfrak{A}$. Soient A et B de $Ob(\mathcal{C})$, soit f de $\mathcal{C}[A, B]$.*
On pose

$$\mathcal{R}_\kappa(f) = (\mathbb{R}_\kappa(B))^{-1} \circ f \circ \mathbb{R}_\kappa(A)$$

$$\begin{tikzcd}
\mathcal{R}_\kappa(A) \arrow[r, "\mathbb{R}_\kappa(A)"] \arrow[d, "\mathcal{R}_\kappa(f)"'] & A \arrow[d, "f"] \\
\mathcal{R}_\kappa(B) \arrow[r, bend right, "\mathbb{R}_\kappa(B)"'] & B \arrow[l, bend right, "\mathbb{R}_{\kappa^{-1}}(\mathcal{R}_\kappa(B))"']
\end{tikzcd}$$

C'est un morphisme de $\mathcal{C}[\mathcal{R}_\kappa(A), \mathcal{R}_\kappa(B)]$

Théorème 8.1 *Pour toute bijection κ, $< \mathcal{R}_\kappa, \kappa >$ définit un isofoncteur de catégorie de signatures d'enregistrements de $\mathfrak{C}$ dans elle-même. De plus, $\mathbb{R}_\kappa$ est une transformation naturelle de $\mathcal{R}_\kappa$ vers $Id_{\mathcal{C}}$, elle-même iso (voir figure de la définition 8.7).*

Preuve On montre facilement les points suivants :

- **1)** $\mathcal{R}_\kappa(id_A) = id_{\mathcal{R}_\kappa(A)}$.
- **2)** $\mathcal{R}_\kappa(g \circ f) = \mathcal{R}_\kappa(g) \circ \mathcal{R}_\kappa(f)$.
- **3)** $\mathcal{R}_\kappa(\mathcal{F}(A)) = \mathcal{F}(\mathcal{R}_\kappa(A))$.
- **4)** $\mathcal{R}_\kappa(p(A)) = p(\mathcal{R}_\kappa(A))$.
- **5)** $\mathcal{R}_\kappa(f *^a B) = \mathcal{R}_\kappa(f) *^{\kappa(a)} \mathcal{R}_\kappa(B)$
- **6)** $\mathcal{R}_\kappa(q^a(f, B)) = q^{k(a)}(\mathcal{R}_\kappa(f), \mathcal{R}_\kappa(B))$.
- **7)** $\mathcal{R}_\kappa^{-1} = \mathcal{R}_{\kappa^{-1}}$.
- **8)** $\mathbb{R}_\kappa^{-1}(A) = \mathbb{R}_{\kappa^{-1}}(\mathcal{R}_k(A))$.

□

8.2.4 Régularité

Définition 8.8 *On dit que $\mathfrak{C}$ est une catégorie de signatures d'enregistrements régulière sur les signatures, ssi pour tous A, B, C de $Ob(\mathcal{C})$, pour tout n tel que*

$$\mathcal{F}(A) = \mathcal{F}^n(B) = \mathcal{F}^n(C) \quad et \quad Im(\mathfrak{L}_B) \cap Im(\mathfrak{L}_A) = Im(\mathfrak{L}_{\mathcal{F}(A)}) \quad et \quad Im(\mathfrak{L}_C) \cap Im(\mathfrak{L}_A) = Im(\mathfrak{L}_{\mathcal{F}(A)})$$

$$\begin{tikzcd} p_A *^n B = p_A *^n C \arrow[r] \arrow[rr, bend left] \arrow[d, "p^n_{p_A *^n C}"'] & B \arrow[dr, "p^n_B"'] & C \arrow[d, "p^n_C"] \\ A \arrow[rr, "p_A"'] & & \mathcal{F}(A) = \mathcal{F}^n(B) = \mathcal{F}^n(C) \end{tikzcd}$$

$p_A *^n B = p_A *^n C$ implique $B = C$.

Définition 8.9 *On dit que $\mathfrak{C}$ est une catégorie de signatures d'enregistrements régulière sur les morphismes, ssi pour tous A, B de de $Ob(\mathcal{C})$, pour tout n tel que $\mathcal{F}(A) = \mathcal{F}^n(B)$ et $Im(\mathfrak{L}_B) \cap Im(\mathfrak{L}_A) = Im(\mathfrak{L}_{\mathcal{F}(A)})$, $q^n(p_A, B)$ est un épimorphisme :*

Pour tout C, f et g de $\mathcal{C}[B,C]$,
$f \circ q^n(p_A, B) = g \circ q^n(p_A, B)$
implique $f = g$.

$$\begin{tikzcd} p_A *^n B \arrow[r, "q^n(p_A,B)"] & B \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & C \end{tikzcd}$$

Remarque 8.4 *Dans les D-systèmes, ces 2 notions correspondent respectivement à la régularité sur les contextes et les substitutions (voir 7.2.6).*

Dans un système d'enregistrement à la Pollack (avec η-conversion), il n'est pas très difficile de montrer que ces 2 propriétés sont vérifiées.

Définition 8.10 *On dira que $\mathfrak{C}$ est régulière si elle est régulière sur les signatures et les morphismes. La remarque qui précède montre que les catégories de signatures "raisonnables" sont régulières.*

Définition 8.11 *On dit qu'un morphisme f est régulier, si f est un épimorphisme, et si pour tout A et B tels que $\mathfrak{L}_A(|A|) \notin Im(\mathfrak{L}_{Dom(f)})$, $\mathfrak{L}_B(|B|) \notin Im(\mathfrak{L}_{Dom(f)})$ et $\mathcal{F}(A) = \mathcal{F}(B) = Im(f)$, $f * A = f * B$ implique $A = B$.*

Définition 8.12 *On dit qu'un morphisme f de $\mathcal{C}$ est ultra-régulier ssi pour tous A et tout n tels que $\mathcal{F}^n(A) = Im(f)$ et $Im(\mathfrak{L}_A) \cap Im(\mathfrak{L}_{Dom(f)}) = Im(\mathfrak{L}_{\mathcal{F}^n(A)}) \cap Im(\mathfrak{L}_{Dom(f)})$, alors $q^n(f, A)$ est régulier.*

Lemme 8.10 *Soit f un morphisme de $\mathcal{C}$ ultra-régulier, alors pour tout A tel que $\mathfrak{L}_A(|A|) \notin Dom(f)$, alors $q(f, A)$ est ultra-régulier.*

Preuve Trivial, il suffit de remarquer que sous l'hypothèse $\mathcal{F}^n(B) = A$, et $Im(\mathfrak{L}_B) \cap Im(\mathfrak{L}_{Dom(f)*A}) = Im(\mathfrak{L}_{\mathcal{F}^n(B)}) \cap Im(\mathfrak{L}_{Dom(f)*A})$, alors $q^n(q(f, A), B) = q^{n+1}(f, B)$.

□

Remarque 8.5 *Dans toute catégorie de signatures d'enregistrements $\mathfrak{C}$, pour tout objet A, id_A est ultra-régulier.*

$\mathfrak{C}$ est régulière si pour tout A, p_A est ultra-régulier.

8.3 Oubli de champs

On modélise maintenant l'oubli de champs sur les interfaces, et la conversion associée au niveau des collections. Cette relation d'oubli a lieu **sans permutation** de champs et **sans renommage** d'étiquettes : étant donné 2 signatures, l'unicité des étiquettes garantit alors qu'il n'y a qu'une façon d'oublier. On considèrera les permutations dans la sous-section 8.4.

On définit ainsi ci-dessous la relation $A \lhd B$ qui indique qu'on peut passer d'une signature B à une signature A en "oubliant" des champs de B. Pour regarder si $A \lhd B$, on s'appuie sur les étiquettes. Ainsi si le dernier champ de A a la même étiquette que celui de B, alors ces champs "représentent" le même champ : on vérifie que $\mathcal{F}(A) \lhd \mathcal{F}(B)$ puis que B étend $\mathcal{F}(B)$ en lui ajoutant le dernier champ de A. Pour exprimer cette "extension", il faut en fait définir la conversion des enregistrements de signature B sur ceux de signature A (notée $\mathbb{C}(B, A)$) en même temps que la relation d'oubli.

Sinon, si les derniers champs de A et B ont des étiquettes distinctes, comme B doit avoir au moins tous les champs de A et dans le même ordre, cela signifie que le dernier champ de B ne figure pas dans A : on vérifie que $A \lhd \mathcal{F}(B)$.

Définition 8.13 *On définit $A \lhd B$ et $\mathbb{C}(B, A)$ par induction mutuelle sur $|B|$.*

1) *Si $B = \mathbf{1}$ alors $A \lhd B$ ssi $A = \mathbf{1}$, et dans ce cas, $\mathbb{C}(B, A) = id_{\mathbf{1}}$.*

2) *Si $A \neq \mathbf{1}$ et $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$,*

$$A \lhd B \ \ ssi \ \left\{ \begin{array}{c} \mathcal{F}(A) \lhd \mathcal{F}(B) \\ et \\ B = \mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)) * A \end{array} \right.$$

$$\begin{tikzcd} B \arrow[r, "\mathbb{C}(B,A)"] \arrow[d, "p_B"'] & A \arrow[d, "p_A"] \\ \mathcal{F}(B) \arrow[r, "\mathbb{C}(\mathcal{F}(B),\mathcal{F}(A))"'] & \mathcal{F}(A) \end{tikzcd}$$

Dans ce cas, on pose

$$\mathbb{C}(B, A) = q(\mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)), A)$$

3) *Sinon,*

$$A \lhd B \ \ ssi \ \ A \lhd \mathcal{F}(B)$$

Dans ce cas, on pose

$$\mathbb{C}(B, A) = \mathbb{C}(\mathcal{F}(B), A) \circ p_B$$

$$\begin{tikzcd} B \arrow[d, "p_B"'] \arrow[dr, "\mathbb{C}(B,A)"] & \\ \mathcal{F}(B) \arrow[r, "\mathbb{C}(\mathcal{F}(B),A)"'] & A \end{tikzcd}$$

8.3.1 Propriétés élémentaires de l'oubli

La propriété suivante montre que l'oubli préserve l'ordre des étiquettes dans les signatures. Pour toute signature A, si a est une étiquette de $Im(\mathfrak{L}_A)$, alors $\mathfrak{L}_A^{-1}(a)$ désigne la position de a dans cette signature.

Lemme 8.11 *Supposons $A \lhd B$. Alors, $Im(\mathfrak{L}_A) \subseteq Im(\mathfrak{L}_B)$. De plus, pour tous a et b de $Im(\mathfrak{L}_A)$, $\mathfrak{L}_A^{-1}(a) < \mathfrak{L}_A^{-1}(b)$ équivaut à $\mathfrak{L}_B^{-1}(a) < \mathfrak{L}_B^{-1}(b)$.*

Preuve Par induction sur $|B|$, puis pas cas.

1) Si $B = \mathbf{1}$, $A = \mathbf{1}$ alors évident.

2) Si $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$. On a $\mathcal{F}(A) \lhd \mathcal{F}(B)$ et $B = \mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$. On pose $x = \mathfrak{L}_B(|B|)$.

On a : $Im(\mathfrak{L}_A) = Im(\mathfrak{L}_{\mathcal{F}(A)}) \cup \{x\}$ et $Im(\mathfrak{L}_B) = Im(\mathfrak{L}_{\mathcal{F}(B)}) \cup \{x\}$.
Par hypothèse de récurrence, $Im(\mathfrak{L}_{\mathcal{F}(A)}) \subseteq Im(\mathfrak{L}_{\mathcal{F}(B)})$, donc $Im(\mathfrak{L}_A) \subseteq Im(\mathfrak{L}_B)$.
De plus, par hypothèse de récurrence, pour tout a et b de $Im(\mathfrak{L}_{\mathcal{F}(A)})$, $\mathfrak{L}^{-1}_{\mathcal{F}(A)}(a) < \mathfrak{L}^{-1}_{\mathcal{F}(A)}(b)$ équivaut à $\mathfrak{L}^{-1}_{\mathcal{F}(B)}(a) < \mathfrak{L}^{-1}_{\mathcal{F}(B)}(b)$. Or pour tout a de $Im(\mathfrak{L}_A)$, si a est différent de x, on a $\mathfrak{L}^{-1}_A(a) = \mathfrak{L}^{-1}_{\mathcal{F}(A)}(a)$, $\mathfrak{L}^{-1}_B(a) = \mathfrak{L}^{-1}_{\mathcal{F}(B)}(a)$, $\mathfrak{L}^{-1}_A(a) < |A| = \mathfrak{L}^{-1}_A(x)$, $\mathfrak{L}^{-1}_B(a) < |B| = \mathfrak{L}^{-1}_B(x)$. On en déduit la propriété recherchée.

3) Sinon, $A \lhd \mathcal{F}(B)$. Par hypothèse de récurrence $Im(\mathfrak{L}_A) \subseteq Im(\mathfrak{L}_{\mathcal{F}(B)})$. Or $Im(\mathfrak{L}_{\mathcal{F}(B)}) \subseteq Im(\mathfrak{L}_B)$, donc $Im(\mathfrak{L}_A) \subseteq Im(\mathfrak{L}_A)$.
De plus, par hypothèse de récurrence, pour tout a et b de $Im(\mathfrak{L}_A)$, $\mathfrak{L}^{-1}_A(a) < \mathfrak{L}^{-1}_A(b)$ équivaut à $\mathfrak{L}^{-1}_{\mathcal{F}(B)}(a) < \mathfrak{L}^{-1}_{\mathcal{F}(B)}(b)$. Or $a \in Im(\mathfrak{L}_A)$ implique $\mathfrak{L}^{-1}_B(a) = \mathfrak{L}^{-1}_{\mathcal{F}(B)}(a)$. On en déduit la propriété recherchée.

□

Remarque 8.6 *On déduit en particulier du lemme précédent que $A \lhd B$ implique $|A| \leq |B|$.*

Les trois lemmes suivants montrent que $\lhd$ est une relation d'ordre.

Lemme 8.12 *$A \lhd A$ et $\mathbb{C}(A, A) = id_A$.*

Preuve Trivial par induction sur $|A|$.

□

Lemme 8.13 *$A \lhd B$ et $Im(\mathfrak{L}_A) = Im(\mathfrak{L}_B)$ implique $A = B$.*

Preuve Par induction sur B.

1) Si $B = \mathbf{1}$, $A = \mathbf{1}$ alors évident.

2) Sinon, comme $Im(\mathfrak{L}_A) = Im(\mathfrak{L}_B)$, on a : $Im(\mathfrak{L}_{\mathcal{F}(A)}) = Im(\mathfrak{L}_{\mathcal{F}(B)})$ et $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$. Donc $\mathcal{F}(A) \lhd \mathcal{F}(B)$ et $B = \mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$.
Par hypothèse de récurrence, on en déduit que $\mathcal{F}(A) = \mathcal{F}(B)$. Et par le lemme 8.12, on peut en conclure que $\mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)) = id_{\mathcal{F}(A)}$.
Donc $B = id_{\mathcal{F}(A)} * A$, et par l'axiome 8, $B = A$.

□

Lemme 8.14 *$A \lhd B$ et $B \lhd C$ implique $A \lhd C$ et $\mathbb{C}(C, A) = \mathbb{C}(B, A) \circ \mathbb{C}(C, B)$.*

Preuve Par induction sur $|C|$, puis par cas.

1) Si $C = \mathbf{1}$ alors $B = \mathbf{1}$ et $A = \mathbf{1}$.

2) Si $\mathfrak{L}_C(|C|) = \mathfrak{L}_B(|B|)$. On a $\mathcal{F}(B) \lhd \mathcal{F}(C)$ et $C = \mathbb{C}(\mathcal{F}(C), \mathcal{F}(B)) * B$.

2.1) Si $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$, on a $\mathcal{F}(A) \lhd \mathcal{F}(B)$, et $B = \mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$.
Donc $\mathfrak{L}_C(|C|) = \mathfrak{L}_A(|A|)$. Par hypothèse de récurrence, $\mathcal{F}(A) \lhd \mathcal{F}(C)$ et $\mathbb{C}(\mathcal{F}(C), \mathcal{F}(A)) = \mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)) \circ \mathbb{C}(\mathcal{F}(C), \mathcal{F}(B))$.
On a bien $C = \mathbb{C}(\mathcal{F}(C), \mathcal{F}(A)) * A$ et $\mathbb{C}(C, A) = \mathbb{C}(B, A) \circ \mathbb{C}(C, B)$.

2.2) Sinon, $\mathfrak{L}_B(|B|) \neq \mathfrak{L}_A(|A|)$ et $A \lhd \mathcal{F}(B)$.
On a : $\mathfrak{L}_C(|C|) \neq \mathfrak{L}_A(|A|)$. Et par hypothèse de récurrence, $A \lhd \mathcal{F}(C)$ et $\mathbb{C}(\mathcal{F}(C), A) = \mathbb{C}(\mathcal{F}(B), A) \circ \mathbb{C}(\mathcal{F}(C), \mathcal{F}(B))$.
On a bien $A \lhd C$ et

$$\begin{aligned}
&\mathbb{C}(C, A) \\
&\quad = \mathbb{C}(\mathcal{F}(C), A) \circ p_C \\
&\quad = \mathbb{C}(\mathcal{F}(B), A) \circ (\mathbb{C}(\mathcal{F}(C), \mathcal{F}(B)) \circ p_C) \\
&\quad = \mathbb{C}(\mathcal{F}(B), A) \circ p_B \circ q(\mathbb{C}(\mathcal{F}(C), \mathcal{F}(B)), B) \\
&\quad = \mathbb{C}(B, A) \circ \mathbb{C}(C, B)
\end{aligned}$$

3) Sinon $B \lhd \mathcal{F}(C)$ et $A \lhd B$.
Comme $Im(\mathfrak{L}_A) \subseteq Im(\mathfrak{L}_B) \subseteq Im(\mathfrak{L}_{\mathcal{F}(C)})$, $\mathfrak{L}_C(|C|) \notin Im(\mathfrak{L}_A)$.
Par hypothèse de récurrence, $A \lhd \mathcal{F}(C)$ et $\mathbb{C}(\mathcal{F}(C), A) = \mathbb{C}(B, A) \circ \mathbb{C}(\mathcal{F}(C), B)$.
On a bien $A \lhd C$ et $\mathbb{C}(C, A) = \mathbb{C}(B, A) \circ \mathbb{C}(\mathcal{F}(C), B) \circ p_C = \mathbb{C}(B, A) \circ \mathbb{C}(C, B)$.

□

Théorème 8.2 *$\lhd$ est une relation d'ordre.*

Preuve Pour l'antisymétrie, il suffit de voir que $A \lhd B$ et $B \lhd A$ implique $Im(\mathfrak{L}_A) = Im(\mathfrak{L}_B)$. Et que $A \lhd B$ et $Im(\mathfrak{L}_A) = Im(\mathfrak{L}_B)$ implique $A = B$.

□

Lemme 8.15 *Si $A \lhd B$, alors pour tout n, $\mathcal{F}^n(A) \lhd \mathcal{F}^n(B)$.*

Preuve Par induction sur $|B|$ (pour tout n), puis pas cas.

1) Si $B = \mathbf{1}$, $A = \mathbf{1}$ alors évident.

2) Si $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$. On a $\mathcal{F}(A) \lhd \mathcal{F}(B)$,
Si $n = 0$ alors évident.
Sinon par hypothèse de récurrence, $\mathcal{F}^{n-1}(\mathcal{F}(A)) \lhd \mathcal{F}^{n-1}(\mathcal{F}(B))$, donc $\mathcal{F}^n(A) \lhd \mathcal{F}^n(B)$.

3) Sinon, $A \lhd \mathcal{F}(B)$. Or $\mathcal{F}(A) \lhd A$. Par transitivité, $\mathcal{F}(A) \lhd \mathcal{F}(B)$. On conclut comme pour le cas précédent.

□

Définition 8.14 *On définit $\mathfrak{L}[A, B]$ comme l'ensemble des étiquettes de B qui ne sont pas dans A. Autrement dit, on pose $\mathfrak{L}[A, B] = Im(\mathfrak{L}_B) \backslash Im(\mathfrak{L}_A)$.*

Le lemme ci-dessous montre que si $A \lhd B$ et si A et B portent les mêmes étiquettes sur les n derniers champs, alors ces n derniers champs sont en correspondance (c'est-à-dire identiques intuitivement). Plus généralement, $\lhd$ correspond à l'intuition : sur A et B les champs de même étiquettes sont en correspondance, c'est-à-dire $B = \mathbb{C}(\mathcal{F}^n(B), \mathcal{F}^n(A)) *^n A$.

Lemme 8.16 *Si $A \lhd B$ et $n = \min(\{|B| - \mathfrak{L}_B^{-1}(x) \ / \ x \in \mathfrak{L}[A, B]\} \cup \{|B|\})$ alors $\mathcal{F}^n(A) \lhd \mathcal{F}^n(B)$, $B = \mathbb{C}(\mathcal{F}^n(B), \mathcal{F}^n(A)) *^n A$, et $\mathbb{C}(B, A) = q^n(\mathbb{C}(\mathcal{F}^n(B), \mathcal{F}^n(A)), A)$*

Preuve Si $\mathfrak{L}[A, B] = \emptyset$, alors par le lemme 8.13 $A = B$, et $n = |B|$, donc c'est vrai.
Sinon, on supposera en $\mathfrak{L}[A, B] \neq \emptyset$, et on montre le lemme avec cette nouvelle hypothèse par induction sur B.

1) Si $B = \mathbf{1}$, $A = \mathbf{1}$ alors évident.

2) Si $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$. On a $\mathcal{F}(A) \lhd \mathcal{F}(B)$ et $B = \mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$. On pose $y = \mathfrak{L}_B(|B|)$.
On a $Im(\mathfrak{L}_A) = Im(\mathfrak{L}_{\mathcal{F}(A)}) \cup \{y\}$ et $Im(\mathfrak{L}_B) = Im(\mathfrak{L}_{\mathcal{F}(B)}) \cup \{y\}$. Donc $\mathfrak{L}[A, B] = \mathfrak{L}[\mathcal{F}(A), \mathcal{F}(B)]$. Comme pour $x \in Im(\mathfrak{L}_{\mathcal{F}(B)})$, $\mathfrak{L}^{-1}_{\mathcal{F}(B)}(x) = \mathfrak{L}^{-1}_B(x)$, on a $n = \min(\{|\mathcal{F}(B)| + 1 - \mathfrak{L}^{-1}_{\mathcal{F}(B)}(x) \ / \ x \in \mathfrak{L}[\mathcal{F}(A), \mathcal{F}(B)]\})$.
Comme $\mathfrak{L}[\mathcal{F}(A), \mathcal{F}(B)] \neq \emptyset$, en appliquant l'hypothèse de récurrence, on trouve que $\mathcal{F}^{n-1}(\mathcal{F}(A)) \lhd \mathcal{F}^{n-1}(\mathcal{F}(B))$, $\mathcal{F}(B) = \mathbb{C}(\mathcal{F}^{n-1}(\mathcal{F}(B)), \mathcal{F}^{n-1}(\mathcal{F}(A))) *^{n-1} \mathcal{F}(A)$, et $\mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)) = q^{n-1}(\mathbb{C}(\mathcal{F}^n(B), \mathcal{F}^n(A)), \mathcal{F}(A))$. On a donc bien $\mathcal{F}^n(A) \lhd \mathcal{F}^n(B)$, $B = \mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)) * A = q^{n-1}(\mathbb{C}(\mathcal{F}^n(B), \mathcal{F}^n(A)), \mathcal{F}(A)) * A = \mathbb{C}(\mathcal{F}^n(B), \mathcal{F}^n(A)) *^n A$, et $\mathbb{C}(B, A) = q(q^{n-1}(\mathbb{C}(\mathcal{F}^n(B), \mathcal{F}^n(A)), \mathcal{F}(A)), A) = q^n(\mathbb{C}(\mathcal{F}^n(B), \mathcal{F}^n(A)), A)$.

3) Sinon, $\mathfrak{L}_B(|B|) \neq \mathfrak{L}_A(|A|)$. Dans ce cas, $n = 0$, et le lemme est trivialement vrai.

□

8.3.2 Oubli et régularité

On étudie maintenant les propriétés de l'oubli lorsque $\mathfrak{C}$ est régulière sur les signatures. Le lemme 8.18 ci-dessous montre que dans ce cas, lorsque l'oubli est possible, il est uniquement déterminé par l'oubli sur les étiquettes. Plus précisément, si $A \lhd C$ alors pour tout B qui a les mêmes étiquettes que A et tel que $B \lhd C$, on a $B = A$ (il suffit d'appliquer les lemmes 8.18 et 8.13).

Lemme 8.17 *Soit $\mathfrak{C}$ régulière sur les signatures. Soient A, B et C trois objets. Soit n un entier tel que $\mathcal{F}^n(B) = \mathcal{F}^n(C)$. On pose $D = \mathcal{F}^n(B) = \mathcal{F}^n(C)$. Si $D \lhd A$, $Im(\mathfrak{L}_B) \cap Im(\mathfrak{L}_A) = Im(\mathfrak{L}_D)$, et $Im(\mathfrak{L}_C) \cap Im(\mathfrak{L}_A) = Im(\mathfrak{L}_D)$, alors $\mathbb{C}(A, D) *^n B = \mathbb{C}(A, D) *^n C$ implique $B = C$.*

Preuve Par induction sur $|A|$, à B et C fixé (pour tout n).

1) Si $A = \mathbf{1}$ alors $D = \mathbf{1}$ et $id_{\mathbf{1}} *^n B = id_{\mathbf{1}} *^n C$ implique $B = C$.

2) Si $\mathfrak{L}_A(|A|) = \mathfrak{L}_D(|D|)$. On a $\mathcal{F}(D) \lhd \mathcal{F}(A)$, $A = \mathbb{C}(\mathcal{F}(A), \mathcal{F}(D)) * D$, $\mathbb{C}(A, D) = q(\mathbb{C}(\mathcal{F}(A), \mathcal{F}(D)), D)$.
Donc $\mathbb{C}(A, D) *^n B = \mathbb{C}(\mathcal{F}(A), \mathcal{F}(D)) *^{n+1} B$. Par hypothèse de récurrence, $\mathbb{C}(\mathcal{F}(A), \mathcal{F}(D)) *^{n+1} B = \mathbb{C}(\mathcal{F}(A), \mathcal{F}(D)) *^{n+1} C$ implique $B = C$.

3) Sinon, $D \lhd \mathcal{F}(A)$, et $\mathbb{C}(A, D) = \mathbb{C}(\mathcal{F}(A), D) \circ p_A$.
On a : $\mathbb{C}(A, D) *^n B = (\mathbb{C}(\mathcal{F}(A), D) \circ p_A) *^n B = p_A *^n (\mathbb{C}(\mathcal{F}(A), D) *^n B)$.
Donc par régularité sur de p_A, $\mathbb{C}(A, D) *^n B = \mathbb{C}(A, D) *^n C$ implique $\mathbb{C}(\mathcal{F}(A), D) *^n B = \mathbb{C}(\mathcal{F}(A), D) *^n C$.
Par hypothèse de récurrence, cela implique $B = C$.

□

Lemme 8.18 *Soit $\mathfrak{C}$ régulière sur les signatures. Soient A, B et C. Si $A \lhd C$, $B \lhd C$, et $Im(\mathfrak{L}(A)) \subset Im(\mathfrak{L}(B))$ alors $A \lhd B$.*

Preuve Par induction sur $|C|$.

1) Si $C = \mathbf{1}$ alors $B = \mathbf{1}$ et $A = \mathbf{1}$. Evident.

2) Si $\mathfrak{L}_C(|C|) = \mathfrak{L}_B(|B|)$. On a $\mathcal{F}(B) \lhd \mathcal{F}(C)$.

2.1) Si $\mathfrak{L}_C(|C|) = \mathfrak{L}_A(|A|)$. On a $\mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$ et $\mathcal{F}(A) \lhd \mathcal{F}(C)$. Comme, $Im(\mathfrak{L}(\mathcal{F}(A))) \subset Im(\mathfrak{L}(\mathcal{F}(B)))$, par hypothèse de récurrence, $\mathcal{F}(A) \lhd \mathcal{F}(B)$.
$C = \mathbb{C}(\mathcal{F}(C), \mathcal{F}(B)) * (\mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)) * A)$. D'après le lemme précédent, $B = \mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$.
On en déduit $A \lhd B$.

2.2) Sinon, $\mathfrak{L}_C(|C|) \notin Im(\mathfrak{L}_A)$. $A \lhd \mathcal{F}(C)$ et $\mathcal{F}(B) \lhd \mathcal{F}(C)$. Comme, $Im(\mathfrak{L}(A)) \subset Im(\mathfrak{L}(\mathcal{F}(B)))$, par hypothèse de récurrence, $A \lhd \mathcal{F}(B)$.
On en déduit $A \lhd B$.

3) Sinon, $A \lhd \mathcal{F}(C)$ et $B \lhd \mathcal{F}(C)$. Par hypothèse de récurrence, $A \lhd B$.

□

8.4 Permutations de champs

On définit ci-dessous la relation $A \sim_n B$ qui indique que l'on passe de A à B (ou de B à A) en permutant le champ n avec le champ $n+1$. Le point 1 ci-dessous dit que avant le $n+2$-ième dernier champ, A et B sont égaux. Le point 2 dit que $\mathcal{F}^n(A)$ s'obtient à partir de $\mathcal{F}^{n+1}(A)$ en ajoutant le $n+1$-ième dernier champs de B. Le point 3 dit que réciproquement $\mathcal{F}^n(B)$ s'obtient à partir de $\mathcal{F}^{n+1}(B)$ en ajoutant le $n+1$-ième dernier champs de A. Ces deux points permettent de construire le morphisme $q(p_{\mathcal{F}^{n+1}(A)}, \mathcal{F}^{n+1}(B)) \curlywedge_{\mathcal{F}^n(B)} p_{\mathcal{F}^n(A)}$ de $\mathcal{F}^n(A)$ vers $\mathcal{F}^n(B)$, qui permute le dernier champ d'un enregistrement de signature $\mathcal{F}^n(A)$ avec l'avant-dernier champ, pour donner un enregistrement de signature $\mathcal{F}^n(B)$. Le point 4 affirme que les n derniers de A champs sont en correspondance avec ceux de B vis-à-vis de ce morphisme (voir figure page 169).

Définition 8.15 (Transposition) *On dit que $A \sim_n B$ ssi $|A| \geq n+2$ et*

1) $\mathcal{F}^{n+2}(A) = \mathcal{F}^{n+2}(B)$

2) $\mathcal{F}^n(A) = p_{\mathcal{F}^{n+1}(A)} * \mathcal{F}^{n+1}(B)$

3) $\mathcal{F}^n(B) = p_{\mathcal{F}^{n+1}(B)} * \mathcal{F}^{n+1}(A)$

4) $A = (q(p_{\mathcal{F}^{n+1}(A)}, \mathcal{F}^{n+1}(B)) \curlywedge_{\mathcal{F}^n(B)} p_{\mathcal{F}^n(A)}) *^n B$

Preuve Cette définition est valide, car $\mathcal{F}^n(A) = p_{\mathcal{F}^{n+1}(A)} * \mathcal{F}^{n+1}(B)$ implique $p_{\mathcal{F}^{n+1}(B)} \circ q(p_{\mathcal{F}^{n+1}(A)}, \mathcal{F}^{n+1}(B)) = p_{\mathcal{F}^{n+1}(A)} \circ p_{\mathcal{F}^n(A)}$. Donc $q(p_{\mathcal{F}^{n+1}(A)}, \mathcal{F}^{n+1}(B)) \curlywedge_{\mathcal{F}^n(B)} p_{\mathcal{F}^n(A)}$ existe !!

□

Lemme 8.19 *Avec les mêmes hypothèses, $|A| = |B|$. On pose $m = |A| - n$. Pour tout k, si $k \notin \{m, m-1\}$ alors $\mathfrak{L}_B(k) = \mathfrak{L}_A(k)$; sinon $\mathfrak{L}_B(m-1) = a$ et $\mathfrak{L}_B(m) = \mathfrak{L}_A(m-1)$.*

Remarque 8.7 *On en tire que si $A \sim_n B$ et $A \sim_m B$ alors $n = m$.*

Lemme 8.20 *Avec les mêmes hypothèses, $A \sim_n B$ implique $B \sim_n A$.*

Preuve Supposons $A \sim_n B$.
En applicant le point 4 de la définition, puis le lemme 8.6 (page 161) on trouve :

$$\begin{aligned}
&(q(p_{\mathcal{F}^{n+1}(B)}, \mathcal{F}^{n+1}(A)) \curlywedge_{\mathcal{F}^n(A)} p_{\mathcal{F}^n(B)}) *^n A \\
&\quad = ((q(p_{\mathcal{F}^{n+1}(A)}, \mathcal{F}^{n+1}(B)) \curlywedge_{\mathcal{F}^n(B)} p_{\mathcal{F}^n(A)}) \circ (q(p_{\mathcal{F}^{n+1}(B)}, \mathcal{F}^{n+1}(A)) \curlywedge_{\mathcal{F}^n(A)} p_{\mathcal{F}^n(B)})) *^n B \\
&\quad \text{ puis en appliquant le lemme 8.2 (page 153), on obtient :} \\
&\quad = ((id_{\mathcal{F}^{n+1}(B)} \circ p_{\mathcal{F}^n(B)}) \curlywedge_{\mathcal{F}^n(B)} (id_{\mathcal{F}^{n+1}(A)} \circ q(p_{\mathcal{F}^{n+1}(B)}, \mathcal{F}^{n+1}(A)))) *^n B \\
&\quad = id_{\mathcal{F}^n(B)} *^n B \\
&\quad = B
\end{aligned}$$

□

$$\begin{tikzcd}
A \arrow[rr] \arrow[d, "p^n_A"'] & & B \arrow[d, "p^n_B"] & \\
\mathcal{F}^n(A) \arrow[rr, dotted] \arrow[rrr, bend left, "p_{\mathcal{F}^n(A)}"] \arrow[d, "p_{\mathcal{F}^n(A)}"'] \arrow[drr, "q(p_{\mathcal{F}^{n+1}(A)}, \mathcal{F}^{n+1}(B))"'] & & \begin{matrix} p_{\mathcal{F}^{n+1}(B)} * \mathcal{F}^{n+1}(A) \\ = \mathcal{F}^n(B) \end{matrix} \arrow[r] \arrow[d, "p_{\mathcal{F}^n(B)}"] & \mathcal{F}^{n+1}(A) \arrow[d, "p_{\mathcal{F}^{n+1}(A)}"] \\
\mathcal{F}^{n+1}(A) \arrow[drr, bend right, "p_{\mathcal{F}^{n+1}(A)}"'] & & \mathcal{F}^{n+1}(B) \arrow[r, "p_{\mathcal{F}^{n+1}(B)}"] \arrow[d, "p_{\mathcal{F}^{n+1}(B)}"'] & \mathcal{F}^{n+2}(A) \arrow[dl, equal] \\
 & & \mathcal{F}^{n+2}(B) &
\end{tikzcd}$$

FIG. 8.1 – Définition de $A \sim_n B$

Définition 8.16 (Conversion associée aux transpositions) *Si $A \sim_n B$ alors on définit le morphisme $\mathbb{I}(A, B)$ de $\mathcal{C}[A, B]$, qui est la conversion associée à cette permutation. On pose :*

$$\mathbb{I}(A, B) = q^n(q(p_{\mathcal{F}^{n+1}(A)}, \mathcal{F}^{n+1}(B)) \curlywedge_{\mathcal{F}^n(B)} p_{\mathcal{F}^n(A)}, B)$$

Lemme 8.21 $\mathbb{I}(A, B) \circ \mathbb{I}(B, A) = id_B$.

Preuve Même preuve que pour le lemme 8.20, en utilisant la composition de q.

□

Définition 8.17 (Suite de transpositions) *Soit $S = (A_i)_{i \in [1,n]}$ une suite d'objets de $\mathcal{C}$. Soit A_0 un objet de $\mathcal{C}$. On dit que S est une suite de transpositions sur A_0 ssi pour tout $i \in [1, n]$:*

1) $|A_0| = |A_i|$.

2) *il existe un (unique) m_i tel que $\mathfrak{L}_{A_{i-1}}(m_i) = \mathfrak{L}_{A_i}(m_i - 1)$, $\mathfrak{L}_{A_{i-1}}(m_i - 1) = \mathfrak{L}_{A_i}(m_i)$ et pour tout $k \notin \{m_i, m_i - 1\}$ $\mathfrak{L}_{A_{i-1}}(k) = \mathfrak{L}_{A_i}(k)$.*

3) $A_{i-1} \sim_{|A_0| - m_i} A_i$.

Si $n = 0$ on note $\emptyset_{A_0}$ la suite de transposition sur A_0.

Remarque 8.8 *Si $S = (A_i)_{i \in [1,n]}$ et $S' = (B_i)_{i \in [1,m]}$ sont deux suites de transpositions sur A et B respectivement, telles que $B = A_n$ alors $S; S'$ est une suite de transpositions.*

Définition 8.18 *Si $S = (A_i)_{i \in [1,n]}$ est une suite de transpositions sur A_0, on pose $S^{-1} = (A_{n-i})_{i \in [1,n]}$. C'est une suite de transpositions sur A_n.*

Définition 8.19 (Conversion associée aux suites de transpositions) *Si $S = (A_i)_{i \in [1,n]}$ est une suite de transpositions sur A_0, on pose*

$$\mathbb{I}(S) = \mathbb{I}(A_{n-1}, A_n) \circ \ldots \circ \mathbb{I}(A_1, A_2) \circ \mathbb{I}(A_1, A_0)$$

(si $n = 0$ on pose $\mathbb{I}(S) = id_{A_0}$).

Remarque 8.9 *On a*

$$\mathbb{I}(S; S') = \mathbb{I}(S') \circ \mathbb{I}(S)$$

Lemme 8.22 *On a :*

$$\mathbb{I}(S^{-1}) = \mathbb{I}(S)^{-1}$$

Comme pour l'oubli, la régularité de $\mathfrak{C}$ implique qu'il n'y a qu'une façon de permuter.

Lemme 8.23 *Si $\mathfrak{C}$ est régulière, alors $A \sim_n B$ et $B \sim_n C$ implique $A = C$, et $\mathbb{I}(B, C) \circ \mathbb{I}(A, B) = id_A$*

Preuve Supposons $A \sim_n B$ et $B \sim_n C$. Par régularité, comme $\mathcal{F}^n(B) = p_{\mathcal{F}^{n+1}(B)} * \mathcal{F}^{n+1}(A) = p_{\mathcal{F}^{n+1}(B)} * \mathcal{F}^{n+1}(C)$, on en déduit que $\mathcal{F}^{n+1}(B) = \mathcal{F}^{n+1}(A)$. Comme, $\mathcal{F}^n(A) = p_{\mathcal{F}^{n+1}(A)} * \mathcal{F}^{n+1}(B) = p_{\mathcal{F}^{n+1}(C)} * \mathcal{F}^{n+1}(B)$, on en déduit que $\mathcal{F}^n(A) = \mathcal{F}^n(C)$. En reprenant le raisonnement du lemme 8.20, on montre que

$$A = id_{\mathcal{F}^n(C)} *^n C = C$$

Idem pour $\mathbb{I}(B, C) \circ \mathbb{I}(A, B)$.

□

8.5 Morphismes décomposables en définitions élémentaires

Soit A tel que $A \neq \mathbf{1}$, on rappelle qu'un morphisme s de $\mathcal{C}[\mathcal{F}(A), A]$ est une *section* si et seulement si $p_A \circ s = id_{\mathcal{F}(A)}$. Intuitivement, le morphisme s définit donc le dernier champ de A à partir de $\mathcal{F}(A)$, sans modifier les champs de $\mathcal{F}(A)$. La notion de section correspond dans les D-systèmes à la notion de terme, rendue ici implicite (voir 7.2.4). Avec ce point de vue $\mathcal{F}(A)$ correspond au contexte de définition de ce terme. Dans [Str91], Streicher montre l'importance de la notion de section dans les catégories contextuelles.

On souhaite maintenant étendre la notion de section de la manière suivante : soient A et B deux signatures et f de $\mathcal{C}[B, A]$, on aimerait exprimer que f définit les champs de A qui ne sont pas dans B, éventuellement à partir de ceux qui sont dans B, mais sans modifier ceux qui sont déjà dans B.

Pour caractériser ces morphismes f, on va exiger que f se *"décompose en définitions élémentaires"*. Nous définissons cette notion d'une manière un peu similaire à la notion d'oubli. Dans la suite S dénote une suite de morphismes $(f_i)_{1 \leq i \leq n}$. La relation $S \in [A \ltimes B]$ signifie que S représente la décomposition d'un certain morphisme f de $\mathcal{C}[B, A]$ en définitions élementaires (les f_i). On appelle $\mathbb{G}[A, B](S)$ le morphisme f de $\mathcal{C}[B, A]$ associée à la décomposition S. Il faut noter que pour A, B et f de $\mathcal{C}[B, A]$ quelconque, il n'existe pas forcément de suite S telle que $S \in [A \ltimes B]$ et $\mathbb{G}[A, B](S) = f$.

Définition 8.20 *Soient $S = (f_i)_{1 \leq i \leq n}$ une suite de morphismes, et deux objets A et B. On définit la relation $S \in [A \ltimes B]$ et, lorsque $S \in [A \ltimes B]$, le morphisme $\mathbb{G}[A, B](S)$ de $\mathcal{C}[B, A]$. On procède par induction mutuelle sur $|A|$.*

1) *Si $A = \mathbf{1}$ alors $S \in [A \ltimes B]$ ssi $B = \mathbf{1}$ et $S = \emptyset$ (ie $n = 0$). Dans ce cas, on pose $\mathbb{G}[\mathbf{1}, \mathbf{1}](S) = id_{\mathbf{1}}$.*

2) *Si* $B \neq \mathbf{1}$ *et* $\mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$,

$$S \in [A \ltimes B] \;\; ssi \;\; \left\{ \begin{array}{c} S \in [\mathcal{F}(A) \ltimes \mathcal{F}(B)] \\ et \\ B = \mathbb{G}[\mathcal{F}(A), \mathcal{F}(B)](S) * A \end{array} \right.$$

$$\begin{tikzcd} B \arrow[r, "{\mathbb{G}[A,B](S)}"] \arrow[d, "p_B"'] & A \arrow[d, "p_A"] \\ \mathcal{F}(B) \arrow[r, "{\mathbb{G}[\mathcal{F}(A),\mathcal{F}(B)](S)}"'] & \mathcal{F}(A) \end{tikzcd}$$

Dans ce cas, on pose

$$\mathbb{G}[A, B](S) = q(\mathbb{G}[\mathcal{F}(A), \mathcal{F}(B)](S), A)$$

3) *Sinon,*

$$S \in [A \ltimes B] \;\; ssi \;\; \left\{ \begin{array}{c} n \geq 1, \\ f_n \in \mathcal{C}[B, A], \\ S' = (f_i)_{1 \leq i \leq n-1} \in [\mathcal{F}(A) \ltimes B], \\ et \\ p_A \circ f_n = \mathbb{G}[\mathcal{F}(A), B](S') \end{array} \right.$$

$$\begin{tikzcd} & A \arrow[d, "p_A"] \\ B \arrow[ur, "f_n"] \arrow[r, "{\mathbb{G}[\mathcal{F}(A),B](S')}"'] & \mathcal{F}(A) \end{tikzcd}$$

Dans ce cas, on pose

$$\mathbb{G}[A, B](S) = f_n$$

Soit f un morphisme de $\mathcal{C}[B, A]$. On dit que f se décompose en définitions élémentaires *s'il existe une suite S de morphismes telle que $S \in [A \ltimes B]$ et $f = \mathbb{G}[A, B](S)$.*

Lemme 8.24 (Exemple de telles décompositions)
Soient A et n tels que $n \leq |A|$.
Soit s de $\mathcal{C}[\mathcal{F}^n(A), A]$ tel que $p^n(A) \circ s = id_{\mathcal{F}^n(A)}$.
On pose

$$S = (p^{n-k}(A) \circ s)_{1 \leq k \leq n}$$

On a :
$S \in [A \ltimes \mathcal{F}^n(A)]$ *et* $\mathbb{G}[A, \mathcal{F}^n(A)](S) = s$

C'est vrai en particulier pour tout morphisme s de $\mathcal{C}[\mathbf{1}, A]$.

$$\begin{tikzcd} & A \arrow[d] \arrow[ddd, bend left, "p^n(A)"] \\ & \mathcal{F}(A) \arrow[d, dashed] \\ & \mathcal{F}^{n-1}(A) \arrow[d] \\ \mathcal{F}^n(A) \arrow[uuur, "s"] \arrow[uur] \arrow[ur] \arrow[r, "id_{\mathcal{F}^n(A)}"'] & \mathcal{F}^n(A) \end{tikzcd}$$

8.5.1 Propriétés élémentaires des décompositions

Comme pour l'oubli, si $S \in [A \ltimes B]$, alors A représente une certaine extension de B. En particulier, l'ordre des étiquettes dans A préserve celui de B.

Lemme 8.25 *Supposons $S \in [A \ltimes B]$. Alors, $Im(\mathfrak{L}_B) \subseteq Im(\mathfrak{L}_A)$. De plus, pour tous a et b de $Im(\mathfrak{L}_B)$, $\mathfrak{L}_B^{-1}(a) < \mathfrak{L}_B^{-1}(b)$ équivaut à $\mathfrak{L}_A^{-1}(a) < \mathfrak{L}_A^{-1}(b)$*

Preuve Par induction sur $|A|$.

$\square$

Lemme 8.26 *Pour tout objet A,*

1) $\emptyset \in [A \ltimes A]$ *et* $\mathbb{G}[A,A](\emptyset) = id_A$.

2) *Pour toute suite de morphisme* S, $S \in [A \ltimes A]$ *implique* $S = \emptyset$.

Preuve Trivial par induction sur $|A|$.

□

Lemme 8.27 $S \in [A \ltimes B]$ *et* $Im(\mathfrak{L}_A) = Im(\mathfrak{L}_B)$ *implique* $A = B$.

Preuve Trivial par induction sur $|A|$ en utilisant le lemme précédent.

□

On va maintenant montrer comment ces décompositions peuvent se composer.

Lemme-Définition 8.1 *Si* $S \in [A \ltimes B]$ *et* $S' \in [B \ltimes C]$*,on définit la suite* $(S \circledcirc_{A,B,C} S')$ *tel que* $(S \circledcirc_{A,B,C} S') \in [A \ltimes C]$ *et et* $\mathbb{G}[A,C](S \circledcirc_{A,B,C} S') = \mathbb{G}[A,B](S) \circ \mathbb{G}[B,C](S')$.

Construction Par induction sur $|A|$, puis par cas.

1) Si $A = \mathbf{1}$, alors $B = \mathbf{1}$ et $C = \mathbf{1}$. $S = S' = \emptyset$. On pose $\boxed{S \circledcirc_{A,B,C} S' = \emptyset}$.

2) Si $\mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$. On a $S \in [\mathcal{F}(A) \ltimes \mathcal{F}(B)]$ et $B = \mathbb{G}[\mathcal{F}(A), \mathcal{F}(B)](S) * A$.

2.1) Si $\mathfrak{L}_B(|B|) = \mathfrak{L}_C(|C|)$. On a $S' \in [\mathcal{F}(B) \ltimes \mathcal{F}(C)]$ et $C = \mathbb{G}[\mathcal{F}(B), \mathcal{F}(C)](S') * B$.
Donc $C = (\mathbb{G}[\mathcal{F}(A), \mathcal{F}(B)](S) \circ \mathbb{G}[\mathcal{F}(B), \mathcal{F}(C)](S')) * A$
Par hypothèse de récurrence, $(S \circledcirc_{\mathcal{F}(A),\mathcal{F}(B),\mathcal{F}(C)} S') \in [\mathcal{F}(A) \ltimes \mathcal{F}(C)]$ et
$\mathbb{G}[\mathcal{F}(A), \mathcal{F}(B)](S) \circ \mathbb{G}[\mathcal{F}(B), \mathcal{F}(C)](S') = \mathbb{G}[\mathcal{F}(A), \mathcal{F}(C)](S' \circledcirc_{\mathcal{F}(A),\mathcal{F}(B),\mathcal{F}(C)} S)$
On pose

$$\boxed{S \circledcirc_{A,B,C} S' = S \circledcirc_{\mathcal{F}(A),\mathcal{F}(B),\mathcal{F}(C)} S'}$$

Comme, $\mathfrak{L}_A(|A|) = \mathfrak{L}_C(|C|)$, on en déduit $(S \circledcirc_{A,B,C} S') \in [A \ltimes C]$.
Comme, $\mathbb{G}[A,B](S) = q(\mathbb{G}[\mathcal{F}(A), \mathcal{F}(B)](S), A)$ $\mathbb{G}[B,C](S') = q(\mathbb{G}[\mathcal{F}(B), \mathcal{F}(C)](S'), B)$
On en déduit $\mathbb{G}[A,C](S \circledcirc_{A,B,C} S') = \mathbb{G}[A,B](S) \circ \mathbb{G}[B,C](S')$

2.2) Si $\mathfrak{L}_B(|B|) \neq \mathfrak{L}_C(|C|)$. On a $S' = S''; g$ avec g dans $\mathcal{C}[C,B]$, et $S'' \in [\mathcal{F}(B) \ltimes C]$ et $p_B \circ g = \mathbb{G}[\mathcal{F}(B), C](S'')$.
Comme $\mathbb{G}[A,B](S) = \mathbb{G}[\mathcal{F}(A), \mathcal{F}(B)](S)$,

$$\begin{aligned} & p_A \circ (\mathbb{G}[A,B](S) \circ g) \\ &\quad = p_A \circ q(\mathbb{G}[\mathcal{F}(A), \mathcal{F}(B)](S), A) \circ g \\ &\quad = \mathbb{G}[\mathcal{F}(A), \mathcal{F}(B)](S) \circ p_B \circ g \\ &\quad = \mathbb{G}[\mathcal{F}(A), \mathcal{F}(B)](S) \circ \mathbb{G}[\mathcal{F}(B), \mathcal{F}(C)](S'') \end{aligned}$$

Par hypothèse de récurrence, $S \circledcirc_{\mathcal{F}(A),\mathcal{F}(B),C} S'' \in [\mathcal{F}(A) \ltimes C]$ et
$\mathbb{G}[\mathcal{F}(A), \mathcal{F}(B)](S) \circ \mathbb{G}[\mathcal{F}(B), C](S'') = \mathbb{G}[\mathcal{F}(A), C](S \circledcirc_{\mathcal{F}(A),\mathcal{F}(B),C} S'')$.
On pose

$$\boxed{S \circledcirc_{A,B,C} (S''; g) = (S \circledcirc_{\mathcal{F}(A),\mathcal{F}(B),C} S''); (\mathbb{G}[A,B](S) \circ g)}$$

Comme $\mathfrak{L}_A(|A|) \neq \mathfrak{L}_C(|C|)$, on en déduit que $S \circledcirc_{A,B,C} (S''; g) \in [A \ltimes C]$.
Et comme $\mathbb{G}[B,C](S''; g) = g$ et $\mathbb{G}[A,C](S \circledcirc_{A,B,C} (S''; g)) = \mathbb{G}[A,B](S) \circ g$, on a bien que $\mathbb{G}[A,C](S \circledcirc_{A,B,C} (S''; g)) = \mathbb{G}[A,B](S) \circ \mathbb{G}[B,C](S''; g)$

3) Sinon, on a $S = S''; f$ avec f de $\mathcal{C}[B, A]$ et $S'' \in [\mathcal{F}(A) \ltimes B]$ tels que $p_A \circ f = \mathbb{G}[\mathcal{F}(A), B](S'')$ et $S' \in [B \ltimes C]$.
On en déduit que $p_A \circ (f \circ \mathbb{G}[B, C](S')) = \mathbb{G}[\mathcal{F}(A), B](S'') \circ \mathbb{G}[B, C](S')$.
Par hypothèse de récurrence, $S'' \circledcirc_{\mathcal{F}(A),B,C} S' \in [\mathcal{F}(A) \ltimes C]$, et $\mathbb{G}[\mathcal{F}(A), B](S'') \circ \mathbb{G}[B, C](S') = \mathbb{G}[\mathcal{F}(A), C](S'' \circledcirc_{\mathcal{F}(A),B,C} S')$
On pose

$$\boxed{(S''; f) \circledcirc_{A,B,C} S' = (S'' \circledcirc_{\mathcal{F}(A),B,C} S'); (f \circ \mathbb{G}[B, C](S'))}$$

Comme $Im(\mathfrak{L}_C) \subseteq Im(\mathfrak{L}_B) \subseteq Im(\mathfrak{L}_{\mathcal{F}(A)})$, $\mathfrak{L}_A(|A|) \notin Im(\mathfrak{L}_C)$, on a bien $(S''; f) \circledcirc_{A,B,C} S' \in [A \ltimes C]$.
De plus, $\mathbb{G}[A, B](S''; f) = f$ et $\mathbb{G}[A, C]((S''; f) \circledcirc_{A,B,C} S') = f \circ \mathbb{G}[B, C](S')$, on en déduit que $\mathbb{G}[A, C]((S''; f) \circledcirc_{A,B,C} S') = \mathbb{G}[A, B](S''; f) \circ \mathbb{G}[B, C](S')$.

□

8.5.2 Construction d'une CS des décompositions

Afin de représenter les espèces de FocKer, on essaye de construire une catégorie dont les objets seraient des sortes d'enregistrements, dont certains champs pourraient être définis, et d'autres simplement déclarés. Les décompositions que l'on vient de définir sont de bons candidats pour devenir être ces objets. On construit donc dans ce qui suit une catégorie de signatures d'enregistrements, dont ces décompositions sont les objets.

8.5.2.a La catégorie $\mathcal{C}_{\ltimes}$ et sa structure d'arbre

Lemme 8.28 *On peut définir une catégorie $\mathcal{C}_{\ltimes}$ telle que*

1) *Les objets de $\mathcal{C}_{\ltimes}$ sont les triplets $< A, B, S >$ tels que $S \in [A \ltimes B]$.*

2) *Si $< A_1, B_1, S_1 >$ et $< A_2, B_2, S_2 >$ sont des objets de $\mathcal{C}_{\ltimes}$, les morphismes de $< A_1, B_1, S_1 >$ vers $< A_2, B_2, S_2 >$ sont les couples $< g, h >$ où $g \in \mathcal{C}[A_1, A_2]$ et $h \in \mathcal{C}[B_1, B_2]$ tels que*

$$\begin{tikzcd} B_1 \arrow[r, "{\mathbb{G}[A_1, B_1](S_1)}"] \arrow[d, "h"'] & A_1 \arrow[d, "g"] \\ B_2 \arrow[r, "{\mathbb{G}[A_2, B_2](S_2)}"'] & A_2 \end{tikzcd}$$

$$g \circ \mathbb{G}[A_1, B_1](S_1) = \mathbb{G}[A_2, B_2](S_2) \circ h$$

3) *La composition est définie par $< g_2, h_2 > \circ < g_1, h_1 > = < g_2 \circ g_1, h_2 \circ h_1 >$*

4) *L'identité sur $< A, B, S >$ est le morphisme $< id_A, id_B >$*

Preuve

1) On vérifie que l'identité et la composition définissent bien des morphismes. Pour la composition : pour i dans $\{1, 2, 3\}$, soit $X_i = < A_i, B_i, S_i >$ un "objet" de $\mathcal{C}_{\ltimes}$. Soient $< g_1, h_1 >$ de $\mathcal{C}_{\ltimes}[X_1, X_2]$ et $< g_2, h_2 >$ de $\mathcal{C}_{\ltimes}[X_2, X_3]$.

$$\begin{aligned} & g_2 \circ g_1 \circ \mathbb{G}[A_1, B_1](S_1) \\ & \quad = g_2 \circ \mathbb{G}[A_2, B_2](S_2) \circ h_1 \\ & \quad = \mathbb{G}[A_3, B_3](S_3) \circ g_2 \circ h_1 \end{aligned}$$

$< h_1 \circ g_1, h_2 \circ g_2 >$ est bien dans $\mathcal{C}_{\ltimes}[X_1, X_3]$.

2) L'associativité de la composition dans $\mathcal{C}_{\ltimes}$ découle directement de celle dans $\mathcal{C}$.

3) Idem pour la neutralité de l'identité vis-à-vis de la composition.

□

On va maintenant donner à cette catégorie une structure d'arbre.

Lemme 8.29 *Soit* $\mathbf{1}_\ltimes =< \mathbf{1}, \mathbf{1}, \emptyset >$. $\mathbf{1}_\ltimes$ *est terminal dans* $\mathcal{C}_\ltimes$.

Preuve Soit $X =< A, B, S >$ un objet de $\mathcal{C}_\ltimes$. Comme $\mathbf{1}$ est terminal dans $\mathcal{C}$, il existe un unique g de $\mathcal{C}[A, \mathbf{1}]$ et un unique h de $\mathcal{C}[B, \mathbf{1}]$. $g \circ \mathbb{G}[A, B](S)$ est dans $\mathcal{C}[B, \mathbf{1}]$. Donc par unicité de h, $h = g \circ \mathbb{G}[A, B](S)$. On en déduit que $< g, h >$ est dans $\mathcal{C}_\ltimes[X, \mathbf{1}]$, et qu'il est unique par unicité de g et h.

□

Lemme 8.30 *Soit* $X =< A, B, S >$ *de* $\mathcal{C}_\ltimes$. *On a* $X = \mathbf{1}_\ltimes$ *ssi* $A = \mathbf{1}$.

Preuve Immédiat par cas sur A.

□

Définition 8.21 *Si* $A \neq \mathbf{1}$ *et* $\mathfrak{L}_A(|A|) \notin Im(\mathfrak{L}_B)$, *on pose* $\mathcal{F}[A](B) = B$ *sinon on pose* $\mathcal{F}[A](B) = \mathcal{F}(B)$.

Définition 8.22 *Soit* $S \in [A \ltimes B]$. *On définit, par cas sur* A, $\mathcal{F}[A, B](S) \in [\mathcal{F}(A) \ltimes \mathcal{F}[A](B)]$.

1) *Si* $A = \mathbf{1}$ *alors* $\mathcal{F}[A, B](S) = \emptyset$.

2) *Si* $\mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$, *alors* $S \in [\mathcal{F}(A) \ltimes \mathcal{F}(B)]$ *et on pose* $\mathcal{F}[A, B](S) = S$.

3) *Sinon,* $S = S'; f$ *avec* $S' \in [\mathcal{F}(A) \ltimes B]$. *On pose* $\mathcal{F}[A, B](S) = S'$

Définition 8.23 *Pour* $< A, B, S > \in Ob(\mathcal{C}_\ltimes)$, *on pose* $\mathcal{F}_\ltimes(< A, B, S >) =< \mathcal{F}(A), \mathcal{F}[A](B), \mathcal{F}[A, B](S) >$. $\mathcal{F}_\ltimes(< A, B, S >) \in Ob(\mathcal{C}_\ltimes)$.

Lemme 8.31 $< Ob(\mathcal{C}_\ltimes), \mathcal{F}_\ltimes >$ *est un arbre dont* $\mathbf{1}_\ltimes$ *est la racine, et dont la profondeur est définie par* $|< A, B, S >|_\ltimes = |A|$.

Définition 8.24 *Soit* $X =< A, B, S >$ *de* $\mathcal{C}_\ltimes$, *on pose* $\mathfrak{L}_{\ltimes X} = \mathfrak{L}_A$.

Définition 8.25 *Pour* $A \neq \mathbf{1}$, *si* $\mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$, *on pose* $p[A]_B = p_B$, *sinon on pose* $p[A]_B = id_B$

Définition 8.26 *Si* $X =< A, B, S >$ *différent de* $\mathbf{1}_\ltimes$. *On pose* $p_{\ltimes X} =< p_A, p[A]_B >$. $p_{\ltimes X}$ *est un morphisme de* $\mathcal{C}_\ltimes[X, \mathcal{F}_\ltimes(X)]$.

Preuve On montre $p_A \circ \mathbb{G}[A, B](S) = \mathbb{G}[\mathcal{F}(A), \mathcal{F}[A](B)](\mathcal{F}[A, B](S)) \circ p[A]_B$ par cas sur A.

□

8.5.2.b Construction du produit fibré

Notation A partir de maintenant, on s'autorise à identifier la suite de décomposition S avec le(s) morphisme(s) qu'elle décomposent : on autorise l'abus de notation S au lieu de $\mathbb{G}[A, B](S)$. On notera alors $\mathcal{F}(S)$ au lieu de $\mathcal{F}[A, B](S)$. Plus précisement, on écrit :

1) Si $A = \mathbf{1}$, $S \in [A \ltimes B]$ ssi $B = \mathbf{1}$ et $S = id_{\mathbf{1}}$.

2) Si $B \neq \mathbf{1}$ et $\mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$, $S \in [A \ltimes B]$ ssi $B = \mathcal{F}(S) * A$ et $S = q(\mathcal{F}(S), A)$.

3) Sinon, $S \in [A \ltimes B]$ ssi $p_A \circ S = \mathcal{F}(S)$.

On notera **[comme ceci]** les passages non ambigüs, en particulier lorsqu'il faut définir une nouvelle suite, ou lorsqu'une même suite peut faire référence à plusieurs morphismes.

On commence maintenant par construire l'opérateur "$*$" pour $\mathcal{C}_\ltimes$ (noté "$*_\ltimes$") et "q" qui sera noté "$Q_\ltimes$".

Lemme-Définition 8.2 *Soient $X =< A_1, B_1, S_1 >$ et $Y =< A_2, B_2, S_2 >$ avec $Y \neq \mathbf{1}_\ltimes$, et soit $a \notin Im(\mathfrak{L}_{A_1})$. Soit $F =< g, h >$ un morphisme de $\mathcal{C}_\ltimes[X, \mathcal{F}_\ltimes(Y)]$. On construit :*

- *$F *^a_\ltimes Y$ un objet de $\mathcal{C}_\ltimes$ tel que $\mathcal{F}_\ltimes(F *^a_\ltimes Y) = X$.*
- *un morphisme de $\mathcal{C}_\ltimes[F *^a_\ltimes Y, Y]$, noté $Q^a_\ltimes(F, Y)$,*

Construction Par cas sur A_2 (en utilisant le fait que $A_2 \neq \mathbf{1}$).

1) Si $\mathfrak{L}_{A_2}(|A_2|) \in Im(\mathfrak{L}_{B_2})$, $B_2 = \mathcal{F}(S_2) * A_2$ et $S_2 = q(\mathcal{F}(S_2), A_2)$.

$$\begin{tikzcd}
F *^a_\ltimes Y & h *^a B_2 \arrow[rrr, "q^a(h, B_2)"] \arrow[dr, dotted, "q^a(S_1, g *^a A_2)"] \arrow[ddd, "p_{h*^a B_2}"'] & & & B_2 \arrow[dl, "S_2"'] \arrow[ddd, "p_{B_2}"] & Y \\
 & & g *^a A_2 \arrow[r, "q^a(g, A_2)"'] \arrow[d, "p_{g*^a A_2}"] & A_2 \arrow[d, "p_{A_2}"'] & & \\
 & & A_1 \arrow[r, "g"'] & \mathcal{F}(A_2) & & \\
X & B_1 \arrow[ur, "S_1"] \arrow[rrr, "h"'] & & & \mathcal{F}(B_2) \arrow[ul, "\mathcal{F}(S_2)"'] &
\end{tikzcd}$$

L'objet $h *^a B_2$ est bien défini car $a \notin Im(\mathfrak{L}_{A_1}) \supseteq Im(\mathfrak{L}_{B_1})$. De même $g *^a A_2$ est bien défini. De plus, on a :

$$\begin{aligned}
& h *^a B_2 \\
&\quad = h *^a (\mathcal{F}(S_2) * A_2) \\
&\quad = (\mathcal{F}(S_2) \circ h) *^a A_2 \\
&\quad = (g \circ S_1) *^a A_2 \\
&\quad = S_1 * (g *^a A_2)
\end{aligned}$$

[On pose $\phi_F = S_1$, on a $\phi_F \in [g *^a A_2 \ltimes h *^a B_2]$ et
$\mathbb{G}[g *^a A_2, h *^a B_2](\phi_F) = q(\mathbb{G}[A_1, B_1](S_1), g *^a A_2)]$. Avec les abus de notation précédent, on a plus simplement que $\phi_F = q(S_1, g *^a A_2)$.

On pose

$$\boxed{F *^a_\ltimes Y =< g *^a A_2, h *^a B_2, \phi_F >}$$

On a bien $\mathcal{F}_\ltimes(F *^a_\ltimes Y) = X$.

De plus,

$$\begin{aligned}
& q^a(g, A_2) \circ \phi_F \\
&\quad = q^a(g, A_2) \circ q(S_1, g *^a A_2) \\
&\quad = q^a(g \circ S_1, A_2) \\
&\quad = q^a(\mathcal{F}(S_2) \circ h, A_2) \\
&\quad = q(\mathcal{F}(S_2), A_2) \circ q^a(h, \mathcal{F}(S_2) * A_2) \\
&\quad = S_2 \circ q^a(h, B_2)
\end{aligned}$$

On pose

$$\boxed{Q_{\ltimes}(F,Y)^a = < q^a(g,A_2), q^a(h,B_2) >}$$

C'est bien un morphisme de $\mathcal{C}_{\ltimes}[F *^a_{\ltimes} Y, Y]$.

2) Sinon, $\mathfrak{L}_{A_2}(|A_2|) \notin Im(\mathfrak{L}_{B_2})$, $p_{A_2} \circ S_2 = \mathcal{F}(S_2)$.

$$\begin{tikzcd}
g *^a A_2 \arrow[rr, "q^a(g,A_2)"] \arrow[d, "p_{g*^a A_2}"] & & A_2 \arrow[d, "p_{A_2}"] \\
A_1 \arrow[rr, "g"] & & \mathcal{F}(A_2) \\
B_1 \arrow[u, "S_1"] \arrow[uu, dotted, bend left, "S_1 \curlywedge_{g*^a A_2} S_2 \circ h"] \arrow[rr, "h"] & & B_2 \arrow[u, "\mathcal{F}(S_2)"'] \arrow[uu, bend right, "S_2"']
\end{tikzcd}$$

L'objet $g *^a A_2$ est bien défini et $g \circ S_1 = \mathcal{F}(S_2) \circ h = p_{A_2} \circ S_2 \circ h$.

Donc $S_1 \curlywedge_{g*^a A_2} S_2 \circ h$ est bien défini.

On appelle ϕ_F ce morphisme de B_1 vers $g *^a A_2$, il vérifie $p_{g*^a A_2} \circ \phi_F = S_1$.

Donc $\phi_F \in [g *^a A_2 \ltimes B_1]$

[c'est-à-dire, on a implicitement posé $\phi_F = S_1; (\mathbb{G}[A_1, B_1](S_1) \curlywedge_{g*^a A_2} \mathbb{G}[A_2, B_2](S_2) \circ h)$**]**

On pose

$$\boxed{F *^a_{\ltimes} Y = < g *^a A_2, B_1, \phi_F >}$$

On a bien $\mathcal{F}_{\ltimes}(F *^a_{\ltimes} Y) = X$.

De plus, par définition de ϕ_F, on a $q^a(g,A_2) \circ \phi_F = S_2 \circ h$.

On pose

$$\boxed{Q^a_{\ltimes}(F,Y) = < q^a(g,A_2), h >}$$

C'est bien un morphisme de $\mathcal{C}_{\ltimes}[F *^a_{\ltimes} Y, Y]$.

$\square$

Lemme 8.32 *Sous les mêmes hypothèses,*

$$F \circ p_{\ltimes F *^a_{\ltimes} Y} = p_{\ltimes Y} \circ Q^a_{\ltimes}(F,Y)$$

Preuve Par cas sur A_2.

1) Si $\mathfrak{L}_{A_2}(|A_2|) \in Im(\mathfrak{L}_{B_2})$,

$$\begin{aligned}
& F \circ p_{\ltimes F *^a_{\ltimes} Y} \\
& \quad = < g, h > \circ < p_{g*^a A_2}, p_{h*^a B_2} > \\
& \quad = < g \circ p_{g*^a A_2}, h \circ p_{h*^a B_2} > \\
& \quad = < p_{A_2} \circ q^a(g,A_2), p_{B_2} \circ q^a(h,B_2) > \\
& \quad = < p_{A_2}, p_{B_2} > \circ < q^a(g,A_2), q^a(h,B_2) > \\
& \quad = p_{\ltimes Y} \circ Q^a_{\ltimes}(F,Y)
\end{aligned}$$

2) Sinon, $\mathfrak{L}_{A_2}(|A_2|) \notin Im(\mathfrak{L}_{B_2})$,

$$\begin{aligned}
& F \circ p_{\ltimes F *^a_{\ltimes} Y} \\
& \quad = < g, h > \circ < p_{g*^a A_2}, id_{B_1} > \\
& \quad = < g \circ p_{g*^a A_2}, h > \\
& \quad = < p_{A_2} \circ q^a(g,A_2), h > \\
& \quad = < p_{A_2}, id_{B_2} > \circ < q^a(g,A_2), h > \\
& \quad = p_{\ltimes Y} \circ Q^a_{\ltimes}(F,Y)
\end{aligned}$$

$\square$

On construit maintenant le morphisme " $\curlywedge$ " de ce produit fibré.

Lemme 8.33 *Soit $Z =< A_3, B_3, S_3 >$ de $\mathcal{C}_\ltimes$ tel qu'il existe $G =< g_1, h_1 >$ de $\mathcal{C}_\ltimes[Z, X]$ et $H =< g_2, h_2 >$ de $\mathcal{C}_\ltimes[Z, Y]$ tels que $F \circ G = p_{\ltimes Y} \circ H$.*

Il existe un unique morphisme K de $\mathcal{C}_\ltimes[Z, F *^a_\ltimes Y]$ tel que $p_{\ltimes F *^a_\ltimes Y} \circ K = G$ et $Q_\ltimes(F,Y)^a \circ K = H$.

Preuve On a les égalités suivantes

$$g_1 \circ S_3 = S_1 \circ h_1 \qquad g_2 \circ S_3 = S_2 \circ h_2 \qquad g \circ g_1 = p_{A_2} \circ g_2$$

On pose $g_3 = g_1 \curlywedge_{g *^a A_2} g_2$. Par définition, g_3 est l'unique morphisme $\mathcal{C}[A_3, g *^a A_2]$ à vérifier $p_{g *^a A_2} \circ g_3 = g_1$ et $q^a(g, A_2) \circ g_3 = g_2$.
On a

$$g \circ g_1 \circ S_3 = p_{A_2} \circ g_2 \circ S_3$$

Donc il existe un unique j de $\mathcal{C}[B_3, g *^a A_2]$ tel que $p_{g *^a A_2} \circ j = g_1 \circ S_3$ et $q^a(g, A_2) \circ j = g_2 \circ S_3$.
Or $p_{g *^a A_2} \circ g_3 \circ S_3 = g_1 \circ S_3$ et $q^a(g, A_2) \circ g_3 \circ S_3 = g_2 \circ S_3$.
Donc $j = g_3 \circ S_3$.
Par cas sur A_2.

1) Si $\mathfrak{L}_{A_2}(|A_2|) \in Im(\mathfrak{L}_{B_2})$. On a $h \circ h_1 = p_{B_2} \circ h_2$.

On pose $h_3 = h_1 \curlywedge_{h *^a B_2} h_2$. Par définition, h_3 est l'unique morphisme de $\mathcal{C}[B_3, h *^a B_2]$ à vérifier $p_{h *^a B_2} \circ h_3 = h_1$ et $q^a(h, B_2) \circ h_3 = h_2$.

On a,

$$\begin{aligned} & p_{g *^a A_2} \circ \phi_F \circ h_3 \\ &= S_1 \circ p_{h *^a B_2} \circ h_3 \\ &= S_1 \circ h_1 \\ &= g_1 \circ S_3 \end{aligned}$$

Et,

$$\begin{aligned} & q^a(g, A_2) \circ \phi_F \circ h_3 \\ &= S_2 \circ q^a(h, B_2) \circ h_3 \\ &= S_2 \circ h_2 \\ &= g_2 \circ S_3 \end{aligned}$$

On en déduit que $j = g_3 \circ S_3 = \phi_F \circ h_3$.

On pose $\boxed{K =< g_3, h_3 >}$. C'est un morphisme de $\mathcal{C}_\ltimes[Z, F *^a_\ltimes Y]$. De plus

$$< p_{g *^a A_2}, p_{h *^a B_2} > \circ < g_3, h_3 > = < g_1, h_1 >$$

Et

$$< q^a(g, A_2), q^a(h, B_2) > \circ < g_3, h_3 > = < g_2, h_2 >$$

K est l'unique morphisme à vérifier cette relation, par l'unicité de g_3 et h_3.

2) Sinon, $h \circ h_1 = g_2$.

On a :

$$\begin{aligned} & p_{g *^a A_2} \circ \phi_F \circ h_1 \\ &= S_1 \circ h_1 \\ &= g_1 \circ S_3 \end{aligned}$$

Et,

$$\begin{aligned} & q^a(g, A_2) \circ \phi_F \circ h_1 \\ &= S_2 \circ h \circ h_1 \\ &= S_2 \circ g_2 \\ &= g_2 \circ S_3 \end{aligned}$$

On en déduit que $j = g_3 \circ S_3 = \phi_F \circ h_1$.

On pose $\boxed{K =< g_3, h_1 >}$. C'est un morphisme de $\mathcal{C}_\ltimes[Z, F *^a_\ltimes Y]$. De plus

$$< p_{g *^a A_2}, id_{B_1} > \circ < g_3, h_1 > = < g_1, h_1 >$$

Et

$$< q^a(g, A_2), h > \circ < g_3, h_1 > = < g_2, h_2 >$$

K est l'unique morphisme à vérifier cette relation, par l'unicité de g_3, et le fait que h_1 est l'unique morphisme x à vérifier $id_{B_1} \circ x = h_1$.

□

Lemme 8.34 *Soit* $X =< A, B, S >\neq \mathbf{1}_{\ltimes}$.

$$id_{\mathcal{F}_{\ltimes}(X)} *_{\ltimes} X = X \qquad et \qquad Q_{\ltimes}(id_{\mathcal{F}_{\ltimes}(X)}, X) = id_X$$

Preuve Par cas.

1) Si $\mathfrak{L}_A(|A|) \in Im(\mathfrak{L}_B)$. $S = q(\mathcal{F}(S), A)$ On a :

$$\begin{aligned} & id_{\mathcal{F}_{\ltimes}(X)} *_{\ltimes} X \\ & \quad =< id_{\mathcal{F}(A)} * A, id_{\mathcal{F}(B)} * B, q(\mathcal{F}(S), id_{\mathcal{F}(A)} * A) > \\ & \quad =< A, B, S > \end{aligned}$$

Et,

$$\begin{aligned} & Q_{\ltimes}(id_{\mathcal{F}_{\ltimes}(X)}, X) \\ & \quad =< q(id_{\mathcal{F}(A)}, A), q(id_{\mathcal{F}(B)}, B) > \\ & \quad =< id_A, id_B > \end{aligned}$$

2) Sinon, $p_A \circ S = \mathcal{F}(S)$.

$$\begin{aligned} & id_{\mathcal{F}_{\ltimes}(X)} *_{\ltimes} X \\ & \quad =< id_{\mathcal{F}(A)} * A, B, \mathcal{F}(S) \curlywedge_{id_{\mathcal{F}(A)}*A} S \circ id_B > \\ & \quad =< A, B, \mathcal{F}(S) \curlywedge_A S > \end{aligned}$$

Or par définition $q(id_{\mathcal{F}(A)}, A) \circ \mathcal{F}(S) \curlywedge_A S = S$. Donc $\mathcal{F}(S) \curlywedge_A S = S$.
Et,

$$\begin{aligned} & Q_{\ltimes}(id_{\mathcal{F}_{\ltimes}(X)}, X) \\ & \quad =< q(id_{\mathcal{F}(A)}, A), id_B > \\ & \quad =< id_A, id_B > \end{aligned}$$

□

Lemme 8.35 *Soit* $X_i =< A_i, B_i, S_i >$, *pour* $i \in [1,3]$ *de* $Ob(\mathcal{C}_{\ltimes})$, *et* a *et* b *de* $\mathfrak{A}$ *tels que* $a \notin Im(\mathfrak{L}_{A_1})$ *et* $b \notin Im(\mathfrak{L}_{A_2})$. *et* $F =< g_1, h_1 >$ *de* $\mathcal{C}_{\ltimes}[X_1, X_2]$ *et* $G =< g_2, h_2 >$ *de* $\mathcal{C}[X_2, \mathcal{F}_{\ltimes}(X_3)]$,

$$(G \circ F) *^a_{\ltimes} X_3 = F *^a_{\ltimes} (G *^b_{\ltimes} X_3) \qquad et \qquad Q^a_{\ltimes}(G \circ F, X_3) = Q^b_{\ltimes}(G, X_3) \circ Q^a_{\ltimes}(F, G *^b_{\ltimes} X_3)$$

Preuve Par cas.

1) Si $\mathfrak{L}_{A_3}(|A_3|) = \mathfrak{L}_{B_3}(|B_3|)$,

$$\begin{aligned} & F *^a_{\ltimes} (G *^b_{\ltimes} X_3) \\ & \quad =< g_1, h_1 > *^a_{\ltimes} < g_2 *^b A_3, h_2 *^b B_3, q(S_2, g_2 *^b A_3) > \\ & \quad =< g_1 *^a (g_2 *^b A_3), h_1 *^a (h_2 *^b B_3), q(S_1, g_1 *^a (g_2 *^b A_3)) > \\ & \quad =< (g_2 \circ g_1) *^a A_3, (h_2 \circ h_1) *^a B_3, q(S_1, (g_2 \circ g_1) *^a A_3) > \\ & \quad = (G \circ F) *^a_{\ltimes} X_3 \end{aligned}$$

$$\begin{aligned} & Q^b_{\ltimes}(G, X_3) \circ Q^a_{\ltimes}(F, G *^b_{\ltimes} X_3) \\ & \quad =< q^b(g_2, A_3), q^b(h_2, B_3) > \circ < q^a(g_1, g_2 *^b A_3), q^a(h_1, h_2 *^b B_3) > \\ & \quad =< q^a(g_2, A_3) \circ q^a(g_1, g_2 *^b A_3), q^b(h_2, B_3) \circ q^a(h_1, h_2 *^b B_3) > \\ & \quad =< q^a(g_2 \circ g_1, A_3), q^a(h_2 \circ h_1, B_3) > \\ & \quad = Q^a_{\ltimes}(G \circ F, X_3) \end{aligned}$$

2) Sinon, $\mathfrak{L}_{A_3}(|A_3|) \notin Im(\mathfrak{L}_{B_3})$,

$$\begin{aligned}
&F *^a_{\ltimes} (G *^b_{\ltimes} X_3)\\
&\quad = < g_1, h_1 > *^a_{\ltimes} < g_2 *^b A_3, B_3, S_2 \curlywedge_{g_2 *^b A_3} S_3 \circ h_2 >\\
&\quad = < g_1 *^a (g_2 *^b A_3), B_3, S_1 \curlywedge_{g_1 *^a (g_2 *^b A_3)} (S_2 \curlywedge_{g_2 *^b A_3} S_3 \circ h_2) \circ h_1 >\\
&\quad = < (g_2 \circ g_1) *^a A_3, B_3, S_1 \curlywedge_{(g_2 \circ g_1) *^a A_3} S_3 \circ (h_2 \circ h_1) >
\end{aligned}$$

En effet, $S_1 \curlywedge_{g_1 *^a (g_2 *^b A_3)} (S_2 \curlywedge_{g_2 *^b A_3} S_3 \circ h_2) \circ h_1$ est un morphisme ϕ qui vérifie $p_{g_1 *^a (g_2 *^b A_3)} \circ \phi = S_1$ et $q^a(g_1, g_2 *^b A_3) \circ \phi = (S_2 \curlywedge_{g_2 *^b A_3} S_3 \circ h_2) \circ h_1$
Donc

$$\begin{aligned}
&q^a(g_2 \circ g_1, A_3) \circ \phi\\
&\quad = q^b(g_2, A_3) \circ q^a(g_1, g_2 *^b A_3) \circ \phi\\
&\quad = q^b(g_2, A_3) \circ (S_2 \curlywedge_{g_2 *^b A_3} S_3 \circ h_2) \circ h_1\\
&\quad = S_3 \circ h_2 \circ h_1
\end{aligned}$$

Or $S_1 \curlywedge_{(g_2 \circ g_1) *^a A_3} S_3 \circ (h_2 \circ h_1)$ est le seul morphisme ψ tel que $p_{g_1 *^a (g_2 *^b A_3)} \circ \psi = S_1$ et $q^a(g_2 \circ g_1, A_3) \circ \psi = S_3 \circ h_2 \circ h_1$. On en conclut que $\psi = \phi$.

$$\begin{aligned}
&Q^b_{\ltimes}(G, X_3) \circ Q^a_{\ltimes}(F, G *^b_{\ltimes} X_3)\\
&\quad = < q^b(g_2, A_3), h_2 > \circ < q^a(g_1, g_2 *^b A_3), h_1 >\\
&\quad = < q^b(g_2 \circ g_1, A_3), h_2 \circ h_1 >\\
&\quad = Q^a_{\ltimes}(G \circ F, X_3)
\end{aligned}$$

□

Théorème 8.3 *La structure* $\mathfrak{C}_{\ltimes} =< \mathcal{C}_{\ltimes}, \mathfrak{A}, \mathcal{F}_{\ltimes}, \mathfrak{L}_{\ltimes}, \mathbf{1}_{\ltimes}, p_{\ltimes}, *_{\ltimes}, Q_{\ltimes} >$ *est une catégorie de signatures d'enregistrements.*

Définition 8.27 *Soit* $._{\ltimes}$ *la transformation qui envoie tout objet* A *de* $\mathcal{C}$ *vers un objet* $A_{\ltimes} =< A, A, [\emptyset] >$ *de* $\mathcal{C}_{\ltimes}$*, et tout morphisme* f *de* $\mathcal{C}[A, B]$ *vers un morphisme* $< f, f >$ *de* $\mathcal{C}_{\ltimes}[A_{\ltimes}, B_{\ltimes}]$.

Lemme 8.36 $._{\ltimes}$ *est un foncteur fidèle de catégorie de signatures d'enregistrements.*

8.5.2.c Préservation de la propriété de régularité

On va maintenant montrer que si $\mathfrak{C}$ est régulière, alors $\mathfrak{C}_{\ltimes}$ l'est aussi.

Lemme 8.37 *Soit* $F =< g, h >$ *un morphisme de* $\mathcal{C}_{\ltimes}$ *tels que* g *et* h *sont réguliers. Alors* F *est régulier.*

Preuve F est un épimorphisme, c'est évident.
On montre donc que pour tout $X =< A, B, S >$ et $Y =< A', B', S' >$ de $\mathcal{C}_{\ltimes}$ tels que $\mathfrak{L}_X(|X|) \notin Im(\mathfrak{L}_{Dom(F)})$, $\mathfrak{L}_Y(|Y|) \notin Im(\mathfrak{L}_{Dom(F)})$ et $\mathcal{F}_{\ltimes}(X) = \mathcal{F}_{\ltimes}(Y) = Im(F)$, $F *_{\ltimes} X = F *_{\ltimes} Y$ implique $X = Y$.
On suppose $Dom(F) =< A_0, B_0, S_0 >$. On raisonne par cas :

1) Si $\mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$ et $\mathfrak{L}_{A'}(|A'|) = \mathfrak{L}_{B'}(|B'|)$. On a :
$F *_{\ltimes} X =< g * A, h * B, [\mathbf{S_0}] >$ et $F *_{\ltimes} X =< g * A', h * B', [\mathbf{S_0}] >$

Par régularité de g et h, alors $A = A'$ et $B = B'$. De plus $\mathcal{F}_{\ltimes}(X) =< \mathcal{F}(A), \mathcal{F}(B), [\mathbf{S}] >$ et $\mathcal{F}_{\ltimes}(Y) =< \mathcal{F}(A'), \mathcal{F}(B'), [\mathbf{S'}] >$. Comme $\mathcal{F}_{\ltimes}(X) = \mathcal{F}_{\ltimes}(Y)$, on en déduit que $S = S'$.

2) Si $\mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$ et $\mathfrak{L}_{A'}(|A'|) \notin Im(\mathfrak{L}_{B'})$. On en déduit que $h * B = B'$, et en particulier que $\mathfrak{L}_B(|B|) = \mathfrak{L}_{B'}(|B'|)$. Or $B' = \mathcal{F}(B)$, d'où $\mathfrak{L}_B(|B|) \notin Im(\mathfrak{L}_{B'})$. C'est absurde.

3) Si $\mathfrak{L}_A(|A|) \notin Im(\mathfrak{L}_B)$ et $\mathfrak{L}_{A'}(|A'|) = \mathfrak{L}_{B'}(|B'|)$. Cas symétrique du précédent : impossible.

4) Si $\mathfrak{L}_A(|A|) \notin Im(\mathfrak{L}_B)$ et $\mathfrak{L}_{A'}(|A'|) \notin Im(\mathfrak{L}_{B'})$. On a : $[\mathbf{S} = \mathcal{F}[\mathbf{A},\mathbf{B}](\mathbf{S}); \mathbb{G}[\mathbf{A},\mathbf{B}](\mathbf{S})]$, et $[\mathbf{S}' = \mathcal{F}[\mathbf{A}',\mathbf{B}'](\mathbf{S}'); \mathbb{G}[\mathbf{A}',\mathbf{B}'](\mathbf{S}')]$. On déduit de $\mathcal{F}_{\ltimes}(X) = \mathcal{F}_{\ltimes}(Y)$ que $\mathcal{F}[A,B](S) = \mathcal{F}[A',B'](S')$ et $B = B'$. De plus,

$$F * X =< g * A, B_0, [\mathbf{S_0}; \phi] > \qquad \text{et} \qquad g * X =< p_{A_0} * A', B_0, [\mathbf{S_0}; \phi'] >$$

où $\phi = \mathbb{G}[A_0, B_0](S_0) \curlywedge_{g*A} \mathbb{G}[A,B](S) \circ h$ et $\phi' = \mathbb{G}[A_0, B_0](S_0) \curlywedge_{g*A'} \mathbb{G}[A',B'](S') \circ h$

On en déduit que $A = A'$ et $\phi = \phi'$. Donc $q(g,A) \circ \phi = \mathbb{G}[A,B](S) \circ h = \mathbb{G}[A',B'](S') \circ h$. Par épimorphisme de h, on en déduit que $\mathbb{G}[A,B](S) = \mathbb{G}[A',B'](S')$. Et donc finalement que $S = S'$.

□

Lemme 8.38 *Soit $F =< g, h >$ tels que g et h sont ultra-réguliers, soit X tel que $\mathfrak{L}_X(|X|) \notin Im(\mathfrak{L}_{Dom(F)})$ et $\mathcal{F}_{\ltimes}(X) = Im(F)$. Alors il existe deux morphismes de $\mathcal{C}$ ultra-réguliers g' et h' tels que $Q_{\ltimes}(F,X) =< g', h' >$.*

Preuve Supposons $X =< A, B, S >$. On raisonne par cas :

1) Si $\mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$, alors $Q_{\ltimes}(F,X) =< q(g,A), q(h,B) >$.

2) Sinon, $Q_{\ltimes}(F,X) =< q(g,A), h >$.

On conclut dans les 2 cas en utilisant le lemme 8.10

□

Théorème 8.4 *si $\mathfrak{C}$ est régulière, alors $\mathfrak{C}_{\ltimes}$ aussi.*

Preuve Soient X, Y et n tels que $\mathcal{F}_{\ltimes}(X) = \mathcal{F}^n_{\ltimes}(Y)$ et $\mathfrak{L}_X(|X|) \notin Im(\mathfrak{L}_Y)$.
On montre par récurrence sur n que $Q^n_{\ltimes}(p_{\ltimes X}, Y) =< g, h >$ où g et h sont ultra-réguliers.
On procède par induction sur n.

1) si $n = 0$, on doit montrer que $p_{\ltimes X} =< g, h >$ avec g et h ultra-réguliers. On utilise le fait que pour tout objet A de $\mathcal{C}$, id_A et p_A sont ultra-réguliers.

2) sinon, on applique le lemme précédent sur l'hypothèse de récurrence.

On en déduit que $Q^n_{\ltimes}(p_{\ltimes X}, Y) =< g, h >$ où g et h sont réguliers. On peut donc appliquer le lemme 8.37 et en déduire que pour tout objet X de $\mathcal{C}_{\ltimes}$, $p_{\ltimes X}$ est ultra-régulier.

□

Chapitre 9

Catégories de générateurs d'enregistrements (ou CG)

Etant donnée une catégorie de signatures d'enregistrement $\mathfrak{C}$, on a construit une catégorie de signature d'enregistrement $\mathfrak{C}_{\ltimes}$ qui étend $\mathfrak{C}$ de manière à proposer une notion de signature dans laquelle certains champs sont définis, et d'autres sont simplement déclarés.

Fidèle à la démarche initiale, on cherche maintenant une description plus abstraite de $\mathfrak{C}_{\ltimes}$, par propriétés. Cela conduit à la notion de "générateurs d'enregistrements". Comme pour les espèces de FocKer, les objets de $\mathfrak{C}_{\ltimes}$ peuvent se voir comme des fonctions qui prennent en entrée des champs non encore définis, et qui retourne un enregistrement complètement défini. Cela se traduit par un foncteur $\mathcal{H}$ qui associe à chaque objet A la signature contenant les champs non définis de cet objet A, et une transformation naturelle $\mathbb{G}$ qui associe à A le morphisme définissant les champs de A à partir de $\mathcal{H}(A)$.

Une autre idée est que si le dernier champ de A est défini (on dira ici *manifeste*), on peut toujours oublier sa définition pour le rendre simplement déclaré (on dira ici *abstrait*) : comme pour les signatures d'enregistrements, on caractérise les *générateurs d'enregistrements* par les propriétés de l'oubli. Cet oubli est caractérisé d'une part par l'objet $\mathcal{A}(A)$ qui correspond à A dans lequel la définition éventuelle du dernier champ à été oubliée, et d'autre part par le morphisme α_A de A dans $\mathcal{A}(A)$ qui est la conversion associée.

Enfin si A est un générateur dont le dernier champ est abstrait, et s est une section de $\mathcal{H}(A)$, on peut construire le générateur $\mathcal{M}(s)$ qui définit ce champ.

On peut donc considérer un générateur comme une signature avec des champs manifestes et des champs abstraits. On verra ainsi dans le chapitre 10 une axiomatisation catégorique de la notion de signature avec champs manifestes proposée par Pollack, et on montrera que celle-ci forme une catégorie de générateurs d'enregistrements.

Les espèces de FocKer seront une forme particulière de générateurs d'enregistrements (qui ne correspondent pas par exemple aux signatures avec champs manifestes de Pollack). La notion de générateurs d'enregistrements nous permet de faire apparaître petit-à-petit les propriétés des espèces.

9.1 Définition

Définition 9.1 *Une* catégorie de générateurs d'enregistrements *(ou CG en abrégé)* $\mathfrak{C}$ *est un uplet* $< \mathcal{C}, \mathtt{A}, \mathcal{F}, \mathfrak{L}, \mathbf{1}, p, *, q, \mathcal{A}, \alpha, \mathcal{H}, \mathbb{G}, \mathcal{M} >$ *tel que :*

1) $< \mathcal{C}, \mathtt{A}, \mathcal{F}, \mathfrak{L}, \mathbf{1}, p, *, q >$ *est une catégorie des signatures d'enregistrements.*

2) $\mathcal{A} : Ob(\mathcal{C}) \to Ob(\mathcal{C})$ *tel que pour tout* $A \neq \mathbf{1}$,

$$\mathcal{F}(\mathcal{A}(A)) = \mathcal{F}(A) \qquad et \qquad \mathfrak{L}(\mathcal{A}(A)) = \mathfrak{L}(A)$$

$\mathcal{A}(A)$ *s'appelle* l'abstrait *de* A.

3) *Pour tout* $A \neq \mathbf{1}$, *il existe un morphisme* α_A *de* $\mathcal{C}[A, \mathcal{A}(A)]$ *tel que*

$$p_{\mathcal{A}(A)} \circ \alpha_A = p_A$$

$$\begin{tikzcd} A \arrow[rr, "\alpha_A"] \arrow[dr, "p_A"'] & & \mathcal{A}(A) \arrow[dl, "p_{\mathcal{A}(A)}"] \\ & \mathcal{F}(A) & \end{tikzcd}$$

α_A *s'appelle* l'abstraction *de* A.

4) $\mathcal{A}(\mathbf{1}) = \mathbf{1}$ *et pour tout* A, $\mathcal{A}(\mathcal{A}(A)) = \mathcal{A}(A)$ *et* $\alpha_{\mathcal{A}(A)} = id_{\mathcal{A}(A)}$

5) *Pour tous* A *et* B *de* $Ob(\mathcal{C})$, *pour tout* a *de* $\mathtt{A}$ *avec* $a \notin Im(\mathfrak{L}_A)$, *pour tout* f *de* $\mathcal{C}[A, \mathcal{F}(B)]$,

$$\mathcal{A}(f *^a B) = f *^a \mathcal{A}(B)$$
$$et$$
$$q^a(f, \mathcal{A}(B)) \circ \alpha_{f *^a B} = \alpha_B \circ q^a(f, B)$$

$$\begin{tikzcd} f *^a B \arrow[r, "\alpha_{f *^a B}"] \arrow[drr, "q^a(f,B)"] \arrow[dd, "p_{f *^a B}"'] & \mathcal{A}(f *^a B) = f *^a \mathcal{A}(B) \arrow[drr, "q^a(f,\mathcal{A}(B))"] \arrow[ddl, "p_{f *^a \mathcal{A}(B)}"] & & \\ & & B \arrow[r, "\alpha_B"'] \arrow[d, "p_B"'] & \mathcal{A}(B) \arrow[dl, "p_{\mathcal{A}(B)}"] \\ A \arrow[rr, "f"'] & & \mathcal{F}(B) & \end{tikzcd}$$

6) *Avec les hypothèses précédentes,* $f *^a B = f *^a \mathcal{A}(B)$ *implique* $B = \mathcal{A}(B)$

7) $\mathcal{H}$ *est un endofoncteur, appelé* foncteur hypothèse. *Et* $\mathbb{G}$ *une transformation naturelle de* $\mathcal{H}$ *vers* $Id_{\mathcal{C}}$ *(c'est-à-dire que le diagramme ci-contre commute). Pour tout* A, $\mathbb{G}(A)$ *est appelé le* générateur *de* A.

$$\begin{tikzcd} \mathcal{H}(A) \arrow[r, "\mathbb{G}(A)"] \arrow[d, "\mathcal{H}(f)"'] & A \arrow[d, "f"] \\ \mathcal{H}(B) \arrow[r, "\mathbb{G}(B)"'] & B \end{tikzcd}$$

8) $\mathcal{H}(\mathbf{1}) = \mathbf{1}$ *et* $G(\mathbf{1}) = id_{\mathbf{1}}$.

9) *Pour tout* A, *si* $A = \mathcal{A}(A) \neq \mathbf{1}$,

$$\mathcal{H}(A) = \mathbb{G}(\mathcal{F}(A)) * A$$
$$et$$
$$\mathcal{H}(p_A) = p_{\mathcal{H}(A)}$$
$$et$$
$$\mathbb{G}(A) = q(\mathbb{G}(\mathcal{F}(A)), A)$$

$$\begin{tikzcd} \mathbb{G}(\mathcal{F}(A)) * A = \mathcal{H}(A) \arrow[r, "\mathbb{G}(A)"] \arrow[d, "p_{\mathcal{H}(A)}"'] & A = \mathcal{A}(A) \arrow[d, "p_A"] \\ \mathcal{F}(\mathcal{H}(A)) = \mathcal{H}(\mathcal{F}(A)) \arrow[r, "\mathbb{G}(\mathcal{F}(A))"'] & \mathcal{F}(A) \end{tikzcd}$$

10) *Pour tout* A, *si* $A \neq \mathcal{A}(A)$,

$$\mathcal{H}(A) = \mathcal{H}(\mathcal{F}(A)) \quad et \quad \mathcal{H}(p_A) = id_{\mathcal{H}(A)}$$

$$\begin{tikzcd} & A \arrow[dr, "p_A"] & \\ \mathcal{H}(A) = \mathcal{H}(\mathcal{F}(A)) \arrow[ur, "\mathbb{G}(A)"] \arrow[rr, "\mathbb{G}(\mathcal{F}(A))"] & & \mathcal{F}(A) \end{tikzcd}$$

11) *Pour* $A = \mathcal{A}(A)$, *pour tout morphisme* s *de* $\mathcal{C}[\mathcal{H}(\mathcal{F}(A)), \mathcal{H}(A)]$ *tel que* $p_{\mathcal{H}(A)} \circ s = id_{\mathcal{H}(\mathcal{F}(A))}$, *il existe un unique objet* $\mathcal{M}(s)$ *de* $\mathcal{C}$ *tel que*

$$\mathcal{A}(\mathcal{M}(s)) = A \quad et \quad A \neq \mathcal{M}(s) \quad et \quad \mathbb{G}(A) \circ s = \alpha_{\mathcal{M}(s)} \circ \mathbb{G}(\mathcal{M}(s))$$

$$\begin{tikzcd} \mathcal{H}(A) \arrow[r, "\mathbb{G}(A)"'] & A = \mathcal{A}(\mathcal{M}(s)) \\ \mathcal{H}(\mathcal{M}(s)) = \mathcal{H}(\mathcal{F}(A)) \arrow[u, "s"] \arrow[r, "\mathbb{G}(\mathcal{M}(s))"'] & \mathcal{M}(s) \arrow[u, "\alpha_{\mathcal{M}(s)}"'] \end{tikzcd}$$

$\mathcal{M}(s)$ *s'appelle le* manifeste *de* s.

9.1.1 Exemple de $\mathcal{C}_{\ltimes}$

Soit $< \mathcal{C}, \mathfrak{A}, \mathcal{F}, \mathfrak{L}, \mathbf{1}, p, *, q >$ une catégorie des signatures d'enregistrements. On va construire pour $\mathcal{C}_{\ltimes}$ une structure de catégorie de générateurs d'enregistrements. On s'appuie sur l'intuition que dans un objet de $\mathcal{C}_{\ltimes}$ de la forme $X =< A, B, S >$, l'objet A représente l'interface de X, l'objet B représente les hypothèses de X et S permet d'engendrer un enregistrement (dans $\mathcal{C}$) de signature A à partir d'un enregistrement de signature B.

9.1.1.a Construction et propriétés de $\mathcal{A}_{\ltimes}$ et $\alpha_{\ltimes}$

Définition 9.2 *Soit* $X =< A, B, S >$ *de* $\mathcal{C}_{\ltimes}$. *On définit* $\mathcal{A}_{\ltimes}(X)$ *de* $\mathcal{C}_{\ltimes}$.

- *Si* $A = \mathbf{1}$ *ou* $\mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$, *on pose* $\boxed{\mathcal{A}_{\ltimes}(X) = X}$.
- *Sinon,* $\mathfrak{L}_A(|A|) \notin Im(\mathfrak{L}_B)$, *et on a* $p_A \circ S = \mathcal{F}(S)$.
  Soit $C = \mathcal{F}(S) * A$,
  et soit $S' = q(\mathcal{F}(S), A)$ [**ie** $S' = \mathcal{F}(S)$].
  Comme $\mathfrak{L}_A(|A|) = \mathfrak{L}_C(|C|)$, *on a* $S' \in [A \ltimes C]$.
  On pose

$$\boxed{\mathcal{A}_{\ltimes}(X) =< A, C, S' >}$$

$$\begin{tikzcd} C \arrow[r, "S'"] \arrow[d, "p_C"'] & A \arrow[d, "p_A"] \\ B \arrow[ur, "S"] \arrow[r, "\mathcal{F}(S)"'] & \mathcal{F}(A) \end{tikzcd}$$

Lemme 9.1 *Avec les mêmes hypothèses. On a* $\mathcal{A}_{\ltimes}(\mathcal{A}_{\ltimes}(X)) = \mathcal{A}_{\ltimes}(X)$, $\mathfrak{L}_{\ltimes}(\mathcal{A}_{\ltimes}(X)) = \mathfrak{L}_{\ltimes}(X)$. $\mathcal{F}_{\ltimes}(\mathcal{A}_{\ltimes}(X)) = \mathcal{F}_{\ltimes}(X)$,

Preuve Pour ce dernier point, si $\mathfrak{L}_A(|A|) \notin Im(\mathfrak{L}_B)$. On a bien $\mathcal{F}_{\ltimes}(\mathcal{A}_{\ltimes}(X)) =< \mathcal{F}(A), B, \mathcal{F}(S) >$.

□

Définition 9.3 *Avec les mêmes hypothèses, on définit* $\alpha_{\ltimes X}$ *de* $\mathcal{C}_\ltimes[X, \mathcal{A}_\ltimes(X)]$.

– *Si* $A = \mathbf{1}$ *ou* $\mathfrak{L}_A(|A|) \in Im(\mathfrak{L}_B)$, *on pose*

$$\boxed{\alpha_{\ltimes X} = id_X}$$

– *Sinon, on a* $p_A \circ S = \mathcal{F}(S)$, *donc* $id_B \curlywedge_C S$ *est bien défini. On note* ψ_X *ce morphisme. On pose*

$$\boxed{\alpha_{\ltimes X} =< id_A, \psi_X >}$$

D'après la définition de ψ_X *vérifie* $id_A \circ S = S' \circ \psi_X$. *C'est donc bien un morphisme de* $\mathcal{C}_\ltimes[X, \mathcal{A}_\ltimes(X)]$.

Lemme 9.2 $\alpha_{\ltimes \mathcal{A}_\ltimes(X)} = id_{\mathcal{A}_\ltimes(X)}$ *et* $p_{\ltimes \mathcal{A}_\ltimes(X)} \circ \alpha_{\ltimes X} = p_{\ltimes X}$

Preuve Pour ce dernier point, on doit montrer $< p_A, p_C > \circ < id_A, \psi_X >=< p_A, id_B >$. Or par définition de ψ_X, $p_C \circ \psi_X = id_B$.

□

Lemme 9.3 *Soient* $X =< A_1, B_1, S_1 >$ *et* $Y =< A_2, B_2, S_2 >$ *de* $Ob(\mathcal{C}_\ltimes)$, *tels que* $Y \neq \mathbf{1}_\ltimes$, *et soit* $F =< g, h >$ *de* $\mathcal{C}_\ltimes[X, \mathcal{F}_\ltimes(Y)]$.

$\mathcal{A}_\ltimes(F *^a_\ltimes Y) = F *_\ltimes \mathcal{A}(Y)$ *et* $F *_\ltimes \mathcal{A}(Y) = F *_\ltimes Y$ *implique* $\mathcal{A}(Y) = Y$

Preuve

1) Si $\mathfrak{L}_{A_2}(|A_2|) = \mathfrak{L}_{B_2}(|B_2|)$, alors $Y = \mathcal{A}_\ltimes(Y)$ et

$$\begin{aligned}
&\mathcal{A}_\ltimes(F *^a_\ltimes Y)\\
&\quad = \mathcal{A}_\ltimes(< g *^a A_2, h *^a B_2, q(S_1, g *^a A_2) >)\\
&\quad = < g *^a A_2, h *^a B_2, q(S_1, g *^a A_2) >\\
&\quad = F *^a_\ltimes \mathcal{A}_\ltimes(Y)
\end{aligned}$$

Remarquons aussi que $\mathcal{A}_\ltimes(F *^a_\ltimes Y) = F *^a_\ltimes Y$

2) Sinon, on pose $\phi_F = S_1 \curlywedge_{g *^a A_2} S_2 \circ h$ (on rappelle que $\mathcal{F}(\phi_F) = S_1$). On a : $\mathcal{A}(Y) =< A_2, \mathcal{F}(S_2) * A_2, q(\mathcal{F}(S_2), A_2) >$.

$$\begin{aligned}
&\mathcal{A}_\ltimes(F *^a_\ltimes Y)\\
&\quad = \mathcal{A}_\ltimes(< g *^a A_2, B_1, \phi_F >)\\
&\quad = < g *^a A_2, S_1 * (g *^a A_2), q(S_1, g *^a A_2) >\\
&\quad = < g *^a A_2, (g \circ S_1) *^a A_2, q(S_1, g *^a A_2) >\\
&\quad = < g *^a A_2, (\mathcal{F}(S_2) \circ h) *^a A_2, q(S_1, g *^a A_2) >\\
&\quad = < g *^a A_2, h *^a (\mathcal{F}(S_2) * A_2), q(S_1, g *^a A_2) >\\
&\quad = F *_\ltimes \mathcal{A}(Y)
\end{aligned}$$

On remarque que dans ce cas, $\mathcal{A}_\ltimes(F *^a_\ltimes Y) \neq F *^a_\ltimes Y$, car $S_1 * (g *^a A_2) \neq B_1$ (en effet $|S_1 * (g *^a A_2)| > |A_1| \leq |B_1|$).

On en déduit que $\mathcal{A}_\ltimes(F *^a_\ltimes Y) = F *_\ltimes \mathcal{A}(Y)$, et que $Y \neq \mathcal{A}(Y)$ implique $\mathcal{A}_\ltimes(F *^a_\ltimes Y) \neq F *^a_\ltimes Y$.

□

Lemme 9.4

$$Q^a_\ltimes(F, \mathcal{A}_\ltimes(Y)) \circ \alpha_{\ltimes F *^a_\ltimes Y} = \alpha_{\ltimes Y} \circ Q^a_\ltimes(F, Y)$$

Preuve

1) Si $\mathfrak{L}_{A_2}(|A_2|) = \mathfrak{L}_{B_2}(|B_2|)$ alors $\mathcal{A}_{\ltimes}(Y) = Y$, $\alpha_{\ltimes Y} = id_Y$ et $\alpha_{\ltimes F *^a_{\ltimes} Y} = id_{F *^a_{\ltimes} Y}$. L'égalité à démontrer se réduit donc à $Q^a_{\ltimes}(F,Y) = Q^a_{\ltimes}(F,Y)$.

2) Sinon, $\mathcal{A}_{\ltimes}(Y) =< A_2, \mathcal{F}(S_2) * A_2, q(\mathcal{F}(S_2), A_2) >$, $\alpha_{\ltimes Y} =< id_{A_2}, id_{B_2} \curlywedge_{\mathcal{F}(S_2)*A_2} S_2 >$ et $\alpha_{\ltimes F *^a_{\ltimes} Y} =< id_{g *^a A_2}, id_{B_1} \curlywedge_{S_1 * (g *^a A_2)} (S_1 \curlywedge_{g *^a A_2} S_2 \circ h) >$.

On pose $\phi_1 = id_{B_1} \curlywedge_{S_1 * (g *^a A_2)} (S_1 \curlywedge_{g *^a A_2} S_2 \circ h)$, et
$\phi_2 = id_{B_1} \curlywedge_{h *^a (\mathcal{F}(S_2) * A_2)} (h \curlywedge_{\mathcal{F}(S_2)*A_2} S_2 \circ h)$. On montre que $\phi_1 = \phi_2$.

On rappelle que $S_1 * (g *^a A_2) = h *^a (\mathcal{F}(S_2) * A_2)$. Il suffit en fait simplement de montrer que $q(S_1, (g *^a A_2)) \circ \phi_2 = S_1 \curlywedge_{g *^a A_2} S_2 \circ h$.

Premièrement,

$$\begin{aligned} & p_{g *^a A_2} \circ (q(S_1, (g *^a A_2)) \circ \phi_2) \\ & \quad = S_1 \circ p_{S_1 * (g *^a A_2)} \circ \phi_2 \\ & \quad = S_1 \circ p_{h *^a (\mathcal{F}(S_2) * A_2)} \circ \phi_2 \\ & \quad = S_1 \end{aligned}$$

Deuxièment,

$$\begin{aligned} & q^a(g, A_2) \circ (q(S_1, (g *^a A_2)) \circ \phi_2) \\ & \quad = q^a(g \circ S_1, A_2) \circ \phi_2 \\ & \quad = q^a(\mathcal{F}(S_2) \circ h, A_2) \circ \phi_2 \\ & \quad = q(\mathcal{F}(S_2), A_2) \circ q^a(h, \mathcal{F}(S_2) * A_2) \circ \phi_2 \\ & \quad = S_2 \circ h \end{aligned}$$

On en déduit que $\phi_1 = \phi_2$ et en particulier :

$$\begin{aligned} & Q^a_{\ltimes}(F, \mathcal{A}_{\ltimes}(Y)) \circ \alpha_{\ltimes F *^a_{\ltimes} Y} \\ & \quad =< q^a(g, A_2), q^a(h, \mathcal{F}(S_2) * A_2) > \circ < id_{g *^a A_2}, \phi_2 > \\ & \quad =< q^a(g, A_2), (h \curlywedge_{\mathcal{F}(S_2)*A_2} S_2 \circ h) > \\ & \quad =< q^a(g, A_2), (id_{B_2} \curlywedge_{\mathcal{F}(S_2)*A_2} S_2) \circ h > \\ & \quad =< id_{A_2}, id_{B_2} \curlywedge_{\mathcal{F}(S_2)*A_2} S_2 > \circ < q^a(g, A_2), h > \\ & \quad = \alpha_{\ltimes Y} \circ Q^a_{\ltimes}(F, Y) \end{aligned}$$

□

9.1.1.b Construction et propriétés de $\mathcal{H}_{\ltimes}$ et $\mathbb{G}_{\ltimes}$

Définition 9.4

On définit les fonctions suivantes :
- $\mathcal{H}_{\ltimes}(<A, B, S>) = B_{\ltimes}$ *(ie* $=<B, B, id_B>$*)*
- $\mathcal{H}_{\ltimes}(< g, h >) = h_{\ltimes}$ *(ie* $=< h, h >$*)*
- $\mathbb{G}_{\ltimes}(< A, B, S >) =< S, id_B >$

$$\begin{tikzcd} B \arrow[r, "S"] & A \\ B \arrow[u, "id_B"] \arrow[r, "id_B"'] & B \arrow[u, "S"'] \end{tikzcd} \qquad \mathcal{H}_{\ltimes}(X) \ \text{(gauche)}, \ X \ \text{(droite)}$$

Lemme 9.5 *$\mathcal{H}_{\ltimes}$ est un endofoncteur et $\mathbb{G}_{\ltimes}$ est une transformation naturelle de $\mathcal{H}_{\ltimes}$ vers $Id_{\mathcal{C}_{\ltimes}}$. De plus,*

1) $\mathcal{H}_{\ltimes}(\mathbf{1}_{\ltimes}) = \mathbf{1}_{\ltimes}$ *et* $\mathbb{G}_{\ltimes}(\mathbf{1}_{\ltimes}) = id_{\mathbf{1}_{\ltimes}}$.

2) *Pour tout X, si $X = \mathcal{A}_{\ltimes}(X) \neq \mathbf{1}_{\ltimes}$, $\mathcal{H}_{\ltimes}(X) = \mathbb{G}_{\ltimes}(\mathcal{F}_{\ltimes}(X)) *_{\ltimes} X$, $\mathcal{H}_{\ltimes}(p_{\ltimes X}) = p_{\mathcal{H}_{\ltimes}(X)}$, $\mathbb{G}_{\ltimes}(X) = Q_{\ltimes}(\mathbb{G}_{\ltimes}(\mathcal{F}_{\ltimes}(X)), X)$*

3) *Pour tout* X, *si* $X \neq \mathcal{A}_\ltimes(X)$, $\mathcal{H}_\ltimes(X) = \mathcal{H}_\ltimes(\mathcal{F}_\ltimes(X))$ *et* $\mathcal{H}_\ltimes(p_{\ltimes X}) = id_{\mathcal{H}_\ltimes(X)}$.

Preuve Il est immédiat de vérifier que $\mathcal{H}_\ltimes$ est un foncteur.
On vérifie que $\mathbb{G}_\ltimes$ est une transformation naturelle. Soient $X =< A_1, B_1, S_1 >$ et $Y =< A_2, B_2, S_2 >$ deux objets de $\mathcal{C}_\ltimes$, et $F =< g, h >$ de $\mathcal{C}_\ltimes[X, Y]$.
On a :

$$\begin{aligned} & F \circ \mathbb{G}_\ltimes(X) \\ & \quad =< g \circ S_1, h \circ id_{B_1} > \\ & \quad =< S_2 \circ h, id_{B_2} \circ h > \\ & \quad = \mathbb{G}_\ltimes(Y) \circ \mathcal{H}_\ltimes(F) \end{aligned}$$

$$\begin{tikzcd} \mathcal{H}_\ltimes(X) \arrow[r, "\mathbb{G}_\ltimes(X)"] \arrow[d, "\mathcal{H}_\ltimes(F)"'] & X \arrow[d, "F"] \\ \mathcal{H}_\ltimes(Y) \arrow[r, "\mathbb{G}_\ltimes(Y)"'] & Y \end{tikzcd}$$

On vérifient maintenant les points 1, 2 et 3 ci-dessus.

1) $\mathcal{H}_\ltimes(\mathbf{1}_\ltimes) =< \mathbf{1}, \mathbf{1}, id_\mathbf{1} >$ et $\mathbb{G}_\ltimes(\mathbf{1}_\ltimes) =< id_\mathbf{1}, id_\mathbf{1} >$.

2) Soit $X =< A, B, S >$ tel que $X = \mathcal{A}_\ltimes(X) \neq \mathbf{1}_\ltimes$: On a $\mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$.

$$\begin{aligned} & \mathbb{G}_\ltimes(\mathcal{F}_\ltimes(X)) *_\ltimes X \\ & \quad =< \mathcal{F}(S), id_{\mathcal{F}(B_1)} > *_\ltimes < A, B, S > \\ & \quad =< \mathcal{F}(S) * A, B, q(id_{\mathcal{F}(B)}, \mathcal{F}(S) * A) > \\ & \quad =< B, B, id_B > \\ & \quad = \mathcal{H}_\ltimes(X) \end{aligned}$$

De plus, $\mathcal{H}_\ltimes(p_{\ltimes X}) =< p_B, p_B >= p_{\mathcal{H}_\ltimes(X)}$. Et,

$$\begin{aligned} & Q_\ltimes(\mathbb{G}_\ltimes(\mathcal{F}_\ltimes(X)), X) \\ & \quad =< q(\mathcal{F}(S), A), q(id_{\mathcal{F}(B)}, B) > \\ & \quad =< S, id_B > \end{aligned}$$

3) Soit $X =< A, B, S >$ tel que $X = \mathcal{A}_\ltimes(X) \neq \mathbf{1}_\ltimes$: On a $\mathfrak{L}_A(|A|) \notin Im(\mathfrak{L}_B)$: $\mathcal{F}_\ltimes(X) =< \mathcal{F}(A), B, \mathcal{F}(S) >$. Donc $\mathcal{H}_\ltimes(X) =< B, B, id_B >= \mathcal{H}_\ltimes(\mathcal{F}_\ltimes(X))$. De même, $\mathcal{H}_\ltimes(p_{\ltimes X}) = id_{\mathcal{H}_\ltimes(X)}$.

□

9.1.1.c Construction et propriétés de $\mathcal{M}_\ltimes$

Définition 9.5 *Soient un objet* $X =< A, B, S >$ *tel que* $X = \mathcal{A}(X)$ *et un morphisme* F *de* $\mathcal{C}_\ltimes[\mathcal{H}_\ltimes(\mathcal{F}_\ltimes(X)), \mathcal{H}_\ltimes(X)]$ *tel que* $p_{\ltimes \mathcal{H}(X)} \circ F = id_{\mathcal{H}_\ltimes(\mathcal{F}_\ltimes(X))}$.

On a $B = \mathcal{F}(S) * A$, $S = q(\mathcal{F}(S), A)$ *et* $F =< s, s >$ *avec* s *de* $\mathcal{C}[\mathcal{F}(B), B]$ *tel que* $p_B \circ s = id_{\mathcal{F}(B)}$. *Donc* $p_A \circ (q(\mathcal{F}(S), A) \circ s) = \mathcal{F}(S)$.

[Soit $S' = S; \mathbb{G}[A, B](S) \circ s$.**]** *Donc* $S' \in [A \ltimes \mathcal{F}(B)]$.

On pose

$$\boxed{\mathcal{M}_\ltimes(F) =< A, \mathcal{F}(B), S' >}$$

On a bien $\mathcal{A}_\ltimes(\mathcal{M}_\ltimes(F)) = X$.
$\mathcal{A}_\ltimes(\mathcal{M}_\ltimes(F)) \neq \mathcal{M}_\ltimes(F)$ *car* $\mathcal{F}(B) \neq B$.

Lemme 9.6 *Sous les hypothèses de la définition précédente :*

$$\alpha_{\ltimes \mathcal{M}_\ltimes(F)} \circ \mathbb{G}_\ltimes(\mathcal{M}_\ltimes(F)) = \mathbb{G}(A) \circ F$$

Preuve

$$\begin{aligned}&\alpha_{\ltimes\,\mathcal{M}_\ltimes(F)} \circ \mathbb{G}_\ltimes(\mathcal{M}_\ltimes(F))\\ &\qquad = < id_A, id_{\mathcal{F}(B)} \curlywedge_{\mathcal{F}(S)*A} S' > \circ < S', id_{\mathcal{F}(B)} >\\ &\qquad = < S \circ s, s >\end{aligned}$$

En effet, $id_{\mathcal{F}(B)} \curlywedge_{\mathcal{F}(S)*A} S'$ est le seul morphisme ψ à vérifier $p_B \circ \psi = id_{\mathcal{F}(B)}$ et $S \circ \psi = S' = S \circ s$. Or s vérifie ces égalités. On en conclue que

$$\alpha_{\ltimes\,\mathcal{M}_\ltimes(F)} \circ \mathbb{G}_\ltimes(\mathcal{M}_\ltimes(F)) =< S, id_B > \circ F = \mathbb{G}(A) \circ F$$

□

Lemme 9.7 *Avec les mêmes hypothèses, soit $Y \neq X$ tel que $\mathcal{A}_\ltimes(Y) = X$ et $\alpha_{\ltimes Y} \circ \mathbb{G}_\ltimes(Y) = \mathbb{G}(A) \circ F$. On a $Y = \mathcal{M}_\ltimes(F)$.*

Preuve On a $Y =< A, \mathcal{F}(B), S'' >$. Il faut donc montrer que $S'' = S \circ s$.
On le déduit directement de la première composante de $\alpha_{\ltimes Y} \circ \mathbb{G}_\ltimes(Y) = \mathbb{G}(A) \circ F$.

□

Théorème 9.1 $\mathfrak{C}_\ltimes =< \mathcal{C}_\ltimes, \mathtt{A}, \mathcal{F}_\ltimes, \mathfrak{L}_\ltimes, \mathbf{1}_\ltimes, p_\ltimes, *_\ltimes, Q_\ltimes, \mathcal{A}_\ltimes, \alpha_\ltimes, \mathcal{H}_\ltimes, \mathbb{G}_\ltimes, \mathcal{M}_\ltimes >$ *est une catégorie de générateurs d'enregistrements.*

9.1.2 Propriétés de base

Définition 9.6 *Soit A un objet de $\mathcal{C}$. On définit l'ensemble $\mathfrak{M}_A$ des* champs manifestes *de A (c'est-à-dire l'ensemble des étiquettes associées à un champ manifeste dans A) :*

$$\mathfrak{M}_A = \{\mathfrak{L}_A(i) \;/\; \mathcal{A}(\mathcal{F}^{|A|-i}(A)) \neq \mathcal{F}^{|A|-i}(A)\}$$

Lemme 9.8 *Pour tous A et B de $Ob(\mathcal{C})$, pour tout a de $\mathtt{A}$ avec $a \notin Im(\mathfrak{L}_A)$, pour tout f de $\mathcal{C}[A, \mathcal{F}(B)]$,*

$$\mathcal{A}(B) = B \Longleftrightarrow f *^a B = \mathcal{A}(f *^a B)$$

Preuve Par l'axiome 5, $\mathcal{A}(f *^a B) = f *^a \mathcal{A}(B)$. Donc le sens $\Rightarrow$ découle de la compatibilité de "$*$" avec l'égalité, et le sens $\Leftarrow$ découle directement de l'axiome 6.

□

Lemme 9.9 *Pour tous A, B de $Ob(\mathcal{C})$, a de $\mathtt{A}$ tel que $a \notin Im(\mathfrak{L}_A)$, pour tout f de $\mathcal{C}[A, \mathcal{F}(B)]$,*

$$\alpha_{f*^a B} = p_{f*^a B} \curlywedge_{f*^a \mathcal{A}(B)} (\alpha_B \circ q^a(f, B))$$

Preuve C'est une reformulation des axiomes 3 et 5 des CG :

$$p_{f*^a\mathcal{A}(B)} \circ \alpha_{f*^a B} = p_{f*^a B} \qquad \text{et} \qquad q^a(f, \mathcal{A}(B)) \circ \alpha_{f*^a B} = \alpha_B \circ q^a(f, B)$$

□

9.1.2.a Propriétés de $\mathcal{H}$ et $\mathbb{G}$

On va maintenant détailler les propriétés de $\mathcal{H}$ et $\mathbb{G}$ pour vérifier qu'elles correspondent à l'intuition donnée en introduction de cette section.

Lemme 9.10 *Pour tout A, $\mathcal{H}(A)$ est complètement abstrait (c'est-à-dire $\mathfrak{M}_{\mathcal{H}(A)} = \emptyset$), et l'ensemble des étiquettes de $\mathcal{H}(A)$ correspond à l'ensemble des champs abstraits de A (c'est-à-dire $\mathfrak{L}[A, \mathcal{H}(A)] = \mathfrak{M}_A$).*

Preuve Immédiat par induction sur $|A|$.

□

Lemme 9.11 *Pour tout A tel que $\mathfrak{M}_A = \emptyset$, alors $\mathcal{H}(A) = A$ et $\mathbb{G}(A) = id_A$.*

Preuve Par induction sur $|A|$.

1) si $A = \mathbf{1}$, évident

2) sinon $A = \mathcal{A}(A)$, et $\mathcal{H}(A) = \mathbb{G}(\mathcal{F}(A)) * A$ et $\mathbb{G}(A) = q(\mathbb{G}(\mathcal{F}(A)), A)$. Or $\mathfrak{M}_{\mathcal{F}(A)} = \emptyset$, donc par hypothèse de récurrence, $\mathbb{G}(\mathcal{F}(A)) = id_{\mathcal{F}(A)}$. On en déduit que $\mathcal{H}(A) = A$ et $\mathbb{G}(A) = id_A$

□

Lemme 9.12 *Soient A et B dans $Ob(\mathcal{C})$ et f dans $\mathcal{C}[A, B]$. Si $\mathfrak{M}_A = \emptyset$ et $\mathfrak{M}_B = \emptyset$, alors*

$$\mathcal{H}(f) = f$$

Preuve Par propriété de transformation naturelle, $\mathbb{G}(B) \circ \mathcal{H}(f) = f \circ \mathbb{G}(A)$. Or avec les hypothèses du lemme, $\mathbb{G}(A) = id_A$ et $\mathbb{G}(B) = id_B$. Donc $\mathcal{H}(f) = f$.

□

Lemme 9.13 *Pour tout A,*

$$\mathcal{H}(\mathcal{H}(A)) = \mathcal{H}(A) \qquad \text{et} \qquad \mathcal{H}(\mathbb{G}(A)) = id_{\mathcal{H}(A)}$$

Preuve $\mathcal{H}(\mathcal{H}(A)) = \mathcal{H}(A)$ se déduit des lemmes précédents. Pour montrer $\mathcal{H}(\mathbb{G}(A)) = id_{\mathcal{H}(A)}$, on raisonne par induction sur $|A|$:

1) si $A = \mathbf{1}$, alors $\mathbb{G}(A) = id_{\mathbf{1}}$, donc $\mathcal{H}(\mathbb{G}(A)) = id_{\mathbf{1}}$.

2) si $A = \mathcal{A}(A) \neq \mathbf{1}$, Par la propriété de transformation naturelle et comme $\mathbb{G}(\mathcal{H}(A)) = id_{\mathcal{H}(A)}$

$$\mathbb{G}(A) \circ \mathcal{H}(\mathbb{G}(A)) = \mathbb{G}(A)$$

Or $\mathcal{H}(p_A) = p_{\mathcal{H}(A)}$, donc

$$\begin{aligned} & p_{\mathcal{H}(A)} \circ \mathcal{H}(\mathbb{G}(A)) \\ &= \mathcal{H}(p_A \circ \mathbb{G}(A)) \\ &= \mathcal{H}(\mathbb{G}(\mathcal{F}(A)) \circ \mathcal{H}(p_A)) \\ &= \mathcal{H}(\mathbb{G}(\mathcal{F}(A))) \circ \mathcal{H}(p_A) \end{aligned}$$

En appliquant l'hypothèse de récurrence, on trouve finalement que

$$p_{\mathcal{H}(A)} \circ \mathcal{H}(\mathbb{G}(A)) = p_{\mathcal{H}(A)}$$

Or $\mathcal{H}(A) = \mathbb{G}(\mathcal{F}(A)) * A$ et $\mathbb{G}(A) = q(\mathbb{G}(\mathcal{F}(A)), A)$, donc
$\mathcal{H}(\mathbb{G}(A)) = p_{\mathcal{H}(A)} \curlywedge_{\mathcal{H}(A)} q(\mathbb{G}(\mathcal{F}(A)), A)$.

Comme $id_{\mathcal{H}(A)}$ est le seul morphisme ϕ à vérifier

$$p_{\mathcal{H}(A)} \circ \phi = p_{\mathcal{H}(A)} \qquad \text{et} \qquad q(\mathbb{G}(\mathcal{F}(A)), A) \circ \phi = q(\mathbb{G}(\mathcal{F}(A)), A)$$

on en déduit

$$\mathcal{H}(\mathbb{G}(A)) = id_{\mathcal{H}(A)}$$

3) sinon, $\mathcal{H}(A) = \mathcal{H}(\mathcal{F}(A))$, $\mathcal{H}(p_A) = id_{\mathcal{H}(A)}$, donc $p_A \circ \mathbb{G}(A) = \mathbb{G}(\mathcal{F}(A))$. Et, $\mathcal{H}(p_A) \circ \mathcal{H}(\mathbb{G}(A)) = \mathcal{H}(\mathbb{G}(\mathcal{F}(A)))$. En appliquant l'hypothèse de récurrence, on trouve $\mathcal{H}(\mathbb{G}(A)) = id_{\mathcal{H}(A)}$.

□

Théorème 9.2 *Pour tous A, B de $Ob(\mathcal{C})$, f de $\mathcal{C}[A, \mathcal{F}(B)]$, et a de $\mathfrak{A}$, tel que $a \notin Im(\mathfrak{L}_A)$.*

1) *Si $B = \mathcal{A}(B)$, alors*

$$\mathcal{H}(f *^a B) = \mathcal{H}(f) *^a \mathcal{H}(B) \qquad \textit{et} \qquad \mathcal{H}(q^a(f, B)) = q^a(\mathcal{H}(f), \mathcal{H}(B))$$

2) *Si $B \neq \mathcal{A}(B)$, alors*

$$\mathcal{H}(f *^a B) = \mathcal{H}(A) \qquad \textit{et} \qquad \mathcal{H}(q^a(f, B)) = \mathcal{H}(f)$$

Preuve

1) Si $B = \mathcal{A}(B)$, alors $f *^a B = \mathcal{A}(f *^a B)$, donc

$$\begin{aligned} &\mathcal{H}(f *^a B) \\ &\quad = \mathbb{G}(A) * (f *^a B) \\ &\quad = (f \circ \mathbb{G}(A)) *^a B \\ &\quad = (\mathbb{G}(B) \circ \mathcal{H}(f)) *^a B \\ &\quad = \mathcal{H}(f) *^a (\mathbb{G}(B) * B) \\ &\quad = \mathcal{H}(f) *^a \mathcal{H}(B) \end{aligned} \qquad \text{et} \qquad \begin{aligned} &q^a(\mathcal{H}(f), \mathcal{H}(B)) \\ &\quad = \mathcal{H}(q^a(\mathcal{H}(f), \mathcal{H}(B))) \\ &\quad = \mathcal{H}(q^a(\mathbb{G}(B) \circ \mathcal{H}(f), B)) \\ &\quad = \mathcal{H}(q^a(f \circ \mathbb{G}(A), B)) \\ &\quad = \mathcal{H}(q^a(f, B) \circ q(\mathbb{G}(A), f *^a B)) \\ &\quad = \mathcal{H}(q^a(f, B)) \circ \mathcal{H}(\mathbb{G}(f *^a B)) \\ &\quad = \mathcal{H}(q^a(f, B)) \end{aligned}$$

2) Sinon, $B \neq \mathcal{A}(B)$ et $f *^a B \neq \mathcal{A}(f *^a B)$. Donc $\mathcal{H}(B) = \mathcal{H}(\mathcal{F}(B))$, $p_B = id_{\mathcal{H}(B)}$, $\mathcal{H}(f *^a B) = \mathcal{H}(A)$ et $p_{\mathcal{H}(f *^a B)} = id_{\mathcal{H}(A)}$. Comme $p_B \circ q^a(f, B) = f \circ p_{f *^a B}$, on en déduit que $\mathcal{H}(q^a(f, B)) = \mathcal{H}(f)$.

□

La définition suivante affirme que pour tout A, le morphisme $\mathbb{G}(A)$ *se décompose en définitions élémentaires* (voir section 8.5). Intuitivement c'est plutôt réconfortant, puisque cela dit que $\mathbb{G}(A)$ définit les champs de A à partir de ceux de $\mathcal{H}(A)$, ce qui est la sémantique de $\mathbb{G}(A)$.

Définition 9.7 *Soit A de $\mathcal{C}$. On définit la suite n_i, qui vérifie :*

- $n_0 = -1$.
- $n_{i+1} = \max\{k = |A| - \mathfrak{L}_A^{-1}(a) \ / \ k > n_i \wedge a \in \mathfrak{M}_A\}$.

On pose $S_A = (\mathbb{G}(\mathcal{F}^{n_i}(A)))_{1 \leq i}$.

Lemme 9.14 *Pour tout A, $S_A \in [A \ltimes \mathcal{H}(A)]$, et $\mathbb{G}[A, \mathcal{H}(A)](S_A) = \mathbb{G}(A)$*

Preuve Par induction sur $|A|$.

1) Si $A = \mathbf{1}$ alors évident.

2) Si $A = \mathcal{A}(A)$, alors $S_A = S_{\mathcal{F}(A)}$ et $\mathcal{H}(A) = \mathbb{G}(\mathcal{F}(A)) * A$. Par hypothèse de récurrence, $S_A = S_{\mathcal{F}(A)} \in [\mathcal{F}(A) \ltimes \mathcal{H}(\mathcal{F}(A))]$, et $\mathbb{G}(\mathcal{F}(A)) = \mathbb{G}[\mathcal{F}(A), \mathcal{H}(\mathcal{F}(A))](S_A)$. Comme, $\mathfrak{L}_A(|A|) = \mathfrak{L}_{\mathcal{H}(A)}(|\mathcal{H}(A)|)$, on en déduit que $S_A \in [A \ltimes \mathcal{H}(A)]$. De plus,

$$\begin{aligned}
&\mathbb{G}(A)\\
&\quad = q(\mathbb{G}(\mathcal{F}(A)), A)\\
&\quad = q(\mathbb{G}[\mathcal{F}(A), \mathcal{H}(\mathcal{F}(A))](S_A), A)\\
&\quad = \mathbb{G}[A, \mathcal{H}(A)](S_A)
\end{aligned}$$

3) Sinon, $S_A = S_{\mathcal{F}(A)}; \mathbb{G}(A)$, $\mathcal{H}(A) = \mathcal{H}(\mathcal{F}(A))$ et $p_A \circ \mathbb{G}(A) = \mathbb{G}(\mathcal{F}(A))$. Par hypothèse de récurrence, $S_{\mathcal{F}(A)} \in [\mathcal{F}(A) \ltimes \mathcal{H}(A)]$, et $\mathbb{G}(\mathcal{F}(A)) = \mathbb{G}[\mathcal{F}(A), \mathcal{H}(A)](S_{\mathcal{F}(A)})$. Comme, $\mathfrak{L}_A(|A|) \notin Im(\mathfrak{L}_{\mathcal{F}(A)}) \supseteq Im(\mathfrak{L}_{\mathcal{H}(A)})$, on en déduit que $S_A \in [A \ltimes \mathcal{H}(A)]$ et $\mathbb{G}[A, \mathcal{H}(A)](S_A) = \mathbb{G}(A)$.

□

9.1.2.b Propriétés de $\mathcal{M}$

Le lemme ci-dessous montre que tout objet A de cette catégorie est soit abstrait (ie $A = \mathcal{A}(A)$), soit est le manifeste d'une certaine section, qui est en fait $\mathcal{H}(\alpha_A)$. Le lemme suivant montre alors que $\mathcal{H}(\alpha_A)$ est l'unique section dont A est le manifeste.

Lemme 9.15 *Soit A tel que $A \neq \mathcal{A}(A)$. Le morphisme $\mathcal{H}(\alpha_A)$ est une section de $\mathcal{C}[\mathcal{H}(A), \mathcal{H}(\mathcal{A}(A))]$. De plus, $A = \mathcal{M}(\mathcal{H}(\alpha_A))$.*

Preuve On a bien $\mathcal{F}(\mathcal{H}(\mathcal{A}(A))) = \mathcal{H}(A)$. De plus,

$$\begin{aligned}
&p_{\mathcal{H}(\mathcal{A}(A))} \circ \mathcal{H}(\alpha_A)\\
&\quad = \mathcal{H}(p_{\mathcal{A}(A)} \circ \alpha_A)\\
&\quad = \mathcal{H}(p_A)\\
&\quad = id_{\mathcal{H}(A)}
\end{aligned}$$

$\mathcal{H}(\alpha_A)$ est une section de $\mathcal{C}[\mathcal{H}(A), \mathcal{H}(\mathcal{A}(A))]$.
De plus $\mathbb{G}(\mathcal{A}(A)) \circ \mathcal{H}(\alpha_A) = \alpha_A \circ \mathbb{G}(A)$. Donc par unicité de $\mathcal{M}(\mathcal{H}(\alpha_A))$, on a $\mathcal{M}(\mathcal{H}(\alpha_A)) = A$.

□

Lemme 9.16 *Soit A tel que $\mathcal{A}(A) = A$ et s une section de $\mathcal{C}[\mathcal{H}(\mathcal{F}(A)), \mathcal{H}(A)]$. On a $s = \mathcal{H}(\alpha_{\mathcal{M}(s)})$.*

Preuve On a $p_{\mathcal{H}(A)} \circ s = id_{\mathcal{H}(\mathcal{M}(s))}$ et $\mathbb{G}(A) = q(\mathbb{G}(\mathcal{F}(A)), A)$ Donc $s = id_{\mathcal{H}(\mathcal{M}(s))} \curlywedge_{\mathcal{H}(A)} G(A) \circ s$. Or, par le lemme qui précède $p_{\mathcal{H}(A)} \circ \mathcal{H}(\alpha_{\mathcal{M}(s)}) = id_{\mathcal{H}(\mathcal{M}(s))}$.
De plus,

$$\begin{aligned}
&q(\mathbb{G}(\mathcal{F}(A)), A) \circ \mathcal{H}(\alpha_{\mathcal{M}(s)})\\
&\quad = G(A) \circ \mathcal{H}(\alpha_{\mathcal{M}(s)})\\
&\quad = G(\mathcal{A}(\mathcal{M}(s))) \circ \mathcal{H}(\alpha_{\mathcal{M}(s)})\\
&\quad = \alpha_{\mathcal{M}(s)} \circ G(\mathcal{M}(s))\\
&\quad = G(A) \circ s
\end{aligned}$$

Donc $s = \mathcal{H}(\alpha_{\mathcal{M}(s)})$

□

Théorème 9.3 *Pour tout A, si $A \neq \mathcal{A}(A)$ alors $\mathcal{H}(\alpha_A)$ est l'unique section s de $\mathcal{C}[\mathcal{H}(\mathcal{F}(A)), \mathcal{H}(A)]$ telle que $A = \mathcal{M}(s)$.*

Preuve Découle directement des 2 lemmes précédents.

□

9.1.3 Foncteurs de CG & Renommage

Lemme 9.17 *Pour* $i \in [1,2]$, *soit* $\mathfrak{C}_i =< \mathcal{C}_i, \mathfrak{A}_i, \mathcal{F}_i, \mathfrak{L}_i, \mathbf{1}_i, p_i, *_i, q_i, \mathcal{A}_i, \alpha_i, \mathcal{H}_i, \mathbb{G}_i, \mathcal{M}_i >$ *deux catégories de générateurs d'enregistrements. Soit* $< F, L >$ *un foncteur de catégories de signatures d'enregistrements, tel que pour tout* A *de* $Ob(\mathcal{C}_1)$ *:*

$$F(\mathcal{A}_1(A)) = \mathcal{A}_2(F(A)) \qquad et \qquad F(\alpha_{1A}) = \alpha_{2F(A)}$$

Alors, pour tout A *de* $Ob(\mathcal{C}_1)$, $\mathcal{H}_2(F(A)) = F(\mathcal{H}_1(A))$ *et* $\mathbb{G}_2(F(A)) = F(\mathbb{G}_1(A))$.

Preuve On procède par récurrence sur $|A|$.
Pour $A = \mathbf{1}_1$, c'est trivial.
Pour $A = \mathcal{A}_1(A) \neq \mathbf{1}_1$, par récurrence on a $F(\mathcal{H}_1(\mathcal{F}_1(A))) = \mathcal{H}_2(\mathcal{F}_2(F(A)))$ et $F(\mathbb{G}_1(\mathcal{F}_1(A))) = \mathbb{G}_2(\mathcal{F}_2(F(A)))$. Comme $\mathcal{A}_2(F(A)) = F(A)$, on en déduit :

$$\begin{aligned}
&F(\mathcal{H}_1(A)) \\
&\quad = F(\mathbb{G}_1(\mathcal{F}_1(A)) *_1 A) \\
&\quad = F(\mathbb{G}_1(\mathcal{F}_1(A))) *_2 F(A) \\
&\quad = \mathbb{G}_2(\mathcal{F}_2(F(A))) *_2 F(A) \\
&\quad = \mathcal{H}_2(F(A))
\end{aligned}$$

et de même $F(\mathbb{G}_1(A)) = F(q_1(\mathbb{G}_1(\mathcal{F}_1(A)), A)) = G_2(F(A))$.
Pour $A \neq \mathcal{A}_1(A)$, comme $\mathcal{A}_2(F(A)) \neq F(A)$, on en déduit :

$$\begin{aligned}
&F(\mathcal{H}_1(A)) \\
&\quad = F(\mathcal{H}_1(\mathcal{F}_1(A))) \\
&\quad = \mathcal{H}_2(\mathcal{F}_2(F(A))) \\
&\quad = \mathcal{H}_2(F(A))
\end{aligned}$$

Montrons que $\mathcal{H}_2(F(\mathbb{G}_1(A))) = id_{\mathcal{H}_2(\mathcal{F}_2(F(A)))}$. En effet, $\mathcal{H}_2(p_{2F(A)}) = id_{\mathcal{F}_2(F(A))}$, donc

$$\begin{aligned}
&\mathcal{H}_2(F(\mathbb{G}_1(A))) \\
&\quad = \mathcal{H}_2(p_{2F(A)}) \circ \mathcal{H}_2(F(\mathbb{G}_1(A))) \\
&\quad = \mathcal{H}_2(p_{2F(A)} \circ F(\mathbb{G}_1(A))) \\
&\quad = \mathcal{H}_2(F(p_{1A} \circ \mathbb{G}_1(A))) \\
&\quad = \mathcal{H}_2(F(\mathbb{G}_1(\mathcal{F}_1(A))))
\end{aligned}$$

Par récurrence, $F(\mathbb{G}_1(\mathcal{F}(A))) = \mathbb{G}_2(F(\mathcal{F}_1(A)))$. On en déduit donc que

$$\begin{aligned}
&\mathcal{H}_2(F(\mathbb{G}_1(A))) \\
&\quad = \mathcal{H}_2(\mathbb{G}_2(\mathcal{F}_2(F(A)))) \\
&\quad = id_{\mathcal{H}_2(\mathcal{F}_2(F(A)))}
\end{aligned}$$

En utilisant cette dernière égalité, on montre

$$\begin{aligned}
&F(\mathbb{G}_1(A)) \\
&\quad = F(\mathbb{G}_1(A)) \circ id_{\mathcal{H}_2(F(A))} \\
&\quad = F(\mathbb{G}_1(A)) \circ \mathbb{G}_2(\mathcal{H}_2(F(A))) \\
&\quad = \mathbb{G}_2(F(A)) \circ \mathcal{H}_2(F(\mathbb{G}_1(A))) \\
&\quad = \mathbb{G}_2(F(A))
\end{aligned}$$

$\square$

Le lemme précédent amène la définition suivante :

Définition 9.8 *Pour $i \in [1,2]$, soit $\mathfrak{C}_i =< \mathcal{C}_i, \mathtt{A}_i, \mathcal{F}_i, \mathfrak{L}_i, \mathbf{1}_i, p_i, *_i, q_i, \mathcal{A}_i, \alpha_i, \mathcal{H}_i, \mathbb{G}_i, \mathcal{M}_i >$ une CG. Un foncteur de CG de $\mathfrak{C}_1$ vers $\mathfrak{C}_2$ est un couple $< F, L >$ où $< F, L >$ est un foncteur de CS, tel que :*

1) $F(\mathcal{A}_1(A)) = \mathcal{A}_2(F(A))$

2) $F(\alpha_{1A}) = \alpha_{2F(A)}$

3) $F(\mathcal{H}_1(f)) = \mathcal{H}_2(F(f))$

Lemme 9.18 *Pour tout A différent de $\mathbf{1}_1$ tel que $\mathcal{A}_1(A) = A$, pour toute section s de A, alors $F(\mathcal{M}_1(s)) = \mathcal{M}_2(F(s))$.*

Preuve Remarquons que si F est un foncteur de CS, alors pour toute section s, $F(s)$ est une section. De plus, sous l'hypothèse du point 1, si $\mathcal{A}_1(A) = A$ alors $\mathcal{A}_2(F(A)) = F(A)$. Donc $\mathcal{M}_2(F(s))$ existe.
On a

$$\begin{aligned}
&\mathbb{G}_2(F(A)) \circ F(s) \\
&= F(\mathbb{G}_1(A) \circ s) \\
&= F(\alpha_{1\mathcal{M}_1(s)} \circ \mathbb{G}_1(\mathcal{M}_1(s)) \\
&= \alpha_{2F(\mathcal{M}_1(s))} \circ \mathbb{G}_2(F(\mathcal{M}_1(s)))
\end{aligned}$$

Par unicité de $\mathcal{M}_2(F(s))$, on en déduit que $\mathcal{M}_2(F(s)) = F(\mathcal{M}_1(s))$.

□

Théorème 9.4 *Si $\mathfrak{C}$ est une CG, alors pour toute bijection κ de $\mathtt{A}$ dans $\mathtt{A}$, le foncteur de CS $< \mathcal{R}_\kappa, \kappa >$ est un foncteur de CG.*

De plus pour tout A, $\mathbb{R}_\kappa(\mathcal{H}(A)) = \mathcal{H}(\mathbb{R}_\kappa(A))$.

Preuve

1) $\mathcal{R}_\kappa(\mathcal{A}(A)) = id_{\mathbf{1}} *^{\kappa(\mathfrak{L}_A(1))\dots\kappa(\mathfrak{L}_A(|A|))} \mathcal{A}(A)$, donc $\mathcal{R}_\kappa(\mathcal{A}(A)) = \mathcal{A}(\mathcal{R}_\kappa(A))$.

2)

$$\begin{aligned}
&\mathcal{R}_\kappa(\alpha_A) \\
&= (\mathbb{R}_\kappa(\mathcal{A}(A)))^{-1} \circ \alpha_A \circ q^{\kappa(\mathfrak{L}_A(|A|))}(\mathbb{R}_\kappa(\mathcal{F}(A)), A) \\
&= (\mathbb{R}_\kappa(\mathcal{A}(A)))^{-1} \circ q^{\kappa(\mathfrak{L}_A(|A|))}(\mathbb{R}_\kappa(\mathcal{F}(A)), \mathcal{A}(A)) \circ \alpha_{\mathbb{R}_\kappa(\mathcal{F}(A)) *^{\kappa(\mathfrak{L}_A(|A|))} A} \\
&= (\mathbb{R}_\kappa(\mathcal{A}(A)))^{-1} \circ \mathbb{R}_\kappa(\mathcal{A}(A)) \circ \alpha_{\mathcal{R}_\kappa(A)} \\
&= \alpha_{\mathcal{R}_\kappa(A)}
\end{aligned}$$

3) Pour tout objet A, on montre que $\mathbb{R}_\kappa(\mathcal{H}(A)) = \mathcal{H}(\mathbb{R}_\kappa(A))$. On utilise le fait que étant donné les points précédents, $\mathcal{R}_\kappa(\mathcal{H}(A)) = \mathcal{H}(\mathcal{R}_\kappa(A))$ et $\mathcal{R}_\kappa(\mathbb{G}(A)) = \mathbb{G}(\mathcal{R}_\kappa(A))$. On procède par induction sur $|A|$.
Si $A = \mathbf{1}$, alors trivial.
Sinon $\mathbb{R}_\kappa(A) = q^{\kappa(\mathfrak{L}_A(|A|))}(\mathbb{R}_\kappa(\mathcal{F}(A)), A)$ et par récurrence $\mathcal{H}(\mathbb{R}_\kappa(\mathcal{F}(A))) = \mathbb{R}_\kappa(\mathcal{H}(\mathcal{F}(A)))$. Si $A \neq \mathcal{A}(A)$, alors par le théorème 9.2 $\mathcal{H}(\mathbb{R}_\kappa(A)) = \mathbb{R}_\kappa(\mathcal{H}(A))$.
Sinon $A = \mathcal{A}(A)$ et $\mathfrak{L}_A(|A|) = \mathfrak{L}_{\mathcal{H}(A)}(|\mathcal{H}(A)|)$, on a :

$$\begin{aligned}
&\mathcal{H}(\mathbb{R}_\kappa(A)) \\
&= q^{\kappa(\mathfrak{L}_A(|A|))}(\mathcal{H}(\mathbb{R}_\kappa(\mathcal{F}(A))), \mathcal{H}(A)) \\
&= q^{\kappa(\mathfrak{L}_A(|A|))}(\mathbb{R}_\kappa(\mathcal{H}(\mathcal{F}(A))), \mathcal{H}(A)) \\
&= \mathbb{R}_\kappa(\mathcal{H}(A))
\end{aligned}$$

4) Pour tout f de $\mathcal{C}[A,B]$, on montre que $\mathcal{R}_\kappa(\mathcal{H}(f)) = \mathcal{H}(\mathcal{R}_\kappa(f))$. On utilise le point précédent :

$$\begin{aligned}
&\mathcal{R}_\kappa(\mathcal{H}(f))\\
&\quad = \mathbb{R}_{\kappa^{-1}}(\mathcal{H}(\mathcal{R}_\kappa(B))) \circ \mathcal{H}(f) \circ \mathbb{R}_\kappa(\mathcal{H}(A))\\
&\quad = \mathcal{H}(\mathbb{R}_{\kappa^{-1}}(\mathcal{R}_\kappa(B))) \circ \mathcal{H}(f) \circ \mathcal{H}(\mathbb{R}_\kappa(A))\\
&\quad = \mathcal{H}(\mathbb{R}_{\kappa^{-1}}(\mathcal{R}_\kappa(B)) \circ f \circ \mathbb{R}_\kappa(A))\\
&\quad = \mathcal{H}(\mathcal{R}_\kappa(f))
\end{aligned}$$

□

9.2 Oubli de champs ou de définitions

On étend l'oubli de champs (noté $\lhd$), en ajoutant une notion d'oubli de définitions. Cet oubli (noté $\blacktriangleleft$) définit ces deux notions de manière imbriquée, car elles se composent mal, si on ne suppose pas l'existence d'une notion de sur-signature pleinement abstraite. (cf 9.4 et 10.4).

Définition 9.9 *On définit $A \blacktriangleleft B$ et $\mathbb{C}(B,A)$ par induction mutuelle sur B.*

1) *Si $B = \mathbf{1}$ alors $A \blacktriangleleft B$ ssi $A = \mathbf{1}$, et dans ce cas, $\mathbb{C}(B,A) = id_{\mathbf{1}}$.*

2) *Si $A \neq \mathbf{1}$, $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$, $B \neq \mathcal{A}(B)$ et $A = \mathcal{A}(A)$,*

$$A \blacktriangleleft B \text{ ssi } \mathcal{F}(A) \blacktriangleleft \mathcal{F}(B) \text{ et } \mathcal{A}(B) = \mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$$

Dans ce cas, $\mathbb{C}(B,A) = q(\mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)), A) \circ \alpha_B$.

3) *Sinon, si $A \neq \mathbf{1}$ et $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$,*

$$A \blacktriangleleft B \text{ ssi } \mathcal{F}(A) \blacktriangleleft \mathcal{F}(B) \text{ et } B = \mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$$

Dans ce cas $\mathbb{C}(B,A) = q(\mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)), A)$.

4) *Sinon, $A \blacktriangleleft B$ ssi $A \blacktriangleleft \mathcal{F}(B)$. Dans ce cas $\mathbb{C}(B,A) = \mathbb{C}(\mathcal{F}(B), A) \circ p_B$.*

Remarque 9.1 *D'après le cas 3 ci-dessus, si $A \neq \mathbf{1}$ et $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$, avec $B \neq \mathcal{A}(B)$ et $A = \mathcal{A}(A)$ alors $A \not\blacktriangleleft B$. En effet, on a vu que pour tout f de $\mathcal{C}[\mathcal{F}(B), \mathcal{F}(A)]$, $B = f * A$ et $A = \mathcal{A}(A)$ implique $B = \mathcal{A}(B)$.*

On étend sans difficulté les propriétés de $\lhd$ à $\blacktriangleleft$. A titre d'exemple, on donne juste ici la preuve de la transitivité de $\blacktriangleleft$ (cf. 9.20).

Lemme 9.19 *Si A et B sont des objets de $\mathcal{C}$, on a les propriétés suivantes :*

1) *Si $A \blacktriangleleft B$ alors $Im(\mathfrak{L}_A) \subseteq Im(\mathfrak{L}_B)$, pour tout a et b de $Im(\mathfrak{L}_A)$, $\mathfrak{L}_A^{-1}(a) < \mathfrak{L}_A^{-1}(b)$ équivaut à $\mathfrak{L}_B^{-1}(a) < \mathfrak{L}_B^{-1}(b)$.*

2) *Si $A \blacktriangleleft B$ alors $\mathfrak{M}_A \subseteq \mathfrak{M}_B$.*

3) *$A \blacktriangleleft A$ et $\mathbb{C}(A,A) = id_A$.*

4) *$A \blacktriangleleft B$, $Im(\mathfrak{L}_A) = Im(\mathfrak{L}_B)$ et $\mathfrak{M}_A = \mathfrak{M}_B$ implique $A = B$.*

Preuve Il suffit d'étendre les preuves faites pour $\lhd$. C'est sans difficulté.

□

Lemme 9.20 *$A \blacktriangleleft B$ et $B \blacktriangleleft C$ implique $A \blacktriangleleft C$ et $\mathbb{C}(C,A) = \mathbb{C}(B,A) \circ \mathbb{C}(C,B)$.*

Preuve Par induction sur $|C|$, puis par cas sur B.

1) Si $C = \mathbf{1}$, alors $B = \mathbf{1}$ et $A = \mathbf{1}$. Évident.

2) Si $\mathfrak{L}_C(|C|) = \mathfrak{L}_B(|B|)$, $C \neq \mathcal{A}(C)$ et $B = \mathcal{A}(B)$. On a $\mathcal{F}(B) \blacktriangleleft \mathcal{F}(C)$ et $\mathcal{A}(C) = \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)) * B$.

2.1) Si $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$, alors $\mathcal{F}(A) \blacktriangleleft \mathcal{F}(B)$ et $B = \mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$.

On a $\mathfrak{L}_C(|C|) = \mathfrak{L}_A(|A|)$. De plus, comme $\mathcal{A}(B) = \mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) * \mathcal{A}(A)$ et $B = \mathcal{A}(B)$ donc $\mathcal{A}(A) = A$.

Et par hypothèse de récurrence, $\mathcal{F}(A) \blacktriangleleft \mathcal{F}(C)$, et $\mathbf{C}(\mathcal{F}(C), \mathcal{F}(A)) = (\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) \circ \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)))$.

$$\begin{aligned}
&\mathcal{A}(C)\\
&\quad = \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)) * ((\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A))) * A)\\
&\quad = (\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) \circ \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B))) * A\\
&\quad = \mathbf{C}(\mathcal{F}(C), \mathcal{F}(A)) * A
\end{aligned}$$

On en déduit que $A \blacktriangleleft C$.

De plus,

$$\begin{aligned}
&\mathbf{C}(C, A)\\
&\quad = q(\mathbf{C}(\mathcal{F}(C), \mathcal{F}(A)), A) \circ \alpha_C\\
&\quad = q(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) \circ \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)), A) \circ \alpha_C\\
&\quad = q(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), A) \circ q(\mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)), B) \circ \alpha_C\\
&\quad = \mathbf{C}(B, A) \circ \mathbf{C}(C, B)
\end{aligned}$$

2.2) Sinon, $\mathfrak{L}_B(|B|) \neq \mathfrak{L}_A(|A|)$ et $A \blacktriangleleft \mathcal{F}(B)$.

Donc $\mathfrak{L}_C(|C|) \neq \mathfrak{L}_A(|A|)$.

Par hypothèse de récurrence, $A \blacktriangleleft \mathcal{F}(C)$, et $\mathbf{C}(\mathcal{F}(C), A) = (\mathbf{C}(\mathcal{F}(B), A) \circ \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)))$. On en déduit que $A \blacktriangleleft C$ et

$$\begin{aligned}
&\mathbf{C}(C, A)\\
&\quad = \mathbf{C}(\mathcal{F}(C), A) \circ p_C\\
&\quad = (\mathbf{C}(\mathcal{F}(B), A) \circ \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B))) \circ (p_{\mathcal{A}(C)} \circ \alpha_C)\\
&\quad = \mathbf{C}(F(B), A) \circ (\mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)) \circ p_{\mathcal{A}(C)}) \circ \alpha_C\\
&\quad = \mathbf{C}(F(B), A) \circ (p_B \circ q(\mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)), B)) \circ \alpha_C\\
&\quad = (\mathbf{C}(F(B), A) \circ p_B) \circ (q(\mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)), B) \circ \alpha_C)\\
&\quad = \mathbf{C}(B, A) \circ \mathbf{C}(C, B)
\end{aligned}$$

3) Sinon, si $\mathfrak{L}_C(|C|) = \mathfrak{L}_B(|B|)$. On a $\mathcal{F}(B) \blacktriangleleft \mathcal{F}(C)$, $C = \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)) * B$.

3.1) Si $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$, $B \neq \mathcal{A}(B)$ et $A = \mathcal{A}(A)$. On a $\mathcal{F}(A) \blacktriangleleft \mathcal{F}(B)$, $\mathcal{A}(B) = \mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$.

On a $\mathfrak{L}_C(|C|) = \mathfrak{L}_A(|A|)$. De plus, on a $\mathcal{A}(C) \neq C$ (car $\mathcal{A}(B) \neq B$) (point 6 de la définition).

Par hypothèse de récurrence, $\mathcal{F}(A) \blacktriangleleft \mathcal{F}(C)$, et $\mathbf{C}(\mathcal{F}(C), \mathcal{F}(A)) = (\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) \circ \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)))$. En appliquant le point 5 de la définition,

$$\begin{aligned}
&\mathcal{A}(C)\\
&\quad = \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)) * \mathcal{A}(B)\\
&\quad = \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)) * ((\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A))) * A)\\
&\quad = (\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) \circ \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B))) * A\\
&\quad = \mathbf{C}(\mathcal{F}(C), \mathcal{F}(A)) * A
\end{aligned}$$

Comme on en déduit que $A \blacktriangleleft C$. De plus,

$$\begin{aligned}
&\mathbf{C}(C, A)\\
&\quad = q(\mathbf{C}(\mathcal{F}(C), \mathcal{F}(A)), A) \circ \alpha_C\\
&\quad = q(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) \circ \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)), A) \circ \alpha_C\\
&\quad = q(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), A) \circ q(\mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)), \mathcal{A}(B)) \circ \alpha_{\mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)) * B}\\
&\quad = q(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), A) \circ \alpha_B \circ q(\mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)), B)\\
&\quad = \mathbf{C}(B, A) \circ \mathbf{C}(C, B)
\end{aligned}$$

3.2) Sinon, si $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$. On a $\mathcal{F}(A) \blacktriangleleft \mathcal{F}(B)$ et $B = \mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$.
On a $\mathfrak{L}_C(|C|) = \mathfrak{L}_A(|A|)$.
Par hypothèse de récurrence, $\mathcal{F}(A) \blacktriangleleft \mathcal{F}(C)$, et
$\mathbf{C}(\mathcal{F}(C), \mathcal{F}(A)) = (\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) \circ \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)))$.

$$\begin{aligned}
&C\\
&\quad = \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)) * ((\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A))) * A)\\
&\quad = (\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) \circ \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B))) * A\\
&\quad = \mathbf{C}(\mathcal{F}(C), \mathcal{F}(A)) * A
\end{aligned}$$

On en déduit bien que $A \blacktriangleleft C$. De plus et

$$\begin{aligned}
&\mathbf{C}(C, A)\\
&\quad = q(\mathbf{C}(\mathcal{F}(C), \mathcal{F}(A)), A)\\
&\quad = q(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) \circ \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)), A)\\
&\quad = q(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), A) \circ q(\mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)), B)\\
&\quad = \mathbf{C}(B, A) \circ \mathbf{C}(C, B)
\end{aligned}$$

3.3) Sinon $\mathfrak{L}_B(|B|) \neq \mathfrak{L}_A(|A|)$ et $A \blacktriangleleft \mathcal{F}(B)$.
$\mathfrak{L}_C(|C|) \neq \mathfrak{L}_A(|A|)$.
Par hypothèse de récurrence, $A \blacktriangleleft \mathcal{F}(C)$, et
$\mathbf{C}(\mathcal{F}(C), A) = (\mathbf{C}(\mathcal{F}(B), A) \circ \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)))$. On en déduit que $A \blacktriangleleft C$ et

$$\begin{aligned}
&\mathbf{C}(C, A)\\
&\quad = \mathbf{C}(\mathcal{F}(C), A) \circ p_C\\
&\quad = (\mathbf{C}(\mathcal{F}(B), A) \circ \mathbf{C}(\mathcal{F}(C), \mathcal{F}(B))) \circ p_C\\
&\quad = \mathbf{C}(F(B), A) \circ (\mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)) \circ p_C)\\
&\quad = \mathbf{C}(F(B), A) \circ (p_B \circ q(\mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)), B))\\
&\quad = (\mathbf{C}(F(B), A) \circ p_B) \circ q(\mathbf{C}(\mathcal{F}(C), \mathcal{F}(B)), B)\\
&\quad = \mathbf{C}(B, A) \circ \mathbf{C}(C, B)
\end{aligned}$$

4) Sinon, $\mathfrak{L}_C(|C|) \notin Im(\mathfrak{L}_B)$, $B \blacktriangleleft \mathcal{F}(C)$. Par hypothèse de récurrence $A \blacktriangleleft \mathcal{F}(C)$ et $\mathbf{C}(\mathcal{F}(C), A) = (\mathbf{C}(B, A) \circ \mathbf{C}(\mathcal{F}(C), B))$.
Comme $Im(\mathfrak{L}_A) \subseteq Im(\mathfrak{L}_B)$, $\mathfrak{L}_C(|C|) \notin Im(\mathfrak{L}_A)$.

$$\begin{aligned}
&\mathbf{C}(C, A)\\
&\quad = \mathbf{C}(\mathcal{F}(C), A) \circ p_C\\
&\quad = \mathbf{C}(B, A) \circ (\mathbf{C}(\mathcal{F}(C), B) \circ p_C)\\
&\quad = \mathbf{C}(B, A) \circ \mathbf{C}(C, B)
\end{aligned}$$

□

Lemme 9.21 *Soient A et B tels que $A \blacktriangleleft B$. Pour tout n, $\mathcal{F}^n(A) \blacktriangleleft \mathcal{F}^n(B)$.*

Preuve Comme pour $\lhd$.

□

Lemme 9.22 *Si $A \blacktriangleleft B$ et $A \neq \mathbf{1}$ alors il existe $C \blacktriangleleft B$ tel que $\mathcal{F}(A) \blacktriangleleft \mathcal{F}(C)$ et $C = \mathbf{C}(\mathcal{F}(C), \mathcal{F}(A)) * A$.*

Preuve Par induction sur $|B|$. $B \neq \mathbf{1}$ car $A \neq \mathbf{1}$.

1) Si $\mathfrak{L}_B(|B|) \in Im(\mathfrak{L}_A)$, $B \neq \mathcal{A}(B)$ et $A = \mathcal{A}(A)$, $\mathcal{F}(A) \blacktriangleleft \mathcal{F}(B)$ et $\mathcal{A}(B) = \mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$. Dans ce cas, on pose $C = \mathcal{A}(B)$. évident.

2) Sinon, si $\mathfrak{L}_B(|B|) \in Im(\mathfrak{L}_A)$, $\mathcal{F}(A) \blacktriangleleft \mathcal{F}(B)$ et $B = \mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$. Dans ce cas, on pose $C = B$. évident.

3) Sinon, si $\mathfrak{L}_B(|B|) \notin Im(\mathfrak{L}_A)$ et $A \blacktriangleleft \mathcal{F}(B)$. On prend le C donné par l'hypothèse de récurrence. On vérifie bien que $C \blacktriangleleft B$ car $\mathfrak{L}_B(|B|) \notin Im(\mathfrak{L}_{\mathcal{F}(B)}) \supseteq Im(\mathfrak{L}_C)$.

□

9.2.1 Oubli de champs

Revenons sur $\lhd$. Le lemme suivant montre que $\blacktriangleleft$ est une extension de $\lhd$.

Lemme 9.23 *Si A et B sont des objets de $\mathcal{C}$, on a les propriétés suivantes :*

1) *$A \lhd B$ implique $A \blacktriangleleft B$ et $\mathbf{C}(B, A) = \mathbb{C}(B, A)$.*

2) *$A \blacktriangleleft B$ et $\mathfrak{M}_A = \mathfrak{M}_B \cap \mathfrak{L}_A$ implique $A \lhd B$*

Preuve Facile par induction sur $|B|$ dans les deux cas.

□

La relation $\lhd$ est préservée sur les hypothèses.

Lemme 9.24 *$A \lhd B$ implique $\mathcal{H}(A) \lhd \mathcal{H}(B)$ et $\mathcal{H}(\mathbb{C}(B, A)) = \mathbb{C}(\mathcal{H}(B), \mathcal{H}(A))$.*

Preuve Par induction sur $|B|$.

1) Si $B = \mathbf{1}$ alors $A = \mathbf{1}$. Evident.

2) Si $\mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$, alors $\mathcal{F}(A) \lhd \mathcal{F}(B)$, $B = \mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$ et $\mathbb{C}(B, A) = q(\mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)), A)$.
Par hypothèse de récurrence, $\mathcal{H}(\mathcal{F}(A)) \lhd \mathcal{H}(\mathcal{F}(B))$ et $\mathcal{H}(\mathbb{C}(\mathcal{F}(B), \mathcal{F}(A))) = \mathbb{C}(\mathcal{H}(\mathcal{F}(B)), \mathcal{H}(\mathcal{F}(A)))$.
Si $A = \mathcal{A}(A)$ alors $B = \mathcal{A}(B)$, $\mathcal{F}(\mathcal{H}(A)) = \mathcal{H}(\mathcal{F}(A))$ et $\mathcal{F}(\mathcal{H}(B)) = \mathcal{H}(\mathcal{F}(B))$.
En appliquant le théorème 9.2, on trouve que $\mathcal{H}(B) = \mathcal{H}(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A))) * \mathcal{H}(A)$ et $\mathcal{H}(\mathbf{C}(B, A)) = q(\mathcal{H}(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A))), \mathcal{H}(A))$.
Comme $\mathfrak{L}_{\mathcal{H}(A)}(|\mathcal{H}(A)|) = \mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|) = \mathfrak{L}_{\mathcal{H}(B)}(|\mathcal{H}(B)|)$, on peut conclure.
Sinon $\mathcal{A}(A) \neq A$ et $\mathcal{A}(B) \neq B$. On a : $\mathcal{H}(A) = \mathcal{H}(\mathcal{F}(A))$, $\mathcal{H}(B) = \mathcal{H}(\mathcal{F}(B))$ et $\mathcal{H}(\mathbb{C}(B, A)) = \mathcal{H}(\mathbb{C}(\mathcal{F}(B), \mathcal{F}(A)))$. On peut conclure.

3) Sinon, $\mathfrak{L}_A(|A|) \notin Im(\mathfrak{L}_B)$, $A \lhd \mathcal{F}(B)$ et $\mathbb{C}(B,A) = \mathbb{C}(\mathcal{F}(B),A) \circ p_B$.
Par hypothèse de récurrence, $\mathcal{H}(A) \lhd \mathcal{H}(\mathcal{F}(B))$ et $\mathcal{H}(\mathbb{C}(\mathcal{F}(B),A)) = \mathbb{C}(\mathcal{H}(\mathcal{F}(B)),\mathcal{H}(A))$.
On en déduit que $\mathfrak{L}_{\mathcal{H}(A)} \subset \mathfrak{L}_{\mathcal{H}(\mathcal{F}(B))}$.
Si $B = \mathcal{A}(B)$ alors $\mathcal{F}(\mathcal{H}(B)) = \mathcal{H}(\mathcal{F}(B))$. On a donc $\mathcal{H}(A) \lhd \mathcal{F}(\mathcal{H}(B))$.
Comme $\mathfrak{L}_{\mathcal{H}(B)}(|\mathcal{H}(B)|) \notin Im(\mathfrak{L}_{\mathcal{F}(\mathcal{H}(B))}) \supset Im(\mathfrak{L}_{\mathcal{H}(A)})$, on en conclut que $\mathcal{H}(A) \lhd \mathcal{H}(B)$.
De plus, $\mathcal{H}(\mathbb{C}(B,A)) = \mathcal{H}(\mathbb{C}(\mathcal{F}(B),A)) \circ \mathcal{H}(p_B)$. Comme, $\mathcal{H}(p_B) = p_{\mathcal{H}(B)}$, on peut conclure.
Sinon $\mathcal{A}(B) \neq B$. On a : $\mathcal{H}(B) = \mathcal{H}(\mathcal{F}(B))$ et $\mathcal{H}(\mathbb{C}(B,A)) = \mathcal{H}(\mathbb{C}(\mathcal{F}(B),A))$. On peut conclure.

□

9.2.2 Oubli de définitions

On étudie maintenant spécifiquement l'oubli de définitions.

Définition 9.10 *On pose $A \propto_l B$ ssi $A \blacktriangleleft B$ avec $l = \mathfrak{M}_B \backslash \mathfrak{M}_A$, et $\mathfrak{L}[A,B] = \emptyset$.*
On notera aussi $A \propto B$ ssi $A \blacktriangleleft B$ et $\mathfrak{L}[A,B] = \emptyset$.

Lemme 9.25 *On a les propriétés suivantes :*
1) *Si $A = \mathbf{1}$ alors $A \propto_\emptyset B$ équivaut à $B = \mathbf{1}$ alors $A \propto_\emptyset B$. Dans ce cas $\mathbf{C}(B,A) = id_{\mathbf{1}}$.*
2) *Si $B \neq \mathcal{A}(B)$ et $\mathcal{A}(A) = A$, alors*
$$A \propto_l B \text{ équivaut à } l = l' \cup \{\mathfrak{L}_A(|A|)\},\ \mathcal{F}(A) \propto_{l'} \mathcal{F}(B) \text{ et } \mathcal{A}(B) = \mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)) * A$$
Dans ce cas, $\mathbf{C}(B,A) = q(\mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)),A) \circ \alpha_B$
3) *Sinon, $A \propto_l B$ équivaut à $\mathcal{F}(A) \propto_l \mathcal{F}(B)$ et $B = \mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)) * A$. Dans ce cas, $\mathbf{C}(B,A) = q(\mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)),A)$*

Preuve C'est juste une reformulation de la définition de $\blacktriangleleft$ en utilisant les conditions sur les étiquettes.

□

Lemme 9.26 *Si $A \propto B$ alors pour tout n, $\mathcal{F}^n(A) \propto \mathcal{F}^n(B)$.*

On montre maintenant que si $A \propto B$ alors $\mathcal{H}(\mathbf{C}(B,A))$ se décompose en définitions élémentaires. Intuitivement, c'est cohérent puisque $A \propto B$ signifie que B est plus défini que A : il décharge des hypothèses de A.

Définition 9.11 *Soient A et B tels que $A \propto B$. On définit la suite n_i, qui vérifie :*
- $n_0 = -1$
- $n_{i+1} = \max\{k = |A| - \mathfrak{L}_A^{-1}(a)\ /\ k \geq n_i \wedge a \in \mathfrak{M}_B \backslash \mathfrak{M}_A\}$.

On pose $S_{A,B} = (\mathcal{H}(\mathbf{C}(\mathcal{F}^{n_i}(B),\mathcal{F}^{n_i}(A))))_{1 \geq i}$.

Lemme 9.27 *Pour tout A et B, $A \propto B$ implique $S_{A,B} \in [\mathcal{H}(A) \ltimes \mathcal{H}(B)]$, et $\mathbb{G}[\mathcal{H}(A),\mathcal{H}(B)](S_{A,B}) = \mathcal{H}(\mathbf{C}(B,A))$*

Preuve Par induction sur $|A|$.
1) Si $A = \mathbf{1}$ alors $B = \mathbf{1}$. Évident.

2) Si $B \neq \mathcal{A}(B)$ et $\mathcal{A}(A) = A$, alors $\mathcal{F}(A) \propto \mathcal{F}(B)$ et $\mathcal{A}(B) = \mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$.
On a : $S_{A,B} = S_{\mathcal{F}(A),\mathcal{F}(B)}; \mathcal{H}(\mathbf{C}(B, A))$, et $\mathcal{H}(B) = \mathcal{H}(\mathcal{F}(B))$, $\mathcal{H}(\mathcal{F}(A)) = \mathcal{F}(\mathcal{H}(A))$ et $\mathcal{H}(p_A) = p_{\mathcal{H}(A)}$.
Par hypothèse de récurrence, $S_{\mathcal{F}(A),\mathcal{F}(B)} \in [\mathcal{F}(\mathcal{H}(A)) \ltimes \mathcal{H}(B)]$ et $\mathbb{G}[\mathcal{F}(\mathcal{H}(A)), \mathcal{H}(B)](S_{\mathcal{F}(A),\mathcal{F}(B)}) = \mathcal{H}(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)))$.
De plus $\mathbf{C}(B, A) = q(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), A) \circ \alpha_B$, Donc

$$\begin{aligned}
&p_{\mathcal{H}(A)} \circ \mathcal{H}(\mathbf{C}(B, A)) \\
&\quad = \mathcal{H}(p_A \circ \mathbf{C}(B, A)) \\
&\quad = \mathcal{H}(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) \circ p_{\mathcal{A}(B)} \circ \alpha_B) \\
&\quad = \mathcal{H}(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A))) \circ \mathcal{H}(p_B) \\
&\quad = \mathbb{G}[\mathcal{F}(\mathcal{H}(A)), \mathcal{H}(B)](S_{\mathcal{F}(A),\mathcal{F}(B)}) \circ id_{\mathcal{H}(B)}
\end{aligned}$$

On en déduit les énoncés recherchés.

3) Sinon, $\mathcal{F}(A) \propto \mathcal{F}(B)$, $B = \mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$, $\mathbf{C}(B, A) = q(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), A)$, $S_{A,B} = S_{\mathcal{F}(A),\mathcal{F}(B)}$.
Par hypothèse de récurrence, $S_{\mathcal{F}(A),\mathcal{F}(B)} \in [\mathcal{H}(\mathcal{F}(A)) \ltimes \mathcal{H}(\mathcal{F}(B))]$ et $\mathbb{G}[\mathcal{H}(\mathcal{F}(A)), \mathcal{H}(\mathcal{F}(B))](S_{\mathcal{F}(A),\mathcal{F}(B)}) = \mathcal{H}(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)))$.
Si $A = \mathcal{A}(A)$ alors $B = \mathcal{A}(B)$, $\mathcal{F}(\mathcal{H}(A)) = \mathcal{H}(\mathcal{F}(A))$ et $\mathcal{F}(\mathcal{H}(B)) = \mathcal{H}(\mathcal{F}(B))$. En appliquant le théorème 9.2, on trouve que $\mathcal{H}(B) = \mathcal{H}(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A))) * \mathcal{H}(A)$ et $\mathcal{H}(\mathbf{C}(B, A)) = q(\mathcal{H}(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A))), \mathcal{H}(A))$. On peut conclure.
Sinon $A \neq \mathcal{A}(A)$ et $B \neq \mathcal{A}(B)$. On a $\mathcal{H}(A) = \mathcal{H}(\mathcal{F}(A))$, $\mathcal{H}(B) = \mathcal{H}(\mathcal{F}(B))$, et $\mathcal{H}(\mathbf{C}(B, A)) = \mathcal{H}(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)))$. On peut donc conclure.

□

Remarque 9.2 *Sous les mêmes hypothèses, en utilisant la définition 8.1 et le lemme 9.14, on montre que* $(S_A \circledcirc_{A,\mathcal{H}(A),\mathcal{H}(B)} S_{A,B}) \in [A \ltimes \mathcal{H}(B)]$, *et que*

$$\mathbb{G}[A, \mathcal{H}(B)](S_A \circledcirc_{A,\mathcal{H}(A),\mathcal{H}(B)} S_{A,B}) = \mathbb{G}(A) \circ \mathcal{H}(\mathbf{C}(B, A)) = \mathbf{C}(B, A) \circ \mathbb{G}(B)$$

9.3 Définition d'un champ

On cherche à définir le k-ième champ (en partant de la fin) encore abstrait d'un générateur d'enregistrement B. Donner la définition, c'est ici donner une section s d'une certaine signature A : $\mathcal{F}(A)$ est le "contexte" de la définition du champ. Dans $\mathcal{F}(A)$, les champs ne sont pas forcément abstraits : il peut y avoir des dépendances transparentes entre le dernier champ de A et les champs de $\mathcal{F}(A)$.

On reste très général quant au plongement exigé entre $\mathcal{F}^{k+1}(B)$ (le contexte dans lequel sera placée cette définition dans B) et $\mathcal{F}(A)$. Tout ce que l'on exige est un morphisme f de $\mathcal{F}^{k+1}(B)$ dans $\mathcal{F}(A)$ tel que $\mathcal{F}^k(B) = f * A$ (le type du champ dans A est le même que dans B après "conversion" suivant f). On est en fait ainsi dans l'esprit du "coercive subtyping" de Luo et Soloviev.

Lemme-Définition 9.1 *Soit* A *avec* $\mathcal{A}(A) = A$ *et* s *une section de* $\mathcal{H}(A)$. *Soient* B *et* f *de* $\mathcal{C}[\mathcal{F}^{k+1}(B), \mathcal{F}(A)]$ *tel que* $\mathcal{F}^k(B) = f * A$. *on construit* $B \oslash_f s$ *tel que* $B \propto_{\{\mathfrak{L}_A(|A|)\}} (B \oslash_f s)$.

$$
\begin{tikzcd}
B \oslash_f s \arrow[rr] \arrow[dd, dashed, "p^k(B \oslash_f s)"'] & & B \arrow[dd, dashed, "p^k(B)"'] & & & & \\
& & & \mathcal{M}(s) \arrow[rr] \arrow[ddr] & & A \arrow[ddl] & \mathcal{H}(A) \arrow[l] \\
f * \mathcal{M}(s) \arrow[urrr] \arrow[rr] \arrow[ddr] & & f * A \arrow[urrr] \arrow[ddl] & & & & \\
& & & & \mathcal{F}(A) & & \mathcal{H}(\mathcal{F}(A)) \arrow[ll] \arrow[uu, "s"'] \\
& \mathcal{F}^{k+1}(B) \arrow[urrr, "f"'] & & & & &
\end{tikzcd}
$$

Construction Par induction sur k.

1) Si $k = 0$. On pose

$$\boxed{B \oslash_f s = f * \mathcal{M}(s)}$$

C'est valide car $\mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|) \notin Im(\mathfrak{L}_{\mathcal{F}(B)})$.
On a $\mathcal{A}(B \oslash_f s) = B$ et $\mathcal{A}(B \oslash_f s) \neq B \oslash_f s$,
donc $B \propto_{\{\mathfrak{L}_A(|A|)\}} (B \oslash_f s)$.

2) Sinon, par récurrence, $\mathcal{F}(B) \propto_{\{\mathfrak{L}_A(|A|)\}} (\mathcal{F}(B) \oslash_f s)$. On pose

$$\boxed{(B \oslash_f s) = \mathbf{C}((\mathcal{F}(B) \oslash_f s), \mathcal{F}(B)) * B}$$

Donc $B \propto_{\{\mathfrak{L}_A(|A|)\}} (B \oslash_f s)$.

□

Lemme 9.28 *Avec les mêmes hypothèses, on a :*

$$q(f, A) \circ \alpha_{\mathcal{F}^k(B \oslash_f s)} \circ \mathbb{G}(\mathcal{F}^k(B \oslash_f s)) = \mathbb{G}(A) \circ s \circ \mathcal{H}(f)$$

Preuve

$$
\begin{aligned}
& q(f, A) \circ \alpha_{\mathcal{F}^k(B \oslash_f s)} \circ \mathbb{G}(\mathcal{F}^k(B \oslash_f s)) \\
& \quad = \alpha_{\mathcal{M}(s)} \circ q(f, \mathcal{M}(s)) \circ \mathbb{G}(\mathcal{F}^k(B \oslash_f s)) \\
& \quad = \alpha_{\mathcal{M}(s)} \circ \mathbb{G}(\mathcal{M}(s)) \circ \mathcal{H}(f) \\
& \quad = \mathbb{G}(A) \circ s \circ \mathcal{H}(f)
\end{aligned}
$$

□

Remarque 9.3 *Ceci donne une notion d'héritage très générale, puisque n'importe quelle morphisme f convient : on peut par exemple prendre oubli ou oubli+permutation ou oubli+permutation+renommage etc.*

9.4 Notion de sur-signature pleinement abstraite

Dans les espèces de FocKer, les types des méthodes ne peuvent pas dépendre de manière transparente des autres méthodes : à chaque espèce correspond une interface contenant les mêmes champs, c'est-à-dire qui est obtenue en oubliant toutes les définitions. Par exemple, en supposant que $list\_n : (n : nat)Set$ représente les listes d'entiers de longueur n, on ne veut pas représenter comme une espèce la signature avec champs manifeste suivante :

$$A = \{T : Set = nat \ ; \ x : T \ ; \ l : (list\_n \ x)\}$$

En effet, si on oublie la définition associée à l'étiquette T, alors cette signature devient mal typée sur le champ d'étiquette l. Dans cette signature, le champ d'étiquette l dépend de manière transparente de T (voir la discussion à ce sujet dans la section 10.4).

Cette contrainte sur les espèces s'exprime au niveau des générateurs d'enregistrements par l'existence *d'une notion de sur-signature pleinement abstraite* :

Définition 9.12 *Une notion de sur-signature pleinement abstraite est une fonction $\mathcal{T}$ de $Ob(\mathcal{C})$ dans $Ob(\mathcal{C})$ telle que*

1) *pour tout A, $\mathcal{T}(A) \propto_{\mathfrak{M}_A} A$.*

2) *pour tous A et B, $A \propto B$ implique $\mathcal{T}(A) = \mathcal{T}(B)$.*

9.4.1 Exemple de $\mathfrak{C}_\ltimes$

On construit une notion de sur-signature pleinement abstraite dans $\mathfrak{C}_\ltimes$.

Définition 9.13 *On pose $\mathcal{T}_\ltimes(< A, B, S >) = A_\ltimes$.*

Théorème 9.5 *$\mathcal{T}_\ltimes$ est une notion de sur-signature pleinement abstraite dans la catégorie des générateurs d'enregistrements de $\mathcal{C}_\ltimes$*

Preuve Le point 1 de la définition découle du lemme 9.31 ci-dessous, et le point 2 de la définition découle du lemme 9.29 ci-dessous.

□

Ci-dessous, on note $\propto_\ltimes$ la relation $\propto$ sur la CG $\mathfrak{C}_\ltimes$.

Lemme 9.29 *Soit $X_1 =< A_1, B_1, S_1 >$ et $X_2 =< A_2, B_2, S_2 >$, tel que $X_1 \propto_\ltimes X_2$. Soit $< g, h >= \mathbf{C}_\ltimes(X_2, X_1)$. On a $A_1 = A_2$ et $g = id_{A_1}$.*

Preuve Par récurrence sur $|X_2|$.

1) Si $X_1 = X_2 = \mathbf{1}_\ltimes$. On a bien $A_1 = \mathbf{1}$, $A_2 = \mathbf{1}$.

2) Si $X_2 \neq \mathcal{A}_\ltimes(X_2)$ et $\mathcal{A}_\ltimes(X_1) = X_1$, alors $\mathcal{F}_\ltimes(X_1) \propto_\ltimes \mathcal{F}_\ltimes(X_2)$ et $\mathcal{A}_\ltimes(X_2) = \mathbf{C}_\ltimes(\mathcal{F}_\ltimes(X_2), \mathcal{F}_\ltimes(X_1)) *_\ltimes X_1$.
Par récurrence, on a : $\mathcal{F}(A_1) = \mathcal{F}(A_2)$ et $A_2 = id_{\mathcal{F}(A_1)} * A_1$. Donc $A_2 = A_1$.
De même $\mathbf{C}_\ltimes(X_2, X_1) = Q_\ltimes(\mathbf{C}_\ltimes(\mathcal{F}_\ltimes(X_2), \mathcal{F}_\ltimes(X_1)), X_1) \circ \alpha_{\ltimes X_1}$ implique $g = q(id_{\mathcal{F}(A_1)}, A_1) \circ id_{A_2} = id_{A_1}$.

3) Sinon, $\mathcal{F}_\ltimes(X_1) \propto_\ltimes \mathcal{F}_\ltimes(X_2)$ et $X_2 = \mathbf{C}_\ltimes(\mathcal{F}_\ltimes(X_2), \mathcal{F}_\ltimes(X_1)) *_\ltimes X_1$. Par le même raisonnement que précédemment, on trouve $A_2 = A_1$ et $g = id_{A_1}$.

□

Lemme 9.30 *Soit $X =< A, B, S >$ tels que $A_\ltimes \propto_\ltimes X$. On a $\mathbf{C}(X, A_\ltimes) =< id_A, S >$.*

Preuve On sait en fait déjà par le lemme qui précède que $g = id_A$. Par récurrence sur $|X|$. On pose $< g, h >= \mathbf{C}(X, A_\ltimes)$.

1) Si $X = \mathbf{1}_\ltimes$. On a bien $A = B = \mathbf{1}$, $S = id_\mathbf{1}$.

2) Si $X \neq \mathcal{A}_\ltimes(X)$, alors $\mathcal{F}(A)_\ltimes \propto_\ltimes \mathcal{F}_\ltimes(X)$ et $\mathcal{A}_\ltimes(X) = \mathbb{C}_\ltimes(\mathcal{F}_\ltimes(X), \mathcal{F}(A)_\ltimes) *_\ltimes A_\ltimes$.
On a $\mathfrak{L}_A(|A|) \notin Im(\mathfrak{L}_B)$, et :

$$\begin{aligned}
&\mathbf{C}_\ltimes(X, A_\ltimes)\\
&\quad = Q_\ltimes(\mathbb{C}_\ltimes(\mathcal{F}_\ltimes(X), \mathcal{F}(A)_\ltimes), A_\ltimes) \circ \alpha_{\ltimes X}\\
&\quad = < q(id_{\mathcal{F}(A)}, A), q(\mathcal{F}(S), A) > \circ < id_A, id_B \curlywedge_{\mathcal{F}(S)*A} S >\\
&\quad = < id_A, S >
\end{aligned}$$

3) Sinon, $\mathcal{F}(A)_\ltimes \propto_\ltimes \mathcal{F}_\ltimes(X)$ et $X = \mathbb{C}_\ltimes(\mathcal{F}_\ltimes(X), \mathcal{F}(A)_\ltimes) *_\ltimes A_\ltimes$.
On a $\mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$, et $S = q(\mathcal{F}(S), A)$:

$$\begin{aligned}
&\mathbb{C}_\ltimes(X, A_\ltimes)\\
&\quad = Q_\ltimes(\mathbb{C}_\ltimes(\mathcal{F}_\ltimes(X), \mathcal{F}(A)_\ltimes), A_\ltimes)\\
&\quad = < q(id_{\mathcal{F}(A)}, A), q(\mathcal{F}(S), A) >\\
&\quad = < id_A, S >
\end{aligned}$$

□

Lemme 9.31 *Soit $X =< A, B, S >$, alors $A_\ltimes \propto_\ltimes X$.*

Preuve Par récurrence sur $|X|$.

1) Si $X = \mathbf{1}_\ltimes$. $A_\ltimes = \mathbf{1}_\ltimes$. Évident.

2) Si $X \neq \mathcal{A}_\ltimes(X)$, alors par récurrence, $\mathcal{F}(A)_\ltimes \propto_\ltimes \mathcal{F}_\ltimes(X)$. Par le lemme qui précède, $\mathbf{C}(\mathcal{F}_\ltimes(X), \mathcal{F}(A)_\ltimes) =< id_{\mathcal{F}(A)}, \mathcal{F}(S) >$.

$$\begin{aligned}
&\mathbf{C}(\mathcal{F}_\ltimes(X), \mathcal{F}(A)_\ltimes) *_\ltimes A_\ltimes\\
&\quad = < id_{\mathcal{F}(A)} * A, \mathcal{F}(S) * A, q(\mathcal{F}(S), A) >\\
&\quad = \mathcal{A}_\ltimes(X)
\end{aligned}$$

On en déduit que $A_\ltimes \propto_\ltimes X$.

3) Sinon $X = \mathcal{A}_\ltimes(X)$. Par le même raisonnement, on montre que
$X = \mathbf{C}(\mathcal{F}_\ltimes(X), \mathcal{F}(A)_\ltimes) *_\ltimes A_\ltimes$, et $A_\ltimes \propto_\ltimes X$.

□

9.4.2 Propriétés de base

Lemme 9.32 *Si $\mathfrak{M}_A = \emptyset$, alors $\mathcal{T}(A) = A$. En particulier, pour tout B, $\mathcal{T}(\mathcal{T}(B)) = \mathcal{T}(B)$.*

Preuve $\mathcal{T}(A) \blacktriangleleft A$, avec $\mathfrak{L}_{\mathcal{T}(A)} = \mathfrak{L}_A$ et $\mathfrak{M}_{\mathcal{T}(A)} = \emptyset = \mathfrak{M}_A$. Par le lemme 9.19, $\mathcal{T}(A) = A$.

□

Lemme 9.33 *Pour tout A, $\mathcal{F}(\mathcal{T}(A)) = \mathcal{T}(\mathcal{F}(A))$.*

Preuve En effet, comme $\mathcal{T}(A) \propto A$, on a $\mathcal{F}(\mathcal{T}(A)) \propto \mathcal{F}(A)$. On en déduit que $\mathcal{T}(\mathcal{F}(\mathcal{T}(A))) = \mathcal{T}(\mathcal{F}(A))$. Or d'après le lemme précédent, comme $\mathfrak{M}_{\mathcal{F}(\mathcal{T}(A))} = \emptyset$, on a $\mathcal{T}(\mathcal{F}(\mathcal{T}(A))) = \mathcal{F}(\mathcal{T}(A))$.

□

On montre ici que d'avoir une notion de sur-signature pleinement abstraite permet de décomposer $\blacktriangleleft$ en $\triangleleft$ puis $\propto$.

Lemme-Définition 9.2 *Supposons $A \blacktriangleleft B$. On construit $I[A,B]$ tel que $A \triangleleft I[A,B]$ et $I[A,B] \propto B$.*

Construction Par récurrence sur $|B|$.

1) Si $B = \mathbf{1}$ et $A = \mathbf{1}$, on pose $I[A,B] = \mathbf{1}$.

2) Si $A \neq \mathbf{1}$, $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$, $B \neq \mathcal{A}(B)$ et $A = \mathcal{A}(A)$. Alors, $\mathcal{F}(A) \blacktriangleleft \mathcal{F}(B)$ et $\mathcal{A}(B) = \mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$.

Par hypothèse de récurrence, $\mathcal{F}(A) \triangleleft I[\mathcal{F}(A), \mathcal{F}(B)]$ et $I[\mathcal{F}(A), \mathcal{F}(B)] \propto \mathcal{F}(B)$.

On pose

$$\boxed{I[A,B] = \mathbb{C}(I[\mathcal{F}(A), \mathcal{F}(B)], \mathcal{F}(A)) * A}$$

Comme $\mathfrak{L}_{I[A,B]}(|I[A,B]|) = \mathfrak{L}_A(|A|)$, on a bien $A \triangleleft I[A,B]$.

Comme $\mathcal{F}(A) \blacktriangleleft I[\mathcal{F}(A), \mathcal{F}(B)]$, Par transitivité,
$\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) = \mathbf{C}(I[\mathcal{F}(A), \mathcal{F}(B)], \mathcal{F}(A)) \circ \mathbf{C}(\mathcal{F}(B), I[\mathcal{F}(A), \mathcal{F}(B)])$ Comme
$\mathbf{C}(I[\mathcal{F}(A), \mathcal{F}(B)], \mathcal{F}(A)) = \mathbb{C}(I[\mathcal{F}(A), \mathcal{F}(B)], \mathcal{F}(A))$, on en conclut que
$\mathcal{A}(B) = \mathbf{C}(\mathcal{F}(B), I[\mathcal{F}(A), \mathcal{F}(B)]) * I[\mathcal{F}(A), \mathcal{F}(B)]$.

Comme $\mathfrak{L}_{I[A,B]}(|I[A,B]|) = \mathfrak{L}_B(|B|)$ et $I[A,B] = \mathcal{A}(I[A,B])$, on a bien $I[A,B] \propto B$.

3) Sinon, si $A \neq \mathbf{1}$ et $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$, $\mathcal{F}(A) \blacktriangleleft \mathcal{F}(B)$ et $B = \mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$.

Par hypothèse de récurrence, $\mathcal{F}(A) \triangleleft I[\mathcal{F}(A), \mathcal{F}(B)]$ et $I[\mathcal{F}(A), \mathcal{F}(B)] \propto \mathcal{F}(B)$.

On pose

$$\boxed{I[A,B] = \mathbb{C}(I[\mathcal{F}(A), \mathcal{F}(B)], \mathcal{F}(A)) * A}$$

On a donc $\mathcal{F}(I[A,B]) = I[\mathcal{F}(A), \mathcal{F}(B)]$.

Comme $\mathfrak{L}_{I[A,B]}(|I[A,B]|) = \mathfrak{L}_A(|A|)$, on a bien $A \triangleleft I[A,B]$.

Comme $\mathcal{F}(A) \blacktriangleleft I[\mathcal{F}(A), \mathcal{F}(B)]$, Par transitivité,
$\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) = \mathbf{C}(\mathcal{F}(I[A,B]), \mathcal{F}(A)) \circ \mathbf{C}(\mathcal{F}(B), \mathcal{F}(I[A,B]))$ Comme
$\mathbf{C}(\mathcal{F}(I[A,B]), \mathcal{F}(A)) = \mathbb{C}(\mathcal{F}(I[A,B]), \mathcal{F}(A))$, on en conclut que
$B = \mathbf{C}(\mathcal{F}(B), \mathcal{F}(I[A,B])) * I[A,B]$, et $I[A,B] \propto B$.

4) Sinon, $A \blacktriangleleft \mathcal{F}(B)$.

Par hypothèse de récurrence, $A \triangleleft I[A, \mathcal{F}(B)]$ et $I[A, \mathcal{F}(B)] \propto \mathcal{F}(B)$.

Donc $\mathcal{T}(I[A, \mathcal{F}(B)]) = \mathcal{T}(\mathcal{F}(B)) = \mathcal{F}(\mathcal{T}(B))$.

On pose

$$\boxed{I[A,B] = \mathbf{C}(I[A, \mathcal{F}(B)], \mathcal{F}(\mathcal{T}(B))) * \mathcal{T}(B)}$$

On a donc $\mathcal{F}(I[A,B]) = I[A, \mathcal{F}(B)]$. $A \triangleleft \mathcal{F}(I[A,B])$ donc $A \triangleleft I[A,B]$.

De plus, par transitivité,
$\mathbf{C}(\mathcal{F}(B), \mathcal{F}(\mathcal{T}(B))) = \mathbf{C}(\mathcal{F}(I[A,B]), \mathcal{F}(\mathcal{T}(B))) \circ \mathbf{C}(\mathcal{F}(B), \mathcal{F}(I[A,B]))$ On en déduit que
$B = \mathbf{C}(\mathcal{F}(B), \mathcal{F}(I[A,B])) * I[A,B]$, et $I[A,B] \propto B$.

$\square$

Remarque 9.4 *Par transitivité, on sait que $A \propto B$ et $B \triangleleft C$ implique $A \blacktriangleleft C$, c'est-à-dire $A \triangleleft I[A,C] \propto C$.*

Remarque 9.5 *Si $A \blacktriangleleft B$, alors $\mathcal{H}(\mathbf{C}(B,A)) = \mathbb{C}(\mathcal{H}(I[A,B]), \mathcal{H}(A)) \circ \mathbb{G}[\mathcal{H}(I[A,B]), \mathcal{H}(B)](S_{I[A,B],B})$.*

Lemme 9.34 *Si $A \blacktriangleleft B$, alors $\mathfrak{M}_{I[A,B]} = \mathfrak{M}_A$.*

Preuve Par récurrence sur $|B|$.

1) Si $B = \mathbf{1}$ et $A = \mathbf{1}$, évident.

2) Si $A \neq \mathbf{1}$, $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$, $B \neq \mathcal{A}(B)$ et $A = \mathcal{A}(A)$, $I[A,B] = \mathbb{C}(I[\mathcal{F}(A), \mathcal{F}(B)], \mathcal{F}(A)) * A$, Par récurrence, $\mathfrak{M}_{\mathcal{F}(I[A,B])} = \mathfrak{M}_{\mathcal{F}(A)}$. donc $\mathfrak{M}_{I[A,B]} = \mathfrak{M}_{\mathcal{F}(A)} = \mathfrak{M}_A$.

3) Sinon, si $A \neq \mathbf{1}$ et $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$, $I[A,B] = \mathbb{C}(I[\mathcal{F}(A), \mathcal{F}(B)], \mathcal{F}(A)) * A$. Par récurrence, $\mathfrak{M}_{\mathcal{F}(I[A,B])} = \mathfrak{M}_{\mathcal{F}(A)}$. donc $\mathfrak{M}_{I[A,B]} = \mathfrak{M}_A$.

4) Sinon, $I[A,B] = \mathbf{C}(I[A, \mathcal{F}(B)], \mathcal{F}(\mathcal{T}(B))) * \mathcal{T}(B)$ Par récurrence, $\mathfrak{M}_{\mathcal{F}(I[A,B])} = \mathfrak{M}_A$. $\mathcal{A}(I[A,B]) = I[A,B]$, donc $\mathfrak{M}_{I[A,B]} = \mathfrak{M}_{\mathcal{F}(I[A,B])} = \mathfrak{M}_A$.

□

9.4.3 Fusion de deux générateurs de même sur-signature pleinement abstraite, et de champs manifestes disjoints

Comme pour les mixDrecs (voir page 126), on définit ci-dessous l'opération qui correspond à la fusion de deux espèces de même interface. En fait, on impose ici de plus que les méthodes que définissent ces espèces aient des noms différent. A ce niveau, on ne sait en effet pas encore "effacer" des définitions.

Lemme-Définition 9.3 *Soient A et B tels que $\mathcal{T}(A) = \mathcal{T}(B)$ avec $\mathfrak{M}_A \cap \mathfrak{M}_B = \emptyset$. On construit $A \oplus B$ tel que $A \propto (A \oplus B)$ et $B \propto (A \oplus B)$.*

Construction Par induction sur $|A|$.

1) Si $A = \mathbf{1}$ alors $\mathcal{T}(A) = \mathcal{T}(B) = B = \mathbf{1}$. On pose $\boxed{A \oplus B = \mathbf{1}}$.

2) Si $A \neq \mathcal{A}(A)$, alors $B = \mathcal{A}(B)$.

On a : $\mathcal{T}(\mathcal{F}(A)) \propto \mathcal{F}(A)$, $\mathcal{T}(\mathcal{F}(B)) \propto \mathcal{F}(B)$ et $B = \mathbf{C}(\mathcal{F}(B), \mathcal{T}(\mathcal{F}(A))) * \mathcal{T}(A)$. Par hypothèse de récurrence,

$$\mathcal{F}(A) \propto (\mathcal{F}(A) \oplus \mathcal{F}(B)) \qquad \text{et} \qquad \mathcal{F}(B) \propto (\mathcal{F}(A) \oplus \mathcal{F}(B))$$

On pose

$$\boxed{A \oplus B = \mathbf{C}(\mathcal{F}(A) \oplus \mathcal{F}(B), \mathcal{F}(A)) * A}$$

On en déduit que $A \propto (A \oplus B)$, et donc :

$$\begin{aligned}
&\mathcal{A}(A \oplus B) \\
&\quad = \mathbf{C}(\mathcal{F}(A) \oplus \mathcal{F}(B), \mathcal{T}(\mathcal{F}(A))) * \mathcal{T}(A)) \\
&\quad = (\mathbf{C}(\mathcal{F}(B), \mathcal{T}(\mathcal{F}(A))) \circ \mathbf{C}(\mathcal{F}(A) \oplus \mathcal{F}(B), \mathcal{F}(B))) * \mathcal{T}(A) \\
&\quad = \mathbf{C}(\mathcal{F}(A) \oplus \mathcal{F}(B), \mathcal{F}(B)) * (\mathbf{C}(\mathcal{F}(B), \mathcal{T}(\mathcal{F}(A))) * \mathcal{T}(A)) \\
&\quad = \mathbf{C}(\mathcal{F}(A) \oplus \mathcal{F}(B), \mathcal{F}(B)) * B
\end{aligned}$$

On en déduit que $B \propto (A \oplus B)$.

3) Si $B \neq \mathcal{A}(B)$, alors $A = \mathcal{A}(A)$, cas symétrique du précédent. On pose :

$$\boxed{A \oplus B = \mathbf{C}(\mathcal{F}(A) \oplus \mathcal{F}(B), \mathcal{F}(B)) * B}$$

4) Sinon, si $\mathcal{A}(A) = A$ et $\mathcal{A}(B) = B$, on pose aussi :

$$\boxed{A \oplus B = \mathbf{C}(\mathcal{F}(A) \oplus \mathcal{F}(B), \mathcal{F}(B)) * B}$$

On conclut par un raisonnement similaire...

□

Lemme 9.35 *Avec les mêmes hypothèses,* $\mathfrak{M}_{A\oplus B} = \mathfrak{M}_A \cup \mathfrak{M}_B$*, et* $A \oplus B = B \oplus A$

9.4.4 Plongement d'un générateur dans une signature abstraite "plus grande"

Comme pour les mixDrecs (voir page 126), on définit ci-dessous l'opération qui correspond au plongement d'une espèce dans une interface *"plus grande"*, au sens de la relation d'inclusion entre interface, mais sans permutation. On interdit ici les permutations, car pour permuter des méthodes en toute généralité au niveau des espèces, il faut savoir effacer des définitions (par exemple, quand on permute, une méthode a avec une méthode b alors que la définition de a dépend de b).

Lemme-Définition 9.4 *Soient* A*,* T *de* $Ob(\mathcal{C})$ *tels que* $\mathcal{T}(A) \lhd T$ *et* $\mathfrak{M}_T = \emptyset$*. On construit* $\Uparrow^T A$ *tel que* $\mathcal{T}(\Uparrow^T A) = T$*, avec* $A \lhd (\Uparrow^T A)$*.*

Construction On procède par induction sur $|T|$:

1) Si $T = \mathbf{1}$ alors A aussi. On pose $\boxed{\Uparrow^T A = \mathbf{1}}$.

2) Si $\mathfrak{L}_T(|T|) = \mathfrak{L}_A(|A|)$, on a $\mathcal{T}(\mathcal{F}(A)) \lhd \mathcal{F}(T)$. Par récurrence, on a : $\mathcal{T}(\Uparrow^{\mathcal{F}(T)} \mathcal{F}(A)) = \mathcal{F}(T)$ et $\mathcal{F}(A) \lhd (\Uparrow^{\mathcal{F}(T)} \mathcal{F}(A))$. On pose

$$\boxed{\Uparrow^T A = \mathbb{C}(\Uparrow^{\mathcal{F}(T)} \mathcal{F}(A), \mathcal{F}(A)) * A}$$

On a donc bien $A \lhd (\Uparrow^T A)$. De plus

$$\begin{aligned}
&\mathcal{A}(\Uparrow^T A)\\
&\quad = \mathbf{C}(\Uparrow^{\mathcal{F}(T)} \mathcal{F}(A), \mathcal{F}(A)) * \mathcal{A}(A)\\
&\quad = \mathbf{C}(\Uparrow^{\mathcal{F}(T)} \mathcal{F}(A), \mathcal{F}(A)) * (\mathbf{C}(\mathcal{F}(A), \mathcal{F}(\mathcal{T}(A))) * \mathcal{T}(A))\\
&\quad = \mathbf{C}(\Uparrow^{\mathcal{F}(T)} \mathcal{F}(A), \mathcal{F}(\mathcal{T}(A))) * \mathcal{T}(A)\\
&\quad = \mathbf{C}(\Uparrow^{\mathcal{F}(T)} \mathcal{F}(A), \mathcal{F}(T)) * (\mathbf{C}(\mathcal{F}(T), \mathcal{F}(\mathcal{T}(A))) * \mathcal{T}(A))\\
&\quad = \mathbf{C}(\Uparrow^{\mathcal{F}(T)} \mathcal{F}(A), \mathcal{F}(T)) * T
\end{aligned}$$

On en déduit que $T \propto \Uparrow^T A$. Ce qui permet de conclure que $\mathcal{T}(\Uparrow^T A) = T$.

3) Sinon, $\mathcal{T}(A) \lhd \mathcal{F}(T)$. Par récurrence, on a que $\mathcal{T}(\Uparrow^{\mathcal{F}(T)} A) = \mathcal{F}(T)$ et $A \lhd \Uparrow^{\mathcal{F}(T)} A$.
On pose

$$\boxed{\Uparrow^T A = \mathbf{C}(\Uparrow^{\mathcal{F}(T)} A, \mathcal{F}(T)) * T}$$

On a donc bien $T \propto \Uparrow^T A$ et $\mathcal{T}(\Uparrow^T A) = T$. De plus $A \lhd \Uparrow^{\mathcal{F}(T)} A = \mathcal{F}(\Uparrow^T A)$. Donc $A \lhd \Uparrow^T A$.

□

Chapitre 10

CS avec champs manifestes

Cette section est une petite digression dans la présentation de notre modèle : elle permet surtout de montrer que les signatures d'enregistrements avec champs manifestes (au sens de Pollack) forment une catégorie de générateurs d'enregistrements. On axiomatise les signatures avec champs manifestes avec une nouvelle sorte de catégories.

En fait, les catégories de générateurs d'enregistrements que l'on va considérer dans notre modèle ne sont pas des catégories de signatures avec champs manifestes. Dans FocKer, c'est exprimé par le fait que les types des méthodes ne peuvent pas dépendre de manière transparente des types des autres méthodes. Ce n'est pas le cas général des signatures d'enregistrements avec champs manifestes.

10.1 Définition et exemples

La définition des catégories de signatures d'enregistrements avec champs manifestes, est presque la même que pour les catégories de générateurs d'enregistrements. La seule différence, est qu'on ne suppose pas l'existence de $\mathcal{H}$ et de $\mathbb{G}$. A la place, on suppose l'existence d'une fonction $\mathcal{S}$ qui a toute signature A dont le dernier champ est manifeste, associe une section qui est un isomorphime (avec $p(A)$ comme inverse).

Définition 10.1 *Une* catégorie de signatures d'enregistrements avec champs manifestes $\mathfrak{C}$ *est un uplet* $< \mathcal{C}, \mathtt{'A}, \mathcal{F}, \mathfrak{L}, \mathbf{1}, p, *, q, \mathcal{A}, \alpha, \mathcal{M}, \mathcal{S} >$ *tel que :*

1) $< \mathcal{C}, \mathtt{'A}, \mathcal{F}, \mathfrak{L}, \mathbf{1}, p, *, q >$ *est une catégorie des signatures d'enregistrements.*

2) $\mathcal{A} : Ob(\mathcal{C}) \to Ob(\mathcal{C})$ *tel que pour tout* $A \neq \mathbf{1}$

$$\mathcal{F}(\mathcal{A}(A)) = \mathcal{F}(A) \qquad et \qquad \mathfrak{L}(\mathcal{A}(A)) = \mathfrak{L}(A)$$

3) *Pour tout* A*, il existe un morphisme* α_A *de* $\mathcal{C}[A, \mathcal{A}(A)]$ *tel que*

$$p_{\mathcal{A}(A)} \circ \alpha_A = p_A$$

4) $\mathcal{A}(\mathbf{1}) = \mathbf{1}$ *et pour tout* A*,* $\mathcal{A}(\mathcal{A}(A)) = \mathcal{A}(A)$ *et* $\alpha_{\mathcal{A}(A)} = id_{\mathcal{A}(A)}$

5) *Pour tous* A*,* B *de* $Ob(\mathcal{C})$*, pour tout* a *de* $\mathtt{'A}$ *tel que* $a \notin Im(\mathfrak{L}_A)$*, pour tout* f *de* $\mathcal{C}[A, \mathcal{F}(B)]$*,*

$$\mathcal{A}(f *^a B) = f *^a \mathcal{A}(B) \qquad et \qquad q^a(f, \mathcal{A}(B)) \circ \alpha_{f *^a B} = \alpha_B \circ q^a(f, B)$$

6) *Avec les hypothèses précédentes,* $f *^a B = f *^a \mathcal{A}(B)$ *implique* $B = \mathcal{A}(B)$

7) *Si* $A \neq \mathcal{A}(A)$*, alors il existe un morphisme* $\mathcal{S}(A)$ *de* $\mathcal{C}[\mathcal{F}(A), A]$ *tel que*

$$p(A) \circ \mathcal{S}(A) = id_{\mathcal{F}(A)} \qquad et \qquad \mathcal{S}(A) \circ p(A) = id_A$$

8) *Pour* $A = \mathcal{A}(A)$, *pour tout morphisme* s *de* $\mathcal{C}[\mathcal{F}(A), A]$ *tel que* $p(A) \circ s = id_{\mathcal{F}(A)}$, *il existe un unique objet* $\mathcal{M}(s)$ *de* $\mathcal{C}$ *tel que*

$$\begin{array}{c} \mathcal{A}(\mathcal{M}(s)) = A \\ \textit{et} \\ A \neq \mathcal{M}(s) \\ \textit{et} \\ s = \alpha_{\mathcal{M}(s)} \circ \mathcal{S}(\mathcal{M}(s)) \end{array}
\qquad
\begin{tikzcd}
\mathcal{M}(s) \arrow[rr, "\alpha_{\mathcal{M}(s)}"] \arrow[dr, "p_{\mathcal{M}(s)}"] & & A \\
& \begin{array}{c}\mathcal{F}(\mathcal{M}(s)) \\ = \mathcal{F}(A)\end{array} \arrow[ul, bend left, "\mathcal{S}(\mathcal{M}(s))"] \arrow[ur, "s"'] &
\end{tikzcd}$$

10.1.1 Exemple : enregistrements de Pollack

En adaptant la présentation de Pollack des signatures avec champs manifestes (cf. [Pol00]) sur la présentation des enregistrements associatifs à gauche vue en section 8.1.1, on décrit ci-dessous un système de signatures avec champs manifestes en théorie des types. On va montrer que ces enregistrements *forment une catégorie de signature avec champs manifestes.* Pour les raisons invoquées en section 8.1.1.b, cette construction n'est pas valable en théorie des types, mais uniquement dans les modèles qui possèdent des η-règles : η-conversion et surjective pairing.

Les signatures avec champs manifestes sont données par les règles ci-dessous. La sorte $Sign$ est maintenant paramétrée par une liste d'étiquettes elles-mêmes annotées par $\mathbb{A}$ ou $\mathbb{M}$. Ces annotations indiquent si l'étiquette correspond dans la signature à un champs abstrait ou manifeste.

$$Sign : ([\mathbb{A}|\mathbb{M}] \times \text{`}\mathtt{A})^* \to Type$$

$$\frac{}{\{\} : Sign_{\emptyset}}\ \textbf{EForm} \qquad \frac{A : Sign_l \quad a \notin l \quad U : A \to Type}{\{A\ ;\ a : U\} : Sign_{l;\mathbb{A}a}}\ \textbf{AForm}$$

$$\frac{A : Sign_l \quad a \notin l \quad U : A \to Type \quad x : (X : A)(U\ X)}{\{A\ ;\ a : U = x\} : Sign_{l;\mathbb{M}a}}\ \textbf{MForm}$$

$$\frac{}{<> : \{\}}\ \textbf{EIntro} \qquad \frac{\{A\ ;\ a : U\} : Sign_{l;\mathbb{A}a} \quad X : A \quad x : (U\ X)}{<X\ ;\ a = x> : \{A\ ;\ a : U\}}\ \textbf{AIntro}$$

$$\frac{\{A\ ;\ a : U = x\} : Sign_{l;\mathbb{M}a} \qquad X : A}{<X\ ;\ a = (x\ X)> : \{A\ ;\ a : U = x\}}\ \textbf{MIntro}$$

- Les règles $\mathcal{F}$-DefC, p-Def, L-Def, $*$-Def, q-Def, $\curlywedge$ -Def de la section 8.1.1.c s'étendent sans problème à ce nouveau système, mais chacune d'elle donne 2 règles (pour le cas $\mathbb{A}$ et le cas $\mathbb{M}$). Ainsi, les règles L-Def et $*$-Def deviennent respectivement :

$$\frac{\{A\ ;\ a : U\} : Sign_{l;\mathbb{A}a}}{L_{\{A\ ;\ a:U\}}(<X\ ;\ a = t>) \overset{def}{=} t}\ \textit{L}\textbf{-DefA}$$

$$\frac{\{A\ ;\ a : U = x\} : Sign_{l;\mathbb{M}a}}{L_{\{A\ ;\ a:U=x\}}(<X\ ;\ a = t>) \overset{def}{=} t}\ \textit{L}\textbf{-DefM}$$

$$\frac{A_1 : Sign_{l_1} \quad b \notin l_1 \quad F : A_1 \to A_2 \quad \{A_2 \ ; \ a : U\} : Sign_{l_2;\mathbb{A}a}}{F *^b \{A_2 \ ; \ a : U\} \stackrel{def}{=} \{A_1 \ ; \ b : [X : A_1](U \ (F \ X))\}} *\textbf{-DefA}$$

$$\frac{A_1 : Sign_{l_1} \quad b \notin l_1 \quad F : A_1 \to A_2 \qquad \{A_2 \ ; \ a : U = x\} : Sign_{l_2;\mathbb{M}a}}{F *^b \{A_2 \ ; \ a : U = x\} \stackrel{def}{=} \{A_1 \ ; \ b : [X : A_1](U \ (F \ X)) = [X : A_1](x \ (F \ X))\}} *\textbf{-DefM}$$

Remarquons que dans les règles q-Def et $\curlywedge$-Def, seules les prémisses sont à adapter, le reste étant inchangé.

- La fonction $\mathcal{A}$ est donnée par la règle suivante :

$$\frac{A : Sign_{l;\mathbb{A}a}}{\mathcal{A}(A) \stackrel{def}{=} A} \mathcal{A}\textbf{-DefA} \qquad \frac{\{A \ ; \ a : U = x\} : Sign_{l;\mathbb{M}a}}{\mathcal{A}(\{A \ ; \ a : U = x\}) \stackrel{def}{=} \{A \ ; \ a : U\}} \mathcal{A}\textbf{-DefM}$$

- De même α est donnée par :

$$\frac{A : Sign_{l;\mathbb{A}a}}{\alpha_A \stackrel{def}{=} [X : A]X} \alpha\textbf{-DefA} \qquad \frac{A : Sign_{l;\mathbb{M}a}}{\alpha_A : A \to \mathcal{A}(A) \stackrel{def}{=} [X : A]X} \alpha\textbf{-DefM}$$

- La fonction $\mathcal{S}$ est donnée par :

$$\frac{\{A \ ; \ a : U = x\} : Sign_{l;\mathbb{M}a}}{\mathcal{S}(\{A \ ; \ a : U = x\}) \stackrel{def}{=} [X : A]{<}X \ ; \ a = (x \ X){>}} \mathcal{S}\textbf{-Def}$$

- La fonction $\mathcal{M}$ est donnée par :

$$\frac{A : Sign_{l;\mathbb{A}a} \quad s : \mathcal{F}(A) \to A \quad p_A \circ s = id_{\mathcal{F}(A)}}{\mathcal{M}(s) \stackrel{def}{=} \{\mathcal{F}(A) \ ; \ a : U = [X : A](L_A \ (s \ X))\}} \mathcal{M}\textbf{-Def}$$

10.2 Propriétés de base

Lemme 10.1 *Soit A de $Ob(\mathcal{C})$ avec $A \neq \mathcal{A}(A)$, $\mathcal{S}(A)$ est l'unique section de A.*

Preuve Soit s de $\mathcal{C}[\mathcal{F}(A), A]$ telle que $p(A) \circ s = id_{\mathcal{F}(A)}$. On a :

$$\begin{array}{l} s \\ \quad = id_A \circ s \\ \quad = \mathcal{S}(A) \circ p(A) \circ s \\ \quad = \mathcal{S}(A) \end{array}$$

□

Lemme 10.2 *Soient A et B de $Ob(\mathcal{C})$ avec $B \neq \mathcal{A}(B)$, soit a de $\mathfrak{A}$ avec $a \notin Im(\mathfrak{L}_A)$, et soit f de $\mathcal{C}[A, \mathcal{F}(B)]$. Comme, $p_B \circ (\mathcal{S}(B) \circ f) = f = f \circ id_A$, on peut construire le morphisme $id_A \curlywedge_{f *^a B} (\mathcal{S}(B) \circ f)$. De plus, comme $\mathcal{A}(f *^a B) \neq f *^a B$, $\mathcal{S}(f *^a B)$ existe. On a :*

$$\mathcal{S}(f *^a B) = id_A \curlywedge_{f *^a B} (\mathcal{S}(B) \circ f)$$

En particulier $q(f, B) \circ \mathcal{S}(f *^a B) = \mathcal{S}(B) \circ f$.

Preuve On pose $k = id_A \curlywedge_{f *^a B} (\mathcal{S}(B) \circ f)$. Par construction, $p_{f *^a B} \circ k = id_{\mathcal{F}(A)}$. Or $\mathcal{S}(f *^a B) \circ p_{f *^a B} = id_{f *^a B}$, donc $\mathcal{S}(f *^a B) = k$.

□

10.3 Propriété fondamentale

On montre maintenant que les catégories de signatures avec champs manifestes sont un cas particulier de catégories avec générateurs d'enregistrements.

Définition 10.2 *Pour tout A de $Ob(\mathcal{C})$. On définit $\mathcal{H}(A)$ de $Ob(\mathcal{C})$ et $\mathbb{G}(A)$ de $\mathcal{C}[\mathcal{H}(A), A]$ par induction mutuelle sur $|A|$.*

1) *Pour $A = \mathbf{1}$, on pose $\mathcal{H}(A) = \mathbf{1}$ et $\mathbb{G}(\mathbf{1}) = id_{\mathbf{1}}$.*

2) *Sinon, si $A = \mathcal{A}(A)$, on pose $\mathcal{H}(A) = \mathbb{G}(\mathcal{F}(A)) * A$ et $\mathbb{G}(A) = q(\mathbb{G}(\mathcal{F}(A)), A)$.*

3) *Sinon $A \neq \mathcal{A}(A)$, on pose $\mathcal{H}(A) = \mathcal{H}(\mathcal{F}(A))$ et $\mathbb{G}(A) = \mathcal{S}(A) \circ \mathbb{G}(\mathcal{F}(A))$.*

Lemme-Définition 10.1 *On construit un morphisme $\mathbb{G}'(A)$ de $\mathcal{C}[A, \mathcal{H}(A)]$ tel que $\mathbb{G}(A) \circ \mathbb{G}'(A) = id_A$.*

Construction Par induction sur $|A|$.

1) si $A = \mathbf{1}$, on pose $\mathbb{G}'(A) = id_{\mathbf{1}}$.

2) si $A = \mathcal{A}(A) \neq \mathbf{1}$, par récurrence, $\mathbb{G}(\mathcal{F}(A)) \circ \mathbb{G}'(\mathcal{F}(A)) = id_A$. Donc

$$\begin{aligned} & A \\ & \quad = (\mathbb{G}(\mathcal{F}(A)) \circ \mathbb{G}'(\mathcal{F}(A))) * A \\ & \quad = \mathbb{G}'(\mathcal{F}(A)) * (\mathbb{G}(F(A)) * A) \\ & \quad = \mathbb{G}'(\mathcal{F}(A)) * \mathcal{H}(A) \end{aligned}$$

On pose

$$\boxed{\mathbb{G}'(A) = q(\mathbb{G}'(\mathcal{F}(A)), \mathcal{H}(A))}$$

On a bien $\mathbb{G}(A) \circ \mathbb{G}'(A) = q(\mathbb{G}(\mathcal{F}(A)), A) \circ q(\mathbb{G}'(\mathcal{F}(A)), \mathbb{G}(\mathcal{F}(A)) * A) = id_A$.

3) sinon, on pose $\boxed{\mathbb{G}'(A) = \mathbb{G}'(\mathcal{F}(A)) \circ p_A}$. On a

$$\begin{aligned} & \mathbb{G}(A) \circ \mathbb{G}'(A) \\ & \quad = \mathcal{S}(A) \circ \mathbb{G}(\mathcal{F}(A)) \circ \mathbb{G}'(\mathcal{F}(A)) \circ p_A \\ & \quad = id_A \end{aligned}$$

□

Lemme 10.3 *Pour tout A, $\mathbb{G}'(A) \circ \mathbb{G}(A) = id_{\mathcal{H}(A)}$.*

Preuve Par induction sur $|A|$.

1) si $A = \mathbf{1}$, trivial.

2) si $A = \mathcal{A}(A) \neq \mathbf{1}$,

$$\begin{aligned} & \mathbb{G}'(A) \circ \mathbb{G}(A) \\ & \quad = q(\mathbb{G}'(\mathcal{F}(A)), \mathcal{H}(A)) \circ q(\mathbb{G}(\mathcal{F}(A)), A) \\ & \quad = q(\mathbb{G}'(\mathcal{F}(A)), \mathcal{H}(A)) \circ q(\mathbb{G}(\mathcal{F}(A)), \mathbb{G}'(\mathcal{F}(A)) * \mathcal{H}(A)) \\ & \quad = q(\mathbb{G}'(\mathcal{F}(A)) \circ \mathbb{G}'(\mathcal{F}(A)), \mathcal{H}(A)) \\ & \quad = q(id_{\mathcal{F}(\mathcal{H}(A))}, \mathcal{H}(A)) \\ & \quad = id_{\mathcal{H}(A)} \end{aligned}$$

3) sinon,

$$\begin{aligned}&\mathbb{G}'(A)\circ\mathbb{G}(A)\\&\quad=\mathbb{G}'(\mathcal{F}(A))\circ p_A\circ\mathcal{S}(A)\circ\mathbb{G}(\mathcal{F}(A))\\&\quad=id_{\mathcal{H}(F(A))}\\&\quad=id_{\mathcal{H}(A)}\end{aligned}$$

□

Remarque 10.1 *Pour tout A, $\mathbb{G}(A)$ est inversible et on a*

$$\mathbb{G}^{-1}(A)=\mathbb{G}'(A)$$

Définition 10.3 *Pour tout f de $\mathcal{C}[A,B]$, on pose*

$$\mathcal{H}(f)=\mathbb{G}^{-1}(B)\circ f\circ\mathbb{G}(A)$$

Lemme 10.4 *Pour tout f, $\mathcal{H}(f)\in\mathcal{C}[\mathcal{H}(A),\mathcal{H}(B)]$ et*

$$f\circ\mathbb{G}(A)=\mathbb{G}(B)\circ\mathcal{H}(f)$$

Preuve Immédiat.

□

Remarque 10.2 *Il est immédiat de vérifier que $\mathcal{H}$ définit un foncteur et $\mathbb{G}$ une transformation naturelle de $\mathcal{H}$ vers $Id_{\mathcal{C}}$.*

Définition 10.4 *Pour $A=\mathcal{A}(A)$, pour tout morphisme s de $\mathcal{C}[\mathcal{H}(\mathcal{F}(A)),\mathcal{H}(A)]$ tel que $p_{\mathcal{H}(A)}\circ s=id_{\mathcal{H}(\mathcal{F}(A))}$, on pose $\mathcal{M}'(s)=\mathcal{M}(\mathbb{G}(A)\circ s\circ\mathbb{G}^{-1}(\mathcal{F}(A)))$.*

Preuve Cette définition est valide car $\mathbb{G}(A)\circ s\circ\mathbb{G}^{-1}(\mathcal{F}(A))$ est une section de A. En effet :

$$\begin{aligned}&p_A\circ G(A)\circ s\circ\mathbb{G}^{-1}(\mathcal{F}(A))\\&\quad=\mathbb{G}(\mathcal{F}(A))\circ p_{\mathcal{H}(A)}\circ s\circ\mathbb{G}^{-1}(\mathcal{F}(A))\\&\quad=id_{\mathcal{F}(A)}\end{aligned}$$

□

Lemme 10.5 *Sous les mêmes hypothèses, $\mathcal{A}(\mathcal{M}'(s))=A$ et*

$$\mathbb{G}(A)\circ s=\alpha_{\mathcal{M}'(s)}\circ\mathbb{G}(\mathcal{M}'(s))$$

Preuve

$$\begin{aligned}&\alpha_{\mathcal{M}'(s)}\circ\mathbb{G}(\mathcal{M}'(s))\\&\quad=\alpha_{\mathcal{M}'(s)}\circ\mathcal{S}(\mathcal{M}'(s))\circ p_{M'(s)}\circ\mathbb{G}(\mathcal{M}'(s))\\&\quad=(\alpha_{\mathcal{M}'(s)}\circ\mathcal{S}(\mathcal{M}'(s)))\circ\mathbb{G}(\mathcal{F}(\mathcal{M}'(s)))\\&\quad=(\mathbb{G}(A)\circ s\circ\mathbb{G}^{-1}(\mathcal{F}(A)))\circ\mathbb{G}(\mathcal{F}(A))\\&\quad=\mathbb{G}(A)\circ s\end{aligned}$$

□

Lemme 10.6 *Sous les mêmes hypothèses, soit $B\neq A$ tel que $\mathcal{A}(B)=A$ et $\mathbb{G}(A)\circ s=\alpha_B\circ\mathbb{G}(B)$, alors $B=\mathcal{M}'(s)$.*

Preuve

$$
\begin{aligned}
&\mathbb{G}(A) \circ s \circ \mathbb{G}^{-1}(\mathcal{F}(A)) \\
&\quad = \alpha_B \circ \mathbb{G}(B) \circ \mathbb{G}^{-1}(\mathcal{F}(A)) \\
&\quad = \alpha_B \circ \mathcal{S}(B) \circ \mathbb{G}(\mathcal{F}(B)) \circ \mathbb{G}^{-1}(\mathcal{F}(B)) \\
&\quad = \alpha_B \circ \mathcal{S}(B)
\end{aligned}
$$

□

Théorème 10.1 $< \mathcal{C}, \mathfrak{A}, \mathcal{F}, \mathfrak{L}, \mathbf{1}, p, *, q, \mathcal{A}, \alpha, \mathcal{H}, \mathbb{G}, \mathcal{M}' >$ *définit une catégorie de générateurs d'enregistrements.*

Preuve Découle des lemmes précédents.

□

10.4 Incompatibilité entre la notion de sur-signature pleinement abstraite et l'axiome "$\mathcal{S}$"

Soient A et B deux signatures, on dit que B est une *sur-signature pleinement abstraite* de A si $B \propto A$ avec $\mathfrak{M}_B = \emptyset$. On rappelle (cf. section 9.4) qu'une notion de sur-signature pleinement abstraite est une fonction $\mathcal{T}$ telle que pour tout A, $\mathcal{T}(A)$ est une sur-signature pleinement abstraite de A et que pour tout B telle que $B \propto A$, alors $\mathcal{T}(B) = \mathcal{T}(A)$.

10.4.1 Etude d'un contre-exemple

On va ici montrer intuitivement que dans les exemples "intéressants" de signatures avec champs manifestes, il n'existe pas de notion de sur-signature pleinement abstraite.

Par exemple, supposons que $list\_n : (n : nat)Set$ représente les listes d'entiers de longueur n, et on considère la signature suivante :

$$A = \{T : Set = nat \ ; \ x : T \ ; \ l : (list\_n \ x)\}$$

Dans le système à la Pollack, cette signature s'écrit formellement comme ci-dessous (où on utilise la notation "." pour désigner l'opération d'accès au champs, qui n'a pas été définie ici, mais que le lecteur peut facilement imaginer) :

$$\{\{\{\{\} \ ; \ T : [X_1]Set = [X_1]nat\} \ ; \ x : [X_2]X_2.T\} \ ; \ l : [X_3](list\_n \ x)\}$$

Pour plus de lisibilité, on a omis ici les types des variables liées X_1, X_2 et X_3 qui sont respectivement :

- $\{\}$
- $\{\{\} \ ; \ T : [X_1]Set = [X_1]nat\}$
- $\{\{\{\} \ ; \ T : [X_1]Set = [X_1]nat\} \ ; \ x : [X_2]X_2.T\}$

Considérons maintenant $\mathcal{F}(A)$, et cherchons les signatures B telles que $B \ltimes \mathcal{F}(A)$ avec $\mathfrak{M}_B = \emptyset$. Une telle signature est de la forme $B = \{\{\{\} \ ; \ T : [X_1]Set\} \ ; \ x : f\}$ où f est une fonction de type $\{\{\} \ ; \ T : [X_1]Set\} \rightarrow Type$. Comme on doit avoir $\mathcal{F}(A) = \alpha_{\mathcal{F}^2(A)} * B$, il est facile de voir que $f \circ \alpha_{\mathcal{F}^2(A)} = [X_2]X_2.T$ (où X_2 a toujours implicitement le type ci-dessus). Autrement dit, on a

$$f = [X : \{\{\} \ ; \ T : [X_1]Set\}]X.T$$

Avec la notation "informelle", $B = \{T : Set \ ; \ x : T\}$.

En reproduisant le même raisonnement, il est facile de se convaincre que il n'existe pas de C tel que $C \ltimes A$ avec $\mathfrak{M}_C = \emptyset$: car il n'est pas possible de typer le champ d'étiquette l dans le contexte B.

Remarquons cependant qu'il est intuitivement possible de lui trouver une signature isomorphe qui admette une sur-signature pleinement abstraite. Par exemple :

$$A' = \{T : Set = nat \ ; \ x : nat \ ; \ l : (list\_n \ x)\}$$

ou encore

$$A'' = \{T : Set = nat \ ; \ P : T \rightarrow Set = [n : nat](list\_n \ n); \ x : T; \ l : (P \ x)\}$$

On serait alors tenter de dire que le fait qu'un tel C n'existe pas provient d'une faiblesse du système de signature avec champs manifestes choisi. Par exemple, on pourrait vouloir imaginer un système dans lequel la signature A ci-dessus est égale à la signature A' ci-dessus (il n'est pas possible de considérer que $A = A''$ à cause de la différence du nombre d'étiquettes).

Mais $\mathcal{F}(A)$ aurait alors 2 sur-signatures pleinement abstraites : la signature B précedente et la signature $B' = \{T : Set \ ; \ x : nat\}$. Dans les systèmes intéressants, ces signatures B et B' ne peuvent pas être égales. Supposons maintenant que ce système admette une notion de sur-signature pleinement abstraite $\mathcal{T}$. Du fait que $B \propto \mathcal{F}(A)$ on déduit que $\mathcal{T}(B) = \mathcal{T}(\mathcal{F}(A))$ et de même $\mathcal{T}(B') = \mathcal{T}(\mathcal{F}(A))$. Donc $\mathcal{T}(B) = \mathcal{T}(B')$. Or on a vu que $\mathfrak{M}_B = \emptyset$ implique $\mathcal{T}(B) = B$. Donc $B = B'$. On en déduit qu'un tel système n'admet pas de notion de sur-signature pleinement abstraite.

En conclusion, une CG qui peut représenter une telle signature A n'admet pas de notion de sur-signature pleinement abstraite.

10.4.2 Conséquences de ce contre-exemple

Dans un système à la Pollack, le fait que A soit un objet de la catégorie empêche aussi l'existence de $I[X, Y]$ lorsque $X \blacktriangleleft Y$. En effet, on peut montrer que $\{T : Set\} \blacktriangleleft A$, mais on ne peut pas trouver B tel que $\{T : Set\} \lhd B \propto A$: un tel B serait justement une sur-signature pleinement abstraite de A. En fait, dans ce cas, $\lhd$ et $\propto$ commutent mal : si $X \propto Y$ et $Y \lhd Z$, il n'existe pas forcément Y' tel que $X \lhd Y'$ et $Y' \propto Z$.

Pour avoir une notion de sur-signature pleinement abstraite, on a envie d'interdire les signatures, comme A, dont les types de certains champs dépendent de manière transparente des autres champs. Malheureusement, cela ne suffit pas résoudre le problème. Montrons par exemple comment construire A dans une CS avec champs manifestes à partir de signatures qui n'ont pas de dépendances transparentes. On a vu par exemple que $\mathcal{F}(A) = \{T : Set = nat \ ; \ x : T\}$ est une telle signature, qui admet $\{T : Set \ ; \ x : T\}$ comme sur-signature pleinement abstraite. On montre que $\mathcal{H}(\{T : Set = nat \ ; \ x : T\}) = \{x : nat\}$ et que $\mathbb{G}(\{T : Set = nat \ ; \ x : T\})$ est la fonction qui à $<x = n>$ associe $<T = nat \ ; \ x = n>$. On a vu que dans les catégories de signatures avec champs manifestes, $\mathbb{G}$ est inversible : posons $f = \mathbb{G}^{-1}(\{T : Set = nat \ ; \ x : T\})$. La fonction f associe à $<T = nat \ ; \ x = n>$ l'enregistrement $<x = n>$. Soit $C = \{x : nat \ ; \ l : (list\_n \ x)\}$. La signature C est pleinement abstraite, et il est facile de voir que $f * C$ égale A.

Ce dernier exemple montre donc qu'il ne suffit pas de "supprimer" des objets pour avoir une notion de sur-signature pleinement abstraite. Il faut aussi "supprimer" des morphismes (comme le morphisme f ci-dessus). Ainsi, dans $\mathcal{C}_{\ltimes}$, on a imposé que toute signature ait une sur-signature pleinement abstraite. Et, étant donné 2 signatures X et Y de sur-signatures pleinement abstraites respectives X' et Y', on a aussi imposé que pour tout morphisme f de X dans Y, il existe un morphisme de f' de X' dans Y' tel que $f' \circ \mathbf{C}(X, X') = \mathbf{C}(Y, Y') \circ f$. Autrement dit, $\mathcal{T}_{\ltimes}$ est un foncteur, et la transformation $X \mapsto \mathbb{C}(X, \mathcal{T}_{\ltimes}(X))$ est une transformation naturelle de $Id_{\mathcal{C}_{\ltimes}}$ vers $\mathcal{T}_{\ltimes}$. Supposer l'axiome $\mathcal{S}$ avec ces contraintes revient à interdire les dépendances transparentes. En effet avec ces contraintes, supposer que pour tout X tel que $X \neq \mathcal{A}(X)$, il existe une section de X, revient à dire que la définition sur le dernier champ de X doit s'exprimer comme une section de $\mathcal{T}(X)$. Autrement dit, le dernier champ de X ne peut pas dépendre de manière opaque des autres champs de X.

Chapitre 11

CG avec effacement de définition (ou CGED)

On a vu que les catégories de générateurs d'enregistrements permettent de définir une notion de *définition d'un champ encore abstrait.* Pour décrire l'opération de redéfinition, on décide de passer par l'intermédiaire d'une opération d'effacement d'une définition. C'est une idée qui se retrouve par exemple dans les opérations sur les mixins (cf. [Anc98]).

Pour que cette fonction d'effacement soit totale (c'est-à-dire toujours possible), il faut avoir une notion de sur-signature pleinement abstraite $\mathcal{T}$. Cette contrainte garantit que les types des champs gardent un sens même si on efface les définitions des champs dont ils dépendent.

De plus, cette opération d'effacement doit non seulement effacer la définition du champ désiré, mais aussi procéder à l'effacement sur tous les autres champs qui en dépendent de manière transparente. Il va donc falloir introduire les notions de dépendances transparentes et dépendances opaques. Pour tout objet A, on suppose qu'il existe un objet $\mathbb{D}(A)$ qui représente la plus petite (au sens de "$\blacktriangleleft$") signature définissable telle que $\mathbb{D}(A) \blacktriangleleft A$, qui contiennent la dernière étiquette de A, et telle que $\mathcal{A}(A) = A \Leftrightarrow \mathcal{A}(\mathbb{D}(A)) = \mathbb{D}(A)$. L'objet $\mathbb{D}(A)$ est donc la plus petite sur-signature de A dans laquelle le dernier champ de A a un sens. Les champs manifestes de $\mathbb{D}(A)$ sont donc les champs dont le champ de A dépend de manière transparente. Les champs abstraits de $\mathbb{D}(A)$ sont les champs dont le dernier champ de A dépend de manière opaque. Remarquons en particulier que si ce dernier champ de A est abstrait (ie $\mathcal{A}(A) = A$), alors comme $\mathcal{T}(A) \blacktriangleleft A$ avec $\mathcal{A}(\mathcal{T}(A)) = \mathcal{T}(A)$, on a $\mathbb{D}(A) \blacktriangleleft \mathcal{T}(A)$: cela signifie en particulier que tous les champs de $\mathbb{D}(A)$ sont abstraits, il n'y a donc pas de dépendance transparente. Cela correspond à l'intuition dans FocKer que les types des méthodes ne peuvent pas dépendre de manière transparentes des autres méthodes.

Ceci dit, pour qu'une telle $\mathbb{D}(A)$ ait le sens qu'on en attend, il faut que l'oubli de définition soit surjectif (ou régulier) : en particulier, on ne doit pas déformer l'information par oubli. Par exemple, si $A = \{x : nat = 0\ ;\ y : (eq\ x\ 0) = (refl\_eq\ 0)\}$ et $B = \{x : nat\ ;\ y : (eq\ x\ x) = (refl\_eq\ x)\}$, il faut interdire les diagrammes du genre :

$$\begin{tikzcd}
\alpha_{\mathcal{F}(A)} * B = A \arrow[r, "q(\alpha_{\mathcal{F}(A)}, B)"] \arrow[d, "p_A"'] & B \arrow[d, "p_B"] \\
\{x : nat = 0\} \arrow[r, "\alpha_{\mathcal{F}(A)}"'] & \{x : nat\}
\end{tikzcd}$$

Dans ce diagramme, on change au cours de l'effacement la dépendance transparente[1] entre x et y dans A, en une dépendance opaque dans B.

On va donc exiger que pour tout objet A, p_A et α_A soient ultra-réguliers. On a vu que les CS raisonnables ont la propriété d'ultra-régularité de p_A. En particulier, les CG raisonnables ont aussi cette propriété. On a vu par exemple que cette propriété se transmet via le foncteur $\cdot_\alpha$.

Par contre, on peut trouver des CG raisonnables qui n'ont pas la propriété d'ultra-régularité de α_A. Si on simplifie l'exemple précédent il est facile de montrer que dans un système de signatures avec champs manifestes à la Pollack $X = \{x : nat = 0\ ;\ y : (eq\ x\ 0)\}$ admet pour sur-signatures pleinement abstraites $Y = \{x : nat\ ;\ y : (eq\ x\ x)\}$ et $Z = \{x : nat\ ;\ y : (eq\ x\ 0)\}$ sans que ces 2 dernières signatures soit égales :

$$X = \alpha_{\mathcal{F}(X)} * Y = \alpha_{\mathcal{F}(X)} * Z$$

On n'a malheureusement pas eu le temps de trouver un modèle pour une CG avec ultra-régularité de α_A (et de p_A). On donne néanmoins des pistes pour le faire en fin de section (cf. 11.6). On peut déjà remarquer que le contre-exemple de régularité de α_A ci-dessus n'est pas possible dans la CG $\mathfrak{C}_\alpha$, car cette catégorie admet une notion de sur-signature pleinement abstraite : pour toute signature A, il existe une unique signature B telle que $B \propto_{\mathfrak{M}_A} A$.

Avant de donner l'axiomatisation de FocKer, on étudie ces propriétés de régularités.

11.1 CG régulières (ou CGR)

La régularité porte sur deux aspects : régularité sur les signatures et régularité sur les morphismes. *Si $\mathfrak{C}$ est une CG, on dit que $\mathfrak{C}$ est régulière ssi pour tout objet A, p_A et α_A sont ultra-réguliers.* On détaille maintenant les propriétés de régularité des signatures et des morphismes.

11.1.1 Régularité sur les signatures

Définition 11.1 *On dit que $\mathfrak{C}$ est une catégorie de générateurs d'enregistrements régulière sur les signatures, ssi $\mathfrak{C}$ est régulière sur les signatures en tant que catégorie des signatures d'enregistrements et si, pour tout A, B, C de $Ob(\mathcal{C})$, pour tout n tel que $\mathcal{A}(A) = \mathcal{F}^n(B) = \mathcal{F}^n(C)$, $\alpha_A *^n B = \alpha_A *^n C$ implique $B = C$.*

Remarque 11.1 *La définition précédente est valide du point de vue des étiquettes, car si $\mathcal{A}(A) = \mathcal{F}^n(B)$, comme $\mathfrak{L}_{\mathcal{A}(A)} = \mathfrak{L}_A$ alors $Im(\mathfrak{L}_B) \cap Im(\mathfrak{L}_A) = Im(\mathfrak{L}_{\mathcal{F}^n(B)})$.*

Lemme 11.1 *Soit $\mathfrak{C}$ est régulière sur les signatures. Soient A et B tels que $A \blacktriangleleft B$. Pour tout C et D, et pour tout n tel que $Im(\mathfrak{L}_C) \cap Im(\mathfrak{L}_B) = Im(\mathfrak{L}_{\mathcal{F}^n(C)}) \cap Im(\mathfrak{L}_B)$, $Im(\mathfrak{L}_D) \cap Im(\mathfrak{L}_B) = Im(\mathfrak{L}_{\mathcal{F}^n(D)}) \cap Im(\mathfrak{L}_B)$, et $\mathcal{F}^n(C) = \mathcal{F}^n(D) = A$, alors $\mathbb{C}(B, A) *^n C = \mathbb{C}(B, A) *^n D$ implique $C = D$.*

Preuve Par induction sur $|B|$ (pour tout C, D et n).

1) Si $B = \mathbf{1}$. $A = \mathbf{1}$. $id_{\mathbf{1}} *^n C = id_{\mathbf{1}} *^n D$ implique $C = D$.

2) Si $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$, $B \neq \mathcal{A}(B)$ et $A = \mathcal{A}(A)$. Alors, $\mathcal{F}(A) \blacktriangleleft \mathcal{F}(B)$, $\mathcal{A}(B) = \mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$, et $\mathbf{C}(B, A) = q(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), A) \circ \alpha_B$.

[1]En effet $(refl\_eq\ 0)$ a le type $(eq\ 0\ 0)$. Pour lui donner le type $(eq\ x\ 0)$ il faut donc utiliser le fait que x égale 0.

On a :

$$\begin{aligned}&\mathbf{C}(B,A) *^n C\\ &\quad = (q(\mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)),A) \circ \alpha_B) *^n C\\ &\quad = \alpha_B *^n (\mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)) *^{n+1} C)\end{aligned}$$

Donc par régularité de α_B, $\mathbf{C}(B,A) *^n C = \mathbf{C}(B,A) *^n D$ implique $\mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)) *^{n+1} C = \mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)) *^{n+1} D$.

Et par hypothèse de récurrence, $\mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)) *^{n+1} C = \mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)) *^{n+1} D$ implique $C = D$.

3) Si $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$. Alors, $\mathcal{F}(A) \blacktriangleleft \mathcal{F}(B)$. $B = \mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)) * A$.
$\mathbf{C}(B,A) = q(\mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)),A)$.
$\mathbf{C}(B,A) *^n C = \mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)) *^{n+1} C$. Par hypothèse de récurrence, $\mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)) *^{n+1} C = \mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)) *^{n+1} D$ implique $C = D$.

4) Sinon, $A \blacktriangleleft \mathcal{F}(B)$, et $\mathbf{C}(B,A) = \mathbf{C}(\mathcal{F}(B),A) \circ p_B$.
$\mathbf{C}(B,A) *^n C = (\mathbf{C}(\mathcal{F}(B),A) \circ p_B) *^n C = p_B *^n (\mathbf{C}(\mathcal{F}(B),A) *^n C)$. Donc $\mathbf{C}(B,A) *^n C = \mathbf{C}(B,A) *^n D$ implique $(\mathbf{C}(\mathcal{F}(B),A) *^n C = (\mathbf{C}(\mathcal{F}(B),A) *^n D$. Par hypothèse de récurrence, cela implique $C = D$.

□

Lemme 11.2 *Soit $\mathfrak{C}$ régulière sur les signatures. Soient A, B et C. Si $A \blacktriangleleft C$, $B \blacktriangleleft C$, $Im(\mathfrak{L}(A)) \subset Im(\mathfrak{L}(B))$ et $\mathfrak{M}_A \subset \mathfrak{M}_B$ alors $A \blacktriangleleft B$.*

Preuve Par induction sur $|C|$.

1) $C = \mathbf{1}$. $B = \mathbf{1}$ et $A = \mathbf{1}$. Evident.

2) Si $\mathfrak{L}_C(|C|) = \mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$.
On a : $\mathcal{F}(A) \blacktriangleleft \mathcal{F}(C)$ et $\mathcal{F}(B) \blacktriangleleft \mathcal{F}(C)$. Comme $Im(\mathfrak{L}(\mathcal{F}(A))) \subset Im(\mathfrak{L}(\mathcal{F}(B)))$, et $\mathfrak{M}_{\mathcal{F}(A)} \subset \mathfrak{M}_{\mathcal{F}(B)}$, par hypothèse de récurrence, $\mathcal{F}(A) \blacktriangleleft \mathcal{F}(B)$. De plus $\mathbf{C}(\mathcal{F}(C),\mathcal{F}(A)) = \mathbf{C}(\mathcal{F}(C),\mathcal{F}(B)) \circ \mathbf{C}(\mathcal{F}(B),\mathcal{F}(A))$.

2.1) Si $C \neq \mathcal{A}(C)$ et $B = \mathcal{A}(B)$. Comme $\mathfrak{M}_A \subset \mathfrak{M}_B$ alors $A = \mathcal{A}(A)$.
$\mathcal{A}(C) = \mathbf{C}(\mathcal{F}(C),\mathcal{F}(A)) * A = \mathbf{C}(\mathcal{F}(C),\mathcal{F}(B)) * (\mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)) * A)$.
D'après le lemme précédent, $B = \mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)) * A$. On en déduit $A \blacktriangleleft B$.

2.2) Si $C \neq \mathcal{A}(C)$ et $B \neq \mathcal{A}(B)$ et $A = \mathcal{A}(A)$.
$\mathcal{A}(C) = \mathbf{C}(\mathcal{F}(C),\mathcal{F}(B)) * (\mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)) * A)$.
D'après le lemme précédent, $\mathcal{A}(B) = \mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)) * A$.
On en déduit $A \blacktriangleleft B$.

2.3) Sinon, $C = \mathcal{A}(C)$ équivaut à $B = \mathcal{A}(B)$ et équivaut à $A = \mathcal{A}(A)$.
$C = \mathbf{C}(\mathcal{F}(C),\mathcal{F}(B)) * (\mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)) * A)$.
D'après le lemme précédent, $B = \mathbf{C}(\mathcal{F}(B),\mathcal{F}(A)) * A$.
On en déduit $A \blacktriangleleft B$.

3) Si $\mathfrak{L}_C(|C|) = \mathfrak{L}_B(|B|)$ et $\mathfrak{L}_C(|C|) \notin Im(\mathfrak{L}_A)$. On a : $A \blacktriangleleft \mathcal{F}(C)$ et $\mathcal{F}(B) \blacktriangleleft \mathcal{F}(C)$. Comme, $Im(\mathfrak{L}(A)) \subset Im(\mathfrak{L}(\mathcal{F}(B)))$, et $\mathfrak{M}_A \subset \mathfrak{M}_{\mathcal{F}(B)}$ par hypothèse de récurrence, $A \blacktriangleleft \mathcal{F}(B)$.
On en déduit $A \blacktriangleleft B$.

4) Sinon, $A \blacktriangleleft \mathcal{F}(C)$ et $B \blacktriangleleft \mathcal{F}(C)$. Par hypothèse de récurrence, $A \blacktriangleleft B$.

□

Le théorème suivant montre que la régularité implique qu'il n'y a qu'une seule façon d'oublier...

Théorème 11.1 *Soit $\mathfrak{C}$ est régulière sur les signatures. Soient A, B et C tels que $A \blacktriangleleft C$ et $B \blacktriangleleft C$ avec $\mathfrak{M}_A = \mathfrak{M}_B$ et $Im(\mathfrak{L}_A) = Im(\mathfrak{L}_B)$, alors $A = B$.*

Preuve D'après le lemme précédent, on a $A \blacktriangleleft B$. Par le lemme 9.19, on en déduit que $A = B$.

□

Lemme 11.3 *Si $\mathfrak{C}$ est régulière sur les signatures, et si $\mathcal{T}$ est une fonction de $Ob(\mathcal{C})$ dans $Ob(\mathcal{C})$ telle que pour tout A, $\mathcal{T}(A) \propto_{\mathfrak{M}_A} A$, alors $\mathcal{T}$ est une notion de sur-signature pleinement abstraite de $\mathfrak{C}$.*

Preuve Se déduit directement du lemme précédent.

□

11.1.2 Régularité sur les morphismes

Définition 11.2 *On dit que $\mathfrak{C}$ est une catégorie de générateurs d'enregistrements régulière sur les morphismes, ssi $\mathfrak{C}$ est régulière sur les morphismes en tant que catégorie des signatures d'enregistrements et ssi pour tout A, B de de $Ob(\mathcal{C})$ et pour tout n tel que $\mathcal{A}(A) = \mathcal{F}^n(B)$, pour tout C, f et g de $\mathcal{C}[B, C]$, $f \circ q^n(\alpha_A, B) = g \circ q^n(\alpha_A, B)$ implique $f = g$.*

Lemme 11.4 *Soit $\mathfrak{C}$ régulière sur les morphismes. Soient A et B tels que $A \blacktriangleleft B$. Pour tout n et pour tout C tels que $Im(\mathfrak{L}_C) \cap Im(\mathfrak{L}_B) = Im(\mathfrak{L}_{\mathcal{F}^n(C)}) \cap Im(\mathfrak{L}_B)$, et $A = \mathcal{F}^n(C)$ et pour tout D, f et g de $\mathcal{C}[C, D]$, $f \circ q^n(\mathbf{C}(B, A), C) = g \circ q^n(\mathbf{C}(B, A), C)$ implique $f = g$.*

Preuve Par induction sur $|B|$ (pour tout C, D, n, f et g).

1) Si $B = \mathbf{1}$. $A = \mathbf{1}$. $f \circ id_C = g \circ id_C$ implique $f = g$.

2) Si $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$, $B \neq \mathcal{A}(B)$ et $A = \mathcal{A}(A)$. Alors, $\mathcal{F}(A) \blacktriangleleft \mathcal{F}(B)$, $\mathcal{A}(B) = \mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$ et $\mathbf{C}(B, A) = q(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), A) \circ \alpha_B$.

$$\begin{aligned}
&q^n(\mathbf{C}(B, A), C)\\
&\quad = q^n(q(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), A) \circ \alpha_B, C)\\
&\quad = q^n(q(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), A), C) \circ q^n(\alpha_B, \mathbf{C}(\mathcal{A}(B), A) *^n C)\\
&\quad = q^{n+1}(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), C) \circ q^n(\alpha_B, \mathbf{C}(\mathcal{A}(B), A) *^n C)
\end{aligned}$$

Par épimorphisme de $q^n(\alpha_B, \mathbf{C}(\mathcal{A}(B), A) *^n C)$, $f \circ q^n(\mathbf{C}(B, A), C) = g \circ q^n(\mathbf{C}(B, A), C)$ implique $f \circ q^{n+1}(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), A), C) = g \circ q^{n+1}(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), A), C)$.

Et par hypothèse de récurrence, cette dernière égalité implique $f = g$.

3) Si $\mathfrak{L}_B(|B|) = \mathfrak{L}_A(|A|)$. Alors, $\mathcal{F}(A) \blacktriangleleft \mathcal{F}(B)$, $B = \mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$ et $\mathbf{C}(B, A) = q(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), A)$.

$q^n(\mathbf{C}(B, A), C) = q^{n+1}(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), C)$. Par hypothèse de récurrence, $f \circ q^{n+1}(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), C) = g \circ q^{n+1}(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), C)$ implique $f = g$.

4) Sinon, $A \blacktriangleleft \mathcal{F}(B)$, et $\mathbf{C}(B, A) = \mathbf{C}(\mathcal{F}(B), A) \circ p_B$.

$q^n(\mathbf{C}(B, A), C) = q^n(\mathbf{C}(\mathcal{F}(B), A), C) \circ q^n(p_B, \mathbf{C}(\mathcal{F}(B), A) *^n C)$

Donc $f \circ q^n(\mathbf{C}(B, A), C) = g \circ q^n(\mathbf{C}(B, A), C)$ implique $f \circ q^n(\mathbf{C}(\mathcal{F}(B), A), C) = g \circ q^n(\mathbf{C}(\mathcal{F}(B), A), C)$. Ce qui par hypothèse de récurrence implique $f = g$.

□

Définition 11.3 *Soient f et g deux morphismes de $\mathcal{C}$. On dira que $f \blacktriangleleft g$ ssi*

$$Dom(f) \blacktriangleleft Dom(g) \qquad et \qquad Im(f) \blacktriangleleft Im(g)$$

Et,

$$\mathbb{C}(Im(g), Im(f)) \circ g = f \circ \mathbb{C}(Dom(g), Dom(f))$$

Lemme 11.5 *La relation $\blacktriangleleft$ est une relation d'ordre sur les morphismes.*

Preuve Pour la réflexivité et l'antisymétrie, c'est trivial. Pour la transivité, on suppose $f \blacktriangleleft g$ et $g \blacktriangleleft h$. On a :

$$\begin{aligned} &\mathbb{C}(Im(h), Im(f)) \circ h \\ &\quad = \mathbb{C}(Im(g), Im(f)) \circ \mathbb{C}(Im(h), Im(g)) \circ h \\ &\quad = \mathbb{C}(Im(g), Im(f)) \circ g \circ \mathbb{C}(Dom(h), Dom(g)) \\ &\quad = f \circ \mathbb{C}(Dom(g), Dom(f)) \circ \mathbb{C}(Dom(h), Dom(g)) \\ &\quad = f \circ \mathbb{C}(Dom(h), Dom(f)) \end{aligned}$$

□

Lemme 11.6 *Soient f_1, f_2, g_1 et g_2 des morphismes de $\mathcal{C}$, tels que $f_1 \blacktriangleleft f_2$ et $g_1 \blacktriangleleft g_2$, $Im(f_1) = Dom(g_1)$ et $Im(f_2) = Dom(g_2)$.*

On a

$$(g_1 \circ f_1) \blacktriangleleft (g_2 \circ f_2)$$

Preuve

$$\begin{aligned} &\mathbb{C}(Im(g_2 \circ f_2), Im(g_1 \circ f_1)) \circ g_2 \circ f_2 \\ &\quad = \mathbb{C}(Im(g_2), Im(g_1)) \circ g_2 \circ f_2 \\ &\quad = g_1 \circ \mathbb{C}(Dom(g_2), Dom(g_1)) \circ f_2 \\ &\quad = g_1 \circ f_1 \circ \mathbb{C}(Dom(f_2), Dom(f_1)) \\ &\quad = g_1 \circ f_1 \circ \mathbb{C}(Dom(g_2 \circ f_2), Dom(g_2 \circ f_1)) \end{aligned}$$

□

Lemme 11.7 *Soient f et g deux morphismes de $\mathcal{C}$. Soient A et B tels que $Dom(f) \blacktriangleleft A \blacktriangleleft Dom(g)$ et $Im(f) \blacktriangleleft B \blacktriangleleft Im(g)$, on a :*

$$f \blacktriangleleft g \text{ ssi } (f \circ \mathbb{C}(A, Dom(f))) \blacktriangleleft (\mathbb{C}(Im(g), B) \circ g)$$

Preuve On suppose $f \blacktriangleleft g$:

$$\begin{aligned} &\mathbb{C}(B, Im(f)) \circ (\mathbb{C}(Im(g), B) \circ g) \\ &\quad = \mathbb{C}(Im(g), Im(f)) \circ g \\ &\quad = f \circ \mathbb{C}(Dom(g), Dom(f)) \\ &\quad = (f \circ \mathbb{C}(A, Dom(f))) \circ \mathbb{C}(Dom(g), A) \end{aligned}$$

La réciproque s'obtient de manière identique.

□

Le théorème suivant est l'équivalent sur les morphismes du lemme 11.2. Autrement dit, il n'y a qu'une façon d'oublier sur les morphismes, qui est determiné par l'oubli sur les domaines et les codomaines des morphismes.

Théorème 11.2 *Soient f, g et h des morphismes de $\mathcal{C}$, tels que $f \blacktriangleleft h$ et $g \blacktriangleleft h$, $Im(f) \blacktriangleleft Im(g)$ et $Dom(f) \blacktriangleleft Dom(g)$.*

On a

$$f \blacktriangleleft g$$

Preuve

$$\begin{aligned} & f \circ \mathbf{C}(Dom(g), Dom(f)) \circ \mathbf{C}(Dom(h), Dom(g)) \\ & \quad = f \circ \mathbf{C}(Dom(h), Dom(f)) \\ & \quad = \mathbf{C}(Im(h), Im(f)) \circ h \\ & \quad = \mathbf{C}(Im(g), Im(f)) \circ \mathbf{C}(Im(h), Im(g)) \circ h \\ & \quad = \mathbf{C}(Im(g), Im(f)) \circ g \circ \mathbf{C}(Dom(h), Dom(g)) \end{aligned}$$

Donc par épimorphisme de $\mathbf{C}(Dom(h), Dom(g))$, on en déduit que

$$f \circ \mathbf{C}(Dom(g), Dom(f)) = \mathbf{C}(Im(g), Im(f)) \circ g$$

□

Lemme 11.8 *Soient f et g deux morphismes de $\mathcal{C}$ tels que $f \blacktriangleleft g$. Soient A et B tels que $A \blacktriangleleft B$, $\mathcal{F}(A) = Im(f)$ et $\mathcal{F}(B) = Im(g)$. Soit a de $\mathfrak{A}$ tel que $a \notin Dom(f) \cup Dom(g)$*

$$f *^a A \blacktriangleleft g *^a B \qquad \text{et} \qquad q^a(f, A) \blacktriangleleft q^a(g, B)$$

Preuve

1) Si $B \neq \mathcal{A}(B)$ et $A = \mathcal{A}(A)$, on a :

$$\begin{aligned} & \mathcal{A}(g *_a B) \\ & \quad = g *^a \mathcal{A}(B) \\ & \quad = g *^a (\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) * A) \\ & \quad = (\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) \circ g) *^a A \\ & \quad = (f \circ \mathbf{C}(Dom(g), Dom(f))) *^a A \\ & \quad = \mathbf{C}(Dom(g), Dom(f)) * (f *^a A) \end{aligned}$$

On en déduit que $f *^a A \blacktriangleleft g *^a B$, et
$\mathbf{C}(g *^a B, f *^a A) = q(\mathbf{C}(Dom(g), Dom(f)), f *^a A) \circ \alpha_{g *^a B}$.
De même,

$$\begin{aligned} & \mathbf{C}(B, A) \circ q^a(g, B) \\ & \quad = q(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), A) \circ \alpha_B \circ q^a(g, B) \\ & \quad = q(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), A) \circ q^a(g, \mathcal{A}(B)) \circ \alpha_{g *^a B} \\ & \quad = q(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)), A) \circ q^a(g, \mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) * A) \circ \alpha_{g *^a B} \\ & \quad = q^a(\mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) \circ g, A) \circ \alpha_{g *^a B} \\ & \quad = q^a(f \circ \mathbf{C}(Dom(g), Dom(f)), A) \circ \alpha_{g *^a B} \\ & \quad = q^a(f, A) \circ q(\mathbf{C}(Dom(g), Dom(f)), f *^a A) \circ \alpha_{g *^a B} \\ & \quad = q^a(f, A) \circ \mathbf{C}(g *^a B, f *^a A) \end{aligned}$$

2) Sinon, on fait le même raisonnement pour $B = \mathbf{C}(\mathcal{F}(B), \mathcal{F}(A)) * A$.

□

11.1.3 Union

On se place dans cette sous-section dans une CGR. On définit la notion d'union de signature.

Lemme-Définition 11.1 *On suppose $A \blacktriangleleft C$ et $B \blacktriangleleft C$. On définit $A \cup^C B$ tel que $(A \cup^C B) \blacktriangleleft C$ et $Im(\mathfrak{L}_{A \cup^C B}) = Im(\mathfrak{L}_A) \cup Im(\mathfrak{L}_B)$ et $\mathfrak{M}_{A \cup^C B} = \mathfrak{M}_A \cup \mathfrak{M}_B$.*

Construction Par induction sur $|C|$.

1) Si $C = \mathbf{1}$ alors $A = B = \mathbf{1}$, et dans ce cas, on pose $\boxed{A \cup^C B = \mathbf{1}}$.

2) Si $\mathfrak{L}_C(|C|) = \mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$.

Par hypothèse de récurrence, $(\mathcal{F}(A) \cup^{\mathcal{F}(C)} \mathcal{F}(B)) \blacktriangleleft \mathcal{F}(C)$, avec $Im(\mathfrak{L}_{\mathcal{F}(A) \cup^{\mathcal{F}(C)} \mathcal{F}(B)}) = Im(\mathfrak{L}_{\mathcal{F}(A)}) \cup Im(\mathfrak{L}_{\mathcal{F}(B)})$ et $\mathfrak{M}_{\mathcal{F}(A) \cup^{\mathcal{F}(C)} \mathcal{F}(B)} = \mathfrak{M}_{\mathcal{F}(A)} \cup \mathfrak{M}_{\mathcal{F}(B)}$.

Par régularité, on en déduit $\mathcal{F}(A) \blacktriangleleft (\mathcal{F}(A) \cup^{\mathcal{F}(C)} \mathcal{F}(B))$ et $\mathcal{F}(B) \blacktriangleleft (\mathcal{F}(A) \cup^{\mathcal{F}(C)} \mathcal{F}(B))$.

2.1) Si $C \neq \mathcal{A}(C)$.

2.1.1) Si $A = \mathcal{A}(A)$ et $B = \mathcal{A}(B)$.

On pose
$$\boxed{A \cup^C B = \mathbf{C}(\mathcal{F}(A) \cup^{\mathcal{F}(C)} \mathcal{F}(B), \mathcal{F}(A)) * A}$$

On a $\mathcal{F}(A \cup^C B) = \mathcal{F}(A) \cup^{\mathcal{F}(C)} \mathcal{F}(B)$.

Par transitivité, $\mathbf{C}(\mathcal{F}(C), \mathcal{F}(A)) = \mathbf{C}(\mathcal{F}(A \cup^C B), \mathcal{F}(A)) \circ \mathbf{C}(\mathcal{F}(C), \mathcal{F}(A \cup^C B))$.
On en déduit que $\mathcal{A}(C) = \mathbf{C}(\mathcal{F}(C), \mathcal{F}(A \cup^C B)) * (A \cup^C B)$.

$A \cup^C B$ vérifie bien les conditions souhaitées.

2.1.2) Si $A = \mathcal{A}(A)$ et $B \neq \mathcal{A}(B)$.

On pose
$$\boxed{A \cup^C B = \mathbf{C}(\mathcal{F}(A) \cup^{\mathcal{F}(C)} \mathcal{F}(B), \mathcal{F}(B)) * B}$$

On a $\mathcal{F}(A \cup^C B) = \mathcal{F}(A) \cup^{\mathcal{F}(C)} \mathcal{F}(B)$.

On a : $\mathcal{C} = \mathbf{C}(\mathcal{F}(C), \mathcal{F}(A \cup^C B)) * (A \cup^C B)$.

$A \cup^C B$ vérifie bien les conditions souhaitées.

2.1.3) Si $A \neq \mathcal{A}(A)$ et $B = \mathcal{A}(B)$. Cas symétrique du précédent, on pose
$$\boxed{A \cup^C B = \mathbf{C}(\mathcal{F}(A) \cup^{\mathcal{F}(C)} \mathcal{F}(B), \mathcal{F}(A)) * A}$$

2.2) Sinon, on pose $\boxed{A \cup^C B = \mathbf{C}(\mathcal{F}(A) \cup \mathcal{F}(B), \mathcal{F}(A)) * A}$. Trivialement, $A \cup^C B$ vérifie les conditions souhaitées.

3) Si $\mathfrak{L}_C(|C|) = \mathfrak{L}_A(|A|)$ et $\mathfrak{L}_C(|C|) \notin Im(\mathfrak{L}_B)$.

Par hypothèse de récurrence, $(\mathcal{F}(A) \cup^{\mathcal{F}(C)} \mathcal{F}(B)) \blacktriangleleft \mathcal{F}(C)$, avec $Im(\mathfrak{L}_{\mathcal{F}(A) \cup^{\mathcal{F}(C)} B}) = Im(\mathfrak{L}_{\mathcal{F}(A)}) \cup Im(\mathfrak{L}_B)$ et $\mathfrak{M}_{\mathcal{F}(A) \cup^{\mathcal{F}(C)} B} = \mathfrak{M}_{\mathcal{F}(A)} \cup \mathfrak{M}_B$.

On en déduit $\mathcal{F}(A) \blacktriangleleft (\mathcal{F}(A) \cup^{\mathcal{F}(C)} B)$ et $B \blacktriangleleft (\mathcal{F}(A) \cup^{\mathcal{F}(C)} B)$.

3.1) Si $C \neq \mathcal{A}(C)$, $A = \mathcal{A}(A)$

On pose
$$\boxed{A \cup^C B = \mathbf{C}(\mathcal{F}(A) \cup^{\mathcal{F}(C)} B, \mathcal{F}(A)) * A}$$

On a $\mathcal{F}(A \cup^C B) = \mathcal{F}(A) \cup^{\mathcal{F}(C)} B$, et $\mathcal{A}(C) = \mathbf{C}(\mathcal{F}(C), \mathcal{F}(A \cup^C B)) * (A \cup^C B)$.

$A \cup^C B$ vérifie bien les conditions souhaitées.

3.2) Sinon, on pose $\boxed{A \cup^C B = \mathbf{C}(\mathcal{F}(A) \cup^{\mathcal{F}(C)} B, \mathcal{F}(A)) * A}$. On a bien $C = \mathbf{C}(\mathcal{F}(C), \mathcal{F}(A \cup^C B)) * (A \cup^C B)$.

4) Sinon, si $\mathfrak{L}_C(|C|) = \mathfrak{L}_B(|B|)$ et $\mathfrak{L}_C(|C|) \notin Im(\mathfrak{L}_A)$. Cas symétrique du précédent. On pose

$$\boxed{A \cup^C B = \mathbf{C}(A \cup^{\mathcal{F}(C)} \mathcal{F}(B), \mathcal{F}(B)) * B}$$

5) Si $\mathfrak{L}_C(|C|) \notin Im(\mathfrak{L}_A) \cap Im(\mathfrak{L}_B)$. Par hypothèse de récurrence, $(A \cup^{\mathcal{F}(C)} B) \blacktriangleleft \mathcal{F}(C)$, avec $Im(\mathfrak{L}_{A \cup^{\mathcal{F}(C)} B}) = Im(\mathfrak{L}_A) \cup Im(\mathfrak{L}_B)$ et $\mathfrak{M}_{A \cup^{\mathcal{F}(C)} B} = \mathfrak{M}_A \cup \mathfrak{M}_B$.
On pose

$$\boxed{A \cup^C B = A \cup^{\mathcal{F}(C)} B}$$

On a bien $A \cup^C B \blacktriangleleft C$.

$\square$

Lemme 11.9 *On a*

$$A \blacktriangleleft (A \cup^C B) \qquad \textit{et} \qquad B \blacktriangleleft (A \cup^C B)$$

Preuve Immédiat.

$\square$

Remarque 11.2 *Pour tous A, B, C tels que $A \blacktriangleleft C$ et $B \blacktriangleleft C$. Pour tout D tel que $D \blacktriangleleft C$, $A \blacktriangleleft D$ et $B \blacktriangleleft D$, on a $A \cup^D B \blacktriangleleft D \blacktriangleleft C$, donc par régularité, $A \cup^D B = A \cup^C B$.*

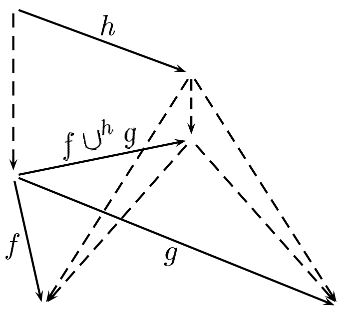
On définit maintenant aussi l'union sur les morphismes. On commence à définir une version "simple" de cette union, dans laquelle les domaines des deux morphismes dont on fait l'union sont égaux. On étendra ensuite cette union lorsque les domaines sont quelconques.

Lemme-Définition 11.2
Soient f, g et h des morphismes de $\mathcal{C}$ tels que $f \blacktriangleleft h$, $g \blacktriangleleft h$ avec $Dom(f) = Dom(g)$.

On note $A = Dom(f)$.

On construit $f \cup^h g$ de $\mathcal{C}[A, Im(f) \cup^{Im(h)} Im(g)]$ tel que

$$f \cup^h g \blacktriangleleft h$$



Construction Par induction sur $|Im(h)|$ (la construction de $Im(f) \cup^{Im(h)} Im(g)$).

1) Si $Im(h) = \mathbf{1}$ alors $Im(f) = Im(g) = \mathbf{1}$, et $f = g = h = id_{\mathbf{1}}$. On pose $\boxed{f \cup^h g = id_{\mathbf{1}}}$.

2) Si $\mathfrak{L}_{Im(h)}(|Im(h)|) = \mathfrak{L}_{Im(f)}(|Im(f)|) = \mathfrak{L}_{Im(g)}(|Im(g)|)$,
On a : $Im(f) \cup^{Im(h)} Im(g) = \mathbf{C}(\mathcal{F}(Im(f)) \cup^{\mathcal{F}(Im(h))} \mathcal{F}(Im(g)), \mathcal{F}(Im(f))) * Im(f)$.
On a $p_{Im(f)} \blacktriangleleft p_{Im(h)}$, donc par compatibilité de $\circ$ sur $\blacktriangleleft$, on a :

$$(p_{Im(f)} \circ f) \blacktriangleleft (p_{Im(h)} \circ h)$$

Et de même, $(p_{Im(g)} \circ g) \blacktriangleleft (p_{Im(h)} \circ h)$.
Par récurrence, on calcule $k = (p_{Im(f)} \circ f) \cup^{p_{Im(h)} \circ h} (p_{Im(g)} \circ g)$. On a donc $k \blacktriangleleft (p_{Im(h)} \circ h)$, avec $Im(k) = \mathcal{F}(Im(f)) \cup^{\mathcal{F}(Im(h))} \mathcal{F}(Im(g)) = \mathcal{F}(Im(f) \cup^{Im(h)} Im(g))$.
On en déduit $(p_{Im(f)} \circ f) \blacktriangleleft k$, c'est-à-dire, comme $Dom(k) = Dom(f) = A$,

$$p_{Im(f)} \circ f = \mathbf{C}(\mathcal{F}(Im(f) \cup^{Im(h)} Im(g)), \mathcal{F}(Im(f))) \circ k$$

2.1) Si $Im(h) \neq \mathcal{A}(Im(h))$.

2.1.1) Si $Im(f) = \mathcal{A}(Im(f))$ et $Im(g) = \mathcal{A}(Im(g))$.
On pose donc

$$\boxed{f \cup^h g = k \curlywedge_{Im(f) \cup^{Im(h)} Im(g)} f}$$

On note $\phi = (f \cup^h g) \circ \mathbf{C}(Dom(h), A)$. ϕ vérifie :
- $p_{Im(f) \cup^{Im(h)} Im(g)} \circ \phi = k \circ \mathbf{C}(Dom(h), A)$
- $\mathbf{C}(Im(f) \cup^{Im(h)} Im(g), Im(f)) \circ \phi = f \circ \mathbf{C}(Dom(h), A)$.

Donc $\phi = k \circ \mathbf{C}(Dom(h), A) \curlywedge_{Im(f) \cup^{Im(h)} Im(g)} f \circ \mathbf{C}(Dom(h), A)$.
Or $\mathbf{C}(Im(h), Im(f) \cup^{Im(h)} Im(g)) \circ h$ vérifie ces mêmes égalités. Donc

$$\phi = \mathbf{C}(Im(h), Im(f) \cup^{Im(h)} Im(g)) \circ h$$

Ce qui correspond à $(f \cup^h g) \blacktriangleleft h'$.

2.1.2) Si $Im(f) = \mathcal{A}(Im(f))$ et $Im(g) \neq \mathcal{A}(Im(g))$.
On pose

$$\boxed{f \cup^h g = k \curlywedge_{Im(f) \cup^{Im(h)} Im(g)} g}$$

Avec le même raisonnement que précédemment, on montre l'égalité souhaitée.

2.1.3) Si $Im(f) \neq \mathcal{A}(Im(f))$ et $Im(g) = \mathcal{A}(Im(g))$. Cas symétrique du précédent, on pose

$$\boxed{f \cup^h g = ((p_{Im(f)} \circ f) \cup^{p_{Im(h)} \circ h} (p_{Im(g)} \circ g)) \curlywedge_{Im(f) \cup^{Im(h)} Im(g)} f}$$

2.2) Sinon, on pose

$$\boxed{f \cup^h g = k \curlywedge_{Im(f) \cup^{Im(h)} Im(g)} f}$$

3) Si $\mathfrak{L}_{Im(h)}(|Im(h)|) = \mathfrak{L}_{Im(f)}(|Im(f)|)$ et $\mathfrak{L}_{Im(h)}(|Im(h)|) \notin Im(\mathfrak{L}_{Im(g)})$.
$Im(f) \cup^{Im(h)} Im(g) = \mathbf{C}(\mathcal{F}(Im(f)) \cup^{\mathcal{F}(Im(h))} Im(g), \mathcal{F}(Im(f))) * Im(f)$.
Comme, $p_{Im(f)} \blacktriangleleft p_{Im(h)}$ et $id_{Im(g)} \blacktriangleleft p_{Im(g)}$, on a :

$(p_{Im(f)} \circ f) \blacktriangleleft (p_{Im(h)} \circ h)$ et $g \blacktriangleleft (p_{Im(h)} \circ h)$

On pose

$$\boxed{f \cup^h g = ((p_{Im(f)} \circ f) \cup^{p_{Im(h)} \circ h} g) \curlywedge_{Im(f) \cup^{Im(h)} Im(g)} f}$$

4) Si $\mathfrak{L}_{Im(h)}(|Im(h)|) = \mathfrak{L}_{Im(g)}(|Im(g)|)$ et $\mathfrak{L}_{Im(h)}(|Im(h)|) \notin Im(\mathfrak{L}_{Im(f)})$. Cas symétrique du précédent, on pose

$$\boxed{f \cup^h g = (f \cup^{p_{Im(h)} \circ h} (p_{Im(g)} \circ g)) \curlywedge_{Im(f) \cup^{Im(h)} Im(g)} g}$$

5) Si $\mathfrak{L}_{Im(g)}(|Im(g)|) \notin Im(\mathfrak{L}_A) \cap Im(\mathfrak{L}_{Im(f)})$. $A \cup^{Im(g)} Im(f) = A \cup^{\mathcal{F}(Im(g))} Im(f)$. On a :

$$f \blacktriangleleft (p_{Im(h)} \circ h) \qquad \text{et} \qquad g \blacktriangleleft (p_{Im(h)} \circ h)$$

On pose

$$\boxed{f \cup^h g = f \cup^{p_{Im(h)} \circ h} g}$$

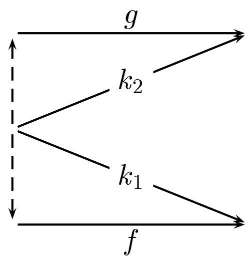
□

Définition 11.4 *Soient f, g et h des morphismes de $\mathcal{C}$ tels que $f \blacktriangleleft h$ et $g \blacktriangleleft h$.*

On pose

- $k_1 = (f \circ \mathbf{C}(Dom(f) \cup^{Dom(h)} Dom(g), Dom(f)))$
- $k_2 = (g \circ \mathbf{C}(Dom(f) \cup^{Dom(h)} Dom(g), Dom(g)))$

On a : $k_1 \blacktriangleleft h$ et $k_2 \blacktriangleleft h$ avec $Dom(k_1) = Dom(k_2) = Dom(f) \cup^{Dom(h)} Dom(g)$.



On étend la construction "$\cup^h$" en posant :

$$f \cup^h g = k_1 \cup^h k_2$$

On a : $f \cup^h g \blacktriangleleft h$.

Lemme 11.10 *On a*

$$f \blacktriangleleft (f \cup^h g) \qquad et \qquad g \blacktriangleleft (f \cup^h g)$$

Preuve Immédiat par le théorème 11.2

□

Remarque 11.3 *Soient f, g et h des morphismes de $\mathcal{C}$ tels que $f \blacktriangleleft h$ et $g \blacktriangleleft h$.*

Soit h' un morphisme tel que $h' \blacktriangleleft h$, $f \blacktriangleleft h'$ et $g \blacktriangleleft h'$, alors $(f \cup^{h'} g) \blacktriangleleft h' \blacktriangleleft h$. Par régularité $f \cup^{h'} g = f \cup^h g$.

Lemme 11.11 *Soient A_1, A_2, A_3 et A des objets de $\mathcal{C}$ tels que pour $i \in [1,3]$, $A_i \blacktriangleleft A$.*

$$(A_1 \cup^A A_2) \cup^A A_3 = A_1 \cup^A (A_2 \cup^A A_3)$$

On notera donc cette objet $\bigcup_i^A A_i$.

Preuve Immédiat par régularité.

□

Lemme 11.12 *Soient f_1, f_2, f_3 et f des morphismes, tels que pour $i \in [1,3]$, $f_i \blacktriangleleft f$.*

$$(f_1 \cup^f f_2) \cup^f f_3 = f_1 \cup^f (f_2 \cup^f f_3)$$

On notera donc ce morphisme $\bigcup_i^f f_i$.

Preuve Immédiat par le théorème 11.2

□

Lemme 11.13 *Avec la même technique, on prouve la commutativité*

$$A \cup^C B = B \cup^C A \qquad et \qquad f \cup^h g = g \cup^h f$$

Lemme 11.14 *Soient f et g des morphismes de $\mathcal{C}$ tels que $Im(f) = Dom(g)$. Pour $i \in [1,n]$, soient f_i et g_i des morphismes tels que $Im(f_i) = Dom(g_i)$ et $f_i \blacktriangleleft f$ et $g_i \blacktriangleleft g$.*

$$\bigcup_{i\in[1,n]}^{g\circ f} (g_i \circ f_i) = (\bigcup_{i\in[1,n]}^{g} g_i) \circ (\bigcup_{i\in[1,n]}^{f} f_i)$$

Preuve Ces deux morphismes ont même domaine et même image : $\bigcup_{i\in[1,n]} Dom(f_i)$, et resp. $\bigcup_{i\in[1,n]} Im(g_i)$.
Par compatibilité de $\circ$ sur $\blacktriangleleft$, on a

$$(\bigcup_{i\in[1,n]}^{g} g_i) \circ (\bigcup_{i\in[1,n]}^{f} f_i) \blacktriangleleft (g \circ f)$$

Par le théorème 11.2, et l'antisymétrie de $\blacktriangleleft$, on obtient l'égalité.

□

11.2 Définition des CGED

La notion de CGED est le pilier de l'axiomatisation de FocKer. Par rapport aux CG, on suppose ici une notion d'intersection entre générateurs et de sur-signatures. Ces axiomes, et en particulier la notion d'intersection, pourraient sembler plus forts que le simple effacement de définition. Néanmoins le théorème 11.3 ci-dessous et les remarques introductives en début de section indiquent que cette axiomatisation semble appropriée. Elle a de plus le mérite de la concision et de la clarté.

Définition 11.5 *Une* catégorie de générateurs d'enregistrements avec effacement de définition *(ou CGED en abrégé) $\mathfrak{C}$ est un uplet $< \mathcal{C}, \mathfrak{A}, \mathcal{F}, \mathfrak{L}, \mathbf{1}, p, *, q, \mathcal{A}, \alpha, \mathcal{M}, \mathcal{H}, \mathbb{G}, \mathcal{T}, \cap >$ tel que :*

1) *$< \mathcal{C}, \mathfrak{A}, \mathcal{F}, \mathfrak{L}, \mathbf{1}, p, *, q, \mathcal{A}, \alpha, \mathcal{M}, \mathcal{H}, \mathbb{G} >$ est une catégorie de générateurs d'enregistrements.*

2) *Pour tout A, $\mathcal{T}(A) \propto_{\mathfrak{M}_A} A$.*

3) *Soient A, B et C tels que $A \blacktriangleleft C$ et $B \blacktriangleleft C$ alors $A \cap^C B$ est un objet de $\mathcal{C}$, et $(A \cap^C B) \blacktriangleleft A$ et $(A \cap^C B) \blacktriangleleft B$, avec $Im(\mathfrak{L}_{A\cap^C B}) = Im(\mathfrak{L}_A) \cap Im(\mathfrak{L}_B)$ et $\mathfrak{M}_{A\cap^C B} = \mathfrak{M}_A \cap \mathfrak{M}_B$.*

Lemme 11.15 *$\mathfrak{C}$ est régulière sur les signatures.*

Preuve

1) On montre tout d'abord que pour tout A, B, C de $Ob(\mathcal{C})$ et pour tout n tels $\mathcal{F}(A) = \mathcal{F}^n(B) = \mathcal{F}^n(C)$ et $\mathfrak{L}_A(|A|) \notin Im(\mathfrak{L}_B) \cup Im(\mathfrak{L}_C)$, alors $p_A *^n B = p_A *^n C$ implique $B = C$.

On pose $X = p_A *^n B$. On a : $B \blacktriangleleft X$ et $C \blacktriangleleft X$. On pose $Y = B \cap^X C$. Comme $Im(\mathfrak{L}_B) = Im(\mathfrak{L}_C)$ et $\mathfrak{M}_B = \mathfrak{M}_C$, donc $Y = B = A$, par 9.19.

2) De même, on montre que pour tout A, B, C de $Ob(\mathcal{C})$ et pour tout n tels que $\mathcal{A}(A) = \mathcal{F}^n(B) = \mathcal{F}^n(C)$, alors $\alpha_A *^n B = \alpha_A *^n C$ implique $B = C$.

□

Remarque 11.4 *On ne peut pas montrer la régularité sur les morphismes : il faudrait supposer une intersection pour les morphismes. Mais on n'a pas besoin de telles hypothèses pour écrire les opérations de FocKer.*

Remarque 11.5 *Par régularité, $\mathcal{T}$ est une notion de sur-signature pleinement abstraite.*

Lemme 11.16 *Soient A, B, C et D tels que $A \blacktriangleleft C$, $A \blacktriangleleft D$, $B \blacktriangleleft C$ et $B \blacktriangleleft D$. On a :*

$$A \cap^C B = A \cap^D B$$

Preuve Il suffit de remarquer que $A \cap^C B \blacktriangleleft A$ et $A \cap^D B \blacktriangleleft B$ avec $\mathfrak{L}_{A\cap^C B} = \mathfrak{L}_{A\cap^D B}$ et $\mathfrak{M}_{A\cap^C B} = \mathfrak{M}_{A\cap^D B}$. On conclut alors en utilisant le théorème 11.1

□

Lemme 11.17 *Soient A_1, A_2, A_3 et A des objets de $\mathcal{C}$ tels que pour $i \in [1,3]$, $A_i \blacktriangleleft A$.*

$$(A_1 \cap^A A_2) \cap^A A_3 = A_1 \cap^A (A_2 \cap^A A_3)$$

On notera donc cette objet $\bigcap_i^A A_i$.

Preuve Par régularité sur les signatures.

□

Remarque 11.6 *Soit A de $\mathcal{C}$. On déduit de la régularité sur les signatures que l'ensemble $\{X \ / \ X \blacktriangleleft A\}$ est fini. Si n est la profondeur de $\mathfrak{M}_A$, son cardinal vaut au plus $2^{|A|} * 2^n$.*

Si A est un objet, on définit maintenant l'objet $\mathbb{D}(A)$ introduit en introduction de ce chapitre : cet objet est la plus petite sur-signature de A contenant la dernière étiquette de A. Il représente l'ensemble des champs dont dépend le dernier champs de A

Définition 11.6 *Soit A un objet de $\mathcal{C}$. On pose $\mathbb{D}(A) = \bigcap^A\{X \ / \ X \blacktriangleleft A \wedge \mathfrak{L}_A(|A|) \in Im(\mathfrak{L}_X) \wedge \mathfrak{M}_A \cap \{\mathfrak{L}_A(|A|)\} \subset \mathfrak{M}_X\}$. (Ce dernier ensemble est non vide, car A vérifie la condition).*

Lemme 11.18 *Soit A un objet de $\mathcal{C}$. Alors, $\mathbb{D}(A) \blacktriangleleft A$. De plus :*

1) *Si $\mathcal{A}(A) = A$ alors $\mathfrak{M}_{\mathbb{D}(A)} = \emptyset$, et pour tout B tel que $B \blacktriangleleft A$ et $\mathfrak{L}_A(|A|) \in Im(\mathfrak{L}_B)$ alors $\mathbb{D}(A) \blacktriangleleft B$.*
2) *Sinon pour tout B tel que $B \blacktriangleleft A$ et $\mathfrak{L}_A(|A|) \in \mathfrak{M}_B$ alors $\mathbb{D}(A) \blacktriangleleft B$.*

Preuve Il suffit d'appliquer la définition. Pour le cas $\mathcal{A}(A) = A$, on déduit $\mathfrak{M}_{\mathbb{D}(A)} = \emptyset$ du fait que $\mathcal{T}(A) \blacktriangleleft A$ avec $\mathfrak{L}_A(|A|) \in Im(\mathfrak{L}_{\mathcal{T}(A)})$ et donc que $\mathbb{D}(A) \blacktriangleleft \mathcal{T}(A)$.

□

Théorème 11.3 *Une catégorie de générateurs d'enregistrements $\mathfrak{C}$ définit une catégorie de catégorie de générateurs d'enregistrements avec effacement de définition ssi $\mathfrak{C}$ est régulière sur les signatures, et il existe $\mathbb{D}$ une fonction de $Ob(\mathcal{C})$ dans $Ob(\mathcal{C})$ telle que :*

1) *Pour tout A, $\mathbb{D}(A) \blacktriangleleft A$ avec $\mathfrak{L}_A(|A|) \in Im(\mathfrak{L}_{\mathbb{D}(A)})$ et $\mathfrak{M}_A \cap \{\mathfrak{L}_A(|A|)\} \subset \mathfrak{M}_{\mathbb{D}(A)}$.*
2) *Pour tous A et B tel que $B \blacktriangleleft A$ et $\mathfrak{L}_A(|A|) \in Im(\mathfrak{L}_B)$ et $\mathfrak{M}_A \cap \{\mathfrak{L}_A(|A|)\} \subset \mathfrak{M}_B$, alors $\mathbb{D}(A) \blacktriangleleft B$.*
3) *Pour tout A, si $\mathcal{A}(A) = A$ alors $\mathfrak{M}_{\mathbb{D}(A)} = \emptyset$.*

Preuve Les lemmes précédents permettent d'établir la régularité de signatures et l'existence de $\mathbb{D}$ sous l'hypothèse que $\mathfrak{C}$ est une catégorie de générateurs d'enregistrements. Reste à montrer la réciproque.
On commence par construire $\mathcal{T}$. Soit A de $Ob(\mathcal{C})$, on pose $\boxed{\mathcal{T}(A) = \bigcup_n^A \mathbb{D}(\mathcal{A}(\mathcal{F}^n(A)))}$. Donc $\mathcal{T}(A) \blacktriangleleft A$. De plus $Im(\mathfrak{L}_{\mathcal{T}(A)}) = Im(A)$ et $\mathfrak{M}_{\mathcal{T}(A)} = \emptyset$. Donc $\mathcal{T}(A) \propto_{\mathfrak{M}(A)} A$.
Soient A, B et C tels que $A \blacktriangleleft C$ et $B \blacktriangleleft C$, on construit maintenant $A \cap^C B$. On procède par induction sur $|C|$.

1) Si $C = \mathbf{1}$ alors $A = B = \mathbf{1}$, et dans ce cas, on pose $\boxed{A \cap^C B = \mathbf{1}}$.

2) Si $\mathfrak{L}_C(|C|) = \mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$, avec $C = \mathcal{A}(C)$ ou $A = \mathcal{A}(A)$ ou $B = \mathcal{A}(B)$.

Par hypothèse sur $\mathfrak{C}$, $\mathbb{D}(\mathcal{A}(C)) \blacktriangleleft A$ et $\mathbb{D}(\mathcal{A}(C)) \blacktriangleleft B$. Comme $\mathfrak{L}_{\mathbb{D}(\mathcal{A}(C))}(|\mathbb{D}(\mathcal{A}(C))|) = \mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$, on a aussi $\mathcal{F}(\mathbb{D}(\mathcal{A}(C))) \blacktriangleleft \mathcal{F}(A)$ et $\mathcal{F}(\mathbb{D}(\mathcal{A}(C))) \blacktriangleleft \mathcal{F}(B)$.

Par hypothèse de récurrence, $(\mathcal{F}(A) \cap^{\mathcal{F}(C)} \mathcal{F}(B)) \blacktriangleleft \mathcal{F}(A)$ et $(\mathcal{F}(A) \cap^{\mathcal{F}(C)} \mathcal{F}(B)) \blacktriangleleft \mathcal{F}(B)$ avec $Im(\mathfrak{L}_{\mathcal{F}(A)\cap^{\mathcal{F}(C)}\mathcal{F}(B)}) = Im(\mathfrak{L}_{\mathcal{F}(A)}) \cap Im(\mathfrak{L}_{\mathcal{F}(B)})$ et $\mathfrak{M}_{\mathcal{F}(A)\cap^{\mathcal{F}(C)}\mathcal{F}(B)} = \mathfrak{M}_{\mathcal{F}(A)} \cap \mathfrak{M}_{\mathcal{F}(B)}$.

On a donc $Im(\mathfrak{L}_{\mathcal{F}(\mathbb{D}(\mathcal{A}(C)))}) \subseteq Im(\mathfrak{L}_{\mathcal{F}(A)\cap^{\mathcal{F}(C)}\mathcal{F}(B)})$ et $\mathfrak{M}_{\mathcal{F}(\mathbb{D}(\mathcal{A}(C)))} \subseteq \mathfrak{M}_{\mathfrak{L}_{\mathcal{F}(A)\cap^{\mathcal{F}(C)}\mathcal{F}(B)}}$.

Par régularité sur les signatures, on en déduit que $\mathcal{F}(\mathbb{D}(\mathcal{A}(C))) \blacktriangleleft \mathcal{F}(A) \cap^{\mathcal{F}(C)} \mathcal{F}(B)$.

On pose

$$\boxed{A \cap^C B = \mathbf{C}(\mathcal{F}(A) \cap^{\mathcal{F}(C)} \mathcal{F}(B), \mathcal{F}(\mathbb{D}(\mathcal{A}(C)))) * \mathbb{D}(\mathcal{A}(C))}$$

On montre que $A \cap^C B \blacktriangleleft C$. On en déduit que $A \cap^C B \blacktriangleleft A$ et $A \cap^C B \blacktriangleleft B$ par régularité sur les signatures. On vérifie aisément que $Im(\mathfrak{L}_{A\cap^C B}) = Im(\mathfrak{L}_A) \cap Im(\mathfrak{L}_B)$ et $\mathfrak{M}_{A\cap^C B} = \mathfrak{M}_A \cap \mathfrak{M}_B$.

3) Sinon, si $\mathfrak{L}_C(|C|) = \mathfrak{L}_A(|A|) = \mathfrak{L}_B(|B|)$. $C \neq \mathcal{A}(C)$, $A \neq \mathcal{A}(A)$ et $B \neq \mathcal{A}(B)$. On fait le même raisonnement avec

$$\boxed{A \cap^C B = \mathbf{C}(\mathcal{F}(A) \cap^{\mathcal{F}(C)} \mathcal{F}(B), \mathcal{F}(\mathbb{D}(C))) * \mathbb{D}(C)}$$

4) Sinon, $\mathfrak{L}_C(|C|) \notin Im(\mathfrak{L}_A)$ ou $\mathfrak{L}_C(|C|) \notin Im(\mathfrak{L}_B)$. On pose

$$\boxed{A \cap^C B = \mathcal{F}(A) \cap^{\mathcal{F}(C)} \mathcal{F}(B)}$$

□

Remarque 11.7 *Ce théorème peut être d'une grande utilité pour exhiber des modèles des CGED (voir la discussion en section 11.6).*

11.3 Effacement de définitions

Définition 11.7 *Soit A un objet de $\mathcal{C}$, on pose $\mathfrak{T}_A = \mathfrak{M}_{\mathbb{D}(A)}$. L'ensemble d'étiquettes $\mathfrak{T}_A$ est l'ensemble des champs de A dont le dernier champ de A dépend de manière transparente.*

On montre que cette notion de dépendance transparente est close par transitivité : si le dernier champ de A dépend de manière transparente du k-ième dernier champ, alors le dernier champ dépend de manière transparente de tous les champs dont ce k-ième dernier champ dépend de manière transparente. C'est-à-dire que, $\mathfrak{T}_{\mathcal{F}^k(A)} \supset \mathfrak{T}_A$. On montre en fait un lemme plus fort :

Lemme 11.19 *Soient A et k tel que $\mathfrak{L}_A(|A| - k) \in \mathfrak{M}_{\mathbb{D}(A)}$. On a : $\mathbb{D}(\mathcal{F}^k(A)) \blacktriangleleft \mathbb{D}(A)$.*

Preuve On déduit des hypothèses que $\mathcal{A}(A) \neq A$ et $\mathcal{A}(\mathcal{F}^k(A)) \neq \mathcal{F}^k(A)$. On pose $n = |\mathbb{D}(A)| - \mathfrak{L}^{-1}_{\mathbb{D}(A)}(|A| - k)$. On a : $\mathcal{F}^n(\mathbb{D}(A)) \blacktriangleleft \mathcal{F}^k(A)$, avec $\mathfrak{L}_{\mathcal{F}^k(A)}(|\mathcal{F}^k(A)|) \in \mathfrak{M}_{\mathcal{F}^n(\mathbb{D}(A))}$. Donc $\mathbb{D}(\mathcal{F}^k(A)) \blacktriangleleft \mathcal{F}^n(\mathbb{D}(A)) \blacktriangleleft \mathbb{D}(A)$.

□

Définition 11.8 *Soient A et a de $Im(\mathfrak{L}_A)$ on pose $\mathfrak{D}_A(a) = \{b \in \mathfrak{M}_A \ / \ a = b \vee a \in \mathfrak{T}_{\mathcal{F}^{|A|-\mathfrak{L}_A^{-1}(b)}(A)}\}$. $\mathfrak{D}_A(a)$ est l'ensemble des champs de A dont la définition dépend de manière transparente de celle du champs d'étiquette a.*

Remarque 11.8 *Si $a \notin \mathfrak{M}_A$, $\mathfrak{D}_{\mathcal{F}(A)}(a) = \emptyset$.*

Lemme 11.20 *Si $\mathcal{A}(A) \neq A$ et $a \notin \mathfrak{T}_A$, $\mathfrak{D}_{\mathcal{F}(A)}(a) \cap \mathfrak{T}_A = \emptyset$.*

Preuve Supposons $a \notin \mathfrak{T}_A$.
Soit $b \in \mathfrak{M}_A$.
Si $b = a$, $b \notin \mathfrak{T}_A$.
Sinon, d'après le lemme 11.19, $b \in \mathfrak{T}_A$ implique $\mathfrak{T}_{\mathcal{F}^{|A|-\mathfrak{L}_A^{-1}(b)}(A)}$.
Donc $a \in \mathfrak{T}_{\mathcal{F}^{|A|-\mathfrak{L}_A^{-1}(b)}(A)}$ alors $a \in \mathfrak{T}_A$. Par contraposé, $b \notin \mathfrak{T}_A$.
Donc $b \in \mathfrak{D}_{\mathcal{F}(A)}(a)$ implique $b \notin \mathfrak{T}_A$.

□

Lemme 11.21 *Si $\mathcal{A}(A) \neq A$ et $a \in \mathfrak{T}_A$, $\mathfrak{D}_A(a) = \mathfrak{L}_A(|A|) \cup \mathfrak{D}_{\mathcal{F}(A)}$.*

Lemme-Définition 11.3 (effacement d'une définition) *Soit A un objet de $\mathcal{C}$. Soit $a \in Im(\mathfrak{L}_A)$. On construit un objet $A_{|a}$ tel que $A_{|a} \propto_{\mathfrak{D}_A(a)} A$.*

Construction Par induction sur A puis par cas sur a :

1) Si $a = \mathfrak{L}_A(|A|)$, on pose $\boxed{A_{|a} = \mathcal{A}(A)}$.

2) Si $\mathcal{A}(A) \neq A$ et $a \notin \mathfrak{T}_A$. On calcule $\mathcal{F}(A)_{|a}$.
Par hypothèse de récurrence, $\mathcal{F}(A)_{|a} \propto_{\mathfrak{D}_{\mathcal{F}(A)}(a)} \mathcal{F}(A)$.Or $\mathcal{F}(\mathbb{D}(A)) \blacktriangleleft \mathcal{F}(A)$. En particulier, $Im(\mathfrak{L}_{\mathcal{F}(\mathbb{D}(A))}) \subseteq Im(\mathcal{F}(A)_{|a})$.
De plus, comme $\mathfrak{D}_{\mathcal{F}(A)}(a) \cap \mathfrak{T}_A = \emptyset$, et $\mathfrak{M}_{\mathcal{F}(A)_{|a}} = \mathfrak{M}_{\mathcal{F}(A)} \backslash \mathfrak{D}_{\mathcal{F}(A)}(a)$, on en déduit : $\mathfrak{M}_{\mathcal{F}(\mathbb{D}(A))} \subseteq \mathfrak{M}_{\mathcal{F}(A)_{|a}}$. Par régularité sur les signatures, on a donc : $\mathcal{F}(\mathbb{D}(A)) \blacktriangleleft \mathcal{F}(A)_{|a}$.
On pose $\boxed{A_{|a} = \mathbf{C}(\mathcal{F}(A)_{|a}, \mathcal{F}(\mathbb{D}(A))) * \mathbb{D}(A)}$.
On a donc $\mathcal{F}(A_{|a}) = \mathcal{F}(A)_{|a}$.

$$\begin{aligned}
A & \\
&= \mathbf{C}(\mathcal{F}(A), \mathcal{F}(\mathbb{D}(A))) * \mathbb{D}(A) \\
&= (\mathbf{C}(\mathcal{F}(A)_{|a}, \mathcal{F}(\mathbb{D}(A))) \circ \mathbf{C}(\mathcal{F}(A), \mathcal{F}(A)_{|a})) * \mathbb{D}(A) \\
&= \mathbf{C}(\mathcal{F}(A), \mathcal{F}(A_{|a})) * (\mathbf{C}(\mathcal{F}(A_{|a}), \mathcal{F}(\mathbb{D}(A))) * \mathbb{D}(A)) \\
&= \mathbf{C}(\mathcal{F}(A), \mathcal{F}(A_{|a})) * A_{|a}
\end{aligned}$$

donc d'après le lemme 9.25, $A_{|a} \propto_{\mathfrak{D}_{\mathcal{F}(A)}(a)} A$. Et comme $a \notin \mathfrak{T}_A$, $\mathfrak{D}_A(a) = \mathfrak{D}_{\mathcal{F}(A)}(a)$. On en déduit $A_{|a} \propto_{\mathfrak{D}_A(a)} A$.

3) Sinon $\mathcal{A}(A) = A$ ou $a \in \mathfrak{T}_A$. On calcule $\mathcal{F}(A)_{|a}$.
Par hypothèse de récurrence, $\mathcal{F}(A)_{|a} \propto_{\mathfrak{D}_{\mathcal{F}(A)}(a)} \mathcal{F}(A)$.
On a $\mathcal{T}(\mathcal{F}(A)_{|a}) = \mathcal{T}(\mathcal{F}(A)) = \mathcal{F}(\mathcal{T}(A))$. On pose $\boxed{A_{|a} = \mathbf{C}(\mathcal{F}(A)_{|a}, \mathcal{F}(\mathcal{T}(A))) * \mathcal{T}(A)}$.

On a donc $\mathcal{F}(A_{|a}) = \mathcal{F}(A)_{|a}$ et $\mathcal{A}(A_{|a}) = A_{|a}$.

$$\begin{aligned}
&\mathcal{A}(A)\\
&\quad = \mathbf{C}(\mathcal{F}(A), \mathcal{F}(\mathcal{T}(A))) * \mathcal{T}(A)\\
&\quad = (\mathbf{C}(\mathcal{F}(A)_{|a}, \mathcal{F}(\mathcal{T}(A))) \circ \mathbf{C}(\mathcal{F}(A), \mathcal{F}(A_{|a}))) * \mathcal{T}(A)\\
&\quad = \mathbf{C}(\mathcal{F}(A), \mathcal{F}(A_{|a})) * A_{|a}
\end{aligned}$$

Si $\mathcal{A}(A) = A$, alors $\mathfrak{D}_A(a) = \mathfrak{D}_{\mathcal{F}(A)}(a)$ et $A_{|a} \propto_{\mathfrak{D}_A(a)} A$.
Sinon $a \in \mathfrak{T}_A$. D'où $\mathfrak{D}_A(a) = \{\mathfrak{L}_A(|A|)\} \cup \mathfrak{D}_{\mathcal{F}(A)}(a)$ et $A_{|a} \propto_{\mathfrak{D}_A(a)} A$.

□

Remarque 11.9 *On a prouvé au cours de la définition précédente que*

$$\mathcal{F}(A_{|a}) = \mathcal{F}(A)_{|a}$$

De plus si $a \notin \mathfrak{M}_A$ alors $A_{|a} = A$.

On montre maintenant que l'opération d'effacement construit la plus petite (au sens de la relation de sous-signature) sur-signature de A dans lequel le champ a n'est pas défini.

Théorème 11.4 *Soit A un objet de $\mathcal{C}$. Soit $a \in Im(\mathfrak{L}_A)$. Soit B tel que $B \blacktriangleleft A$ avec $a \notin \mathfrak{M}_B$, alors $B \blacktriangleleft A_{|a}$.*

Preuve Supposons $B \blacktriangleleft A$ avec $a \notin \mathfrak{M}_B$. On pose $l = \mathfrak{M}_B \cap \mathfrak{D}_A(a)$.

1) Si $l = \emptyset$, alors comme $A_{|a} \propto_{\mathfrak{D}_A(a)} A$, on a $\mathfrak{M}_B \subset \mathfrak{M}_{A_{|a}}$. De plus $Im(FL_{A_{|a}}) = Im(\mathfrak{L}_A)$. Par régularité sur les signatures on en conclut que $B \blacktriangleleft A_{|a}$.

2) Sinon, soit $k = \max\{|A| - \mathfrak{L}_A^{-1}(b) \ / \ b \in l\}$. On pose $C = \mathcal{F}^k(A)$. On a $C \neq \mathcal{A}(C)$ et $a \in \mathfrak{T}_C$. On pose $D = \mathcal{F}^{|B| - \mathfrak{L}_B(\mathfrak{L}_C(|C|))}(B)$. On a $D \blacktriangleleft C$ avec $\mathfrak{L}_C(|C|) \in \mathfrak{M}_D$, donc $\mathbb{D}(C) \blacktriangleleft D$
Comme $a \notin \mathfrak{M}_B$, $a \notin \mathfrak{M}_D$. Mais comme $a \in \mathfrak{T}_C$ et $\mathbb{D}(C) \blacktriangleleft D$, $a \in \mathfrak{M}_D$. C'est impossible.

□

Lemme 11.22 *Pour tous a et b de $Im(\mathfrak{L}_A)$,*

$$(A_{|a})_{|b} = (A_{|b})_{|a}$$

Preuve On a que $(A_{|a})_{|b} \propto_{\mathfrak{D}_A(a) \cup \mathfrak{D}_{A_{|a}}(b)} A$.
En appliquant la régularité sur les signatures, il suffit de montrer que
$\mathfrak{D}_A(a) \cup \mathfrak{D}_{A_{|a}}(b) = \mathfrak{D}_A(b) \cup \mathfrak{D}_{A_{|b}}(a)$.
On montre en fait facilement que

$$\mathfrak{D}_A(a) \cup \mathfrak{D}_{A_{|a}}(b) = \{c \in \mathfrak{M}_A \ / \ c \in \{a, b\} \vee \mathfrak{T}_{\mathcal{F}^{|A| - \mathfrak{L}_A^{-1}(c)}(A)} \cap \{a, b\} \neq \emptyset\}$$

□

Définition 11.9 (effacement de plusieurs définitions) *Soit A un objet de $\mathcal{C}$. Soit s une suite d'éléments de $Im(\mathfrak{L}_A)$. On généralise la définition précédente en posant*

$$A_{|\emptyset} = A \qquad et \qquad A_{|s;a} = (A_{|s})_{|a}$$

Lemme 11.23 *Pour tout s_1 et s_2 on a :*

$$A_{|s_1;s_2} = A_{|s_2;s_1}$$

Preuve Immédiat en utilisant le lemme 11.22.

□

Remarque 11.10 *Si $l \subset Im(\mathfrak{L}_A)$, on s'autorise donc désormais à écrire $A_{|l}$. En effet, vu le lemme précédent l'ordre dans lequel on prend les éléments de l n'a pas d'importance.*

11.4 Plongement d'un générateur vis-à-vis d'une permutation

On a vu en section 9.4.4 l'opération qui correspond au plongement d'une espèce dans une interface *"plus grande"*, mais sans permutation. On définit maintenant l'opération qui permute les champs d'un objet A à partir d'une permutation au niveau de $\mathcal{T}(A)$. Au cours de cette opération on peut être amené à effacer des définitions de A, lorsque la permutation brise des dépendances au niveau des définitions de A.

Lemme-Définition 11.4 (Plongement par transposition) *Soient A et T deux objets tels que $\mathcal{T}(A) \sim_n T$. On construit un objet $\Uparrow^T_\sim A$ tel que $\mathcal{T}(\Uparrow^T_\sim A) = T$ et*

1) *Si $\mathfrak{L}_A(|A| - n) \in \mathfrak{M}_A$ et $\mathfrak{L}_A(|A| - n) \notin Im(\mathbb{D}(\mathcal{F}^n(A)))$, alors $\Uparrow^T_\sim A \sim_n A$*

2) *Sinon, $\Uparrow^T_\sim A \sim_n A_{|\mathfrak{L}_A(|A|-n+1)}$.*

Construction Par cas sur A :

1) Si $\mathfrak{L}_A(|A| - n) \in \mathfrak{M}_A$ et $\mathfrak{L}_A(|A| - n) \notin Im(\mathbb{D}(\mathcal{F}^n(A)))$.
On a : $\mathcal{F}(\mathbb{D}(\mathcal{F}^n(A))) \blacktriangleleft \mathcal{F}^{n+2}(A)$. Et,
$\mathcal{F}^n(A) = (\mathbf{C}(\mathcal{F}^{n+2}(A), \mathcal{F}(\mathbb{D}(\mathcal{F}^n(A)))) \circ p_{\mathcal{F}^{n+1}(A)}) * \mathbb{D}(\mathcal{F}^n(A))$.
On pose $C = \mathbf{C}(\mathcal{F}^{n+2}(A), \mathcal{F}(\mathbb{D}(\mathcal{F}^n(A)))) * \mathbb{D}(\mathcal{F}^n(A))$. On a : $\mathcal{F}^n(A) = p_{\mathcal{F}^{n+1}(A)} * C$.
On pose

$$\boxed{\Uparrow^T_\sim A = (q(p_C, \mathcal{F}^{n+1}(A)) \curlywedge_{p_C * \mathcal{F}^{n+1}(A)} p_{\mathcal{F}^n(A)}) *^n A}$$

2) Sinon, on pose $B = A_{|\mathfrak{L}_A(|A|-n+1)}$. $\mathcal{T}(B) = \mathcal{T}(A)$.
On a $\mathcal{F}^n(B) = \mathbf{C}(\mathcal{F}^{n+1}(B), \mathcal{F}^{n+1}(\mathcal{T}(B))) * \mathcal{F}^n(\mathcal{T}(B))$.
Comme, $\mathcal{F}^{n+2}(\mathcal{T}(B)) = \mathcal{F}^{n+2}(T)$, on pose $C = \mathbf{C}(\mathcal{F}^{n+2}(B), \mathcal{F}^{n+2}(\mathcal{T}(B))) * \mathcal{F}^{n+1}(T)$.
On a :

$$\begin{aligned}
&\mathcal{F}^n(B)\\
&\quad = \mathbf{C}(\mathcal{F}^{n+1}(B), \mathcal{F}^{n+1}(\mathcal{T}(B))) * (p_{\mathcal{F}^{n+1}(\mathcal{T}(B))} * \mathcal{F}^{n+1}(T))\\
&\quad = (p_{\mathcal{F}^{n+1}(\mathcal{T}(B))} \circ \mathbf{C}(\mathcal{F}^{n+1}(B), \mathcal{F}^{n+1}(\mathcal{T}(B)))) * \mathcal{F}^{n+1}(T)\\
&\quad = (\mathbf{C}(\mathcal{F}^{n+2}(B), \mathcal{F}^{n+2}(\mathcal{T}(B))) \circ p_{\mathcal{F}^{n+1}(B)}) * \mathcal{F}^{n+1}(T)\\
&\quad = p_{\mathcal{F}^{n+1}(B)} * C
\end{aligned}$$

On pose

$$\boxed{\Uparrow^T_\sim A = (q(p_C, \mathcal{F}^{n+1}(B)) \curlywedge_{p_C * \mathcal{F}^{n+1}(B)} p_{\mathcal{F}^n(B)}) *^n B}$$

□

On étend maintenant $\Uparrow_\sim$ aux suites de transpositions.

Définition 11.10 *Soit A un objet de $\mathcal{C}$. Soit S une suite de transpositions sur $\mathcal{T}(A)$. On définit $\Uparrow^S_\sim A$ par récurrence sur $|S|$:*

1) *Si $|S| = 0$, on pose $\Uparrow^S_\sim A = A$.*

2) *Sinon, $S = T; S'$ où S' est une suite de transposition sur T. On pose $\Uparrow^S_\sim A = \Uparrow^{S'}_\sim (\Uparrow^T_\sim A)$.*

11.5 Les CGED comme axiomatisation de FocKer

Les espèces sont axiomatisées par les catégories de générateurs d'enregistrements avec effacement de définition. Plus précisement, si $\mathfrak{C}$ est une CGED, alors une espèce est un objet de $\mathfrak{C}$. Une interface est un objet pleinement abstrait de $\mathfrak{C}$. Si A est une interface, alors une collection d'interface A est un morphisme de $\mathbf{1}$ vers A.

Dans notre axiomatisation proposée, la paramétrisation est assez limitée : on ne peut paramétrer une collection que par une seule autre collection. Une collection d'interface B paramétrée par une collection d'interface A est un morphisme de A vers B. Dans ce cadre, l'instanciation d'une collection paramétrée est juste la composition.

De même, il n'y a pas vraiment d'espèce paramétrée par une collection. Mais on peut coder une certaine forme de paramétrisation en laissant certains champs "abstraits" dans l'espèce. A partir d'une telle espèce A, on engendre la collection paramétrée correspondant au morphisme $\mathbf{C}(A, \mathcal{T}(A)) \circ \mathbb{G}(A)$. Ainsi A joue le rôle d'une espèce paramétrée par une collection $\mathcal{H}(A)$, les champs de $\mathcal{H}(A)$ ayant été "aplatis" à l'intérieur de A.

11.5.1 Description des opérations de FocKer

Au niveau des interfaces,
- La relation d'inclusion (sans permutation ni renommage) est donnée par la relation $\lhd$.
- La relation de permutation est exprimée par l'existence d'une suite de transpositions.
- L'opération de renommage par une bijection κ sur les étiquettes est $\mathcal{R}_\kappa$.

Au niveau des collections,
- La conversion associée à l'inclusion est $\mathbb{C}$.
- La conversion associée à la permutation est $\mathbb{I}$.
- La conversion associée au renommage par une bijection κ sur les étiquettes est $\mathbb{R}_{\kappa^{-1}}$.

Au niveau des espèces,
- L'interface associée à une espèce A est $\mathcal{T}(A)$.
- Lorsqu'une espèce A est complètement définie (c'est-à-dire lorsque $\mathfrak{M}_A = Im(\mathfrak{L}_A)$), elle engendre la collection $\mathbf{C}(A, \mathcal{T}(A)) \circ \mathbb{G}(A)$.
- L'enrichissement correspond à l'opération "$*$".
- La spécialisation correspond à l'opération "$\ominus$".
- La redéfinition correspond à l'opération "$\cdot_{|\cdot}$" suivie de "$\ominus$".
- Pour la fusion de A et de B, il faut commencer par trouver une signature T (si elle existe) telle que $\mathcal{T}(A) \lhd T$ et telle qu'il existe une suite de transpositions S sur $\mathcal{T}(B)$ telle que $S(|S|) \lhd T$. On commence par calculer $B' = \Uparrow^T (\Uparrow^S_\sim B)$. La fusion est alors $(\Uparrow^T A) \oplus (B'_{|\mathfrak{M}'_B \cap \mathfrak{M}_A})$.
- Renommer une espèce par une bijection κ sur les étiquettes, c'est appliquer $\mathcal{R}_\kappa$.

11.5.2 Extensions et Améliorations

Vis-à-vis de FocKer, cette axiomatisation n'est pas complètement parfaite. Premièrement, il faudrait mieux représenter les collections et les espèces paramétrées. On pourrait s'inspirer de la notion de *"produits de familles de types"* des catégories contextuelles, présentée dans [Str91]. Cette notion introduite par Cartmell permet par exemple de représenter le λ-calcul avec types dépendants dans les catégories contextuelles.

Deuxièmement, l'opération de fusion proposée ci-dessus n'est pas très optimale. En effet, lors de l'opération de fusion de A et B, on commence par effacer dans B toutes les méthodes déjà définies dans A. Par conséquent, si A et B partagent une méthode qui porte la même définition (car par exemple ce sont deux espèces qui héritent d'une même espèce C où cette méthode a été définie), on va perdre toutes les méthodes de B qui dépendaient de manière transparente de cette méthode. Or on aurait pu les garder puisqu'on conserve finalement dans la fusion cette méthode définie dans A. Il faudrait donc affiner la définition de $\oplus$ de manière à pouvoir faire la fusion de deux générateurs ayant des définitions de méthode en commun.

11.6 Construction des modèles

Au cours de cette dernière partie, on a fourni des modèles aux catégories de signatures d'enregistrements et catégories de générateurs d'enregistrements (voir figure récapitulative 231). En particulier, on a montré comment passer d'une CS $\mathfrak{C}$ à une CG $\mathfrak{C}_\alpha$, et montré que si $\mathfrak{C}$ était régulière, alors $\mathfrak{C}_\alpha$ est une CS régulière.

Malheureusement, on n'a pas réussi au cours de cette thèse à faire une telle construction pour exhiber une CGED à partir d'une CG ou d'une CS. Malgré tout, pour toute CGR $\mathfrak{C}$ (CG avec ultra-régularité de p_A et α_A pour tout A), on a pu construire une CGED $\mathfrak{C}_D$. On ne va pas présenter cette construction dans cette thèse, car elle est assez technique (ie pénible), et d'un intérêt limité à notre avis.

En effet, le principal intérêt de cette construction est la technique employée. Il s'agit de stocker dans les générateurs, les dépendances de chaque champ. Ces dépendances ne sont pas "calculées", mais elles sont une donnée qui entre dans la construction du générateur au même titre que les noms des champs, leur type, leur définition. Deux "mêmes" générateurs ayant des relations de dépendances distinctes deviennent ainsi distincts. Plus précisément, dans cette construction, chaque champ d'une signature A est vu comme une signature élémentaire (c'est-à-dire la plus grande sur-signature de A dans laquelle ce champ est définissable). La signature A est alors égale à la réunion (au sens de $\bigcup^A$) de ces signatures élémentaires. C'est pour donner un sens à l'union, que l'on a dû se placer dans une CGR et non pas dans une simple CG. Pour écrire le pullback dans cette catégorie, on a du aussi utilisé la notion d'union sur les morphismes : les morphismes dans cette catégorie sont aussi décomposés en morphismes élémentaires pour exprimer les dépendances. En utilisant ces décompositions, on peut facilement construire pour tout objet A de cette catégorie, un objet $\mathbb{D}(A)$. Ainsi on utilise le théorème 11.3 pour montrer que cette catégorie est une CGED.

D'un autre côté, comme on l'a souligné en introduction de cette section la condition d'ultra-régularité de α_A exprime en fait déjà que les morphismes conservent les dépendances. Donc à notre avis, c'était une mauvaise piste d'essayer de bâtir une CGED à partir d'une CGR, et c'est pourquoi on ne présente pas cette construction $\mathfrak{C}_D$.

A notre avis, il faudrait en fait construire une CGED directement à partir d'une CS régulière. Il faudrait pour cela compliquer la construction de $\mathfrak{C}_\ltimes$ (où $\mathfrak{C}$ est une CS régulière) en conservant pour chaque définition élémentaire, les dépendances de cette définition. Dans une telle construction, on pourrait alors sans doute réutiliser les idées de la construction $\mathfrak{C}_D$. Cela semble cependant une tâche assez technique.

En effet, une telle construction serait en fait la description mathématique d'une implantation de l'environnement FOC. Cette implantation serait de plus générique, pour toute CS régulière.

On peut en particulier se convaincre que Coq en est une. Par exemple, $\mathfrak{C}_{\ltimes}$ est en fait déjà une implantation (générique mais pas très efficace) des CG à partir des CS.

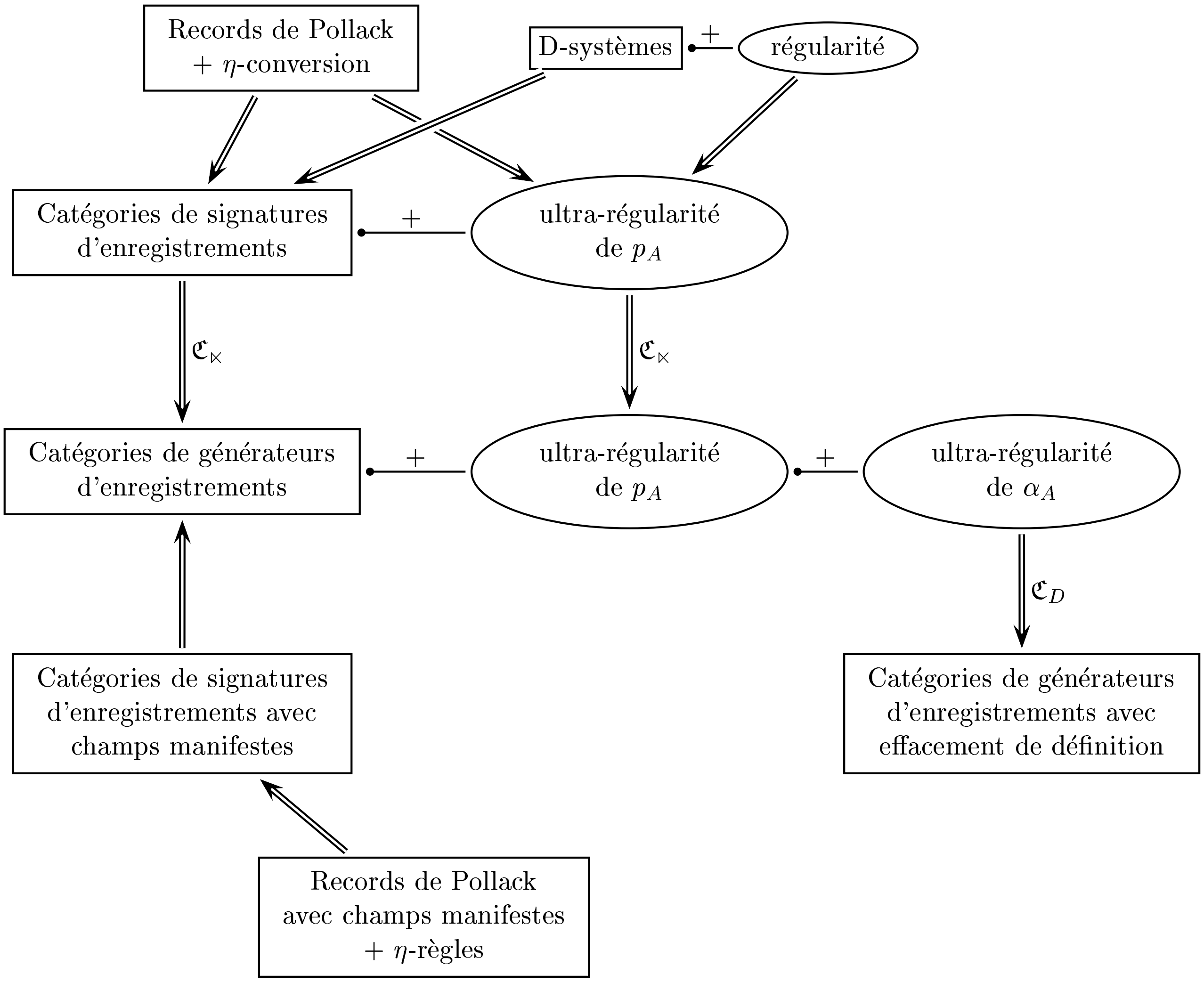


FIG. 11.1 – Modèles des différentes notions catégoriques

11.7 Conclusion

Ce chapitre propose donc un cadre mathématique pour spécifier FocKer. Ce cadre mathématique a l'avantage de ne pas fixer de syntaxe pour les interfaces, espèces et collections. Il reste cependant proche des types dépendants, et donc de Coq. On peut donc espérer que n'importe quel type d'implantation (plongement profond en Coq, plongement superficiel, etc ...) pourra s'y exprimer "naturellement".

Insistons sur le fait que ce cadre expose uniquement les propriétés sémantiques des opérations de FocKer. Il permet donc de se concentrer uniquement sur ces propriétés. Par exemple, la difficulté autour de l'ultra-régularité de α_A tient au fait, que l'on manipule une égalité entre méthodes a priori plus fine que l'égalité syntaxique. C'est difficile, car il faut déterminer exactement les propriétés de cette égalité. En contre-partie, si on réussit à surmonter cette difficulté, on peut espérer connaître l'égalité optimale qui satisfait cette propriété.

Ceci dit ce cadre présente parfois une certaine lourdeur dûe justement au manque de syntaxe : on ne peut pas s'appuyer sur les propriétés implicites que possède toute "bonne" syntaxe.

Remarquons finalement, qu'une partie de la pénibilité des preuves vient aussi de l'impression d'introduire deux fois les mêmes notions et de prouver deux fois les mêmes propriétés : une fois au niveau des objets et une fois au niveau des morphismes. Cela vient de la profonde symétrie entre objets et morphismes. Cette symétrie est exprimée en théories de types par le fait que les types sont eux-mêmes des termes. Cette symétrie n'est ici pas exprimée de façon générale par un axiome catégorique, mais elle se retrouve dans chacune des constructions catégoriques. Il y a donc peut-être moyen de simplifier cette axiomatisation, en utilisant la notion de *collection de types* utilisée par Streicher pour exprimer le fait que les types sont aussi des termes dans le calcul des constructions.

Chapitre 12

Conclusion

Résumé du travail Cette thèse spécifie le noyau d'un langage pour spécifier, implanter et prouver des bibliothèques de programmes de calcul formel. En fait, ce noyau pourrait être utilisé pour un tout autre domaine que le calcul formel, pourvu que la structuration des connaissances de ce domaine puisse se faire de manière analogue. Les contraintes de développement que nous avons eu à respecter sont en particulier :

1) Avoir un style de programmation "proche" des algorithmes de calcul formel et de la description mathématique des structures algébriques. Il faut en outre bien distinguer les structures algébriques (ex : algèbre libre) de leurs entités (polynômes), et de même distinguer la représentation informatique d'une entité mathématique de cette entité elle-même. En effet une entité mathématique (ex. un polynôme) peut avoir plusieurs représentations informatiques (ex. une liste ou un tableau), et inversement une même représentation informatique (ex. un mot machine) peut correspondre à plusieurs entités mathématiques distinctes (ex. 2 entiers modulaires sur 2 anneaux d'entiers distincts).

2) Pouvoir gérer facilement plusieurs niveaux d'abstraction pour effectuer les calculs : algorithmes génériques pour une large gamme de structures algébriques, algorithmes spécialisés pour des représentations des données particulières. Cette contrainte engendre beaucoup de reformulations du même concept : il faut donc favoriser une large réutilisation du code et des preuves.

Ces deux points correspondent à des paradigmes de programmation historiquement distincts : le style "type abstrait de donnée" pour le premier point, et le style "orienté objet" pour le second. En particulier, pour satisfaire le second point, on peut utiliser une notion d'héritage avec un mécanisme de redéfinition.

On a montré au chapitre 5 comment ces deux styles peuvent cohabiter en OCaml. Il reste que la redéfinition au cours de l'héritage est un exercice périlleux, car lorsqu'on redéfinit une méthode, il faut être conscient que cela peut avoir un effet sur l'ensemble des méthodes.

C'est justement là que l'environnement d'aide à la démonstration proposé devient une aide pour le programmeur. En effet, en fonction des spécifications que le programmeur a écrit pour chacune des méthodes, l'assistant engendre des obligations de preuve lors des redéfinitions, afin de garantir la cohérence de ces redéfinitions. Même si le programmeur ne donne pas immédiatement une preuve formelle de ces obligations, ces obligations lui donneront un support pour vérifier informellement que ces redéfinitions sont correctes.

Cette notion d'obligation de preuves repose en fait sur la notion de dépendance transparente et dépendance opaque dans les types dépendants. Ces deux types de dépendances distinguent en fait les démonstrations génériques de propriétés d'une méthode, et les démonstrations spécialisées.

Au niveau du langage de programmation, notre approche est assez originale, puisqu'elle propose une forme restreinte de liaison tardive, qui est statique et non récursive. En cela, notre approche se distingue radicalement de la sémantique récursive (ou coinductive) usuelle de l'héritage dans les langages orientés objet. Comme remarqué dans la conclusion du chapitre 5, les collections (nos unités de bibliothèque) peuvent être créées statiquement. Le système des espèces, qui permet de spécifier la construction des collections, peuvent donc être vues au niveau du langage de programmation, comme une sorte de langage de macros qui sont exécutées à la compilation. Ceci laisse entrevoir de possibles optimisations par rapport au prototype actuel en OCaml, qui n'utilise pas vraiment ce caractère statique des collections.

Côté spécification (et preuves), ce système présenté informellement en 4 a été d'abord formalisé en Coq (chapitre 6). Cette formalisation représente les interfaces, les collections et les espèces comme des termes Coq. Elle représente surtout les liens entre ces différents éléments. Il s'agit donc d'une formalisation des opérations de hiérarchisation des unités de la bibliothèque FOC. Autrement dit, il s'agit en quelque sorte d'une sémantique opérationnelle de la hiérarchie. Des propriétés méta-théoriques de ce système ont ainsi été dégagées. Le développement effectué en Coq garantit en fait la cohérence de la hiérarchie et de ses mécanismes d'extension. Il est aussi possible d'utiliser directement cette formalisation pour spécifier et prouver une telle bibliothèque en Coq : les méthodes dans les unités de bibliothèque peuvent être des termes Coq quelconque. Cette possibilité est cependant surtout théorique, car en pratique, ce système codé en Coq est encore très lourd à interpréter en Coq.

Dans la partie III, nous avons tenté de dégager un modèle qui soit *une caractérisation* de la hiérarchie. Nous nous sommes appuyés pour cela sur les langages pour types dépendants et en particulier les catégories contextuelles introduites par Cartmell. En effet, les éléments de notre hiérarchie contiennent aussi des énoncés et des preuves engendrant des dépendances de différente nature. Nous avons choisi la théorie mathématique des catégories comme méta-langage de notre axiomatisation, car ce méta-langage n'exige pas de syntaxe. En effet, celui-ci ne parle (presque) que de morphismes, que nous interprétons comme des liens de dépendance entre unités.

Le but de notre étude est de caractériser l'ajout d'une unité dans la hiérarchie comme une suite d'opérations élémentaires. Il s'agit alors de définir une sémantique pour chacune de ces opérations qui préserve la cohérence globale de la hiérarchie. Une unité de la hiérarchie est d'abord abstraite en un ensemble de champs étiquettés constituant une signature d'enregistrement. La hiérarchie elle-même est vue comme une catégorie de signatures d'enregistrements, c'est-à-dire un arbre dont les nœuds sont des signatures, muni d'une opération d'extension qui préserve la structure d'arbre. Cette opération d'extension est en fait un produit fibré, qui traduit la notion de paire dépendante des systèmes de types dépendants. Avec cette modélisation, on peut alors identifier les opérations de renommage de champs, d'oubli de champs, de permutations de champs, comme la construction de morphismes vérifiant certaines propriétés. Cette première approche permet aussi d'introduire la notion de morphismes décomposables en définitions élémentaires. Un tel morphisme représente donc le processus de création d'une unité de la bibliothèque comme une suite de déclarations (contenues dans le domaine de ce morphisme) et de telles définitions élémentaires.

Dans les catégories de signatures, on ne dit cependant rien sur la capacité des signatures à contenir des définitions. Ceci est en fait précisé par une nouvelle notion catégorique (qui "hérite" de la notion de catégorie de signature) : les catégories de générateurs d'enregistrements (ou CG), chapitre 9. Les objets de cette catégorie sont donc des signatures qui contiennent des champs avec des déclarations (champs dits abstraits) et des champs avec des définitions (champs dits manifestes). La sémantique de ces objets est donnée par la notion de morphismes décomposables en définitions élémentaires : chacun de ces objets représente un tel morphisme, dont le domaine est constitué uniquement des champs abstraits de l'objet. Il faut noter (cf. chapitre 10) que ces catégories représentent une notion de signature plus générale que la notion de signatures avec champs manifestes comme celle de Pollack ([Pol00]) ou que l'on rencontre dans les systèmes de modules comme dans [Cou98]. En effet, on a besoin ici d'une variante de ces systèmes dans lesquelles chaque signature admet une sur-signature principale unique et dont les champs soient tous abstraits. Ceci est nécessaire pour garantir que les champs soient redéfinissables.

Cette possibilité de redéfinir des champs est exprimée dans une spécialisation de la notion de CG : les CG avec effacement de définition (ou CGED). En effet, la redéfinition d'un champ est vue comme l'effacement de la définition de ce champ (qui provoque aussi l'effacement de toutes les définitions qui dépendaient de manière transparente de ce champ) suivi d'une nouvelle définition de ce champ. Toute catégorie qui est une CGED peut être vue comme un modèle (ou une implantation) de FocKer. On a en effet décrit dans cette notion catégorique l'ensemble des traits de FocKer. Il reste toutefois quelques maladresses : une gestion rudimentaire du paramétrage des collections et des espèces, et une définition assez naïve de l'héritage multiple.

Dans l'embryon de l'environnement FOC ([Pre00]), Virgile Prevosto et Damien Doligez ont choisi d'utiliser un codage plus "superficiel" en Coq que celui présenté au chapitre 6, car il est moins lourd à utiliser. L'axiomatisation catégorique fournit un cadre formel à cette implantation.

Liens avec d'autres travaux Au long des deux derniers chapitres, on a montré les connexions de ce travail avec les systèmes d'enregistrements dépendants (cf. [Pol00] et [Bet98]) ou de modules (cf. [Cou98]). Le travail du chapitre 6 est surtout proche de [Pol00], car il présente un codage des enregistrements à l'intérieur de la théorie des types.

Remarquons aussi que l'idée de mélanger la programmation style "type abstrait de donnée" avec la programmation orientée objet est présente dans de nombreux travaux, comme [Apo96], [Vou98], [Anc98] ou [Hir00]. Il faut noter que l'approche de [Hir00] est proche de la notre dans le sens où ce travail ajoute de l'héritage aux modules de OCaml en préservant leur caractère statique.

Enfin, dans [Cas99], Castagna et Chen proposent un système[1] de surcharge avec liaison tardive dans les types dépendants qui est fortement normalisant. Contrairement à notre approche qui exprime l'héritage via la notion d'enregistrements extensibles, ces travaux expriment l'héritage en ajoutant au $\lambda\Pi$-calcul les notions de sous-typage et de surcharge. La liaison tardive de Castagna est en fait une forme fine de liaison dynamique : la sélection du code associé à un nom de fonction est déterminée par le type des arguments de cette fonction à l'exécution. En effet, à cause du sous-typage, chaque terme a un type statique, et un type dynamique qui est sous-type de son type statique : par exemple, si f est une fonction $[x : A]b$, et t un terme, l'application $(f\ t)$ est bien typée si le type statique de t est un sous-type de celui de A, et à l'évaluation de cette application, le type dynamique de x dans b est le type dynamique de t.

Donc notre approche se distingue de celle de Castagna sur deux points. D'une part, dans notre approche, l'information de type à l'exécution n'est pas nécessaire à l'exécution et sélectionner une méthode, c'est sélectionner le champ d'un enregistrement. D'autre part, la liaison est statique. La liaison entre méthodes d'une même espèce n'est "retardée" que jusqu'à la compilation.

L'approche de Castagna est donc strictement plus expressive et semble très prometteuse. Cependant, nous nous situons dans un cadre différent puisque nous désirons, dans l'état actuel des connaissances et des outils à notre disposition, conserver un typage statique et une liaison statique au niveau du langage de programmation sous-jacent à notre système. Plusieurs raisons justifient (à

[1]extension de [Cas95]

l'heure actuelle) ce choix. D'une part ces mécanismes dynamiques sont plus coûteux à l'exécution que les mécanismes statiques : en particulier dans le cadre d'une programmation offensive (qui utilise les propriétés statiques des données). Mais surtout, les mécanismes statiques nous semblent plus sûrs, pour écrire des programmes que l'on ne souhaite pas prouver. En effet, les mécanismes de surcharge rendent la sémantique des programmes plus dynamique, et donc plus difficile à maîtriser par le programmeur : celui-ci doit faire plus d'effort pour connaître le code associé à chaque occurrence du nom d'une fonction. Il faut rappeler dans le cadre du projet FOC, nous souhaitons laisser la possibilité à l'utilisateur de ne pas tout prouver. Nous souhaitons cependant que les programmes non prouvés de l'utilisateur soient fiables et cohérents vis-à-vis des programmes prouvés. C'est pour cela que nous souhaitons que le langage de programmation soit le même que ce soit pour écrire des programmes prouvés ou pas. Cela laisse de plus la possibilité de retarder la preuve de certains programmes (pour par exemple tester leur efficacité avant de les prouver).

Perspectives Outre les améliorations à apporter à l'axiomatisation pour qu'elle corresponde mieux à FocKer (cf. section 11.5.2), on a mentionné dans la section 5.3 quelques traits de programmation qu'on aimerait retrouver dans FocKer, comme une notion de liaison ancêtre ou d'espèce récursive.

Il reste que la méthodologie proposée au développeur de bibliothèques certifiées FOC n'est pas encore suffisament élaborée pour être réellement utilisable en pratique. En particulier, elle part du paradigme qu'il faut "rester abstrait et déclaratif" le plus possible, ce qui est indispensable pour permettre la réutilisation de preuves, mais ce qui rend très pénible la construction des preuves. On voit en effet ressurgir ici une manière "axiomatique" de spécifier, dont un système comme Coq a essayé de s'écarter, en favorisant un style plus "constructif" de spécification. Autrement dit, dans notre approche, comme on n'a que des types "abstraits", on perd le bénéfice de pouvoir calculer sur ces types au cours des démonstrations. Il faudra compenser cette perte en intégrant dans le futur environnement FOC des outils de réécriture pour le développeur. Il faudra que ces outils s'adaptent facilement aux spécifications du développeur, afin que celui-ci puisse les utiliser pour certifier ses programmes. Si dans cette thèse, on a essayé de concilier *abstraction de types et héritage*, il reste maintenant à concilier *abstraction de types et calcul au cours des preuves.*

Découpage du temps de la thèse Dans toute expérience de spécification d'un système informatique, il est intéressant de regarder le temps passé dans chacune des phases de ce travail. Dans mon cas, ce temps est découpé de la manière suivante :
- 1ère année : mise au point de la spécification informelle et prototypage en OCaml.
- 2ième année : développement du système de spécification en Coq.
- 3ième année : axiomatisation catégorique et rédaction de la thèse.

Il faut noter que ces deux premières années ont béneficié de la collaboration au sein de l'action *"Calcul Formel Certifié"* [2] et des riches discussions qu'elle a suscitées.

En conclusion, la taille de chapitres de cette thèse ne reflète en rien le temps de travail qu'a nécessité leur élaboration. La plus grosse partie du temps a consisté à essayer de comprendre "ce que l'on voulait faire".

[2]action INRIA entre les équipes SPI et CALFOR du LIP6, et les équipes CROAP (maintenant LEMME), COQ (maintenant LOGICAL) et CRISTAL de l'INRIA.

Bibliographie

[Aba96] M. Abadi et L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[Abr96] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[Anc98] D. Ancona et E. Zucca. An algebra of mixin modules. Dans *WADT'97*, LNCS 1376. Springer-Verlag, janvier 1998.

[Apo96] M.-V. Aponte et G. Castagna. Programmation modulaire avec surcharge et liaison tardive. Dans *Journées Francophones des Langages Applicatifs*. 1996.

[Asp91] A. Asperti et G. Longo. *Categories, Types and Structure*. MIT-Press, 1991.

[Bar99] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, 1999.

[Bar00] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Muñoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin-Mohring, A. Saïbi et B. Werner. *The Coq Proof Assistant Reference Manual, version 6.3.1*. Coq Project, INRIA-Rocquencourt, mai 2000.

[Bet98] G. Betarte. *Dependent record types and formal abstract reasoning*. Thèse de doctorat, Chalmers University of Technology, 1998.

[Bor89] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang et V. Pascual. Centaur : the system. Dans *Proceedings of the ACM SIGSOFT'88 on Practical Software Development Environments*, tome 24(2) de *SIGPLAN Notices*. ACM, 1989.

[Bou94] S. Boutin. *Correctness proof of a mini-ml compiler.* Coq Contribution, décembre 1994. Disponible sur `http ://pauillac.inria.fr/coq/contribs/mini-ml.html`

[Bou97] S. Boutin. *Réflexions sur les quotients*. Thèse de doctorat, Paris 7, avril 1997.

[Bou99a] S. Boulmé, T. Hardin et R. Rioboo. Modules, objets et calcul formel. Dans *Journées Francophones des Langages Applicatifs*. INRIA, février 1999.

[Bou99b] S. Boulmé, T. Hardin, D. Hirschkoff, V. Ménissier-Morain et R. Rioboo. On the way to certify computer algebra systems. Dans *Calculemus 99 Systems for Integrated Computation and Deduction*, tome 23 de *ENTCS*. Elsevier, 1999.

[Bou00] S. Boulmé. Opérateurs de raffinement sur les structures algébriques. Dans *Journées Francophones des Langages Applicatifs*. INRIA, février 2000.

[Bro71] W.-S. Brown et J.-F. Traub. On Euclid's algorithm and the theory of subresultants. *J. ACM*, pp. 505–514, octobre 1971.

[Bro78] W.-S. Brown. The subresultant PRS algorithm. *ACM TOMS*, pp. 237–279, septembre 1978.

[Car86] J. Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32 :209–243, 1986.

[Cas95] G. Castagna, G. Ghelli et G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1) :115–135, 1995.

[Cas97] G. Castagna. *Object-Oriented Programming : A Unified Foundation*. Progress in Theoretical Computer Science Series. Birkäuser, Boston, 1997.

[Cas99] G. Castagna et G. Chen. Dependent types with subtyping and late-bound overloading. *Information and Computation*, 1999. To appear.

[Cha00] E. Chailloux, P. Manoury et B. Pagano. *Développement d'applications avec Objective Caml*. O'REILLY, 2000.

[Col67] G.-E. Collins. Subresultant and reduced polynomial remainder sequences. *J. ACM*, pp. 128–142, janvier 1967.

[Coq85] T. Coquand. *Une théorie des constructions*. Thèse de doctorat, Université Paris 7, 1985.

[Coq86] T. Coquand. An analysis of Girard's paradox. Dans *Symposium on Logic in Computer Science*. Cambridge, MA. IEEE PRESS, 1986.

[Coq88] T. Coquand et G. Huet. The calculus of constructions. *Information and Computation*, 76 :95–120, 1988.

[Coq90] T. Coquand et C. Paulin-Mohring. Inductively defined types. Dans *Proceedings of Colog'88*, tome 417 de *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[Cor97] C. Cornes. *Conception d'un langage de haut niveau de représentation de preuves*. Thèse de doctorat, Université Paris 7, 1997.

[Cou98] J. Courant. *Un calcul de modules pour les systèmes de types purs*. Thèse de doctorat, Ecole Normale Supérieure de Lyon, 1998.
Disponible sur `ftp ://ftp.lip.ens-lyon.fr/pub/LIP/Rapports/PhD/PhD98-03.ps.Z`

[Dav93] J. Davenport, Y. Siret, E. Tournier et D. Lazard. *Computer Algebra*. Masson, 1993.

[dB91] N. G. de Bruijn. Telescoping mapping in typed lambda calculus. *Information and Computation*, pp. 189–204, avril 1991.

[Dol93] D. Doligez et X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. Dans *Proc. 20th symp. Principles of Programming Languages*, pp. 113–123. ACM press, 1993.

[Dol95] D. Doligez. *Conception, réalisation et certification d'un glaneur de cellule concurrent*. Thèse de doctorat, Université Paris 7, 1995.

[Dow98] G. Dowek, T. Hardin et C. Kirchner. Theorem proving modulo. Rapport technique 178, INRIA, avril 1998.

[Dow99a] G. Dowek. *La part du calcul*. Mémoire d'habilitation à diriger des recherches, Université Paris 7, 1999.

[Dow99b] G. Dowek. *Théories des types*. Notes de cours du DEA Sémantique, Preuves et Programmation, 1999.
Disponible sur `http ://pauillac.inria.fr/~dowek/cours.html`

[Dub99] C. Dubois et V. Ménissier-Morain. Certification of a type inference tool for ml : damasmilner within coq. *Journal of Automated Reasoning*, 23(3-4), 1999.

[Duc96] L. Ducos. Algorithme de Bareiss, algorithme des sous-résultants. *Informatique théorique et Applications*, pp. 319–347, avril 1996.

[Fil99] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. Thèse de doctorat, Université Paris-Sud, juillet 1999.
Disponible sur `http ://www.lri.fr/~filliatr/ftp/publis/these.ps.gz`

[Ged93] K.-O. Geddes, S.-R. Czappor et G. Labahn. *Algorithms for computer algebra*. Kluwer Academic Publishers, 1993.

[Har94] R. Harper et M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. Dans *21st Symposium on Principles of Programming Languages*, pp. 123–137. ACM PRESS, 1994.

[Hir00] T. Hirschowitz. *Modules mixins, typage et implantation*. Mémoire de DEA, Paris 7-INRIA, 2000.

[Ho96] C.-J. Ho et C.-K. Yap. The Habicht approach to subresultants. *Journal of Symbolic Computation*, janvier 1996.

[Hue99] G. Huet, G. Kahn et C. Paulin-Mohring. The Coq proof assistant - a tutorial, version 6.2.4. Rapport technique, INRIA, janvier 1999.

[Jen92] R.-D. Jenks et R.-S. Stutor. *AXIOM, The Scientific Computation System*. Springer-Verlag, 1992.

[Kap97] D. Kapur. Automated geometric reasoning : Dixon resultants, Gröbner bases, and characteristic sets. Dans *Proceedings of Automated Deduction in Geometry'96*, tome 1360 de *Lecture Notes in Artificial Intelligence*, pp. 1–36. Springer-Verlag, 1997.

[Ler92] X. Leroy. *Typage polymorphe d'un langage algorithmique*. Thèse de doctorat, Université Paris 7, 1992.

[Ler94] X. Leroy. Manifest types, modules, and separate compilation. Dans *Proc. 21st symp. Principles of Programming Languages*, pp. 109–122. ACM press, 1994.

[Ler95] X. Leroy. Applicative functors and fully transparent higher-order modules. Dans *Proc. 22nd symp. Principles of Programming Languages*, pp. 142–153. ACM Press, 1995.

[Ler98] X. Leroy. Introduction. Dans *Proceedings of the 1998 Types in Compilation workshop*, tome 1473 de *Lecture Notes in Computer Science*, pp. 1–8. Springer-Verlag, 1998.

[Ler99] X. Leroy, D. Rémy, J. Vouillon et D. Doligez. *The Objective Caml system, release 2.04*. Projet Cristal, INRIA-Rocquencourt, 1999.
Disponible sur `http ://pauillac.inria.fr/ocaml/`

[Ler00a] X. Leroy. A modular module system. *Journal of Functional Programming*, 2000. To appear.

[Ler00b] X. Leroy et F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 2000. To appear.

[Lom00] H. Lombardi, M.-F. Roy et M. Safey El Din. New structure theorems for subresultant. *Journal of Symbolic Computation*, avril 2000. Special Issue on Symbolic Computation in Algebra, Analysis and Geometry.

[Loo83] R. Loos. Generalized polynomial remainder sequences. Dans *Computer Algebra*, pp. 115–137. Springer-Verlag, 1983.

[Luo94] Z. Luo. *Computation and Reasoning, A Type Theory for Computer Science*. International Series of Monographs on Computer Science. Oxford University Press, 1994.

[Luo99] Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1) :105–130, 1999.

[MM94] V. Ménissier-Morain. *Arithmétique exacte : conception, algorithmique et performances d'une implémentation informatique en précision arbitraire*. Thèse de doctorat, Université Paris 7, décembre 1994.

[Mon96] J.-F. Monin. *Comprendre les méthodes formelles*. Masson, 1996.

[Mor95] M. Moreno Maza et R. Rioboo. Polynomial gcd computations of towers of algebraic extensions. Dans *Proceedings of AAECC'95*, tome 948 de *Lecture Notes in Computer Science*, pp. 365–382. Springer-Verlag, 1995.

[Mot99] M. Mottl. *Translation to OCaml of Okasaki's SML-sources from "Purely Functional Data Structures"*. Communication sur Internet, 1999.
Disponible sur `http ://miss.wu-wien.ac.at/~mottl/ocaml_sources/intro.html`

[Mul89] J.-M. Muller. *Arithmétique des ordinateurs*. Études et recherches en informatique. Masson, 1989.

[Mul95] J.-M. Muller. Algorithmes de division pour microprocesseurs : illustration à l'aide du "bug" du Pentium. *Technique et Science Informatiques*, 14(8), octobre 1995.

[Nag89] E. Nagel, J.-R. Newman, K. Gödel et J.-Y. Girard. *Le théorème de Gödel*. Collection Points. Éditions du Seuil, 1989.

[Oka98] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[Pfe93] F. Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 19 :185–199, 1993.

[PM89] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. Thèse de doctorat, Paris 7, janvier 1989.
Disponible sur `http ://www.lri.fr/~paulin/these.ps.gz`

[PM93a] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. Dans *Proceedings of the conference Typed Lambda Calculi and Applications*, numéro 664 dans Lecture Notes in Computer Science. 1993.

[PM93b] C. Paulin-Mohring et B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15 :607–640, 1993.

[Pol00] R. Pollack. Dependently typed records for representing mathematical structures. Dans *TPHOLs'00*. Springer-Verlag, 2000.

[Pon99] O. Pons. *Conception et réalisation d'outils d'aide au développement de grosses théories dans les systèmes de preuves interactifs*. Thèse de doctorat, Conservatoire National des Arts et Métiers, 1999.

[Pot98] L. Pottier. *Les polynômes comme image par le foncteur adjoint du foncteur d'oubli des algèbres vers les ensembles (D'après D. Lazard)*. Contribution Coq, 1998.
Disponible sur `http ://pauillac.inria.fr/coq/contribs/algebra.html`

[Pot99] L. Pottier. *Télescopes : des records généralisés définis dans Coq lui-même*. Communication personnelle, 1999.
Disponible sur `http ://www-sop.inria.fr/croap/CFC/Tel/index.html`

[Pre00] V. Prevosto. *Vers une interface utilisateur pour FoC*. Mémoire de DEA, LIP6-INRIA, 2000.

[Rec90] T. Recio, L. Gonzáles-Vega, H. Lombardi et M.-F. Roy. Spécialisation de la suite de Sturm et sous-résultants. *Informatique théorique et Applications*, pp. 561–588, décembre 1990.

[Rém97] D. Rémy et J. Vouillon. Objective ML : A simple object-oriented extension of ml. Dans *Proceedings of the 24th ACM Conference on Principles of Programming Languages*, pp. 40–53. Paris, France, janvier 1997.

[Rém98] D. Rémy. *Des enregistrements aux objets*. Mémoire d'habilitation à diriger des recherches, Université de Paris 7, 1998.

[Rio92] R. Rioboo. Real algebraic closure of an ordered field, implementation in Axiom. Dans *Proceedings of Issac'92*, pp. 206–215. ACM-Press, 1992.

[Saï99] A. Saïbi. *Outils génériques de modélisation et de démonstration pour la formalisation des mathématiques en théorie des types ; application à la théorie des catégories.*. Thèse de doctorat, Université Paris 6, 1999.

[Sol99a] S. Soloviev et Z. Luo. Coercion completion and conservativity in coercive subtyping. *Annals of Pure and Applied Logic*, To appear, 1999.

[Sol99b] S. Soloviev et Z. Luo. Dependent coercion. Dans *Proceedings of CTCS'99*, tome 29 de *ENTCS*. Elsevier, 1999.

[Str91] T. Streicher. *Semantics of Type Theory*. Progress in Theoretical Computer Science. Birkäuser, 1991.

[Vou98] J. Vouillon. Using modules as classes. Dans *Informal proceedings of the FOOL'5 workshop*. 1998.
Disponible sur `http ://pauillac.inria.fr/~remy/fool`

[Wat95] S. Watt, P. Broadbery, S. D. P. Iglio, S. Morrison, J. Steinbach et R. Sutor. *AXIOM Library Compiler User Guide*. NAG Ltd, mars 1995.

[Wer94] B. Werner. *Une Théorie des Constructions Inductives*. Thèse de doctorat, Université Paris 7, 1994.

[Xan00] S. Xanthakis, P. Régnier et C. Karapoulios. *Le test des logiciels*. Hermes Science Publications, 2000.

[Zip93] R. Zippel. *Effective polynomial computation*. Kluwer Academic Publishers, 1993.

Index