Un environnement pour la programmation avec types dépendants

Sozeau Matthieu

ORSAY
N^o d'ordre : 9279

UNIVERSITÉ DE PARIS-SUD 11

CENTRE D'ORSAY

THÈSE

présentée
pour obtenir

le grade de docteur en sciences DE L'UNIVERSITÉ PARIS XI

PAR

Matthieu Sozeau

—×—

SUJET :

Un environnement pour la programmation avec types dépendants

soutenue le 8 décembre 2008 devant la commission d'examen composée de

Mme	Véronique	Benzaken	
Mr	Thierry	Coquand	Rapporteur
Mr	James	McKinna	
Mme	Christine	Paulin-Mohring	Directrice
Mr	François	Pottier	Rapporteur
Mr	Philip	Wadler	

Remerciements

Cette thèse n'aurait jamais pu se réaliser sans les nombreuses personnes qui m'ont soutenu ou que j'ai simplement rencontré au fil de ces trois années, et qui m'ont donné à vivre la plus enrichissante des expériences professionelles. Ces années sont passées bien trop vite à mon goût, aussi j'aimerais ici prendre le temps de remercier quelques-uns des participants. J'adresse :

Un *merci* respectueux à Christine Paulin-Mohring, pour sa direction des plus discrètes mais néanmoins efficace et sa grande souplesse (horaire en particulier). Toujours là lorsque j'avais des doutes et attentive à mes balbutiements, je ne sais quoi écrire pour saluer tant de sagesse curieuse et de bonne volonté.

Un *merci* admiratif à Thierry Coquand et François Pottier pour avoir accepté d'être mes rapporteurs et de m'avoir fournit des remarques précises, nombreuses et pertinentes sur ce manuscript, qui ne serait pas en si bon état sans leur concours.

An honored *thanks* to my external examiners James McKinna and Philip Wadler for accepting to be part of this jury and for their inspiring work.

Un *merci* d'écolier à ma professeur Véronique Benzaken pour avoir accepté de prendre part à ce jury.

Un *merci* espiègle à Jean-Christophe Filliâtre et Sylvain Conchon, le premier m'a donné la vocation de la recherche comme prolongement logique d'une jeunesse passée à hacker, le second m'a aidé à passer de l'autre côté du bureau. Surtout, ces deux-là communiquent une bonne humeur (au coin café et en dehors) que je ne retrouverait nulle part ailleurs.

Un *merci* chaleureux à la grande famille des Démons, toujours prêts à partager un café, un gâteau et bien plus encore : Évelyne et Claude, Marc, Sylvie, Louis, les convives éphémères Andrei, Nicolas, Jean-François, Darek et Goshia, et mes chers compagnons d'infortune d'hier, Julien, Pierre et Thierry, et d'aujourd'hui : Florence et Yannick (prem's !), Johannes, Stéphane, Romain et Wendy (courage !), et les derniers arrivants : Cédric et François.

Un *merci* particulier à Nicolas Oury pour les discussions à n'en plus finir (d'ailleurs, ce n'est pas fini !) et son iconoclasme savoureux !

Un *merci* gallinacé aux membres de l'équipe TypiCal : Hugo, Benjamin, Bruno, Gilles et Jean-Marc, sources inépuisables de réflexions sur Coq, la théorie des types et bien d'autres sujets, et les jeunes pousses : Arnaud (je ne rentrerai pas dans les détails !), Élie, Denis, Lisa, Pierre-Yves, Mathieu, Matthias, Bruno et Danko.

Un *merci* amusé à Pierre, Stéphane et Nicolas de PPS et à Pierre, Julien et Xavier du CNAM, casser Coq est un métier à temps plein.

Un *merci* chameau aux membres de l'équipe Gallium : Xavier, Gérard, Didier et Damien ainsi qu'Arthur, les deux Benoît, Boris, Jean-Baptiste, Nicolas, Yann et Zaynah. Merci

aux déjà nombreuses générations de chercheurs ayant contribué à ces magnifiques outils que sont OCaml et Coq.

Un *merci* caféiné à Assia, François, Georges, Guillaume et Jean-Jacques du laboratoire commun INRIA-Microsoft, toujours une halte agréable.

Un *merci* british à Conor, Thorsten, Peter et Wouter pour m'avoir acceuilli quelques jours à Nottingham.

Un cocktail de *merci* à Alycia et Andrew Tolmach qui m'ont accueilli à Portland et un *merci* d'avance à Avi, Greg et Paul pour m'avoir donné un avant goût de Boston.

Un *merci* éloigné à la bande des thésards, Cédric, Guillaume, Kim et Sylvain.

Un *merci* amical à Grégoire, Julien, Pierre-Loïc et Pascaline, Marion et Nicolas, Vincent, Kassandra et Julie pour m'avoir rammené sur terre quand il le fallait.

Un *merci* familial à Bastien, Joëlle, Christine et Edwige, toujours prêts à m'épauler, même quand je fait l'ours.

Enfin, un *merci* amoureux à Claire, qui ne cessera jamais de me surprendre.

Résumé

Les systèmes basés sur la Théorie des Types prennent une importance considérable tant pour la vérification de programmes qu'en tant qu'outils permettant la preuve formelle de théorèmes mettant en jeu des calculs conséquents et complexes.

Ces systèmes nécessitent aujourd'hui une grande expertise pour être utilisés efficacement. Dans cette thèse, nous étudions des extensions au système Coq facilitant la programmation, le raisonnement et l'organisation des développements tout en conservant et mettant en avant toutes les capacités du langage à types dépendants sous-jacent. Nos contributions se situent en dehors du noyau de Coq et sont ainsi exportables à d'autres systèmes basés sur la Théorie des Types.

Dans une première partie, nous étudions un langage source plus flexible au dessus de Coq qui permet d'identifier les objets calculatoirement équivalents mais qui n'ont pas forcément les mêmes propriétés. Nous développons ensuite une interprétation des termes bien typés de ce langage dans Coq qui donne lieu à la génération d'obligations de preuve. On peut séparer ainsi le typage d'une fonction fortement spécifiée de la preuve de ses conditions de correction, tout comme il est communément fait pour la preuve de programmes impératifs. Nous démontrons les propriétés métathéoriques essentielles du système, dont les preuves sont en partie mécanisées, et détaillons son implémentation dans l'assistant de preuve Coq.

D'autre part, nous décrivons l'intégration et l'extension d'un système de "Type Classes" venu d'Haskell et Isabelle à Coq via une simple interprétation des constructions liée aux classes dans la théorie des types sous-jacente. Nous démontrons l'utilité des classes de types dépendantes pour la spécification et la preuve et présentons notamment une implémentation économique et puissante d'une tactique de réécriture généralisée basée sur les classes.

Pour mettre à l'épreuve ces deux implémentations, nous avons développé une bibliothèque de manipulation de "Finger Trees", une structure de données complexe. En programmant avec les types sous-ensemble, les familles inductives ainsi que la surcharge apportée par les classes, nous construisons une implémentation certifiée de cette structure. Nous montrons comment à partir de celle-ci et de façon modulaire nous pouvons obtenir des implémentations de structures de plus haut niveau avec autant de garanties statiques.

Table des matières

Table des figures

Chapitre 1

Introduction

Sommaire

1.1 Un peu d'histoire

Qu'est-ce que le calcul, qu'est que le raisonnement ? Qu'est-ce qu'un programme, qu'est-ce qu'une preuve ? Que peut-on mécaniser, qu'est-ce qui échappe à la formalisation en mathématique ? Peut-on formaliser l'ensemble de la connaissance et en déduire toutes les lois pour prédire les effets des phénomènes qui nous entourent ?

Ces questions, pour certaines toujours d'actualité, ont pris une importance considérable avec la naissance des premiers calculateurs, sous la forme de la Pascaline ou encore de l'ordinateur de Babbage au 19ème. Ces "*machines*" ont été inventées dans le but d'éviter à l'homme les travaux fastidieux, répétitifs et ennuyeux du calcul numérique. Mais surtout, elles ont été inventées dans le but de fournir des résultats *corrects* desquels on ne puisse douter. Babbage comptait corriger avec cet outil les nombreuses erreurs qui apparaissaient dans les tables de logarithmes rédigées par ses contemporains. Il n'y parvint pas, la mécanique de l'époque ne lui permettant pas de construire sa machine. Mais l'idée du calcul mécanisé lui a survécu, et au cours des deux derniers siècles des progrès considérables ont

été réalisés sur les machines pour qu'aujourd'hui nous soyons entourés d'ordinateurs qui calculent, ici la météo de demain, là les 1 241 milliards premières décimales de π, mais permettent aussi de communiquer via internet ou encore d'exprimer son talent grâce à des interfaces ingénieuses. Malheureusement, la correction des machines et plus encore celle des innombrables programmes qu'elles exécutent n'est pas plus assurée qu'il y a deux cent ans, exceptée pour une infime partie de ceux-ci. Leur complexité a suivi d'assez près la progression des moyens de calcul.

Cette incorrection est aujourd'hui rentrée dans les mœurs. "Bugs" et "plantages" font partie du quotidien des utilisateurs de système informatiques, qui bien heureusement ne se retournent jamais contre leurs concepteurs. Cet état de fait est tout simplement le résultat d'une difficulté majeure dans la conception de programmes corrects : les outils permettant leur vérification n'ont été développés que très récemment et leur utilisation nécessite une expertise rare.

1.1.1 Incomplétude, Indécidabilité

Il faut dire qu'au cours du 20$^{\text{ème}}$ siècle, on a vu un projet de formalisation très ambitieux des mathématiques lancé par Hilbert se heurter à un résultat d'impossibilité essentiel démontré par Gödel, qui a refroidi (parfois à raison) bien des ardeurs à vouloir mécaniser le raisonnement. On pourra consulter le livre de van Heijenoort (2002) pour une compilation commentée des articles correspondants, et celui de Girard (2006) pour une introduction complète au développement de la logique au 20ème siècle.

Revenons un peu sur l'histoire récente qui nous a mené à cette situation. Au début du vingtième siècle, des travaux sur les fondations des mathématiques ont préoccupé l'ensemble des mathématiciens, avec la découverte des paradoxes célèbres de Bertrand Russell et Burali Forti dans la théorie naïve des ensembles. Sans rentrer dans les détails, cela a donné lieu au développement d'une nouvelle théorie des ensembles, dite théorie des ensembles de Zermelo-Fraenkel ou ZF qui est la fondation des mathématiques la plus utilisée aujourd'hui et qui n'a donc pas de paradoxe connu.

Au dessus de cette théorie axiomatique des ensembles, il devait être possible de développer l'ensemble des mathématiques. La programme de Hilbert (1920) avait pour objectif de formaliser l'ensemble des mathématiques dans un système formel, une logique dont les raisonnements, les démonstrations ne pourraient être construites que par un certain nombre d'axiomes bien compris. Malheureusement, il n'existe pas de système cohérent utile, c'est-à-dire qui ne prouve pas l'absurde et permet de faire au moins du raisonnement arithmétique, dans lequel tout théorème ou son contraire serait prouvable. C'est le premier théorème d'incomplétude de Gödel. Le projet est donc mis à mal dans le sens où l'on ne peut pas trouver de formalisme "universel" qui permette sa réalisation.

Il est néanmoins possible de construire des systèmes logiques utiles et d'y travailler à la formalisation des mathématiques. C'est un des projets des logiciens et des mathématiciens (en particulier les constructivistes) depuis cette époque : construire des systèmes logiques suffisamment expressifs et relativement consistants (par rapport à ZF) pour formaliser des mathématiques. La distinction apportée par la logique intuitionniste, première représentante des logiques constructives, porte en premier lieu sur l'acceptation de l'axiome suivant dans la théorie des ensembles, appelé axiome du tiers-exclu :

$$\frac{}{A \vee \neg A}$$

Cet axiome dit en substance que tout énoncé est soit prouvable, soit contradictoire, mais ne donne aucun moyen effectif de décider dans quel cas nous nous trouvons. Il est

rejeté dans la logique intuitionniste, qui est donc distincte de la logique dite classique qui le comprend. En logique intuitionniste, on veut non seulement qu'une preuve nous assure de la validité d'un énoncé, mais aussi un moyen d'interpréter toute preuve en un moyen effectif de vérifier l'énoncé. Une preuve de $\exists\, x, x = 0$ doit donc nous fournir un naturel égal à 0, et une preuve de $A \vee B$ nous fournir une preuve de A ou une preuve de B. On voit bien que si l'on ajoute le tiers-exclu naïvement, il faudrait "inventer" une preuve d'une des branches qui n'apparaît pas dans la dérivation.

1.1.2 λ-calcul

Parallèlement à ce développement, Haskell Curry développe la logique combinatoire comme fondement des mathématiques (il fit sa thèse avec Hilbert) et Alonzo Church développe le λ-calcul, une notation pour étudier le calcul aujourd'hui à la base des langages fonctionnels. Le λ-calcul a un ensemble très restreint de constructions :

$$\Lambda, t, u := x \mid \lambda x, t \mid t\ u$$

On a trois constructeurs de termes différents : les variables x prises parmi un ensemble infini, l'abstraction $\lambda x, t$ permettant d'introduire une variable x dans un terme t (similaire à la notation d'une application $x \mapsto t$) et finalement l'application d'une fonction à un argument $f\ x$ notée par juxtaposition (plutôt qu'en utilisant des parenthèses $f(x)$). L'intérêt de ce formalisme réside dans sa règle de calcul unique appelée β-réduction :

$$(\lambda x, t)\ u \to_\beta t[u/x]$$

Ici $t[u/x]$ dénote la substitution de toutes les occurrences de la variable x dans t par u. Dans ce langage de programmation, on peut exprimer l'ensemble des fonctions récursives et cela donne donc un socle pour l'étude de la calculabilité (étude poursuivie en particulier par Kleene, étudiant de Church). Le problème est que ce système permet d'écrire des termes ayant peu de sens comme le célèbre $\Delta \triangleq \lambda x, x\ x$ ou dont l'évaluation ne termine pas comme $\Delta\ \Delta$ (le paradoxe de Kleene-Rosser). Pour remédier à ce problème Church introduit un système de *typage* qui permet de classifier et combiner les termes seulement lorsqu'ils ont un sens. L'algèbre des types est formée par des constantes c prises dans un ensemble infini (on dénote ces constantes par les métavariables α, β) et du constructeur de types flèches $\to$. La figure suivante donne les règles de ce calcul.

$$\textsc{Var}\ \frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha} \qquad \textsc{Abs}\ \frac{\Gamma, x : \alpha \vdash t : \beta}{\Gamma \vdash \lambda x, t : \alpha \to \beta} \qquad \textsc{App}\ \frac{\Gamma \vdash f : \alpha \to \beta \qquad \Gamma \vdash e : \alpha}{\Gamma \vdash f\ e : \beta}$$

Fig. 1.1: λ-calcul simplement typé ($\lambda^{\to}$)

Si on peut trouver un type à un certain terme du λ-calcul, alors on a la garantie que celui-ci termine (on ne peut pas appliquer infiniment la règle $\to_\beta$). Ce formalisme est en fait équivalent à la logique combinatoire de Curry, mais cette découverte sera assez tardive.

La logique combinatoire étudiée par Curry est aussi un formalisme d'étude du calcul, mais sa version typée est équivalente à un système permettant de faire des preuves intuitionnistes (un système de déduction à la Hilbert). Comme on l'a vu, la logique intuitionniste donne lieu à une interprétation des preuves comme des programmes réalisant leurs énoncés. C'est le principe sous-jacent à la réalisabilité étudiée par Kleene puis Kreisel et Troelstra. Cette technique permet effectivement d'extraire un algorithme à partir d'une preuve, qui donne donc le témoin dont on parlait précédemment (voir (Paulin-Mohring 2008) pour une introduction à la réalisabilité de ses débuts à son utilisation en Coq).

1.1.3 Des programmes et des preuves

Dans les années 70, la (re)découverte de la correspondance de Curry-Howard explicite la relation entre système logique et formalisme de calcul. L'idée que chaque preuve est un programme et que chaque type est une formule logique a eu un impact énorme sur un ensemble de domaines jusque la considérés comme séparés : la logique et en particulier la théorie de la démonstration et la théorie des types, la théorie de la calculabilité ou encore les méta-mathématiques. Depuis cette unification, on a étudié cette correspondance pour de nombreux formalismes, basés sur des logiques et des calculs originaux, avec des résultats fascinants : logique linéaire, calculs pour la logique classique...

La fondation sur laquelle Coq repose est une évolution de la Théorie des Types de Martin-Löf inventée à cette époque (Martin-Löf 1975). D'un point de vue logique, cette théorie correspond à une logique des prédicats d'ordre supérieur qui permet donc de quantifier sur les objets dans les énoncés. Le principe original sous-jacent est qu'on peut internaliser l'identification des preuves et des programmes. Techniquement, cette théorie utilise un quantificateur général appelé produit dépendant qui permet de quantifier aussi bien sur les objets que les types. Ainsi, on peut décrire l'énoncé suivant : $\Pi\ n\ m : \mathbb{N}, n + m = m + n$ qui énonce la commutativité de l'addition. Une preuve de ce théorème sera un objet de la forme $\lambda n\ m, p$ où p sera un objet-preuve de type $n + m = m + n$. Les types dépendants permettent d'utiliser des valeurs au niveau des types et donc d'exposer dans ceux-ci des énoncés non-triviaux. En comparaison, la théorie des types simples a un pouvoir de spécification très limité.

On montrera plus précisément en Coq comment cette idée est réalisée en pratique. Coq implémente une variante du Calcul des Constructions (Coquand et Huet 1988), une théorie des types qu'on peut voir comme le plus général des λ-calculs typés étudiés par Barendregt (Barendregt 1993). Il intègre une logique d'ordre supérieur polymorphe (i.e. on peut quantifier sur les types) et un langage de programmation avec types dépendants. Nous allons introduire le système qui étend ce formalisme avec des types inductifs primitifs dans la prochaine section.

1.2 Présentation de Coq

Coq est un assistant de preuve dont la première version date de 1985, et qui est aujourd'hui développé dans le projet PCRI TypiCal (anciennement LogiCal) commun à l'Institut National de Recherche en Informatique et Automatique (INRIA), au Laboratoire d'Informatique de l'X (LIX), au Laboratoire de Recherche en Informatique (LRI) et au Centre National de la Recherche Scientifique (CNRS). Originellement basé sur le Calcul des Constructions (CC) (Coquand et Huet 1988), il a été étendu au Calcul des Constructions (Co-)Inductives (CCI) (Paulin-Mohring 1993; Giménez 1996) et contient aujourd'hui de nombreuses améliorations telles qu'un système sophistiqué d'extraction de programmes (Letouzey 2004) ou encore des procédures de décision pour automatiser la preuve (Grégoire et Mahboubi 2005; Corbineau 2004).

Le développement de Coq est intimement lié à l'isomorphisme de Curry-Howard qui montre le lien entre logique intuitionniste et calcul. De cet isomorphisme, on peut déduire qu'élaborer une preuve du calcul propositionnel intuitionniste est équivalent à écrire un terme du λ-calcul simplement typé ($\lambda^{\rightarrow}$). Par exemple, montrer que $A \Rightarrow A$ pour un certain A revient à écrire la fonction identité $\lambda x : A.x$ qui a bien pour type $A \rightarrow A$. Chaque logique constructive est donc associée à un λ-calcul particulier. Dans Coq, on utilise cet isomorphisme pour vérifier les preuves. Le noyau est simplement un typeur pour CCI. Si on peut typer un terme t de type T, alors on est assuré d'avoir trouvé

une preuve constructive t de la formule T. Cette dualité se reflète aussi à l'utilisation de Coq où l'on a les deux visions : logique (développement mathématique, preuve) et calcul (développement informatique, programme).

Coq est utilisé le plus souvent pour élaborer des théories mathématiques prouvées mécaniquement (voir Coquand (2008) pour une revue récente). Dans cette optique, l'utilisateur modélise un problème par des structures mathématiques et veut prouver certaines propriétés sur ce modèle (par exemple la preuve du théorème des quatre couleurs (Gonthier et Werner 2005) utilisait des résultats de géométrie algébrique).

En général, on utilise uniquement le fragment simplement typé ou polymorphe du langage Gallina à la base de Coq pour écrire des programmes, et l'on utilise la richesse des types dépendants uniquement pour la spécification des propriétés dans un deuxième temps : c'est le modèle de vérification *a posteriori*. Le fragment du langage considéré correspond en fait au fragment purement fonctionnel de ML (sans effets de bords, non-terminaison incluse), dans une version au polymorphisme explicite. Il contient donc non seulement un λ-calcul typé mais aussi des types inductifs et coinductifs correspondants aux types algébriques. On va introduire sa syntaxe concrète par l'exemple.

Sur ce document Ce document a été rédigé avec LaTeX et produit par `pdflatex`. Les scripts Coq présentés sont tous vérifiés automatiquement avant de produire la documentation avec l'outil `coqdoc`. Dans la version électronique, les scripts et les références dans le texte sont hyperliés : les identificateurs définis font référence à leurs définitions dans le document ou dans la documentation de Coq disponible à l'adresse `http://coq.inria.fr`. Les conventions de couleurs et de fontes sont inspirées de celles utilisées par McBride :

- Les `mots-clés` sont en police "typewriter" rouge clair ;
- Les définitions sont en serif vert ;
- Les inductifs sont en script bleu ;
- Les constructeurs d'un type inductif sont en script bordeaux ;
- Les *variables* sont en italique magenta.

1.2.1 Types simples

On peut déclarer des types algébriques simples comme le type option suivant :

```
Inductive option (A : Type) :=
| None : option A
| Some : A → option A.
```

Comme son nom l'indique ce type permet de représenter les valeurs optionelles de type A. Un objet de type option A est soit vide (None) soit Some a où a est de type A.

On peut aussi déclarer des types inductifs récursifs (sous certaines restrictions qui garantissent la cohérence du système). Voici par exemple la déclaration du type des listes polymorphes en Coq :

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

Ce type inductif a deux constructeurs, nil pour la liste vide et cons a l pour la liste formée d'un élément a et d'une liste l.

Typiquement, on peut écrire en Coq la fonction suivante qui retourne le $n$ième élément d'une liste.

```
Fixpoint nth {A : Type} (l : list A) (n : nat) : option A :=
  match l with
```

```
  | nil ⇒ None A
  | cons a l' ⇒
    match n with
      | O ⇒ Some A a
      | S n ⇒ nth l' n
    end
end.
```

La fonction compare l'index n et la liste l. Si la liste est vide, on ne peut pas retourner l'élément souhaité et l'on renvoie None. Sinon, c'est qu'elle est de la forme cons a l'. Dans ce cas, on discrimine sur l'index n et l'on retourne soit le premier élément a soit le résultat de l'appel récursif nth l n.

1.2.2 Polymorphisme et arguments implicites

On voit ici apparaître une difficulté due au polymorphisme explicite : on doit indiquer pour None et Some l'instance du type polymorphe utilisé. C'est en effet un argument explicite du constructeur : None a pour type $\forall$ A : Type, option A. Heureusement, Coq utilise un mécanisme d'arguments implicites qui permet d'éviter d'indiquer ces informations lorsqu'elles sont inférables à partir du contexte. Ici le contexte consiste en la contrainte de typage option A qu'on a donnée dans la signature de la fonction, et qui est transportée dans chaque branche. Lorsqu'on type None, on a donc la contrainte de typage option A et il est facile de deviner que son premier argument doit être A, par unification.

On a ici un autre exemple d'utilisation de ce mécanisme pour éviter de répéter le type A de la liste polymorphe l lorsqu'on appelle nth. La déclaration de l'argument {A : Type} indique que A est un argument implicite[1] de la fonction. Cela veut donc dire qu'on doit pouvoir inférer la *valeur* de A à partir des autres arguments et de leurs types. Ici on voit bien que si les arguments explicites comprennent la liste l, on peut retrouver A en inspectant le type de l. Le système permet aussi d'inférer automatiquement les arguments implicites pour toute définition, mais dans ce document on se limitera à la déclaration explicite des arguments implicites la plupart du temps. Par exemple, pour la définition de list, le système serait capable d'inférer que le type A peut être rendu implicite pour le constructeur cons : $\forall$ (A : Type), $A \to$ list $A \to$ list A. On peut aussi l'indiquer explicitement après la définition :

```
Implicit Arguments nil [[A]].
Implicit Arguments cons [[A]].
Implicit Arguments None [[A]].
Implicit Arguments Some [[A]].
```

À l'aide des arguments implicites, on peut retrouver plus ou moins les mêmes facilités d'écriture que dans les langages avec inférence à la Hindley-Milner où les types n'apparaissent jamais explicitement dans les termes, si l'on donne suffisament d'information dans les signatures.

```
Definition id {A} (x : A) := x.
Definition compose {A B C} (g : B → C) (f : A → B) := λ x, g (f x).
Definition compose_ids : nat → option nat := compose Some id.
```

Dans ce dernier exemple, on voit bien que les deux applications partielles de Some et id sont respectivement de type ?B → option ?B et ?A → ?A et que le système parvient à unifier les variables ?A et ?B avec le type nat.

[1] En jargon Coq, maximal

Désormais, si l'on veut faire référence à la version explicite de id, on doit écrire :

```
Check (@id : ∀ A, A → A).
```

Le système supporte aussi un système de notations qu'on utilisera dans la suite. On peut par exemple utiliser le symbole infixe usuel pour la composition :

```
Infix "∘" := compose (at level 40, left associativity).
```

1.2.3 Preuves

Une fois qu'on a défini nos fonctions, on peut se lancer dans des preuves. Par exemple on peut montrer que la composition de fonctions est associative. Dans la suite, on utilise Π ou ∀ indifféremment pour dénoter le quantificateur universel (aussi appelé produit dépendant).

```
Lemma compose_assoc {A B C D} : Π (h : C → D) (g : B → C) (f : A → B),
  h ∘ g ∘ f = h ∘ (g ∘ f).
Proof. intros. reflexivity. Qed.
```

Le mode de preuve de Coq est basé sur un langage de tactiques à la LCF nommé $\mathcal{L}_{\text{tac}}$ (Delahaye 2000) qui peut être étendu directement dans le système. Les tactiques utilisées dans les scripts peuvent être arbitrairement complexes, de l'application de la réflexivité de l'égalité à des procédures de décision pour les anneaux ou bien encore des tactiques de réécriture complexes comme celle développée au chapitre 9. Le point important de ce système est que les tactiques sont des combinateurs qui créent des termes de preuve qui témoignent des manipulations logiques qu'elles réalisent dans les formules logiques (les types de ces termes). À la fin d'une preuve, on crée un terme combinant les résultats des tactiques individuelles et on vérifie que ce terme a bien pour type la spécification donnée par l'énoncé. Ainsi, même si une tactique contient un bug et crée des termes arbitraires, son résultat sera vérifié par le noyau qui signalera si une erreur a été commise. On ne peut donc pas mettre en danger la sûreté du système en développant des tactiques incorrectes.

Notons que pour cette preuve on a utilisé la quantification universelle pour lier les valeurs f, g et h. On a en fait construit une fonction dépendante prenant ces arguments formels et retournant une preuve de $h \circ g \circ f = h \circ (g \circ f)$.

Reprenons l'exemple des listes et essayons de faire une preuve sur celles-ci. On veut dire que si l'indice n recherché est bien parmi les éléments de la liste, alors on trouvera un élément associé. On a besoin d'une définition de la longueur d'une liste.

```
Fixpoint length {A} (l : list A) : nat :=
  match l with
    | nil ⇒ O
    | cons a l' ⇒ S (length l')
  end.

Lemma nth_correct {A} : Π (l : list A) n, length l > n → ∃ x, nth l n = Some x.
Proof. induction l; intros.
```

Pour montrer ce théorème, on applique une tactique d'induction sur la liste l. Arrêtons nous au premier sous-but :

```
A : Type
n : nat
H : length nil > n
```

```
============================
 ∃ x : A, nth nil n = Some x
```

Ici nous sommes dans un cas impossible, puisque length nil = O et donc l'hypothèse H est fausse. La tactique `inversion` nous permet d'éliminer ce sous-but.

```
  inversion H.
```

Dans le cas récursif, on examine n et on applique l'hypothèse d'induction le cas échéant.

```
A : Type
a : A
l : list A
IHl : Π n : nat, length l > n → ∃ x : A, nth l n = Some x
n : nat
H : length (cons a l) > n
============================
 ∃ x : A, nth (cons a l) n = Some x
```

```
  destruct n ; simpl in × ; [ (∃ a ; reflexivity) | apply IHl ; omega ].
Qed.
```

1.2.4 Spécifications fortes

On voit ici que l'on peut raisonner sur la propriété logique length $l > n$ et les objets inductifs (listes, naturels) de façon très agréable pour faire la preuve. Peut-on en faire autant dans les programmes ? Intuitivement, vu ce qu'on vient de montrer avec le lemme nth_correct, il devrait être possible d'écrire une nouvelle fonction nth qui prend en argument une preuve de length $l > n$ et renvoie toujours un objet de type A, le témoin de la preuve de nth_correct. Autrement dit, on devrait pouvoir écrire une fonction totale :

```
Definition nth_safe {A} (l : list A) (n : nat) : length l > n → A.
```

Malheureusement, écrire cette fonction requiert de manipuler explicitement la preuve de length $l > n$ et cela rend l'exercice très difficile. Le terme correspondant tient sur une page à peu près. Le problème est essentiellement qu'on doit faire des manipulations dans le terme de preuve qui correspondent seulement à des étapes de raisonnement et non pas de calcul ! Le programme qu'on veut obtenir à la fin devrait être le même que nth, excepté qu'on n'aurait pas à retourner None puisqu'on aurait une absurdité dans le cas où l = nil. On peut bien sûr écrire cette définition à l'aide de tactiques :

```
induction l ; intros ; [ (elimtype False ; inversion H) | destruct n ; simpl in × ].
```

Arrêtons nous un instant sur ce sous-but :

```
A : Type
a : A
l : list A
IHl : Π n : nat, length l > n → A
H : S (length l) > O
============================
```

```
        A
```

Pour indication, nous sommes dans la première branche de l'analyse de cas sur n, où la liste a déjà été décomposée en a et l. On peut remarquer que les scripts de preuve ne sont pas structurés en général, donc il est à peu près impossible de déterminer l'algorithme sous jacent. De plus, si l'on utilise des tactiques un peu complexes et en particulier des tactiques automatiques, il est difficile de déterminer leur effet sur le terme de preuve. Ici, si nous avions deux objets de type A dans l'environnement, une tactique comme `assumption` qui prend une hypothèse du type du but pour l'appliquer pourrait choisir l'une ou l'autre indifférement : ces tactiques sous-spécifient le programme.

```
  exact a.
  apply (IHl n); omega.
Qed.
```

On voudrait pouvoir écrire des programmes fortement spécifiés comme nth_safe en écrivant directement l'algorithme, comme pour les fonctions simplement spécifiées tout en permettant de faire les preuves à l'aide des tactiques habituelles de Coq.

En effet, il est reconnu aujourd'hui (McKinna 2006; Chlipala 2007; Oury et Swierstra 2008) que la programmation avec types dépendants a des bénéfices bien supérieurs à la solution qui utilise la vérification *a posteriori*. La possibilité de raffiner la forme des données et de refléter dans les types la sémantique des termes permet non seulement des optimisations mais surtout une approche de la programmation comme explicitation d'une solution à un problème très bénéfique pour la compréhension des programmes.

Pour cela, on va développer un nouveau langage dans lequel les preuves ne seront pas manipulables mais complètement implicites : lorsqu'une preuve est nécessaire en Coq à un point du programme, on n'aura pas besoin de la donner explicitement. Cependant, une obligation de preuve correspondante sera générée.

Type sous-ensemble

Pour mettre en œuvre ce mécanisme de façon transparente, on va s'appuyer sur une construction existante de Coq : le type sous-ensemble.

```
Inductive sig {A : Type} (P : A → Prop) : Type :=
  exist : ∀ x : A, P x → sig P.
Notation "{ x : A | P }" := (sig (fun x : A ⇒ P)).
```

Un type sous-ensemble est défini à partir d'un type A et d'un prédicat P sur ce type. Un élément de ce type est une paire dépendante d'un objet x de type A et une preuve $P\ x$. Informellement, $\{\ x : A \mid P\ \}$ est le type des objets de type A vérifiant P. Par exemple, on peut construire le type $\{\ x : \mathsf{nat} \mid x > \mathsf{O}\ \}$ des naturels supérieurs à O. Pour créer un objet de ce type, il faut fournir non seulement un naturel mais aussi une preuve qu'il est supérieur à O, de façon tout à fait identique au type $\exists\ x : \mathsf{nat},\ x > \mathsf{O}$ qu'on a vu précédemment : c'est le principe d'introduction des paires dépendantes dans une logique constructive. Une subtilité différencie cependant ces deux types : tandis que $\{\ x : A \mid P\ \}$ est de type Type, $\exists\ x : A,\ P$ est de type Prop.

1.2.5 La distinction Prop/Type

La distinction entre Prop et Type de Coq correspond à la distinction sémantique naturelle entre preuves et programmes. Dans le monde des preuves, la forme des termes est

indifférente, seule l'existence importe (aucun mathématicien classique n'a jamais *vu* une preuve, seulement des démonstrations d'existence !). On ne veut jamais distinguer deux preuves de $n > \mathsf{O}$ et faire des choix différents dans un algorithme suivant la forme de la preuve. Comme les termes Coq peuvent mélanger objets logiques et informatifs comme dans nth_safe, on interdit donc de travailler par cas sur un objet logique lorsqu'on définit un objet informatif, excepté dans certains cas bien précis où le filtrage n'a qu'un cas possible (pour l'égalité par exemple). Techniquement, on restreint les éliminations autorisées lors du filtrage.

Dans Coq, toutes les constructions logiques sont donc développées dans `Prop` (égalité, conjonction, disjonction, équivalence logique...). Dans la vision "*propositions-as-types*", la conjonction logique de deux propositions est la même chose que le produit cartésien de deux types. En Coq, on permet simplement de distinguer les deux parce qu'on ne veut pas nécessairement les traiter de la même façon. En particulier, le principe d'indifférence aux preuves qui dit que toutes preuves du même énoncé sont égales est cohérent avec ce traitement des objets propositionnels, même si ce n'est qu'un axiome dans Coq.

Pour les types dans `Type` en revanche, l'identité de deux objets est non-triviale. Il est en effet crucial de pouvoir distinguer deux naturels ou deux listes différentes.

Extraction

Cette distinction est non seulement utile pour catégoriser objets preuves et programmes, mais elle a un attrait tout particulier lorsqu'on veut extraire des programmes décrits en Coq. En effet, comme on a l'invariant que les objets informatifs définis en Coq ne dépendent pas de la forme des preuves, on peut tout simplement éliminer ces dernières lorsqu'on veut évaluer les programmes. L'extraction de Coq (Letouzey 2004) permet donc d'obtenir le programme suivant en OCaml à partir de nth_safe :

```
let rec nth_safe l n =
  match l with
    | Nil → assert false
    | Cons (y, l0) → (match n with
                        | O → y
                        | S n0 → nth_safe l0 n0)
```

On peut voir que l'argument de preuve de type length $l > n$ a été entièrement éliminé et que les cas impossibles sont devenus des `assert` *false*. On comprend bien que cette fonction n'est utilisable que si sa précondition logique est vérifiée.

À l'extraction, on élimine ainsi toutes les parties logiques pour ne garder que l'algorithme. Lorsqu'on a un objet de type sous-ensemble $\{ x : A \mid P \}$, on a une paire (a, p) ou a est un objet informatif de type A et p une preuve de $P\ a$. On peut donc construire des programmes sur de tels objets dans Coq, et à l'extraction les preuves portées par exist disparaîtront, ne laissant que les témoins. En revanche on ne peut pas écrire un programme sur un objet de type $\exists\ x : A, P$, puisque ce type a pour sorte `Prop` : le témoin de cette preuve ne peut pas être utilisé pour écrire un objet calculatoire. Cette distinction forte permet notamment d'utiliser des axiomes comme le tiers-exclu uniquement dans `Prop` sans mettre en danger la calculabilité des programmes.

1.3 Contributions

Pour résumer, on a vu que le système Coq permet de mélanger preuves (sorte `Prop`) et programmes (sorte `Type`) et que le système de tactiques n'était pas adapté à la program-

mation avec de telles spécifications fortes. On va se baser sur le type sous-ensemble qui permet d'associer des propriétés à des objets tout en les gardant cependant séparés des preuves pour créer un mode de développement de programmes plus agréable pour écrire des fonctions fortement spécifiées. Dans ce système, il devient possible d'écrire la fonction nth_safe comme espéré et de laisser les obligations être résolues par tactiques. La version extraite du code devient alors tout a fait similaire à la version originale.

```
Require Import Program.

Program Fixpoint nth_safe' {A : Type} (l : list A) (n : nat | length l > n) : A :=
  match l with
    | nil ⇒ !
    | cons a l' ⇒
      match n with
        | O ⇒ a
        | S n' ⇒ nth_safe' l' n'
      end
  end.

  Solve Obligations using program_simplify; simpl in ×; auto with ×.
```

L'idée ici est d'associer aux objets des propriétés arbitraires comme pour $(n : \mathsf{nat} \mid \mathsf{length}\ l > n)$ mais de continuer à utiliser l'objet n comme si ce n'était qu'un naturel dans le reste du programme. Dans le cas cons $a\ l'$, lorsqu'on fait un filtrage sur n, on agit comme si c'était simplement un naturel. En Coq classique, on devrait utiliser la projection :

```
Definition proj1_sig {A} {P : A → Prop} (s : sig P) : A :=
  match s with exist a p ⇒ a end.
```

pour pouvoir récupérer le naturel stocké dans la paire dépendante. De plus, lorsqu'on fait l'appel récursif nth_safe' $l\ n'$, il faudrait normalement donner une preuve de la forme exist $n'\ p'$ où $p' : \mathsf{length}\ l' > n'$ vu le type de nth_safe'. Finalement, en lieu et place de ! on devrait avoir une preuve du fait que le cas $l = \mathsf{nil}$ est impossible.

Toutes ces manipulations sont faites implicitements par notre outil Program. L'idée principale est que dans ce langage on peut utiliser un objet de type A là où l'on attend un objet de type $\{\ x : A \mid P\ \}$ pour n'importe quelle propriété et vice-versa. Bien sûr, il faudra montrer *a posteriori* que les objets ont bien les propriétés supposées (que n' est bien plus petit que la longueur de l' dans notre exemple). Mais si le raisonnement est fait *a posteriori*, l'algorithme quant à lui est décrit immédiatement.

1.3.1 "*Predicate Subtyping*"

Cette idée d'utiliser un système de sous-typage pour permettre d'utiliser les objets de type sous-ensemble de façon très flexible n'est pas nouvelle. En effet, c'est l'idée à la base du *Predicate subtyping* de PVS. L'assistant de preuve PVS (Owre et Shankar 1997) contient lui aussi un langage avec types dépendants utilisé pour spécifier et programmer et a une notion similaire de type sous-ensemble. L'idée du *Predicate subtyping* implémenté dans PVS (Shankar et Owre 1999; Rushby, Owre, et Shankar 1998) est de considérer tout objet de type T comme un objet de type $\{\ x : T \mid P\ \}$ inconditionnellement et vice-versa. Comme tout objet t de type T ne vérifie pas forcément la propriété P, on génère une "*Type-Checking Condition*" (TCC) au typage lorsqu'on injecte un objet dans un sous-ensemble. On demande ensuite à l'utilisateur de prouver $P[t/x]$ pour assurer que le programme est correct. En général, les TCCs sont déchargées automatiquement par des tactiques. On a donc une séparation entre la phase de codage qui génère des obligations et la phase de preuve qui permet de les décharger.

1.3.2 Coercions

Cependant, PVS n'a pas du tout la même architecture que Coq, et en particulier le langage à types dépendants n'est pas utilisé pour représenter les preuves et il n'y a donc pas de noyau bien défini permettant de vérifier ces preuves dans PVS. Il faut donc faire confiance à la quasi totalité du code pour croire en la correction des programmes vérifiés, ce qui ne satisfait pas le critère de De Bruijn. En Coq en revanche, on a une séparation claire entre le noyau et le reste du système, et le noyau lui même a été formellement vérifié (Barras 1996) (plus précisément, un modèle du noyau).

Dans notre cas, il faut donc générer des termes de preuve qui vont contenir des témoins de ce sous-typage. Une littérature importante (Chen 2003; Luo 1996) existe autour des systèmes à coercions explicites dont nous nous sommes inspirés pour réaliser la génération des termes. Dans un système à coercions explicites, on peut faire des abus de notations comme utiliser un objet de type T à la place d'un de type U, mais on applique une coercion qui amène l'objet vers le type U avant de retyper dans un système sans coercions. Généralement les coercions sont très similaires à des identités, c'est-à-dire qu'elles sont calculatoirement insignifiantes mais leur utilisation facilite le développement. Dans Coq par exemple le système de coercions (Saïbi 1997) a permis de développer des théories algébriques réutilisables sur plusieurs structures instantanément (un théorème sur les corps pouvant s'appliquer aux anneaux grâce à la coercion/projection évidente). Ce concept a été un outil essentiel dans le développement de la librairie CoRN qui développe les bases de l'analyse et l'algèbre constructives en Coq.

1.3.3 Architecture

On va donc procéder par élaboration des termes d'un calcul faisant la coercion implicitement entre T et $\{ x : T \mid P \}$ vers Coq. Cette fois, la coercion n'est pas une simple application de fonction puisqu'on peut avoir à prouver l'appartenance de l'objet coercé au sous-ensemble. Notre traduction va donc insérer des "trous" (techniquement, des variables existentielles) dans le terme Coq qui représenteront les endroits où l'on a fait des abus de notation. Si l'on peut remplir ces trous en donnant des termes de preuve correspondants, on obtient un terme Coq complet. La difficulté essentielle dans cette méhode va être de montrer la correction de cette traduction soit que tout terme typable dans le système étendu a une traduction typable en Coq. On verra que la traduction s'appuie implicitement sur le principe d'indifférence aux preuves.

Cette architecture d'élaboration d'un langage de surface plus souple vers un noyau aux contraintes très fortes respecte le critère de De Bruijn et est assez originale de ce point de vue. Seul le langage de programmation avec types dépendants Epigram est construit de façon similaire. Agda par exemple inclut un noyau de vérification pour une théorie des types mais sa construction de haut-niveau de filtrage dépendant fait partie intégrante du noyau du système : on perd en fiabilité et en simplicité en intégrant toujours plus de constructions au noyau. *A contrario*, l'ensemble des constructions décrites dans cette thèse se réduisent définitionnellement à des constructions plus simples du CC, dans certains cas étendu par le principe d'indifférence aux preuves. On peut utiliser ce langage noyau comme un assembleur pour construire un langage source riche sans sacrifier les propriétés fortes offertes par ce système.

1.3.4 Familles inductives

Cette idée de coercion des types sous-ensemble peut aussi s'étendre à une autre construction centrale de Coq : les familles inductives.

Une famille inductive est un type inductif indexé par un certain nombre de valeurs. L'exemple typique de famille inductive utilisée pour la programmation est la famille des vecteurs indexés par leur longueur.

```
Inductive vector {A : Type} : nat → Type :=
| vnil : vector O
| vcons : Π {n}, A → vector n → vector (S n).
Implicit Arguments vector [].
```

Dans cette déclaration, on a tout d'abord déclaré que vector était un type inductif paramétré par un type A. Le fait de rendre A implicite ici permet de ne pas le répéter dans les constructeurs, on le rend explicite en dehors de la définition. Le type vector est aussi indexé par un objet de type nat. Cela signifie qu'on peut exprimer le type vector A 4 par exemple et en particulier les constructeurs du type vector peuvent utiliser différentes instanciation de cet index. Ici, le vecteur vnil construit une instance de vector A O : c'est le vecteur de taille O. Quant à vcons, il prend comme arguments n, a et v' : vector A n pour construire un vecteur de taille $n + 1$. L'indice fait apparaître ici une information sur la taille du vecteur. On va maintenant pouvoir utiliser cette information directement dans les spécifications pour exprimer des propriétés sur la taille des vecteurs, sans avoir à la calculer explicitement à l'aide d'une fonction comme pour les listes. Typiquement, on peut maintenant écrire la fonction projetant le $n$ième élément d'un vecteur de la façon suivante :

```
Program Fixpoint nth {A m} (l : vector A m) (n : nat | n < m) : A :=
  match l with
    | vnil ⇒ !
    | vcons n' a l' ⇒
      match n with O ⇒ a | S n' ⇒ nth l' n' end
  end.
```

Filtrage dépendant et coercions

Les problèmes commencent lorsque l'on veut écrire des fonctions construisant des objets dans des familles inductives. Dans certains cas, tout se passe bien, par exemple on peut écrire la fonction de concaténation de deux vecteurs de la façon suivante :

```
Fixpoint app {A m n} (v : vector A m) (w : vector A n) : vector A (m + n) :=
  match v in vector _ m return vector A (m + n) with
    | vnil ⇒ w
    | vcons n' a v' ⇒ vcons a (app v' w)
  end.
```

Lorsqu'on fait du filtrage sur un type dépendant comme vector A m, on peut expliciter ce que le filtrage retourne en fonction de l'index de m. Intuitivement, on peut vouloir renvoyer des termes de types différents en fonction de m. Ici, c'est le cas : dans la première branche on retourne w : vector A n alors que le type attendu est vector A (O + n). Comme ces deux types sont convertibles, le typage réussit. Dans le deuxième cas on attend un objet de type vector A (S n' + n) qu'on obtient en construisant vcons a (app v' w) : vector A (S (n' + n)). En quelque sorte, le filtrage dépendant fait simultanément le filtrage sur l'objet et sur ses indices.

Malheureusement, il n'est pas toujours le cas que le type attendu et le type naturellement retourné dans une branche de filtrage soient convertibles. Par exemple considérons la fonction suivante :

```
Program Definition vtail {A n} (v : vector A (S n)) : vector A n :=
```

```
match v with
  | vnil ⇒ !
  | vcons n' a v' ⇒ v'
end.
```

Ici, dans la deuxième branche, on sait que v' est de type vector A n' alors que la constrainte de typage nous oblige à renvoyer un terme de type vector A n. Or n et n' ne sont pas convertibles, et on n'a même aucune information sur le lien entre n' et n. Il faut donc coercer v' vers le type vector A n et trouver l'information nécessaire pour montrer l'égalité de n et n'. Notre outil est capable de traiter ce problème en réfléchissant la construction de filtrage sur v au niveau logique et en introduisant dans chaque branche l'information nécessaire. Il s'agit encore une fois d'une *élaboration* du langage source vers un terme Coq décoré témoignant des manipulations logiques nécessaires.

Tout comme pour la projection d'un élément dans une liste, on pourra montrer ici dans la première branche que le point de programme est inaccessible puisqu'on ne peut pas filtrer v par vnil : cela impliquerait pour les indices que O est égal à S n.

On démontrera l'utilité de ces constructions par le développement d'une structure algorithmique complexe à l'aide de Program : les Finger Trees.

1.3.5 Classes de types

Pour améliorer encore la lisibilité et la flexibilité du langage de Coq, nous avons développé un système de classes de types inspiré de Haskell en Coq. Encore une fois, cette extension se réduit à des constructions existantes du langage et seule une amélioration des facilités données par le langage source Gallina a été nécessaire, en particulier dans la gestion des arguments implicites.

Les classes de types s'étendent aisément au cas dépendant. On peut ainsi indexer une classe par une valeur, ce qui permet d'associer des propriétés à des valeurs arbitraires très facilement, sans pour autant avoir à modifier leur type. Le système de classes de types permet aussi de faire de la programmation logique pendant le typage des termes, étendant l'algorithme d'unification par un algorithme de résolution similaire à Prolog. Finalement, il peut servir plus classiquement pour représenter des structures mathématiques abstraites et permettre d'utiliser la surcharge pour spécifier avec celles-ci. On démontrera l'utilité de cette extension pour le développement de tactiques automatisées utilisant la résolution des classes pour faire de la recherche de preuve. Le système de classes est en particulier lié au système de tactiques et leur intégration s'avère très profitable pour faire un pont entre développements utilisateurs et tactiques.

L'idée derrière cette architecture est d'une part que l'organisation d'un développement dans un assistant de preuve est primordiale et a besoin d'un certain nombre d'outils comme les classes, qui soient suffisamment souples et permettent d'automatiser une partie du travail de la preuve. D'autre part cette organisation peut être gérée en utilisant les fonctionnalités d'abstraction, de calcul et de structuration du langage à types dépendant lui-même avec une représentation de première classe, suivant l'exemple donné par McBride (2005).

1.4 Plan

Dans une première partie, nous allons développer le langage Russell qui intègre le "*Predicate Subtyping*" de PVS à une variante du Calcul des Constructions. On montrera chapitre 2 que ce système jouit de la propriété d'autoréduction et on présentera une preuve mécanisée de ce résultat (§ 2.2). On montrera aussi que le typage dans ce système est dé-

cidable section 2.3. Au chapitre 3 on présentera la traduction de Russell vers Coq et on démontrera sa correction. Finalement, on présentera au chapitre 4 Program, l'implémentation de ce système dans Coq.

Dans une seconde partie, on présentera un système de classes de types pour Coq, qui permet non seulement de gérer la surcharge dans le langage source mais aussi d'ajouter une forme de programmation logique au système. On présentera l'implémentation qui s'appuie sur les arguments implicites, les types inductifs et les tactiques de recherche de preuve dans le chapitre 7. On démontrera ensuite sur des exemples son utilité pour la programmation et la preuve sur des structures complexes (chapitre 8). Les classes de types s'intégrant au système de tactiques de Coq, on présentera une application de cette combinaison à travers une nouvelle tactique de réécriture généralisée chapitre 9.

Finalement, on utilisera l'ensemble de ces contributions dans la partie III, lors du développement d'une bibliothèque implémentant les Finger Trees (chapitre 11). On montrera comment cette structure peut être vérifiée complètement à l'aide de Program en utilisant types sous-ensemble, familles inductives et classes de types. Finalement, on montrera aussi une instanciation modulaire des Finger Trees qui permet d'obtenir des séquences à accès aléatoire certifiées.

Première partie

Russell

Chapitre 2

Le calcul de coercion par prédicats

Sommaire

Nous avons développé un langage supportant le *Predicate subtyping* utilisable dans Coq. L'utilisateur peut définir des programmes dans un langage souple puis prouver certains buts pour obtenir finalement un terme de CCI complet vérifiable par le noyau. Grâce à cette méthode, on peut manipuler les objets à type dépendant comme s'ils avaient des types simples et s'occuper des dépendances dans un deuxième temps, pour la preuve. L'architecture de notre système est la suivante : on type le programme dans notre langage Russell où l'on peut faire des abus de notations avec les objets de type sous-ensemble, puis l'on réécrit le terme typé dans CCI en laissant des "trous" dans les termes qui désambiguïsent les abus et une tactique se charge de générer les obligations correspondant à ces trous. Une fois les obligations résolues, on peut livrer le terme au noyau de Coq pour vérification.

On va donc tout d'abord présenter le langage Russell (section 2.1) et sa métathéorie (§2.2), puis un algorithme de typage correct et complet pour les programmes écrits en Russell (§2.3). On montrera dans le chapitre suivant comment plonger ce langage dans CCI en ajoutant les coercions adéquates et enfin on expliquera comment se déroule la génération des obligations de preuves à partir des termes engendrés par le plongement.

2.1 Le langage Russell

Nous voulons développer un langage très proche du fragment pur d'ML, plus les annotations nécessaires pour avoir un typage précis et décidable. On va procéder par élaboration du langage noyau de Coq en ajoutant des constructions suplémentaires pour obtenir un

langage similaire à ML. On peut voir le langage du noyau étudié ici comme une restriction de ML, purement fonctionnelle et sans filtrage. On n'a donc pas de types inductifs mais on considère des types existentiels (ou Σ-types) primitifs, qui généralisent les types des produits cartésiens de ML, formés par l'opérateur $*$. Les existentielles de la théorie des types permettent d'encoder par exemple les enregistrements dépendants.

2.1.1 Syntaxe

La syntaxe (figure 2.1) est directement inspirée des langages fonctionnels. Dans les systèmes basés sur la correspondance de Curry-Howard, on présente généralement la syntaxe en n'utilisant qu'une catégorie syntaxique. On en présente ici une version stratifiée entre termes et types, sachant que ces deux catégories seront fusionnées sous la dénomination de terme ensuite.

On part donc du λ-calcul (variables, abstraction et application) auquel on ajoute le constructeur de couples $(,)_{\Sigma x:\tau,\tau}$. C'est un système à la Church où les abstractions sont décorées par un type, tout comme les constructeurs des sommes dépendantes. C'est aussi un système au polymorphisme explicite, on a donc une production $(\alpha\ \tau)$ permettant d'appliquer un terme à un type. Enfin on a les deux projections des paires dépendantes.

Du côté des types, on a de même les variables de types, l'abstraction de type et l'application d'un constructeur de type à un type ou un terme. Le produit dépendant noté $\Pi x : \tau, \tau$ est une généralisation de l'espace fonctionnel noté $\tau \to \tau$. $\tau_1 \to \tau_2$ est donc une notation pour $\Pi x : \tau_1, \tau_2$ lorsque la variable x n'apparait pas dans le codomaine. De façon similaire, la somme dépendante notée $\Sigma x : \tau, \tau$ est une généralisation du produit cartésien noté $\tau \times \tau$. Le type sous-ensemble $\{x : \tau \mid \tau\}$ est une version distinguée du type somme (c'est aussi une somme dépendante) qui sera centrale dans la suite. On a finalement les sortes $\{\texttt{Set}, \texttt{Prop}, \texttt{Type}\}$ qui vont servir à typer (ou sorter) les types eux-mêmes.

$$
\begin{array}{rcl}
\alpha & ::= & x \\
 & \mid & \lambda x : \tau.\alpha \\
 & \mid & \alpha\ \alpha \\
 & \mid & \alpha\ \tau \\
 & \mid & (\alpha, \alpha)_{\Sigma x:\tau.\tau} \\
 & \mid & \pi_1\ \alpha \mid \pi_2\ \alpha
\end{array}
\qquad
\begin{array}{rcl}
\tau & ::= & x \\
 & \mid & \tau\ \tau \\
 & \mid & \tau\ \alpha \\
 & \mid & \lambda x : \tau.\tau \\
 & \mid & \Pi x : \tau.\tau \\
 & \mid & \Sigma x : \tau.\tau \\
 & \mid & \{\ x : \tau \mid \tau\ \} \\
 & \mid & \texttt{Set} \\
 & \mid & \texttt{Prop} \\
 & \mid & \texttt{Type}
\end{array}
$$

FIG. 2.1: Syntaxe

2.1.2 Sémantique

La sémantique (statique) du langage nous est donnée par un système de typage (figure 2.2 page 21). Le jugement de typage est défini inductivement par un ensemble de règles d'inférence. Dans notre cas ce sont les règles du Calcul des Constructions (CC) étendues avec les Σ-types auxquelles on a ajouté une règle de coercion (COERCE, figure 2.2) que l'on trouve classiquement dans les systèmes avec sous-typage sous le nom de subsumption.

Le jugement $\Gamma \vdash t : T$ se lit : dans l'environnement Γ, t est de type T. Les règles de typage sont standard. On peut créer des contextes bien formés à partir de types bien formés

$$\textsc{Wf-Empty}\ \frac{}{\vdash []} \qquad \textsc{Wf-Var}\ \frac{\Gamma \vdash A : s}{\vdash \Gamma, x : A}\ s \in \mathcal{S} \wedge x \notin \Gamma$$

$$\textsc{Var}\ \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\textsc{Axiom}\ \frac{\vdash \Gamma}{\Gamma \vdash s_1 : s_2}\ (s_1, s_2) \in \mathcal{A}$$

$$\textsc{Prod}\ \frac{\Gamma \vdash T : s_1 \qquad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash \Pi x : T.U : s_3}\ (s_1, s_2, s_3) \in \mathcal{R}$$

$$\textsc{Abs}\ \frac{\Gamma \vdash \Pi x : T.U : s \qquad \Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x : T.M : \Pi x : T.U}$$

$$\textsc{App}\ \frac{\Gamma \vdash f : \Pi x : V.W \qquad \Gamma \vdash u : V}{\Gamma \vdash (f u) : W[u/x]}$$

$$\textsc{Sum}\ \frac{\Gamma \vdash T : s \qquad \Gamma, x : T \vdash U : s}{\Gamma \vdash \Sigma x : T.U : s}\ s \in \{\texttt{Prop}, \texttt{Set}\}$$

$$\textsc{Pair}\ \frac{\Gamma \vdash \Sigma x : T.U : s \qquad \Gamma \vdash t : T \qquad \Gamma \vdash u : U[t/x]}{\Gamma \vdash (t, u)_{\Sigma x:T.U} : \Sigma x : T.U}$$

$$\textsc{Pi-1}\ \frac{\Gamma \vdash t : \Sigma x : T.U}{\Gamma \vdash \pi_1\ t : T} \qquad \textsc{Pi-2}\ \frac{\Gamma \vdash t : \Sigma x : T.U}{\Gamma \vdash \pi_2\ t : U[\pi_1\ t/x]}$$

$$\textsc{Subset}\ \frac{\Gamma \vdash U : \texttt{Set} \qquad \Gamma, x : U \vdash P : \texttt{Prop}}{\Gamma \vdash \{\ x : U \mid P\ \} : \texttt{Set}}$$

$$\textsc{Coerce}\ \frac{\Gamma \vdash t : U \qquad \Gamma \vdash U \rhd T : s}{\Gamma \vdash t : T}$$

Fig. 2.2: Calcul de coercion par prédicats - version déclarative

(sortés) (Wf-Empty, Wf-Var) et retrouver le type d'une variable si elle appartient au contexte (Var).

La règle Axiom décrit le sortage : la relation $\mathcal{A}$ est définie par $(\texttt{Prop}, \texttt{Type}) \in \mathcal{A} \wedge (\texttt{Set}, \texttt{Type}) \in \mathcal{A}$. C'est un calcul sans univers, au contraire de celui de Coq qui est basé sur ECC (Luo 1990).

La formation des produits dépendants (Prod) est controlée par la relation fonctionnelle $\mathcal{R}$ définie par les règles données figure 2.4.

On a un système proche du Calcul des Constructions mais avec Set prédicatif, comme dans les versions récentes de Coq. On n'a pas $(\texttt{Prop}, \texttt{Set}, \texttt{Set})$ dans notre relation $\mathcal{R}$ pour une bonne raison. Cela permet de créer des fonctions dépendant de propositions, par exemple $\Pi n : \mathsf{nat}, n > 0 \to \Pi l : \mathsf{list}\ A\ n \to A$. Or on veut éviter d'introduire des termes de preuve dans notre langage, et l'on voit que cette fonction pourrait naturellement s'écrire $\Pi n : \{\ n : \mathsf{nat} \mid n > 0\ \} \to \Pi l : \mathsf{list}\ A\ n \to A$ par curryfication. Encore une fois le type sous-ensemble nous permet d'éviter d'avoir à passer des termes de preuve directement.

Les règles d'abstraction et d'application (Abs, App) sont classiques.

Les sommes formables dans le système sont réduites aux couples d'objets de types de même sorte $s \in \{\texttt{Prop}, \texttt{Set}\}$ (Sum). Dans le premier cas les habitants sont les couples de preuves (codage du $\wedge$), dans le second ce sont les couples d'objets, soit les paires de ML. Intuitivement, c'est le type sous-ensemble $\{\ x : T \mid P\ \}$ qui permet de faire des couples

$$\rhd\text{-Conv}\ \frac{\Gamma \vdash T \equiv_{\beta\pi} U : s}{\Gamma \vdash T \rhd U : s} \qquad \rhd\text{-Trans}\ \frac{\Gamma \vdash S \rhd T : s \qquad \Gamma \vdash T \rhd U : s}{\Gamma \vdash S \rhd U : s}$$

$$\rhd\text{-Prod}\ \frac{\Gamma \vdash U \rhd T : s_1 \qquad \Gamma, x : U \vdash V \rhd W : s_2}{\Gamma \vdash \Pi x : T.V \rhd \Pi x : U.W : s_2}$$

$$\rhd\text{-Sum}\ \frac{\Gamma \vdash T \rhd U : s \qquad \Gamma, x : T \vdash V \rhd W : s}{\Gamma \vdash \Sigma x : T.V \rhd \Sigma y : U.W : s}\ s \in \{\texttt{Set}, \texttt{Prop}\}$$

$$\rhd\text{-Subset}\ \frac{\Gamma \vdash U \rhd V : \texttt{Set} \qquad \Gamma, x : U \vdash P : \texttt{Prop}}{\Gamma \vdash \{\ x : U \mid P\ \} \rhd V : \texttt{Set}}$$

$$\rhd\text{-Proof}\ \frac{\Gamma \vdash U \rhd V : \texttt{Set} \qquad \Gamma, x : V \vdash P : \texttt{Prop}}{\Gamma \vdash U \rhd \{\ x : V \mid P\ \} : \texttt{Set}}$$

Fig. 2.3: Coercion par prédicats - version déclarative

s_1	s_2	$s_3 = s_2$	Trait
Set	Set	Set	Produit dépendant
Type	Set	Set	Polymorphisme
Set	Type	Type	Constructeur de type dépendant
Set	Prop	Prop	Quantification universelle
Prop	Prop	Prop	Implication logique
Type	Prop	Prop	Quantification imprédicative dans Prop

Fig. 2.4: Définition de $\mathcal{R}$

Set, Prop habitant Set. Les types $\Sigma x : U.V$ où U : Prop et V : Set n'ont pas d'intérêt dans notre cas puisqu'ils représentent des objets de type $U \wedge V$ isomorphes à $\{\ \_ : V \mid U\ \}$.

Il est à noter qu'à l'introduction des paires dépendantes (Pair), on annote l'objet construit par son type. C'est le seul moyen d'assurer l'unicité du typage en présence de sommes dépendantes : il est en effet impossible d'inférer un type le plus général pour (t, u) avec t et u arbitraires, on le force donc dans le terme. Les éliminateurs (Pi-1, Pi-2) n'ont en revanche pas besoin d'annotations.

Vient finalement la règle Subset de formation des types sous-ensemble dont la première composante est un objet de sorte Set et la seconde une preuve. On ne donne aucune règle d'introduction ni d'élimination pour les objets de type sous-ensemble. C'est la coercion qui permettra de promouvoir un objet dans un sous-ensemble ou de le projeter d'un sous-ensemble vers son support implicitement.

Coercion, Conversion

La règle Coerce formalise l'idée que l'on peut utiliser un terme de type T à la place d'un terme de type U si T et U sont dans une certaine relation notée $\Gamma \vdash T \rhd U : s$. C'est là qu'interviennent les types sous-ensemble. CC contient une règle de typage similaire à Coerce, la règle de conversion (Conv), qui dit essentiellement que deux types convertibles (on rappelle que l'on peut calculer dans les types puisqu'on a l'abstraction, l'application, etc...) sont équivalents. La conversion utilisée ici est $\equiv_{\beta\pi}$, soit la β-réduction classique ainsi que les projections π_i pour les paires, dans une présentation inductive

("*judgemental*", tout comme dans les premières versions du CC (Coquand et Huet 1988)) et typée. On définit la relation de conversion entre types par un ensemble de règles d'inférence (vision sémantique), plutôt que *via* la conversion syntaxique.

Définition 2.1.1 (Égalité de types)**.** *L'égalité de types* $\Gamma \vdash T \equiv_{\beta\pi} U : s$ *est définie par la clôture réflexive, symétrique et transitive du jugement* $\Gamma \vdash T = U : s$ *dont les règles sont éoncées figure 2.5.*

L'égalité de types formalise la congruence de la réduction à travers toutes les constructions du langage. On a de plus une règle ConvJeq qui formalise le fait que deux termes peuvent être comparés sur n'importe quel type auquel ils sont coercibles. Cela rend l'ensemble des jugements de typage, coercion et égalité mutuellement récursifs. Cette présentation est à comparer à la version non typée utilisée dans la présentation usuelle des Pure Type Systems (PTS) :

$$\textsc{Conv}\ \frac{\Gamma \vdash t : U \qquad \Gamma \vdash T : s \qquad U \equiv_{\beta\pi} T}{\Gamma \vdash t : T}$$

La version typée est plus adaptée pour manipuler et faire des preuves sur les dérivations de typage comme la traduction que l'on présentera au chapitre 3. Elle pose cependant un problème important pour la preuve d'autoréduction que nous discuterons section 2.2, au même titre que la coercion, à cause de la règle de transititivé. En revanche, la version basée sur la réduction a un caractère très opérationnel et s'avère plus difficile à traiter dans les preuves sémantiques. Par exemple, il n'est pas évident que les chemins de conversion entre deux termes ne contiennent pas de termes non typables, surtout tant qu'on n'a pas montré l'autoréduction.

Jugement de coercion

Notre système de coercion par prédicats permet à l'utilisateur d'utiliser une valeur de type U là où l'on attend une valeur de type $\{\ x : V \mid P\ \}$ ($\rhd$-Proof) si U est lui-même coercible en V. A l'inverse, on permet aussi d'utiliser une valeur de type $\{\ x : U \mid P\ \}$ ($\rhd$-Subset) à la place d'une valeur de type V si U est coercible vers V.

Les règles $\rhd$-Prod et $\rhd$-Sum permettent de faire des coercions dans les types composites. Classiquement, la règle pour le produit fonctionnel est contravariante à gauche et covariante à droite (une fonction sous-type d'une autre accepte plus d'entrées mais donne une sortie plus fine (Castagna 1995)) et la règle pour le produit cartésien covariante sur les deux composantes (une paire est coercible en une autre si leurs composantes sont coercibles deux-à-deux). Le sens des coercions n'a pas d'importance dans le système déclaratif puisqu'il est symétrique mais il est essentiel lors de la création des coercions que nous décrirons plus tard.

La symétrie du système suggère qu'on laisse beaucoup de liberté à l'utilisateur au moment du typage. Par exemple on peut dériver dans ce système $u : \mathsf{nat} \vdash u : \{\ x : \mathsf{nat} \mid \bot\ \}$. Seulement, lors de la traduction de la dérivation de coercion $\mathsf{nat} \rhd \{\ x : \mathsf{nat} \mid \bot\ \}$ (nécessaire pour traduire l'abus de notation $x : \{\ x : \mathsf{nat} \mid \bot\ \}$), l'utilisateur aura à résoudre une obligation de preuve de $\bot$. On reposera donc toujours sur la cohérence du Calcul des Constructions.

La règle $\rhd$-Trans assure que l'on a un système compositionnel. Il y a ici une analogie avec l'élimination des coupures dans les systèmes logiques, où l'on montre que toute dérivation utilisant la règle de *modus ponens* ($A \Rightarrow B$ et $B \Rightarrow C$ implique $A \Rightarrow C$) peut se réécrire en une dérivation ne l'utilisant jamais. Dans les systèmes à sous-typage, on montre de façon équivalente que l'on peut éliminer la règle de transitivité ; première étape vers un système décidable.

$$\textsc{VarJeq}\ \frac{x : X \in \Gamma}{\Gamma \vdash x = x : X}$$

$$\textsc{SortJeq}\ \frac{}{\Gamma \vdash s = s : \texttt{Type}}\ s \in \{\texttt{Set}, \texttt{Prop}\}$$

$$\textsc{ProdJeq}\ \frac{\Gamma \vdash A = A' : s_1 \qquad \Gamma, x : A \vdash B = B' : s_2}{\Gamma \vdash \Pi x : A.B = \Pi x : A'.B' : s_3}\ (s_1, s_2, s_3) \in \mathcal{R}$$

$$\textsc{LambdaJeq}\ \frac{\Gamma \vdash A = A' : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad \Gamma, x : A \vdash M = M' : B}{\Gamma \vdash \lambda x : A.M = \lambda x : A'.M' : \Pi x : A.B}\ (s_1, s_2, s_2) \in \mathcal{R}$$

$$\textsc{AppJeq}\ \frac{\Gamma \vdash M = M' : \Pi x : A.B \qquad \Gamma \vdash N = N' : A}{\Gamma \vdash MN = M'N' : B[N/x]}$$

$$\textsc{BetaJeq}\ \frac{\begin{array}{c}\Gamma \vdash A : s_1 \\ \Gamma, x : A \vdash B : s_2 \qquad \Gamma, x : A \vdash v : B \qquad \Gamma \vdash e : A\end{array}}{\Gamma \vdash (\lambda x : A.v)\ e = v[e/x] : B[e/x]}\ (s_1, s_2, s_2) \in \mathcal{R}$$

$$\textsc{SigmaJeq}\ \frac{\Gamma \vdash A = A' : s \qquad \Gamma, x : A \vdash B = B' : s}{\Gamma \vdash \Sigma x : A.B = \Sigma x : C.D : s}\ s \in \{\texttt{Set}, \texttt{Prop}\}$$

$$\textsc{PairJeq}\ \frac{\begin{array}{c}\Gamma \vdash A = A' : s \\ \Gamma, x : A \vdash B = B' : s \qquad \Gamma \vdash e_1 = e_1' : A \qquad \Gamma \vdash e_2 = e_2' : B[u/x]\end{array}}{\Gamma \vdash (e_1, e_2)_{\Sigma x:A.B} = (e_1', e_2')_{\Sigma x:A'.B'} : \Sigma x : A.B}\ s \in \{\texttt{Set}, \texttt{Prop}\}$$

$$\textsc{PiLeftJeq}\ \frac{\Gamma \vdash t = t' : \Sigma x : A.B}{\Gamma \vdash \pi_1\ t = \pi_1\ t' : A}$$

$$\textsc{PiLeftRedJeq}\ \frac{\begin{array}{c}\Gamma \vdash A : s \\ \Gamma, x : A \vdash B : s \qquad \Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B[e/x]\end{array}}{\Gamma \vdash \pi_1\ (e_1, e_2)_{\Sigma x:A.B} = e_1 : A}\ s \in \{\texttt{Set}, \texttt{Prop}\}$$

$$\textsc{PiRightJeq}\ \frac{\Gamma \vdash t = t' : \Sigma x : A.B}{\Gamma \vdash \pi_2\ t = \pi_2\ t' : B[\pi_1\ t/x]}$$

$$\textsc{PiRightRedJeq}\ \frac{\begin{array}{c}\Gamma \vdash A : s \\ \Gamma, x : A \vdash B : s \qquad \Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B[e/x]\end{array}}{\Gamma \vdash \pi_2\ (e_1, e_2)_{\Sigma x:A.B} = e_2 : B[e_1/x]}\ s \in \{\texttt{Set}, \texttt{Prop}\}$$

$$\textsc{SubsetJeq}\ \frac{\Gamma \vdash A = B : \texttt{Set} \qquad \Gamma, x : A \vdash P = P' : \texttt{Prop}}{\Gamma \vdash \{\ x : A \mid P\ \} = \{\ x : B \mid P'\ \} : \texttt{Set}}$$

$$\textsc{ConvJeq}\ \frac{\Gamma \vdash M = N : A \qquad \Gamma \vdash A \rhd B : s}{\Gamma \vdash M = N : B}$$

Fig. 2.5: Jugement d'égalité

Notre jugement de coercion identifie les types U et $\{\ x : U \mid P\ \}$ mais notre système de typage ne permet pas d'éliminer (prendre la partie preuve) ou d'introduire (créer un couple témoin,preuve) des objets de type sous-ensemble. Cela nous assure une certaine cohérence, puisque même si l'on ne vérifie pas qu'un objet de type U a bien la propriété P, on ne peut pas raisonner sur le fait que U a ladite propriété dans le langage.

2.2 Métathéorie

Nous allons maintenant étudier la métathéorie de ce système et en particulier s'intéresser à la preuve d'autoréduction ou Subject Reduction (SR) du système déclaratif :

$$\forall\ \Gamma\ t\ T,\ \Gamma \vdash t : T \Rightarrow \forall\ t',\ t \rightarrow_{\beta\pi} t' \Rightarrow \Gamma \vdash t' : T$$

On montrera à la fois ce résultat et l'équivalence de la version déclarative et d'une version plus algorithmique du système.

La preuve d'autoréduction (SR) que nous allons présenter ne dépend pas de la normalisation et a été développée et mécanisée entièrement dans Coq (Sozeau 2006). Cette preuve est rendue difficile par la transitivité de la coercion et de l'égalité. Ce problème a déjà été identifié par Geuvers et Werner (1994) qui montrent l'équivalence des présentations déclaratives (sémantique) et algorithmiques (syntaxique) des PTS (avec η) mais s'appuient sur la preuve de SR du système syntaxique. Adams (2006) a résolu ce problème dans le cadre des PTS fonctionnels. Il propose en effet une solution pour montrer l'équivalence entre la présentation des PTS fonctionnels avec une équivalence définitionnelle ou avec un jugement d'égalité sans s'appuyer ni sur la normalisation ni l'autoréduction, mais seulement sur la confluence de la réduction. Nous allons étendre ce résultat pour Russell, en ajoutant les deux difficultés que représentent la relation de coercion et le traitement des sommes dépendantes. Finalement, notre preuve a été vérifiée mécaniquement.

Le reste de cette section s'organise de la façon suivante : Dans un premier temps, nous allons exposer la difficulté qu'on rencontre lorsque l'on veut montrer la propriété d'autoréduction d'un système avec un jugement d'égalité (§2.2.1), puis nous présenterons une vue d'ensemble de la preuve en faisant référence aux lemmes prouvés en Coq (§2.2.2), enfin on détaillera les parties intéressantes de cette formalisation (§2.2.3).

2.2.1 Jugement d'égalité, conversion et autoréduction

Il n'est pas possible de prouver directement SR dans notre système, à cause de la relation d'équivalence enrichie ainsi que l'égalité de jugement utilisées. En effet, la conversion typée nous empêche d'utiliser la même technique de preuve que celle du CC. Regardons où cela bloque. Dans le cas de l'application, lorsqu'on considère une réduction β à la racine on a :

$$\frac{\Gamma \vdash (\lambda x : A.M) : \Pi x : B.C \qquad \Gamma \vdash N : B}{\Gamma \vdash (\lambda x : A.M)\ N : C[N/x]}$$

Par induction on a : $\forall t', (\lambda x : A.M) \rightarrow_{\beta\pi} t' \Rightarrow \Gamma \vdash t' : \Pi x : B.C$ et $\forall N', N \rightarrow_{\beta\pi} N' \Rightarrow \Gamma \vdash N' : B$. On veut montrer $\Gamma \vdash M[N/x] : C[N/x]$.

Par inversion sur le jugement de l'abstraction, on a qu'il existe D et $(s, s') \in \mathcal{A}$ tels que :

$$\begin{aligned} \Gamma &\vdash A : s \\ \Gamma, x : A &\vdash D : s' \\ \Gamma, x : A &\vdash M : D \end{aligned}$$

avec $\Pi x : B.C \equiv_{\beta\pi} \Pi x : A.D$ (ou $\Gamma \vdash \Pi x : A.B \equiv_{\beta\pi} \Pi x : C.D : s$ dans la version avec jugement). Par la propriété de Church-Rosser (CR) pour $\equiv_{\beta\pi}$ on obtient $A \equiv_{\beta\pi} B$ et $C \equiv_{\beta\pi} D$. On a donc $\Gamma \vdash N : A$ et $\Gamma, x : A \vdash M : C$ par conversion (Conv). Par substitution on peut donc dériver $\Gamma \vdash M[N/x] : C[N/x]$.

Injectivité des produits Dans la version typée, on ne peut pas utiliser la propriété de CR qui ne vaut que pour la relation de réduction. Nous devons trouver un autre moyen de prouver l'injectivité du produit (et des sommes) dépendants :

$$\Gamma \vdash \Pi x : A.B \equiv_{\beta\pi} \Pi x : C.D : s' \Rightarrow \exists s, \Gamma \vdash A \equiv_{\beta\pi} C : s \wedge \Gamma, x : A \vdash B \equiv_{\beta\pi} D : s \wedge (s, s') \in \mathcal{A}$$

Or ce lemme ne semble pas prouvable puisqu'on peut utiliser la règle de transitivité dans la prémisse pour dériver l'égalité : $\Gamma \vdash \Pi x : A.B \equiv_{\beta\pi} X_0 \wedge \ldots \wedge X_n \equiv_{\beta\pi} \Pi x : C.D : s$

Dans ce cas, on ne pourra rien dire sur les X_i, même si l'on sait qu'ils réduisent vers des produits, sans SR pour nous assurer que cela préserve le type. Si l'on considère la relation de coercion plutôt que l'égalité, on n'aurait même pas cette propriété.

On a donc besoin de développer un nouveau système, équivalent à celui-ci mais dans lequel on peut prouver SR. Ce système repose sur la notion de réduction parallèle typée ("*typed parrallel one-step reduction*". (TPOSR)) sur des termes étiquetés, qui génère l'égalité de type sur les termes originaux. En ajoutant ces étiquettes et en mettant la relation de réduction au centre du système, on peut prouver une variante de Church-Rosser typée et éliminer le problème de transitivité. On peut dès lors prouver l'unicité du typage et l'injectivité des produits et des sommes dépendants. Prouver l'autoréduction devient immédiat et il n'y a plus qu'à montrer les équivalences entre systèmes pour transporter ce résultat.

2.2.2 Une preuve détournée d'autoréduction

Cette preuve a été développée entièrement en Coq, nous allons donc nous référer au développement disponible sur le web pour les preuves (Sozeau 2006). On présentera le développement plus en détail section 2.2.3.

Le développement du système TPOSR est dû à Robin Adams. Intuitivement, on recherche une relation $\Gamma \vdash M \to N : A$ ayant les propriétés suivantes :

- La propriété du diamant : si $\Gamma \vdash M \to N : A$ et $\Gamma \vdash M \to P : A$ alors il existe Q tel que $\Gamma \vdash N \to Q : A$ et $\Gamma \vdash P \to Q : A$.
- Elle génère l'égalité de types du système : on note $\Gamma \vdash M =_\to N : A$ la clôture symétrique et transitive de $\Gamma \vdash M \to N : A$. Cette relation $\Gamma \vdash M =_\to N : A$ représente la conversion dans ce système (on commencera par montrer sa réflexivité).
- Elle respecte la formation des produits et des sommes : Si $\Gamma \vdash \Pi x : A.B \to C : s$ alors il existe A', B' tels que $\Gamma \vdash A \to A' : s_1$ et $\Gamma, x : A' \vdash B \to B' : s_2$ avec $C = \Pi x : A'.B'$.
- Elle satisfait l'unicité du typage : si $\Gamma \vdash M \to N : A$ et $\Gamma \vdash M \to P : B$ alors A et B sont la même sorte ou $\Gamma \vdash A \simeq B : s$, où $\simeq$ représente la clôture réflexive, symétrique et transitive de $\to$.

Une telle relation peut être construite sur des termes étiquetés : on indique le codomaine sur les noeuds application et on indique le type complet sur lequel s'appliquent les projections (TPOSR.Terms). On écrit donc $\mathrm{app}_{(x)E}(MN)$ pour l'application MN avec l'idée que le codomaine de M est équivalent (pour nous, coercible) à E. On annote de même les projections : $\pi_1\ M$ devient $\pi_{1_{\Sigma x:A.B}}\ M$. La réduction associée à ces termes est l'extension de la réduction sur les termes par la réduction sur les étiquettes. On ne force pas les étiquettes à correspondre lorsqu'on a un radical, en fait on les ignore complètement :

$$\pi_{1_\tau}\ (M, N)_{\tau'} \to_{\pi_1} M$$

De cette façon, on conserve la linéarité à gauche des règles de réduction tout en conservant les étiquettes, qui seront de toute façon coercibles par typage. Contrairement à ce qui était

annoncé dans l'article d'Adams, cela ne pose pas de problème particulier. On peut déjà montrer que cette réduction jouit aussi de la propriété de Church-Rosser (TPOSR.Conv).

On appelle TPOSR la relation $\Gamma \vdash M \to N : A$ qui combine les règles de typage du système avec la réduction parallèle. Elle est mutuellement récursive avec la relation de coercion $\Gamma \vdash M \rhd_\to N : s$ et la relation d'égalité $\Gamma \vdash M =_\to N : A$ (TPOSR.Types). Cette dernière est par définition la clôture symétrique et transitive de TPOSR. La relation TPOSR est une simple transposition de la relation de typage de Russell avec une règle supplémentaire par radical. Dans les règles impliquant des étiquettes, on permet la coercion entre étiquettes. En effet, on peut toujours voir une application ayant pour type de retour B comme un objet de type B' si ces deux types sont coercibles. Les règles pour l'application deviennent donc :

$$\text{TPOSR-APP}\ \frac{\Gamma \vdash A \to A' : s_1 \qquad \Gamma, x : A \vdash B \rhd_\to B' : s_2 \qquad \Gamma \vdash M \to M' : \Pi x : A.B \qquad \Gamma \vdash N \to N' : A}{\Gamma \vdash \mathrm{app}_B(M\ N) \to \mathrm{app}_{B'}(M'\ N') : B[N/x]}$$

$$\text{TPOSR-}\beta\ \frac{\Gamma \vdash A \to A' : s_1 \qquad \Gamma, x : A \vdash B \to B' : s_2 \qquad \Gamma, x : A \vdash M \to M' : B \qquad \Gamma \vdash N \to N' : A}{\Gamma \vdash \mathrm{app}_B((\lambda x : A, M)\ N) \to M'[N'/x] : B[N/x]}$$

On peut remarquer l'asymétrie inhérente à ce formalisme, où l'on utilise seulement les termes à gauche de la flèche pour former le type de la conclusion. Une bonne partie du travail va consister à montrer que la partie droite est elle aussi typable avec ce même type, une forme de preuve d'autoréduction anticipée.

La définition de la coercion est inchangée, on ajoute simplement des conditions de sortage dans les règles qui facilitent les preuves mais qu'on peut montrer dérivables (TPOSR.Derivable).

Le système TPOSR

On commence par prouver la métathéorie basique de ce système, qui inclut l'affaiblissement (TPOSR.Thinning), la substitution de termes (TPOSR.Substitution) et la substitution des dérivations de TPOSR (TPOSR.SubstitutionTPOSR). La plupart des preuves sont de simples inductions mutuelles sur les dérivations de typage, bonne formation des contextes, équivalence et coercion. En revanche, il faut faire très attention à l'enchaînement des lemmes. TPOSR est une relation asymétrique et il faut donc commencer par faire les preuves s'intéressant seulement à la gauche de la flèche avant de pouvoir attaquer la droite. On dénote par $\Gamma \vdash J$ l'un des quatre jugements et $\mathcal{R}$ pour chacune des relations de $\to$ (typage), $=_\to$ (égalité) et $\rhd_\to$ (coercion).

On peut compléter la preuve dans l'ordre suivant :

1. Tout d'abord, on prouve la réflexivité à gauche (TPOSR.LeftReflexivity) : Si $\Gamma \vdash t \to t' : T$ alors $\Gamma \vdash t \to t : T$.
2. Ensuite, une forme préliminaire de la coercion de contexte (TPOSR.PreCtxCoercion) : Si $\Gamma, x : A, \Delta \vdash J$ et $\Gamma \vdash A \rhd B : s$, $\Gamma \vdash A \to A : s$ et $\Gamma \vdash B \to B : s$ alors $\Gamma, x : B, \Delta \vdash J$. La coercion de contexte inclut la conversion de contextes.
3. Puis la substitution : si $\Gamma, x : U, \Delta \vdash J$ and $\Gamma \vdash u \to u : U$ alors $\Gamma, \Delta[u/x] \vdash J[u/x]$ (TPOSR.PreSubstitution).
4. On généralise à une forme préliminaire de la substitution des dérivations de TPOSR (où les deux termes peuvent différer) (TPOSR.PreSubstitutionTPOSR) : Si $\Gamma, x : U, \Delta \vdash t\ \mathcal{R}\ t' : T$ et $\Gamma \vdash u \to u' : U$ avec $\Gamma \vdash u' \to u' : U$ alors $\Gamma, \Delta[u/x] \vdash t[u/x]\ \mathcal{R}\ t'[u'/x] : T[u/x]$.

5. Finalement on peut prouver la réflexivité à droite (TPOSR.RightReflexivity) : Si $\Gamma \vdash t \mathrel{\mathcal{R}} t' : T$ alors $\Gamma \vdash t' \mathrel{\mathcal{R}} t' : T$.

Après avoir prouvé la réflexivité à droite, on peut enlever les conditions de bord à la substitution (TPOSR.Substitution) ("avec $\Gamma \vdash u' \to u' : U$") et la coercion de contextes (TPOSR.CtxCoercion).

Ceci fait, on peut prouver des lemmes de génération pour le jugement TPOSR (TPOSR.Generation). On les utilise pour montrer la validité : si $\Gamma \vdash t \to t : T$ alors $T = s$ pour $s \in \mathcal{S}$ ou il existe $s \in \mathcal{S}$ telle que $G \vdash T \to T : s$ (TPOSR.Validity). Une fois la validité prouvée, il suffit d'avoir la fonctionnalité des types pour obtenir leur unicité. La fonctionnalité dit simplement que deux types aux composantes équivalentes sont équivalents, par exemple pour les produits dépendants : si $\Gamma \vdash A \simeq B : s1$ et $\Gamma, x : A \vdash C \simeq D : s2$ alors $\Gamma \vdash \Pi x : A.C \simeq \Pi x : B.D : s2$ (TPOSR.TypesFunctionality).

On peut finalement prouver l'unicité du typage, modulo coercion : si $\Gamma \vdash t \to ? : T$ et $\Gamma \vdash t \to ? : U$ alors $T = U = \texttt{Type}$ ou il existe une sorte s telle que $\Gamma \vdash T \rhd_{\to} U : s$. C'est pour cette preuve que les étiquettes sont nécessaires. En effet au cas de l'application dans le système original on a les dérivations suivantes :

$$\frac{\Gamma \vdash M : \Pi x : A.C \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : C[N/x]}$$

$$\frac{\Gamma \vdash M : \Pi x : B.D \qquad \Gamma \vdash N : B}{\Gamma \vdash M\ N : D[N/x]}$$

Par hypothèse d'induction on a $\Gamma \vdash \Pi x : A.C \rhd \Pi x : B.D : s$ pour un s donné. Seulement, on ne peut pas encore déduire que cela implique $\Gamma, x : B \vdash C \rhd D : s$, puisque ça requiert l'injectivité des produits que nous ne pouvons pas prouver, comme pour la preuve d'autoréduction. Si nous ajoutons les étiquettes dans le système TPOSR on obtient :

$$\frac{\Gamma \vdash M : \Pi x : A.C \qquad \Gamma \vdash N : A}{\Gamma \vdash \mathrm{app}_{(x)E}(M\ N) \to ? : C[N/x]}$$

$$\frac{\Gamma \vdash M : \Pi x : B.D \qquad \Gamma \vdash N : B}{\Gamma \vdash \mathrm{app}_{(x)E}(M\ N) \to ? : D[N/x]}$$

Le lemme de génération de l'application nous donne les hypothèses supplémentaires $\Gamma \vdash E[N/x] \rhd C[N/x] : s$ et $\Gamma \vdash E[N/x] \rhd D[N/x]$. On peut donc conclure la preuve par transitivité de la coercion. La même technique permet de traiter les projections.

Une fois l'unicité prouvée on peut faire la preuve de Church-Rosser pour TPOSR. On montre d'abord la confluence : si $\Gamma \vdash t \to u : U$ et $\Gamma \vdash t \to v : V$ alors il existe t' tel que :

$$\Gamma \vdash u \to t' : U$$
$$\Gamma \vdash u \to t' : V$$
$$\Gamma \vdash v \to t' : U$$
$$\Gamma \vdash v \to t' : V$$

Cette preuve est par induction sur la *taille* de la première dérivation de TPOSR (TPOSR.ChurchRosserDepth).

En corollaire, on obtient (TPOSR.ChurchRosser) : Si $\Gamma \vdash t \simeq u : s$ alors il existe x tel que $\Gamma \vdash t \to^+_{\beta\pi} x : s$ et $\Gamma \vdash u \to^+_{\beta\pi} x : s$, où $\_ \vdash \_ \to^+_{\beta\pi} \_ : \_$ dénote la clôture transitive de TPOSR.

On peut alors facilement obtenir l'injectivité des types produits et sommes en utilisant un argument similaire à celui utilisé pour la conversion non typée. Si $\Gamma \vdash \Pi x : A.C \simeq \Pi x : B.D : s$ alors il existe t tel que $\Gamma \vdash \Pi x : A.C \to^+_{\beta\pi} t : s$ et $\Gamma \vdash \Pi x : B.D \to^+_{\beta\pi} t : s$. Par Church-Rosser, on sait qu'il existe E, F tels que $t \equiv \Pi x : E.F$ et s' tel que $\Gamma \vdash A \simeq E \simeq C : s'$ et $\Gamma, x : B \vdash C \simeq F \simeq D : s$. Par transitivité de l'équivalence, on obtient le résultat escompté. On peut faire de même pour les sommes.

L'histoire n'est cependant pas terminée pour l'injectivité puisque nous avons enrichi le système par une relation de coercion. Il nous faut donc prouver une propriété similaire pour ce système. Encore une fois, ce n'est pas une affaire triviale puisqu'on a une règle de transitivité qui pose les mêmes problèmes que pour l'égalité : l'équivalent de l'injectivité des produits n'est pas prouvable pour la coercion. Il faut donc éliminer la transitivité du système de coercion. La preuve est par induction sur la somme des tailles de deux sous-dérivations à partir desquelles on crée une nouvelle dérivation sans transitivité (TPOSR.Transitivity). On détaillera une preuve similaire dans la suite (lemme 2.3.19). L'idée est la suivante : on part d'une présentation de la coercion sans transitivité (TPOSR.CoerceNoTrans) mais avec deux règles supplémentaires pour faire de la conversion n'importe où dans la dérivation :

$$\frac{\Gamma \vdash A \simeq B : s \qquad \Gamma \vdash B \rhd_\to C : s}{\Gamma \vdash A \rhd_\to C : s} \qquad \frac{\Gamma \vdash A \rhd_\to B : s \qquad \Gamma \vdash B \simeq C : s}{\Gamma \vdash A \rhd_\to C : s}$$

On montre simplement que cette relation est transitive.

Ensuite on peut prouver l'injectivité des produits et sommes vis-à-vis de la coercion (TPOSR.Injectivity) : si $\Gamma \vdash \Pi x : A.C \rhd \Pi x : B.D : s$ alors il existe s' tel que $\Gamma \vdash B \rhd A : s'$ et $\Gamma, x : B \vdash C \rhd D : s$.

Une fois l'injectivité prouvée pour les constructeurs de types, on peut aisément prouver l'autoréduction du système TPOSR (TPOSR.SubjectReduction) de façon classique.

Équivalence des systèmes

Pour montrer l'équivalence du système TPOSR avec le système original, on doit montrer des propriétés de la fonction d'effacement des étiquettes (dénotée $|\_|$) (TPOSR.Unlab). Dans ce module, à part des propriétés simples de commutation de l'effacement avec la substitution et l'affaiblissement, on montre que les étapes de réduction étiquetées correspondent aux étapes non étiquetées : si $|t'| \to_{\beta\pi} u$, alors il existe u' tel que $|u'| = u$ et $t' \to_{\beta\pi} u'$.

Il est ensuite facile de montrer que toute dérivation d'un jugement dans TPOSR donne lieu à un jugement dans Russell pour les mêmes termes effacés (Meta.TPOSR_JRussell). On montre par exemple par simple induction sur la dérivation de typage : Si $\Gamma \vdash t \to t' : T$ alors $|\Gamma| \vdash |t| : |T|$ et $|\Gamma| \vdash |t'| : |T|$.

Dans l'autre sens, on a besoin d'un lemme supplémentaire : si deux termes étiquetés typables ont des effacés convertibles alors ils ont un réduit commun dans TPOSR et donc des types coercibles (TPOSR.UnlabConv). Une fois armés de ce résultat et son extension aux contextes on peut prouver que l'effacement est complet vis-à-vis du système original. On va en fait montrer ce résultat par rapport à un système algorithmique (noté $\vdash_{\mathcal{A}}$) où l'on a déjà enlevé la transitivité de la coercion et utilisé une égalité définitionnelle non typée (Meta.Russell). On montre qu'à tout jugement de ce système correspond un jugement de TPOSR (Meta.Russell_TPOSR) : si $\Gamma \vdash_{\mathcal{A}} t : T$ alors il existe Γ', t' et T' tels que $|\Gamma'| = \Gamma$, $|t'| = t$ et $T' = |T|$ avec $|\Gamma'| \vdash |t'| \to |t'| : |T'|$.

Il est facile de montrer que toute dérivation du système déclaratif donne lieu à une dérivation du système algorithmique (Meta.JRussell_Russell). En particulier, on obtient

que l'égalité de jugement $\Gamma \vdash T \equiv_{\beta\pi} U : s$ est équivalente à la conversion lorsque les types sont bien sortés : $\Gamma \vdash_{\mathcal{A}} T : s \wedge \Gamma \vdash_{\mathcal{A}} U : s \wedge T \equiv_{\beta\pi} U$. On obtient en combinant ces deux résultats que le système déclaratif est équivalent au système TPOSR.

Il est finalement possible de transposer la preuve d'autoréduction dans les systèmes algorithmiques et déclaratifs (Meta.SubjectReduction). On traduit le jugement de typage dans TPOSR, puis on transforme la séquence de réductions non étiquetées en réductions étiquetées. On utilise l'autoréduction de TPOSR puis l'on revient dans le système original.

Résultats Les principaux résultats de la preuve complète sont que les systèmes algorithmiques et déclaratifs sont équivalents et valident tout les deux l'autoréduction comme nous l'avons décrit précédemment.

On obtient en particulier le théorème généralisé suivant pour Russell :

Théorème 2.2.1 (Clôture par conversion)**.** *Si* $\Gamma \vdash t : T$, $\Gamma \vdash u : U$ *et* $t \equiv_{\beta\pi} u$ *alors il existe une sorte* s *telle que* $\Gamma \vdash T \rhd U : s$.

Démonstration. Par Church-Rosser et l'autoréduction, on a un réduit commun v tel que $\Gamma \vdash v : T, U$. Par unicité du typage on obtient $\Gamma \vdash T \rhd U : s$. □

Nous avons aussi montré que l'on pouvait adapter la technique de Adams (2006) à un système avec sous-typage, où deux types équivalents ne sont pas nécessairement convertibles (on peut passer par un type sous-ensemble dans notre cas). Pour cela, on a internalisé dans les règles du système TPOSR le fait que deux termes avec étiquettes étaient équivalents si celles-ci étaient coercibles.

On a aussi traité les sommes dépendantes en ajoutant des étiquettes aux projections. Les règles de réduction associées restent linéaires à gauche, on ne force pas de correspondance entre l'annotation et l'objet réduit lorsqu'on rencontre un radical.

2.2.3 Une preuve mécanisée

Pour mécaniser cette preuve, nous sommes partis de la formalisation du CC développée par Barras (1996) (sans les familles inductives) qu'on a étendu avec des sommes (CCSum.Types). On a adapté toute la métathéorie jusqu'à l'autoréduction. Cette formalisation utilise une présentation du calcul comme PTS, donc avec une règle de conversion non typée, définitionnelle.

Initialement, nous ne nous étions intéressés qu'à l'élimination de la transitivité de la coercion dans un système avec égalité définitionnelle (Russell.Types). Ce système est finalement le premier raffinement vers un système algorithmique que nous décrirons dans la suite, qui inclut lui aussi une preuve d'élimination de la transitivité. Le système développé dans (JRussell.Types) correspond à la présentation décrite au début de ce chapitre, avec la coercion et un jugement d'égalité. On a développé la métathéorie basique de ces deux systèmes, jusqu'aux lemmes de génération. Dans l'impossibilité de prouver l'autoréduction pour l'un de ces deux systèmes, on a adopté la technique de Robin Adams et développé une version de TPOSR appropriée (TPOSR.Types). Les modules contenus dans (Meta) établissent les relations entre ces trois systèmes, culminant dans les preuves d'autoréduction (Meta.SubjectReduction).

Le développement complet comprend environ 20000 lignes de Coq dont 6000 de spécification (programmes et énoncés).

La preuve est développée dans l'espace de nom (Lambda). À la racine, nous avons la définition de l'algèbre de termes et des fonctions de substitution, "*lifting*", et réduction associées. On utilise des indices de de Bruijn (1972) pour représenter les variables et de

simples listes pour les contextes. Cette partie a été reprise de (Barras 1996) et simplement étendue avec des sommes dépendantes et un constructeur de type pour les sous-ensembles (.Terms). Ces modules incluent aussi des définitions et preuves sur la réduction $\beta\pi$ associée à ces termes (réductions à un pas et parallèles), dont une preuve de la propriété de Church-Rosser.

L'extension de la formalisation du CC par Barras avec les sommes dépendantes fut directe. Ce système n'est pas utilisé dans la preuve mais il nous a permis de tester nos définitions sur les sommes. À terme, on voudrait mécaniser la preuve de traduction qu'on présentera au chapitre suivant, qui plonge Russell dans une variante du CC avec métavariables. On pourra alors utiliser ce système.

On a ensuite les deux variantes de Russell, l'une avec égalité de jugement et règle de transitivité pour la coercion et l'autre basée sur l'égalité définitionnelle définie pour le CC avec sommes et un jugement de coercion dont on a éliminé la règle transitivité. La preuve démontre l'équivalence de ces deux systèmes et contient pour chacun leur métathéorie de base jusqu'à l'autoréduction.

Représentation

Les preuves des propriétés structurelles basiques de ces systèmes sont routinières. On opère toujours par induction mutuelle sur l'ensemble des jugements. La difficulté essentielle réside dans l'expression des lemmes en présence d'indices de de Bruijn. Par exemple, pour démontrer un lemme l'affaiblissement, il faut exprimer qu'un contexte est un découpage d'un autre contexte, dans lequel on a inséré un nouveau type et opérer le "*lifting*" correspondant sur le terme et le type du jugement :

$$\begin{array}{l}\forall\ \Gamma\ A\ s, \Gamma \vdash A : s \rightarrow \\ \forall\ \Delta\ t\ T, \Delta \vdash t : T \rightarrow \\ \forall\ n\ \Delta', \mathit{Env.ins\_in\_env}\ A\ n\ \Delta\ \Delta' \rightarrow \\ \mathit{Env.trunc}\ \_\ n\ \Delta\ \Gamma \rightarrow \\ \Delta' \vdash (\texttt{lift\_rec}\ 1\ t\ n) : (\texttt{lift\_rec}\ 1\ T\ n)\end{array}$$

Ici Env.ins_in_env représente l'insertion d'un type dans un contexte à un indice n donné et Env.trunc le préfixe d'un contexte après un indice n (les contextes sont représentés à l'envers). En dehors de cela, les preuves sont relativement simples. Faute d'expérience, nous n'avons pas beaucoup utilisé les possibilités d'automatisation offertes par Coq dans ces preuves. Une partie de celles-ci se résume en effet à faire une induction puis à trouver des instances pour les variables n'apparaissant pas dans la conclusion d'une règle de typage. Il est probablement possible de réduire considérablement la taille de ces preuves.

Induction, inversion

Une difficulté plus importante a été rencontrée lorsqu'on a voulu montrer les lemmes de génération. Typiquement, un lemme de génération se présente comme ceci :

Lemma gen_sorting_var : $\forall\ \Gamma\ n\ s, \Gamma \vdash \mathsf{Ref}\ n : \mathsf{Srt}\ s \rightarrow s = \mathsf{prop} \vee s = \mathsf{set}$.

On montre ce lemme par induction sur la dérivation de typage. Dans ce cas on va appliquer le principe d'induction sur la prémisse $\Gamma \vdash \mathsf{Ref}\ n : \mathsf{Srt}\ s$. Or le principe d'induction associée à la relation de typage a une conclusion de la forme :

$\forall\ \Gamma\ t\ T, \Gamma \vdash t : T \rightarrow P\ \Gamma\ t\ T$ pour un certain $P : \forall\ (\Gamma : \mathrm{env})\ (t\ T : \mathsf{term}), \mathtt{Prop}$.

Lorsqu'on applique ce principe sur notre prémisse, le prédicat d'élimination trouvé par Coq sera $(\lambda\ \Gamma\ t\ T,\ s = \mathsf{prop} \vee s = \mathsf{set})$ puisqu'aucun des termes Γ, $\mathsf{Ref}\ n$ ou $\mathsf{Str}\ s$

n'apparaît dans le but. On a ici perdu toute l'information sur l'instance du jugement de typage qu'on élimine : on se retrouve à devoir prouver ce but pour n'importe quelle instanciation du jugement de typage. Pour remédier à ce problème, il existe une technique bien connue : on généralise le but par un certain nombre d'égalités qui vont contraindre l'instance généralisée :

```
Lemma gen_sorting_var_aux : ∀ Γ t T, Γ ⊢ t : T → ∀ n s, t = Ref n → T = Srt s →
  s = prop ∨ s = set.
```

On peut maintenant faire l'induction sur la prémisse générale sans perdre l'information sur l'instance particulière à laquelle on s'intéresse. À l'aide d'un peu de raisonnement équationnel, on peut éliminer tout les cas impossibles que constituent les règles de typage dont la conclusion ne s'unifie pas avec $\Gamma \vdash \mathsf{Ref}\ n : \mathsf{Srt}\ s$. Évidemment, on peut obtenir le lemme gen_sorting_var à partir de celui-ci en instanciant t et T par $\mathsf{Ref}\ n$ et $\mathsf{Srt}\ s$ et en fournissant des preuves de réflexivité de l'égalité pour les prémisses supplémentaires.

Au moment d'écrire ces preuves, Coq n'était pas en mesure d'automatiser cette manipulation, on a donc du dupliquer les énoncés d'un bon nombre de lemmes de cette façon. Nous avons depuis développé une tactique nommée `dependent induction` permettant de faire directement ce genre d'induction-inversion en adaptant la technique utilisée par McBride (2000). C'est en fait la même technique qui est à la base de la compilation du pattern-matching dépendant qu'on étudiera section 4.2.1.

Induction sur les preuves

Nous avons rencontré un dernier obstacle important lors des preuves de Church-Rosser pour TPOSR et pour l'élimination de la transitivité. En effet, ces deux preuves sont par induction sur la taille des dérivations. Seulement, nos dérivations sont des objets de sorte Prop dans Coq. Il n'est donc pas possible d'écrire une fonction donnant la taille d'une dérivation sous forme d'un naturel par récurrence sur la dérivation : l'élimination d'une proposition vers un type informatif est interdite. Il faut donc ruser pour associer une taille à chaque dérivation. Nous avons expérimenté deux techniques pour y parvenir :

1. Pour la preuve de Church-Rosser du système TPOSR (TPOSR.ChurchRosserDepth), on a créé un nouvel inductif mutuel isomorphe aux jugements du système mais ou l'on a ajouté un index donnant la profondeur de la dérivation. On montre facilement que pour toute dérivation du système original, il existe un naturel et une dérivation annotée du même jugement, avec une existentielle dans Prop (TPOSR.TypesDepth). Inversement, on peut faire une fonction d'oubli des dérivations annotées aux dérivations originales. On montre alors la propriété de Church-Rosser par induction bien fondée sur la taille de la première dérivation. La preuve s'appuyant sur les lemmes de génération, il faut aussi faire des versions de ces lemmes qui assurent que les dérivations obtenues sont bien de taille inférieure à la dérivation inversée.
2. Pour la preuve d'élimination de la transitivité, on a été un peu plus loin et montré une équivalence entre la version des jugements dans Prop annotés par un entier et une version dans Set. On peut en effet montrer cette équivalence par induction sur l'entier dans un sens et sur la dérivation dans l'autre. On peut alors définir une fonction depth donnant la profondeur d'une dérivation et procéder par induction sur celle-ci (TPOSR.TransitivitySet) :

```
Theorem coerce_trans_aux : ∀ (n : nat) Γ A B C s
  (d1 : Γ ⊢ A ▷→Set B : s) (d2 : Γ ⊢ B ▷→Set C : s),
  n = (depth d1) + (depth d2) →
  Γ ⊢ A ▷Set→ C : s.
```

L'entier n nous permet de lancer la preuve par induction bien fondée sur la somme des tailles des dérivations. À l'inverse de la preuve de CR, celle-ci ne nécessite pas de lemmes de génération, mais l'on combine les sous-dérivations obtenues par analyse de cas pour en donner de nouvelles. Dans cette formulation, on peut ainsi utiliser le calcul pour décharger une grande partie du travail de vérification que les sous-dérivations ont la bonne taille.

2.2.4 Conclusion

Nous avons présenté une preuve mécanisée de l'autoréduction pour le système Russell ainsi que l'équivalence avec un système algorithmique. Nous allons maintenant raffiner ce système pour obtenir un système équivalent dont les problèmes d'inférence et de vérification de types sont décidables.

2.3 Élaboration du système algorithmique et propriétés

Pour pouvoir implanter le typeur, il nous faut un système dirigé par la syntaxe. Ce n'est pas le cas du système déclaratif, aussi bien pour le typage que pour la coercion. On va donc raffiner ces systèmes dans l'optique d'en extraire un algorithme. On note $\vdash_\bullet$ le séquent de chaque jugement de bonne formation, typage algorithmique et coercion algorithmiques définis figure 2.7 et 2.9 page 35. Ces deux jugements sont quelque peu éloignés des originaux, néanmoins nous allons montrer qu'ils sont corrects et complets vis-à-vis de leurs géniteurs. La correction (si l'on a une dérivation d'un jugement dans le système algorithmique, alors c'est un jugement valide du système déclaratif) est le sens le plus facile à montrer, nous allons donc commencer par là. On décrira ensuite la méthode de construction des systèmes algorithmiques pour aboutir d'une part au théorème de complétude qui montre qu'on peut dériver les mêmes jugements (à coercion près) dans le système algorithmique que dans le système déclaratif et d'autre part à la décidabilité du jugement de typage algorithmique.

Le système algorithmique présenté ici est sensiblement différent de celui formalisé dans la preuve Coq. Essentiellement, on utilise la mise en forme normale de tête plutôt que la conversion. Dans la version algorithmique qu'on a formalisée on a éliminé la transitivité et ajouté la règle :

$$\frac{\Gamma \vdash A, B, C, D : s \quad A \equiv_{\beta\pi} B \qquad \Gamma \vdash B \rhd C : s \qquad C \equiv_{\beta\pi} D}{\Gamma \vdash A \rhd D : s}$$

On peut montrer l'équivalence avec le système présenté ici après avoir montré le lemme 2.3.17.

Vis-à-vis de la preuve mécanisée présentée à la section précédente, cette section donne les démonstrations en langage naturel d'un certain nombre des preuves basiques du système et en particulier une preuve d'admissibilité de la transitivité (lemme 2.3.19).

Il nous a fallu changer quelque peu les règles pour obtenir le système algorithmique. En particulier, on a utilisé la fonction μ_0 de PVS (Owre et Shankar 1997) renommée $\mu_\bullet()$ (figure 2.8) ici pour opérer des *décompréhensions*. Cette fonction efface les constructeurs de type sous-ensemble en tête d'un type, par exemple : $\mu_\bullet(\{\ f : \mathsf{nat} \to \mathsf{nat} \mid f\ 0 = 0\ \}) = \mathsf{nat} \to \mathsf{nat}$. On verra son utilité dans la suite. Notons aussi que les jugements de la forme $\Gamma \vdash_\bullet t : s$ où $s \in \{\mathtt{Set}, \mathtt{Prop}, \mathtt{Type}\}$ sont une abréviation pour $\Gamma \vdash_\bullet t : T \wedge T \to_{\beta\rho} s$, c'est une pratique courante lorsque l'on présente un système algorithmique basé sur les systèmes de types purs.

Notations

On introduit la notation $\Gamma \vdash_\bullet T, U : s$ pour $\Gamma \vdash_\bullet T : s \wedge \Gamma \vdash_\bullet U : s$.

On note $x^\downarrow$ la mise en forme normale de tête de x selon la réduction $\beta\pi$ définie de la façon suivante :

$$\begin{array}{lcll} ((\lambda x : T.e)\ v)^\downarrow & = & e[v/x]^\downarrow & \\ \pi_1\ (x, y)^\downarrow & = & x^\downarrow & \\ \pi_2\ (x, y)^\downarrow & = & y^\downarrow & \\ e^\downarrow & = & e & \{\text{si e est d'une autre forme}\} \end{array}$$

Fig. 2.6: Définition de la réduction de tête

$$\textsc{Wf-Empty}\ \frac{}{\vdash_\bullet []} \qquad \textsc{Wf-Var}\ \frac{\Gamma \vdash_\bullet A : s}{\vdash_\bullet \Gamma, x : A}\ s \in \mathcal{S} \wedge x \notin \Gamma$$

$$\textsc{Axiom}\ \frac{\vdash_\bullet \Gamma}{\Gamma \vdash_\bullet s_1 : s_2}\ (s_1, s_2) \in \mathcal{A}$$

$$\textsc{Var}\ \frac{\vdash_\bullet \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash_\bullet x : A}$$

$$\textsc{Prod}\ \frac{\Gamma \vdash_\bullet T : s_1 \qquad \Gamma, x : T \vdash_\bullet U : s_2}{\Gamma \vdash_\bullet \Pi x : T.U : s_3}\ (s_1, s_2, s_3) \in \mathcal{R}$$

$$\textsc{Abs}\ \frac{\Gamma \vdash_\bullet \Pi x : T.U : s \qquad \Gamma, x : T \vdash_\bullet M : U}{\Gamma \vdash_\bullet \lambda x : T.M : \Pi x : T.U}$$

$$\textsc{App}\ \frac{\Gamma \vdash_\bullet f : T \quad \mu_\bullet(T) = \Pi x : V.W \qquad \Gamma \vdash_\bullet u : U \qquad \Gamma \vdash_\bullet U \rhd V : s}{\Gamma \vdash_\bullet (f u) : W[u/x]}$$

$$\textsc{Sum}\ \frac{\Gamma \vdash_\bullet T : s \qquad \Gamma, x : T \vdash_\bullet U : s}{\Gamma \vdash_\bullet \Sigma x : T.U : s}\ s \in \{\texttt{Prop}, \texttt{Set}\}$$

$$\textsc{Pair}\ \frac{\Gamma \vdash_\bullet \Sigma x : T.U : s \qquad \begin{array}{c}\Gamma \vdash_\bullet t : T' \rhd T : s \\ \Gamma \vdash_\bullet u : U' \rhd U[t/x] : s\end{array}}{\Gamma \vdash_\bullet (t, u)_{\Sigma x:T.U} : \Sigma x : T.U}$$

$$\textsc{Pi-1}\ \frac{\Gamma \vdash_\bullet t : S \qquad \mu_\bullet(S) = \Sigma x : T.U}{\Gamma \vdash_\bullet \pi_1\ t : T} \qquad \textsc{Pi-2}\ \frac{\Gamma \vdash_\bullet t : S \qquad \mu_\bullet(S) = \Sigma x : T.U}{\Gamma \vdash_\bullet \pi_2\ t : U[\pi_1\ t/x]}$$

$$\textsc{Subset}\ \frac{\Gamma \vdash_\bullet U : \texttt{Set} \qquad \Gamma, x : U \vdash_\bullet P : \texttt{Prop}}{\Gamma \vdash_\bullet \{\ x : U \mid P\ \} : \texttt{Set}}$$

FIG. 2.7: Calcul de coercion par prédicats - version algorithmique

On omet le contexte et la sorte pour le jugement de coercion lorsqu'ils sont dérivables du contexte.

$$\begin{array}{lcll} \mu_\bullet(t) & = & \mu_\bullet(U) & \text{si } t^\downarrow = \{\ x : U \mid P\ \} \\ \mu_\bullet(t) & = & t & \text{sinon} \end{array}$$

FIG. 2.8: Définition de $\mu_\bullet()$

Propriétés élémentaires

Voici quelques propriétés élémentaires du système :

Lemme 2.3.1 (Bonne formation des contextes)**.** *Si* $\Gamma \vdash_\bullet t : T$ *alors* $\vdash \Gamma$.

Démonstration. Par induction sur la dérivation de typage. □

Fait 2.3.2 (Inversion du jugement de bonne formation)**.** *Si* $\vdash \Gamma, x : U$ *alors il existe* s, $\Gamma \vdash_\bullet U : s$ *et* $s \in \{\texttt{Set}, \texttt{Prop}, \texttt{Type}\}$.

Lemme 2.3.3 (Affaiblissement)**.** *Si* $\Gamma, \Delta \vdash_\bullet t : T$ *alors pour tout* $x : S \notin \Gamma, \Delta$ *tel que* $\vdash \Gamma, x : S, \Delta$ *on a* $\Gamma, x : S, \Delta \vdash_\bullet t : T$

$$\rhd\text{-Conv}\ \frac{\Gamma \vdash_\bullet T, U : s \qquad T \equiv_{\beta\pi} U}{\Gamma \vdash_\bullet T \rhd U : s}\ T = T^\downarrow \wedge T \neq \Pi, \Sigma, \{|\} \wedge U = U^\downarrow$$

$$\rhd\text{-}\downarrow\ \frac{\Gamma \vdash_\bullet T^\downarrow \rhd U^\downarrow : s \qquad \Gamma \vdash_\bullet T, U : s}{\Gamma \vdash_\bullet T \rhd U : s}\ T \neq T^\downarrow \vee U \neq U^\downarrow$$

$$\rhd\text{-Prod}\ \frac{\Gamma \vdash_\bullet U \rhd T : s_1 \qquad \Gamma, x : U \vdash_\bullet V \rhd W : s_2}{\Gamma \vdash_\bullet \Pi x : T.V \rhd \Pi x : U.W : s_2}$$

$$\rhd\text{-Sum}\ \frac{\Gamma \vdash_\bullet T \rhd U : s \qquad \Gamma, x : T \vdash_\bullet V \rhd W : s}{\Gamma \vdash_\bullet \Sigma x : T.V \rhd \Sigma x : U.W : s}\ s \in \{\texttt{Set}, \texttt{Prop}\}$$

$$\rhd\text{-Proof}\ \frac{\Gamma \vdash_\bullet T \rhd U : \texttt{Set}}{\Gamma \vdash_\bullet T \rhd \{\ x : U \mid P\ \} : \texttt{Set}}\ T = T^\downarrow$$

$$\rhd\text{-Subset}\ \frac{\Gamma \vdash_\bullet U \rhd T : \texttt{Set} \qquad \Gamma, x : U \vdash_\bullet P : \texttt{Prop}}{\Gamma \vdash_\bullet \{\ x : U \mid P\ \} \rhd T : \texttt{Set}}\ T = T^\downarrow$$

Fig. 2.9: Coercion par prédicats - version algorithmique

Démonstration. Par induction sur la dérivation de typage.

- PropSet : Trivial.
- Var : On a $x : S \notin \Gamma, \Delta$, donc $\Gamma, x : S, \Delta \vdash_\bullet y : T$ est toujours dérivable.
- Prod : Par induction $\Gamma, x : S, \Delta \vdash_\bullet T : s1$ et $\Gamma, x : S, \Delta, y : T \vdash_\bullet U : s2$. On applique Prod pour obtenir $\Gamma, x : S, \Delta \vdash_\bullet \Pi x : T.U : s2$. De même pour le reste des règles.

□

2.3.1 Correction

On montre tout d'abord la correction de la coercion algorithmique qui sera nécessaire pour la correction du typage :

Théorème 2.3.4 (Correction de la coercion)**.** *Si* $\Gamma \vdash_\bullet U \rhd V :$ *alors* $U \rhd V$*.*

Démonstration. Les règles du système algorithmique sont un sous-ensemble des règles du système déclaratif, excepté pour la règle $\rhd$-$\downarrow$. On utilise $\rhd$-Conv et $\rhd$-Trans pour montrer son admissibilité dans le système déclaratif.

$$\dfrac{\dfrac{U \equiv_{\beta\pi} U^\downarrow}{U\ \rhd U^\downarrow} \qquad \dfrac{U^\downarrow\ \rhd T^\downarrow \qquad \dfrac{T^\downarrow \equiv_{\beta\pi} T}{T^\downarrow\ \rhd T}}{U^\downarrow\ \rhd T}}{U \rhd T}$$

□

On a besoin d'un lemme sur l'opération $\mu_\bullet()$ définie figure 2.8.

Lemme 2.3.5 ($\mu_\bullet()$ et coercion)**.** *Si* $\Gamma \vdash_\bullet T : s$ *alors* $\Gamma \vdash_\bullet T \rhd \mu_\bullet(T) :$.

Démonstration. Il suffit de suivre la définition de $\mu_\bullet()$. La mise en forme normale de tête est équivalente à l'utilisation de $\rhd$-$\downarrow$ dans notre jugement de coercion. $\mu_\bullet()$ est en fait l'application répétée de la règle $\rhd$-Subset. □

Lemme 2.3.6 (Conservation des sortes par $\mu_\bullet()$)**.** *Si* $\Gamma \vdash_\bullet S : s$ *alors* $\Gamma \vdash_\bullet \mu_\bullet(S) : s$

Démonstration. Par le simple fait que si $S = \{\ x : U \mid P\ \}$ et $G \vdash_\bullet S : s$ alors $S :$ `Set` et $U :$ `Set` (par Subset), sinon $S = \mu_\bullet(S)$. □

Théorème 2.3.7 (Correction du typage)**.** *Si* $\Gamma \vdash_\bullet t : T$ *alors* $\Gamma \vdash t : T$

Démonstration. Par induction sur la dérivation de typage dans le système algorithmique.

- Wf-Empty,Wf-Var,PropSet,Var,Prod,Abs, Sum, Sum : règles inchangées.
- App : On a

$$\frac{\Gamma \vdash_\bullet f : T \quad \mu_\bullet(T) = \Pi x : V.W \qquad \Gamma \vdash_\bullet u : U \qquad \Gamma \vdash_\bullet U \rhd V : s}{\Gamma \vdash_\bullet (fu) : W[u/x]}$$

  Par induction, $\Gamma \vdash f : T$, et $T \rhd \Pi x : V.W$ par le lemme 2.3.5 et la correction de la coercion. On peut donc dériver $\Gamma \vdash f : \Pi x : V.W$ à l'aide de la règle Coerce. Par le lemme 2.3.4 appliqué à $\Gamma \vdash_\bullet U \rhd V :$ et l'hypothèse $\Gamma \vdash u : U$, on obtient $\Gamma \vdash u : V$ par Coerce. Donc, par App, on a bien $\Gamma \vdash fu : W[u/x]$.

- Pair : On a

$$\frac{\begin{array}{cc} & \Gamma \vdash_\bullet t : T' \rhd T : s \\ \Gamma \vdash_\bullet \Sigma x : T.U : s & \Gamma \vdash_\bullet u : U' \rhd U[t/x] : s \end{array}}{\Gamma \vdash_\bullet (t, u)_{\Sigma x:T.U} : \Sigma x : T.U}$$

  Par induction et correction de la coercion, on peut dériver : $\Gamma \vdash t : T$ et $\Gamma \vdash u : U[t/x]$. On a $\Gamma \vdash \Sigma x : T.U : s$, on peut donc appliquer Pair

- Pi-2 : On a

$$\frac{\Gamma \vdash_\bullet t : S \qquad \mu_\bullet(S) = \Sigma x : T.U}{\Gamma \vdash_\bullet \pi_2\ t : U[\pi_1\ t/x]}$$

  Par induction, $\Gamma \vdash t : S$, et par le lemme 2.3.5 et la correction de la coercion $S \rhd \Sigma x : T.U$. On peut donc dériver $\Gamma \vdash t : \Sigma x : T.U$ à l'aide de Coerce. On peut directement appliquer Pi-2 à cette prémisse pour obtenir $\Gamma \vdash \pi_2\ t : U[\pi_1\ t/x]$. De même pour Pi-1.

□

À partir de cette preuve, on peut déjà déduire que le système algorithmique a l'unicité des sortes pour les termes convertibles :

Lemme 2.3.8 (Unicité des sortes et conversion pour le système algorithmique)**.** *Si* $\Gamma \vdash_\bullet T : s$ *et* $\Gamma \vdash_\bullet U : s'$ *avec* $T \equiv_{\beta\pi} U$ *alors* $s = s'$.

Démonstration. Par la correction et la clôture par conversion (lemme 2.2.1) du système déclaratif. □

On a prouvé que notre système algorithmique était correct, c'est-à-dire que ses jugements valides sont bien inclus dans ceux du système déclaratif, il faut maintenant montrer qu'il les inclut tous (ou presque !).

2.3.2 Complétude et décidabilité

On va maintenant repartir des systèmes déclaratifs pour montrer comment l'on a construit les systèmes algorithmiques.

On s'intéresse tout d'abord au jugement de coercion. Pour rendre le jugement de coercion décidable, il faut traiter les règles $\rhd$-Conv et $\rhd$-$\downarrow$ qu'on peut appliquer à n'importe quel moment et la règle $\rhd$-Trans qui n'est pas dirigée par la syntaxe (il faut "deviner" un type T). Le système de coercion algorithmique (figure 2.9) est le même que le système déclaratif (figure 2.3) mais où l'on applique $\rhd$-Conv seulement si aucune autre règle ne s'applique après avoir appliqué $\rhd$-$\downarrow$ et sans la règle $\rhd$-Trans.

Décidabilité et complétude de la coercion

On va montrer que les deux systèmes de coercion sont équivalents vis-à-vis de la conversion. On montrera plus tard pourquoi on peut éliminer la règle de transitivité.

Il nous faut tout d'abord des lemmes d'inversion sur la conversion :

Lemme 2.3.9. *Si* $\Pi x : T.U \equiv_{\beta\pi} S$ *alors* $S^{\downarrow} = \Pi x : T'.U'$ *avec* $T \equiv_{\beta\pi} T'$ *et* $U \equiv_{\beta\pi} U'$.

Lemme 2.3.10. *Si* $\Sigma x : T.U \equiv_{\beta\pi} S$ *alors* $S^{\downarrow} = \Sigma x : T'.U'$ *avec* $T \equiv_{\beta\pi} T'$ *et* $U \equiv_{\beta\pi} U'$.

On montre que les sortes ne peuvent se coercer que vers elles-mêmes.

Lemme 2.3.11 (Coercion de sortes)**.** *Si* $\Gamma \vdash_\bullet T \rhd s :$ *ou* $\Gamma \vdash_\bullet s \rhd T :$ *où* $s \in \{\texttt{Set}, \texttt{Prop}, \texttt{Type}\}$ *alors* $T \equiv_{\beta\pi} s$.

Démonstration. Les seuls règles permettant de dériver de tels jugements sont $\rhd$-Conv, $\rhd$-$\downarrow$ et $\rhd$-Proof ou $\rhd$-Subset.

- $\rhd$-Conv : Trivial.
- $\rhd$-$\downarrow$: Par induction.
- $\rhd$-Proof, $\rhd$-Subset : On ne peut pas avoir un jugement de la forme $\Gamma \vdash_\bullet s \rhd \{ x : U \mid P \}$: puisque cela impliquerait que $\Gamma \vdash_\bullet s, U : s'$ avec $\Gamma \vdash_\bullet U : \texttt{Set}$ donc $\Gamma \vdash_\bullet s : \texttt{Set}$ ce qui est impossible. De façon symétrique pour $\rhd$-Subset.

□

Lemme 2.3.12 (Convertibilité avec une sorte et réduction)**.** *Si* $T \equiv_{\beta\pi} s$ *alors* $T \to_{\beta\rho} s$.

Démonstration. Par la propriété de Church-Rosser pour la réduction $\to_{\beta\rho}$ il existe v tel que $T \twoheadrightarrow_{\beta\rho} v \twoheadleftarrow_{\beta\rho} s$. Or s ne peut par se réduire sur un autre terme, on a donc $T \twoheadrightarrow_{\beta\rho} s$. □

Corollaire 2.3.13 (Coercion de sortes et réduction)**.** *Si* $\Gamma \vdash_\bullet T \rhd s :$ *ou* $\Gamma \vdash_\bullet s \rhd T :$ *où* $s \in \{\texttt{Set}, \texttt{Prop}, \texttt{Type}\}$ *alors* $T \twoheadrightarrow_{\beta\pi} s$.

L'affaiblissement montre que notre notion de coercion joue un rôle similaire à la seule conversion vis-à-vis du typage. On peut dériver les mêmes jugements dans des contextes où les variables ont des types coercibles. Ici la taille des dérivations ne change pas.

Lemme 2.3.14 (Affaiblissement)**.**

$$\Gamma \vdash_\bullet S \rhd S' : s \Rightarrow \begin{cases} \vdash \Gamma, x : S, \Delta & \Rightarrow \vdash \Gamma, x : S', \Delta \\ \Gamma, x : S, \Delta \vdash_\bullet t : T & \Rightarrow \Gamma, x : S', \Delta \vdash_\bullet t : T' \rhd T \\ \Gamma, x : S, \Delta \vdash_\bullet T \rhd T' : s & \Rightarrow \Gamma, x : S', \Delta \vdash_\bullet T \rhd T' : s \end{cases}$$

Démonstration. Par induction mutuelle sur les dérivations de typage, coercion et bonne formation.

– Wf-Empty : Trivial.

– Wf-Var : La conclusion est $\vdash \Gamma, x : S, \Delta$. Par induction sur la taille de Δ.

• $\Delta = []$: La racine de la dérivation est de la forme :

$$\frac{\Gamma \vdash_\bullet S : s}{\vdash \Gamma, x : S} \; s \in \{\texttt{Set}, \texttt{Prop}, \texttt{Type}\}$$

On a donc $\Gamma \vdash_\bullet S' : s$, et par Wf-Var, $\vdash \Gamma, x : S'$.

• $\Delta \equiv \Delta', y : U$: La racine de la dérivation est de la forme :

$$\frac{\Gamma, x : S, \Delta' \vdash_\bullet U : s}{\vdash \Gamma, x : S, \Delta', y : U} \; s \in \{\texttt{Set}, \texttt{Prop}, \texttt{Type}\}$$

Par induction sur la dérivation de typage $\Gamma, x : S', \Delta' \vdash_\bullet U : s$, on a donc bien $\vdash \Gamma, x : S', \Delta', y : U$ par Wf-Var.

– PropSet : Par induction, $\vdash \Gamma, x : S', \Delta$, on applique simplement la règle.

– Var : Par induction, $\vdash \Gamma, x : S', \Delta$. La seule différence avec le contexte précédent est le type associé à x, donc si $t \neq x$, on peut simplement réappliquer Var. Si $t \equiv x$ on a la dérivation :

$$\frac{\vdash \Gamma, x : S', \Delta \qquad x : S' \in \Gamma}{\Gamma, x : S', \Delta \vdash_\bullet x : S'}$$

On a bien $S' \rhd S$, la propriété est donc bien vérifiée.

– Prod : Par induction, $\Gamma, x : S', \Delta \vdash_\bullet T : V \rhd s_1$ et $\Gamma, x : S', \Delta, y : T \vdash_\bullet U : W \rhd s_2$. On a donc $V \twoheadrightarrow_{\beta\pi} s_1$ et $W \twoheadrightarrow_{\beta\pi} s_2$. On en déduit $\Gamma, x : S', \Delta \vdash_\bullet T : s_1$ et $\Gamma, x : S', \Delta, y : T \vdash_\bullet U : s_2$. On applique Prod pour obtenir $\Gamma, x : S', \Delta \vdash_\bullet \Pi x : T.U : s_3$. De même, direct par induction pour le reste des règles.

– $\rhd$-Prod, $\rhd$-Sum : Par induction on a $\Gamma, x : S', \Delta \vdash_\bullet U \rhd T : s$ et $\Gamma, x : S', \Delta, x : U \vdash_\bullet V \rhd W : s'$, il suffit d'appliquer $\rhd$-Prod. De même pour $\rhd$-Sum.

– $\rhd$-Conv,$\rhd$-$\downarrow$,$\rhd$-Proof,$\rhd$-Subset : De même, direct par induction. On utilise l'hypothèse d'induction mutuelle pour les dérivations de typage en prémisse.

□

On peut maintenant montrer :

Lemme 2.3.15 (Conservation de la conversion par coercion)**.** *Si* $\Gamma \vdash_\bullet T, U : s$ *et* $T \equiv_{\beta\pi} U$ *alors* $\Gamma \vdash_\bullet T \rhd U : s$.

Démonstration. Par induction sur le nombre de constructeurs $\Pi, \Sigma, \{|\}$ dans la forme normale complète de T.

– $T^\downarrow = \Pi x : X.Y$: Alors $U^\downarrow = \Pi x : V.W$ et $X \equiv_{\beta\pi} V$, $Y \equiv_{\beta\pi} W$ d'après le lemme 2.3.9. Par inversion de la règle Prod on a $\Gamma \vdash_\bullet X : s_1$, $\Gamma \vdash_\bullet V : s_1'$, $\Gamma, x : X \vdash_\bullet Y : s_2$ et $\Gamma, x : V \vdash_\bullet W : s_2'$. Par le lemme 2.3.8 on a $s_1 = s_1'$ et $s_2 = s_2'$. Par induction on a donc $\Gamma \vdash_\bullet V \rhd X : s_1$. On peut donc appliquer le lemme 2.3.14 pour obtenir $\Gamma, x : V \vdash_\bullet Y, W : s_2$. On applique l'hypothèse d'induction pour obtenir $\Gamma, x : V \vdash_\bullet Y \rhd W : s$. On applique alors $\rhd$-Prod à ces deux prémisses.

– $T^\downarrow = \Sigma x : X.Y$: Alors $U^\downarrow = \Sigma x : V.W$, avec $X \equiv_{\beta\pi} V$ et $Y \equiv_{\beta\pi} W$. Par induction et application de $\rhd$-Sum en utilisant le lemme 2.3.14 pour la deuxième prémisse.

– $T^{\downarrow} \equiv \{\ x : X \mid P\ \}$: On a alors $U^{\downarrow} = \{\ x : X' \mid P'\ \}$ avec $X \equiv_{\beta\pi} X'$, $P \equiv_{\beta\pi} P'$, et la propriété est vraie par $\rhd$-Left et $\rhd$-Right :

$$\rhd\text{-Right}\ \frac{\rhd\text{-Left}\ \dfrac{\Gamma \vdash_\bullet X \rhd X' : \texttt{Set}}{\Gamma \vdash_\bullet \{\ x : X \mid P\ \} \rhd X' : \texttt{Set}}}{\Gamma \vdash_\bullet \{\ x : X \mid P\ \} \rhd \{\ x : X' \mid P'\ \} : \texttt{Set}}$$

– Sinon : On applique obligatoirement $\rhd$-Conv et l'on a la prémisse $T \equiv_{\beta\pi} U$, c'est donc direct.

□

Il n'y a pas de problème d'identification de sortes dans ce système, contrairement au système $\lambda\, C_{\leq}$ de Gang Chen (Chen 1998), puisqu'on n'a pas de cumulativité. Par exemple on n'a pas besoin d'identifier $\texttt{nat} : \texttt{Set}$ et $(\lambda x : \texttt{Type}.x)\ \texttt{nat} : \texttt{Type}$ puisque le deuxième terme n'est pas typable dans notre système.

On va maintenant montrer que la règle $\rhd$-Trans est admissible dans notre système algorithmique. Tout d'abord quelques lemmes nécessaires pour la preuve :

Lemme 2.3.16 (Coercion et $\mu_\bullet()$)**.**
– *Si* $\Pi x : X.Y \rhd U$ *alors* $\mu_\bullet(U) = \Pi x : X'.Y'$ *et* $X' \rhd X$, $Y \rhd Y'$.
– *Si* $\Sigma x : X.Y \rhd U$ *alors* $\mu_\bullet(U) = \Sigma x : X'.Y'$ *et* $X \rhd X'$, $Y \rhd Y'$.
– *Pour tout* S, U, $S \rhd \mu_\bullet(U)$ *si et seulement si* $S \rhd U$.

Démonstration. Par induction sur les dérivations de $\rhd$ et la définition de $\mu_\bullet()$.

Dans le dernier cas, de gauche à droite on construit la dérivation en ajoutant des applications de $\rhd$-Proof et dans l'autre sens on est assuré de trouver la preuve dans la dérivation même de $S \rhd U$: si U n'est pas de la forme sous-ensemble c'est direct. Sinon, on peut trouver dans la preuve (en partant de la racine) la première utilisation de la règle $\rhd$-Proof. A partir de là, on cherche la première utilisation d'une règle autre que $\rhd$-Proof ou $\rhd$-Subset. On a une dérivation de $S' \rhd \mu_\bullet(U)$, on peut réappliquer les règles $\rhd$-Subset oubliées précédemment pour obtenir la preuve de $S \rhd \mu_\bullet(U)$. □

Lemme 2.3.17 (Coercion et conversion)**.** *Si* $\Gamma \vdash_\bullet S, T, U : s$, $S \equiv_{\beta\pi} T$ *et* $\Gamma \vdash_\bullet T \rhd U : s$ *alors* $\Gamma \vdash_\bullet S \rhd U : s$.

Démonstration. Par simple inspection des règles on voit que le jugement ne peut distinguer deux termes $\beta\pi$-équivalents (ils ont forcément les mêmes constructeurs de tête appliqués à des termes équivalents). □

Lemme 2.3.18 (Coercion et formes normales de tête)**.** *Si* $\Gamma \vdash_\bullet T \rhd U : s$ *alors* $T^{\downarrow} \rhd U^{\downarrow}$ *est dérivable par une dérivation de taille plus petite ou égale.*

Démonstration. Par induction sur la dérivation de sous-typage dans le système algorithmique.

– $\rhd$-Conv : Trivial.
– $\rhd$-$\downarrow$: On prend la dérivation en prémisse.
– $\rhd$-Prod, $\rhd$-Sum : T et U sont égaux à leurs formes normales de tête, direct.
– $\rhd$-Proof : Par induction $T^{\downarrow} \rhd V^{\downarrow}$, on applique $\rhd$-Proof
– $\rhd$-Subset : idem.

□

On est prêts à montrer la compositionalité du système de coercion algorithmique :

Lemme 2.3.19 (Transitivité de la coercion)**.** *Pour tout S, T, U, si $\Gamma \vdash_\bullet S \rhd T : s$ et $\Gamma \vdash_\bullet T \rhd U : s$ alors $\Gamma \vdash_\bullet S \rhd U : s$.*

Démonstration. On procède par élimination d'une hypothétique règle $\rhd$-Trans aparaissant dans toute dérivation de $\Gamma \vdash_\bullet S \rhd U : s$. On procède par induction sur l'ordre lexicographique $< \mathrm{depth}(\Gamma \vdash_\bullet S \rhd T : s), \mathrm{depth}(\Gamma \vdash_\bullet T \rhd U : s) >$. On peut supposer qu'il n'y a pas d'applications successives de la règle $\rhd$-$\downarrow$ dans nos dérivations, par idempotence de la mise en forme normale de tête et les conditions $T \neq T^\downarrow \wedge U \neq U^\downarrow$ de cette règle. On omet les contextes et les sortes dans la preuve pour plus de clarté, on a déjà présentée une version mécanisée de la preuve pour un système très proche.

– $\rhd$-Conv, _ :

$$\dfrac{\dfrac{S \equiv_{\beta\pi} T}{S \rhd T} \qquad T \rhd U}{S \rhd U}$$

Par le lemme 2.3.17, on élimine trivialement $\rhd$-Trans.

– $\rhd$-$\downarrow$, _ :

$$\dfrac{\dfrac{S^\downarrow \ \rhd T^\downarrow}{S \rhd T} \qquad T \rhd U}{S \rhd U}$$

Par le lemme 2.3.18, il existe une dérivation de $T^\downarrow \rhd U^\downarrow$ de taille plus petite ou égale à la dérivation de $T \rhd U$. On peut donc appliquer l'hypothèse d'induction sur la dérivation de $S^\downarrow \ \rhd T^\downarrow$ et cette dernière pour obtenir une dérivation de $S^\downarrow \rhd U^\downarrow$ donc de $S \rhd U$ par $\rhd$-$\downarrow$.

On peut faire le même raisonnement si la dérivation de $T \rhd U$ se termine par une application de $\rhd$-$\downarrow$. On peut donc se restreindre aux cas où l'on n'utilise ni la règle $\rhd$-$\downarrow$ ni la règle $\rhd$-Conv dans les prémisses.

– $\rhd$-Prod :

$$\dfrac{\dfrac{C \rhd A \qquad B \rhd D}{\Pi x : A.B \rhd \Pi x : C.D} \qquad \Pi x : C.D \rhd U}{\Pi x : A.B \rhd U}$$

On a seulement à traiter le cas où $\Pi x : C.D \rhd U$ est dérivé par $\rhd$-Prod ou $\rhd$-Proof.

- $\rhd$-Prod : Alors on a

$$\dfrac{E \rhd C \qquad D \rhd F}{\Pi x : C.D \rhd \Pi x : E.F}$$

On construit la dérivation :

$$\dfrac{\dfrac{E \rhd C \qquad C \rhd A}{E \rhd A} \qquad \dfrac{B \rhd D \qquad D \rhd F}{B \rhd F}}{S = \Pi x : A.B \rhd \Pi x : E.F = U}$$

La taille des dérivations de $E \rhd C$, $C \rhd A$ et $B \rhd D$, $D \rhd F$ étant plus petites que $\Pi x : A.B \rhd \Pi x : C.D$ et $\Pi x : C.D \rhd \Pi x : E.F$, on élimine bien la transitivité dans ce cas.

- $\rhd$-Proof : On a :

$$\dfrac{\Pi x : C.D \rhd E}{\Pi x : C.D \rhd \{\ y : E \mid P\ \}}$$

Par induction, on a :

$$\dfrac{\dfrac{\Pi x : A.B \rhd \Pi x : C.D \qquad \Pi x : C.D \rhd E}{\Pi x : A.B \rhd E}}{S = \Pi x : A.B \rhd \{\ y : E \mid P\ \} = U}$$

Car $\Pi x : C.D \rhd E$ est une dérivation plus petite que $T \rhd U$.

– $\rhd$-Sum : De façon similaire au produit.

– $\rhd$-Proof :

$$\dfrac{\dfrac{S \rhd C}{S \rhd T = \{\ y : C \mid P\ \}} \qquad T \rhd U}{S \rhd U}$$

Encore une fois, par cas sur la dérivation de $\{\ y : C \mid P\ \} = T \rhd U$:

- $\rhd$-Conv : Trivial.
- $\rhd$-Subset : On a :

$$\dfrac{C \rhd U}{\{\ y : C \mid P\ \} = T \rhd U}$$

Par induction, on obtient directement $S \rhd U$ avec les dérivations de $S \rhd C$ et $C \rhd U$.

- $\rhd$-Proof : On a :

$$\dfrac{T \rhd D}{\{\ y : C \mid P\ \} = T \rhd \{\ y : D \mid Q\ \} = U}$$

On a donc une dérivation de $S \rhd D$ par application de l'hypothèse d'induction. On en déduit une dérivation de $S \rhd \{\ y : D \mid Q\ \} = U$ par $\rhd$-Proof.

– $\rhd$-Subset : De même.

□

Corollaire 2.3.20 (Complétude de la coercion)**.** *Si* $\Gamma \vdash U \rhd V : s$ *alors* $\Gamma \vdash_\bullet U \rhd V : s$.

Démonstration. Les règles des deux systèmes sont les mêmes excepté $\rhd$-Trans qui est admissible dans le système algorithmique. De plus l'application restreinte de la conversion ne change pas les jugements dérivables (lemme 2.3.15). □

En conséquence les systèmes de coercion algorithmique et déclaratif sont équivalents. Le système d'inférence décrit par la relation $\_ \vdash_\bullet \_ \rhd \_ : \_$ donne donc un algorithme pour décider de la relation de coercion. L'indéterminisme entre les règles $\rhd$-Proof et $\rhd$-Subset ne pose pas de problème : on peut laisser le choix à l'implantation puisque le système est confluent. $\rhd$-$\downarrow$ formalise le fait qu'on peut avoir à réduire en tête avant d'appliquer les autres règles (pour obtenir un produit, une somme ou un sous-ensemble).

Décidabilité et complétude du typage

Le système algorithmique correspond au système déclaratif où l'on a enlevé la règle de coercion Coerce et changé certaines règles pour obtenir un système décidable (voir figure 2.7). On va procéder de façon similaire à l'élimination de la transitivité pour montrer que la règle Coerce n'est plus nécessaire dans le système algorithmique. On va montrer en fait que toute dérivation de typage utilisant Coerce peut se réécrire en une dérivation n'utilisant cette règle qu'à sa racine.

Élimination de la coercion On veut maintenant montrer la complétude de notre système. Dans un système à sous-typage, le théorème correspondant est parfois nommé typage minimal "*minimal typing*" puisque son énoncé revient à dire que tout terme a un type minimal dans les deux systèmes. En effet notre théorème est le suivant : si $\Gamma \vdash t : T$ alors il existe U tel que $\Gamma \vdash_\bullet t : U \rhd T$. Le typage algorithmique assigne bien un seul type à un terme t mais comme on a des coercions, le type inféré U peut être un peu différent du type T. Dans notre cas particulier U sera peut-être un type moins riche que T (par exemple `nat` par rapport à $\{ x : \mathsf{nat} \mid P \}$). Lorsque l'on développera des programmes, on donnera une spécification forte et l'on fera une coercion entre le type inféré et la spécification pour obtenir au final (après réécriture dans `Coq`) un terme du type T le plus riche. On a besoin de quelques lemmes pour montrer que notre système où la règle Coerce a été éliminée est complet :

Lemme 2.3.21 ($\mu_\bullet()$ et types produits et sommes)**.** *Si $X \rhd Y$ et $\mu_\bullet(Y) = \Sigma x : T.U$ alors $\mu_\bullet(X) = \Sigma x : T'.U'$ et $T' \rhd T$, $U' \rhd U$. Si $X \rhd Y$ et $\mu_\bullet(Y) = \Pi x : T.U$ alors $\mu_\bullet(X) = \Pi x : T'.U'$ et $T \rhd T'$, $U' \rhd U$.*

Démonstration. Par induction sur la dérivation de coercion, on fait le cas pour Σ.

- $\rhd$-Conv : Trivial, puisqu'on aura $\mu_\bullet(X) = \mu_\bullet(Y)$.
- $\rhd$-$\downarrow$: Trivial puisque pour tout x, $\mu_\bullet(x) = \mu_\bullet(x^\downarrow)$.
- $\rhd$-Prod : Impossible, $\mu_\bullet()$ ne traversant pas les produits.
- $\rhd$-Sum : Direct, on a une dérivation de $\Sigma x : T'.U' \rhd \Sigma x : T.U$.
- $\rhd$-Proof : Ici, $Y \equiv \{ x : V \mid P \}$, on peut donc déduire que $\mu_\bullet(Y) = \mu_\bullet(V) = \Sigma x : T.U$. On applique l'hypothèse de récurrence avec $X \rhd V$ et on obtient : $\mu_\bullet(X) = \Sigma x : T'.U' \wedge T' \rhd T \wedge U' \rhd U$.
- $\rhd$-Subset : Ici, $X \equiv \{ x : V \mid P \}$. Par induction, $\mu_\bullet(V) = \mu_\bullet(X) = \Sigma x : T'.U' \wedge T' \rhd T \wedge U' \rhd U$.

□

Il nous faut montrer des lemmes faisant intervenir la substitution pour pouvoir prouver la complétude.

Lemme 2.3.22 (Substitutivité de $\mu_\bullet()$)**.** *Si $\mu_\bullet(T) = \Pi y : U.V$ alors $\mu_\bullet(T[u/x]) = \Pi y : U[u/x].V[u/x]$. Si $\mu_\bullet(T) = \Sigma y : U.V$ alors $\mu_\bullet(T[u/x]) = \Sigma y : U[u/x].V[u/x]$.*

Démonstration. On montre la propriété pour les produits, la preuve est similaire pour les sommes. Par induction sur le nombre de constructeurs $\Pi, \Sigma, \{|\}$ dans la forme normale complète de T.

Il suffit de suivre la définition de $\mu_\bullet()$.

- Si $T^\downarrow$ est de la forme $\{ y : T' \mid P \}$ alors on a $\mu_\bullet(T') = \Pi y : U.V$ et par induction $\mu_\bullet(T'[u/x]) = \Pi y : U[u/x].V[u/x]$. Il s'ensuit directement que $\mu_\bullet(T[u/x]) = \Pi y : U[u/x].V[u/x]$.
- Si $T^\downarrow$ est différent d'un type sous-ensemble alors $\mu_\bullet(T) = T^\downarrow$. On a alors $T^\downarrow = \Pi y : U.V$ et donc $T[u/x]^\downarrow = \Pi y : U[u/x].V[u/x] = \mu_\bullet(T[u/x])$.

□

Lemme 2.3.23 (Substitutivité du typage)**.** *Si $\Gamma \vdash_\bullet u : U$ alors*

$$\begin{cases} \Gamma, x : U, \Delta \vdash_\bullet t : T & \Rightarrow \quad \Gamma, \Delta[u/x] \vdash_\bullet t[u/x] : T[u/x] \\ \vdash \Gamma, x : U, \Delta & \Rightarrow \quad \vdash \Gamma, \Delta[u/x] \\ \Gamma, x : U, \Delta \vdash_\bullet U \rhd T : s & \Rightarrow \quad \Gamma, \Delta[u/x] \vdash_\bullet U[u/x] \rhd T[u/x] : s \end{cases}$$

Démonstration. Par induction mutuelle sur les dérivations de typage, bonne formation et coercion.

– Wf-Empty : Trivial.

– Wf-Var : Par induction sur Δ.

- $\Delta = []$: On a alors $\Gamma \vdash_\bullet U : s$ donc $\vdash \Gamma$ et trivialement, $\vdash \Gamma, \Delta[u/x]$.
- $\Delta = \Delta', y : T$: On a alors $\Gamma, x : U, \Delta' \vdash_\bullet T : s$ et par induction $\Gamma, \Delta'[u/x] \vdash_\bullet T[u/x] : s[u/x] = s$. Donc on peut appliquer Wf-Var pour obtenir $\vdash \Gamma, \Delta'[u/x], y : T[u/x]$ soit $\vdash \Gamma, \Delta[u/x]$

– PropSet : La substitution n'a aucun effet et $\Gamma, \Delta[u/x]$ est bien formé par induction.

– Var : Par induction, $\vdash \Gamma, \Delta[u/x]$. Si $t \equiv x$ alors on a $T = U$ et $T[u/x] = U$ puisque x n'apparaît pas dans U. On a donc $\Gamma, \Delta[u/x] \vdash_\bullet t[u/x] = u : T[u/x] = U$, qui peut s'obtenir par affaiblissement de $\Gamma \vdash_\bullet u : U$. Si $y : T \in \Gamma$ alors on applique simplement Var. Si $y : T \in \Delta$ alors $y : T[u/x] \in \Delta[u/x]$ et on obtient $\Gamma, \Delta[u/x] \vdash_\bullet y[u/x] : T[u/x]$ par Var.

– Prod : Par induction $\Gamma, \Delta[u/x] \vdash_\bullet T[u/x] : s_1[u/x]$ et $\Gamma, \Delta[u/x], y : T[u/x] \vdash_\bullet M[u/x] : s_2[u/x]$. On peut appliquer Prod pour obtenir $\Gamma, \Delta[u/x] \vdash_\bullet \Pi y : T[u/x].M[u/x] : s_2$ soit $\Gamma, \Delta[u/x] \vdash_\bullet (\Pi y : T.M)[u/x] : s_2$. De façon similaire pour les autres constructeurs de types.

– App : On étudie le cas de l'application qui requiert un lemme supplémentaire. Par induction, on a $\Gamma, \Delta[u/x] \vdash_\bullet f[u/x] : T[u/x]$ et $\Gamma, \Delta[u/x] \vdash_\bullet a[u/x] : A[u/x]$. Si $\mu_\bullet(T) = \Pi y : V.W$ alors $\mu_\bullet(T[u/x]) = \Pi y : V[u/x].W[u/x]$ (lemme 2.3.22). Par induction, on a aussi $\Gamma, \Delta[u/x] \vdash_\bullet A[u/x], V[u/x] : s$. Enfin, par substitutivité de la coercion on a $\Gamma, \Delta[u/x] \vdash_\bullet A[u/x] \rhd V[u/x] : s$. On peut donc appliquer App pour obtenir $\Gamma, \Delta[u/x] \vdash_\bullet (f[u/x]\ a[u/x]) : W[u/x][a[u/x]/y]$. Or $W[u/x][a[u/x]/y] = W[a/y][u/x]$ $(y \notin \mathcal{FV}(u))$. On a donc bien $\Gamma, \Delta[u/x] \vdash_\bullet (f\ a)[u/x] : (W[a/y])[u/x]$.

– Pi-1 : Par induction on a $\Gamma, \Delta[u/x] \vdash_\bullet t[u/x] : S[u/x]$, et par substitutivité de $\mu_\bullet()$ on a aussi $\mu_\bullet(S[u/x]) = \Sigma y : T[u/x].U[u/x]$. Il suffit alors d'appliquer Pi-1

– Pi-2 : De même on se retrouve avec $\Gamma, \Delta[u/x] \vdash_\bullet t[u/x] : S[u/x]$, et $\mu_\bullet(S[u/x]) = \Sigma y : T[u/x].U[u/x]$. Il suffit alors d'appliquer Pi-2 pour obtenir :

$$\Gamma, \Delta[u/x] \vdash_\bullet \pi_2\ t[u/x] : U[u/x][\pi_1\ t[u/x]/y] = U[\pi_1\ t/y][u/x] \text{ car } y \notin \mathcal{FV}(u)$$

– Pair : On a :

$$\frac{\Gamma, x : V, \Delta \vdash_\bullet \Sigma y : T.V : s \qquad \begin{array}{r}\Gamma, x : V, \Delta \vdash_\bullet t : T' \rhd T : s \\ \Gamma, x : V, \Delta \vdash_\bullet v : V' \rhd V[t/y] : s\end{array}}{\Gamma, x : V, \Delta \vdash_\bullet (t, v)_{\Sigma y:T.V} : \Sigma y : T.V}$$

Par induction et application de Pair :

$$\frac{\begin{array}{c}\Gamma, \Delta[u/x] \vdash_\bullet \Sigma y : T[u/x].V[u/x] : s \\ \Gamma, \Delta[u/x] \vdash_\bullet t[u/x] : T'[u/x] \rhd T[u/x] : s \\ \Gamma, \Delta[u/x] \vdash_\bullet v[u/x] : V'[u/x] \rhd V[u/x][t[u/x]/y] : s\end{array}}{\Gamma, \Delta[u/x] \vdash_\bullet (t[u/x], v[u/x])_{\Sigma y:T[u/x].V[u/x]} : \Sigma y : T[u/x].V[u/x]}$$

On a bien $V[u/x][t[u/x]/y] = V[t/y][u/x]$ car $y \notin \mathcal{FV}(u)$.

– $\rhd$-Conv : Direct par préservation de l'équivalence $\equiv_{\beta\pi}$ par substitution et application de l'hypothèse d'induction pour le typage.

– $\rhd$-$\downarrow$: Par induction, $(U^{\downarrow})[u/x] \rhd (T^{\downarrow})[u/x]$. Par le lemme 2.3.18, $((U^{\downarrow})[u/x])^{\downarrow} \rhd ((T^{\downarrow})[u/x])^{\downarrow}$. Donc $U[u/x]^{\downarrow} \rhd T[u/x]^{\downarrow}$ et par $\rhd$-$\downarrow$, $U[u/x] \rhd T[u/x]$.

– $\rhd$-Prod : Par induction $U[u/x] \rhd T[u/x]$ et $V[u/x] \rhd W[u/x]$, donc par $\rhd$-Prod :

$$\Pi y : T[u/x].V[u/x] \rhd \Pi y : U[u/x].W[u/x]$$

– $\rhd$-Sum : Direct par induction.

– $\rhd$-Subset : Par induction, $U'[u/x] \rhd V[u/x]$. On applique $\rhd$-Left pour obtenir $\{\ y : U'[u/x] \mid P\ \} \rhd V[u/x]$.

– $\rhd$-Right : Direct par induction.

□

Corollaire 2.3.24 (Substitutivité du typage avec coercion)**.** *Si* $\Gamma, x : V \vdash_\bullet t : T \rhd U$ *et* $\Gamma \vdash_\bullet u : V$ *alors* $\Gamma \vdash t[u/x] : T[u/x] \rhd U[u/x]$*.*

On a maintenant tout les ingrédients pour montrer la complétude de notre système de typage vis-à-vis du système déclaratif.

Théorème 2.3.25 (Complétude du typage)**.** *Si* $\Gamma \vdash T : s$ *alors* $\Gamma \vdash_\bullet T : s$*. Si* $\Gamma \vdash t : T$ *alors* $\exists U s, \Gamma \vdash_\bullet t : U \rhd T : s$*. Si* $\vdash \Gamma$ *alors* $\vdash_\bullet \Gamma$*.*

Démonstration. Par induction mutuelle sur les dérivations de typage et bonne formation.

– Wf-Empty : Trivial.

– Wf-Var :

$$\frac{\Gamma \vdash A : s}{\vdash \Gamma, x : A}\ s \in \mathcal{S} \wedge x \notin \Gamma$$

Par induction $\exists s', \Gamma \vdash_\bullet A : s' \rhd s$. On a forcément $s' = s$ puisque les sortes ne sont en relation qu'avec elles-mêmes. On applique Wf-Var pour obtenir $\vdash \Gamma, x : A$.

– PropSet : Trivial.

– Var :

$$\frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A}$$

Par induction $\vdash \Gamma$ et $x : A \in \Gamma$, direct par Var. On vérifie que si A est une sorte on dérive bien la même sorte dans le système algorithmique.

– Prod :

$$\frac{\Gamma \vdash T : s_1 \qquad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash \Pi x : T.U : s_3}\ (s_1, s_2, s_3) \in \mathcal{R}$$

Direct par induction et le fait que $\mathcal{R}$ est fonctionnelle.

– Abs :

$$\frac{\Gamma \vdash \Pi x : T.U : s \qquad \Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x : T.M : \Pi x : T.U}$$

Par induction $\exists U', \Gamma, x : T \vdash_\bullet M : U' \rhd U$ et $\Gamma, x : T \vdash_\bullet U', U : s$. On peut donc dériver $\Gamma \vdash_\bullet \Pi x : T.U' : s$. On a bien $\Gamma \vdash_\bullet \lambda x : T.M : \Pi x : T.U' \rhd \Pi x : T.U$ et les deux types ont la même sorte.

– App : On a

$$\frac{\Gamma \vdash f : \Pi x : V.W \qquad \Gamma \vdash u : V}{\Gamma \vdash (f u) : W[u/x]}$$

Par induction, $\exists T, \Gamma \vdash_\bullet f : T \rhd \Pi x : V.W$ et $\exists U, \Gamma \vdash_\bullet u : U \rhd V$.
Si $T \rhd \Pi x : V.W$ alors $\mu_\bullet(T) = \Pi x : V'.W'$ avec $V \rhd V'$ et $W' \rhd W$ (lemme 2.3.21).
Par transitivité de la coercion : $U \rhd V'$, on peut donc dériver

$$\frac{\Gamma \vdash_\bullet f : T \quad \mu_\bullet(T) = \Pi x : V'.W' \qquad \Gamma \vdash_\bullet u : U \quad \Gamma \vdash_\bullet U, V' : s \qquad U \rhd V'}{\Gamma \vdash_\bullet (fu) : W'[u/x]}$$

Par substitutivité de la coercion (lemme 2.3.23), $W'[u/x] \rhd W[u/x]$, la propriété est donc bien vérifiée. Les conditions de sortes sont vérifiées du fait que $\mu_\bullet()$ conserve les sortes, donc $\Gamma \vdash_\bullet T, \Pi x : V'.W', \Pi x : V.W : s$ puis par inversion, $\Gamma, x : V \vdash_\bullet W', W : s$ et enfin par substitutivité, $\Gamma \vdash_\bullet W'[u/x], W[u/x] : s$.

– Sum :

$$\frac{\Gamma \vdash T : s \qquad \Gamma, x : T \vdash U : s}{\Gamma \vdash \Sigma x : T.U : s}\ s \in \{\texttt{Prop}, \texttt{Set}\}$$

Par induction $\Gamma \vdash_\bullet T : s$ et $\Gamma, x : T \vdash_\bullet U : s$ où $s \in \{\texttt{Prop}, \texttt{Set}\}$. C'est direct par Sum.

– Pair :

$$\frac{\Gamma \vdash \Sigma x : T.U : s \qquad \Gamma \vdash t : T \qquad \Gamma \vdash u : U[t/x]}{\Gamma \vdash (t, u)_{\Sigma x:T.U} : \Sigma x : T.U}$$

Ici, l'annotation nous force à utiliser le jugement de coercion. Par induction, $\Sigma x : T.U : s$, $\exists T', \Gamma \vdash_\bullet t : T' \rhd T$ et $\exists U', \Gamma \vdash_\bullet u : U' \rhd U[t/x]$. On peut montrer $\Gamma \vdash_\bullet \Sigma x : T'.U : s$. En effet, par inversion de $\Gamma \vdash_\bullet \Sigma x : T.U : s$ on a $\Gamma, x : T \vdash_\bullet U : s$ et par le lemme 2.3.14 ($T' \rhd T$), $\Gamma, x : T' \vdash_\bullet U : s$. Comme $T' \rhd T$ on obtient $\Sigma x : T'.U \rhd \Sigma x : T.U$. On peut donc dériver :

$$\frac{\begin{array}{l}\Gamma \vdash_\bullet t : T' \\ \Gamma \vdash_\bullet u : U' \qquad \Gamma \vdash_\bullet U[t/x], U' : s \quad U' \rhd U[t/x] \qquad \Gamma \vdash_\bullet \Sigma x : T'.U : s\end{array}}{\Gamma \vdash_\bullet (x :=\ t, u : U) : \Sigma x : T'.U}$$

– Pi-1, Pi-2 : On a

$$\frac{\Gamma \vdash t : \Sigma x : T.U}{\Gamma \vdash \pi_1\ t : T}$$

Par induction, $\exists T', \Gamma \vdash_\bullet t : T' \rhd \Sigma x : T.U$:. On en déduit que $\mu_\bullet(T') = \Sigma x : T'.U'$ avec $\Gamma \vdash_\bullet T' \rhd T : s_1$ et $\Gamma, x : T' \vdash_\bullet U' \rhd U : s_2$. Clairement, $\Gamma \vdash_\bullet \pi_1\ t : T' \rhd T : s_1$ et $\Gamma \vdash_\bullet \pi_2\ t : U'[\pi_1\ t/x] \rhd U[\pi_1\ t/x] : s_2$ par substitutivité de la coercion.

– Conv, Coerce : Dans les deux cas on a inductivement $\exists T', \Gamma \vdash_\bullet t : T' \rhd T$. Avec Conv on a $T \equiv_{\beta\pi} S$, donc $T' \rhd S$ par le lemme 2.3.17. Pour Coerce on a $T \rhd S$. Par complétude de la coercion, $T \rhd S$ et par transitivité de la coercion, $T' \rhd S$. La propriété est donc bien vérifiée dans les deux cas.

□

On combine les théorèmes de correction et de complétude pour obtenir la propriété suivante entre les deux systèmes :

Corollaire 2.3.26 (Équivalence des systèmes déclaratif et algorithmique)**.** *$\Gamma \vdash t : T$ si et seulement si il existe U tel que $\Gamma \vdash_\bullet t : U$ et $U \rhd T$.*

On a maintenant un système raffiné dérivant les mêmes jugements (à coercion près) que le système déclaratif. On veut en extraire un algorithme de typage. Pour cela on doit pouvoir résoudre deux problèmes :

– **Vérification de type.** On donne Γ,t et T et l'on doit décider si $\Gamma \vdash_\bullet t : T$;

– **Inférence de type.** On donne Γ,t et l'on doit trouver T tel que $\Gamma \vdash_\bullet t : T$ si c'est dérivable, sinon on échoue.

En pratique, la vérification a besoin de l'inférence puisque lorsqu'on vérifie une application $(f\ u) : T$ on doit inférer le type de f. On montre donc les théorèmes suivants :

Théorème 2.3.27 (Décidabilité de l'inférence dans le système algorithmique)**.** *Le problème d'inférence* $\Gamma \vdash_\bullet t : ?$ *est décidable.*

Démonstration. Il suffit d'observer que les règles de typage sont dirigées par la syntaxe du deuxième argument et permettent donc d'inférer un type pour tout terme. En lisant les prémisses de chaque règle de gauche à droite, on voit que l'inférence est décidable. □

Théorème 2.3.28 (Décidabilité de $\vdash_\bullet$)**.** *La relation de typage* $\Gamma \vdash_\bullet t : T$ *est décidable.*

Démonstration. Direct. On utilise le théorème précédent pour le cas de l'application. □

On a désormais un algorithme de typage pour notre système avec coercions. Ce système est très libéral puisqu'il permet de considérer des objets comme vérifiant des propriétés arbitraires sans les montrer. Il nous faut maintenant remettre de la logique dans nos termes pour s'assurer qu'ils sont corrects.

Chapitre 3

Interprétation

Sommaire

On veut désormais traduire les dérivations du système algorithmique dans CCI dont le jugement de typage est $\vdash_?$. Les termes de Russell ne sont pas directement typables dans CCI puisque nous avons permis d'utiliser des objets comme s'ils avaient des types différents de leurs types originaux avec la règle de coercion. Il va donc falloir maintenant expliciter ces coercions pour obtenir des termes typables dans CCI. Cependant, on ne peut pas créer un terme complet à partir de notre dérivation, puisqu'on ne peut pas inférer des preuves arbitraires. On utilise donc des existentielles (intuitivement des trous dont on ne connait que le type des habitants) pour traduire le fait qu'il est de la responsabilité de l'utilisateur de prouver que son utilisation de la coercion n'était pas incorrecte.

3.1 Génération des obligations de preuve

On définit l'interprétation $[\![t]\!]_\Gamma$ par récurrence sur la forme des termes (figure 3.3). Le principe est simplement de traduire le terme t en un terme t' en insérant des coercions pour renvoyer un terme équivalent typable dans CCI. Cette interprétation construit donc un terme t' réécrit que l'on montrera bien typé dans l'environnement CCI $[\![\Gamma]\!]$.

La difficulté essentielle est que dans ce système les termes apparaissent dans les types et il est donc primordial que lorsqu'on traduit un type on obtienne un type essentiellement équivalent en Coq, malgré les coercions. Cela va nous ammener à vérifier des propriétés structurelles fortes sur les termes de coercion, notamment l'unicité, la réflexivité, la symétrie et la transitivité. Comme nous travaillons avec des types dépendants et modulo coercion dans les contextes, la plupart des propriétés doivent être généralisées pour permettre de faire de l'afflaiblissement à volonté (par exemple, lemme 3.2.12).

Définition 3.1.1 (Interprétation des contextes)**.** *On fait l'extension aux contextes de la façon suivante :*

- $[\![[]]\!] = []$
- $[\![\Gamma, x : T]\!] = [\![\Gamma]\!], x : [\![T]\!]_\Gamma$

Chaque jugement de coercion du système algorithmique permet de dériver une coercion explicite qui sera directement appliquée à un objet.

On formalise donc les coercions par des contextes à trous multiples.

Définition 3.1.2 (Coercion)**.** *Une coercion est un terme formé à partir de la grammaire originale des termes à laquelle on ajoute un terminal* $\bullet$*.*

Définition 3.1.3 (Substitution et composition de coercions)**.** *La substitution dans une coercion est notée* $c[d]$*, elle remplace toutes les occurrences de* $\bullet$ *dans* c *par* d *en évitant les captures.*

La composition de deux coercions est notée $c \circ d \triangleq c[d]$*, son élément neutre est* $\bullet$*.*

La substitution d'un terme pour une variable dans une coercion est notée $c[t/x]$ *comme pour les termes.*

3.1.1 Coercions explicites

On définit le jugement $\Gamma \vdash_? c : S \rhd T : s$ (figure 3.2) qui dérive une coercion à partir de deux types S et T dans un environnement Γ. On a introduit du déterminisme par rapport au jugement de coercion algorithmique puisqu'on donne priorité à la règle $\rhd$-Subset par rapport à la règle $\rhd$-Proof (ces règles sont confluentes comme nous le montrerons dans le lemme 3.2.1). On explicite aussi la priorité donnée à la mise en forme normale de tête (figure 2.6) puis à la dérivation par rapport au test de conversion dans la prémisse de $\rhd$-Conv. Le système a de bonnes propriétés pour la preuve et l'implémentation telles que l'unicité et l'admissibilité de la transitivité tout comme le jugement algorithmique. Nous montrerons quelles coercions résultent de ces propriétés.

Notations Pour alléger la présentation, on ignorera les sortes et les conditions de sortage du jugement de coercion dans la suite. On écrira donc simplement $\Gamma \vdash_? c : T \rhd U$. Il est clair qu'on peut écrire un jugement équivalent sans les sortes qui coincide lorsque les types sont bien sortés. On peut donc supposer que les types sont bien formés dans toute sous-dérivation de coercion entre types bien formés.

Comme le jugement de dérivation de la coercion est une simple décoration du jugement de coercion algorithmique qui est lui-même dirigé par la syntaxe, on peut le voir comme une fonction partielle. On note donc $\mathbf{coerce}_\Gamma\ T\ U$ la coercion obtenue en dérivant le jugement $\Gamma \vdash_? ? : T \rhd U$. De même, la fonction de typage algorithmique est notée $\mathbf{type}_\Gamma(t)$.

On note $T^\downarrow$ la forme normale de tête de T (définie figure 2.6) et $T\!\downarrow$ la forme normale de T.

Théorie équationnelle La théorie équationnelle du langage cible $\mathsf{CC}_?$ étend la théorie équationnelle du CC avec sommes :

On définit l'équivalence $\equiv_{\beta\pi\eta\rho\sigma}$ comme la clôture réflexive, symétrique et transitive de la relation définie figure 3.1. Cette relation sera dénotée par $\equiv$ pour plus de clarté. Elle contient la β-réduction et les projections pour les sommes dépendantes qui incluent ici les types sous-ensemble. Les objets de types sous-ensembles sont distingués par leur étiquette sur le constructeur de paires. On ajoute aussi des relations nécessaires pour supporter l'interprétation des termes de Russell dans $\mathsf{CC}_?$. On a donc la règle η pour l'abstraction

$$
\begin{array}{llcll}
(\beta) & (\lambda x : X.e)\ v & \equiv & e[v/x] & \\
(\pi_i) & \pi_i\ (e_1, e_2)_\tau & \equiv & e_i & \\
(\eta) & (\lambda x : X.e\ x) & \equiv & e & \text{si } x \notin FV(e) \\
(\rho) & (\pi_1\ e, \pi_2\ e)_\tau & \equiv & e & \\
(\sigma) & (t, p)_{\{x:E|P\}} & \equiv & (t', p')_{\{x:E|P\}} & \text{si } t \equiv t'
\end{array}
$$

FIG. 3.1: Théorie équationnelle de $\mathsf{CC}_?$

et les règles ρ pour le "*surjective pairing*" qui s'applique aux sommes dépendantes et aux objets de type sous-ensemble. Ces règles standards sont nécessaires lorsqu'on fait des traductions et qu'on génère des formes η-longues. Enfin la règle σ est plus inhabituelle. C'est une forme limitée d'indifférence aux preuves pour les objets de type sous-ensemble : deux objets d'un même type sous-ensemble sont équivalents si et seulement si les témoins sont équivalents.

Dans $\mathsf{CC}_?$, on a aussi une règle de typage supplémentaire pour typer les existentielles :

$$\textsc{Evar}\ \frac{\Gamma \vdash_? P : \mathtt{Prop}}{\Gamma \vdash_? ?_P : P}$$

De façon classique, la substitution se propage dans les existentielles : $?_P[t/x] \triangleq ?_{P[t/x]}$.

$$\frac{T \equiv_{\beta\pi} U \quad \Gamma \vdash_\bullet T, U : s}{\Gamma \vdash_? \bullet : T \rhd U : s}\ T = T^\downarrow \wedge T \neq \Pi, \Sigma, \{|\} \wedge U = U^\downarrow$$

$$\frac{\Gamma \vdash_? c : T^\downarrow \rhd U^\downarrow : s}{\Gamma \vdash_? c : T \rhd U : s}\ T \neq T^\downarrow \vee U \neq U^\downarrow$$

$$\frac{\Gamma \vdash_? c_1 : U \rhd T : s_1 \qquad \Gamma, x : U \vdash_? c_2 : V \rhd W : s_2}{\Gamma \vdash_? \lambda x : [\![U]\!]_\Gamma . c_2[\bullet\ (c_1[x])] : \Pi x : T.V \rhd \Pi x : U.W : s_3}\ (s_1, s_2, s_3) \in \mathcal{R}$$

$$\frac{\Gamma \vdash_? c_1 : T \rhd U : s \qquad \Gamma, x : T \vdash_? c_2 : V \rhd W : s}{\Gamma \vdash_? (c_1[\pi_1\ \bullet], c_2[\pi_2\ \bullet][\pi_1\ \bullet\ /x])_{[\![\Sigma x:U.W]\!]_\Gamma} : \Sigma x : T.V \rhd \Sigma x : U.W : s}\ s \in \{\mathtt{Prop}, \mathtt{Set}\}$$

$$\frac{\Gamma \vdash_? c : U \rhd T : \mathtt{Set} \qquad \Gamma \vdash_\bullet \{\ x : U \mid P\ \} : \mathtt{Set}}{\Gamma \vdash_? c[\pi_1\ \bullet] : \{\ x : U \mid P\ \} \rhd T : \mathtt{Set}}\ T = T^\downarrow$$

$$\frac{\Gamma \vdash_? c : T \rhd U : \mathtt{Set} \qquad \Gamma \vdash_\bullet \{\ x : U \mid P\ \} : \mathtt{Set}}{\Gamma \vdash_? (c, ?_{[\![P]\!]_{\Gamma, x:U}[c/x]})_{[\![\{x:U|P\}]\!]_\Gamma} : T \rhd \{\ x : U \mid P\ \} : \mathtt{Set}}\ T = T^\downarrow$$

FIG. 3.2: Réécriture de la coercion vers CCI

On veut montrer que si l'on a un jugement valide dans notre système algorithmique, alors son image par l'interprétation est un jugement valide de $\mathsf{CC}_?$. On rappelle que dans $\mathsf{CC}_?$ la règle de coercion est remplacée par la règle de conversion.

3.1.2 Coercions et dépendance

Notre problème se ramène à montrer le théorème suivant :

$$\Gamma \vdash_\bullet t : T \Rightarrow [\![\Gamma]\!] \vdash_? [\![t]\!]_\Gamma : [\![T]\!]_\Gamma$$

$$
\begin{array}{lcll}
[\![x]\!]_\Gamma & = & x & \\
[\![s]\!]_\Gamma & = & s & s \in \{\mathtt{Set}, \mathtt{Prop}, \mathtt{Type}\} \\
[\![\Pi x : T.U]\!]_\Gamma & = & \Pi x : [\![T]\!]_\Gamma.[\![U]\!]_{\Gamma,x:T} & \\
[\![\lambda x : \tau.v]\!]_\Gamma & = & \mathtt{let}\ \tau' = [\![\tau]\!]_\Gamma\ \mathtt{in} & \\
 & & \mathtt{let}\ v' = [\![v]\!]_{\Gamma,x:\tau}\ \mathtt{in} & \\
 & & \quad (\lambda x : \tau'.v') & \\
[\![f\ u]\!]_\Gamma & = & \mathtt{let}\ F\ = \mathbf{type}_\Gamma(f)\ \mathtt{and}\ U = \mathbf{type}_\Gamma(u)\ \mathtt{in} & \\
 & & \mathtt{let}\ (\Pi x : V.W) = \mu_\bullet(F)\ \mathtt{in} & \\
 & & \mathtt{let}\ \pi = \mathbf{coerce}_\Gamma\ F\ (\Pi x : V.W)\ \mathtt{in} & \\
 & & \mathtt{let}\ c = \mathbf{coerce}_\Gamma\ U\ V\ \mathtt{in} & \\
 & & \quad (\pi[[\![f]\!]_\Gamma])\ (c[[\![u]\!]_\Gamma]) & \\
[\![\Sigma x : T.U]\!]_\Gamma & = & \Sigma x : [\![T]\!]_\Gamma.[\![U]\!]_{\Gamma,x:T} & \\
[\![(t,u)_{\Sigma x:T.U}]\!]_\Gamma & = & \mathtt{let}\ t' = [\![t]\!]_\Gamma\ \mathtt{in} & \\
 & & \mathtt{let}\ T' = \mathbf{type}_\Gamma(t)\ \mathtt{in} & \\
 & & \mathtt{let}\ ct = \mathbf{coerce}_\Gamma\ T'\ T\ \mathtt{in} & \\
 & & \mathtt{let}\ U' = \mathbf{type}_\Gamma(u)\ \mathtt{in} & \\
 & & \mathtt{let}\ u' = [\![u]\!]_\Gamma\ \mathtt{in} & \\
 & & \mathtt{let}\ cu = \mathbf{coerce}_\Gamma\ U'\ U[t/x]\ \mathtt{in} & \\
 & & \quad (ct[t'], cu[u'])_{[\![\Sigma x:T.U]\!]_\Gamma} & \\
[\![\pi_i\ t]\!]_\Gamma & = & \mathtt{let}\ t' = [\![t]\!]_\Gamma\ \mathtt{in} & i \in \{1,2\} \\
 & & \mathtt{let}\ T = \mathbf{type}_\Gamma(t)\ \mathtt{in} & \\
 & & \mathtt{let}\ \Sigma x : V.W = \mu_\bullet(T)\ \mathtt{in} & \\
 & & \mathtt{let}\ c = \mathbf{coerce}_\Gamma\ T\ (\Sigma x : V.W)\ \mathtt{in} & \\
 & & \quad \pi_i\ c[t'] & \\
[\![\{\ x : U \mid P\ \}]\!]_\Gamma & = & \{\ x : [\![U]\!]_\Gamma \mid [\![P]\!]_{\Gamma,x:U}\ \} &
\end{array}
$$

FIG. 3.3: Interprétation dans CCI

Ce résultat ne se montre pas aisément. En effet le jugement de coercion rend la preuve très difficile à cause de son caractère non local : les termes coercés peuvent être substitués dans d'autres termes contenant déjà des coercions. Pour mieux comprendre ce problème, considérons l'exemple suivant :

Exemple Dans le système algorithmique, on peut très bien dériver $\Pi n : \mathtt{nat}.\ \mathtt{list}\ n \rhd \Pi n : \{\ x : \mathtt{nat} \mid P\ \}.\mathtt{list}\ n$ puisque $\{\ x : \mathtt{nat} \mid P\ \} \rhd \mathtt{nat}$ et $\mathtt{list}\ n \equiv_{\beta\pi} \mathtt{list}\ n$. Si l'on interprète ces deux types, une coercion va être insérée dans le second type : $[\![\Pi n : \{\ x : \mathtt{nat} \mid P\ \}.\ \mathtt{list}\ n]\!]_\Gamma = \Pi n : \{\ x : \mathtt{nat} \mid P\ \}.\ \mathtt{list}\ (\pi_1\ n)$. La coercion générée doit donc avoir pour type : $\Pi n : \mathtt{nat}.\mathtt{list}\ n \to \Pi n : \{\ x : \mathtt{nat} \mid P\ \}.\ \mathtt{list}\ (\pi_1\ n)$, mais elle est dérivée en se basant seulement sur les types algorithmiques. On peut vérifier ici que l'intuition de la coercion par prédicats est bonne, puisqu'on peut dériver ce jugement :

$$\dfrac{\dfrac{\dfrac{\mathsf{nat} \equiv_{\beta\pi} \mathsf{nat}}{\Gamma' \vdash_? \bullet : \mathsf{nat} \rhd \mathsf{nat}}}{\Gamma' \vdash_? \pi_1\ \bullet : \{\ x : \mathsf{nat} \mid P\ \} \rhd \mathsf{nat}} \qquad \dfrac{\mathsf{list}\ n \equiv_{\beta\pi} \mathsf{list}\ n}{\Gamma, n : \{\ x : \mathsf{nat} \mid P\ \} \vdash_? \bullet : \mathsf{list}\ n \rhd \mathsf{list}\ n}}{\Gamma' \vdash_? \lambda x : [\![\{\ x : \mathsf{nat} \mid P\ \}]\!]_\Gamma.\ \bullet\ [\bullet\ (\pi_1\ x)] = \bullet\ (\pi_1 x) : \Pi n : \mathsf{nat}.\ \mathsf{list}\ n \rhd \Pi n : \{\ x : \mathsf{nat} \mid P\ \}.\ \mathsf{list}\ n}$$

Supposons $\Gamma \vdash_? t : \Pi n : \mathsf{nat}.\ \mathsf{list}\ n$ alors on a la dérivation de typage suivante :

$$\dfrac{\dfrac{\Gamma, x : \{\ x : \mathsf{nat} \mid [\![P]\!]_\Gamma\ \} \vdash_\bullet t : \Pi n : \mathsf{nat}.\ \mathsf{list}\ n \qquad \Gamma, x : \{\ x : \mathsf{nat} \mid [\![P]\!]_\Gamma\ \} \vdash_\bullet \pi_1\ x : \mathsf{nat}}{\Gamma, x : \{\ x : \mathsf{nat} \mid [\![P]\!]_\Gamma\ \} \vdash_\bullet t\ (\pi_1\ x) : \mathsf{list}\ (\pi_1\ x)}}{\Gamma \vdash_\bullet \lambda x : [\![\{\ x : \mathsf{nat} \mid P\ \}]\!]_\Gamma . t\ (\pi_1\ x) : \Pi n : [\![\{\ x : \mathsf{nat} \mid P\ \}]\!]_\Gamma.\ \mathsf{list}\ (\pi_1\ x)}$$

On crée donc bien un terme de type $[\![\Pi n : \{\ x : \mathsf{nat} \mid P\ \}.\ \mathsf{list}\ n]\!]_\Gamma$ en appliquant la coercion a un terme de type $[\![\Pi n : \mathsf{nat}.\ \mathsf{list}\ n]\!]_\Gamma$, c'est l'effet recherché.

3.2 Preuve de correction

Pour commencer, nous allons donc étudier les propriétés de la coercion. Nous verrons apparaitre la nécessité d'une propriété de symétrie au cours de la preuve, dont le système présenté plus haut jouit. Mais tout d'abord, nous montrons des propriétés structurelles de base : unicité, réflexivité et respect de la conversion.

3.2.1 Propriétés de la coercion

Lemme 3.2.1 (Unicité de la coercion)**.** *Si* $\Gamma \vdash_\bullet T, U : s$ *alors si* $\Gamma \vdash_? c : T \rhd U$ *et* $\Gamma \vdash_? c' : T \rhd U$ *alors* $c = c'$.

Démonstration. Par simple inspection des règles, on remarque qu'une seule règle s'applique suivant la forme de T, sauf si T et U sont des types sous-ensemble. On montre donc la confluence des deux règles $\rhd$-Subset et $\rhd$-Proof :

$$\dfrac{\dfrac{\Gamma \vdash_? c : T' \rhd U'}{\Gamma \vdash_? c[\pi_1\ \bullet] : \{\ x : T' \mid P\ \} \rhd U'}}{\Gamma \vdash_? c' : T = \{\ x : T' \mid P\ \} \rhd U = \{\ x : U' \mid Q\ \}}$$

avec

$$c' = (c[\pi_1\ \bullet], ?_{[\![Q]\!]_{\Gamma, x:U'}[c[\pi_1\ \bullet]/x]})_{[\![\{x:U' \mid Q\}]\!]_\Gamma}$$

D'autre part :

$$\dfrac{\dfrac{\Gamma \vdash_? c : T' \rhd U'}{\Gamma \vdash_? c'' : T' \rhd \{\ x : U' \mid Q\ \}}}{\Gamma \vdash_? c''[\pi_1\ \bullet] : T = \{\ x : T' \mid P\ \} \rhd U}$$

avec

$$c'' = (c, ?_{[\![Q]\!]_{\Gamma, x:U'}[c/x]})_{[\![\{x:U' \mid Q\}]\!]_\Gamma}$$

Clairement, $c''[\pi_1\ \bullet] = (c[\pi_1\ \bullet], ?_{[\![Q]\!]_{\Gamma, x:U'}[c[\pi_1\ \bullet]/x]})_{[\![\{x:U' \mid Q\}]\!]_\Gamma} = c'$.

La confluence locale suffit puisqu'il n'est pas possible d'appliquer d'autres règles que ces deux là si ces deux là sont applicables. □

On peut donc raisonner comme ceci : si l'on parvient à construire une dérivation de $\Gamma \vdash_? c : T \rhd U$ toute autre dérivation de $T \rhd U$ donne le même terme de coercion.

Lemme 3.2.2 (Réflexivité de la coercion). *Si* $\Gamma \vdash_\bullet A : s$ *alors il existe* c, $\Gamma \vdash_? c : A \rhd A$ *et* $c \equiv \bullet$.

Démonstration. Par induction sur le nombre de constructeurs $\Pi, \Sigma, \{|\}$ dans la forme normale de A.

S'il $A{\downarrow}$ n'a pas de constructeur $\Pi, \Sigma, \{|\}$ en tête, alors $A^{\downarrow}$ n'en a pas en tête et l'on peut directement appliquer $\rhd$-Conv.

Sinon, on va appliquer la ou les règles correspondant au constructeur en tête. Le cas le plus intéressant est si $A^{\downarrow}$ est de la forme $\{\ x : U \mid P\ \}$. Alors on a la dérivation :

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash_? c' \equiv \bullet : U \rhd U}{\Gamma \vdash_? d = c'[\pi_1\ \bullet] : \{\ x : U \mid P\ \} \rhd U}}{\Gamma \vdash_? c = (d, ?_{[\![P]\!]_{\Gamma, x:U}[d/x]})_{[\![\{x:U|P\}]\!]_\Gamma} : A^{\downarrow} = \{\ x : U \mid P\ \} \rhd \{\ x : U \mid P\ \}}}{\Gamma \vdash_? c : A \rhd A}$$

On obtient la dérivation de c' par induction, puisque $U{\downarrow}$ contient strictement moins de constructeurs que $A{\downarrow}$.

On a :

$$\begin{array}{rcl} c & \triangleq & (d, ?_{[\![P]\!]_{\Gamma,x:U}[d/x]})_{[\![\{x:U|P\}]\!]_\Gamma} \\ & \triangleq & ((\pi_1\ \bullet), ?_{[\![P]\!]_{\Gamma,x:U}[\pi_1\ \bullet/x]})_{[\![\{x:U|P\}]\!]_\Gamma} \\ & =_\sigma & ((\pi_1\ \bullet), (\pi_2\ \bullet))_{[\![\{x:U|P\}]\!]_\Gamma} \\ & =_\rho & \bullet \end{array}$$

On utilise ici l'indifférence aux preuves (σ). Comme $(\pi_2\ \bullet)$ est de type $[\![P]\!]_{\Gamma,x:U}[\pi_1\ \bullet/x]$ dans un contexte où $\bullet$ est de type $[\![\{\ x : U \mid P\ \}]\!]_\Gamma$, on peut remplacer directement l'existentielle par ce terme. En pratique, ces preuves pourront être directement déchargées par l'assistant de preuve. □

On étudie maintenant la relation entre coercion et conversion.

Lemme 3.2.3 (Coercion et formes normales de tête). *Si* $\Gamma \vdash_? c : T \rhd U$ *alors* $\Gamma \vdash_? c' : T^{\downarrow} \rhd U^{\downarrow}$ *avec* $c = c'$ *est dérivable par une dérivation plus petite ou égale.*

Démonstration. Par idempotence de la mise en forme normale de tête, on a la même dérivation dans le cas où la dernière règle appliquée était $\rhd$-$\downarrow$, sinon c'est trivial. □

On généralise la réflexivité aux termes convertibles.

Lemme 3.2.4 (Coercion de termes convertibles). *Si* $\Gamma \vdash_\bullet T, U : s$, $T \equiv_{\beta\pi} U$ *alors il existe* c, $\Gamma \vdash_? c : T \rhd U$ *avec* $c \equiv \bullet$.

Démonstration. Par induction sur le nombre de constructeurs $\Pi, \Sigma, \{|\}$ dans les formes normales de T et U.

– Si $T{\downarrow}$ n'a pas pour symbole de tête, Π, Σ ou $\{|\}$, alors $\rhd$-Conv est la seule règle applicable et on a $c = \bullet$.

– Si $T^{\downarrow} = \Pi y : A.B$, alors $U^{\downarrow} = \Pi y : A'.B'$ avec $A \equiv_{\beta\pi} A'$, $B \equiv_{\beta\pi} B'$. Par induction, $\Gamma \vdash_? c_1 \equiv \bullet : A' \rhd A$ et $\Gamma, x : A' \vdash_? c_2 \equiv \bullet : B \rhd B'$ ($A{\downarrow}, A'{\downarrow}, B{\downarrow}, B'{\downarrow}$ sont des sous-termes stricts de $T{\downarrow}$ et $U{\downarrow}$). On peut donc dériver : $\Gamma \vdash_? \lambda x : [\![A']\!]_\Gamma.c_2[\bullet\ c_1[x]] : \Pi y : A.B \rhd \Pi y : A'.B'$. Puis par application de $\rhd$-$\downarrow$, la coercion de T à U.

On a donc :

$$\begin{array}{rcl} c & \triangleq & \lambda x : [\![A']\!]_\Gamma . c_2[\bullet\ (c_1[x])] \\ & \equiv & \lambda x : [\![A']\!]_\Gamma . \bullet\ x \\ & =_\eta & \bullet \end{array}$$

– Si $T^\downarrow = \Sigma y : A.B$ alors $U^\downarrow = \Sigma y : A'.B'$ avec $A \equiv_{\beta\pi} A'$, $B \equiv_{\beta\pi} B'$. Par induction, $\Gamma \vdash_? c_1 \equiv \bullet : A \rhd A'$ et $\Gamma, y : A \vdash_? c_2 \equiv \bullet : B \rhd B'$ ($A\downarrow, A'\downarrow, B\downarrow, B'\downarrow$ sont des sous-termes stricts de $T\downarrow$ et $U\downarrow$). On peut donc dériver :

$$\Gamma \vdash_? (c_1[\pi_1\ \bullet], c_2[\pi_2\ \bullet][\pi_1\ \bullet / y])_{[\![\Sigma y:A'.B']\!]_\Gamma} : \Sigma y : A.B \rhd \Sigma y : A'.B'$$

On applique $\rhd$-$\downarrow$ pour obtenir la coercion de T à U.
On a donc :

$$\begin{array}{rcl} c & \triangleq & (c_1[\pi_1\ \bullet], c_2[\pi_2\ \bullet][\pi_1\ \bullet / y])_{[\![\Sigma y:A'.B']\!]_\Gamma} \\ & \equiv & (\pi_1\ \bullet, (\pi_2\ \bullet)[\pi_1\ \bullet / y])_{[\![\Sigma y:A'.B']\!]_\Gamma} \\ & = & (\pi_1\ \bullet, \pi_2\ \bullet)_{[\![\Sigma y:A'.B']\!]_\Gamma} \\ & =_\rho & \bullet \end{array}$$

La substitution est inutile puisque dans un contexte où $\bullet : \Sigma y : A.B$, $y \notin \pi_2\ \bullet$.
– Le cas des sous-ensembles est un peu différent. Si $T^\downarrow = \{\ x : T' \mid P\ \}$ alors $U^\downarrow = \{\ x : U' \mid P'\ \}$ avec $T' \equiv_{\beta\pi} U'$ et $P \equiv_{\beta\pi} P'$. La dérivation va avoir la forme suivante :

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash_? c' \equiv \bullet : T' \rhd U'}{\Gamma \vdash_? d = c'[\pi_1\ \bullet] : \{\ x : T' \mid P\ \} \rhd U'}}{\Gamma \vdash_? c = (d, ?_{[\![P']\!]_{\Gamma, x:U'}[d/x]})_{[\![\{x:U'|P'\}]\!]_\Gamma} : \{\ x : T' \mid P\ \} \rhd \{\ x : U' \mid P'\ \}}}{\Gamma \vdash_? c : T \rhd U}$$

On a donc $d = c'[\pi_1\ \bullet] \equiv \pi_1\ \bullet$.
On peut vérifier :

$$\begin{array}{rcl} c & \triangleq & (d, ?_{[\![P']\!]_{\Gamma, x:U'}[d/x]})_{[\![\{x:U'|P'\}]\!]_\Gamma} \\ & \equiv & ((\pi_1\ \bullet), ?_{[\![P']\!]_{\Gamma, x:U'}[\pi_1\ \bullet/x]})_{[\![\{x:U'|P'\}]\!]_\Gamma} \\ & =_\sigma & ((\pi_1\ \bullet), (\pi_2\ \bullet))_{[\![\{x:U'|P'\}]\!]_\Gamma} \\ & =_\rho & \bullet \end{array}$$

On fait ici un usage très libéral de l'indifférence aux preuves. En effet nous ne pouvons pas encore montrer que $\pi_2\ \bullet : [\![P']\!]_{\Gamma, x:U'}[\pi_1\ \bullet / x]$ puisqu'on sait seulement que dans le contexte où $\bullet : [\![\{\ x : T' \mid P\ \}]\!]_\Gamma, \pi_2\ \bullet : [\![P]\!]_{\Gamma, x:T'}[\pi_1\ \bullet / x]$. Montrer que les interprétations de P et P' sont équivalentes requiert la stabilité de la convertibilité par interprétation, ce que nous montrerons plus tard.

□

Lemme 3.2.5 (Coercion de sortes)**.** *Si* $\Gamma \vdash_? e : s \rhd T$ *ou* $\Gamma \vdash_? e : T \rhd s$ *alors* $T \equiv_{\beta\pi} s$ *et* $e = \bullet$.

Démonstration. Clairement on ne peut dériver $s \rhd T$ que par $\rhd$-Conv (éventuellement précédé de $\rhd$-$\downarrow$). En effet seule la règle $\rhd$-Proof pourrait s'appliquer, mais cela impliquerait que $T \equiv_{\beta\pi} \{\ x : U \mid P\ \}$ avec $s \rhd U$ et ainsi de suite. La seule possibilité est de dériver $s \equiv_{\beta\pi} T$ ou $s \equiv_{\beta\pi} U$, auquel cas U est une sorte ce qui contredit le fait que $\{\ x : U \mid P\ \} : s$ dans le cas précédent. On dérive donc $s \rhd T$ si et seulement si $s \equiv_{\beta\pi} T$. □

Lemme 3.2.6 (Affaiblissement et interprétation)**.** *Si* $\Gamma \vdash_\bullet t : T$ *alors pour tout* $\Delta \supseteq \Gamma$, $[\![t]\!]_\Gamma = [\![t]\!]_\Delta$.

Démonstration. Les seuls endroits où les environnement sont utilisés dans l'interprétation est lors des appels à la fonction de typage algorithmique, or on a bien la propriété que si $\Gamma \vdash_\bullet t : T$ alors $\Delta \vdash_\bullet t : T$ quand $\Gamma \subseteq \Delta$ par affaiblissement. Cette propriété est donc bien vérifiée. □

3.2.2 Substitution et transitivité

On peut déjà montrer que l'interprétation est stable par substitution pour les termes bien typés. Pour la coercion, on montre qu'elle est substitutive et que la coercion produite est équivalente à la coercion initiale dans laquelle on substitue.

Lemme 3.2.7 (Stabilité par substitution)**.** *Si* $\Gamma \vdash_\bullet u : U$*, alors*

$$\begin{array}{lcl} \vdash_\bullet \Gamma, x : U, \Delta & \Rightarrow & [\![\Gamma, \Delta[u/x]]\!] \equiv [\![\Gamma, x : U, \Delta]\!][[\![u]\!]_\Gamma/x] \\ \Gamma, x : U, \Delta \vdash_\bullet t : T & \Rightarrow & [\![t[u/x]]\!]_{\Gamma,\Delta[u/x]} \equiv [\![t]\!]_{\Gamma,x:U,\Delta}[[\![u]\!]_\Gamma/x] \\ \Gamma, x : U, \Delta \vdash_? c : T \rhd T' & \Rightarrow & \Gamma, \Delta[u/x] \vdash_? c' : T[u/x] \rhd T'[u/x] \wedge c' \equiv c[[\![u]\!]_\Gamma/x] \end{array}$$

Démonstration. Par induction mutuelle sur les dérivations de bonne formation, typage et coercion.

– Wf-Empty : Trivial.

– Wf-Var : Par induction sur la longueur de Δ.

- $\Delta = []$: Alors on a :

$$\frac{\Gamma \vdash_\bullet U : s}{\vdash_\bullet \Gamma, x : U} \; s \in \{\texttt{Set}, \texttt{Prop}, \texttt{Type}\} \wedge x \notin \Gamma$$

  Clairement $[\![\Gamma, \Delta[u/x]]\!] = [\![\Gamma]\!] = [\![\Gamma, x : U]\!][[\![u]\!]_\Gamma/x]$ puisque $x \notin \Gamma$.

- $\Delta = \Delta', y : A$: Alors on a :

$$\frac{\Gamma, x : U, \Delta' \vdash_\bullet A : s}{\vdash_\bullet \Gamma, x : U, \Delta', y : A}$$

  Par induction on a

$$[\![A[u/x]]\!]_{\Gamma,\Delta'[u/x]} \equiv [\![A]\!]_{\Gamma,x:U,\Delta'}[[\![u]\!]_\Gamma/x] \wedge [\![\Gamma, x : U, \Delta']\!][[\![u]\!]_\Gamma/x] \equiv [\![\Gamma, \Delta'[u/x]]\!]$$

  donc

$$\begin{array}{rcl} [\![\Gamma, \Delta[u/x]]\!] & = & [\![\Gamma, \Delta'[u/x], A[u/x]]\!] \\ & \triangleq & [\![\Gamma, \Delta'[u/x]]\!], [\![A[u/x]]\!]_{\Gamma,\Delta'[u/x]} \\ & = & [\![\Gamma, x : U, \Delta']\!][[\![u]\!]_\Gamma/x], y : [\![A]\!]_{\Gamma,x:U,\Delta'}[[\![u]\!]_\Gamma/x] \\ & = & [\![\Gamma, x : U, \Delta', y : A]\!][[\![u]\!]_\Gamma/x] \\ & = & [\![\Gamma, x : U, \Delta]\!][[\![u]\!]_\Gamma/x] \end{array}$$

– $\rhd$-$\downarrow$: On a :

$$\frac{\Gamma, x : U, \Delta \vdash_? c : T^\downarrow \rhd T'^\downarrow}{\Gamma, x : U, \Delta \vdash_? c : T \rhd T'}$$

Par induction on a $\Gamma, \Delta[u/x] \vdash_? c' : T^\downarrow[u/x] \rhd T'^\downarrow[u/x]$ avec $c' \equiv c[[\![u]\!]_\Gamma/x]$. Si $T[u/x]^\downarrow = T^\downarrow[u/x]$ et $T'[u/x]^\downarrow = T'^\downarrow[u/x]$ c'est direct par induction. Sinon, on a $T^\downarrow = x\ \overrightarrow{a}$ ou $T'^\downarrow = x\ \overrightarrow{a}$. Les deux cas sont similaires, on traite le cas ou $T^\downarrow = x\ \overrightarrow{a}$. Le jugement $x\ \overrightarrow{a} \rhd T'^\downarrow$ ne peut être dérivé que par $\rhd$-Proof ou $\rhd$-Conv.

- $\rhd$-Proof : On a :

$$\frac{\Gamma, x : U, \Delta \vdash_? d : x\ \overrightarrow{a} \rhd U' \qquad \Gamma, x : U, \Delta \vdash_\bullet \{\ y : U' \mid P\ \} : \texttt{Set}}{\Gamma, x : U, \Delta \vdash_? c : x\ \overrightarrow{a} \rhd T'^{\downarrow} = \{\ y : U' \mid P\ \}}$$

avec

$$c = (d, ?_{[\![P]\!]_{\Gamma,x:U,\Delta,y:U'}[d/y]})_{[\![\{x:U'|P\}]\!]_{\Gamma,x:U,\Delta}}$$

Par induction on a donc une dérivation de $\Gamma, \Delta[u/x] \vdash_? d' : u\ \overrightarrow{a[u/x]} \rhd U'[u/x]$ avec $d' \equiv d[[\![u]\!]_\Gamma/x]$. Par le lemme 3.2.3 on a aussi une dérivation de $\Gamma, \Delta[u/x] \vdash_? d'' : (u\ \overrightarrow{a[u/x]})^{\downarrow} \rhd (U'[u/x])^{\downarrow}$ avec $d'' \equiv d'[[\![u]\!]_\Gamma/x]$.
Par inversion $\Gamma, x : U, \Delta \vdash_\bullet U' : \texttt{Set}$ et $\Gamma, x : U, \Delta, y : U' \vdash_\bullet P : \texttt{Prop}$. On peut donc appliquer l'hypothèse d'induction pour obtenir : $[\![U'[u/x]]\!]_{\Gamma,\Delta[u/x]} \equiv [\![U']\!]_{\Gamma,x:U,\Delta}[[\![u]\!]_\Gamma/x]$ et $[\![P[u/x]]\!]_{\Gamma,\Delta[u/x],y:U'[u/x]} \equiv [\![P]\!]_{\Gamma,x:U,\Delta,y:U'}[[\![u]\!]_\Gamma/x]$. Par substitutivité on a aussi $\Gamma, \Delta[u/x] \vdash_\bullet \{\ y : U'[u/x] \mid P[u/x]\ \} : \texttt{Set}$. On peut donc dériver :

$$\frac{\dfrac{\dfrac{\Gamma, \Delta[u/x] \vdash_? d' : (u\ \overrightarrow{a[u/x]})^{\downarrow} \rhd (U'[u/x])^{\downarrow}}{\Gamma, \Delta[u/x] \vdash_? d' : (u\ \overrightarrow{a[u/x]})^{\downarrow} \rhd U'[u/x]} \qquad \Gamma, \Delta[u/x] \vdash_? c' : T[u/x]^{\downarrow} \rhd (T'[u/x])^{\downarrow} = \{\ y : U'[u/x] \mid P[u/x]\ \}}{\Gamma, \Delta[u/x] \vdash_? c' : T[u/x] \rhd T'[u/x]}$$

avec

$$c' = (d', ?_{[\![P[u/x]]\!]_{\Gamma,\Delta[u/x],y:U'[u/x]}[d'/y]})_{[\![\{y:U'[u/x]|P[u/x]\}]\!]_{\Gamma,\Delta[u/x]}}$$

On a aussi :

$$\begin{aligned}
[\![P[u/x]]\!]_{\Gamma,\Delta[u/x],y:U'[u/x]}[d'/y] &\equiv [\![P]\!]_{\Gamma,x:U,\Delta,y:U'}[[\![u]\!]_\Gamma/x][d'/y] \\
&\equiv [\![P]\!]_{\Gamma,x:U,\Delta,y:U'}[[\![u]\!]_\Gamma/x][d[[\![u]\!]_\Gamma/x]/y] \\
&= [\![P]\!]_{\Gamma,x:U,\Delta,y:U'}[d/y][[\![u]\!]_\Gamma/x]
\end{aligned}$$

Soit

$$A \triangleq [\![P[u/x]]\!]_{\Gamma,\Delta[u/x],y:U'[u/x]}[d'/y] \wedge B \triangleq [\![P]\!]_{\Gamma,x:U,\Delta,y:U'}[d/y]$$

on a donc :

$$\begin{aligned}
c' &\triangleq (d', ?_A)_{[\![\{y:U'[u/x]|P[u/x]\}]\!]_{\Gamma,\Delta[u/x]}} \\
&\equiv (d[[\![u]\!]_\Gamma/x], ?_A)_{[\![\{y:U'[u/x]|P[u/x]\}]\!]_{\Gamma,\Delta[u/x]}} \\
&\equiv (d[[\![u]\!]_\Gamma/x], (?_B)[[\![u]\!]_\Gamma/x])_{[\![\{y:U'|P\}]\!]_{\Gamma,x:U,\Delta}[[\![u]\!]_\Gamma/x]} \\
&= ((d, ?_B)_{[\![\{y:U'|P\}]\!]_{\Gamma,x:U,\Delta}})[[\![u]\!]_\Gamma/x] \\
&= c[[\![u]\!]_\Gamma/x]
\end{aligned}$$

- $\rhd$-Conv : Alors on a $T^{\downarrow} = x\ \overrightarrow{a}$ et $T'^{\downarrow} = x\ \overrightarrow{b}$ où $\overrightarrow{a} \equiv \overrightarrow{b}$. Par substitutivité de l'équivalence, $T^{\downarrow}[u/x] \equiv T'^{\downarrow}[u/x]$, donc par le lemme 3.2.4, on a $\Gamma, \Delta[u/x] \vdash_? c' : u\ \overrightarrow{a[u/x]} \rhd u\ \overrightarrow{b[u/x]}$ est dérivable et $c' \equiv \bullet = \bullet[[\![u]\!]_\Gamma/x]$.

Dans le cas où $T'^{\downarrow} = x\ \overrightarrow{a}$ et $T^{\downarrow} \neq x\ \overrightarrow{a}$, le jugement $\Gamma, x : U, \Delta \vdash_? c : T^{\downarrow} \rhd x\ \overrightarrow{a}$ ne peut être dérivé que par $\rhd$-Subset ou $\rhd$-Conv.

- $\rhd$-Subset : On a :

$$\frac{\Gamma, x : U, \Delta \vdash_? c' : U' \rhd x\ \overrightarrow{a}}{\Gamma, x : U, \Delta \vdash_? c = c'[\pi_1\ \bullet] : T^{\downarrow} = \{\ y : U' \mid P\ \} \rhd x\ \overrightarrow{a}}$$

Par induction, on a une dérivation de $\Gamma, \Delta[u/x] \vdash_? c'' : U'[u/x] \rhd u\ \overrightarrow{a[u/x]}$ avec $c'' \equiv c'[[\![u]\!]_\Gamma/x]$. On peut donc dériver :

$$\dfrac{\dfrac{\Gamma, \Delta[u/x] \vdash_? c'' : U'[u/x] \rhd u\ \overrightarrow{a[u/x]}}{\Gamma, \Delta[u/x] \vdash_? c''[\pi_1\ \bullet] : T[u/x]^\downarrow = \{\ y : U'[u/x] \mid P[u/x]\ \} \rhd T'[u/x]^\downarrow}}{\Gamma, \Delta[u/x] \vdash_? c''[\pi_1\ \bullet] : T[u/x] \rhd T'[u/x]}$$

Clairement, $c''[\pi_1\ \bullet] \equiv c'[[\![u]\!]_\Gamma/x][\pi_1\ \bullet] = c'[\pi_1\ \bullet][[\![u]\!]_\Gamma/x] = c[[\![u]\!]_\Gamma/x]$.

– $\rhd$-Conv : On a $T = T^\downarrow$, $T' = T'^\downarrow$, $T \equiv_{\beta\pi} T'$ et $c \equiv \bullet$. Par substitutivité de la $\beta\rho$-équivalence, on a aussi $T[u/x] \equiv_{\beta\pi} T'[u/x]$. Par le lemme 3.2.4, on sait qu'il existe une coercion c' telle que le jugement $\Gamma, \Delta[u/x] \vdash_? c' : T[u/x] \rhd T'[u/x]$ est dérivable et $c' \equiv \bullet$. On a bien $c' \equiv c[[\![u]\!]_\Gamma/x]$.

– $\rhd$-Prod : On a :

$$\dfrac{\Gamma, x : U, \Delta \vdash_? c_1 : A' \rhd A \qquad \Gamma, x : U, \Delta, y : A' \vdash_? c_2 : B \rhd B'}{\Gamma, x : U, \Delta \vdash_? \lambda y : [\![A']\!]_{\Gamma, x:U, \Delta}.\ c_2[\bullet\ c_1[y]] : \Pi y : A.B \rhd \Pi y : A'.B'}$$

Par induction et application de $\rhd$-Prod :

$$\dfrac{\begin{array}{c}\Gamma, \Delta[u/x] \vdash_? c_1' : A'[u/x] \rhd A[u/x] \\ \Gamma, \Delta[u/x], y : A'[u/x] \vdash_? c_2' : B[u/x] \rhd B'[u/x]\end{array}}{\Gamma, \Delta[u/x] \vdash_? c' : \Pi y : A[u/x].B[u/x] \rhd \Pi y : A'[u/x].B'[u/x]}$$

où

$$c' = \lambda y : [\![A'[u/x]]\!]_{\Gamma, \Delta[u/x]}.\ c_2'[\bullet\ c_1'[y]] \quad c_1' \equiv c_1[[\![u]\!]_\Gamma/x] \quad c_2' \equiv c_2[[\![u]\!]_\Gamma/x]$$

On peut supposer que $\Gamma, x : U, \Delta \vdash_\bullet A', A : s_1$ puisqu'on a une dérivation de $\Gamma, x : U, \Delta \vdash_? \_ : \Pi y : A'.B' \rhd \Pi y : A.Bs$ (3.1.1). On applique l'hypothèse d'induction pour obtenir $[\![A'[u/x]]\!]_{\Gamma, \Delta[u/x]} \equiv [\![A']\!]_{\Gamma, x:U, \Delta}[[\![u]\!]_\Gamma/x]$. On a donc bien $c' \equiv c[[\![u]\!]_\Gamma/x]$.

– $\rhd$-Sum, $\rhd$-Proof, $\rhd$-Subset : Idem, direct par induction.

– PropSet : Trivial.

– Var : On a :

$$\dfrac{\vdash_\bullet \Gamma, x : U, \Delta \qquad y : T \in (\Gamma, x : U, \Delta)}{\Gamma, x : U, \Delta \vdash_\bullet y : T}$$

- Si $x \neq y$, alors par définition de l'interprétation, on montre :

$$[\![y[u/x]]\!]_{\Gamma, \Delta[u/x]} = y = [\![y]\!]_{\Gamma, x:U, \Delta}[[\![u]\!]_\Gamma/x]$$

- Sinon, $t[u/x] = u$ et $[\![u]\!]_{\Gamma, \Delta[u/x]} = [\![x]\!]_{\Gamma, x:U, \Delta}[[\![u]\!]_\Gamma/x] = [\![u]\!]_{\Gamma, x:U, \Delta}$. On utilise ici le fait que $[\![t]\!]_\Gamma = [\![t]\!]_\Delta$ si $\Gamma \vdash_\bullet t : T$ pour tout Δ incluant Γ (lemme 3.2.6).

– App :

$$\dfrac{\begin{array}{c}\Gamma, x : U, \Delta \vdash_\bullet f : F \\ \mu_\bullet(F) = \Pi y : A.B : s \qquad \Gamma, x : U, \Delta \vdash_\bullet e : E \qquad \Gamma, x : U, \Delta \vdash_\bullet E \rhd A : s'\end{array}}{\Gamma, x : U, \Delta \vdash_\bullet (f\ e) : B[e/y]}$$

Par induction :

$$[\![f[u/x]]\!]_{\Gamma, \Delta[u/x]} \equiv [\![f]\!]_{\Gamma, x:U, \Delta}[[\![u]\!]_\Gamma/x]$$
$$[\![e[u/x]]\!]_{\Gamma, \Delta[u/x]} \equiv [\![e]\!]_{\Gamma, x:U, \Delta}[[\![u]\!]_\Gamma/x]$$

Par définition de la substitution et de l'interprétation,

$$\begin{array}{rcl}
[\![(f\ e)]\!]_{\Gamma,x:U,\Delta} & = & (\pi_F\ [\![f]\!]_{\Gamma,x:U,\Delta})\ (c_e\ [\![e]\!]_{\Gamma,x:U,\Delta}) \\
\text{où} & & \\
\pi_F & = & \mathbf{coerce}_{\Gamma,x:U,\Delta}\ F\ (\Pi y : A.B) \\
c_e & = & \mathbf{coerce}_{\Gamma,x:U,\Delta}\ E\ A.
\end{array}$$

On a la substitutivité du typage (lemme 2.3.23), le jugement substitué est :

$$\frac{\begin{array}{cc}\Gamma,\Delta[u/x] \vdash_\bullet f[u/x] : F[u/x] & \Gamma,\Delta[u/x] \vdash_\bullet e[u/x] : E[u/x] \\ \mu_\bullet(F[u/x]) = \Pi y : A[u/x].B[u/x] : s & \Gamma,\Delta[u/x] \vdash_\bullet E[u/x] \rhd A[u/x] : s'\end{array}}{\Gamma,\Delta[u/x] \vdash_\bullet (f\ e)[u/x] : B'[e[u/x]/y]}$$

Soit $e' = e[u/x]$ et $f' = f[u/x]$, on a donc d'autre part :

$$\begin{array}{rcl}
[\![(f\ e)[u/x]]\!]_{\Gamma,\Delta[u/x]} & = & [\![f'\ e']\!]_{\Gamma,\Delta[u/x]} \\
& = & \pi_{F[u/x]}[[\![f']\!]_{\Gamma,\Delta[u/x]}]\ c_{e'}[[\![e']\!]_{\Gamma,\Delta[u/x]}] \\
\text{où} & & \\
\pi_{F[u/x]} & = & \mathbf{coerce}_{\Gamma,\Delta[u/x]}\ F[u/x]\ (\Pi y : A[u/x].B[u/x]) \\
c_{e'} & = & \mathbf{coerce}_{\Gamma,\Delta[u/x]}\ E[u/x]\ A[u/x]
\end{array}$$

Par induction, il existe des coercions d, e telles que : $\Gamma,\Delta[u/x] \vdash_?\ d : F[u/x] \rhd (\Pi y : A.B)[u/x]$ et $\Gamma,\Delta[u/x] \vdash_?\ e : E[u/x] \rhd A[u/x]$ avec $d \equiv \pi_F[[\![u]\!]_\Gamma/x]$ et $e \equiv c_e[[\![u]\!]_\Gamma/x]$. Par unicité des coercions on en déduit que $d = \pi_{F[u/x]}$ et $e = c_{e'}$.
On peut vérifier :

$$\begin{array}{rl}
& [\![f\ e]\!]_{\Gamma,x:U,\Delta}[[\![u]\!]_\Gamma/x] \\
& \quad \textbf{\{ Définition de l'interprétation \}} \\
\triangleq & (\pi_F[[\![f]\!]_{\Gamma,x:U,\Delta}])\ (c_e[[\![e]\!]_{\Gamma,x:U,\Delta}])[[\![u]\!]_\Gamma/x] \\
& \quad \textbf{\{ Définition de la substitution \}} \\
= & (\pi_F[[\![u]\!]_\Gamma/x][[\![f]\!]_{\Gamma,x:U,\Delta}[[\![u]\!]_\Gamma/x]])\ (c_e[[\![u]\!]_\Gamma/x][[\![e]\!]_{\Gamma,x:U,\Delta}[[\![u]\!]_\Gamma/x]]) \\
& \quad \textbf{\{ Application de l'hypothèse d'induction pour les termes \}} \\
\equiv & (\pi_F[[\![u]\!]_\Gamma/x][[\![f']\!]_{\Gamma,\Delta[u/x]}])\ (c_e[[\![u]\!]_\Gamma/x][[\![e']\!]_{\Gamma,\Delta[u/x]}]) \\
& \quad \textbf{\{ Application de l'hypothèse d'induction pour les coercions \}} \\
\equiv & (d[[\![f']\!]_{\Gamma,\Delta[u/x]}])\ e[[\![e']\!]_{\Gamma,\Delta[u/x]}] \\
& \quad \textbf{\{ Unicité des coercions \}} \\
= & \pi_{F[u/x]}[[\![f']\!]_{\Gamma,\Delta[u/x]}]\ c_{e'}[[\![e']\!]_{\Gamma,\Delta[u/x]}] \\
& \quad \textbf{\{ Définition de l'interprétation \}} \\
\triangleq & [\![(f\ e)[u/x]]\!]_{\Gamma,\Delta[u/x]}
\end{array}$$

– Prod, Sum, Subset : Par induction.

– Abs : On a :

$$\frac{\Gamma, x : U, \Delta \vdash_\bullet \Pi y : T.U : s \qquad \Gamma, x : U, \Delta, y : T \vdash_\bullet M : U}{\Gamma, x : U, \Delta \vdash_\bullet \lambda y : T.M : \Pi y : T.U}$$

On a bien :

$$\begin{array}{rl} & [\![\lambda y : T.M]\!]_{\Gamma,x:U,\Delta}[[\![u]\!]_\Gamma/x] \\ & \quad \textbf{\{ Définition de l'interprétation \}} \\ \triangleq & \lambda y : [\![T]\!]_{\Gamma,x:U,\Delta}[[\![u]\!]_\Gamma/x].[\![M]\!]_{\Gamma,x:U,\Delta,y:T}[[\![u]\!]_\Gamma/x] \\ & \quad \textbf{\{ Application de l'hypothèse de récurrence \}} \\ \equiv & \lambda y : [\![T[u/x]]\!]_{\Gamma,\Delta[u/x]}.[\![M[u/x]]\!]_{\Gamma,\Delta[u/x],y:T[u/x]} \\ & \quad \textbf{\{ Définition de l'interprétation \}} \\ \triangleq & [\![\lambda y : T[u/x].M[u/x]]\!]_{\Gamma,\Delta[u/x]} \end{array}$$

– Pair : On a

$$\frac{\Gamma, x : U, \Delta \vdash_\bullet \Sigma y : T.V : s \qquad \Gamma, x : U, \Delta \vdash_\bullet t : T' \rhd T : s \qquad \Gamma, x : U, \Delta \vdash_\bullet v : V' \rhd V[t/y] : s}{\Gamma, x : U, \Delta \vdash_\bullet (t, v)_{\Sigma y:T.V} : \Sigma y : T.V}$$

et le jugement substitué :

$$\frac{\Gamma, \Delta[u/x] \vdash_\bullet \Sigma y : T[u/x].V[u/x] : s \qquad \Gamma, \Delta[u/x] \vdash_\bullet t[u/x] : T'[u/x] \rhd T[u/x] : s \qquad \Gamma, \Delta[u/x] \vdash_\bullet v[u/x] : V'[u/x] \rhd V[u/x][t[u/x]/y] : s}{\Gamma, \Delta[u/x] \vdash_\bullet (t[u/x], v[u/x])_{\Sigma y:T[u/x].V[u/x]} : \Sigma y : T[u/x].V[u/x]}$$

On a $V[u/x][t[u/x]/y] = V[t/y][u/x]$ puisque $y \notin \mathcal{FV}(u)$. Ici on a les coercions $\Gamma, x : U, \Delta \vdash_? c : T' \rhd T$ et $\Gamma, x : U, \Delta \vdash_? d : V' \rhd V[t/y]$. Par induction on obtient $\Gamma, \Delta[u/x] \vdash_? c' : T'[u/x] \rhd T[u/x]$ et $\Gamma, \Delta[u/x] \vdash_? c' : V'[u/x] \rhd V[t/y][u/x]$ avec $c' \equiv c[[\![u]\!]_\Gamma/x]$ et $d' \equiv d[[\![u]\!]_\Gamma/x]$.

$$\begin{array}{rl} & [\![(t, v)_{\Sigma x:T.V}]\!]_{\Gamma,x:U,\Delta}[[\![u]\!]_\Gamma/x] \\ & \quad \textbf{\{ Définition de l'interprétation \}} \\ \triangleq & (c[[\![t]\!]_{\Gamma,x:U,\Delta}], d[[\![v]\!]_{\Gamma,x:U,\Delta}])_{[\![\Sigma x:T.V]\!]_{\Gamma,x:U,\Delta}}[[\![u]\!]_\Gamma/x] \\ & \quad \textbf{\{ Application de l'hypothèse de récurrence \}} \\ \equiv & (c'[[\![t[u/x]]\!]_{\Gamma,\Delta[u/x]}], d'[[\![v[u/x]]\!]_{\Gamma,\Delta[u/x]}])_{[\![(\Sigma x:T.V)[u/x]]\!]_{\Gamma,\Delta[u/x]}} \\ & \quad \textbf{\{ Définition de l'interprétation \}} \\ \triangleq & [\![((t, u)_{\Sigma x:T.V})[u/x]]\!]_{\Gamma,\Delta[u/x]} \end{array}$$

– Pi-1, Pi-2 : On a

$$\frac{\Gamma, x : U, \Delta \vdash_\bullet t : S \qquad \mu_\bullet(S) = \Sigma y : T.U}{\Gamma, x : U, \Delta \vdash_\bullet \pi_i\ t : \_}$$

$$\frac{\Gamma, \Delta[u/x] \vdash_\bullet t[u/x] : S[u/x] \qquad \mu_\bullet(S[u/x]) = (\Sigma y : T.U)[u/x]}{\Gamma, \Delta[u/x] \vdash_\bullet \pi_i\ t[u/x] : \_}$$

Encore une fois on obtient la coercion $\Gamma, \Delta[u/x] \vdash_? c' : S[u/x] \rhd (\Sigma y : T.U)[u/x]$ par induction sur la dérivation de $\Gamma, x : U, \Delta \vdash_? c : S \rhd \Sigma y : T.U$. On a $c' \equiv c[[\![u]\!]_\Gamma/x]$.

$$\begin{array}{rl} & [\![\pi_i\ t]\!]_{\Gamma,x:U,\Delta}[[\![u]\!]_\Gamma/x] \\ & \quad \textbf{\{ Définition de l'interprétation \}} \\ \triangleq & \pi_i\ c[[\![t]\!]_{\Gamma,x:U,\Delta}][[\![u]\!]_\Gamma/x] \\ & \quad \textbf{\{ Application de l'hypothèse de récurrence \}} \\ \equiv & \pi_i\ c'[[\![t[u/x]]\!]_{\Gamma,\Delta[u/x]}] \\ & \quad \textbf{\{ Définition de l'interprétation \}} \\ \triangleq & [\![\pi_i\ t[u/x]]\!]_{\Gamma,\Delta[u/x]} \end{array}$$

□

On va maintenant étendre la relation de coercion aux contextes de manière canonique.

Définition 3.2.8 (Coercion de contextes)**.** *On définit inductivement la coercion de deux contextes de coercions algorithmiques par les règles suivantes :*
- $[] \rhd []$
- $(\Gamma, x : T) \rhd (\Gamma', x : T')$ *si* $\Gamma \rhd \Gamma'$ *et* $T \rhd T'$.

De même pour les coercions explicites dérivées par le jugement $\Gamma \vdash_? c : T \rhd S$.

Définition 3.2.9 (Coercion explicites de contextes)**.** *On définit inductivement la coercion de deux contextes de coercions explicites par les règles suivantes :*
- $[] \rhd []$
- $(\rho, c) : (\Gamma, x : T) \rhd (\Gamma', x : T')$ *si* $\rho : \Gamma \rhd \Gamma'$ *et* $\Gamma \vdash_? c : T \rhd T'$.

Clairement toute coercion de contexte algorithmique correspond à une coercion de contexte explicite et vice-versa.

Définition 3.2.10 (Extension de la substitution aux coercions de contextes)**.** *On définit la substitution d'une coercion de contexte inductivement :*
- $t[[]] = t$
- $t[(\rho, c) : (\Gamma, x : T) \rhd (\Gamma', x : T')] = t[\rho : \Gamma \rhd \Gamma'][c[x]/x]$

Lorsqu'on fait des preuves sur la coercion, on doit en général passer "sous" les types produits et sommes dépendantes. Cela requiert de pouvoir changer les contextes de typage de certaines dérivations par des contextes coercibles sans changer leur taille.

Lemme 3.2.11 (Stabilité par affaiblissement des coercions)**.** *Si* $\Gamma \vdash_? c : T \rhd T'$*, alors pour tout* Δ*,* $\rho : \Delta \rhd \Gamma$ *tels que* $[\![T']\!]_\Delta \equiv [\![T']\!]_\Gamma[\rho]$*, on a* $\Delta \vdash_? c' : T \rhd T'$ *et* $c' \equiv c[\rho]$ *avec une dérivation de même taille.*

Démonstration. Par induction sur la dérivation de coercion $\Gamma \vdash_? c : T \rhd T$.

- $\rhd$-$\downarrow$: On a $\Gamma \vdash_? c : T^\downarrow \rhd T'^\downarrow$. Par induction, $\Delta \vdash_? c[\rho] : T^\downarrow \rhd T'^\downarrow$. On peut donc dériver $\Delta \vdash_? c[\rho] : T \rhd T'$ par $\rhd$-$\downarrow$.
- $\rhd$-Conv : On a $T \equiv_{\beta\pi} T'$ et $c = \bullet$. On peut donc dériver $\Delta \vdash_? c' = \bullet : T \rhd T'$, on a bien $c' \equiv c[\rho]$.
- $\rhd$-Prod : On a :

$$\frac{\Gamma \vdash_? c_1 : C \rhd A \qquad \Gamma, x : C \vdash_? c_2 : B \rhd D}{\Gamma \vdash_? \lambda x : [\![C]\!]_\Gamma.\ c_2[\bullet\ c_1[x]] : \Pi x : A.B \rhd \Pi x : C.D}$$

Par induction on a $\Delta \vdash_? c_1[\rho] : C \rhd A$. On peut définir la coercion $\sigma = (\rho, \bullet) : (\Delta, x : C) \rhd (\Gamma, x : C)$ et obtenir par induction : $\Delta, x : C \vdash_? c_2' : B \rhd D$ avec $c_2' \equiv c_2[\sigma]$.

On peut alors appliquer $\rhd$-Prod pour obtenir :

$$\Delta \vdash_? c' = \lambda x : [\![C]\!]_\Delta.\ c_2[\sigma][\bullet\ c_1[\rho][x]] : \Pi x : A.B \rhd \Pi x : C.D$$

On a :

$$\begin{array}{rl} & (\lambda x : [\![C]\!]_\Gamma.\ c_2[\bullet\ c_1[x]])[\rho] \\ & \quad \textbf{\{ Définition de la substitution \}} \\ = & \lambda x : [\![C]\!]_\Gamma[\rho].(c_2[\rho])[\bullet\ (c_1[\rho])[x]] \\ & \quad \textbf{\{ Condition } [\![T']\!]_\Delta \equiv [\![T']\!]_\Gamma[\rho] \textbf{ \}} \\ \equiv & \lambda x : [\![C]\!]_\Delta.(c_2[\rho])[\bullet\ (c_1[\rho])[x]] \\ & \quad \textbf{\{ Coercion identité dans } \sigma \textbf{ \}} \\ = & \lambda x : [\![C]\!]_\Delta.(c_2[\sigma])[\bullet\ (c_1[\rho])[x]] \end{array}$$

– $\rhd$-Sum :

$$\frac{\Gamma \vdash_? c_1 : A \rhd C \qquad \Gamma, x : A \vdash_? c_2 : B \rhd D}{\Gamma \vdash_? (c_1[\pi_1\ \bullet], c_2[\pi_1\ \bullet/x][\pi_2\ \bullet])_{[\![\Sigma x:C.D]\!]_\Gamma} : \Sigma x : A.B \rhd \Sigma x : C.D}$$

Par induction on a $\Delta \vdash_? c_1[\rho] : A \rhd C$. On peut définir la coercion $\sigma = \rho, \bullet : (\Delta, x : A) \rhd (\Gamma, x : A)$ et obtenir par induction : $\Delta, x : A \vdash_? c_2[\sigma] : B \rhd D$

On peut alors appliquer $\rhd$-Sum pour obtenir :

$$\Delta \vdash_? c' = ((c_1[\rho])[\pi_1\ \bullet], (c_2[\sigma])[\pi_1\ \bullet/x][\pi_2\ \bullet])_{[\![\Sigma x:C.D]\!]_\Delta} : \Sigma x : A.B \rhd \Sigma x : C.D$$

On a :

$$\begin{array}{rl} & c[\rho] \\ & \quad \textbf{\{ Définition \}} \\ \triangleq & ((c_1[\pi_1\ \bullet], c_2[\pi_2\ \bullet][\pi_1\ \bullet/x])_{[\![\Sigma x:C.D]\!]_\Gamma})[\rho] \\ & \quad \textbf{\{ Définition de la substitution, } x \notin \rho \textbf{ \}} \\ = & (c_1[\rho][\pi_1\ \bullet], c_2[\rho][\pi_2\ \bullet][\pi_1\ \bullet/x])_{[\![\Sigma x:C.D]\!]_\Gamma[\rho]} \\ & \quad \textbf{\{ Condition \}} \\ = & (c_1[\rho][\pi_1\ \bullet], c_2[\rho][\pi_2\ \bullet][\pi_1\ \bullet/x])_{[\![\Sigma x:C.D]\!]_\Delta} \\ & \quad \textbf{\{ Coercion identité dans } \sigma \textbf{ \}} \\ = & (c_1[\rho][\pi_1\ \bullet], c_2[\sigma][\pi_2\ \bullet][\pi_1\ \bullet/x])_{[\![\Sigma x:C.D]\!]_\Delta} \\ & \quad \textbf{\{ Définition \}} \\ \triangleq & c' \end{array}$$

– $\rhd$-Proof :

$$\frac{\Gamma \vdash_? d : T \rhd U}{\Gamma \vdash_? c = (d, ?_{[\![P]\!]_{\Gamma, x:U}[d/x]})_{[\![\{x:U|P\}]\!]_\Gamma} : T \rhd \{\ x : U \mid P\ \}}$$

Par induction, on a $\Delta \vdash_? d[\rho] : T \rhd U$. On peut donc dériver :

$$\Delta \vdash_? c' = (d[\rho], ?_{[\![P]\!]_{\Delta, x:U}[d[\rho]/x]})_{[\![\{x:U|P\}]\!]_\Delta} : T \rhd \{\ x : U \mid P\ \}$$

On a bien $c[\rho] \equiv c'$, car :

$$
\begin{aligned}
& c[\rho] \\
& \quad \textbf{\{ Définition \}} \\
\triangleq\ & ((d, ?_{[\![P]\!]_{\Gamma,x:U}[d/x]})_{[\![\{x:U|P\}]\!]_\Gamma})[\rho] \\
& \quad \textbf{\{ Condition } [\![\{\ x : U \mid P\ \}]\!]_\Delta \equiv [\![\{\ x : U \mid P\ \}]\!]_\Gamma[\rho] \textbf{ \}} \\
\equiv\ & (d[\rho], ?_{[\![P]\!]_{\Gamma,x:U}[d/x][\rho]})_{[\![\{x:U|P\}]\!]_\Delta} \\
& \quad \textbf{\{ Définition de la substitution \}} \\
=\ & (d[\rho], ?_{[\![P]\!]_{\Gamma,x:U}[\rho][d[\rho]/x]})_{[\![\{x:U|P\}]\!]_\Delta} \\
& \quad \textbf{\{ Condition } [\![P]\!]_{\Delta,x:U} \equiv [\![P]\!]_{\Gamma,x:U}[\rho] \textbf{ \}} \\
\equiv\ & (d[\rho], ?_{[\![P]\!]_{\Delta,x:U}[d[\rho]/x]})_{[\![\{x:U|P\}]\!]_\Delta} \\
& \quad \textbf{\{ Définition \}} \\
\triangleq\ & c'
\end{aligned}
$$

– $\rhd$-Subset : Direct par induction.

□

Pour montrer la stabilité par affaiblissement pour le typage, on a besoin d'une propriété essentielle de la coercion, la transitivité.

3.2.3 Transitivité de la coercion

On a déjà montré l'élimination de la transitivité pour les dérivations algorithmiques précédemment, mais lorsqu'on étudie la dérivation de coercions on a besoin d'un résultat reliant les coercions initiales et la coercion composée. Notons qu'on doit aussi permettre un affaiblissement pour passer sous les produits.

Lemme 3.2.12 (Transitivité de la coercion avec affaiblissement)**.** *S'il existe c_1, c_2 tels que $\Gamma \vdash_? c_1 : S \rhd T$ et $\Delta \vdash_? c_2 : T \rhd U$ avec $\rho : \Delta \rhd \Gamma$ et $[\![T]\!]_\Gamma[\rho] \equiv [\![T]\!]_\Delta$, alors $\exists! c, \Delta \vdash_? c : S \rhd U$ et $c \equiv c_2 \circ c_1[\rho]$.*

Démonstration. Par induction lexicographique sur la paire de dérivations de c_1 et c_2.

– $\rhd$-Conv : On va traiter les cas où cette règle est utilisée en racine d'une des deux dérivations, d'abord à gauche puis à droite.

$$\frac{S \equiv_{\beta\pi} T}{\Gamma \vdash_? c_1 = \bullet : S \rhd T} \qquad \Delta \vdash_? c_2 : T \rhd U$$

Les conditions de bord de $\rhd$-Conv nous donnent comme hypothèses que S et T sont en forme normale de tête et que $S \neq \Pi, \Sigma, \{|\}$. Par inversion de la $\beta\rho$-équivalence $S \equiv_{\beta\pi} T$, on a aussi $T \neq \Pi, \Sigma, \{|\}$. Les seules règles pouvant s'appliquer a la fin de la dérivation de c_2 sont donc $\rhd$-Conv, $\rhd$-$\downarrow$ et $\rhd$-Proof.

- $\rhd$-Conv : Alors on a $U = U^\downarrow \neq \Pi, \Sigma, \{|\}$. La coercion composée est alors $\bullet \circ \bullet$. C'est bien la coercion dérivée pour le jugement $\Delta \vdash_? \bullet : S \rhd U$ par $\rhd$-Conv.
- $\rhd$-$\downarrow$: On a alors $\Delta \vdash_? c_2 : T^\downarrow \rhd U^\downarrow$. Comme $T = T^\downarrow$, on peut appliquer l'hypothèse d'induction pour obtenir une coercion c telle que $\Delta \vdash_? c : S \rhd U^\downarrow$ et $c \equiv c_2 \circ c_1[\rho]$. Une application de $\rhd$-$\downarrow$ suffit pour obtenir une dérivation de $\Delta \vdash_? c : S \rhd U$ avec $c \equiv c_2 \circ c_1[\rho]$.

- $\rhd$-Proof : Ici on a :

$$\frac{\Delta \vdash_? d : T \rhd U'}{\Delta \vdash_? c_2 = (d, ?_{[\![P]\!]_\Delta[d/x]})_{[\![\{x:U'|P\}]\!]_\Delta} : T \rhd U = \{\ x : U' \mid P\ \}}$$

  Par induction, il existe une coercion $d' \equiv d \circ c_1[\rho] = d$ telle que $\Delta \vdash_? d' : S \rhd U'$. On applique $\rhd$-Proof pour obtenir la coercion c de S à U. Clairement $c \equiv c_2 \circ c_1[\rho] = c_2 \circ \bullet = c_2$.

Supposons maintenant que la dérivation de c_2 termine par une application de $\rhd$-Conv. Alors T et U sont en forme normale de tête et $T, U \neq \Pi, \Sigma, \{|\}$. Les seules règles pouvant apparaître en racine de la dérivation de c_1 sont donc $\rhd$-Conv, $\rhd$-$\downarrow$ et $\rhd$-Subset.

- $\rhd$-Conv : Alors on a $S = S^\downarrow \neq \Pi, \Sigma, \{|\}$. La coercion composée est alors $\bullet \circ \bullet$. C'est bien la coercion dérivée pour le jugement $\Delta \vdash_? \bullet : S \rhd U$ par $\rhd$-Conv.
- $\rhd$-$\downarrow$: On a alors $\Gamma \vdash_? c_1 : S^\downarrow \rhd T^\downarrow$. Comme $T = T^\downarrow$, on peut appliquer l'hypothèse d'induction pour obtenir une coercion c telle que $\Delta \vdash_? c : S^\downarrow \rhd U$ et $c \equiv c_2 \circ c_1[\rho]$. Une application de $\rhd$-$\downarrow$ suffit pour obtenir une dérivation de $\Delta \vdash_? c : S \rhd U$ avec $c \equiv c_2 \circ c_1[\rho]$.
- $\rhd$-Subset : Ici on a :

$$\frac{\Gamma \vdash_? d : S' \rhd T}{\Gamma \vdash_? c_1 = d[\pi_1\ \bullet] : S = \{\ x : S' \mid P\ \} \rhd T}$$

  Par induction, il existe une coercion $d' \equiv c_2 \circ d[\rho] = d[\rho]$ telle que $\Delta \vdash_? d' : S' \rhd U$. On applique $\rhd$-Subset pour obtenir la coercion $c = d[\rho][\pi_1\ \bullet]$ de S à U. On a

$$c = d[\rho][\pi_1\ \bullet] = d[\pi_1\ \bullet][\rho] \equiv c_2 \circ c_1[\rho]$$

– $\rhd$-$\downarrow$:

$$\frac{\Gamma \vdash_? c : S^\downarrow \rhd T^\downarrow}{\Gamma \vdash_? c : S \rhd T} \qquad \Delta \vdash_? d : T \rhd U$$

Si $T = T^\downarrow$ alors c'est trivial par induction et application de $\rhd$-$\downarrow$. Sinon, la seule règle permettant de dériver $\Delta \vdash_? d : T \rhd U$ est $\rhd$-$\downarrow$. Il suffit alors d'appliquer l'hypothèse d'induction pour obtenir une dérivation de $\Delta \vdash_? c : S^\downarrow \rhd U^\downarrow$ avec $c \equiv c_2 \circ c_1[\rho]$ puis $\rhd$-$\downarrow$ nous permet de construire le jugement $\Gamma \vdash_? c : S \rhd U$.

De même si l'on a une application de $\rhd$-$\downarrow$ à la racine de la dérivation de droite.

On peut donc se ramener au cas où $\rhd$-Conv et $\rhd$-$\downarrow$ ne sont appliquées à la racine d'aucune des deux dérivations.

– $\rhd$-Prod :

$$\frac{\Gamma \vdash_? c_1 : X' \rhd X \qquad \Gamma, x : X' \vdash_? c_2 : Y \rhd Y'}{\Gamma \vdash_? c = \lambda x : [\![X']\!]_\Gamma . c_2[\bullet\ c_1[x]] : \Pi x : X . Y \rhd \Pi x : X' . Y'} \qquad \Delta \vdash_? d : \Pi x : X' . Y' \rhd U$$

Par induction sur la dérivation $\Delta \vdash_? d : \Pi x : X' . Y' \rhd U$. Seules deux règles peuvent s'appliquer à la racine :

- $\rhd$-Prod : On a $U = \Pi x : S . T$ et la dérivation a la forme :

$$\frac{\Delta \vdash_? d_1 : S \rhd X' \qquad \Delta, x : S \vdash_? d_2 : Y' \rhd T}{\Delta \vdash_? d = (\lambda x : [\![S]\!]_\Delta . d_2[\bullet\ d_1[x]]) : \Pi x : X' . Y' \rhd \Pi x : S . T}$$

On peut construire une coercion de contextes de $\theta = \rho, d_1 : (\Delta, x : S) \rhd (\Gamma, x : X')$. Par le lemme 3.2.11 on obtient la dérivation $\Delta, x : S \vdash_? c_2' : Y \rhd Y'$ de même taille que la dérivation de c_2 telle que $c_2' \equiv c_2[\theta]$. On obtient de même $\Delta \vdash_? c_1' : X' \rhd X$ avec $c_1' \equiv c_1[\rho]$.
En utilisant une coercion de contextes partout l'identité, on obtient par induction avec les dérivations de c_1' et d_1 d'une part et c_2' et d_2 d'autre part, deux coercions e_1, e_2 telles que :

$$\Delta \vdash_? e_1 \equiv c_1[\rho] \circ d_1 : S \rhd X \wedge \Delta, x : S \vdash_? e_2 \equiv d_2 \circ c_2[\rho] : Y \rhd T$$

On en déduit :

$$\frac{\Delta \vdash_? e_1 : S \rhd X \qquad \Delta, x : S \vdash_? e_2 : Y \rhd T}{\Delta \vdash_? e = \lambda x : [\![S]\!]_\Delta . e_2[\bullet\ e_1[x]] : \Pi x : X.Y \rhd \Pi x : S.T}$$

On a bien $e \equiv d \circ c[\rho]$:

$$\begin{array}{ll}
 & d \circ c[\rho] \\
 & \quad \{\ \textbf{Définition de } c \textbf{ et } d\ \} \\
\triangleq & (\lambda x : [\![S]\!]_\Delta . d_2[\bullet\ d_1[x]]) \circ (\lambda y : [\![X']\!]_\Gamma . c_2[\bullet\ c_1[y]])[\rho] \\
 & \quad \{\ \textbf{Composition des contextes}\ \} \\
= & \lambda x : [\![S]\!]_\Delta . d_2[(\lambda y : [\![X']\!]_\Gamma . c_2[\bullet\ c_1[y]])[\rho]\ d_1[x]] \\
 & \quad \{\ \textbf{Substitution dans l'abstraction}\ \} \\
= & \lambda x : [\![S]\!]_\Delta . d_2[(\lambda y : [\![X']\!]_\Gamma [\rho] . c_2[\bullet\ c_1[y]][\rho])\ d_1[x]] \\
 & \quad \{\ \textbf{Réduction}\ \} \\
\rightarrow_\beta & \lambda x : [\![S]\!]_\Delta . d_2[(c_2[\bullet\ c_1[y]][\rho])[d_1[x]/y]] \\
 & \quad \{\ y \notin \rho\ \} \\
= & \lambda x : [\![S]\!]_\Delta . d_2[(c_2[\rho][\bullet\ c_1[\rho][y]])[d_1[x]/y]] \\
 & \quad \{\ \textbf{Définition de } \theta, \ y \notin d1[x]\ \} \\
\triangleq & \lambda x : [\![S]\!]_\Delta . d_2[c_2[\theta][\bullet\ c_1[\theta][d_1[x]]]] \\
 & \quad \{\ d_2 \circ c_2[\theta] \equiv e_2\ \} \\
\equiv & \lambda x : [\![S]\!]_\Delta . e_2[\bullet\ c_1[\theta][d_1[x]]] \\
 & \quad \{\ x \notin c_1 \Rightarrow c_1[\theta] = c_1[\rho]\ \} \\
\equiv & \lambda x : [\![S]\!]_\Delta . e_2[\bullet\ e_1[x]] \\
 & \quad \{\ \textbf{Définition}\ \} \\
\triangleq & e
\end{array}$$

- $\rhd$-Proof : Ici $U = \{\ y : U' \mid P\ \}$ et la dérivation commence par :

$$\frac{\Delta \vdash_? e : \Pi x : X'.Y' \rhd U'}{\Delta \vdash_? d = (e, ?_{[\![P]\!]_\Delta [e/x]})_{[\![\{x:U'|P\}]\!]_\Delta} : \Pi x : X'.Y' \rhd \{\ y : U' \mid P\ \}}$$

Par induction on a $\Delta \vdash_? f \equiv e \circ c[\rho] : \Pi x : X.Y \rhd U'$. On peut donc dériver :

$$\frac{\Delta \vdash_? f : \Pi x : X.Y \rhd U'}{\Delta \vdash_? d' = (f, ?_{[\![P]\!]_\Delta [f/x]})_{[\![\{x:U'|P\}]\!]_\Delta} : \Pi x : X.Y \rhd U}$$

On peut vérifier que $d' \equiv d \circ c[\rho]$:

$$\begin{array}{rcl}
d \circ c[\rho] & \triangleq & ((e, ?_{[\![P]\!]_\Delta [e/x]})_{[\![\{x:U'|P\}]\!]_\Delta})[c[\rho]] \\
 & = & (e[c[\rho]], ?_{[\![P]\!]_\Delta [e[c[\rho]]/x]})_{[\![\{x:U'|P\}]\!]_\Delta} \\
 & \equiv & (f, ?_{[\![P]\!]_\Delta [f/x]})_{[\![\{x:U'|P\}]\!]_\Delta} \\
 & \triangleq & d'
\end{array}$$

– $\rhd$-Sum : De façon équivalente à $\rhd$-Prod, on fait le cas où $\rhd$-Sum est utilisée à la fin de la dérivation de d.

$$\Gamma \vdash_? c : T \rhd \Sigma x : X'.Y' \qquad \frac{\Delta \vdash_? d_1 : X' \rhd X \qquad \Delta, x : X' \vdash_? d_2 : Y' \rhd Y}{\Delta \vdash_? d : \Sigma x : X'.Y' \rhd \Sigma x : X.Y}$$

On a $d \triangleq (d_1[\pi_1\ \bullet], d_2[\pi_2\ \bullet][\pi_1\ \bullet/x])_{[\![\Sigma x:X.Y]\!]_\Delta}$.
Par induction sur la dérivation de $\Gamma \vdash_? c : T \rhd \Sigma x : X'.Y'$:

- $\rhd$-Subset : On a :

$$\frac{\Gamma \vdash_? c_1 : T \rhd \Sigma x : X'.Y'}{\Gamma \vdash_? c = c_1[\pi_1\ \bullet] : \{\ y : T \mid P\ \} \rhd \Sigma x : X'.Y'}$$

  Par induction, $\Delta \vdash_? f \equiv d \circ c_1[\rho] : T \rhd \Sigma x : X.Y$, on a donc :

$$\frac{\Delta \vdash_? f \equiv d \circ c_1[\rho] : T \rhd \Sigma x : X.Y}{\Delta \vdash_? c' = f[\pi_1\ \bullet] : \{\ y : T \mid P\ \} \rhd \Sigma x : X.Y}$$

  On a bien :

$$\begin{array}{rcl} d \circ c[\rho] & \equiv & d \circ c_1[\pi_1\ \bullet][\rho] \\ & = & d \circ c_1[\rho][\pi_1\ \bullet] \\ & \equiv & f[\pi_1\ \bullet] \\ & \triangleq & c' \end{array}$$

- $\rhd$-Sum : On a :

$$\frac{\Gamma \vdash_? c_1 : S \rhd X' \qquad \Gamma, x : S \vdash_? c_2 : T \rhd Y'}{\Gamma \vdash_? c = (c_1[\pi_1\ \bullet], c_2[\pi_1\ \bullet/x][\pi_2\ \bullet])_{[\![\Sigma x:X'.Y']\!]_\Gamma} : \Sigma x : S.T \rhd \Sigma x : X'.Y'}$$

  Par affaiblissement pour les coercions (lemme 3.2.11) on a $\Delta \vdash_? c'_1 : S \rhd X'$ avec $c'_1 \equiv c_1[\rho]$. On peut aussi construire la coercion de contexte $\theta = \rho, \bullet : (\Delta, x : S) \rhd (\Gamma, x : S)$. Par le même lemme on obtient $\Delta, x : S \vdash_? c'_2 : T \rhd Y'$ avec $c'_2 \equiv c_2[\theta] = c_2[\rho]$. On peut enfin construire le jugement $\Delta, x : S \vdash_? d'_2 : Y' \rhd Y$ avec $d'_2 \equiv d_2[c'_1[x]/x]$.

  Par induction en utilisant une coercion de contexte partout l'identité on a donc :

$$\frac{\Delta \vdash_? e_1 \equiv d_1 \circ c_1[\rho] : S \rhd X \qquad \Delta, x : S \vdash_? e_2 \equiv d'_2 \circ c'_2 : T \rhd Y}{\Delta \vdash_? e = (e_1[\pi_1\ \bullet], e_2[\pi_1\ \bullet/x][\pi_2\ \bullet])_{[\![\Sigma x:X.Y]\!]_\Delta} : \Sigma x : S.T \rhd \Sigma x : X.Y}$$

On peut vérifier :

$$
\begin{array}{lll}
 & d \circ c[\rho] & \\
 & \quad \{ \textbf{ Définition de } d \textbf{ } \} & \\
\triangleq & (d_1[\pi_1\ \bullet], d_2[\pi_1\ \bullet/x][\pi_2\ \bullet])_{[\![\Sigma x{:}X.Y]\!]_\Delta}[c[\rho]] & \\
 & \quad \{ \textbf{ Substitution } \} & \\
= & (d_1[\pi_1\ c[\rho]], d_2[\pi_1\ c[\rho]/x][\pi_2\ c[\rho]])_{[\![\Sigma x{:}X.Y]\!]_\Delta} & \\
 & \quad \{ \textbf{ Réduction } \} & \\
\rightarrow_\pi & (d_1[c_1[\rho][\pi_1\ \bullet]], d_2[(c_1[\pi_1\ \bullet])[\rho]/x][c_2[\rho][\pi_1\ \bullet/x][\pi_2\ \bullet]])_{[\![\Sigma x{:}X.Y]\!]_\Delta} & \\
 & \quad \{ \textbf{ Définition de } e_1 \textbf{ } \} & \\
\equiv & (e_1[\pi_1\ \bullet], d_2[c_1[\rho][\pi_1\ \bullet]/x][c_2[\rho][\pi_1\ \bullet/x][\pi_2\ \bullet]])_{[\![\Sigma x{:}X.Y]\!]_\Delta} & \\
 & \quad \{ \textbf{ Définition de } c_1' \textbf{ } \} & \\
\equiv & (e_1[\pi_1\ \bullet], d_2[c_1'[\pi_1\ \bullet]/x][c_2[\rho][\pi_1\ \bullet/x][\pi_2\ \bullet]])_{[\![\Sigma x{:}X.Y]\!]_\Delta} & \\
 & \quad \{ \textbf{ Définition de } d_2' \textbf{ } \} & \\
\equiv & (e_1[\pi_1\ \bullet], d_2'[\pi_1\ \bullet/x][c_2[\rho][\pi_1\ \bullet/x][\pi_2\ \bullet]])_{[\![\Sigma x{:}X.Y]\!]_\Delta} & \\
 & \quad \{ \textbf{ } x \notin c_2[\pi_1\ \bullet/x] \textbf{ } \} & \\
\equiv & (e_1[\pi_1\ \bullet], d_2'[c_2[\rho][\pi_2\ \bullet]][\pi_1\ \bullet/x])_{[\![\Sigma x{:}X.Y]\!]_\Delta} & \\
 & \quad \{ \textbf{ Définition de } c_2' \textbf{ } \} & \\
\equiv & (e_1[\pi_1\ \bullet], d_2'[c_2'[\pi_2\ \bullet]][\pi_1\ \bullet/x])_{[\![\Sigma x{:}X.Y]\!]_\Delta} & \\
 & \quad \{ \textbf{ Définition de } e_2 \textbf{ } \} & \\
\equiv & (e_1[\pi_1\ \bullet], e_2'[\pi_2\ \bullet][\pi_1\ \bullet/x])_{[\![\Sigma x{:}X.Y]\!]_\Delta} & \\
 & \quad \{ \textbf{ } x \notin \pi_2\ \bullet \textbf{ } \} & \\
= & (e_1[\pi_1\ \bullet], e_2[\pi_1\ \bullet/x][\pi_2\ \bullet])_{[\![\Sigma x{:}X.Y]\!]_\Delta} & \\
 & \quad \{ \textbf{ Définition de } e \textbf{ } \} & \\
\triangleq & e &
\end{array}
$$

– $\rhd$-Proof :

$$\frac{\Gamma \vdash_? e : S \rhd T}{\Gamma \vdash_? c_1 = (e, ?_{[\![P]\!]_{\Gamma, x:T[e/x]}})_{[\![\{x:T|P\}]\!]_\Gamma} : S \rhd \{\ x : T \mid P\ \}} \qquad \Delta \vdash_? c_2 : \{\ x : T \mid P\ \} \rhd U$$

Par induction sur la dérivation de c_2.

- $\rhd$-Subset : On a une dérivation $\Delta \vdash_? c_2' : T \rhd U$, donc par induction on a $\Delta \vdash_? c' : S \rhd U$ avec $c' \equiv c_2' \circ e[\rho]$.
  On a bien $c' = c_2 \circ c_1[\rho]$:

$$
\begin{array}{rll}
c_2 \circ c_1[\rho] & \equiv & c_2'[\pi_1\ \bullet] \circ c_1[\rho] \\
 & = & c_2'[\pi_1\ \bullet] \circ ((e, ?_{[\![P]\!]_{\Gamma, x:T[e/x]}})_{[\![\{x:T|P\}]\!]_\Gamma})[\rho] \\
 & \rightarrow_{\sigma_1} & c_2'[e[\rho]] \\
 & \equiv & c'
\end{array}
$$

- $\rhd$-Proof : Alors on a :

$$\frac{\Delta \vdash_? c_2' : \{\ x : T \mid P\ \} \rhd U'}{\Delta \vdash_? c_2 = (c_2', ?_{[\![P']\!]_{\Delta, x:U'[c_2'/x]}})_{[\![\{x:U'|P'\}]\!]_\Delta} : \{\ x : T \mid P\ \} \rhd \{\ x : U' \mid P'\ \}}$$

On peut appliquer l'hypothèse d'induction pour obtenir une dérivation de $\Delta \vdash_? d \equiv c_2' \circ c_1[\rho] : S \rhd U'$ et par application de $\rhd$-Proof on obtient une coercion c' :

$$\begin{aligned} c_2 \circ c_1[\rho] &\triangleq ((c_2', ?_{[\![P']\!]_{\Delta,x:U'}[c_2'/x]})_{[\![\{x:U'|P'\}]\!]_\Delta}) \circ c_1[\rho] \\ &= (c_2'[c_1[\rho]], ?_{[\![P']\!]_{\Delta,x:U'}[c_2'[c_1[\rho]]/x]})_{[\![\{x:U'|P'\}]\!]_\Delta} \\ &\equiv (d, ?_{[\![P']\!]_{\Delta,x:U'}[d/x]})_{[\![\{x:U'|P'\}]\!]_\Delta} \\ &\triangleq c' \end{aligned}$$

– $\rhd$-Subset :

$$\frac{\Gamma \vdash_? c : S' \rhd T}{\Gamma \vdash_? d = c[\pi_1\ \bullet] : S = \{\ x : S' \mid P\ \} \rhd T} \qquad \Delta \vdash_? e : T \rhd U$$

Par induction il existe une dérivation de $\Delta \vdash_? f \equiv e \circ c[\rho] : S' \rhd U$. On peut donc dériver :

$$\frac{\Delta \vdash_? f : S' \rhd U}{\Delta \vdash_? f[\pi_1\ \bullet] : S = \{\ x : S' \mid P\ \} \rhd U}$$

On a bien $f[\pi_1\ \bullet] \equiv (e \circ c[\rho])[\pi_1\ \bullet] = e \circ c[\rho][\pi_1\ \bullet] = e \circ c[\pi_1\ \bullet][\rho] \triangleq e \circ d[\rho]$.
Dans le cas où $\rhd$-Subset est utilisée à droite, la seule règle applicable à gauche est $\rhd$-Proof et l'on a déjà traité ce cas.

□

Dans la version spécialisée, on a encore l'équivalence entre la composition des coercions initiales et la coercion composée.

Corollaire 3.2.13 (Transitivité de la coercion)**.** *Si* $\Gamma \vdash_? c_1 : S \rhd T$ *et* $\Gamma \vdash_? c_2 : T \rhd U$ *alors il existe* $c \equiv c_2 \circ c_1$ *tel que* $\Gamma \vdash_? c : S \rhd U$.

On peut maintenant énoncer le lemme de symétrie, qu'on prouve avec la transitivité :

Lemme 3.2.14 (Symétrie de la coercion)**.** *S'il existe* c *tel que* $\Gamma \vdash_? c : A \rhd B$ *alors* $\exists! c^{-1}, \Gamma \vdash_? c^{-1} : B \rhd A$ *et* $c^{-1} \circ c \equiv \bullet \equiv c \circ c^{-1}$.

Démonstration. On sait que le jugement $\rhd$ est symétrique, c'est à dire qu'on a l'existence des inverses. On utilise la transitivité pour montrer le reste du lemme. Si $\Gamma \vdash_? c : A \rhd B$ et $\Gamma \vdash_? c^{-1} : B \rhd A$ alors il existe des coercions f et f^{-1} telles que $\Gamma \vdash_? f \equiv c^{-1} \circ c : A \rhd A$ et $\Gamma \vdash_? f^{-1} \equiv c \circ c^{-1} : B \rhd B$. Par réflexivité (lemme 3.2.2) et unicité des coercions, $f \equiv \bullet$ et $f^{-1} \equiv \bullet$. □

On peut maintenant s'attaquer à l'affaiblissement d'une dérivation de typage par une coercion de contexte. Une nouvelle coercion α témoigne des coercions dans le terme dues au changement de contexte.

Lemme 3.2.15 (Stabilité par affaiblissement)**.** *Si* $\Gamma \vdash_\bullet t : T$ *alors pour tout* Δ, $\rho : \Delta \rhd \Gamma$, $\Delta \vdash_\bullet t : T'$ *et il existe* α, $\Delta \vdash_? \alpha : T' \rhd T$ *avec* $[\![t]\!]_\Gamma[\rho] \equiv \alpha[[\![t]\!]_\Delta]$.

Démonstration. La première partie du lemme se déduit par applications répétées du lemme de restriction 2.3.14.

On va utiliser à plusieurs reprises le résultat suivant : Si une prémisse du jugement sur lequel on fait la récurrence est de la forme $\Gamma \vdash_\bullet T : s$, alors on peut appliquer l'hypothèse d'induction pour obtenir $\Delta \vdash_\bullet T : s'$ avec α tel que $\Delta \vdash_? \alpha : s \rhd s'$ et $[\![T]\!]_\Gamma[\rho] \equiv \alpha[[\![T]\!]_\Delta]$. Or comme s est une sorte, on a $s' = s$ et $\alpha = \bullet$. On a donc $[\![T]\!]_\Gamma[\rho] \equiv [\![T]\!]_\Delta$.

Par induction sur la dérivation de t.

– Var : On a :

$$\frac{\vdash_\bullet \Gamma \qquad y : T \in \Gamma}{\Gamma \vdash_\bullet y : T}$$

Par définition de la coercion de contextes, il existe $c : (T' \rhd T) \in \rho$. Soit $\alpha = c$, on a bien : $[\![y]\!]_\Gamma[\rho] = c[y] = \alpha[[\![y]\!]_\Delta]$.

– App : On a :

$$\frac{\Gamma \vdash_\bullet f : F \quad \mu_\bullet(F) = \Pi y : A.B \qquad \Gamma \vdash_\bullet e : E \quad \Gamma \vdash_\bullet E, A : s \qquad E \rhd A}{\Gamma \vdash_\bullet (f\ e) : B[e/y]}$$

et donc par induction, $\Delta \vdash_\bullet f : F'$ avec $\Delta \vdash_? \alpha_f : F' \rhd F$ et $\Delta \vdash_\bullet e : E'$ avec $\Delta \vdash_? \alpha_e : E' \rhd E$.

Par définition de l'interprétation,

$$\begin{aligned} [\![(f\ e)]\!]_\Gamma &= (((\pi_F[[\![f]\!]_\Gamma])\ (c_e[[\![e]\!]_\Gamma])) \\ &\text{où} \\ \pi_F &= \mathbf{coerce}_\Gamma\ F\ \Pi y : A.B \\ c_e &= \mathbf{coerce}_\Gamma\ E\ A. \end{aligned}$$

De même :

$$\begin{aligned} [\![f\ e]\!]_\Delta &= (\pi_{F'}[[\![f']\!]_\Delta])\ (c_{e'}[[\![e']\!]_\Delta]) \\ &\text{où} \\ \pi_{F'} &= \mathbf{coerce}_\Delta\ F'\ \Pi y : A'.B' \\ c_{e'} &= \mathbf{coerce}_\Delta\ E'\ A' \end{aligned}$$

On a $\Gamma \vdash_? \pi_F : F \rhd \Pi y : A.B$, donc par le lemme 3.2.11 on obtient $\Delta \vdash_? \pi_F[\rho] : F \rhd \Pi y : A.B$. On peut appliquer ce lemme puisque $[\![\Pi y : A.B]\!]_\Delta \equiv [\![\Pi y : A.B]\!]_\Gamma[\rho]$ par induction (α est nécessairement l'identité).

On a donc les coercions suivantes dans l'environnement Δ :

$$\begin{array}{ccc} F & \xrightarrow[\pi_F[\rho]]{} & \Pi y : A.B \\ {\scriptstyle \alpha_f}\uparrow & & \uparrow{\scriptstyle c_f} \\ F' & \xrightarrow{\pi_{F'}} & \Pi y : A'.B' \end{array}$$

Par symétrie et transitivité de la coercion, on en déduit qu'il existe c_f, $\Delta \vdash_? c_f : \Pi y : A'.B' \rhd \Pi y : A.B$. La coercion c_f est nécessairement de la forme : $\lambda y : [\![A]\!]_\Delta.c_2[\bullet\ c_1[y]]$ où $\Delta \vdash_? c_1 : A \rhd A'$ et $\Delta, x : A \vdash_? c_2 : B' \rhd B$.

Par le lemme 3.2.11 on a $\Delta \vdash_? c_e[\rho] : E \rhd A$, de même que précédemment on obtient la condition nécessaire par induction sur la dérivation de $\Gamma \vdash_\bullet A : s$.

On a donc les coercions suivantes dans l'environnement Δ :

$$\begin{array}{ccc} E & \xrightarrow[c_e[\rho]]{} & A \\ {\scriptstyle \alpha_e}\uparrow & & \downarrow{\scriptstyle c_1} \\ E' & \overset{c_{e'}}{\dashrightarrow} & A' \end{array}$$

Par transitivité, il existe donc une coercion $\Delta \vdash_? c_{E',A} \equiv c_e[\rho] \circ \alpha_e : E' \rhd A$. Par le lemme 3.2.11 en utilisant la coercion de contexte $\bullet, \ldots, c_{E',A} : (\Delta, x : E') \rhd \Delta, x : A$ et la dérivation de c_2 on a donc : $\Delta, x : E' \vdash_? c_2' : B' \rhd B$ avec $c_2' \equiv c_2[(c_e[\rho])[\alpha_e[x]]/x]$.

Par le lemme 3.2.7 avec $\Delta \vdash_? e : E'$ et la dérivation de c_2' on obtient : $\Delta \vdash_? c_2'' : B'[e/x] \rhd B[e/x]$ avec

$$c_2'' \equiv c_2'[[\![e]\!]_\Delta/x] \equiv c_2[(c_e[\rho])[\alpha_e[[\![e]\!]_\Delta]]/x]$$

Soit $\alpha = c_2''$, on peut vérifier :

$$\begin{array}{ll}
& \alpha[[\![f\ e]\!]_\Delta] \\
& \quad \textbf{\{ Définition de l'interprétation \}} \\
\triangleq & \alpha[(\pi_{F'}[[\![f]\!]_\Delta])[c_{e'}[[\![e]\!]_\Delta]]] \\
& \quad \textbf{\{ Définitions de } \alpha \textbf{ et } c_{e'} \textbf{ \}} \\
\equiv & c_2[c_e[\rho]\ [(\alpha_e[[\![e]\!]_\Delta])]/x]\ [((\pi_{F'}[[\![f]\!]_\Delta])\ [(c_1 \circ c_e[\rho] \circ \alpha_e)[[\![e]\!]_\Delta]])] \\
& \quad \textbf{\{ Définition de la composition \}} \\
= & c_2[c_e[\rho]\ [(\alpha_e[[\![e]\!]_\Delta])]/x]\ [((\pi_{F'}[[\![f]\!]_\Delta])\ [c_1[c_e[\rho][\alpha_e[[\![e]\!]_\Delta]]]])] \\
& \quad \textbf{\{ Définition de } c_f\ (= \lambda x.c_2[\bullet\ c_1[x]]) \textbf{ \}} \\
= & c_f[(\pi_{F'}[[\![f]\!]_\Delta])\ [c_e[\rho]\ [\alpha_e[[\![e]\!]_\Delta]]]] \\
& \quad \textbf{\{ Commutation du diagramme pour la fonction \}} \\
\equiv & (\pi_F[\rho][\alpha_f[[\![f]\!]_\Delta]]\ [c_e[\rho][\alpha_e\ [\![e]\!]_\Delta]]) \\
& \quad \textbf{\{ Définitions de } \alpha_f \textbf{ et } \alpha_e \textbf{ \}} \\
\equiv & (\pi_F[\rho][[\![f]\!]_\Gamma[\rho]])\ [c_e[\rho][[\![e]\!]_\Gamma[\rho]]] \\
& \quad \textbf{\{ Définition de l'interprétation \}} \\
\triangleq & [\![f\ e]\!]_\Gamma[\rho]
\end{array}$$

– Prod : On a

$$\frac{\Gamma \vdash_\bullet T : s_1 \qquad \Gamma, x : T \vdash_\bullet U : s_2}{\Gamma \vdash_\bullet \Pi x : T.U : s_3}\ (s_1, s_2, s_3) \in \mathcal{R}$$

Par induction, $\Delta \vdash_\bullet T : s_1$ et $\Delta, x : T \vdash_\bullet U : s_2$ (en utilisant une coercion identité ajoutée à ρ). On peut donc appliquer Prod pour obtenir le résultat désiré avec $\alpha = \bullet$. De même pour Sum, Subset.

– Abs : On a

$$\frac{\Gamma \vdash_\bullet \Pi x : T.U : s \qquad \Gamma, x : T \vdash_\bullet M : U}{\Gamma \vdash_\bullet \lambda x : T.M : \Pi x : T.U}$$

Par induction, $\Delta \vdash_\bullet \Pi x : T.U : s$ et il existe U', $\Delta, x : T \vdash_\bullet M : U'$ avec une coercion $\Delta, x : T \vdash_? \alpha' : U' \rhd U$ telle que $[\![M]\!]_{\Gamma, x:T}[\rho, \bullet] \equiv \alpha'[[\![M]\!]_{\Delta, x:T}]$.

Soit α la coercion : $\Delta \vdash_? \lambda x : [\![T]\!]_\Delta.\alpha'[\bullet\ x] : \Pi x : T.U' \rhd \Pi x : T.U$. On a bien :

$$\begin{array}{rl} & \alpha[[\![\lambda x : T.M]\!]_\Delta] \\ & \quad \textbf{\{ Définition de l'interprétation \}} \\ \triangleq & \alpha[\lambda x : [\![T]\!]_\Delta.[\![M]\!]_{\Delta,x:T}] \\ & \quad \textbf{\{ Définition de la coercion } \alpha \textbf{ \}} \\ \triangleq & \lambda x : [\![T]\!]_\Delta.\alpha'[[\![M]\!]_{\Delta,x:T}[x/x]] \\ & \quad \textbf{\{ Définition de le coercion } \alpha' \textbf{ \}} \\ \equiv & \lambda x : [\![T]\!]_\Delta.[\![M]\!]_{\Gamma,x:T}[\rho] \\ & \quad \textbf{\{ Hypothèse d'induction, } [\![T]\!]_\Delta = [\![T]\!]_\Gamma[\rho] \textbf{ \}} \\ \equiv & \lambda x : [\![T]\!]_\Gamma[\rho].[\![M]\!]_{\Gamma,x:T}[\rho] \\ & \quad \textbf{\{ Définition de la substitution \}} \\ = & (\lambda x : [\![T]\!]_\Gamma.[\![M]\!]_{\Gamma,x:T})[\rho] \end{array}$$

– Pair : On a :

$$\frac{\Gamma \vdash_\bullet \Sigma x : T.U : s \qquad \Gamma \vdash_\bullet t : T' \rhd T : s \qquad \Gamma \vdash_\bullet u : U' \rhd U[t/x] : s}{\Gamma \vdash_\bullet (t,u)_{\Sigma x:T.U} : \Sigma x : T.U}$$

On a donc les coercions : $\Gamma \vdash_? \alpha_t : T' \rhd T$ et $\Gamma \vdash_? \alpha_u : U' \rhd U[t/x]$. On peut les faire passer dans l'environnement Δ par le lemme 3.2.11 (la condition est vérifiée en utilisant la dérivation de $\Gamma \vdash_\bullet \Sigma x : T.U : s$). On obtient :

$$\Delta \vdash_? \alpha_t[\rho] : T' \rhd T \wedge \Delta \vdash_? \alpha_u[\rho] : U' \rhd U[t/x]$$

Par induction :

$$\frac{\Delta \vdash_\bullet \Sigma x : T.U : s \qquad \Delta \vdash_\bullet t : T'' \quad T'' \rhd T \qquad \Delta \vdash_\bullet u : U'' \qquad \Delta \vdash_\bullet U'' : s \quad U'' \rhd U[t/x]}{\Delta \vdash_\bullet (t,u)_{\Sigma x:T.U} : \Sigma x : T.U}$$

Avec les coercions :

$$\Delta \vdash_? \beta_t : T'' \rhd T', \beta_t[[\![t]\!]_\Delta] \equiv [\![t]\!]_\Gamma[\rho]$$
$$\Delta \vdash_? \beta_u : U'' \rhd U', \beta_u[[\![u]\!]_\Delta] \equiv [\![u]\!]_\Gamma[\rho]$$

Par transitivité de la coercion on peut construire les coercions

$$\Delta \vdash_? \chi_t \equiv \alpha_t[\rho] \circ \beta_t : T'' \rhd T \wedge \Delta \vdash_? \chi_u = \alpha_u[\rho] \circ \beta_u : U'' \rhd U$$

utilisées dans le jugement précédent.
Par réflexivité de la coercion, on a $\Delta \vdash_? \alpha \equiv \bullet : \Sigma x : T.U \rhd \Sigma x : T.U$.
On peut vérifier :

$$\begin{array}{rl} & \alpha[[\![(t,u)_{\Sigma x:T.U}]\!]_\Delta] \\ & \quad \textbf{\{ Définition de l'interprétation \}} \\ \triangleq & (\chi_t[[\![t]\!]_\Delta], \chi_u[[\![u]\!]_\Delta])_{[\![\Sigma x:T.U]\!]_\Delta} \\ & \quad \textbf{\{ Définition des coercions \}} \\ \triangleq & (\alpha_t[\rho][\beta_t[[\![t]\!]_\Delta]], \alpha_u[\rho][\beta_u[[\![u]\!]_\Delta]])_{[\![\Sigma x:T.U]\!]_\Delta} \\ & \quad \textbf{\{ Hypothèses sur les coercions \}} \\ \triangleq & (\alpha_t[\rho][[\![t]\!]_\Gamma[\rho]], \alpha_u[\rho][[\![u]\!]_\Gamma[\rho]])_{[\![\Sigma x:T.U]\!]_\Delta} \\ & \quad \textbf{\{ Définition de l'interprétation \}} \\ \triangleq & [\![(t,u)_{\Sigma x:T.U}]\!]_\Gamma[\rho] \end{array}$$

– Pi-1 :

$$\frac{\Gamma \vdash_\bullet t : S \qquad \mu_\bullet(S) = \Sigma x : T.U}{\Gamma \vdash_\bullet \pi_1\ t : T}$$

On nomme μ_S la coercion de S à $\Sigma x : T.U$. Par le lemme 3.2.11 on obtient : $\Delta \vdash_? \mu_S[\rho] : S \rhd \Sigma x : T.U$.
Par induction, il existe α_t, $\Delta \vdash_? \alpha_t : S' \rhd S$ et $\alpha_t[[\![t]\!]_\Delta] = [\![t]\!]_\Gamma[\rho]$. Par transitivité de la coercion, on a $\Delta \vdash_? \alpha_{S'} \equiv \mu_S[\rho] \circ \alpha_t : S' \rhd \Sigma x : T.U$. On en déduit qu'il existe T', U' tels que $\mu_\bullet(S') = \Sigma x : T'.U'$ et qu'il existe une coercion β telle que

$$\Delta \vdash_? \beta = (c_1[\pi_1\ \bullet], c_2[\pi_2\ \bullet][\pi_1\ \bullet/x])_{\Sigma x:T.U} : \Sigma x : T'.U' \rhd \Sigma x : T.U$$

avec $\beta \circ \mu_{S'} \equiv \mu_S[\rho] \circ \alpha_t$.
Soit $\alpha = c_1$, on a bien $\Delta \vdash_? \alpha : T' \rhd T$.
On a :

$$\begin{array}{ll}
& \alpha[[\![\pi_1\ t]\!]_\Delta] \\
& \quad \textbf{\{ Définition de l'interprétation \}} \\
\triangleq & c_1[\pi_1\ \mu_{S'}[[\![t]\!]_\Delta]] \\
& \quad \textbf{\{ } \pi_1\ \beta = c_1[\pi_1\ \bullet] \textbf{ \}} \\
= & (\pi_1\ \beta)[\mu_{S'}[[\![t]\!]_\Delta]] \\
& \quad \textbf{\{ Définition de la substitution dans les contextes \}} \\
= & \pi_1\ (\beta[\mu_{S'}[[\![t]\!]_\Delta]]) \\
& \quad \textbf{\{ Transitivité \}} \\
\equiv & \pi_1\ \mu_S[\rho][\alpha_t[[\![t]\!]_\Delta]] \\
& \quad \textbf{\{ Hypothèse d'induction \}} \\
\equiv & \pi_1\ \mu_S[\rho][[\![t]\!]_\Gamma[\rho]] \\
& \quad \textbf{\{ Définition de l'interprétation \}} \\
\triangleq & [\![\pi_1\ t]\!]_\Gamma[\rho]
\end{array}$$

– Pi-2 :

$$\frac{\Gamma \vdash_\bullet t : S \qquad \mu_\bullet(S) = \Sigma x : T.U}{\Gamma \vdash_\bullet \pi_2\ t : U[\pi_1\ t/x]}$$

On nomme μ_S la coercion de S à $\Sigma x : T.U$. Par le lemme 3.2.11 on obtient : $\Delta \vdash_? \mu_S[\rho] : S \rhd \Sigma x : T.U$.
On peut appliquer l'hypothèse d'induction sur $\Gamma \vdash_\bullet t : S$ pour dériver :

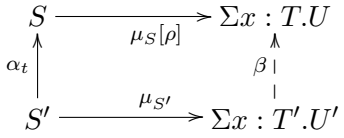
$$\frac{\Delta \vdash_\bullet t : S' \qquad \mu_\bullet(S') = \Sigma x : T'.U'}{\Delta \vdash_\bullet \pi_1\ t : T'}$$

Par induction, il existe α_t, $\Delta \vdash_? \alpha_t : S' \rhd S$ et $\alpha_t[[\![t]\!]_\Delta] = [\![t]\!]_\Gamma[\rho]$. Par transitivité de la coercion, on sait qu'il existe une coercion de S' à $\Sigma x : T.U$. On en déduit qu'il existe T', U' tels que $\mu_\bullet(S') = \Sigma x : T'.U'$ et qu'il existe une coercion β telle que

$$\Delta \vdash_? \beta = (c_1[\pi_1\ \bullet], c_2[\pi_2\ \bullet][\pi_1\ \bullet/x])_{\Sigma x:T.U} : \Sigma x : T'.U' \rhd \Sigma x : T.U$$

avec $\beta \circ \mu_{S'} \equiv \mu_S[\rho] \circ \alpha_t$.

On a donc les coercions suivantes dans l'environnement Δ :



Par substitution pour les coercions, avec les dérivation de $\pi_1\ t$ et de c_2 on obtient : $\Delta \vdash_? c_2' \equiv c_2[[\![\pi_1\ t]\!]_\Delta/x] : U'[\pi_1\ t/x] \rhd U[\pi_1\ t/x]$.
Soit $\alpha = c_2'$. On a :

$$
\begin{array}{rl}
 & \alpha[[\![\pi_2\ t]\!]_\Delta] \\
 & \quad \textbf{\{ Définition de } c_2' \textbf{ \}} \\
\equiv & c_2[[\![\pi_1\ t]\!]_\Delta/x][[\![\pi_2\ t]\!]_\Delta] \\
 & \quad \textbf{\{ Définition de l'interprétation \}} \\
\triangleq & c_2[\pi_1\ \mu_{S'}[[\![t]\!]_\Delta]/x][[\![\pi_2\ t]\!]_\Delta] \\
 & \quad \textbf{\{ Définition de l'interprétation \}} \\
\triangleq & c_2[\pi_1\ \mu_{S'}[[\![t]\!]_\Delta]/x][\pi_2\ \mu_{S'}[[\![t]\!]_\Delta]] \\
 & \quad \textbf{\{ } \bullet \notin [\![t]\!]_\Delta \wedge x \notin \mathcal{FV}(t) \textbf{ \}} \\
\triangleq & c_2[\pi_2\ \mu_{S'}[[\![t]\!]_\Delta]][\pi_1\ \mu_{S'}[[\![t]\!]_\Delta]/x] \\
 & \quad \textbf{\{ } \pi_2\ \beta = c_2[\pi_2\ \bullet][\pi_1\ \ \bullet/x] \textbf{ \}} \\
= & (\pi_2\ \beta)[\mu_{S'}[[\![t]\!]_\Delta]] \\
 & \quad \textbf{\{ Définition de la substitution dans les contextes \}} \\
= & \pi_2\ (\beta[\mu_{S'}[[\![t]\!]_\Delta]]) \\
 & \quad \textbf{\{ Transitivité \}} \\
\equiv & \pi_2\ \mu_S[\rho][\alpha_t[[\![t]\!]_\Delta]] \\
 & \quad \textbf{\{ Hypothèse d'induction \}} \\
\equiv & \pi_2\ \mu_S[\rho][[\![t]\!]_\Gamma[\rho]] \\
 & \quad \textbf{\{ Définition de l'interprétation \}} \\
\triangleq & [\![\pi_2\ t]\!]_\Gamma[\rho]
\end{array}
$$

□

On a besoin d'une spécialisation de ce lemme lorsque les types sont équivalents. Ce lemme montre que les jugements de typage et de coercion du système algorithmique sont stables par conversion et que l'interprétation l'est aussi.

Lemme 3.2.16 (Equivalence et interprétation)**.** *Si* $\Gamma \vdash_\bullet U, U' : s$ *et* $U \equiv_{\beta\pi} U'$ *alors :*
- *si* $\Gamma, x : U, \Delta \vdash_\bullet t : T$*, alors* $\Gamma, x : U', \Delta \vdash_\bullet t : T'$ *avec* $T \equiv_{\beta\pi} T'$ *et* $[\![t]\!]_{\Gamma,x:U,\Delta} \equiv [\![t]\!]_{\Gamma,x:U',\Delta}$ *;*
- *si* $\Gamma, x : U, \Delta \vdash_\bullet T \rhd T' : s$ *alors* $\Gamma, x : U', \Delta \vdash_\bullet T \rhd T' : s$*.*

Démonstration. Pour la première partie, la preuve suit le même schéma que la précédente. Il suffit de vérifier que l'on a toujours $\alpha \equiv \bullet$. Par exemple, dans le cas de l'application, on a $\alpha_f \equiv \alpha_e \equiv \bullet$. On en déduit que $c_f \equiv \bullet$ et $c_1 \equiv c_2 \equiv \bullet$. Clairement $c_2'' \equiv \bullet$.
Pour la seconde partie, c'est direct par induction en utilisant la première partie. □

On peut combiner les lemmes de substitution et affaiblissement en un lemme puissant. Si l'on substitue un terme u de type U pour une variable x déclarée de type V où $c : U \rhd V$,

alors on obtient un nouveau jugement et une nouvelle coercion α qui fait commuter la substitution de x par $c[[\![u]\!]]$.

Lemme 3.2.17 (Substitution et coercion)**.** *Si* $\Gamma \vdash_\bullet u : U$ *et* $\Gamma \vdash_? c : U \rhd V$ *alors*

$$\Gamma, x : V, \Delta \vdash_\bullet t : T \Rightarrow \begin{cases} \Gamma, \Delta[u/x] \vdash_\bullet t[u/x] : T' \\ \exists \alpha, \Gamma, \Delta[u/x] \vdash_? \alpha : T' \rhd T[u/x], \\ \alpha[[\![t[u/x]]\!]_{\Gamma,\Delta[u/x]}] \equiv [\![t]\!]_{\Gamma,x:V,\Delta}[c[[\![u]\!]_\Gamma]/x] \end{cases}$$

Démonstration. Si $\Gamma, x : V, \Delta \vdash_\bullet t : T$, alors soit ρ la coercion de contextes $\bullet, \ldots, c, \bullet, \ldots : (\Gamma, x : U, \Delta) \vdash_? (\Gamma, x : V, \Delta)$. Par le lemme 3.2.15, on a $\Gamma, x : U, \Delta \vdash_\bullet t : T'$ et il existe α,

$$\Gamma, x : U, \Delta \vdash_? \alpha : T' \rhd T \wedge [\![t]\!]_{\Gamma,x:V,\Delta}[\rho] \equiv \alpha[[\![t]\!]_{\Gamma,x:U,\Delta}]$$

Par substitutivité de la coercion, il existe $\alpha' \equiv \alpha[[\![u]\!]_\Gamma/x]$ telle que $\Gamma, \Delta[u/x] \vdash_? \alpha' : T'[u/x] \rhd T[u/x]$.

On a donc, par substitutivité de l'équivalence :

$$[\![t]\!]_{\Gamma,x:V,\Delta}[c[x]/x][[\![u]\!]_\Gamma/x] \equiv \alpha'[[\![t]\!]_{\Gamma,x:U,\Delta}][[\![u]\!]_\Gamma/x]$$

Comme $x \notin \mathcal{FV}(\alpha')$, on a $\alpha'[[\![t]\!]_{\Gamma,x:U,\Delta}][[\![u]\!]_\Gamma/x] = \alpha'[[\![t]\!]_{\Gamma,x:U,\Delta}[[\![u]\!]_\Gamma/x]]$.

Soit, par application du lemme de stabilité par substitution :

$$[\![t]\!]_{\Gamma,x:V,\Delta}[c[[\![u]\!]_\Gamma]/x] \equiv \alpha'[[\![t[u/x]]\!]_{\Gamma,\Delta[u/x]}]$$

On a donc bien construit la coercion α' recherchée. □

Son corollaire nous intéresse particulièrement pour la preuve de correction :

Corollaire 3.2.18 (Substitution dans un type et interprétation)**.** *Si* $\Gamma \vdash_\bullet u : U$, $\Gamma \vdash_? c : U \rhd V$ *et* $\Gamma, x : V \vdash_\bullet T : s$ *alors* $[\![T[u/x]]\!]_\Gamma \equiv [\![T]\!]_{\Gamma,x:V}[c[[\![u]\!]_\Gamma]/x]$.

3.2.4 Préservation de la conversion

On va maintenant s'attacher à montrer que l'équivalence du système algorithmique est préservée par interprétation. Pour cela, on définit une nouvelle relation qui contient à la fois la coercion et l'équivalence, qu'on appelle l'équivalence typée (figure 3.4). On considère sa clôture réflexive, symétrique et transitive.

Cette équivalence capture bien à la fois l'équivalence définitionelle et la coercion :

Proposition 3.2.19 (Equivalence typée)**.** *Si* $\Gamma \vdash_\bullet t : T$, $\Gamma \vdash_\bullet u : U$, *alors* $\Gamma \vdash_\bullet t : T \equiv_{\beta\pi} u : U$ *si et seulement si* $t \equiv_{\beta\pi} u$ *et* $U \rhd T$.

Démonstration. De gauche à droite, c'est trivial par induction. De droite à gauche : il suffit de considérer la relation $\rightarrow_{\beta\pi}$. En effet, si $t \equiv_{\beta\pi} u$ alors il existe v tel que $t \twoheadrightarrow_{\beta\pi} v$ et $u \twoheadrightarrow_{\beta\pi} v$ par confluence de la réduction. Pour tout Γ, t, T tels que $\Gamma \vdash_\bullet t : T$ on a par préservation du typage $\Gamma \vdash_\bullet t' : T'$ avec $T' \rhd T$ pour tout t' tel que $t \rightarrow_{\beta\pi} t'$. Il suffit alors de montrer que $\Gamma \vdash_\bullet t : T \equiv_{\beta\pi} t' : T'$. Par applications répétées de ce résultat et en utilisant la transitivité de l'équivalence typée, on obtient $\Gamma \vdash_\bullet t : T \equiv_{\beta\pi} v : V$ et $\Gamma \vdash_\bullet u : U \equiv_{\beta\pi} v : V$. On applique enfin symétrie et transitivité pour obtenir le résultat $\Gamma \vdash_\bullet t : T \equiv_{\beta\pi} u : U$.

On va donc montrer que pour tout Γ, t, T, $\Gamma \vdash_\bullet t : T$, pour tout u tel que $t \rightarrow_{\beta\pi} u$ et $\Gamma \vdash_\bullet u : U$ on a $\Gamma \vdash_\bullet t : T \equiv_{\beta\pi} u : U$. On a déjà par réduction du sujet que $U \rhd T$. Par induction sur la dérivation de typage de t.

$$\textsc{VarTeq}\ \frac{}{\Gamma \vdash_\bullet x : X \equiv_{\beta\pi} x : X}$$

$$\textsc{SortTeq}\ \frac{}{\Gamma \vdash_\bullet s : \texttt{Type} \equiv_{\beta\pi} s : \texttt{Type}}\ s \in \{\texttt{Set}, \texttt{Prop}\}$$

$$\beta\text{-}\equiv\ \frac{\Gamma \vdash_\bullet U \rhd T : s}{\Gamma \vdash_\bullet (\lambda x : X.v)\ e : T \equiv_{\beta\pi} v[e/x] : U}$$

$$\Pi_1\text{-}\equiv\ \frac{\Gamma \vdash_\bullet U \rhd T : s}{\Gamma \vdash_\bullet \pi_1\ (e_1, e_2)_{\Sigma x:T.V} : T \equiv_{\beta\pi} e_1 : U}$$

$$\Pi_2\text{-}\equiv\ \frac{\Gamma \vdash_\bullet U \rhd T : s}{\Gamma \vdash_\bullet \pi_2\ (e_1, e_2)_{\Sigma x:V.T} : T \equiv_{\beta\pi} e_2 : U}$$

$$\textsc{LambdaTeq}\ \frac{\Gamma \vdash_\bullet X : s_1 \equiv_{\beta\pi} X' : s_1 \qquad \Gamma, x : X' \vdash_\bullet v : Y \equiv_{\beta\pi} v' : Y'}{\Gamma \vdash_\bullet \lambda x : X.v : \Pi x : X.Y \equiv_{\beta\pi} \lambda x : X'.v' : \Pi x : X'.Y'}$$

$$\textsc{App}\text{-}\equiv\ \frac{\begin{array}{ccc}\Gamma \vdash_\bullet M : S \equiv_{\beta\pi} M' : T & & \Gamma \vdash_\bullet U \rhd A : s \\ \mu_\bullet(S) = \Pi x : A.B\ \ \mu_\bullet(T) = \Pi x : C.D & \Gamma \vdash_\bullet N : U \equiv_{\beta\pi} N' : V & \Gamma \vdash_\bullet V \rhd C : s\end{array}}{\Gamma \vdash_\bullet M\ N : B[N/x] \equiv_{\beta\pi} M'\ N' : D[N'/x]}$$

$$\textsc{Pair}\text{-}\equiv\ \frac{\begin{array}{ccc}\Gamma \vdash_\bullet e_1 : A' \equiv_{\beta\pi} e_1' : C' & \Gamma \vdash_\bullet A' \rhd A : s & \Gamma \vdash_\bullet B' \rhd B[e_1/x] : s \\ \Gamma, x : A' \vdash_\bullet e_2 : B' \equiv_{\beta\pi} e_2' : D' & \Gamma \vdash_\bullet C' \rhd C : s & \Gamma \vdash_\bullet D' \rhd D[e_1/x] : s\end{array}}{\Gamma \vdash_\bullet (e_1, e_2)_{\Sigma x:A.B} : \Sigma x : A.B \equiv_{\beta\pi} (e_1', e_2')_{\Sigma x:C.D} : \Sigma x : C.D}$$

$$\textsc{Prod}\text{-}\equiv\ \frac{\Gamma \vdash_\bullet C : s_1 \equiv_{\beta\pi} A : s_1 \qquad \Gamma, x : A \vdash_\bullet B : s_2 \equiv_{\beta\pi} D : s_2}{\Gamma \vdash_\bullet \Pi x : A.B : s_3 \equiv_{\beta\pi} \Pi x : C.D : s_3}$$

$$\textsc{Sigma}\text{-}\equiv\ \frac{\Gamma \vdash_\bullet A : s_1 \equiv_{\beta\pi} C : s_1 \qquad \Gamma, x : A \vdash_\bullet B : s_2 \equiv_{\beta\pi} D : s_2}{\Gamma \vdash_\bullet \Sigma x : A.B : s_3 \equiv_{\beta\pi} \Sigma x : C.D : s_3}$$

$$\textsc{Subset}\text{-}\equiv\ \frac{\Gamma \vdash_\bullet T : \texttt{Set} \equiv_{\beta\pi} U : \texttt{Set} \qquad \Gamma, x : T \vdash_\bullet P : \texttt{Prop} \equiv_{\beta\pi} P' : \texttt{Prop}}{\Gamma \vdash_\bullet \{\ x : T \mid P\ \} : \texttt{Set} \equiv_{\beta\pi} \{\ x : U \mid P'\ \} : \texttt{Set}}$$

Fig. 3.4: Définition du jugement $\Gamma \vdash_\bullet t : T \equiv_{\beta\pi} u : U$

– $(\lambda x : X.v)\ e \to_\beta v[e/x]$:
C'est la première règle, direct.

– π_1, π_2 : De même.

– $f\ e \to f'\ e$: Par inversion de $\Gamma \vdash_\bullet f\ e : T$ on a $\Gamma \vdash_\bullet f : F$, $\mu_\bullet(F) = \Pi x : A.B$ où $T = B[e/x]$, $\Gamma \vdash_\bullet e : E$ et $\Gamma \vdash_\bullet E \rhd A$. Par induction on a donc $\Gamma \vdash_\bullet f : F \equiv_{\beta\pi} f' : F'$ où $\Gamma \vdash_\bullet F' \rhd F$. Par le lemme sur la coercion et $\mu_\bullet()$ on a $\mu_\bullet(F') = \Pi x : A'.B'$ et $\Gamma \vdash_\bullet \Pi x : A'.B' \rhd \Pi x : A.B$. On a donc $\Gamma \vdash_\bullet A \rhd A'$ et par transitivité de la coercion $\Gamma \vdash_\bullet E \rhd A'$. On peut donc appliquer la règle App-$\equiv$. Pour obtenir :

$$\frac{\begin{array}{ccc}\Gamma \vdash_\bullet f : F \equiv_{\beta\pi} f' : F' & & \Gamma \vdash_\bullet E \rhd A \\ \mu_\bullet(F) = \Pi x : A.B \quad \mu_\bullet(F') = \Pi x : A'.B' & \Gamma \vdash_\bullet e : E \equiv_{\beta\pi} e : E & \Gamma \vdash_\bullet E \rhd A'\end{array}}{\Gamma \vdash_\bullet f\ e : B[e/x] \equiv_{\beta\pi} f'\ e : B'[e/x]}$$

– $f\ e \to f\ e'$: Par App-$\equiv$ on obtient $\Gamma \vdash_\bullet f\ e : B[e/x] \equiv_{\beta\pi} f\ e' : B[e'/x]$.

– $\lambda x : X.v \to \lambda x : X'.v$: Direct par LambdaTeq.

– $\lambda x : X.v \to \lambda x : X.v'$: Direct par LambdaTeq. On a $\Gamma, x : X \vdash_\bullet v : Y \equiv_{\beta\pi} v' : Y'$ par induction sur la prémisse $\Gamma, x : X \vdash_\bullet v : Y$.

$$\frac{\Gamma \vdash_\bullet X : s_1 \equiv_{\beta\pi} X : s_1 \qquad \Gamma, x : X \vdash_\bullet v : Y \equiv_{\beta\pi} v' : Y'}{\Gamma \vdash_\bullet \lambda x : X.v : \Pi x : X.Y \equiv_{\beta\pi} \lambda x : X.v' : \Pi x : X.Y'}$$

– $(e_1, e_2)_{\Sigma x:A.B} \to (e'_1, e_2)_{\Sigma x:A.B}$: Par application de Pair-≡. On a $\Gamma \vdash_\bullet e_1 : A' \equiv_{\beta\pi} e'_1 : C'$ par induction sur la prémisse $\Gamma \vdash_\bullet e_1 : A'$. On a $\Gamma \vdash_\bullet C' \rhd A'$ en utilisant l'autre sens de l'équivalence qu'on cherche à prouver. On en déduit $\Gamma \vdash_\bullet C' \rhd A$ par transitivité de la coercion avec $\Gamma \vdash_\bullet A' \rhd A$. D'autre part, comme $e_1 \equiv_{\beta\pi} e'_1$ et $B \equiv_{\beta\pi} B'$ on a $B[e_1/x] \equiv_{\beta\pi} B'[e'_1/x]$. Il existe donc une coercion de B' à $B[e_1/x]$.

$$\frac{\begin{array}{cc} \Gamma \vdash_\bullet e_1 : A' \equiv_{\beta\pi} e'_1 : C' & \Gamma \vdash_\bullet B' \rhd B[e_1/x] \\ \Gamma, x : A' \vdash_\bullet e_2 : B' \equiv_{\beta\pi} e_2 : B' \quad \Gamma \vdash_\bullet A' \rhd A \quad \Gamma \vdash_\bullet C' \rhd C & \Gamma \vdash_\bullet B' \rhd B[e'_1/x] \end{array}}{\Gamma \vdash_\bullet (e_1, e_2)_{\Sigma x:A.B} : \Sigma x : A.B \equiv_{\beta\pi} (e'_1, e_2)_{\Sigma x:A.B} : \Sigma x : A.B}$$

De façon équivalente pour la réduction dans la deuxième composante.

– Direct par induction pour les types produits, sommes et sous-ensemble.

□

Théorème 3.2.20 (Conservation de l'équivalence)**.** *Si* $\Gamma \vdash_\bullet t : T$, $\Gamma \vdash_\bullet u : U$, $t \equiv_{\beta\pi} u$ *et* $T \rhd U$ *alors il existe* α, $\alpha[[\![t]\!]_\Gamma] \equiv [\![u]\!]_\Gamma$ *où* $\Gamma \vdash_? \alpha : T \rhd U$.

Démonstration. Par induction sur la dérivation de $\Gamma \vdash_\bullet t : T \equiv_{\beta\pi} u : U$ obtenue par le lemme 3.2.19.

– Transitivité : On a $\Gamma \vdash_\bullet t : T \equiv_{\beta\pi} v : V$ et $\Gamma \vdash_\bullet v : V \equiv_{\beta\pi} t : T$, donc par induction, il existe α, β telles que : $\alpha[[\![t]\!]_\Gamma] \equiv [\![v]\!]_\Gamma$ et $\beta[[\![v]\!]_\Gamma] \equiv [\![u]\!]_\Gamma$ où $\Gamma \vdash_? \alpha : T \rhd V$ et $\Gamma \vdash_? \beta : V \rhd U$. Par transitivité de la coercion il existe $c \equiv \beta \circ \alpha$ telle que $\Gamma \vdash_? c : T \rhd U$. Clairement, $c[[\![t]\!]_\Gamma] \equiv \beta[\alpha[[\![t]\!]_\Gamma]] \equiv \beta[[\![v]\!]_\Gamma] \equiv [\![u]\!]_\Gamma$.

– Symétrie : On a $\Gamma \vdash_\bullet u : U \equiv_{\beta\pi} t : T$. Par induction, il existe α, $\alpha[[\![u]\!]_\Gamma] \equiv [\![t]\!]_\Gamma$ et $\Gamma \vdash_? \alpha : U \rhd T$. Par symétrie de la coercion, il existe $c \equiv \alpha^{-1}$, $\Gamma \vdash_? c : T \rhd U$ et $c \circ \alpha \equiv \alpha \circ c \equiv \bullet$. On a donc $c[\alpha[[\![u]\!]_\Gamma]] \equiv [\![u]\!]_\Gamma \equiv \alpha^{-1}[[\![t]\!]_\Gamma]$. Par symétrie de l'équivalence, on obtient le résultat désiré : $\alpha^{-1}[[\![t]\!]_\Gamma \equiv [\![u]\!]_\Gamma$.

– Réflexivité : On a $\Gamma \vdash_\bullet t : T \equiv_{\beta\pi} t : T$. Par réflexivité de la coercion, c'est trivial.

– VarTeq : De même, $\alpha = \bullet$.

– SortTeq : Trivial.

– LambdaTeq : Par induction on a $\Gamma \vdash_? \alpha = \bullet : s_1 \rhd s_1$, $[\![X]\!]_\Gamma \equiv [\![X']\!]_\Gamma$ et $\Gamma, x : X' \vdash_? \beta : Y \rhd Y'$ avec $\beta[[\![v]\!]_{\Gamma,x:X'}] = [\![v']\!]_{\Gamma,x:X'}$.
On obtient donc le jugement $\Gamma \vdash_? c = (\lambda x : [\![X']\!]_\Gamma.\beta[\bullet\ \bullet\ [x]]) : \Pi x : X.Y \rhd \Pi x : X'.Y'$ par $\rhd$-Prod. On a bien :

$$
\begin{array}{ll}
& c[[\![\lambda x : X.v]\!]_\Gamma] \\
& \quad \textbf{\{ Définition de l'interprétation \}} \\
\triangleq & c[\lambda x : [\![X]\!]_\Gamma.[\![v]\!]_{\Gamma,x:X}] \\
& \quad \textbf{\{ Définition de } c \textbf{ \}} \\
\equiv & \lambda x : [\![X']\!]_\Gamma.\beta[(\lambda x : [\![X]\!]_\Gamma.[\![v]\!]_{\Gamma,x:X})\ x] \\
& \quad \textbf{\{ Réduction \}} \\
\rightarrow_\beta & \lambda x : [\![X']\!]_\Gamma.\beta[[\![v]\!]_{\Gamma,x:X}] \\
& \quad \textbf{\{ Lemme 3.2.16 :} [\![v]\!]_{\Gamma,x:X} \equiv [\![v]\!]_{\Gamma,x:X'} \textbf{ \}} \\
\equiv & \lambda x : [\![X']\!]_\Gamma.\beta[[\![v]\!]_{\Gamma,x:X'}] \\
& \quad \textbf{\{ Hypothèse sur } \beta \textbf{ \}} \\
\equiv & \lambda x : [\![X']\!]_\Gamma.[\![v']\!]_{\Gamma,x:X'} \\
& \quad \textbf{\{ Définition de l'interprétation \}} \\
\equiv & [\![\lambda x : X'.v']\!]_\Gamma
\end{array}
$$

– β-$\equiv$: On a $[\![(\lambda x : X.v)\ e]\!]_\Gamma = (\lambda x : [\![X]\!]_\Gamma.[\![v]\!]_{\Gamma,x:X})\ c[[\![e]\!]_\Gamma] \rightarrow_\beta [\![v]\!]_{\Gamma,x:X}[c[[\![e]\!]_\Gamma]/x]$ où $\Gamma \vdash_\bullet e : E$, $\Gamma \vdash_? c : E \rhd X$. On doit montrer qu'il existe α tel que $[\![v]\!]_{\Gamma,x:X}[c[[\![e]\!]_\Gamma]/x] \equiv \alpha[[\![v[e/x]]\!]_]$. Par inversion de $\Gamma \vdash_\bullet (\lambda x : X.v)\ e : T$ on a $\Gamma, x : X \vdash_\bullet v : V$ tel que $V[e/x] = T$.

On applique le lemme de substitution et coercion 3.2.17 avec $\Gamma \vdash_\bullet e : E$, $\Gamma \vdash_? c : E \rhd X$ et cette dérivation. On obtient : $\alpha[[\![v[e/x]]\!]_\Gamma \equiv [\![v]\!]_{\Gamma,x:X}[c[[\![e]\!]_\Gamma]/x]$, soit le résultat désiré.

– Π_1-$\equiv$: On a $[\![\pi_1\ (e_1, e_2)_{\Sigma x:T.V}]\!]_\Gamma \triangleq \pi_1\ [\![(e_1, e_2)_{\Sigma x:T.V}]\!]_\Gamma = \pi_1\ ([\![e_1]\!]_\Gamma, [\![e_2]\!]_\Gamma)_{[\![\Sigma x:T.V]\!]_\Gamma} \rightarrow_{\pi_1} [\![e_1]\!]_\Gamma$. On peut donc prendre $\Gamma \vdash_? \alpha : T \rhd T \equiv \bullet$.

– Π_2-$\equiv$: On a $[\![\pi_2\ (e_1, e_2)_{\Sigma x:T.V}]\!]_\Gamma \triangleq \pi_2\ [\![(e_1, e_2)_{\Sigma x:T.V}]\!]_\Gamma = \pi_2\ ([\![e_1]\!]_\Gamma, [\![e_2]\!]_\Gamma)_{[\![\Sigma x:T.V]\!]_\Gamma} \rightarrow_{\pi_2} [\![e_1]\!]_\Gamma$. On peut donc prendre $\Gamma \vdash_? \alpha : T \rhd T \equiv \bullet$.

– App-$\equiv$: On a

$$
\frac{\Gamma \vdash_\bullet M : S \equiv_{\beta\pi} M' : T \qquad \mu_\bullet(S) = \Pi x : A.B\ \ \mu_\bullet(T) = \Pi x : C.D \qquad \Gamma \vdash_\bullet N : U \equiv_{\beta\pi} N' : V \qquad \Gamma \vdash_\bullet U \rhd As \qquad \Gamma \vdash_\bullet V \rhd Cs}{\Gamma \vdash_\bullet M\ N : B[N/x] \equiv_{\beta\pi} M'\ N' : D[N'/x]}
$$

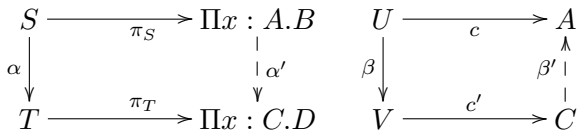
L'interprétation des termes est :

$$
\begin{array}{rcl}
[\![M\ N]\!]_\Gamma & \triangleq & \pi_S[[\![M]\!]_\Gamma]\ c[[\![N]\!]_\Gamma] \\
[\![M'\ N']\!]_\Gamma & \triangleq & \pi_T[[\![M']\!]_\Gamma]\ c'[[\![N']\!]_\Gamma] \\
\pi_S & = & \mathbf{coerce}_\Gamma\ S\ (\Pi x : A.B) \\
\pi_T & = & \mathbf{coerce}_\Gamma\ T\ (\Pi x : C.D) \\
c & = & \mathbf{coerce}_\Gamma\ U\ A \\
c' & = & \mathbf{coerce}_\Gamma\ V\ C
\end{array}
$$

Par induction, il exite α, β, telles que :

$$
\begin{array}{lll}
\Gamma \vdash_? \alpha : S \rhd T & \text{et} & \alpha[[\![M]\!]_\Gamma] \equiv [\![M']\!]_\Gamma \\
\Gamma \vdash_? \beta : U \rhd V & \text{et} & \beta[[\![N]\!]_\Gamma] \equiv [\![N']\!]_\Gamma
\end{array}
$$

On a donc les coercions suivantes :



La coercion α' est nécessairement de la forme $\lambda x : [\![C]\!]_\Gamma.c_2[\bullet\ c_1[x]]$ où $\Gamma \vdash_? c_1 = \beta' : C \rhd A$ et $\Gamma, x : C \vdash_? c_2 : B \rhd D$. Par le lemme 3.2.7 avec $\Gamma \vdash_\bullet N' : V$, $\Gamma \vdash_? c' : V \rhd C$ et $\Gamma, x : C \vdash_? c_2 : B \rhd D$ on a : $\Gamma \vdash_? c'_2 \equiv c_2[c'[[\![N']\!]_\Gamma/x] : B[N'/x] \rhd D[N'/x]$. Or comme $B[N'/x] \equiv_{\beta\pi} B[N/x]$ on a $\Gamma \vdash_? c'_2 : B[N/x] \rhd D[N'/x]$. Soit $\alpha = c'_2$.
On a bien :

$$
\begin{array}{rl}
 & \alpha[[\![M\ N]\!]_\Gamma] \\
 & \quad \textbf{\{ Définition de l'interprétation \}} \\
\triangleq & \alpha[\pi_S[[\![M]\!]_\Gamma]\ c[[\![N]\!]_\Gamma]] \\
 & \quad \textbf{\{ Définition de } \alpha = c'_2 \textbf{ \}} \\
\triangleq & c_2[c'[[\![N']\!]_\Gamma/x][\pi_S[[\![M]\!]_\Gamma]\ c[[\![N]\!]_\Gamma]] \\
 & \quad \textbf{\{ Hypothèse sur } \beta \textbf{ \}} \\
\equiv & c_2[c'[\beta[[\![N]\!]_\Gamma]/x][\pi_S[[\![M]\!]_\Gamma]\ c[[\![N]\!]_\Gamma]] \\
 & \quad \textbf{\{ Commutation du diagrame pour l'argument \}} \\
\equiv & c_2[c'[\beta[[\![N]\!]_\Gamma]/x][\pi_S[[\![M]\!]_\Gamma]\ (\beta'[c'[\beta[[\![N]\!]_\Gamma]]])] \\
 & \quad \textbf{\{ Définition de } \alpha' \textbf{ \}} \\
\equiv & \alpha'[\pi_S[[\![M]\!]_\Gamma]]\ (c'[\beta[[\![N]\!]_\Gamma]]) \\
 & \quad \textbf{\{ Commutation du diagramme pour la fonction \}} \\
\equiv & \pi_T[\alpha[[\![M]\!]_\Gamma]]\ (c'[\beta[[\![N]\!]_\Gamma]]) \\
 & \quad \textbf{\{ Hypothèses sur } \alpha \textbf{ et } \beta \textbf{ \}} \\
\equiv & \pi_T[[\![M']\!]_\Gamma]\ (c'[[\![N']\!]_\Gamma]) \\
 & \quad \textbf{\{ Définition de l'interprétation \}} \\
\triangleq & [\![M'\ N']\!]_\Gamma
\end{array}
$$

– Pair-$\equiv$:

$$
\frac{\Gamma \vdash_\bullet e_1 : A' \equiv_{\beta\pi} e'_1 : C' \qquad \Gamma, x : A' \vdash_\bullet e_2 : B' \equiv_{\beta\pi} e'_2 : D' \qquad \Gamma \vdash_\bullet A' \rhd A \quad \Gamma \vdash_\bullet C' \rhd C \qquad \Gamma \vdash_\bullet B' \rhd B[e_1/x] \qquad \Gamma \vdash_\bullet D' \rhd D[e'_1/x]}{\Gamma \vdash_\bullet (e_1, e_2)_{\Sigma x:A.B} : \Sigma x : A.B \equiv_{\beta\pi} (e'_1, e'_2)_{\Sigma x:C.D} : \Sigma x : C.D}
$$

On a

$$
\begin{array}{rcl}
[\![(e_1, e_2)_{\Sigma x:A.B}]\!]_\Gamma & = & (c_1[[\![e_1]\!]_\Gamma], c_2[[\![e_2]\!]_\Gamma])_{[\![\Sigma x:A.B]\!]_\Gamma} \\
[\![(e'_1, e'_2)_{\Sigma x:C.D}]\!]_\Gamma & = & (c'_1[[\![e'_1]\!]_\Gamma], c'_2[[\![e'_2]\!]_\Gamma])_{[\![\Sigma x:C.D]\!]_\Gamma} \\
c_1 & = & \mathsf{coerce}_\Gamma\ A'\ A \\
c'_1 & = & \mathsf{coerce}_\Gamma\ C'\ C \\
c_2 & = & \mathsf{coerce}_\Gamma\ B'\ B[e_1/x] \\
c'_2 & = & \mathsf{coerce}_\Gamma\ D'\ D[e'_1/x]
\end{array}
$$

Par induction il existe α, β telles que :

$$\Gamma \vdash_? \alpha : A' \rhd C' \quad \text{et} \quad \alpha[[\![e_1]\!]_\Gamma] \equiv [\![e'_1]\!]_\Gamma$$
$$\Gamma \vdash_? \beta : B' \rhd D' \quad \text{et} \quad \beta[[\![e_2]\!]_\Gamma] \equiv [\![e'_2]\!]_\Gamma$$

On a $\Gamma \vdash_\bullet \Sigma x : A.B \rhd \Sigma x : C.D$, on en déduit qu'il existe la dérivation suivante :

$$\frac{\Gamma \vdash_? d_1 : A \rhd C \qquad \Gamma, x : A \vdash_? d_2 : B \rhd D}{\Gamma \vdash_? d = (:, \Sigma)_{[\![\Sigma x:C.D]\!]_{d_1[\pi_1 \ \bullet]} d_2[\pi_2 \ \bullet][\pi_1 \ \bullet/x]} x : A.B \rhd \Sigma x : C.D}$$

Par symétrie et transitivité de la coercion on a $\Gamma \vdash_? d_1 \equiv c'_1 \circ \alpha \circ c_1^{-1} : A \rhd C$.
Par transitivité, il existe aussi une coercion d'_2 telle que $\Gamma \vdash_? d_2 : B[e_1/x] \rhd D[e'_1/x]$ et $d_2 \equiv c'_2 \circ \beta \circ c_2^{-1}$. Or par affaiblissement des coercions on a : $\Gamma, x : A' \vdash_? d_2[c_1[x]/x] : B \rhd D$. On peut appliquer le lemme de substitutivité pour les coercions afin d'obtenir une dérivation de $\Gamma, x : A \vdash_? d_2[c_1[[\![e_1]\!]_\Gamma]/x] : B[e_1/x] \rhd D[e_1/x]$. Par unicité des coercions, on a $d'_2[c_1[[\![e_1]\!]_\Gamma]/x] \equiv d_2$.
On peut vérifier :

$$\begin{aligned}
& d[[\![(e_1, e_2)_{\Sigma x:A.B}]\!]_\Gamma] \\
& \quad \textbf{\{ Définition de l'interprétation \}} \\
\triangleq\ & d[(c_1[[\![e_1]\!]_\Gamma], c_2[[\![e_2]\!]_\Gamma])_{[\![\Sigma x:A.B]\!]_\Gamma}] \\
& \quad \textbf{\{ Définition de } d \textbf{ \}} \\
\triangleq\ & (d_1[c_1[[\![e_1]\!]_\Gamma]], d_2[c_2[[\![e_2]\!]_\Gamma]][c_1[[\![e_1]\!]_\Gamma/x]])_{[\![\Sigma x:C.D]\!]_\Gamma} \\
& \quad \textbf{\{ Définition de } d_1 \textbf{ \}} \\
\triangleq\ & ((c'_1 \circ \alpha \circ c_1^{-1})[c_1[[\![e_1]\!]_\Gamma]], d_2[c_2[[\![e_2]\!]_\Gamma]][c_1[[\![e_1]\!]_\Gamma/x]])_{[\![\Sigma x:C.D]\!]_\Gamma} \\
& \quad \textbf{\{ } c_1^{-1} \circ c_1 \equiv \bullet \textbf{ \}} \\
\triangleq\ & (c'_1[\alpha[[\![e_1]\!]_\Gamma]], d_2[c_2[[\![e_2]\!]_\Gamma]][c_1[[\![e_1]\!]_\Gamma/x]])_{[\![\Sigma x:C.D]\!]_\Gamma} \\
& \quad \textbf{\{ Hypothèse sur } \alpha \textbf{ \}} \\
\triangleq\ & (c'_1[[\![e'_1]\!]_\Gamma], d_2[c_2[[\![e_2]\!]_\Gamma]][c_1[[\![e_1]\!]_\Gamma/x]])_{[\![\Sigma x:C.D]\!]_\Gamma} \\
& \quad \textbf{\{ Définition de la substitution, } x \notin e_2 \textbf{ \}} \\
\triangleq\ & (c'_1[[\![e'_1]\!]_\Gamma], d_2[c_1[[\![e_1]\!]_\Gamma/x]][c_2[[\![e_2]\!]_\Gamma]])_{[\![\Sigma x:C.D]\!]_\Gamma} \\
& \quad \textbf{\{ Définition de } d_2 \textbf{ \}} \\
\triangleq\ & (c'_1[[\![e'_1]\!]_\Gamma], d'_2[c_2[[\![e_2]\!]_\Gamma]])_{[\![\Sigma x:C.D]\!]_\Gamma} \\
& \quad \textbf{\{ Définition de } d'_2 \textbf{ \}} \\
\triangleq\ & (c'_1[[\![e'_1]\!]_\Gamma], (c'_2 \circ \beta \circ c_2^{-1})[c_2[[\![e_2]\!]_\Gamma]])_{[\![\Sigma x:C.D]\!]_\Gamma} \\
& \quad \textbf{\{ } c_2^{-1} \circ c_2 \equiv \bullet \textbf{ \}} \\
\triangleq\ & (c'_1[[\![e'_1]\!]_\Gamma], c'_2[\beta[[\![e_2]\!]_\Gamma]])_{[\![\Sigma x:C.D]\!]_\Gamma} \\
& \quad \textbf{\{ Hypothèse sur } \beta \textbf{ \}} \\
\triangleq\ & (c'_1[[\![e'_1]\!]_\Gamma], c'_2[[\![e'_2]\!]_\Gamma])_{[\![\Sigma x:C.D]\!]_\Gamma} \\
& \quad \textbf{\{ Définition de l'interprétation \}} \\
\triangleq\ & [\![(e'_1, e'_2)_{\Sigma x:C.D}]\!]_\Gamma
\end{aligned}$$

– ProdTeq, Sigma-≡, Subset-≡ : La traduction est un homomorphisme sur ces termes, c'est donc direct par induction.

□

Lorsque la coercion se réduit à une conversion, on a $\alpha \equiv \bullet$ et l'équivalence est bien préservée.

Corollaire 3.2.21 (Conservation de l'équivalence pour les types)**.** *Si* $\Gamma \vdash_\bullet T, U : s$ *et* $T \equiv_{\beta\pi} U$ *alors* $[\![T]\!]_\Gamma \equiv [\![U]\!]_\Gamma$.

3.2.5 Correction de l'interprétation

On peut maintenant montrer notre théorème de correction.

Théorème 3.2.22 (Correction de l'interprétation)**.** *Si* $\Gamma \vdash_\bullet t : T$ *alors on a* $[\![\Gamma]\!] \vdash_? [\![t]\!]_\Gamma : [\![T]\!]_\Gamma$. *Si* $\vdash_\bullet \Gamma$ *alors* $\vdash_\bullet [\![\Gamma]\!]$. *Si* $\Gamma \vdash_\bullet T, U : s$ *et* $T \rhd U$ *alors il existe c,* $\Gamma \vdash_? c : T \rhd U$ *et* $[\![\Gamma]\!] \vdash_? \lambda x : [\![T]\!].c[x] : [\![T]\!]_\Gamma \to [\![U]\!]_\Gamma$.

Démonstration. Par induction mutuelle sur les dérivations de typage, bonne formation et coercion.

- Wf-Empty : Trivial.
- Wf-Var : Par induction $[\![\Gamma]\!] \vdash_? [\![A]\!]_\Gamma : [\![s]\!]_\Gamma$. Par inversion du jugement de bonne formation dans CCI, $\vdash [\![\Gamma]\!]$. Or, $[\![s]\!]_\Gamma = s$ ($s \in \{\texttt{Prop}, \texttt{Set}, \texttt{Type}\}$), donc on peut appliquer Wf-Var dans CCI pour obtenir : $\vdash_\bullet [\![\Gamma]\!], x : [\![A]\!]_\Gamma$, soit $\vdash_\bullet [\![\Gamma, x : A]\!]$.
- PropSet : Direct par induction, $[\![\Gamma]\!] \vdash_? [\![s]\!]_\Gamma = s : [\![\texttt{Type}]\!]_\Gamma = \texttt{Type}$.
- Var : On a :

$$\frac{\vdash_\bullet \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash_\bullet x : A}$$

  Par induction, $\vdash_\bullet [\![\Gamma]\!]$ et par simple inspection de la définition de l'interprétation des contextes, si $x : A \in \Gamma$ alors $x : [\![A]\!]_\Delta \in [\![\Gamma]\!]$ pour $\Delta \subseteq \Gamma$. Par affaiblissement dans CCI, on obtient aisément $[\![\Gamma]\!] \vdash_? x : [\![A]\!]_\Delta$ à partir de $[\![\Delta]\!] \vdash_? x : [\![A]\!]_\Delta$. Or il est clair par la définition de l'interprétation que $[\![A]\!]_\Delta = [\![A]\!]_\Gamma$ puisque les variables libres de A sont strictement contenues dans Δ.
- Prod : On a :

$$\frac{\Gamma \vdash_\bullet T : s_1 \qquad \Gamma, x : T \vdash_\bullet U : s_2}{\Gamma \vdash_\bullet \Pi x : T.U : s_3} (s_1, s_2, s_3) \in \mathcal{R}$$

  Par induction $[\![\Gamma]\!] \vdash_? [\![T]\!]_\Gamma : [\![s_1]\!]_\Gamma = s_1$ et $[\![\Gamma, x : T]\!] \vdash_? [\![U]\!]_{[\![\Gamma, x:T]\!]} : [\![s_2]\!]_{[\![\Gamma,x:T]\!]} = s_2$. On déplie l'interprétation pour obtenir : $[\![\Gamma]\!], x : [\![T]\!]_\Gamma \vdash_? [\![U]\!]_{\Gamma,x:T} : s_2$.

  Par Prod dans CCI, on obtient : $[\![\Gamma]\!] \vdash_? \Pi x : [\![T]\!]_\Gamma.[\![U]\!]_{\Gamma,x:T} : s_3$. Or $[\![\Pi x : T.U]\!]_\Gamma = \Pi x : [\![T]\!]_\Gamma.[\![U]\!]_{\Gamma,x:T}$, donc on a bien : $[\![\Gamma]\!] \vdash_? [\![\Pi x : T.U]\!]_\Gamma : s_3 = [\![s_3]\!]_\Gamma$.
- Abs : On a :

$$\frac{\Gamma \vdash_\bullet \Pi x : T.U : s \qquad \Gamma, x : T \vdash_\bullet M : U}{\Gamma \vdash_\bullet \lambda x : T.M : \Pi x : T.U}$$

  Par induction $\Gamma \vdash_? [\![\Pi x : T.U]\!]_\Gamma : [\![s]\!]_\Gamma$ et $[\![\Gamma, x : T]\!] \vdash_? [\![M]\!]_{\Gamma,x:T} : [\![U]\!]_{\Gamma,x:T}$. On déplie l'interprétation pour obtenir : $[\![\Gamma]\!], x : [\![T]\!]_\Gamma \vdash_? [\![M]\!]_{\Gamma,x:T} : [\![U]\!]_{\Gamma,x:T}$.

  Par Abs dans CCI, on obtient : $[\![\Gamma]\!] \vdash_? \lambda x : [\![T]\!]_\Gamma.[\![M]\!]_{\Gamma,x:T} : [\![\Pi x : T.U]\!]_\Gamma$. C'est équivalent à : $[\![\Gamma]\!] \vdash_? [\![\lambda x : T.U]\!]_\Gamma : [\![\Pi x : T.U]\!]_\Gamma$.
- App : On a :

$$\frac{\Gamma \vdash_\bullet f : T \quad \mu_\bullet(T) = \Pi x : V.W \qquad \Gamma \vdash_\bullet u : U \qquad \Gamma \vdash_\bullet U \rhd V s}{\Gamma \vdash_\bullet (f u) : W[u/x]}$$

C'est le cas intéressant puisqu'on ajoute ici des coercions. Par induction, $[\![\Gamma]\!] \vdash_? [\![f]\!]_\Gamma : [\![T]\!]_\Gamma$, $[\![\Gamma]\!] \vdash_? [\![u]\!]_\Gamma : [\![U]\!]_\Gamma$ et $[\![\Gamma]\!] \vdash_? [\![U]\!]_\Gamma, [\![V]\!]_\Gamma : [\![s']\!]_\Gamma = s'$.
On procéde par étapes : d'abord la construction d'une fonction $[\![\Gamma]\!] \vdash_? \pi[[\![f]\!]_\Gamma] : [\![\Pi x : V.W]\!]_\Gamma$ puis la construction de l'argument $[\![\Gamma]\!] \vdash_? c[[\![u]\!]_\Gamma] : [\![V]\!]_\Gamma$. On a ces deux résultats par application de l'hypothèse de récurrence pour les coercions. On n'a plus qu'à les appliquer pour obtenir le jugement : $[\![\Gamma]\!] \vdash_? [\![f\ u]\!]_\Gamma : [\![W]\!]_{\Gamma,x:V}[c[[\![u]\!]_\Gamma]/x]$. Par le lemme 3.2.17, on a $[\![W]\!]_{\Gamma,x:V}[c[[\![u]\!]_\Gamma]/x] \equiv [\![W[u/x]]\!]_\Gamma$ puisque les coercions de sorte à sorte ne peuvent être que l'identité. On peut donc déduire : $[\![\Gamma]\!] \vdash_? [\![f\ u]\!]_\Gamma : [\![W[u/x]]\!]_\Gamma$ par Conv.

– Sum, Sum, Let-Sum, Subset : Direct par induction ou un raisonnement similaire à App.

– $\rhd$-Conv : Par le lemme 3.2.20 on a $[\![T]\!]_\Gamma \equiv [\![U]\!]_\Gamma$, on peut donc dériver :

$$\frac{[\![\Gamma]\!] \vdash_? t : [\![T]\!]_\Gamma \qquad [\![T]\!]_\Gamma \equiv [\![U]\!]_\Gamma}{[\![\Gamma]\!] \vdash_? c[t] \equiv t : [\![U]\!]_\Gamma}$$

– $\rhd$-$\downarrow$: Par induction on a $[\![\Gamma]\!] \vdash_? t : [\![T^\downarrow]\!]_\Gamma \Rightarrow [\![\Gamma]\!] \vdash_? c[t] : [\![U^\downarrow]\!]_\Gamma$. A l'aide du lemme 3.2.20 on peut dériver :

$$\frac{\dfrac{\dfrac{[\![\Gamma]\!] \vdash_? t : [\![T]\!]_\Gamma \qquad [\![T]\!]_\Gamma \equiv [\![T^\downarrow]\!]_\Gamma}{[\![\Gamma]\!] \vdash_? t : [\![T^\downarrow]\!]_\Gamma}}{[\![\Gamma]\!] \vdash_? c[t] : [\![U^\downarrow]\!]_\Gamma} \qquad [\![U^\downarrow]\!]_\Gamma \equiv [\![U]\!]_\Gamma}{[\![\Gamma]\!] \vdash_? c[t] : [\![U]\!]_\Gamma}$$

– $\rhd$-Prod : On a :

$$\frac{\Gamma \vdash_? c_1 : U \rhd T : s_1 \qquad \Gamma, x : U \vdash_? c_2 : V \rhd W : s_2}{\Gamma \vdash_? \lambda x : [\![U]\!]_\Gamma.c_2[\bullet\ (c_1[x])] : \Pi x : T.V \rhd \Pi x : U.W : s_3}\ (s_1, s_2, s_3) \in \mathcal{R}$$

Supposons $[\![\Gamma]\!] \vdash_? t : [\![\Pi x : T.V]\!]_\Gamma$. Alors $c[t] = \lambda x : [\![U]\!]_\Gamma.c_2[t\ c_1[x]]$.
Par induction, $[\![G, x : U]\!] \vdash_? c_1[x] : [\![T]\!]_\Gamma$, donc par App, $[\![G, x : U]\!] \vdash_? t\ c_1[x] : [\![V]\!]_{\Gamma,x:T}[c_1[x]/x]$. Par stabilité par affaiblissement (3.2.15), $[\![V]\!]_{\Gamma,x:T}[c_1[x]/x] \equiv [\![V]\!]_{\Gamma,x:U}$.
On peut donc dériver par $\rhd$-Conv :

$$[\![G, x : U]\!] \vdash_? t\ c_1[x] : [\![V]\!]_{\Gamma,x:U}$$

Par induction, $[\![G, x : U]\!] \vdash_? c_2[t\ c_1[x]] : [\![W]\!]_\Gamma$, donc par Abs,

$$[\![\Gamma]\!] \vdash_? \lambda x : [\![U]\!]_\Gamma.c_2[t\ c_1[x]] : [\![\Pi x : U.W]\!]_\Gamma$$

– $\rhd$-Sum : On a :

$$\frac{\Gamma \vdash_? c_1 : T \rhd U : s \qquad \Gamma, x : T \vdash_? c_2 : V \rhd W : s}{\Gamma \vdash_? (c_1[\pi_1\ \bullet], c_2[\pi_2\ \bullet][\pi_1\ \bullet/x])_{[\![\Sigma x:U.W]\!]_\Gamma} : \Sigma x : T.V \rhd \Sigma x : U.W : s}\ s \in \{\texttt{Prop}, \texttt{Set}\}$$

Ici, $c[t] = (c_1[\pi_1\ t], c_2[\pi_1\ t/x][\pi_2\ t])_{[\![\Sigma x:U.W]\!]_\Gamma}$. Par application des règles de typage pour les projections : $[\![\Gamma]\!] \vdash_? \pi_1\ t : [\![T]\!]_\Gamma$ et $[\![\Gamma]\!] \vdash_? \pi_2\ t : [\![V]\!]_{\Gamma,x:T}[\pi_1\ t/x]$.
Par induction on obtient $[\![\Gamma]\!] \vdash_? c_1[\pi_1\ t] : [\![U]\!]_\Gamma$. Par induction on a aussi $[\![\Gamma, x : T]\!] \vdash_? \lambda y : [\![V]\!]_\Gamma.c_2[\pi_2\ y] : [\![V]\!]_\Gamma \to [\![W]\!]_\Gamma$, où y n'apparait pas dans c_2.
Par affaiblissement, on passe dans l'environnement $\Gamma, t : \Sigma x : T.V, x : T$:

$$[\![\Gamma, t : \Sigma x : T.V, x : T]\!] \vdash_? \lambda y : [\![V]\!]_\Gamma.c_2[\pi_2\ y] : [\![V]\!]_\Gamma \to [\![W]\!]_\Gamma$$

On peut alors substituer pour obtenir : $[\![\Gamma, t : \Sigma x : T.V]\!] \vdash_? \lambda y : [\![V]\!]_\Gamma[\pi_1\ t/x].c_2[\pi_1\ t/x][\pi_2\ y] : [\![V]\!]_\Gamma[\pi_1\ t/x] \to [\![W]\!]_\Gamma[\pi_1\ t/x]$.
Enfin on applique à $\pi_2\ t$ pour obtenir : $[\![\Gamma, t : \Sigma x : T.V]\!] \vdash_? c_2[\pi_1\ t/x][\pi_2\ t] : [\![W]\!]_\Gamma[\pi_1\ t/x]$.
On a bien :

$$[\![\Gamma]\!] \vdash_? \lambda t : [\![\Sigma x : T.V]\!]_\Gamma.\ (c_1[\pi_1\ t], c_2[\pi_1\ t/x][\pi_2\ t])_{[\![\Sigma x:U.W]\!]_\Gamma} : [\![\Sigma x : T.V]\!]_\Gamma \to [\![\Sigma x : U.W]\!]_\Gamma$$

– $\rhd$-Proof : On a :

$$\frac{\Gamma \vdash_? c : T \rhd U : \texttt{Set} \qquad \Gamma \vdash_\bullet \{\ x : U \mid P\ \} : \texttt{Set}}{\Gamma \vdash_? (c, ?_{[\![P]\!]_{\Gamma,x:U}[c/x]})_{[\![\{x:U|P\}]\!]_\Gamma} : T \rhd \{\ x : U \mid P\ \} : \texttt{Set}}\ T = T^\downarrow$$

Par induction, $[\![\Gamma]\!] \vdash_? \lambda x : [\![T]\!]_\Gamma.c[x] : [\![T]\!]_\Gamma \to [\![U]\!]_\Gamma$ et $[\![\Gamma]\!] \vdash_? \{\ x : [\![U]\!]_\Gamma \mid [\![P]\!]_{\Gamma,x:U}\ \} :$ Set.
On a donc clairement :

$$[\![\Gamma]\!] \vdash_? \lambda t : [\![T]\!]_\Gamma.(c[t], ?_{[\![P]\!]_{\Gamma,x:U}[c[t]/x]})_{[\![\{x:U|P\}]\!]_\Gamma} : [\![T]\!]_\Gamma \to [\![\{\ x : U \mid P\ \}]\!]_\Gamma$$

– $\rhd$-Subset : On a :

$$\frac{\Gamma \vdash_? c : U \rhd T : \texttt{Set} \qquad \Gamma \vdash_\bullet \{\ x : U \mid P\ \} : \texttt{Set}}{\Gamma \vdash_? c[\pi_1\ \bullet] : \{\ x : U \mid P\ \} \rhd T : \texttt{Set}}\ T = T^\downarrow$$

Par induction, $[\![\Gamma]\!] \vdash_? \lambda x : [\![U]\!]_\Gamma.c[x] : [\![U]\!]_\Gamma \to [\![T]\!]_\Gamma$, on a bien :

$$[\![\Gamma]\!] \vdash_? \lambda t : [\![\{\ x : U \mid P\ \}]\!]_\Gamma.c[\pi_1\ t] : [\![\{\ x : U \mid P\ \}]\!]_\Gamma \to [\![T]\!]_\Gamma$$

□

Conclusion

On a donc finalement notre preuve de correction de la traduction à partir du système algorithmique. En combinant les résultats précédents, on a donc que toute dérivation du système déclaratif a une traduction bien typée dans $\mathsf{CC}_?$:

$$\Gamma \vdash t : T \Rightarrow [\![\Gamma]\!] \vdash_? [\![t]\!]_\Gamma : [\![T]\!]_\Gamma$$

Il ne reste plus maintenant qu'à "remplir les trous" en résolvant les obligations pour obtenir un terme Coq complet et bien typé correspondant à notre programme de départ.

Notons cependant que comme Coq n'implémente ni les règles η ni l'indifférence aux preuves, la traduction peut donner des termes non typables.

Chapitre 4

Implémentation

> `Hacker` : One who programs enthusiastically (even obsessively) or who enjoys programming rather than just theorizing about programming.
>
> *Jargon File*

Sommaire

Nous avons mis en oeuvre l'algorithme de typage de Russell et sa traduction vers CCI en Coq. Nous allons maintenant présenter cette implémentation et un certain nombre d'extensions à ce noyau permettant de programmer effectivement en Coq. On démontrera dans une étude de cas au chapitre 11 l'utilité de ces extensions.

4.1 Une extension à Coq

Russell est incarné en Coq sous forme d'un algorithme de typage modifié et d'une suite de commandes pour gérer les obligations. On peut y accéder simplement en préfixant les commandes du mot-clé `Program`. Par exemple :

```
Require Import Program.

Program Definition succ_nz (n : nat) : { n' : nat | n' ≠ 0 } := S n.
```

Pour utiliser Program, il faut toujours importer la librairie Coq.Program.Program qui définit un certain nombre de notations et de tactiques qu'on va présenter.

Lorsqu'on définit succ_nz, le programme est typé dans Russell et traduit vers les termes de Coq avec des variables existentielles. Celles-ci sont alors transformées en autant d'obligations (par la tactique eterm) qu'il faut résoudre pour obtenir une définition de succ_nz complète.

Ici, la traduction produit une variable existentielle correspondant à la coercion

$$n : \mathsf{nat} \vdash (\mathsf{S}\ n, ?_{\mathsf{S}n \neq 0}) : \{\ n' : \mathsf{nat} \mid n' \neq 0\ \}$$

Cette obligation est résolue automatiquement par une tactique par défaut `obligation_tactic` définie dans la librairie Coq.Program.Tactics qui nettoie le but et tente de résoudre le but avec la tactique `auto`. Si celle-ci échoue, on peut lancer une preuve interactive de l'obligation avec la commande `Next Obligation`. Une fois toutes les obligations résolues on obtient automatiquement la définition de la constante succ_nz dans notre environnement.

On introduit la notation ! $\triangleq$ False_rect _ _ qui permet de déclarer un point du programme comme inaccessible. Si l'on écrit ! quelque part dans un programme Russell, cela génère une obligation de prouver $\bot$ dans le contexte correspondant. C'est l'équivalent de l'idiome `assert false` de OCaml, mais ici on doit produire une preuve de l'impossibilité d'arriver à ce point de programme.

On utilise aussi la notation ‘x pour dénoter la première projection d'un objet de type sous-ensemble. Cette projection apparaissant très fréquemment lorsqu'on utilise Program, on lui a octroyé un symbole qui influe le moins possible sur la lisibilité.

4.1.1 Typage et traduction

En réalité, le typage et la traduction ont lieu en même temps dans l'implémentation. On peut tout-à-fait implémenter Russell comme une simple extension de l'algorithme de coercion (Coercions, (Saïbi 1997)) existant en Coq. Par le lemme de substitutivité et théorème de correction de la traduction, on peut en effet faire la coercion entre types Coq directement plutôt que de traduire les dérivations de Russell, potentiellement longues. On a donc fonctorisé l'algorithme de typage de Coq pour qu'il prenne en entrée une implémentation de la coercion qu'on instancie soit avec l'algorithme original soit avec son extension par la coercion de Russell.

La coercion elle-même est implémentée par une fonction récursive sur les deux types qui retourne optionellement une fonction représentant un contexte. Il suffit de l'appliquer à un objet du type de gauche pour obtenir un objet du type de droite. On s'appuie sur l'algorithme de coercion original au lieu de la seule conversion lorsque aucune autre règle ne s'applique.

4.1.2 `eterm`

À partir de notre terme à trous on veut maintenant générer des obligations. Or le système de preuve de Coq est basé sur une notion de sous-but ne coincidant pas avec les variables existentielles. Il est donc impossible de l'utiliser pour faire du vrai raffinement en donnant simplement le terme à trous et son type et en le laissant s'occuper de gérer les sous-buts correspondants. Le problème essentiel est qu'on ne peut pas se référer à l'habitant potentiel d'un sous-but Coq dans un autre sous-but. Par exemple, l'architecture ne supporte pas d'avoir un sous-but pour un type et un autre sous-but pour un objet de ce type. Cependant, dès qu'on a des dépendances comme dans les obligations de Russell, de telles situations se présentent et le système ne fonctionne pas. Un effort ambititeux de réécriture du noyau du système de preuve a été lancé récemment par Arnaud Spiwack pour retirer notamment cette limitation en passant à une vue des sous-buts comme existentielles, mais elle est encore à l'état de prototype. À l'époque du développement de Russell, nous n'avions pas le choix d'y participer ni l'ambition de réécrire complètement la machine de preuve, on a donc développé un outil séparé permettant de gérer les obligations dépendantes.

Essentiellement, on va skolémiser les existentielles pour en faire des définitions Coq séparées et gérer les dépendances de façon externe, simplement en maintenant un graphe de dépendances. On a la propriété que l'ensemble des existentielles qui nous est donné n'a pas de cycle, car cela aurait produit une erreur d'"*occurs-check*" à l'unification. On construit donc une liste des existentielles ordonnée de façon à ce que chaque existentielle ne fasse référence qu'à ses prédécesseurs et on leur assigne un nom de variable unique. On construit ensuite pour chacune d'entre elles leur type skolémisé : pour une variable $?_{\Gamma\vdash\tau}$ on crée le type $\forall\ \Gamma[\sigma], \tau[\sigma]$ où $\mathrm{t}[\sigma]$ représente la substitution des existentielles par leurs noms de variable (les existentielles sont toujours appliquées à une instance de leur contexte en Coq). On fait la même substitution dans le terme mentionnant ces variables. On peut alors retourner la liste des variables avec leurs types substitués et leurs dépendances ainsi que le terme à trous désormais sans trous mais ouvert.

4.1.3 Obligations

À partir de ces informations, on peut créer un gestionnaire d'obligations qui permet de lancer la preuve des obligations n'ayant pas de dépendances non définies. Dès qu'une obligation est définie, on conserve son vrai nom Coq qui est substitué pour la variable correspondante dans le reste des obligations. Une fois toutes les obligations définies, on peut déclarer un terme Coq complet. Pour gérer les obligations, on a un certain nombre de commandes décrites dans le manuel de référence (Sozeau 2008a). On utilise principalement les suivantes :

- `Next Obligation` Démarre la preuve interactive de la prochaine obligation sans dépendances du programme courant.
- `Obligation Tactic := tac` Déclare la tactique par défaut pour décharger les obligations. Elle est appliquée automatiquement après chaque définition sur toutes les obligations générées et à chaque début de preuve interactive d'obligation (e.g. par `Next Obligation`).
- `Solve Obligations using tac` Résoud les obligations du programme courant en utilisant la tactique donnée (sans utiliser celle par défaut).

La tactique par défaut est une tactique de "nettoyage" qui tente de faire les injections d'égalités entre constructeurs, de décomposer les objets de type sous-ensemble et de substituer les égalités entre variables.

Cette machinerie est suffisante pour traiter les programmes générés par Program, mais pour rendre le système vraiment utilisable il faut étendre le mécanisme pour supporter deux constructions essentielles : les types (co-)inductifs et la récursion structurelle et bien-fondée.

4.2 (Co-)Inductifs

L'extension de Russell par des types inductifs n'a pas été formalisée, mais elle est relativement évidente. Les constructeurs sont simplement traités comme des variables déclarées dans l'environnement. L'élimination d'un objet de type inductif en Coq est réalisée par filtrage. Voici par exemple la fonction prédecesseur :

```
Definition pred (x : nat) : option nat :=
  match x return option nat with
    | O ⇒ None
    | S x' ⇒ Some x'
  end.
```

Lorsqu'on filtre sur la variable x de type nat, on doit donner un ensemble de branches couvrant toutes les formes possibles d'un objet de ce type. La clause `return` donne le type de retour du filtrage, qui pourrait être différent dans chaque branche suivant la valeur de x. Ici on a donné une branche par constructeur du type nat, l'algorithme de compilation du filtrage peut donc vérifier que notre filtrage est *exhaustif*. On a donc bien une fonction totale qui à chaque naturel associe optionellement un autre naturel.

Maintenant, supposons que l'on veuille écrire une fonction totale pred' des naturels différents de O vers les naturels. Grâce à Program, on devrait pouvoir écrire :

```
Program Definition pred' (x : nat | x ≠ 0) : nat :=
  match x return nat with
    | O ⇒ !
    | S x' ⇒ x'
  end.
```

Malheureusement, cette définition génère une obligation $\{ x : \mathsf{nat} \mid x \neq \mathsf{O} \} \to \bot$ qui n'est pas prouvable. On a simplement perdu l'information du contexte du !, où l'on sait que $x = 0$ par filtrage. Il faut donc parvenir à refléter le filtrage dans le contexte logique de chaque branche en introduisant des égalités. Cette manoeuvre bien connue dans le folklore des assistants de preuve basés sur la théorie des types est cependant fastidieuse et obscurcit le terme de preuve. On va l'intégrer à la compilation du filtrage par défaut de Program. Il devient ainsi possible d'écrire :

```
Program Definition pred'' (x : nat | x ≠ 0) : nat :=
  match x with
    | O ⇒ !
    | S x' ⇒ x'
  end.
```

L'obligation générée pour ! est maintenant $\{ x : \mathsf{nat} \mid x \neq \mathsf{O} \} \to \mathsf{proj1\_sig}\ x = \mathsf{O} \to \bot$. Celle-ci est parfaitement prouvable puisqu'on une contradiction : la première projection de x est à la fois égale à et différente de O.

En interne, on a transformé la construction de filtrage pour maintenir une égalité entre l'objet filtré et ses différents filtres. La définition Coq équivalente est :

```
Definition pred''' (x : nat | x ≠ O) : nat :=
  match proj1_sig x as y return proj1_sig x = y → nat with
    | O ⇒ λ H : proj1_sig x = O, False_rect _ (proj2_sig x H)
    | S x' ⇒ λ _ : proj1_sig x = S x', x'
  end (refl_equal (proj1_sig x)).
```

On utilise ici une élimination *dépendante* de l'objet proj1_sig x. La clause `as` donne un nom au motif correspondant à l'objet filtré dans chaque branche, que l'on peut utiliser pour faire varier le type de retour de celles-ci. Dans ce type de retour (aussi appelé prédicat d'élimination), on indique que chaque branche est une abstraction dont le domaine est une preuve d'égalité entre l'objet filtré et le motif de ladite branche. On type donc chaque branche en substituant le motif dans le type de retour attendu. Ici on a annoté les abstractions de chaque branche avec le type approprié, mais on voit bien qu'il aurait pu être inféré à partir de l'information existante. Une fois qu'on a vérifié l'exhaustivité du filtrage et typé chaque branche, on obtient un terme du type de retour où l'on a substitué les variables de motif comme y par l'objet filtré correspondant (ici proj1_sig x). On a donc finalement un objet de type $\mathsf{proj1\_sig}\ x = \mathsf{proj1\_sig}\ x \to \mathsf{nat}$, on l'applique à la preuve de réflexivité refl_equal (proj1_sig x) pour obtenir le terme de type nat attendu.

Familles inductives

Cette transformation fonctionne même dans les cas de filtrage avec types dépendants. Par exemple, sur le type vecteur introduit précédemment, on peut écrire :

```
Program Definition vhead A n (v : vector A (S n)) : A :=
  match v with
    | Vnil ⇒ !
    | Vcons x _ _ ⇒ x
  end.
```

Ici le contexte au point du ! contient non seulement une égalité (cette fois *hétérogène*) sur v, mais aussi une égalité sur l'index de v : $\mathsf{O} = \mathsf{S}\ n$, d'où l'on peut obtenir sans peine une contradiction par le principe de *discrimination* des constructeurs. On étudiera formellement la construction de filtrage avec types dépendants dans la prochaine section.

Considérons maintenant la fonction suivante :

```
Program Definition vtail A n (v : vector A (S n)) : vector A n :=
  match v with
    | Vnil ⇒ !
    | Vcons x n' v' ⇒ v'
  end.
```

Pour typer cette fonction nous avons besoin d'encore un peu plus d'expressivité. La branche Vnil est traitée comme dans vhead, mais pour la branche Vcons c'est un peu plus compliqué. En effet, on a v : vector A (S n), $x : A$, n' : nat et v' : vector A n' avec S n = S n' et $v \tilde{=}$ Vcons x n' v'. On retourne v' tandis qu'on attend un objet de type vector A n, il y a donc un problème de typage. Program accepte cependant cette définition, parce qu'il permet de faire des coercions entre instances d'une même famille inductive. Ici en particulier on autorise la coercion vector A $n' \rhd$ vector A n.

$$\frac{I \textbf{ famille inductive} \qquad \Gamma \vdash I\ \overrightarrow{x_i}, I\ \overrightarrow{y_i} : \mathtt{Type} \qquad \Gamma, \overrightarrow{(x_i : X_i \simeq y_i : Y_i)}_0^{j-1} \vdash p_j : (x_j : X_j \simeq y_j : Y_j) \quad (j \in [0..i])}{\Gamma \vdash \mathtt{subst}\overrightarrow{p_i} : I\ \overrightarrow{x_i} \rhd I\ \overrightarrow{y_i} : \mathtt{Type}}$$

FIG. 4.1: Coercion de familles inductives

On ajoute simplement une règle au système de coercion de Russell qui permet de coercer deux instantiations d'une famille inductive (figure 4.1). On formalise simplement le fait que les habitants d'une famille I ayant pour indices $\overrightarrow{x_i}$ peuvent être vus comme des habitants de $I\ \overrightarrow{y_i}$ si leurs indices sont prouvablement égaux. L'égalité devrait être hétérogène si nécessaire, mais nous ne traitons pour l'instant que l'égalité de Leibniz. En pratique nous n'en n'avons pas encore eu besoin. La coercion est construite à l'aide du principe de substitution de l'égalité qui transforme $I\ x$ en $I\ y$ étant donnée une preuve de $x = y$.

À notre connaissance, cette règle est originale mais elle est implicite dans bon nombre de travaux sur les types indexés et GADTs (Chen et Xi 2005; Pottier et Régis-Gianas 2006; Jones, Vytiniotis, Weirich, et Washburn 2006), et dans certains cas l'opérateur de substitution `subst` est utilisé explicitement pour réaliser la coercion McKinna (2006). Elle est essentielle pour travailler avec les familles inductives dans Program comme nous le verrons dans l'exemple des Finger Trees chapitre 11. On peut facilement vérifier que cette règle est symétrique puisque l'égalité est symétrique. Cependant, pour montrer que l'équivalence est préservée, il faut aussi la règle η montrant que $c \circ c^{-1} \equiv \bullet$, et donc

que `subst` $A\ x\ y\ P\ p$ (`subst` $A\ y\ x\ P\ q\ t$) $\equiv \bullet$. Or la règle de réduction de `subst` est : `subst` $A\ x\ x\ P$ (refl_equal $A\ x$) $t \rightarrow_{\delta\iota} t$, il faut donc que les preuves d'égalité entre indices ne "mentent" pas, c'est-à-dire qu'elles se réduisent bien sur le constructeur canonique de réflexivité de l'égalité pour qu'on préserve l'équivalence. En pratique, on ne vérifiera pas cette propriété de symmétrie mais ça n'est généralement pas gènant puisque la préservation de l'équivalence n'est pas non plus utilisée très souvent : elle n'apparaît que lorsqu'on substitue un terme déjà coercé dans un terme contenant une coercion pour la variable substituée et qu'on a besoin de simplifier ces deux coercions. Notons que Coq ne gère pas non plus la "*Proof Irrelevance*" (PI), donc cette propriété n'est pas non plus vérifiée pour les coercions de sous-ensembles. Néanmoins, il est intéressant de voir que cette nouvelle notion de coercion est plus forte que celle sur les sous-ensembles et s'appuie vraiment sur le contenu de la preuve. On peut y voir un problème similaire au marquage des égalités sûres dans le système développé par Blanqui, Jouannaud, et Strub (2007).

4.2.1 Analyse de cas avec types dépendants

Le filtrage de motifs tel qu'implémenté en Coq n'est pas suffisant pour typer les programmes dépendants sans faire des manipulations fastidieuses sur le contexte du filtrage, les termes impliqués et en faisant le raisonnement équationnel associé explicitement. On a vu que le filtrage supporté par Program, associé aux coercions, permet de typer plus de programmes sans annotations. Mais ce filtrage est en fait limité à l'analyse de cas à un niveau sur les objets à type dépendant et ne fonctionne pas dans le cas général avec des filtres imbriqués. En effet, l'idée initiale de cette méthode de compilation est que les filtres bien formés sont un sous-ensemble des termes bien typés et qu'on peut donc ajouter dans chaque branche une égalité (hétérogène en général) entre terme et filtre. Or cette hypothèse est fausse si les filtres sont imbriqués, puisque dans certains cas seuls des filtres non-linéaires seraient bien typés :

```
Definition nested A n (v : vector A n) : nat :=
  match v with
    | Vcons a n (Vcons a' n' v) ⇒ S 0
    | _ ⇒ 0
  end.
```

Ici le filtre Vcons $a\ n$ (Vcons $a'\ n'\ v$) n'a pas de terme bien typé équivalent : si l'on généralise les variables libres on obtient un terme mal typé puisque le vecteur imbriqué, de longueur S n', n'a pas un type équivalent à vector $A\ n$, le type forcé par le motif englobant. Il est donc impossible de faire la compilation du filtrage dépendant de cette façon. On peut cependant noter que les seules valeurs canoniques qui peuvent être filtrées par ce motif sont de la forme Vcons a (S n') (Vcons $a'\ n'\ v$). Le typage force cette non-linéarité, et dans ce cas, il est clair qu'on peut écrire une équation entre le terme filtré et ce motif. L'idée qu'on peut définir le filtrage à partir des seuls filtres bien typés est à la base du filtrage dépendant introduit par Coquand (1992) et raffinée par Goguen, McBride, et McKinna (2006).

Le filtrage de motifs implémenté dans Program est incomplet comme on l'a vu puisqu'il suppose que les motifs donnés par l'utilisateur correspondent toujours à des termes bien typés. Dans les cas non dépendants, cela ne pose aucun problème, même en présence de paramètres parce que ceux-ci sont traités non-linéairement de façon implicite :

```
Definition matchlist A (l : list A) : list A :=
  match l with
    | cons a (cons a' l) ⇒ cons a l
```

```
  | _ ⇒ nil
end.
```

Dans le filtre cons a (cons a' l), on a implicitement instancié les paramètres de type des constructeurs de l'inductif list avec le paramètre de l, soit A, et l'on ne peut pas donner une instance différente de ce paramètre pour le sous-motif cons a' l : il est forcé à A. On peut donc typer le terme correspondant : @cons A a' (@cons A a l). On voit bien ici que si les paramètres étaient considérés comme des arguments à part entière des constructeurs, on aurait le même problème de linéarité que pour le traitement des indices (ou vrais arguments), puisque des variables apparaîtraient de façon non-linéaire dans les filtres.

Si l'on ne considère pas les constructeurs de types dépendants imbriqués, la traduction de Program est toujours correcte. On va maintenant la formaliser.

Définition 4.2.1 (Problème de filtrage)**.** *On travaille sur des problèmes de filtrage*

$$(\Gamma, t, I\ \overrightarrow{t_j}, (\Pi\overrightarrow{(\ell_j : \tau_j)}(i : I\ \overrightarrow{\ell_j}), \psi), \overrightarrow{(\Gamma_i, p_i \Rightarrow b_i)})$$

où :

- Γ *donne le contexte courant,*
- t *est l'objet filtré,*
- I *est la famille inductive concernée avec une instance des indices* $\overrightarrow{t_j}$*,*
- $\Pi\overrightarrow{(\ell_j : \tau_j)}(i : I\ \overrightarrow{\ell_j}), \psi$ *est le type de retour du filtrage dépendant d'une instance abstraite de la famille inductive dans le contexte* Γ *(*$\overrightarrow{t_j}$ *est une instance de* $\overrightarrow{\ell_j : \tau_j}$*),*
- $\overrightarrow{(\Gamma_i, p_i \Rightarrow b_i)}$ *est l'ensemble des clauses de filtrage où l'on suppose qu'on peut voir le motif* p_i *comme un terme bien typé dans* Γ, Γ_i*. Le terme* b_i *n'est pas nécessairement encore typé.*

L'algorithme procède tout d'abord à la construction d'un prédicat d'élimination généralisée. On prend le téléscope $\overrightarrow{\ell_j : \tau_j}$ des indices de la famille I et on construit le type :

$$\Pi\ \overrightarrow{(\ell_j : \tau_j)}\ (i : I\ \overrightarrow{\ell_j}), \overrightarrow{\ell_j \simeq t_j} \to i \simeq t \to \psi$$

Ce type sera le prédicat d'élimination du nouveau filtrage. Pour chaque branche, on va typer le corps dans un environnement enrichi des égalités introduites dans le prédicat d'élimination. Dans la branche i, on type tout d'abord le filtre p_i en un terme p'_i, on récupère son instance des indices $\overrightarrow{t'_j}$ puis on construit le contexte $\Gamma'_i \triangleq \Gamma, \Gamma_i, \Delta_i$ où

$$\Delta_i \triangleq \overrightarrow{px_j : t'_j \simeq t_j}, pi : p'_i \simeq t$$

On type finalement la branche b_i dans Γ'_i pour obtenir un terme Coq b'_i (si le typage réussit). On construit alors le terme $\lambda\Delta_i, b'_i$ qui sera le terme final pour la branche i, où on a introduit autant d'abstractions qu'il y a d'égalités dans le prédicat d'élimination.

On a alors une construction de filtrage de type $\overrightarrow{t_j \simeq t_j} \to t \simeq t \to \psi[t/i]\overrightarrow{[t_j/\ell_j]}$ qu'il suffit d'appliquer aux preuves de réflexivité de $\simeq$ pour obtenir le terme final de type $\psi[t/i]\overrightarrow{[t_j/\ell_j]}$.

Le schéma général est donc l'abstraction de l'instance par des égalités qui sont instanciées dans chaque branche pour conserver l'information sur les indices. La seule source de partialité de ce schéma de compilation est le fait de tester si les filtres sont bien typés, sinon elle c'est une procédure totale.

Il y a deux solutions pour éviter ce problème de typage des filtres : se restreindre aux motifs non-imbriqués ou avoir comme prérequis que les filtres soient bien typés et donc

potentiellement non-linéaires. La deuxième solution est cependant plus complexe puisqu'il faut convaincre Coq que certains motifs sont impossibles par du raisonnement équationnel et que cela nécessite parfois l'utilisation de l'axiome K, on discutera cette solution dans les perspectives (§ IV).

4.3 Récursion

Jusqu'ici, nous n'avons pas introduit de récurseur dans notre formalisme, il est donc à peu près impossible d'écrire quelque programme intéressant que ce soit dans Russell ! Or en Coq il est possible de faire des définitions par récurrence structurelle sur un objet de type inductif (et par corécursion sur un coinductif, mais nous n'entrerons pas dans ces détails, voir (Giménez 1996) pour une présentation détaillée du traitement des coinductifs). Cette construction correspond à un nouveau constructeur de termes $\texttt{fix}_i(f)(\Pi\overrightarrow{\alpha_i}, \beta)$ similaire au combinateur Y du lambda-calcul : si f est une fonctionnelle de type $(\Pi\overrightarrow{\alpha_i}, \beta) \rightarrow (\Pi\overrightarrow{\alpha_i}, \beta)$ alors $\texttt{fix}_i(f)(\Pi\overrightarrow{\alpha_i}, \beta)$ est de type $\Pi\overrightarrow{\alpha_i}, \beta$. Coq vérifie que f est bien formée au sens où α_i doit être un objet de type inductif et toute les appels récursifs dans f doivent être faits sur des applications (n-aires) de sous-termes de l'argument i de f. La notion de sous-terme est définie vis-à-vis du filtrage. Ce principe de bonne formation est appelé la condition de garde et est expliqué en détail dans le manuel de référence.

Dans le cadre de Program, il est trivial de supporter cette construction (et la construction duale pour les coinductifs) : c'est totalement transparent pour le système de coercion et de génération d'obligations. Un seul point est problématique, c'est l'opacité des obligations de preuves. Au moment où l'on définit une fonction récursive en Coq, on ne peut pas appliquer partiellement cette fonction à une définition opaque de Coq, car la condition de garde ne déplie pas les définitions opaques et ne peut donc pas vérifier que la fonction est appelée correctement dans celles-ci. Or il est facile d'obtenir ce cas de figure avec Program. Supposons qu'on écrive :

```
Program Fixpoint id (x : nat) : { y : nat | y = x } :=
  match x with
    | O ⇒ O
    | S x' ⇒ S (id x')
  end.
```

L'obligation générée pour la deuxième branche de cette définition est

```
Π (id : Π x : nat, {y : nat | y = x}) (x x' : nat), S x' = x → S (‘(id x')) = S x'
```

On voit ici le prototype récursif id, utilisé à l'appel récursif id x'.

```
Next Obligation. destruct (id x') as [y Hy]. simpl. subst x'. reflexivity. Defined.
```

La preuve procède par élimination de l'appel récursif, qui introduit une variable y et une preuve de $y = x'$. Par simplification, substitution et réflexivité on montre alors que S (‘(y, Hy)) = S x'. Si l'on termine la preuve par Qed, Coq renvoie une erreur puisqu'il ne peut pas vérifier qu'on n'appelle pas id sur d'autres variables. On doit donc rendre les preuves transparentes dans ce cas en utilisant Defined. Il est même recommandé d'effacer le prototype du contexte lorsqu'on n'en a plus besoin afin d'éviter qu'il soit utilisé par inadvertance par les tactiques automatiques. Program enlève lui-même le prototype s'il peut voir qu'il n'est pas utilisé (comme dans la première branche par exemple qui ne fait pas d'appel récursif).

Un autre problème est dû à la limitation de la condition de garde syntaxique : on ne peut par faire de définitions par récurrence structurelle sur un objet de type sous-ensemble.

```
Program Definition predrec (x : nat | x ≠ 0) : nat :=
  match x with
    | O ⇒ !
    | S 0 ⇒ 0
    | S x' ⇒ S x'
  end.
```

En effet, il y a implicitement un constructeur du type sous-ensemble inséré à l'appel récursif de predrec, la condition de guarde échoue donc immédiatement, sans considérer que le témoin de la preuve x' lui décroit. On peut résoudre ce problème soit en mettant la preuve de côté en curryfiant, soit en utilisant une mesure (§4.3.1) plutôt que la récursion structurelle. Une extension mimimale de la condition de garde ou un passage dans un système où la terminaison est garantie par typage permettrait de résoudre ce problème.

Cet exemple utilise un raffinement du filtrage de Program : la sémantique "*first-match*" du filtrage de motif implique que dans la troisième branche, x' est différent de O, qui a été filtré juste au-dessus. On peut donc introduire une preuve de $x' \neq$ O dans cette branche, et c'est ce que fait Program avant de typer cette branche. Il faut bien voir que le code de cette troisième branche pourrait être dupliqué par l'algorithme de compilation du filtrage de Coq. Celui-ci compile les filtres qui se chevauchent en faisant autant d'expansions que nécessaire pour obtenir des filtres sans chevauchement. Ici la troisième branche s'expanse en une seule branche avec le motif S (S _ as x'), mais si l'on avait des constructeurs supplémentaires on aurait autant de branches supplémentaires. Il est important de typer chaque branche une seule fois avant expansion si la branche génère des obligations, puisque après expansion, on génèrerait la même obligation pour chacune d'entres elles. Ce raffinement respecte donc mieux la structure du programme original, qui capture un ensemble de cas avec un unique filtre, sans perdre d'information grâce aux inégalités ajoutées à l'environnement.

4.3.1 Récursion bien fondée

La récursion bien fondée est en fait codée en Coq par une récursion structurelle sur une preuve de bonne fondation. Ces preuves sont des objets du type Acc suivant :

```
Inductive Acc (A : Type) (R : A → A → Prop) (x : A) : Prop :=
    Acc_intro : (Π y : A, R y x → Acc R y) → Acc R x.

Definition well_founded A R := Π x : A, Acc R x.
```

Un élément x de type A est accessible pour une relation R sur A si, pour tout élément y précédant x par R, y est lui-même accessible. On a implicitement une preuve qu'il n'y a pas de chaîne infinie descendante de R si Π x, Acc R x. En effet, si on a Acc pour tout x, alors il doit exister au moins un y qui n'a pas d'antécédent par R, sinon la preuve serait circulaire. Bien sûr, cette version constructive de l'accessibilité donne un moyen de calculer *via* son principe d'élimination de type :

$$\begin{gathered}\Pi\ (A : \texttt{Type})\ (R : A \to A \to \texttt{Prop})\ (P : A \to \texttt{Type}),\\ (\Pi\ x : A, (\Pi\ y : A, R\ y\ x \to \mathsf{Acc}\ R\ y) \to (\Pi\ y : A, R\ y\ x \to P\ y) \to P\ x) \to\\ \Pi\ x : A, \mathsf{Acc}\ R\ x \to P\ x\end{gathered}$$

En combinant cet éliminateur avec une preuve de bonne fondation de R, écrire une fonction récursive bien fondée sur R revient donc à écrire une fonctionnelle de type Π

$x : A$, $(\Pi\ y : A,\ R\ y\ x \to P\ y) \to P\ x$. Le premier produit introduit x, l'argument récursif initial, et le deuxième le prototype récursif. On peut faire des appels sur tout y plus petit que x par R. Program permet d'utiliser cette construction implicitement si on lui indique qu'on veut écrire une fonction par récursion bien fondée sur une relation donnée. En fait on dé-curryfie la fonctionnelle pour typer le corps d'une fonction récursive f sous l'hypothèse $f : \Pi\ \{\ y : A \mid R\ y\ x\ \},\ P\ `y$. Ainsi, on génère automatiquement les obligations correspondant à la décroissance des arguments récursifs et les preuves n'ont pas à apparaître dans le terme :

```
Program Fixpoint id_wellfounded (a : nat) { wf lt a } : nat :=
  match a with
    | 0 ⇒ 0
    | S n ⇒ S (id_wellfounded n)
  end.

  Next Obligation. apply lt_wf. Defined.
```

Cette fois, on peut rendre toutes les obligations opaques si on le désire, mais en revanche il faut que la preuve de bonne fondation (ici lt_wf) soit transparente puisqu'on l'utilise pour calculer.

On peut utiliser n'importe quel ordre et en généralisant un peu, n'importe quelle mesure sur les arguments, toujours en montrant qu'elles décroissent à chaque appel récursif. Par exemple l'équivalent de l'exemple précédent est (l'ordre par défaut sur les naturels est lt) :

```
Program Fixpoint id_wellfounded' (a : nat) { measure id } : nat :=
  match a with
    | 0 ⇒ 0
    | S n ⇒ S (id_wellfounded' n)
  end.
```

On peut aussi utiliser cette construction pour faire des récurrences sur un objet de type sous-ensemble sans avoir à curryfier :

```
Program Fixpoint predrec (x : nat | x ≠ 0) { measure proj1_sig x } : nat :=
  match x with
    | O ⇒ !
    | S 0 ⇒ 0
    | S x' ⇒ S (predrec x')
  end.
```

Nous verrons dans l'exemple des Finger Trees un autre exemple de l'utilité des mesures.

4.4 Conclusion

Nous avons implémenté une méthode de développement basée sur le langage Russell permettant de typer les programmes dans un système plus flexible, résoudre interactivement les obligations générées par le typage et obtenir finalement un terme Coq bien formé. On a rendu cette extension d'autant plus utile en étendant la construction de filtrage primitive en Coq et en ajoutant le sucre syntaxique nécessaire pour traiter la récursion bien fondée. On verra sur un exemple non-trivial au chapitre 11 l'aide apportée par notre outil pour programmer avec des types dépendants.

Chapitre 5

Conclusion

Sommaire

Nous avons développé un langage de programmation plus souple que le langage de Coq mais conservant sa richesse d'expression, notamment l'utilisation de types dépendants. Il permet de séparer la description algorithmique de la vérification. La correction des termes engendrés est garantie par notre preuve de correction et par le système cible CCI qu'on sait cohérent. En combinaison avec le système d'extraction de Coq, on a la garantie que tout terme de Russell traduisible dans Coq et dont les obligations sont prouvables aura un terme extrait dans ML de même contenu algorithmique.

Utilisations de Program L'extension Program est disponible depuis 2005 et a déjà été utilisée dans un certain nombre de projets extérieurs, notamment :

- Razet (2008) a développé un simulateur de machines d'Eilenberg finies à l'aide de Program. L'ensemble de trois fonctions mutuellement récursives sur une preuve d'accessibilité fortement spécifiées qui implémente l'évaluation des automates sont écrites avec Program.
- Le projet Ynot intégrant la Théorie des Types de Hoare (Nanevski, Morrisett, et Birkedal 2007) au-dessus de Coq utilise Program pour développer les squelettes algorithmiques de programmes à effets et générer les obligations correspondantes : on programme dans une monade généralisée où la combinaison d'actions monadiques génère des obligations.
- Wilson (2008) étend Program par des tactiques de résolution d'obligations automatiques basées sur des techniques de haut-niveau comme le "*rippling*".

5.1 Travaux connexes

On présente maintenant les différents travaux reliés autour des systèmes avec types dépendants et en particulier les travaux autour du type sous-ensemble.

Program peut être vu comme un point dans l'espace de design des langages à types dépendants. Sa principale originalité est la séparation du programme et de la preuve, à la

manière de PVS (Owre, Shankar, Rushby, et Stringer-Calvert 1999), grâce à son traitement des types sous-ensemble. L'extension de la coercion aux types inductifs est originale, mais on retrouve la même idée de coercion automatique derrière les langages à types dépendants restreints sur des domaines décidables comme DML sur l'arithmétique de Presburger (Xi et Pfenning 1999), Twelf (Licata et Harper 2008), voir indécidables (Oury et Altenkirch 2008) et toute la théorie de la déduction modulo et son intégration avec les systèmes à types dépendants (Dowek, Hardin, et Kirchner 2003; Blanqui et al. 2007). Dans ces différents systèmes en revanche, on utilise une coercion *implicite* par l'utilisation de la règle de conversion, ce qui est une façon d'ajouter la règle de réflexion de la théorie des types extensionnelle (Martin-Löf 1975) de façon contrôlée. Ici la coercion est explicite et permet de montrer des équations arbitrairement complexes par du raisonnement logique, de façon similaire à ATS (Chen et Xi 2005), mais avec une couche d'élaboration supplémentaire et dans un système unifié, sans la distinction des termes statiques et dynamiques utilisée dans ce système.

5.1.1 Systèmes à types dépendants

PVS Bien sûr, le calcul de coercion par prédicats étant inspiré du "*predicate subtyping*" de PVS (Owre et Shankar 1997), les langages sont très similaires. Cependant, PVS et Coq (et donc Russell) diffèrent aussi beaucoup dans leur conception : le système PVS, bien qu'intégrant un λ-calcul avec types dépendants, n'est pas basé sur l'idée que ces termes forment les témoins des preuves du système. En conséquence, rien dans ce système ne témoigne de la construction d'une preuve, et l'on doit donc faire confiance à la totalité du système, par exemple le générateur de TCCs ou les procédures de décisions utilisées. Pour Coq, les termes, les preuves et les types font partie de l'unique catégorie des termes que le noyau manipule. La vérification du type d'un terme de preuve à tout moment permet de vérifier qu'une preuve réalise bien son énoncé. C'est évidemment là que réside toute la difficulté de notre traduction : le "*predicate subtyping*" est-il bien toujours anodin vis-à-vis de la validité d'un énoncé ? On répond par l'affirmative : un témoin d'une spécification utilisant la coercion par prédicats peut toujours être traduit vers un terme de Coq n'utilisant pas cette notion, modulo la résolution des obligations de preuves.

De par son architecture, Program se distingue aussi de PVS ou des langages de programmation comme Agda, Concoqtion ou ATS par le fait qu'il s'*élabore* vers un noyau bien étudié et bénéficiant d'une implémentation mature, celui de Coq. Cette architecture est très similaire à celle d'Epigram qui est lui aussi conçu (au moins en théorie) sur un noyau et des couches d'élaboration successives. À vrai dire, le système de coercions à la base de Program pourrait très bien être transposé dans Epigram ou Agda puisqu'il s'appuie uniquement sur les fondations communes de tout les systèmes basés sur la théorie des types.

Program Bien entendu, Program est proche dans son objectif de l'ancienne tactique Program de Parent (1995a). Cette méthode utilisait des heuristiques pour retrouver un terme complet Coq à partir de son type et d'un réalisateur dans F^ω, tandis que la nôtre est plus complète mais fait moins de choses automatiquement : on laisse en grande partie à l'utilisateur le soin de montrer que son terme réalise bien la spécification, en s'assurant qu'on ne perd pas d'information dans les obligations. Le mécanisme était aussi directement lié à l'extraction qui à l'époque donnait des termes de F^ω, alors que nous en sommes complètement séparés.

ATS La famille des "*Applied Type Systems*" ($\mathcal{ATS}$) développée par Xi (2005) forme un ensemble de systèmes basés eux-aussi sur la théorie des types mais où l'on distingue les termes en termes statiques ("*compile-time*") et termes dynamiques ("*runtime*"). Ce système permet d'effacer les termes statiques (typiquement, les types, mais aussi les termes de preuves) à la compilation tout en étant assuré de préserver la sémantique des termes. Cette méthode est très proche de la distinction `Prop`/`Set` utilisée en Coq qui permet à l'extraction de faire de même. Seulement, Coq ne fait pas de ségrégation entres les termes de preuve et les termes du langage utilisé pour écrire les algorithmes, avec raison puisqu'il est indispensable de pouvoir calculer dans les preuves pour faire de la réflexion par exemple. L'avantage d'ATS est qu'il est possible d'avoir un langage de termes impur et non-total puisqu'on ne peut pas l'utiliser au niveau des types pour produire une incohérence.

Dans l'optique d'un langage de programmation, c'est raisonnable, mais l'on aura plus de mal à faire des preuves sur les-dits programmes puisque la logique ne permet pas de parler directement de leur code. Néanmoins, on peut adopter une combinaison de la programmation et de la preuve dans ce langage (Chen et Xi 2005), en manipulant des termes des deux mondes à la fois. Le système ne supporte pas de mécanisme de "*predicate subtyping*" actuellement, mais pourrait en bénéficier puisque les spécifications à base de sous-ensembles y sont légion. En revanche, il intègre le même genre de règle d'équivalence sur les familles inductives que nous allons voir dans la partie suivante, basée sur l'idée que deux instances d'une famille inductive sont équivalentes si leurs indices sont égaux. De la même façon que DML (Xi et Pfenning 1999), ATS considère donc que le type list $(n+m)$ est équivalent au type list $(m+n)$ si une procédure de décision est capable de montrer que $n+m=m+n$. Encore une fois, on s'appuie sur la correction d'un outil externe et aucun témoin n'est gardé pour une telle coercion.

Agda, Epigram On pourrait étendre les langages de programmation fonctionnelle Agda 2 (Norell 2007) et Epigram (McBride et McKinna 2004) de manière complétement identique à notre élaboration de Russell vers CCI. Ces deux langages ne bénéficient pas encore de systèmes de coercions similaires à celui proposé ici, mais sont des terrains tout aussi bien préparés à de telles extensions, voire même mieux préparés que Coq. En effet, ils ont tous les deux un traitement plus mature des variables existentielles et des théories équationnelles incluant des règles η par exemple.

Pangolin Le système Pangolin (Régis-Gianas 2007) permet d'écrire des programmes purements fonctionnels, polymorphes et d'ordre supérieur à la ML et des spécifications dans une logique de même puissance. Le système est basé sur une logique de Hoare (Hoare 1969) permettant donc de spécifier pré- et post-conditions d'une fonction et d'un générateur de conditions de vérification. Les obligations peuvent être soit résolues dans Coq, soit traduites dans une logique de premier ordre (pas nécessairement polymorphe ni même typée) et envoyées à des prouveurs automatiques de la famille SMT ("*Satisfiability Modulo Theories*") tels qu'Alt-Ergo (Conchon et Contejean) ou Simplify (Detlefs, Nelson, et Saxe 2005).

La problématique de réflexion du contrôle (analyse de cas) mais aussi des valeurs (qui sont ici des programmes potentiellement non-terminants) dans la logique apparaît aussi dans ce système, accentuée par la séparation des différentes catégories de valeurs, programmes, types et spécifications. Le système permet néanmoins de traiter élégamment ces problèmes sans céder à la duplication systématique d'un monde dans un autre : le reflet logique d'une fonction est réduit à sa spécification. En comparaison, nous n'avons pas de telles restrictions sur les différentes catégories syntaxiques, mais nous utilisons

effectivement à peu près le même style de preuve avec Program, en utilisant rarement le raisonnement *a posteriori* sur le code.

5.1.2 Types sous-ensembles

Théorie des Types

La première introduction du type sous-ensemble date de l'introduction de la théorie des types de Martin-Löf (1984), sous la forme d'une paire dépendante dont la seconde composante est une preuve.

Martin-Löf L'une des premières introductions au type sous-ensemble pour la programmation avec types dépendants est donnée par Nordström, Petersson, et Smith (1990, partie 2), qui montrent comment construire une variante de la théorie des types de Martin-Löf intensionnelle dans laquelle le raisonnement sur les sous-ensembles est extensionnel. Ce système résoud le problème posé par le type sous-ensemble donné par la théorie des types de base qui est une simple paire dépendante contenant un terme de preuve. Ils éliminent ce terme non-calculatoire pour pouvoir donner une règle d'élimination plus satisfaisante aux objets de type sous-ensemble. Dans la théorie des sous-ensembles, on peut donc dériver à l'aide des éliminateurs pour les sous-ensembles :

$$\frac{a \in \{x : A \mid P(x)\} \qquad Q(x)\ true\ [x \in A, P(x)\ true]}{Q(a)\ true}$$

$$\frac{a \in \{x : A \mid P(x)\} \qquad c(x) \in C(x)\ [x \in A, P(x)\ true]}{c(a) \in C(a)}$$

Le premier éliminateur permet de dériver en particulier $P(a)$ pour tout $a \in \{x : A \mid P(x)\}$ mais aussi $Q(a)$ pour toute propriété impliquée par P. Comparé à Russell, on a un système moins fort puisque l'on se limite à coercer un objet d'un sous-ensemble à un autre quand ceux-ci sont logiquement équivalents et contiennent donc les mêmes éléments. Au contraire, Russell permet d'injecter un naturel dans les naturels positifs par exemple. En revanche, la traduction de cette théorie des sous-ensembles dans la théorie de base est directe, et sans conditions de bords comme les obligations générées par Russell. On pourrait restreindre les coercions de Russell pour que toutes les dérivations de preuves générées par la règle de coercion soient dérivées sur une instance générale du type support et non pas sur l'objet spécifiquement coercé pour obtenir un système équivalent. On peut donc voir Russell comme une réalisation effective de la traduction donnée par Nordström et al., mais dans CCI plutôt que la Théorie des Types de Martin-Löf.

Théories des types extensionnelles et intensionnelles Le type sous-ensemble fait partie de l'ensemble des notions extensionnelles ayant des contreparties intensionnelles intéressantes, avec l'extensionnalité fonctionnelle et les quotients. Ils ont donc été étudiés par ce biais dans le cadre d'intégration de ces concepts en théorie des types par Salvesen et Smith (1988); Hofmann (1993, 1995); Jacobs (1999); Boutin (1997); Oury (2005). En général, il s'agit toujours de créer un nouveau système extensionnel qui se traduit sans conditions vers une théorie de base intensionnelle mais cela pose toujours des problèmes du fait de l'incomplétude de ces traductions : les deux systèmes ne peuvent pas être équivalents.

Notre approche est d'accepter de travailler au-dessus de la théorie des types de base en donnant un sens précis à notre langage *via* son élaboration. On retrouve la même idée derrière l'utilisation de sétoïdes en Coq : on modélise un concept extensionnel explicitement par un encodage intensionnel, et on améliore l'assistant de preuve pour rendre cette élaboration la plus transparente possible. Pour cela on développe des outils comme les coercions implicites (Saïbi 1997) qui internalisent le sous-typage des enregistrements, ou la réécriture généralisée (chapitre 9) qui permet le raisonnement équationnel sur d'autres relations que l'égalité primitive, comme l'égalité sétoïde (Barthe, Capretta, et Pons 2003). Cette méhode a déjà prouvé son utilité dans des développements mathématiques non-triviaux (Capretta 2002; O'Connor et Spitters 2008; Cruz-Filipe, Geuvers, et Wiedijk 2004).

Indifférence aux preuves L'indifférence aux preuves est un concept extensionnel qu'il est impératif de supporter dans les approches fondationnelles telles que celles de Coq, Epigram ou Agda où l'évaluation inutile de preuves peut rendre le système inutilisable pour faire du calcul avec des performances raisonnables. En Coq en particulier, la programmation avec types dépendants est peu utilisée en partie du fait de la difficulté d'obtenir des définitions se comportant bien calculatoirement au sein même de l'assistant, malgré l'utilisation de la machine virtuelle de Grégoire et Leroy (2002).

La métathéorie du Calcul des Constructions étendu par PI a été étudiée par Miquel et Werner (2002). Récemment, Werner (2006) a aussi donné une méthode pour l'implémenter effectivement dans Coq. Il a aussi montré comment cette extension se compare formellement au système PVS. Nous avons travaillé en collaboration avec Werner sur ce sujet mais des difficultés subsistent liées à la hiérarchie d'univers et à certaines constructions autorisées dans Prop en Coq, notamment la récursion bien fondée. Cela fait donc partie de nos perspectives.

La proposition de Altenkirch, McBride, et Swierstra (2007) qui présente le noyau d'Epigram 2 montre l'importance de ce principe pour programmer avec des types dépendants et propose de changer radicalement la définition de l'égalité propositionnelle pour y parvenir.

Prouté (2007) donne une autre présentation intéressante de cette question à base de la théorie des topos, liée aux modèles catégoriques de la théorie des types.

"*Deliverables*" Enfin, McKinna et Burstall (1993) étudient une méthode de programmation basée en partie sur les types sous-ensembles dans LEGO et une interprétation catégorique du raffinement de programmes. L'idée de base est de considérer les spécifications comme des paires constituées par un type et un prédicat sur ce type et de combiner des programmes appelés "*deliverables*" (délivrables), qui transforment les spécifications.

```
Definition spec := { A : Type & A → Prop }.
```

Un déliverable est alors un programme sur les supports des spécifications ainsi qu'une preuve que si la précondition est vraie sur l'argument alors la post-condition est vraie pour l'application de la fonction à cet argument

```
Definition deliverable (pre : spec) (post : spec) : Type :=
  let (prea, prep) := pre in
  let (posta, postp) := post in
    { f : prea → posta | Π x, prep x → postp (f x) }.
```

On peut alors définir des délivrables et les composer pour créer de plus gros programmes de façon modulaire. Une extension des délivrables pour écrire des spécifications plus générales où la pré- et la post-condition partagent une variable est décrite dans la thèse de McKinna (1992). Il y étudie l'interpétation catégorique de ces combinateurs et

démontre sur des exemples qu'ils sont suffisament riches pour spécifier et réaliser des programmes non-triviaux. L'extraction de délivrables est triviale puisqu'il suffit de prendre le témoin du type sous-ensemble correpondant. Le problème principal de cette approche est la difficulté d'utilisation des combinateurs dans les systèmes basés sur la théorie des types avec inductifs qui favorisent l'utilisation de la notation λ et du filtrage. Néanmoins, cette approche est facilitée par les modes de raffinement comme celui proposé par Program qui permettent de définir des délivrables en séparant la définition du témoin de la preuve des obligations pour les preuves associées.

5.1.3 Coercions

Les travaux de Reynolds (1980) sur la construction d'ensembles de coercions cohérents pour modéliser la surcharge d'opérateur fonde un ensemble de travaux autour de l'idée d'intégrer un système de coercion au-dessus ou au sein d'un système de type. Par exemple l'interprétation de l'héritage par des coercions de Breazu-Tannen, Coquand, et Gunter (1991) est faite par traduction d'un calcul avec héritage implicite dans un système où celui-ci est rendu explicite. C'est une compilation définitionnelle comme la nôtre vers un λ-calcul typé, mais dont l'objectif est de donner un modèle à l'héritage du langage source via un modèle du langage cible. Ces formalisations s'appuient sur des traitements catégoriques de la coercion qu'il serait intéressant de considérer pour notre système.

Pour les PTS, Barthe (1995) étudie l'intégration des coercions au niveau du langage source et pose comme problème central la vérification de la cohérence de ces systèmes. Barthe et Pons (2001) proposent une variante de ce système qui intègre les isomorphismes de types définitionnels dans la conversion. En comparaison de ces travaux, notre condition de cohérence se réduit à la preuve que deux coercions inverses donne l'identité, ce qu'on a montré pour le système de coercion de sous-ensembles. Barthe (2005) a récemment proposé une nouvelle notion de coercions symétriques ("*back-and-forth*") avec une relaxation de la cohérence qui requiert habituellement que les deux coercions apparaissent comme un isomorphisme par convertibilité.

Les travaux de Luo (1996); Chen (2003) sont dans le même esprit de témoigner d'un sous-typage par des coercions en théorie des types. Il s'agit toujours dans ce cas de témoigner d'une relation de sous-typage "totale" entre types sans jamais faire intervenir de relation sur les valeurs ou de coercions partielles *via* la génération d'obligations.

Théories des types et contrats

D'autres approches basées sur des théories des types non dépendantes voire même des approches non typées utilisent les sous-ensembles de façon plus ou moins implicite pour la spécification.

"*Refinement Types*" L'idée des types raffinements est d'ajouter une forme limitée de sous-typage dans un langage basée sur des types simples. Freeman et Pfenning (1991) décrivent par exemple une extension de ML où l'on peut raffiner un type de donnée par un autre en enlevant un certain nombre de ses constructeurs. Le système de type se charge de vérifier que les raffinements sont correctememt utilisés. On peut par exemple spécifier le type des listes singletons de la façon suivante :

```
datatype α list = nil | cons of α * α list
rectype α nonempty = cons (α, α list)
```

On peut ensuite déclarer des opérations sur les listes et les listes non-vides et le système vérifiera la bonne utilisation des raffinements. Il est clair que cet encodage cache en fait

un type sous-ensemble : nonempty correspond à la déclaration de type $\{l : \alpha \text{ list} | \exists\ a\ l', l = \text{cons}\ a\ l'\}$. Le typage dans ce système utilise cependant des types intersections pour pouvoir associer plusieurs types à une fonction et le typage est donc assez différent de ce qu'on pourrait obtenir en faisant une traduction dans Russell.

Récemment, Lovas et Pfenning (2008) ont implémenté un système similaire dans LF qui traite donc les types dépendants. D'autre part, Mandelbaum, Walker, et Harper (2003) décrivent un système avec raffinements permettant de spécifier non seulement des raffinements sur les types de données mais aussi sur le comportement des programmes en permettant de donner des spécifications dans une variante décidable de la logique linéaire.

"*Contracts*" Enfin, l'ensemble des langages de spécifications à base de contrats peuvent être vus comme manipulant des types sous-ensembles. Les spécifications de pré- et post-conditions qu'on retrouve dans ces systèmes se traduisent directement vers la construction d'une fonction dépendante entre types sous-ensembles. On peut distinguer en particulier le langage Sage (Gronski, Knowles, Tomb, Freund, et Flanagan 2006) qui utilise explicitement des sous-ensembles pour spécifier le comportement de fonctions mais dont l'interprétation donne lieu suivant la complexité de la spécification à un test dynamique ou à une vérification statique.

5.2 Perspectives

Nous avons de multiples directions dans lesquelles étendre notre système. L'ajout des inductifs et de la récursion structurelle, des univers cumulatifs et des définitions dans les contextes semblent intégrables facilement dans le formalisme.

Russell comme cible On pourrait étudier l'interprétation de certains des calculs proposés pour la spécification avec contrats directement dans Russell, et en particulier les traductions fonctionnelles existantes (Filliâtre 2003; Charguéraud et Pottier 2008). On pourrait ainsi obtenir gratuitement des résultats de correction relative des programmes par traduction en simplifiant celles-ci puisqu'on peut assigner des types riches arbitrairement dans Russell.

Preuve mécanisée

L'extension de la preuve mécanisée pour certifier la correction de l'interprétation semble tout à fait envisageable, en utilisant le support pour la preuve sur les familles inductives et l'automatisation qu'on développera dans les parties suivantes.

On pourrait aussi étendre la relation de coercion à d'autres notions d'équivalence, et en particulier essayer d'intégrer la coercion au sens de (Saïbi 1997) dans le système et la preuve formelle. L'extension aux coercions entre familles inductives (§ 4.2) pourrait aussi faire l'objet d'une restriction qu'on pourrait intégrer à la preuve.

Il devrait être possible de prouver la normalisation forte de notre calcul directement en utilisant la technique de Geuvers (1994) qui nécessite une présentation déclarative du système mais semble suffisament flexible pour être adaptée à la relation de coercion.

On pourrait aussi intégrer cette preuve à la chaîne de compilation certifiée formée à partir du projet CompCert (Gallium, Marelle, CEDRIC, et PPS 2008) qui permettrait à terme de certifier la compilation d'un terme du CC vers du code machine. On pourrait vérifier que la sémantique d'un programme ML est préservée à travers une annotation du programme par des spécifications donnant un programme Russell, puis sa traduction vers CC grâce à la preuve de correction et les preuves de Barras (1999), son extraction vers ML

(Glondu 2007) et enfin sa compilation vers du code assembleur (Blazy, Dargaye, et Leroy 2006; Leroy 2006).

Raffinement Nous sommes partis du principe que la preuve et le programme doivent être séparés pour plus de clarté et avons donné corps à cette idée avec Program. Néanmoins, lorsqu'on programme avec des familles inductives et non pas avec de "simples" types sous-ensemble, on ne peut pas toujours faire l'économie d'un mode de raffinement qui aide l'utilisateur à créer son programme incrémentalement, par un dialogue entre l'homme et la machine. Ce mode interactif n'est aujourd'hui disponible que lorsqu'on développe des preuves dans Coq, mais il n'est pas adapté à la programmation puisque la définition raffinée n'apparaît pas toujours dans le script de preuve.

Program permet de placer des "*wildcard*"s "_" à n'importe quel endroit du terme lorsqu'on veut générer une obligation ou qu'on désire connaître le contexte et la contrainte de typage à un certain endroit du programme. On peut donc programmer à l'aide de cette technique dans un mode de raffinement relativement artificiel : on ne donne pas aujourd'hui d'outils pour retrouver facilement de l'information sur les trous.

Nous comptons expérimenter des variations de Program et du mode de preuve pour s'approcher d'un mode de raffinement similaire à Agda 2 ou Epigram. Pour cela, il faudra sans doute d'abord intégrer Russell dans le mode de preuve de Coq et mettre à profit le travail entrepris par Arnaud Spiwack.

Deuxième partie

Classes de types dépendantes

Chapitre 6

Introduction et état de l'art

Sommaire

Dans cette partie, nous allons nous intéresser à l'ajout d'un système de classes de types à Coq. Ce système permet de faire des développements abstraits succincts et clairs grâce à un mécanisme de surcharge qui s'intègre naturellement aux fonctionnalités d'abstraction du langage et à l'environnement de développement et de preuve. Il peut aussi être utilisé comme pont entre l'utilisateur et le système de tactiques, ce que nous démontrerons dans un deuxième temps avec une nouvelle implémentation de la réécriture généralisée dans Coq.

Il s'agit plus ici d'ingénierie du système de preuve que d'une extension du calcul sous-jacent. On peut en effet considérer le calcul des constructions comme un assembleur qu'il est possible de programmer à un plus haut niveau d'abstraction à l'aide des extensions que nous avons présentées jusqu'ici. Les classes de type apportent à ce langage une forme de *programmation logique* ainsi qu'un support léger pour l'écriture de programmes modulaires.

Initialement, les classes de types ont été introduites dans le but de gérer la surcharge uniforme de fonctions dans le langage Haskell (Wadler et Blott 1989). La surcharge donne la possibilité d'utiliser le même nom pour faire référence à des fonctions différentes, mais qui ont chacune un type instance du même schéma de type polymorphe. L'exemple paradigmatique est le schéma de type des fonctions d'égalité : $\forall\alpha.\alpha \to \alpha \to \mathbb{B}$. Toute fonction d'égalité est une instance de ce schéma, mais il n'est pas possible d'écrire une fonction polymorphe et non triviale de ce type puisqu'elle devrait décider l'égalité sur tout type et ce sans avoir accès à leur représentation (on peut le démontrer par paramétricité (Reynolds 1983; Wadler 1989)). Lorsqu'on veut écrire une fonction qui teste l'égalité sur un type α

arbitraire, il est donc nécessaire de passer en paramètre une implémentation de l'égalité sur α. On pourrait aussi passer une représentation du type α et définir la fonction par cas sur celle-ci, mais on se fixerait un univers particulier (Vytiniotis et Weirich 2007).

Les classes de type permettent de rendre ce passage de paramètres moins ad-hoc en formalisant les fonctionnalités qu'un type doit supporter dans une déclaration de classe, et en permettant de quantifier sur des types *respectant* cette interface. Il n'y a plus alors qu'à implémenter des instances de ladite classe pour pouvoir appliquer ces fonctions polymorphes à des objets concrets.

Il existe aujourd'hui deux implémentations majeures des classes de types, l'une dans le langage de programmation Haskell et l'autre dans l'assistant de preuve Isabelle. Les deux systèmes ont des fonctionnalités de base identiques mais diffèrent dans leur intégration aux systèmes respectifs et leurs extensions.

Nous allons maintenant présenter plus formellement les systèmes de classe de type tels qu'ils sont implémentés dans Haskell 98 (§6.1) ainsi qu'un certain nombre d'extensions populaires. Nous nous intéresserons ensuite à l'implémentation des *type classes* dans Isabelle. Nous reviendrons sur les solutions pour le développement structuré dans les assistants de preuve à la fin de cette partie (§10.1).

6.1 Les classes de types dans Haskell 98

Le langage Haskell fut le premier langage à intégrer un système de classes de type. C'est, avec l'introduction des monades, une des principales contributions de ce langage à la pratique de la programmation fonctionnelle. Comme indiqué plus haut, le problème qu'étaient sensés résoudre les classes à leur introduction était la surcharge d'opérateurs polymorphes tels que l'égalité.

6.1.1 "*Equality types*"

À cette époque, seul le langage Standard ML (SML, Milner, Tofte, et Harper (1990)) proposait une solution à ce problème utilisant les "*equality types*". On introduit une extension à la syntaxe du langage pour pouvoir exprimer qu'une variable de type polymorphe supporte une opération d'égalité booléenne. La signature de l'égalité générique devient alors : $''\alpha \to ''\alpha \to \mathbb{B}$, où les doubles guillemets indiquent les types supportant l'égalité (les simples guillemets indiquant les types polymorphes usuels). Le compilateur se chargeait lui-même de générer l'égalité sur les types de l'utilisateur par programmation générique : on fait passer l'égalité sur les types de base par les constructeurs de type usuels (produit cartésien, sommes disjointes, point-fixe). Cette technique limitée (Gunter, Gunter, et MacQueen 1993) permet d'obtenir une opération surchargée d'égalité mais ne donne pas de contrôle à l'utilisateur et est restreinte à ce seul cas.

6.1.2 "*Type classes*"

Cependant l'idée peut être généralisée à des contraintes de *classes* et à des opérations à peu près arbitraires. C'est l'idée des classes de type. Pour le cas de l'égalité, on introduit la classe de type Eq paramétrée par un type comme suit :

```
class Eq α where
  (==) :: α → α → bool
```

Toute instance de la classe Eq sur un type τ doit fournir une implémentation de l'égalité sur τ. Par exemple, l'égalité sur les booléens peut s'écrire :

```
instance Eq bool where
  x == y = if x then y else not y
```

Il est dès lors possible d'écrire des fonctions qui utilisent l'égalité sur les booléens en utilisant la fonction (==). Plus intéressant, il devient aussi possible d'écrire des fonctions *polymorphes* utilisant l'égalité sur les types qui la supportent. On utilise pour cela un nouveau type de flèche, qui permet de spécifier des contraintes de classes de manière prénexe.

```
=/= :: Eq α ⇒ α → α → bool
x =/= y = not (x == y)
```

La contrainte Eq α indique que la fonction d'inégalité =/= s'applique à tout type supportant l'égalité, et cette contrainte est utilisée lors de l'appel à ==. On remarque ici que le mécanisme est compositionnel, les contraintes se propageant naturellement de fonctions en fonctions *via* leurs types. En fait, l'inférence de types n'est pratiquement pas perturbée par les classes, elle reste décidable en Haskell 98 avec un mécanisme simple : les contraintes de classes résiduelles sont automatiquement généralisées (modulo certaines restrictions imposées sur la forme des instances). Par exemple, si l'on omet la signature ci-dessus, elle sera correctement inférée. Le type d'x et y sera généralisé par une variable de type α comme dans tout système basé sur l'inférence à la Hindley-Milner puis la contrainte résiduelle Eq α provenant de l'appel à (==) sera elle aussi généralisée. On restreint la généralisation aux contraintes de la forme Id $\overrightarrow{\alpha}$, c'est-à-dire où tous les paramètres sont des variables. Les autres doivent être résolues au moment de la définition.

6.1.3 Paramétrisation

Naturellement, on peut étendre les définitions paramétrées aux déclarations d'instance elles-mêmes pour obtenir des instances paramétrées ou conditionnelles. Par exemple, on sait décider l'égalité sur un produit cartésien si l'on sait la décider sur chaque composante séparément :

```
instance (Eq α, Eq β) ⇒ Eq (α × β) where
  (x1, x2) == (y1, y2) = x1 == y1 && x2 == y2
```

Il est ainsi possible de créer un ensemble d'instances qui permet de dériver génériquement une implémentation pour un type arbitraire, formé à partir des constructeurs de base. Plusieurs mécanismes de dérivations ont été développés comme extensions au langage permettant de dériver des instances pour des types utilisateurs généraux et un large nombre de classes (Mitchell et Runciman 2007; Weirich 2006). Il est ainsi possible d'introduire un type algébrique et de demander la dérivation automatique d'une instance d'Eq sur ce type :

```
data Tree α = Leaf | Node α × Tree α × α
     deriving Eq
```

Nous n'entrerons pas dans les détails de ces extensions mais les possibilités de programmation générique liées aux classes sont un domaine intéressant à poursuivre.

6.1.4 Superclasses

Il est aussi possible de paramétrer une classe par une autre, pour créer une hiérarchie. Typiquement, on fait cela pour rajouter des opérations à une classe existante :

```
class (Eq α) ⇒ Ord α where
```

$(<) :: \alpha \to \alpha \to \mathsf{bool}$

La classe Ord étend la classe Eq avec une opération supposée implémenter un ordre strict sur le même type. Lorsqu'on veut implémenter une sous-classe telle que Ord sur un type donné, il faut au préalable avoir implémenté ou supposé donnée une instance de chacune des superclasses sur ce type. En contrepartie, on peut utiliser les méthodes des superclasses partout où l'on peut utiliser les méthodes de la classe elle-même.

Lorsqu'on écrit une fonction sur une classe, on contraint donc toujours implicitement par ces superclasses. Il n'y a pas de contraintes sur la forme des hiérarchies engendrées en Haskell 98, du fait essentiellement qu'un seul paramètre par classe est autorisé et qu'on ne peut définir qu'une instance par type.

Sélection

La procédure de sélection des instances correspondant à une contrainte de classe se déroule après l'unification des types. Elle tente d'unifier chaque contrainte avec les instances déclarées dans l'environnement et sélectionne la première qui s'unifie puis ajoute éventuellement les contraintes sur les superclasses résultant de l'instanciation. La saturation termine lorsque toutes les contraintes ont été résolues. Suivant qu'on ait donné une contrainte de typage explicite ou non et suivant la forme des contraintes résiduelles, l'algorithme de typage termine avec succès ou renvoie une erreur indiquant une contrainte non satisfaite ou non généralisable. L'algorithme fonctionne donc par réduction des contraintes ("*context reduction*"), et d'importantes restrictions sur la forme des classes et des instances permettent d'assurer la terminaison et le déterminisme de l'algorithme. Notamment, les déclarations d'instance ne doivent pas se chevaucher ("*overlapping instances*") et les contraintes doivent syntaxiquement décroître à l'application d'une instance donnée. C'est à peu près les restrictions nécessaires pour obtenir un système de réécriture orthogonal avec une terminaison structurelle.

6.1.5 Extensions

De nombreuses extensions ont été proposées au système de base pour supporter des définitions de classes plus générales et s'affranchir des restrictions sur la forme des instances qui permettent de s'assurer de bonnes propriétés de la procédure de sélection. On peut par exemple donner des indications au compilateur GHC pour relaxer certaines contraintes sur les déclarations d'instances.

Nous présentons ici rapidement deux extensions populaires, la première étant implantée dans les compilateurs les plus utilisés (GHC et Hugs). Nous ne nous intéresserons pas ici aux différents critères permettant d'assurer la terminaison et le déterminisme de la résolution (Sulzmann, Duck, Peyton-Jones, et Stuckey 2007) que nous discuterons section 7.1.3.

"*Multi-parameter type classes*"

L'extension la plus naturelle est la possibilité d'indexer une classe par plus d'un type, de façon à relier plusieurs types et constructeurs de type entre eux (Jones 1992, 2000). L'exemple le plus courant est l'interface des ensembles finis qui peut s'exprimer par une classe Set :

```
class Set α τ where
  empty :: τ
  singleton :: α → τ
```

```
union :: τ → τ → τ
intersection :: τ → τ → τ
member :: α → τ → bool
```

Le paramètre α correspond au type des éléments contenus dans l'ensemble et τ au type de l'ensemble implanté. Cette interface abstrait effectivement le type de l'implémentation des ensembles mais nous permet de parler du type des éléments séparément. On peut ainsi développer un algorithme sur les ensembles d'entiers arbitraires en utilisant une contrainte (Set Int α) qui sera applicable à n'importe quelle implémentation. On peut instancier Set sur les ensembles implémentés par des listes sans doublons par exemple :

```
instance (Eq α) ⇒ Set α [α] where
  empty = []
  singleton x = [x]
  union x y = nub (x ++ y)
  intersection s t = [ x | x == y, x ← s, y ← t ]
  member = elem
```

Les classes à paramètres multiples posent un certain nombre de problèmes d'inférence de type. Par exemple le type de la méthode empty est ambigu, puisque le type des éléments n'y apparaît pas. Ainsi on ne peut pas décider à quelle instance de Set il est fait référence. Lorsque les types ne sont pas ambigus, l'inférence trouve des types trop généraux, par exemple :

```
f :: (Set α τ, Set β τ) ⇒ α → β → τ
f x y = union (singleton x) (singleton y)
```

"*Functional Dependencies*" et "*Associated types*"

Le problème vient du fait que l'interface Set est trop générale car elle n'exprime pas la dépendance entre α et τ. Un système permettant d'exprimer les dépendances fonctionnelles entre paramètres (Jones 2000) permet de résoudre ces problèmes. Une autre extension, permettant d'inclure des types en plus des méthodes exportées par une classe, et qui subsume les dépendances fonctionnelles a aussi été développée (Chakravarty, Keller, Jones, et Marlow 2005) (voir §10.2).

"*Named instances*"

Une dernière extension est la possibilité de nommer les instances et d'autoriser simultanément plusieurs instances de la même classe sur les mêmes paramètres (Kahl et Scheffczyk 2001). Il est alors possible de désambiguer les appels aux fonctions surchargées en donnant explicitement le nom de l'implémentation. On peut ainsi définir deux instances de l'égalité sur les listes, la première réalisant l'égalité point-à-point et la seconde l'égalité modulo permutation.

Maintenant que nous avons fait un tour rapide des classes de types dans Haskell, nous allons nous tourner vers l'implémentation qui en est faite dans l'assistant de preuve Isabelle.

6.2 Les classes de types dans Isabelle

Isabelle (Nipkow, Paulson, et Wenzel 2002) est un assistant de preuve générique, dont l'instanciation la plus courante Isabelle/HOL est basée sur une logique d'ordre supérieur

multi-sortée. Depuis ses débuts l'assistant implémente les classes de types, en les intégrant à l'algorithme de typage de la logique (Nipkow et Snelting 1991; Nipkow et Prehofer 1993). Il s'agit toujours de classes à un seul argument, qui ne peut être qu'un type.

L'extension principale au système originel de Haskell est la possibilité d'ajouter des lois (parfois appelées axiomes) à une classe. Ainsi il est possible de spécifier l'interaction des opérations définies dans une classe et d'utiliser cette information lorsque l'on fait des preuves. Prenons l'exemple des monoïdes, qu'on peut définir en Isabelle de la façon suivante (exemple pris dans Wenzel (1995)) :

```
consts
  times     :: 'a ⇒ 'a ⇒ 'a (infixl ⊙ 70)
  inverse   :: 'a ⇒ 'a ((-1) [1000] 999)
  unit      :: 'a (1)
axclass monoid < term
  assoc       : (x ⊙ y) ⊙ z = x ⊙ (y ⊙ z)
  left-unit   : 1 ⊙ x = x
  right-unit  : x ⊙ 1 = x
```

Où term peut être vu comme la plus haute superclasse du système. Implicitement, la classe monoid ne parle que des opérations de multiplication et l'unité, c'est sa *signature*. Une classe est ici vue essentiellement comme un prédicat sur un type, sans les opérations associées. On peut donc définir d'autres classes sur la même signature en utilisant $<$ pour indiquer une relation de sous-classe, par exemple :

```
axclass semigroup < term
  assoc   : (x ⊙ y) ⊙ z = x ⊙ (y ⊙ z)
axclass group < semigroup
  left-unit     : 1 ⊙ x = x
  left-inverse  : x-1 ⊙ x = 1
```

Il est important de remarquer qu'une classe est intimement liée au type sur laquelle elle est définie (ici $'a$). C'est ce qui permet ensuite d'y faire référence. Plutôt que d'introduire des lieurs spécifiques comme en Haskell, on utilise la quantification habituelle sur les types. Par exemple pour prouver un lemme sur les groupes il suffit de l'énoncer ainsi :

```
theorem group-right-inverse : x ⊙ x-1 = (1 :: 'a :: group)
```

Ce système de classes permet de développer des théories abstraites structurées en utilisant la surcharge et la hiérarchisation par les superclasses. Notons dès à présent que le système est une couche supplémentaire au système, et dès lors, même s'il peut s'interpréter dans un noyau plus simple (Gordon/HOL dans (Wenzel 1997)), l'implémentation ne s'y réduit pas. Les travaux de Haftmann et Wenzel (2006) ont permis d'expliciter les classes de types en termes de *locales* que nous présenterons à la fin du chapitre (§10.1). Cette interprétation est plus proche de la vision Haskell puisqu'elle remet les opérations au sein de la classe.

6.3 Classes de types dans Coq

Nos motivations pour intégrer les classes de types à Coq sont multiples.

6.3.1 Surcharge

D'une part, on va apporter un support élégant de la surcharge pour la programmation et la spécification. Les classes nous offrent la possibilité de surcharger à la fois les noms

de fonctions mais aussi les lemmes, puisqu'en Coq il s'agit de la même catégorie d'objets. Il devient ainsi possible de définir une classe comme la classe neutral ci-dessous pour représenter toutes les preuves de neutralité d'un élément par rapport à une fonction, et d'utiliser la méthode surchargée is_neutral pour y faire référence :

```
Require Import Arith Setoid.

Class neutral {A} (f : A → A → A) (e : A) : Prop :=
  is_neutral : Π x, f x e = x.
Instance neutral_plus_0 : neutral plus 0.

Goal Π x y, x + y = x + y + 0.
Proof. intros. rewrite (is_neutral (e :=0)). reflexivity. Qed.
```

Cela permet d'utiliser les constructions du langage lui-même pour organiser le développement : les classes et les instances sont ici des objets Coq comme les autres.

6.3.2 Programmation logique

Un autre aspect que nous développerons est la possibilité de faire de la résolution à la Prolog au moment du "*type-checking*" grâce aux classes. On peut en effet voir une déclaration de classe comme un prédicat sur un type, et c'est même une terminologie couramment utilisée en Haskell. Les instances donnent alors les différentes façon de construire une preuve du prédicat instancié sur certaines variables.

Seulement, l'algorithme de sélection utilisé par Haskell (et de même pour Isabelle) n'est qu'une forme très restreinte de la résolution SLD. Nous allons totalement relaxer cette restriction et utiliser un véritable algorithme de résolution avec retour sur trace ("*backtracking*") pour résoudre nos contraintes de classes. Il devient ainsi possible d'intégrer de la résolution lors du typage, le tout avec des prédicats sur des valeurs, grâce au produit dépendant. Par exemple, on peut imaginer écrire :

```
Require Import Arith Omega.

Class NonZero (a : nat) := is_nonzero : a ≠ 0.
Program Instance two_nonzero : NonZero 2.
Program Instance three_nonzero : NonZero 3.
Program Instance mult_nonzero [ NonZero a, NonZero b ] : NonZero (a × b).

Definition div (a : nat) (b : nat) [ nb : NonZero b ] : nat.

Check (div 0 (2 × 3)).
```

On a implicitement résolu le but NonZero (2×3) sous l'hypothèse des instances définies précédemment.

Intégration avec les tactiques

Nous allons finalement modifier cet algorithme de résolution pour pouvoir l'étendre par des tactiques écrites en $\mathcal{L}_{\text{tac}}$ arbitraires. Il devient ainsi possible de déployer des méthodes ad-hoc de résolution des contraintes de classe, propres aux domaine d'étude. Nous démontrerons la puissance de cette architecture dans le chapitre 9 où nous développerons un algorithme de résolution de contraintes d'une forme particulière en grande partie grâce à $\mathcal{L}_{\text{tac}}$.

6.3.3 Sémantique des classes et extensions

Enfin on peut voir ce travail comme une interprétation sémantique des classes et des constructions afférentes (superclasses, sous-structures, résolution des contraintes) dans

une théorie des types avec types dépendants. L'extension très naturelle à l'indexation des classes par des valeurs pose des problèmes d'unification et d'inférence que nous n'avons pas entièrement explorés. Il serait aussi intéressant de formaliser la relation entre ce système et les différentes extensions proposées aux classes dans Haskell, qui introduisent des constructions similaires à celles disponibles dans Coq. La possibilité d'associer des types dans les déclarations de classes par exemple est gratuite dans notre système.

Nous laissons aussi de côté la possibilité de réfléchir la recherche d'instance dans le langage lui-même en faisant de la programmation avec univers (dans le sens représentation des types, voir (Oury et Swierstra 2008) pour plusieurs exemples). Il serait probablement possible d'encoder les classes de Haskell grâce à une telle construction, mais il ne semble pas évident de supporter les classes dépendantes telles qu'on les a introduites de façon générale.

Chapitre 7

Typage et reconstruction

Sommaire

Nous allons maintenant aborder la présentation du système que nous avons implémenté dans Coq. Ce travail est le fruit d'une collaboration avec Nicolas Oury et a été publié à TPHOLs'08 (Sozeau et Oury 2008). Nous présenterons incrémentalement les ingrédients nécessaires à l'implémentation et les capacités du système.

7.1 Technique

Nous proposons un système de classes au-dessus du Calcul des Constructions Inductives implémenté par Coq. Pour cela, il nous faut donc élaborer des programmes sources utilisant la surcharge vers des termes explicites du calcul que le noyau pourra alors vérifier. Dans notre cas, cette élaboration correspond exactement à l'étape de compilation du compilateur GHC de Haskell qui traduit l'utilisation des classes de type à un passage de dictionnaire[1].

L'interprétation d'une classe par un enregistrement ("*record*") est en effet des plus naturelles. Une déclaration de classe devient une déclaration de type enregistrement paramétré et une instance est simplement une valeur de ce type enregistrement, potentiellement instancié. Dès lors, chaque méthode de classe devient une projection et à chaque appel d'une méthode surchargée on permet d'omettre l'instance concernée. Le problème de résolution des classes revient donc à trouver quelle instance de la classe utiliser à chaque appel de méthode surchargée.

[1] D'autres implémentations sont possibles, utilisant par exemple une réification des types qui indexent les classes pour faire le "*dispatch*" (Meacham 2007)

7.1.1 Enregistrements dépendants

Pour implémenter les classes de types, nous allons donc avoir besoin de types enregistrements. En Coq, ces types sont un cas particulier des types inductifs ayant un seul constructeur : les inductifs singletons. On peut voir un type enregistrement comme un Σ-type : chaque champ définit un nom et un type et l'ensemble ordonné de ces types forme un type de n-uplets avec les projections nommées évidentes. Les types dépendants donnent simplement la possibilité de faire référence aux champs précédents dans chaque type. On note $\Sigma\ \overrightarrow{\mathrm{f} : \phi}$ le type défini par induction sur le télescope :

- $\Sigma\ \epsilon \stackrel{\text{def}}{=} \mathrm{unit}$
- $\Sigma\ (\mathrm{f}_1 : \phi_1, \overrightarrow{\mathrm{f} : \phi}) \stackrel{\text{def}}{=} \Sigma\ \mathrm{f}_1 : \phi_1, \Sigma\ \overrightarrow{\mathrm{f} : \phi}$

Syntaxe des déclarations

On verra donc la déclaration de classe :

$$\begin{aligned}&\texttt{Class}\ \mathsf{Id}\ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) :=\\ &\quad \mathrm{f}_1 : \phi_1\ ;\\ &\quad\ \vdots\\ &\quad \mathrm{f}_m : \phi_m.\end{aligned}$$

Comme le type $\mathsf{Id}\ \overrightarrow{\alpha : \tau} \stackrel{\text{def}}{=} \Sigma\ \overrightarrow{\mathrm{f} : \phi}$. Tout Σ-type peut s'encoder dans un inductif singleton :

$\texttt{Inductive}\ \mathsf{Id}\ (\overrightarrow{\alpha : \tau}) := \mathrm{mkId} : \forall \overrightarrow{\mathrm{f} : \phi} \to \mathsf{Id}\ \overrightarrow{\alpha}$. On peut alors définir les projections $\overrightarrow{\mathrm{f}}$ par simple "*pattern-matching*" sur un objet de type Id. C'est ainsi que sont encodés les enregistrements et les classes dans Coq.

Il s'ensuit que les instances d'une classe sont des objets du type inductif correspondant à la classe. Nous introduisons une syntaxe "ouverte" (sans parenthèses) pour déclarer des instances :

$$\begin{aligned}&\texttt{Instance}\ \mathrm{id} : \mathsf{Id}\ t_1 \cdots t_n :=\\ &\quad \mathrm{f}_1\ \overrightarrow{x} := b_1\ ;\\ &\quad\ \vdots\\ &\quad \mathrm{f}_m\ \overrightarrow{x} := b_m.\end{aligned}$$

Contrairement à Haskell, il faut ici toujours nommer les instances introduites. Cette déclaration introduit une définition de type $\mathsf{Id}\ \mathrm{t}_1 \cdots \mathrm{t}_n$ dans l'environnement avec le nom id. Les définitions des différents champs doivent évidemment correspondre aux types de la classe Id.

Généralisations

Comme on peut le voir, nous n'imposons aucune restriction sur le contexte d'une classe ou le type des méthodes. Il est donc possible d'indexer une classe par plusieurs paramètres qui peuvent être aussi bien des types que des valeurs et dépendre les unes des autres. Par exemple on peut définir une classe EqDec indexée par une relation d'égalité logique *equiv* que la méthode equals va décider :

$$\begin{aligned}&\texttt{Class}\ \mathsf{EqDec}\ (\alpha : \texttt{Type})\ (\mathit{equiv} : \mathrm{relation}\ \alpha) :=\\ &\quad \mathrm{equals} : \alpha \to \alpha \to \mathsf{bool}\\ &\quad \mathrm{equals\_dec} : \forall\ x\ y : \alpha,\ \mathit{equiv}\ x\ y \leftrightarrow \mathrm{equals}\ x\ y = \mathsf{true}\end{aligned}$$

Il est ainsi possible de requérir la décidabilité de l'égalité de Leibniz (eq) sur un type à l'aide d'une contrainte de classe :

```
Definition neq α (eqα : EqDec α eq) (x y : α) : bool := ...
```

Il est aussi possible de donner des contextes arbitraires aux instances, à la gauche du :, pour créer des instances paramétriques. Par exemple, pour définir une instance de l'égalité de Leibniz sur les paires on a besoin d'instances sur chaque composante :

```
Instance prod_eq_EqDec (α β : Type) (eqa : EqDec α eq)
  (eqb : EqDec β eq) : EqDec (α × β) eq := ...
```

7.1.2 Arguments implicites

Lorsque l'utilisateur entre une expression utilisant des méthodes de classes, il n'a pas à indiquer à quelle instance précise il se réfère : c'est l'essence de la surcharge. Cependant, dans le terme Coq final, il est inévitable de faire apparaître quelle instance a été utilisée lors de chaque appel. Pour permettre à l'utilisateur cette omission on généralise le système d'arguments implicites.

Chaque méthode de classe est implémentée par une projection du type record correspondant. Ces projections ont des types de la forme suivante :

$$\mathrm{f}_i : \forall\ \overrightarrow{\alpha : \tau},\ \forall\ id : \mathsf{Id}\ \overrightarrow{\alpha},\ \phi_i[\ \overrightarrow{\mathrm{f}\ \overrightarrow{\alpha}\ id}/\mathrm{f}\ ]$$

Chaque projection prend en paramètre les paramètres de la classe et une implémentation puis retourne un objet du type de la projection, modulo substitution des dépendances aux autres champs par les objets de la même implémentation.

On peut donc simplement rendre les paramètres et l'implémentation de la classe implicites de façon à ce que l'utilisateur n'ait pas à les indiquer. Comme les paramètres de la classe sont utilisés dans le type des méthodes, on se retrouvera après unification et simplification des contraintes de typage avec des contraintes de la forme $\mathsf{Id}\ \overrightarrow{t}$. Par exemple, pour la définition suivante :

```
Definition neq α (eqα : EqDec α eq) (x y : α) : bool :=
  negb (equals x y).
```

Après typage et unification, il reste une contrainte non résolue correspondant à l'appel "equals x y" :

$$\alpha : \mathtt{Type};\ eqa : \mathsf{EqDec}\ \alpha\ \mathrm{eq};\ x,\ y : \alpha \vdash \mathsf{EqDec}\ \alpha\ ?_{(\mathrm{relation}\ \alpha)}$$

Cette contrainte ne peut être résolue par unification, mais elle peut-être résolue par une recherche d'instance.

7.1.3 Recherche d'instances

La recherche d'instance résout un problème de satisfaction de contraintes de la forme $\Gamma \vdash \mathsf{Id}\ \overrightarrow{t}$ correspondant aux instances non résolues après typage. Pour cela elle utilise une base de lemmes de la forme $\forall\Gamma,\ \mathsf{Id}\ \overrightarrow{t}$ qu'elle tente d'appliquer jusqu'à obtenir une preuve ou qu'il n'y ait plus de lemme applicable. C'est une recherche exhaustive, à la manière de la tactique `eauto` de Coq, et qui prend en compte le contexte local. La recherche réussit si et seulement si toutes les contraintes sont satisfaites.

Pour terminer l'exemple précédent, on peut appliquer l'hypothèse *eqa* pour obtenir une implémentation de EqDec α eq, qui est une solution de EqDec α ?. Une fois les instances de toutes les contraintes substituées dans le terme, on peut le donner au noyau pour le valider et ajouter la définition dans l'environnement.

Notons que l'on peut toujours observer le résultat de la résolution à l'aide des commandes interactives de Coq et qu'on peut spécifier les paramètres implicites d'une méthode à l'aide de la notation (arg := t).

7.2 Raffinements

Ceci conclut la présentation du mécanisme de base : les classes sont des enregistrements dépendants, les méthodes des projections avec arguments implicites et la recherche d'instance une forme de recherche de preuve. Nous allons maintenant nous intéresser à la couche supérieure, "syntaxique" du système.

7.2.1 Quantification implicite

Le premier souci rencontré avec le système de base est sa verbosité. En effet, si l'on a bien implanté une forme de surcharge des méthodes permettant d'omettre l'implémentation d'une classe à chaque appel, il reste relativement lourd de paramétrer nos fonctions par des classes.

La vision des contraintes de classes comme une forme de polymorphisme borné nous incite à utiliser les contraintes de classes comme des lieurs pour les types contraints. Ainsi il semble naturel de vouloir écrire :

```
Definition neq [ EqDec α eq ] (x y : α) : bool := ···
```

De plus comme l'on connaît statiquement les types des paramètres d'une classe, il n'est jamais nécessaire de les spécifier. Ici, α sera automatiquement compris comme un objet de type Type et on vérifiera qu'eq est bien une relation sur α.

Les crochets [ ] indiquent donc un changement d'interprétation des lieurs qui réalise une quantification implicite sur les variables libres. Il est toujours possible de spécifier un nom pour l'implémentation de la classe ou de ne pas utiliser la quantification implicite :

```
Definition neq α [ eqa : EqDec α eq ] (x y : α) : bool := ···
```

Il est cependant à noter que les lieurs implicites se traduisent automatiquement vers des arguments implicites dans le terme final, ce qu'il faut spécifier avec la syntaxe {id} pour les lieurs ordinaires. Les définitions paramétrées par des classes doivent utiliser des arguments implicites pour leurs arguments de type classe afin de permettre au mécanisme de résolution d'entrer en jeu.

La quantification implicite est plus habituelle dans les mathématiques informelles mais aussi dans les langages de programmation logique, par exemple les systèmes basés sur LF (Pfenning 1991). Les classes apportent un peu de cette tradition de programmation logique en donnant la possibilité de faire de la recherche de preuve au typage, lorsque l'on voit les classes comme des prédicats. Mais la quantification implicite est aussi utile pour implémenter les superclasses comme nous allons le voir dans la section suivante.

7.2.2 Sous- et superstructures

Dès lors que l'on fournit une construction comme les classes qui permettent de structurer un développement en regroupant des fonctionnalités communes au sein d'une unique entité, se pose la question de la composabilité des structures. On a bien vite besoin de composer une classe à partir d'une autre et de créer ainsi une hiérarchie. C'est ce que permettent les superclasses évoquées plus haut (§6.1.4).

Superclasses

Les superclasses permettent d'étendre les fonctionnalités d'une combinaison de classes dans une nouvelle classe. Au moment de la déclaration, on va donc déclarer un ensemble de superclasses qu'une nouvelle classe va *étendre*. On pourra alors utiliser les méthodes des superclasses partout où l'on peut utiliser les méthodes de la nouvelle classe. C'est la même forme d'héritage que l'on rencontre dans les langages objets, où les sous-classes héritent des méthodes déclarées dans leurs superclasses.

Pour implémenter ce mécanisme en Coq, il faut que chaque instance de sous-classe comprenne une implémentation de chacune des superclasses. Il y a alors deux possibilités : les instances peuvent être implantées à l'aide de paramètres ou de champs de l'enregistrement représentant la classe. Ce choix n'est pas anodin et nous allons voir comment il détermine les possibilités de spécification des hiérarchies de classe.

Superclasses comme champs Si l'on veut faire de chaque superclasse un champ de l'enregistrement, alors on s'interdit la possibilité de spécifier le partage de sous-structures aisément. Supposons l'ensemble de classes suivantes qui représentent un sétoïde et une preuve de la décidabilité de la relation d'équivalence d'un sétoïde.

```
Class Setoid (α : Type) :=
  equiv : relation α;
  equiv_equiv : equivalence equiv.
Class [ setoid : Setoid α ] ⇒ SetoidDec :=
  equiv_dec : Π x y, { equiv x y } + { ¬ equiv x y }.
```

Le fait de faire de l'implémentation de Setoid un champ rendrait la spécification du partage difficile. Supposons qu'on ait commencé un développement dans un contexte $setoid\alpha$: Setoid α. On veut alors écrire un lemme qui nécessite la décidabilité de la relation d'équivalence sur ce sétoïde et on aimerait donc utiliser la classe SetoidDec pour l'indiquer. Or la seule façon qu'on aurait de spécifier une relation entre une instance de SetoidDec et une instance de Setoid existante serait d'utiliser l'égalité sur la projection setoid de SetoidDec, comme suit :

```
Lemma foo [ sdec : SetoidDec α ] : setoid sdec = setoidα → ...
```

L'utilisation de l'égalité propositionnelle pour spécifier le partage est cependant très coûteuse : elle est difficile à manipuler, tant dans l'écriture des énoncés que pendant les preuves, car elle pollue le raisonnement avec des étapes administratives. Ceci milite pour un encodage des superclasses comme paramètres des classes, qui donne la possibilité de spécifier le partage à l'aide de la quantification universelle de Coq. Pour l'exemple précédent, on peut instancier explicitement le paramètre *set* de la classe SetoidDec avec l'implémentation désirée :

```
Lemma equiv_dec_foo (sdec : SetoidDec setoidα) ...
```

L'utilisation de paramètres par opposition aux champs est bien connue dans la littérature sur les structures et la modularité dans les langages de programmation, sous le nom de "*Pebble-style sharing*". Ce nom est dû au langage Pebble (Lampson et Burstall 1988), un noyau de langage avec type dépendants développé dans les années 80 pour étudier les systèmes de modules et les types abstraits. La distinction a aussi été étudiée en lien avec les développement mathématiques dans les assistants de preuve (Pollack 2000).

Implémentation La présentation par paramêtres est la représentation la plus naturelle des superclasses, puisqu'elle permet aux paramètres de dépendre des superclasses et de

façon générale de correctement spécifier les hiérarchies. Formellement, on interprète la déclaration de classe :

$$\texttt{Class}\ [\ \overrightarrow{s : \mathsf{S}\,\overrightarrow{\sigma}}\ ] \Rightarrow \mathsf{Id}\ \overrightarrow{\alpha : \tau} := \overrightarrow{f : \phi}.$$

comme un enregistrement paramétré équivalent à la somme dépendante :

$$\mathsf{Id} \stackrel{\text{def}}{=} \lambda \overrightarrow{\sigma}, \overrightarrow{s : \mathsf{S}\,\overrightarrow{\sigma}}, \lambda \overrightarrow{\alpha : \tau}, \Sigma \overrightarrow{f : \phi}.$$

Il s'ensuit que les déclarations d'instance doivent fournir une implémentation de chaque superclasse avec ses paramètres, puis une instance des paramètres de la classe elle même et finalement une implémentation des champs. Le contexte de superclasses (avant la flèche $\Rightarrow$) peut en fait être généralisé à un contexte arbitraire. On considère alors que les classes quantifiées dans ce contexte sont des superclasses et que tous les autres paramètres sont des paramètres explicites hérités par la nouvelle classe.

Nous avons vu que la paramétrisation permet d'éviter l'utilisation de l'égalité propositionnelle pour spécifier le partage, mais c'est au prix d'une plus grande verbosité. En effet, les paramètres d'une classe doivent être abstraits et ré-appliqués constamment. Lorsqu'on paramètre une classe par une superclasse, il faut donc quantifier aussi sur les paramètres lui correspondant. Prenons un exemple :

```
Class Monad (η : Type → Type) :=
  unit : ∀ α, α → η α ;
  bind : ∀ α β, η α → (α → η β) → η β.
Class [ mon : Monad η ] ⇒ MonadZero :=
  mzero : ∀ α, η α.
```

Dans cette hiérarchie, la classe MonadZero n'ajoute pas de paramètres à la classe de base Monad. Cependant, si l'on veut écrire une définition sur une MonadZero η, il faut expliciter quelle est la structure de Monad utilisée. Nous utilisons ici la quantification implicite pour automatiquement quantifier sur une implémentation de Monad η lorsqu'elle n'est pas spécifiée. Un lieur [MonadZero η] s'interprète donc en une suite de lieurs implicites "∀{η} {mon : Monad η} {_ : MonadZero mon}". Lorsqu'on quantifie sur une classe ayant des superclasses, on doit spécifier dans l'ordre chacun de ses paramètres explicites.

La verbosité à l'abstraction est ainsi contenue grâce à ce mécanisme de généralisation implicite et le système d'arguments implicites et de recherche d'instances permet de conserver cette concision lorsqu'on applique les définitions paramétrées.

Sous-structures

Si l'utilisation de paramètres peut s'avérer la plus judicieuse pour implémenter les superclasses, il est tout aussi utile de composer des classes par agrégation. Cela correspond à avoir une classe composée de plusieurs champs représentant ses composantes, qui sont eux même des classes.

On va l'utiliser typiquement dans les cas de classes vues comme des prédicats. Supposons données les classes suivantes :

```
Class Reflexive {A} (R : relation A) :=
  reflexivity : ∀ x, R x x.

Class Symmetric {A} (R : relation A) :=
  symmetry : ∀ {x y}, R x y → R⁻¹ x y.

Class Transitive {A} (R : relation A) :=
  transitivity : ∀ {x y z}, R x y → R y z → R x z.
```

Une instance de Reflexive va correspondre à une preuve de réflexivité, etc... On peut alors construire une classe représentant les relations d'équivalence à partir de celles-ci :

```
Class Equivalence (carrier : Type) (equiv : relation carrier) : Prop :=
  Equivalence_Reflexive :> Reflexive equiv ;
  Equivalence_Symmetric :> Symmetric equiv ;
  Equivalence_Transitive :> Transitive equiv.
```

L'enregistrement correspondant, paramétré par un type et une relation aura trois champs correspondants aux preuves de réflexivité, symétrie et transitivité de cette relation. On a utilisé la syntaxe `:>` pour spécifier que chaque champ correspond à une sous-structure. Cela va donner lieu à la déclaration de trois instances qui réalisent cette relation : les projections de la nouvelle classe. On pourra dès lors utiliser les méthodes des classes Reflexive, Symmetric et Transitive dans les contextes où une instance de la classe Equivalence est disponible. L'algorithme de recherche d'instances appliquera les projections automatiquement.

7.2.3 Classes singletons

Les classes singleton sont traitées de façon particulière dans notre implémentation. En effet, une classe à une seule méthode est équivalente à un Σ-type à une seule composante, soit un type. On implémente donc les classes singleton par des définitions et leur projection par une fonction identité. Soit la classe :

```
Class Id α⃗ : τ⃗ := f : φ.
```

On définit le type de la classe par $\mathsf{Id} \stackrel{\text{def}}{=} \lambda \overrightarrow{\alpha : \tau}, \phi$ et on code la projection par $\mathsf{f} \stackrel{\text{def}}{=} \lambda \overrightarrow{\alpha : \tau}, \lambda(i : \mathsf{Id}\ \overrightarrow{\alpha}), i$.

L'avantage de ce codage est que les classes singletons sont transparentes, c'est-à-dire qu'un objet de type $\mathsf{Id}\ \overrightarrow{t}$ est *convertible* à sa projection de type $\phi[\overrightarrow{t/\alpha}]$. Par exemple, les objets de la classe Reflexive sont effectivement des preuves de réflexivité et non pas des objets d'un type inductif à un constructeur contenant une preuve de réflexivité. On peut ainsi aisément faire d'une définition de propriété (telle que l'associativité) une classe sans avoir à modifier les habitants existants (les preuves d'associativité déjà écrites). On pourrait rendre facilement visibles comme instances des preuves déjà écrites de cette manière.

Conclusion

Ceci conclut ce chapitre sur l'implémentation des classes de types en Coq et les différents raffinements apportés à la traduction du système existant en Haskell. Un chapitre du manuel de référence résume les commandes disponibles pour déclarer classes et instances (Sozeau 2008b). Nous allons maintenant développer des exemples d'utilisation.

Chapitre 8

Exemples

Sommaire

Nous allons maintenant présenter deux exemples d'utilisation des classes de types, le premier montrant la facilité de programmation apportée par la surcharge et le second l'utilisation des classes pour la preuve dans les développements mathématiques.

8.1 Le prélude Haskell

```
Require Import Coq.Lists.List Program Setoid.
```

Nous allons développer un exemple d'utilisation des classes de type pour programmer avec des monades sur la catégorie des types de Coq. On utilisera pour cela un certain nombre de définitions existantes dans la bibliothèque de Program, ainsi que les listes standards et la tactique de réécriture sur les setoïdes.

Foncteurs

On introduit tout d'abord la notion de foncteur (covariant) : c'est un constructeur de type muni d'une opération fmap qui permet d'appliquer une fonction à chaque élément contenu dans un objet de type $F\ \alpha$, si l'on voit le foncteur F comme un conteneur.

```
Class Functor (F : Type → Type) :=
  fmap : ∀{α β : Type}, (α → β) → F α → F β;
  fmap_id : ∀α, fmap (id (A :=α)) === id;
  fmap_compose : ∀(α β δ : Type) (f : β → δ) (g : α → β),
    fmap (f ∘ g) === fmap f ∘ fmap g.
```

On a additionnellement des propriétés sur l'application du foncteur : elle préserve l'identité et la composition de fonctions. La première loi indique qu'appliquer l'identité à chaque élément revient à ne rien faire. La seconde montre que l'on peut fusionner deux appels à fmap consécutifs en un seul.

Notons l'utilisation de l'opérateur surchargé === de la classe Equivalence qui permet de faire référence à l'équivalence par défaut sur un type donné. Ici on compare des fonctions

et on n'a pas de relation sur $F\ \alpha$, ce sera donc l'égalité de Leibniz point-à-point qui sera utilisée. L'instance correspondante est définie dans la librairie *Setoid*.

```
Definition Id (A : Type) := A.
```

On instancie la classe sur le foncteur identité. Ce foncteur peut être vu comme un conteneur à une unique place, l'opération d'application est donc triviale a implémenter : on applique f à l'unique élément.

```
Program Instance id_Functor : Functor Id :=
  fmap α β f := f.
```

On utilise ici une variante de la déclaration d'instances utilisant Program. Les champs non instanciés deviennent des obligations et une fois celles-ci déchargées on peut déclarer l'instance. Les preuves sont déchargées automatiquement ici, fmap étant simplement l'identité. Cette variante permet de mieux contrôler l'opacité des différents champs d'une instance, et encore une fois de séparer élégamment les preuves des programmes.

Une instance plus intéressante est celle des listes polymorphes. L'application est la fonction map qui applique une fonction f à chaque élément.

```
Program Instance list_Functor : Functor (λA, list A) :=
  fmap α β f l := map f l.
```

Les deux preuves nécessitent une induction sur la liste. Par exemple la première demande de montrer $\forall\ x, \text{map id}\ x = x$; on la décharge aisément :

```
Next Obligation.
  induction a ; simpl ; auto. rewrite IHa. reflexivity.
Qed.
```

Le foncteur d'état est aussi une monade au contenu calculatoire évident. C'est un foncteur à une place de type α et qui manipule un état de type s. La monade correspondante permet de composer des actions modifiant l'état interne.

```
Definition state (s : Type) (α : Type) := s → α × s.

Program Instance state_Functor : Functor (state s) :=
  fmap α β f m := λs, let (e, s') := m s in (f e, s').
```

L'application exécute la première action, modifie l'objet stocké avec f, et change l'état avec le nouvel état produit par l'action.

Monades

Une monade est un foncteur muni d'une opération unit qui permet d'injecter un élément dans la monade, et une fonction bind pour composer deux objets de la monade. Dans le cas des conteneurs, l'opération de composition va permettre d'appliquer à chaque élément du premier objet une fonction créant un autre élément de la monade pour combiner tout ces résultats dans un objet final.

```
Class [ F : Functor η ] ⇒ Monad :=
  unit : ∀{α}, α → η α ;
  bind : ∀{α β}, η α → (α → η β) → η β ;
  bind_fmap : ∀{α β} (f : α → β) (m : η α),
    fmap f m === bind m (unit ∘ f) ;
  bind_unit_left : ∀{α β} (a : α) (f : α → η β),
    bind (unit a) f === f a ;
  bind_unit_right : ∀{α} (a : η α), bind a unit === a ;
  bind_assoc : ∀{α β δ} (a : η α) (f : α → η β) (g : β → η δ),
```

```
  bind a (λa : α, bind (f a) g) === bind (bind a f) g.
```

Les lois expriment que l'injection est une sorte d'élément neutre pour la composition et que la composition est associative. Prenons l'exemple de la monade d'état :

```
Program Instance state_monad :!Monad (state s) :=
  unit α a := λs, (a, s);
  bind α β m f := λs, let (a, s') := m s in f a s'.
```

On a indiqué avec ! que l'on voulait spéficier les paramètres de la classe plutôt que d'utiliser le mécanisme habituel de traitement de l'application avec arguments implicites de Coq. On aurait aussi pu écrire Monad (@state_Functor s) pour faire directement réference à l'implémentation du foncteur.

L'opération d'injection met un objet dans la monade sans modifier l'état, tandis que la composition fait passer l'état d'une action à une autre.

On introduit des notations usuelles de Haskell pour les monades.

```
Notation "T >>= U" := (bind T U) (at level 55, right associativity).
Notation "T >> U" := (bind T (λ_, U)) (at level 55, right associativity).
Notation "x ← T ;; E" := (bind T (λx, E))
  (at level 60, T at next level, right associativity).
Notation "'return' t" := (unit t) (at level 20).
```

On va maintenant définir un certain nombre d'opérations standard sur toutes les monades en utilisant la surcharge.

```
Section Monad_Defs.
```

La déclaration d'un contexte au sein d'une section indique que toutes les définitions suivantes sont des méthodes surchargées pour le contexte donné. Les crochets indiquent la généralisation implicite comme précédemment.

```
  Context [ mon : Monad η ].
```

L'opérateur suivant donne une autre caractérisation de la monade, qui est plus souvent utilisée dans un cadre catégorique.

```
  Definition join {α} (x : η (η α)) : η α := x >>= id.
```

On peut montrer que les lois usuelles pour join sont respectées. La suppression d'un niveau monadique effectuée après l'ajout d'un niveau trivial *via* unit est l'identité. La preuve utilise les lemmes génériques contenus dans la classe Monad. L'unification permet de déduire sur quelle instance ils sont utilisés.

```
  Lemma join_fmap_unit : ∀{α}, join ∘ fmap unit === (@id (η α)).
  Proof.
    intros. unfold compose, join. intros x.
    rewrite bind_fmap. unfold compose.
    setoid_rewrite ← (do_return_eta _ x) at 2.
    rewrite ← bind_assoc.
    f_equal. extensionality a.
    rewrite bind_unit_left. reflexivity.
  Qed.

  Lemma join_unit : ∀{α}, join ∘ unit === (@id (η α)).
  Proof.
    intros. unfold compose, id, join.
    intros m. setoid_rewrite bind_unit_left. reflexivity.
  Qed.
```

On peut "*lifter*" une fonction pure dans la monade avec liftM.

```
Definition liftM {α β} (f : α → β) : η α → η β :=
  λx, a ← x ;; return (f a).
Definition liftM2 {α β δ} (f : α → β → δ) : η α → η β → η δ :=
  λx y, a ← x ;; b ← y ;; return (f a b).
```

Le système de classes est orthogonal aux fonctionalités avancées du calcul des constructions. On peut écrire des fonctions récursives sur des objets inductifs et des fonctions d'ordre supérieur de façon transparente tout en utilisant les classes. Ici on implémente une fonction qui exécute une liste d'actions en séquence de gauche à droite et retourne la liste des résultats.

```
Fixpoint sequence {α} (l : list (η α)) : η (list α) :=
  match l with
    | [] ⇒ return []
    | hd :: tl ⇒
      x ← hd ;;
      r ← sequence tl ;;
      return (x :: r)
  end.
```

Par composition on peut définir l'application point-à-point d'une fonction produisant une action :

```
  Definition mapM {α β} (f : α → η β) : list α → η (list β) :=
    sequence ∘ map f.
End Monad_Defs.
```

On va maintenant écrire un petit programme utilisant les opérations génériques sur la monade d'état de façon transparente. Notre programme va étiqueter les feuilles d'un arbre de gauche à droite, en ordre croissant[1]. On introduit pour cela un type d'arbres binaires :

```
Inductive tree (α : Type) :=
| Leaf : ∀ x : α, tree α
| Node : ∀ (l : tree α) (r : tree α), tree α.

Implicit Arguments Leaf [[α]].
Implicit Arguments Node [[α]].
```

On va travailler dans la monade d'état où l'état sera un compteur.

On implémente d'abord les actions spécifiques à la monade d'état qui permettent de récupérer l'état courant et de le modifier :

```
Program Definition get {α : Type} : state α α := λs, (s, s).
Program Definition put {α : Type} : α → state α () := λx _, ((), x).
```

Enfin puisque nous utilisons cette monade pour implémenter un compteur, on ajoute l'opération spécifique qui permet de l'incrémenter. On utilise les opérations surchargées sur la monade state définie précédemment.

```
Definition incr : state nat () := s ← get ;; put (S s).
```

Munis de ces opérations, il devient trivial d'écrire une fonction d'étiquetage dans la monade d'état qui prend un arbre et renvoie sa version étiquetée. Les monades donnent un style impératif au code purement fonctionel qui ne fait que passer l'état.

```
Fixpoint label {α} (t : tree α) : state nat (tree nat) :=
```

[1] Cet exemple est tiré d'une présentation de Diana Fulger (2007)

```
match t with
  | Leaf x ⇒ n ← get ;; incr ≫ return (Leaf n)
  | Node l r ⇒
    l' ← label l ;; r' ← label r ;; return (Node l' r')
end.
```

Dans la deuxième branche, on pourrait aussi écrire liftM2 Node (*label l*) (*label r*) pour obtenir le même comportement.

La fonction suivante permet d'exécuter une action de la monade d'état sur un état initial donné.

```
Definition run_state {α β} (a : α) (s : state α β) : β := projT1 (s a).
```

On peut alors composer ces deux fonctions pour effectivement calculer un étiquetage.

```
Definition labels {α} (t : tree α) : tree nat := run_state 0 (label t).
```

Ceci clôt notre exemple d'utilisation des classes pour programmer avec la surcharge.

8.2 Une théorie abstraite des ordres

Nous allons maintenant développer une partie de théorie des ordres avec les classes, qui sera utilisée dans le chapitre 9 sur la réécriture généralisée. On va introduire des classes permettant de déclarer qu'une relation a telle ou telle propriété et d'utiliser cette information dans les tactiques.

Définitions standards

On introduit tout d'abord un ensemble de définitions sur les relations. On rappelle qu'une relation R sur un type A est un prédicat de type $A \to A \to \texttt{Prop}$. La relation inverse, notée R^{-1}, s'obtient simplement en inversant l'ordre des arguments :

```
Notation inverse R := (flip (R : relation _) : relation _).
Notation " R ^-1 " := (inverse R) (at level 0) : relation_scope.
Open Local Scope relation_scope.
```

La relation complémentaire est définie classiquement comme la négation de la relation :

```
Definition complement {A} (R : relation A) : relation A :=
  λx y, R x y → False.
```

On peut étendre une relation sur B aux fonctions de A vers B par l'équivalence point-à-point :

```
Definition pointwise_relation {A B : Type} (R : relation B) : relation (A → B) :=
  λf g, ∀ x : A, R (f x) (g x).
```

Propriétés

On introduit maintenant des classes pour les notions usuelles sur les relations, en utilisant des constructions duales si possible. Les définitions pour Reflexive, Symmetric, Transitive et Equivalence ont déjà été données §7.2.2.

```
Class Irreflexive {A} (R : relation A) :=
  irreflexivity :> Reflexive (complement R).
Class Asymmetric {A} (R : relation A) :=
  asymmetry : ∀ {x y}, R x y → (complement R^-1) x y.
```

Lorsqu'on a besoin d'une preuve de réflexivité sur une relation arbitraire, il devient possible d'appliquer la méthode reflexivity et la recherche d'instance se chargera de trouver la preuve correspondante. Il est à noter que toutes ces classes sont indexées par des *valeurs*. C'est l'apport essentiel des types dépendants au système de classes qui le rend beaucoup plus puissant. On peut grâce à ça associer des propriétés à des objets sans les lier ni syntaxiquement, ni par leur type comme l'on a fait avec les types sous-ensembles.

Instances standard

On peut déjà définir par dualité les instances pour les relations inverses. Toutes les propriétés sont préservées par l'inverse.

```
Instance flip_Reflexive [ Reflexive A R ] : Reflexive R^-1 :=
  reflexivity := reflexivity (R :=R).
Instance flip_Irreflexive [ Irreflexive A R ] : Irreflexive R^-1 :=
  irreflexivity := irreflexivity (R :=R).
Instance flip_Symmetric [ Symmetric A R ] : Symmetric R^-1 :=
  symmetry x y H := symmetry (R :=R) H.
```

Pour les suivantes, on fait la preuve manuellement.

```
Instance flip_Asymmetric [ Asymmetric A R ] : Asymmetric R^-1.
Instance flip_Transitive [ Transitive A R ] : Transitive R^-1.
```

Pour la complémentation, seule la symétrie est préservée.

```
Instance refl_compl_irr [ r : Reflexive A R ] : Irreflexive (complement R).
Instance compl_symm [ S : Symmetric A R ] : Symmetric (complement R).
```

On en vient aux définitions sur les connecteurs usuels. On définit ces instances avec Program qui applique une tactique de simplification des relations puis la procédure de décision `firstorder`. L'implication est réflexive et transitive :

```
Program Instance impl_Reflexive : Reflexive impl.
Program Instance impl_Transitive : Transitive impl.
```

L'équivalence logique et l'égalité de leibniz forment des équivalences :

```
Program Instance iff_Reflexive : Reflexive iff.
Program Instance iff_Symmetric : Symmetric iff.
Program Instance iff_Transitive : Transitive iff.
Program Instance eq_Reflexive : Reflexive (@eq A).
Program Instance eq_Symmetric : Symmetric (@eq A).
Program Instance eq_Transitive : Transitive (@eq A).
```

L'intérêt d'énoncer ces propriétés *via* des instances et non simplement sous forme de lemmes est qu'elles sont maintenant facilement accessibles *via* des opérateurs surchargés. Étant donné une relation R arbitraire, on peux tenter de retrouver une preuve de réflexivité instantanément avec `reflexivity (R :=R)`. Les instances paramétrées comme flip_Symmetric utilisent la généralisation de la simple association d'un terme à un type par une vraie recherche de preuve.

Composition de propriétés

On peut composer ces propriétés pour obtenir les structures usuelles de la théorie des ordres. Pour commencer, un préordre est une relation réflexive et transitive :

```
Class PreOrder A (R : relation A) : Prop :=
```

```
  PreOrder_Reflexive :> Reflexive R;
  PreOrder_Transitive :> Transitive R.
```

Une relation d'équivalence partielle est symétrique et transitive :

```
Class PER (carrier : Type) (pequiv : relation carrier) : Prop :=
  PER_Symmetric :> Symmetric pequiv;
  PER_Transitive :> Transitive pequiv.
```

On a déjà défini l'Equivalence. On peut maintenant définir l'antisymétrie par rapport à une relation d'équivalence :

```
Class {(equ : Equivalence A eqA)} ⇒ Antisymmetric (R : relation A) :=
  antisymmetry : ∀ x y, R x y → R^-1 x y → eqA x y.
```

Elle est préservée par l'inverse. Dans l'exemple suivant, on utilise le modifieur "!" qui bascule le mode d'internalisation de Antisymmetric eq R pour utiliser le traitement habituel des arguments implicites. Dans ce cas, Antisymmetric attend une instance d'Equivalence et une relation en arguments, plutôt qu'un type et deux relations sur ce type (les paramètres d'Antisymmetric).

```
Program Instance flip_antisym [ eq : Equivalence A eqA, ! Antisymmetric eq R ] :
  Antisymmetric eq R^-1.
```

On peut créer trivialement les structures composites à partir des instances définies précédemment pour l'équivalence logique et l'égalité.

```
Program Instance iff_equivalence : Equivalence Prop iff.
Program Instance eq_equivalence : Equivalence A (@eq A) | 10.
```

On utilise ici un raffinement de l'algorithme de recherche d'instances qui permet de donner des priorités aux lemmes applicables (voir §9.2.3). Plus le niveau donné est grand, plus la priorité est basse. L'égalité étant une relation d'équivalence triviale sur tous les types, on ne veut la sélectionner lors d'une recherche d'instance Equivalence A _ que si aucune autre équivalence n'est disponible dans l'environnement.

8.2.1 Prédicats et relations.

On développe maintenant un ensemble de définitions sur les prédicats d'arité arbitraire qui pourront être utilisés pour dériver des instances sur les ensembles (vus comme des prédicats $A \to$ Prop) et les relations binaires homogènes ($A \to A \to$ Prop).

```
Require Import List Program.
Open Local Scope list_scope.
```

Pour construire des définitions génériques sur les prédicats arbitraires, on utilise un codage compact des arités par une liste de types pour les arguments plus un type de retour. On va donc programmer avec le type dépendant arrows l r dans la suite et créer nos fonctions par récurrence sur la liste d'arguments l.

```
Fixpoint arrows (l : list Type) (r : Type) : Type :=
  match l with
    | nil ⇒ r
    | A :: l' ⇒ A → arrows l' r
  end.
```

On peut alors définir des abréviations pour les types d'opérations usuels.

```
Definition unary_operation A := arrows [A] A.
Definition binary_operation A := arrows [A;A] A.
```

```
Definition ternary_operation A := arrows [A;A;A] A.
```

Les prédicats n-aires ont pour type de retour `Prop`.

```
Notation predicate l := (arrows l Prop).
```

Les prédicats unaires, ou ensembles.

```
Definition unary_predicate A := predicate [A].
```

Le type des relations binaires, équivalent à relation.

```
Definition binary_relation A := predicate [A;A].
```

Combinateurs

On définit deux combinateurs pour créer des opérations (repectivement des relations) sur des fonctions (resp. prédicats) à partir d'opérations (resp. relations) binaires en les appliquant point-à-point.

On définit l'extension point-à-point d'une opération binaire sur T aux fonctions ayant T pour codomaine.

```
Fixpoint pointwise_extension {T : Type} (op : binary_operation T)
  (l : list Type) : binary_operation (arrows l T) :=
  match l with
    | nil ⇒ λR R', op R R'
    | A :: tl ⇒ λR R', λx, pointwise_extension op tl (R x) (R' x)
  end.
```

On étend point-à-point une relation sur `Prop` vers une relation sur des prédicats.

```
Fixpoint pointwise_lifting (op : binary_relation Prop) (l : list Type) : binary_relation
(predicate l) :=
    match l with
      | nil ⇒ λR R', op R R'
      | A :: tl ⇒ λR R', ∀x, pointwise_lifting op tl (R x) (R' x)
    end.
```

On définit alors l'équivalence de deux prédicats n-aires comme le lifting de l'équivalence logique, et de même pour l'implication.

```
Definition predicate_equivalence {l : list Type} := pointwise_lifting iff l.
Definition predicate_implication {l : list Type} := pointwise_lifting impl l.
```

On étend la conjonction et la disjonction aux prédicats. Il est à noter que ce sont des opérations et non pas des relations comme précédemment. L'intersection crée un nouveau prédicat à partir de deux prédicats existants.

```
Definition predicate_intersection {l} := pointwise_extension and l.
Definition predicate_union {l} := pointwise_extension or l.
```

Finalement, on introduit un combinateur pour les prédicats constants qu'on instancie pour les prédicats toujours vrais et toujours faux.

```
Fixpoint const_predicate {l : list Type} (P : Prop) : predicate l :=
  match l with
    | nil ⇒ P
    | A :: tl ⇒ λ_, @const_predicate tl P
  end.

Definition true_predicate {l} := @const_predicate l True.
Definition false_predicate {l} := @const_predicate l False.
```

8.2.2 Instances

À partir de ces définitions, on peut construire les structures algébriques classiques sur les prédicats, de façon générique sur leur arité. On montre que l'équivalence de prédicats est une équivalence et l'implication un préordre.

```
Program Instance pred_equiv_equiv {l} : Equivalence (predicate l) predicate_equivalence.
Program Instance pred_impl_preorder {l} : PreOrder (predicate l) predicate_implication.
```

Relations binaires On spécialise les opérations aux relations binaires (homogènes). Il est à noter que Coq ne peut pas inférer la liste correspondant à l'arité automatiquement, il faudrait pour cela inverser la fonction bijective arrows.

```
Definition relation_equivalence {A :Type} : relation (relation A) :=
  @predicate_equivalence [A ;A].
Definition relation_implication {A} : relation (relation A) :=
  @predicate_implication [_ ;_].
```

On introduit des notations pour abréger ces deux opérateurs.

```
Infix "<==>" := relation_equivalence
  (at level 95, no associativity) : predicate_scope.
Infix "==>" := relation_implication
  (at level 70, right associativity) : predicate_scope.
```

On crée une classe spécifique pour l'inclusion de relations, utilisée notamment par la tactique de réécriture. Une relation est incluse dans une autre si elle l'implique pour toute paire d'éléments.

```
Class subrelation {A} (R R' : relation A) : Prop :=
  is_subrelation : R ==> R'.
```

On spécialise aussi les déclarations d'instance génériques qui ne peuvent pas s'appliquer lorsqu'on utilise le type relation à cause du même problème d'inférence.

```
Instance rel_equiv_equiv : Equivalence (relation A) relation_equivalence.
Instance rel_impl_preorder : PreOrder (relation A) subrelation.
```

On spécialise aussi la conjonction et la disjonction sur les relations, qu'on note /~\ et \~/.

Ordre Partiel

Finalement on déclare la classe des ordres partiels vis-à-vis d'une relation d'équivalence donnée. La caractérisation donnée ici est quelque peu différente de la définition usuelle. Elle permet de dériver non seulement l'antisymétrie mais aussi un lemme de compatibilité entre les deux relations.

```
Class [ equ : Equivalence A eqA, pro : PreOrder A R ] => PartialOrder :=
  partial_order_equivalence : eqA <==> (R /~\ R^-1).
```

On dérive aisément l'antisymétrie de l'ordre partiel vis-à-vis de l'équivalence.

```
Instance po_antisym [ PartialOrder A eqA R ] :! Antisymmetric A eqA R.
```

On peut instancier l'ordre partiel sur les prédicats et le spécialiser aux relations.

```
Program Instance predicate_po {l} :
  ! PartialOrder (predicate l) predicate_equivalence predicate_implication.
```

Conclusion

Cette bibliothèque peut être réutilisée pour dériver les constructions usuelles sur les sous-ensembles, mais aussi les relations hétérogènes. On pourrait encore l'étendre avec des notions plus avancées de la théorie des ordres : les différentes variétés d'ordres (stricts, partiels, totaux), les semi-treillis, treillis etc... Nous avons entamé un tel développement (Sozeau 2008d) en même temps que l'implémentation pour la tester sur des exemples plus conséquents. Nous sommes vite arrivés à la conclusion qu'il était nécessaire d'avoir un système de réécriture généralisée puissant et bien intégré aux classes pour arriver à cette fin.

Chapitre 9

Réécriture généralisée

Sommaire

Nous allons maintenant illustrer l'utilisation des classes comme structure pour interfacer le système de tactiques et les développement utilisateurs. Nous allons développer un algorithme de réécriture généralisée, qui permet de faire du raisonnement équationnel sur des relations arbitraires, étant données des preuves de congruence vis-à-vis de ces relations sur les symboles de l'utilisateur.

9.1 Introduction et état de l'art

La réécriture est reconnue comme l'un des formalismes essentiels de la preuve automatique. C'est aussi un outil central de la preuve assistée, depuis ses débuts (Paulson 1983) jusqu'à aujourd'hui (Gonthier et Werner 2005). Gonthier évalue que la réécriture est utilisée pour plus du tiers des étapes de raisonnement dans les preuves formelles qu'il développe.

Notre problème de départ est simple : dans les assistants de preuve, la réécriture n'est disponible que sur la relation d'égalité de base du système (qu'elle soit intentionnelle ou extensionnelle), *via* la caractérisation de Leibniz. En effet, l'égalité propositionelle respecte l'axiome de substitution : $\forall x\ y\ P, x = y \to P\ x \to P\ y$ qui permet de remplacer x par y dans n'importe quel contexte si $x = y$.

Or on voudrait aussi pouvoir traiter d'autres relations comme des relations de réécriture, par exemple l'équivalence logique $\leftrightarrow$, l'égalité ensembliste définie comme double inclusion ou toute relation d'équivalence donnée par l'utilisateur. Il devient ainsi possible de réécrire par une hypothèse de type $x \leftrightarrow y$ dans un but de la forme $x \wedge z \to x$ pour obtenir $y \wedge z \to y$.

Seulement, il n'est pas possible de dériver un lemme de substitution aussi général pour ces relations, car certains contextes sont susceptibles de différencier deux éléments d'une même classe d'équivalence. Soit la relation d'équivalence sur les entiers suivante : $n \equiv m \stackrel{\text{def}}{=} n \bmod 2 = m \bmod 2$, la proposition $\lambda x, x = 2$ n'est pas substitutive pour cette relation, puisque si $2 \equiv 4$, on n'a pas $2 = 4$. On va donc devoir montrer pour chaque opération quelles sont les relations pour lesquelles elle est substitutive (ce qu'on appellera une signature de morphisme).

Une fois les relations d'équivalence et les lemmes de substitution démontrés, il devient possible d'automatiser les preuves nécessaires à chaque réecriture. Cela revient à une recherche des lemmes à appliquer et leur composition, qu'il serait très fastidieux de faire manuellement.

9.1.1 État de l'art

LCF

La généralisation de la réécriture n'est pas nouvelle, les premières implémentations remontent aux systèmes Boyer-Moore et LCF et en particulier à Lawrence Paulson (Paulson 1983) dans Cambdrige LCF qui présente un ensemble de tactiques d'ordre supérieur permettant de construire des tactiques de réécriture complexes. Il étend en particulier la réécriture des termes aux formules, permettant de faire des raisonnements logiques par ce qu'il appelle des "conversions" entre formules équivalentes. La technique est basée sur un ensemble de tactiques, certes modulaires mais qu'il faut instancier manuellement si l'on veut introduire de nouvelles relations et de nouveaux opérateurs congruents. Dans ces systèmes, on cherche surtout à combiner un ensemble de règles de réécriture de façon flexible pour faire de la simplification, et non pas de réécrire modulo congruences.

NuPRL

L'idée est généralisée aux relations arbitraires par Basin (1994) dans NuPRL qui cette fois s'intéresse plus à la combinaison de preuves de congruences pour une réécriture donnée. Il suppose donné un ensemble de lemmes prouvant la congruence des opérations avec les relations, qui seront utilisés lors d'une tentative de réécriture pour prouver celle-ci correcte. Dans ce cadre, il devient possible de donner plusieurs signatures à une opération, par exemple, pour l'addition :

$+ : \{x = x' \to y = y' \to x + y = x' + y'\}$
$+ : \{x < x' \to y \leq y' \to x + y < x' + y'\}$
$+ : \{x \leq x' \to y < y' \to x + y < x' + y'\}$
$+ : \{x \leq x' \to y = y' \to x + y \leq x' + y'\}$
$+ : \{x = x' \to y \leq y' \to x + y \leq x' + y'\}$

On déclare que l'addition est une congruence pour l'égalité, mais aussi qu'elle est monotone pour $<$ et $\leq$. L'algorithme implémenté doit choisir une de ces preuves, et peut donc échouer s'il ne choisit pas la bonne. Les auteurs mentionnent que la stratégie complète qui revient à essayer l'ensemble des signatures possibles est de complexité exponentielle en la taille du terme et c'est pourquoi ils ont préféré implémenter une heuristique très efficace en pratique qui s'appuie sur une information d'inclusion entre les relations impliquées, donnée par l'utilisateur. Tout simplement, on choisit toujours la signature donnant la plus forte (petite) relation entre les résultats. Dans notre exemple, $=$ et $<$ sont incomparables et plus précises que $\leq$, donc on utilisera l'une des trois premières signatures de préférence aux deux dernières si possible.

La technique est encore basée sur un ensemble de tactiques qui sont composées pour créer à la volée une tactique de réécriture complète et déterministe qui pourra être appliquée pour faire progresser le but.

Coq

Finalement, la tactique `setoid_rewrite` développée par Claudio Sacerdoti Coen (2004) dans Coq (après celle de Samuel Boutin, améliorée par Clément Renard) est basée sur une implémentation différente. L'auteur revendique le formalisme de "*Window Inferencing*" proposé dans (Robinson et Staples 1993) comme ancêtre de la réécriture généralisée. Ce formalisme est basé sur le raffinement des buts par simplifications successives du but, pouvant s'opérer à profondeur arbitraire et ouvrant potentiellement des sous-buts à raffiner récursivement.

La tactique diffère de l'implémentation de Basin en trois points importants :

- La tactique est semi-réflexive, c'est à dire qu'elle est séparée en deux parties : une procédure de recherche de preuve qui crée une trace et une preuve Coq que toute trace donne lieu à une réécriture correcte. La trace va indiquer quels sont les lemmes de substitutivité à appliquer. Ces lemmes sont comme toujours donnés par l'utilisateur.
- La tactique est complète. Plutôt que d'implémenter une heuristique comme Basin dans le cas où plusieurs signatures sont possibles, l'algorithme tente toutes les combinaisons de signatures possibles. L'argument donné pour ce choix est que les réécritures individuelles ne se font pas dans des termes profonds et que donc le facteur exponentiel n'est pas déterminant. D'autre part, la tactique n'implémente pas les sous-relations ce qui exclut d'utiliser la même heuristique.
- La tactique prend en compte la variance nativement dans les signatures, et gère donc les relations aymétriques (ainsi que les relations non réflexives). La gestion "native" permet d'éviter l'introduction d'un combinateur pour l'inversion d'une relation qui pourrait troubler l'utilisateur selon l'auteur.

Sacerdoti Coen indique des optimisations possibles à l'algorithme de recherche de preuve monolithique pour accélérer la création de la trace. Malheureusement, dans la pratique ces optimisations ne sont pas suffisantes si l'on veut faire des réécritures profondes dans le but.

Ce système était aussi limité dans son support des relations et morphismes paramétriques.

9.2 Deux algorithmes en un

Notre algorithme se situe entre celui de Basin et de Sacerdoti Coen. Il est séparé en deux parties : la génération de contraintes de morphismes et d'un squelette de preuve d'une part, et la résolution de ces contraintes d'autre part. Cette organisation modulaire permet d'étudier (et plus pratiquement de modifier) ces deux parties indépendamment.

Le système résultant permet d'utiliser une recherche efficace, et supporte les extensions décrites précédemment et d'autres encore : relations arbitraires, morphismes paramétriques, d'ordre supérieur, applications partielles et réécriture sous contextes.

La tactique s'appuie sur les définitions sur les relations vues précédemment (§8.2) et sur une définition des signatures et morphismes dans Coq que nous allons présenter dès maintenant.

9.2.1 Signatures et morphismes

On introduit maintenant les notions de signatures et morphismes.

Morphismes

La notion centrale est celle de respect d'une relation par un objet. On dit qu'un objet est un morphisme pour une relation donnée s'il est dans le noyau de cette relation.

```
Class Morphism {A} (R : relation A) (m : A) : Prop :=
  respect : R m m.
```

Le type du morphisme est implicite, puisqu'il peut être inféré à partir du type de la relation ou du morphisme lui-même. On introduit un espace de notations pour les relations vues comme des signatures.

```
Delimit Scope signature_scope with signature.
Open Local Scope signature_scope.
```

Clairement, tout élément dans un type accompagné d'une relation réflexive est un morphisme pour cette relation.

```
Instance reflexive_morphism [ Reflexive A R ] (x : A) : Morphism R x.
```

Signatures

L'autre notion importante est la signature d'un morphisme lorque son type est une flèche. Une signature donne les relations qui sont respectées en entrée et en sortie pour une fonction donnée.

```
Definition respectful (A B : Type)
  (R : relation A) (R' : relation B) : relation (A → B) :=
  λf g, ∀ x y, R x y → R' (f x) (g y).
```

On indique qu'une fonction respecte la relation R en entrée et R' en sortie si pour toute paire d'arguments liés par R, les résultats de l'application de la fonction sont liés par R'. La definition respectful donne la version relationnelle du respect, qui peut être appliquée à deux fonctions différentes. C'est une formalisation superficielle (\eng{shallow embedding}) des signatures, dans le sens où l'on ne peut pas les manipuler directement dans Coq à la différence de la version précédente de la tactique, mais on pourra toujours le faire *via* des tactiques qui filtrent sur la syntaxe de Coq.

On introduit des notations équivalentes à celles utilisées par la tactique précédente qui permettent de combiner des signatures. La flèche de respect associe à droite comme la flèche de l'espace fonctionnel. La notation ++> indique la covariance, on la confond pour l'instant avec la notation ⟹ pour l'invariance.

```
Notation " R ++> R' " := (@respectful _ _ (R%signature) (R'%signature))
  (right associativity, at level 55) : signature_scope.
Notation " R ⟹ R' " := (@respectful _ _ (R%signature) (R'%signature))
  (right associativity, at level 55) : signature_scope.
Notation " R ⟶ R' " := (@respectful _ _ (inverse (R%signature))
  (R'%signature)) (right associativity, at level 55) : signature_scope.
```

Les signatures de morphismes contravariants de R dans R' sont simplement des signatures de morphismes covariants de R^{-1} dans R'. On peut déclarer à l'aide de ces notations des instances de morphismes pour les opérateurs standards tel que la négation logique.

```
Program Instance not_impl_morphism : Morphism (impl ⟶ impl) not.
```

```
Program Instance not_iff_morphism : Morphism (iff ++> iff) not.
```

Il est aussi possible de déclarer des instances de morphismes générice, par exemple ici pour toute relation transitive on a :

```
Program Instance trans_contra_co_morphism [ Transitive A R ] :
  Morphism (R ⟶ R ++> impl) R.
```

La signature indique que pour toute relation transitive R, on a $R\ x'\ x \to R\ y\ y' \to R\ x\ y \to R\ x'\ y'$, c'est le contenu de la preuve de morphisme. Grâce à ce lemme, on va pouvoir réécrire avec des relations transitives. Par exemple :

```
Goal ∀ [ Transitive A R ] x y z, R x y → R y z → R x z.
Proof. intros A R T x y z H H0.
```

```
A : Type, R : relation A, T : Transitive R
x, y, z : A
H : R x y
H0 : R y z
============================
 R x z
```

```
  rewrite H.
```

```
A : Type, R : relation A, T : Transitive R
x, y, z : A
H : R x y
H0 : R y z
============================
 R y z
```

```
  assumption.
Qed.
```

Avant d'entrer dans une description des instances de morphismes restantes qui permettent de réaliser les réécritures standards, nous allons présenter l'algorithme de génération de contraintes de morphismes qui s'appuiera sur cette base.

9.2.2 Génération de contraintes

L'algorithme de réécriture rew fonctionne par récurrence sur le terme τ dans lequel on réécrit, qui correspond au type du but ou d'une hypothèse. L'algorithme prend en argument l'équation ρ par laquelle on veut réécrire, de la forme $\forall \overrightarrow{\phi}, R\ \overrightarrow{\alpha}\ s\ t$, une contrainte de relation (similaire à une contrainte de typage) ψ correspondant à un type et une relation sur ce type et renvoie optionellement un tuple (p, ψ, τ, τ') contenant une preuve p de la réécriture de τ en τ' par la relation ψ (donc p est de type $\psi\ \tau\ \tau'$).

La contrainte de relation donne la relation supposée être trouvée entre les deux sous-termes. On la définit comme la paire $(A = \texttt{Prop}, R = \text{impl})$ si l'on réécrit dans une hypothèse et $(\texttt{Prop}, \text{impl}^{-1})$ si l'on réécrit dans le but. En effet, si l'on réécrit dans une hypothèse de type τ on cherche une preuve de $\tau \to \tau[s/t]$ que l'on pourra appliquer à l'hypothèse : c'est une étape de preuve en avant. Au contraire pour le but, on cherche

une preuve de $\tau \to {}^{-1}\tau[s/t] \triangleq \tau[s/t] \to \tau$. En général, on ne connaît cette contrainte de relation qu'à la racine du terme, on ne valuera donc parfois que son type et on introduira une métavariable pour la relation.

N.B. : On va supposer que les relations, hypothèses et buts sont toujours dans `Prop`*, mais la construction fonctionnerait aussi bien dans* `Type`*, où les relations auraient un contenu calculatoire.*

Définition 9.2.1 (rew)**.** *L'algorithme est paramétré par un environnement courant Γ, qui correspond initialement à l'environnement du but, une équation ρ et une contrainte de relation ψ. On procède par cas sur τ :*

- ***Unify*** *: $\tau \equiv s'$ où $\rho\ \sigma : R'\ \overrightarrow{\alpha}'\ s'\ t'$. On a trouvé un sous-terme s'unifiant au membre gauche de l'équation avec la substitution σ. Soit $\tau_{s'}$ le type de s' et $\psi' = (\tau_{s'}, R'\ \overrightarrow{\alpha}')$. Si la contrainte de relation ψ est convertible à ψ', on renvoie $(\rho\ \sigma, \psi', s', t')$ contenant la preuve de la réécriture de s en t, instanciée par σ, puis la contrainte et les termes instanciés. Sinon on ajoute une contrainte de sous-relation :*

$$?sub_{R'} : \mathsf{subrelation}\ \psi_A\ (R'\ \overrightarrow{\alpha}')\ \psi_R$$

La preuve $p \triangleq (?sub_{R'}\ s'\ t'\ (\rho\ \sigma))$ est bien de type $(\psi_R\ s'\ t')$, on renvoie donc le tuple (p, ψ, s', t').

- ***App*** *: $\tau \equiv f\ \overrightarrow{e_n}$ Pour une application, on peut réécrire soit la fonction soit les arguments. On suppose $f : \overrightarrow{\beta_n} \to \psi_A$. On définit une nouvelle contrainte de relation sur cet espace fonctionnel*

$$\psi' \triangleq (\overrightarrow{\beta_n} \to \psi_A, pointwise\_relation^n\ \psi_R)$$

qui fait passer la contrainte originale sous la flèche. On appelle récursivement rew sur f avec cette contrainte.

  - *Si la réécriture réussit avec le résultat (p, ψ', f, f') on renvoie la preuve*

$$(p\ \overrightarrow{e_n}, \psi, f\ \overrightarrow{e_n}, f'\ \overrightarrow{e_n})$$

  qui applique les définitions point-à-point à chaque argument.

  - *Si la réécriture échoue, on appelle récursivement rew sur chacun des $\overrightarrow{e_n}$ avec des contraintes de relation instanciées par de nouvelles métavariables. Si au moins une réécriture réussit, on découpe la liste en deux à partir du premier argument réécrit e_k. On obtient une liste de résultats $\overrightarrow{(p_i, \psi_i, e_i, e_i')}^n_k$ qu'on définit point-à-point sur la deuxième liste :*
    - *Si rew Γ None e_i = Some c alors c*
    - *Sinon, on crée le tuple $(respect\ \beta_i\ ?R_i\ e_i\ ?Mor_i, (\beta_i, ?R_i), e_i, e_i)$ où $?R_i$: relation β_i et $?Mor_i$: Morphism $\beta_i\ ?R_i\ e_i$. On renvoie donc une preuve de $?R_i\ e_i\ e_i$ et on ajoute une contrainte de classe pour une preuve que e_i est un morphisme pour $?R_i$, généralement déchargée par une preuve de réflexivité d'une relation sur β_i. C'est la preuve nécessaire pour les arguments inchangés d'un morphisme.*

  On ajoute ensuite une contrainte de morphisme :

$$?Mor_f : \mathsf{Morphism}\ (\overrightarrow{(\psi_{iA})}^n_k \to \psi_A)\ (\psi_{kR} \cdots \Longrightarrow \psi_{nR} \Longrightarrow \psi_R)\ (f\ \overrightarrow{e_i}^{k-1}_1)$$

  Puis on construit la preuve de réécriture

$$p_f \triangleq ?Mor_f\ \overrightarrow{(e_i\ e_i'\ p_i)}^n_k$$

  On retourne enfin le tuple $(p_f, \psi, f\ \overrightarrow{e_n}, f\ \overrightarrow{e_n'})$.

– ***Lambda*** : $\tau \equiv \lambda x : \tau_1, \tau_2$
On suppose qu'on ne veut pas réécrire dans τ_1, mais qu'on veut effectuer la réécriture dans un environnement enrichi par τ_1. On appelle donc rew $(\Gamma, x : \tau_1)$ ψ' τ_2 où $\psi'_R = (\psi_A, \psi_R)$ si $\psi = (\tau_1 \to \psi_A, pointwise\_relation\ \psi_R)$ et $\psi'_R = (\tau_2, ?_{relation\ \tau_2})$ sinon. Si la réécriture échoue, on échoue. Sinon, elle retourne un tuple $(p_2, \psi_2, \tau_2, \tau'_2)$ et on renvoie la preuve :

$$((\lambda x : \tau_1, p_2), (\tau_1 \to \psi_{2A}, pointwise\_relation\ \psi_{2R}), (\lambda x : \tau_1, \tau_2), (\lambda x : \tau_1, \tau'_2))$$

Intuitivement, on a simplement étendu la preuve de réécriture sous $x : \tau_1$ à une preuve de réécriture pour tout $x : \tau_1$.

– ***Arrow*** : $\tau \equiv \tau_1 \to \tau_2$
*On est dans le cas non-dépendant, où τ_1 et τ_2 peuvent être réécrits indépendamment, en utilisant le morphisme impl si τ_1 et τ_2 sont des propositions, ou arrow dans les autres cas. On se ramène alors au cas **App**.*

– ***Pi*** : $\tau \equiv \Pi x : \tau_1, \tau_2$
*Dans ce cas on se ramène aux cas **App** et **Lambda** en utilisant l'équivalence $\Pi x : \tau_1, \tau_2 \equiv all\ \tau_1\ (\lambda x : \tau_1, \tau_2)$ (si τ : Prop). On va chercher un morphisme :*

$$Morphism\ ((\tau_1 \to Prop) \to Prop)\ (pointwise\_relation\ \psi_R)\ (all\ \tau_1)$$

On utilise ici la possibilité d'appliquer partiellement all. On peut aussi utiliser cette construction lorsque la réécriture échoue dans le domaine d'une flèche dans le cas précédent pour bénéficier de l'hypothèse τ_1 lorsqu'on réécrit τ_2.

– ***Fail*** : *sinon, on échoue en retournant None.*

L'algorithme nous retourne donc une preuve à trous de la réécriture de s vers t qu'il suffit d'appliquer à une hypothèse ou au but, laissant un ensemble de contraintes de classe à résoudre pour clore la preuve. Cette partie est laissée entièrement à l'algorithme de résolution des classes utilisant la base d'instance que nous allons décrire maintenant.

9.2.3 Résolution

Les problèmes de recherche de preuve engendrés par la tactique de réécriture sont des ensembles de contraintes de la forme Morphism A $(R_1 \Longrightarrow \cdots \Longrightarrow R_n)$ m, où A et m sont donnés mais où les R_n peuvent être des métavariables. Ces métavariables peuvent d'ailleurs apparaître dans plusieurs contraintes à la fois, notamment à cause des contraintes pour les arguments inchangés.

Les R_i peuvent être des relations arbitraires, elles peuvent notamment être formées à partir du combinateur inverse qu'il faut gérer de façon particulière. Il faut aussi gérer les sous-relations, les morphismes d'ordre supérieur et l'application partielle. Pour cela, nous allons introduire des tactiques $\mathcal{L}_{\text{tac}}$ qui vont étendre l'algorithme de recherche d'instances de Morphism. Elles sont définies dans un module Coq classique que nous présentons maintenant.

On peut déjà déclarer les combinateurs qui préservent les propriétés de compatibilité. Par exemple une relation complémentaire est un morphisme si la relation originale l'est.

```
Program Instance complement_morphism
  [ mR : Morphism (A → A → Prop) (RA ++> RA ++> iff) R ] :
  Morphism (RA ++> RA ++> iff) (complement R).
```

De même pour l'inverse.

```
Program Instance flip_morphism
  [ mor : Morphism (A → B → C) (RA ++> RB ++> RC) f ] :
  Morphism (RB ++> RA ++> RC) (flip f).
```

On déclare les morphismes pour les opérateurs logiques standards de la même façon.

```
Program Instance and_iff_morphism : Morphism (iff ++> iff ++> iff) and.
```

Pour les morphismes d'ordre supérieur, on utilise l'équivalence point-à-point comme indiqué dans la présentation de l'algorithme. Par exemple pour déclarer que le quantificateur existentiel transporte l'équivalence logique, il suffit de prouver :

```
Instance ex_iff_morphism {A : Type} :
  Morphism (pointwise_relation A iff ++> iff) (@ex A).
```

Le contenu de cette preuve est en fait : si pour tout x, $P\ x \leftrightarrow Q\ x$ alors $\exists\ x,\ P\ x \leftrightarrow \exists\ x,\ Q\ x$. On peut maintenant l'utiliser pour réécrire sous les existentielles avec la relation d'équivalence. Par exemple :

```
Goal ∀A (P Q : A → Prop), (∀ x, P x ↔ Q x) → (∃ x, P x) → (∃ x, Q x).
```

On peut aussi définir des morphismes génériques sur des ordres abstraits comme la relation partielle ci-dessous :

```
Instance per_morphism [ PER A R ] : Morphism (R ++> R ++> iff) R.
```

On ne détaillera pas ici l'ensemble des morphismes de la sorte, on va plutôt s'intéresser aux morphismes originaux qui permettent de supporter les capacités mentionnées précédemment.

Application partielle

Lors de la génération de contraintes, on génère des signatures correspondant seulement aux arguments réécrits d'une fonction, en commencant au premier argument réécrit. Cela permet de gérer les morphismes paramétriques de manière transparente : on ne réécrit jamais dans les paramètres du morphisme donc on n'encombre jamais la signature avec des relations sur ces paramètres. Seulement, cela interagit aussi avec les morphismes non paramétriques. Ainsi, si l'on a $P \rightarrow Q$ et que l'on réécrit avec $Q \leftrightarrow Q'$, la contrainte engendrée aura la forme `Morphism (iff ⟹ inverse impl) (impl P)`. Seulement, les morphismes pour l'implication sont de la forme `Morphism (iff ⟹ iff ⟹ inverse impl) impl`. Il nous faut donc un moyen de dériver automatiquement l'un à partir de l'autre. Ce moyen nous est donné par l'instance suivante.

```
Instance partial_app_morphism [ Morphism (A → B) (R ++> R') m,
  Morphism A R x ] : Morphism R' (m x) | 4.
```

Cette instance a une priorité faible, de façon à ne l'appliquer que lorsque aucun morphisme n'est déclaré sur $(m\ x)$.

Sous-relations

Une extension essentielle de l'algorithme de recherche d'instances est l'utilisation des sous-relations. On peut déclarer une relation comme sous-relation d'une autre à l'aide d'une déclaration instance classique. En standard nous avons les déclarations suivantes :

```
Instance iff_impl_subrelation : subrelation iff impl.
Instance iff_inverse_impl_subrelation : subrelation iff (inverse impl).
```

L'équivalence logique est la plus raffinée des trois relations sur les propositions. Cela signifie qu'un morphisme dont la signature se termine par iff peut être vue comme un morphisme donnant l'implication ou l'inverse de l'implication.

On peut prouver que l'équivalence point-à-point est un morphisme covariant pour la relation de sous-relation, ce qui permet de faire simplifier les contraintes de sous-relations sur le combinateur pointwise_relation. Plus explicitement, si on a $\forall\ x\ y,\ R\ x\ y \to S\ x\ y$ alors on obtient : $(\forall x,\ R\ (f\ x)\ (g\ x)) \to (\forall x,\ S\ (f\ x)\ (g\ x))$. Ces instances permettent une sorte de "*bootstrap*" : on peut réécrire automatiquement avec la relation de sous-relation au sein des signatures elles-mêmes. On a une instance pour la classe Morphism pour permettre la réécriture et une instance de subrelation qui étend la relation de sous-relation en exprimant la covariance de pointwise_relation.

```
Instance pointwise_subrelation_morphism :
  Morphism (subrelation ++> subrelation) (@pointwise_relation A B).
Instance pointwise_subrelation A [ sub : subrelation B R R' ] :
  subrelation (pointwise_relation A R) (pointwise_relation A R').
```

Enfin on peut ajouter des instances abstraites qui permettent de simplifier une recherche d'instance de sous-relation. Une instance essentielle est celle qui fait passer à travers les produits. Les sous-relations passent sous les produits de façon classique, contravariants à gauche et covariants à droite :

```
Instance respectful_subrelation
  [ suba : subrelation A R2 R1, subb : subrelation B S1 S2 ] :
  subrelation (R1 ++> S1) (R2 ++> S2).
```

Bien sûr, Morphism lui-même est un morphisme covariant pour les sous-relations.

```
Instance morphism_subrelation_morphism A :
  Morphism (subrelation ++> @eq _ ++> impl) (@Morphism A).
```

Il s'ensuit qu'on peut trouver une instance de morphisme pour m avec la signature R_2 si on trouve une sous-relation R_1 de R_2 pour laquelle m est un morphisme. On utilise ici simplement un lemme, laissé en dehors des instances déclarées.

```
Lemma subrelation_morphism [ sub : subrelation A R1 R2, mor : Morphism A R1 m ] :
  Morphism R2 m.
```

Tactiques En effet, ce lemme est bien trop général pour l'introduire dans la recherche d'instance de Morphism : il est toujours applicable. On va donc créer une tactique qui nous permettra de contrôler quand l'appliquer.

```
Inductive subrelation_done : Prop := did_subrelation.
```

On introduit un inductif qui permet de laisser une marque dans un but lorsqu'on a appliqué le lemme subrelation_morphism.

```
Ltac subrelation_tactic :=
  match goal with
    | [ _ : subrelation_done ⊢ _ ] ⇒ fail 1
    | [ ⊢ @Morphism _ _ _ ] ⇒ let H := fresh "H" in
      set(H :=did_subrelation); eapply @subrelation_morphism
  end.
Hint Extern 4 (@Morphism _ _ _) ⇒ subrelation_tactic : typeclass_instances.
```

La tactique échoue si le lemme a déjà été appliqué pour faire progresser la preuve. Sinon, elle ajoute la marque et applique le lemme. On ajoute cette tactique à la base d'instances qui pourra l'appliquer sur les buts dont la tête est Morphism. Grâce à ce contrôle, on peut faire toute la programmation logique de la tactique dans Coq avec $\mathcal{L}_{\text{tac}}$.

Morphismes duaux

On peut construire une tactique permettant de gérer les signatures contenant inverse de la même façon.

```
Program Instance morphism_inverse_morphism [ Morphism A R m ] : Morphism (R^-1) m.
```

Un objet est un morphisme pour R^{-1} s'il est un morphisme pour R. On va utiliser cette propriété pour dériver les signatures duales de morphismes déclarés en transformant toute signature contenant inverse en une signature équivalente de la forme inverse R. La propriété d'être un morphisme est équivariante pour des signatures équivalentes, on peut donc changer la signature à volonté.

```
Instance morphism_morphism A :
  Morphism (relation_equivalence ++> @eq _ ++> iff) (@Morphism A).
```

On introduit une classe spéciale pour normaliser les signatures vis-à-vis de l'inverse. On demande simplement que les deux relations soient équivalentes.

```
Class (A : Type) ⇒ Normalizes (m : relation A) (m' : relation A) : Prop :=
  normalizes : relation_equivalence m m'.
```

Encore une fois, on introduit un certain nombre d'instances pour réaliser le programme logique. Par exemple, si l'on a une flèche entre deux relations inverses, c'est équivalent à avoir l'inverse de la flèche entre les relations.

```
Instance inverse_respectful_norm : Normalizes (A → B) (R0 ⟶ R1^-1) (R0 ++> R1)^-1.
```

On utilise une tactique comme précédemment pour contrôler l'application. Ceci clôt la présentation de la partie Coq de la tactique.

9.2.4 Analyse

La tactique `setoid_rewrite` fonctionne donc par génération des contraintes puis résolution *via* les instances définies en Coq. L'algorithme de génération de contraintes est clairement linéaire en la taille du terme, et n'a donc aucune influence sur les performances de la tactique entière. La recherche d'instance est par défaut une recherche en profondeur d'abord avec une profondeur limitée (100 par défaut). Il est essentiel que la recherche ne boucle pas, et c'est pour cela que nous contrôlons l'application de certaines instances.

La tactique a une performance bien meilleure que la précédente (Coen 2004) sur les buts conséquents, puisque la recherche en profondeur permet souvent d'obtenir une preuve directe, sans trop de retour en arrière. En pratique, la complexité est donc plutôt linéaire en moyenne. En revanche la recherche ne retourne pas toutes les solutions possibles. Quant au terme de preuve, sa taille est de l'ordre de celle du terme réécrit, plus la taille des preuves trouvées par la recherche d'instance. En effet, l'algorithme renvoie une preuve à trou contenant pour chaque application impliquant un terme réécrit un trou correspondant à l'instance de Morphism pour la fonction ainsi que pour chaque argument jusqu'à deux termes de la même taille plus des trous correspondants aux preuves de Morphism appropriées. Les termes non réécrits apparaissent jusqu'à deux fois dans la preuve s'ils viennent après des arguments réécrits, plus une fois dans la preuve de Morphism correpondante. La taille de la preuve est donc linéaire en la taille du terme réécrit.

Implémentation & expérimentations La tactique présentée ici est disponible dans la dernière version de Coq où elle remplace l'implémentation précédente (Sozeau 2008c).

L'implémentation a donc déjà été testée sur toute la librairie standard ainsi que les contributions envoyées par les utilisateurs. Les performances sont similaires à la tactique précédente sur ces développements, mais bien supérieures sur d'autres développements qui utilisent la réécriture en profondeur, comme celui réalisé par Benton et Tabareau (2009).

Pour accélérer la recherche, on emploie un réseau de discrimination ("*discrimination net*") amélioré par rapport à celui utilisé par `auto`. En effet celui-ci ne gérait pas les buts avec existentielles ce qui était crucial dans notre cas. Grâce à cette modification, la recherche des instances de morphisme sur une définition de l'utilisateur est instantanée puisqu'on peut indexer sur celles-ci, plutôt que de devoir faire une unification assez coûteuse à chaque fois.

On a aussi ajouté une analyse des dépendances entre sous-buts à cette version d'`eauto` qui permet de faire des coupes sûres ("*green cuts*") dans la recherche.

9.2.5 Raffinements

La tactique étend la version précédente puisqu'elle gère aussi l'option `at` qui permet de spécifier quelles instances du lemme doivent être réécrites. Lorsqu'on fait le parcours de gauche à droite pour trouver les sous-termes s'unifiant avec le lemme, on ne réécrit que les occurrences données, en comptant une occurrence par sous-terme s'unifiant et en parcourant récursivement tous les sous-termes non sélectionnés. Il est à noter que la sémantique de cette nouvelle tactique diffère significativement de la précédente vis-à-vis de l'unification. La tactique `rewrite` lorsqu'on lui donne un lemme en argument commence par chercher le premier sous-terme du but avec lequel il s'unifie. Elle réécrit ensuite tous les sous-termes convertibles avec le sous-terme trouvé. En revanche, notre nouvelle tactique ne fait pas d'unification *a priori*. Il est ainsi possible de faire des réécritures sur des instances différentes d'un lemme. Typiquement, cela permet avec l'aide du modifieur `at` de faire des réécritures profondes sans avoir à donner les sous-termes concernés explicitement. Par exemple, soit le but :

$$\forall\ x\ y\ z : \mathbb{N}, (x + y) + z = y + (x + z)$$

Si l'on veut réécrire avec le lemme de commutativité de type $\forall\ x\ y : \mathbb{N}, x+y = y+x$ il y a quatre occurrences possibles avec la nouvelle tactique. La première transforme $(x+y)+z$ en $z+(x+y)$ et la deuxième réécrit le sous-terme $(x+y)$ en $(y+x)$ pour obtenir $(y+x)+z$. Les deux dernières occurrences sont pour la partie droite de l'équation. Avec la tactique précédente, seule la première occurrence aurait pu être réécrite sans mentionner une partie du terme.

Cette nouvelle sémantique, combinée au modifieur `at`, permet d'être plus fin dans la sélection des sous-termes réécrits mais est aussi essentielle pour pouvoir réécrire sous les lieurs. En effet, pour pouvoir capturer la variable introduite par un lieur dans un but il faut absolument faire l'unification des lemmes avec les sous-termes sous contexte. Par exemple, pour réécrire avec le lemme $\forall\ x, x * 0 = 0$ dans le but $\exists\ x, x * 0 \neq 1$.

9.3 Conclusion

Nous avons présenté une nouvelle tactique de réécriture généralisée en Coq, basée sur le système de classes de type. Elle étend la tactique précédente de façon conséquente puisqu'elle gère les sous-relations et la réécriture sous les lieurs en plus des relations arbitraires. La nouvelle architecture permet plus d'extensibilité *via* $\mathcal{L}_{\text{tac}}$ et un meilleur contrôle sur les performances grâce à la séparation de la génération de contraintes et de

leur résolution. Le plongement superficiel des signatures dans Coq avec l'aide des classes permet aussi une intégration simplifiée aux développements utilisateurs.

Travaux futurs

La tactique peut encore être améliorée tant au niveau des performances que des fonctionalités. La résolution est accélérée en utilisant une structure de réseau de discrimination dans l'algorithme de recherche pour choisir les lemmes applicables à un but, plutôt que de tenter toutes les unifications à chaque fois. Malheureusement, l'unification est toujours nécessaire après sélection des lemmes. L'implémentation et la certification d'un réseau de discrimination qui permet de faire de l'unification dans le cas dépendant avec des termes à trou, le tout modulo conversion nous semble une direction intéressante. Un tel outil serait utile pour d'autres algorithmes à base de recherche de preuve.

Comme on le discutera dans la section 10.2.1, l'algorithme de résolution peut être considérablement amélioré et rendu plus flexible. On pourrait en particulier permettre de retourner toutes les solutions d'une réécriture là ou nous choisissons arbitrairement la première instantiation des métavariables trouvée.

Au niveau des fonctionnalités, il reste des difficultés à comprendre dans les cas dépendants. Par exemple, supposons qu'on ait défini la constante $\mathrm{div} : \mathbb{N} \to \{x : \mathbb{N} \mid x \neq 0\} \to \mathbb{N}$ et qu'on veuille prouver le but suivant :

$$x : \mathbb{N}, H_x : x \neq 0, E : x = 1, H_1 : 1 \neq 0 \vdash \mathrm{div}\ 0\ (x, H_x) = \mathrm{div}\ 0\ (1, E)$$

Il devrait être possible de réécrire avec l'hypothèse E pour résoudre ce but. Seulement, si l'on réécrit x en 1, il faut aussi modifier H_x, puisque $(1, H_x)$ n'est pas typable. Il serait intéressant de voir quelles extensions sont nécessaires sur les signatures et dans l'implémentation pour traiter ce genre de cas. La solution aujourd'hui est d'introduire une équivalence sur les types sous-ensemble qui compare les premières projections, effaçant ainsi la dépendance.

Chapitre 10

Conclusion

Sommaire

Nous avons présenté un système de classes de types pour Coq permettant de programmer et prouver de façon générique grâce à la surcharge et à l'extension du système aux types dépendants. Nous avons vu comment ce système permettait de construire des tactiques puissantes développées en grande partie dans Coq, grâce à un mécanisme de résolution extensible et contrôlable.

Nous allons passer en revue les approches similaires et décrire les extensions possibles sur ce travail.

10.1 Développements structurés dans les assistants de preuve

De nombreuses approches ont été proposées pour la formalisation des mathématiques dans les assistants de preuve. Nous présentons les méthodes utilisées actuellement dans Isabelle et Coq comparables aux classes de types.

10.1.1 Isabelle

L'approche poursuivie initialement dans Isabelle avec les locales (Kammüller, Wenzel, et Paulson 1999) est comparable à celle présentée ici. Les locales sont des contextes nommés qui fixent un certain nombre de définitions et théorèmes. Dans la conception originale des classes de types, les symboles de fonctions étaient vraiment surchargés et donc partagés par toutes les classes (Wenzel 1997). Les locales les regroupent avec les axiomes, et suppriment aussi la restriction à un seul type indexé par classe. Haftmann et Wenzel (2006) présentent une nouvelle implémentation du système de classes de types basée sur les locales, se passant complètement de la surcharge.

Les locales permettent de décrire un contexte de preuve arbitraire et vont plus loin que nos classes dans le sens où de l'information extra-logique peut y apparaître (notations, indications pour les tactiques de simplification...). Un certain nombre de primitives (toujours en dehors du noyau) pour manipuler les locales sont aussi disponibles, par exemple pour extraire une locale d'un contexte donné ou bien combiner plusieurs locales pour n'en faire qu'une. Les derniers développements dans ce sens (Ballarin 2006) reviennent au maintien d'un environnement de preuve complet qui s'apparente au développement à l'aide du système de modules de Coq. Une hiérarchie des contextes introduits par l'utilisateur est maintenue et l'on peut accéder facilement à un contexte pour l'instancier sur des définitions arbitraires au cours d'une preuve. Il semble cependant que la plupart des possibilités données par les locales pourraient être implémentées avec nos classes, donc en utilisant directement les primitives du système plutôt qu'en introduisant un système de module en dehors du noyau.

Une difficulté non résolue par les locales est l'impossibilité de quantifier sur les constructeurs de type dans la logique d'HOL. Cela signifie l'impossibilité de faire une classe des foncteurs ou la classe Monad vue précédemment. Il est cependant possible comme le montrent Huffman, Matthews, et White (2005) de travailler dans la logique HOLCF, la logique des fonctions calculables de Church, et par une construction de domaine obtenir cette quantification. C'est cependant une manœuvre assez complexe qui n'a pas lieu d'être en Coq où le produit dépendant n'a pas de telle restriction.

10.1.2 Coq

Coercions

Le mécanisme de coercions implicites de Coq (Saïbi 1997) peut être utilisé lui aussi pour faire des développements mathématiques. Il est par exemple utilisé intensivement dans la bibliothèque CoRN développée à Nijmegen qui formalise les bases de l'algèbre et de l'analyse. Ce mécanisme permet de déclarer des coercions entre types respectant un certain nombre de critères sur la forme du graphe de coercion résultant pour garantir la cohérence. Il existe souvent plusieurs chemins entre deux structures, ils doivent dans ce cas être convertibles. Le mécanisme de coercion suppose que l'on imbrique les structures dans des enregistrements, pour créer des hiérarchies par agrégation. En l'absence de sous-typage sur les enregistrements, les coercions réalisent l'inclusion d'une structure dans une autre par l'insertion automatique de projections.

C'est strictement le même mécanisme qui est à l'oeuvre lorsqu'on utilise des classes imbriquées : si A inclut B qui inclut C, on peut construire une instance de C à partir d'une instance de A en appliquant les projections correspondant à l'inclusion. La différence essentielle est dans l'algorithme de résolution. Dans le cas des coercions, on a un certain nombre de propriétés sur les graphes engendrés par les coercions qui assurent l'unicité de toute combinaison de coercions, modulo convertibilité. Dans le cas des classes, l'algorithme par défaut choisit la première solution trouvée et ne garantit pas son unicité. Il faudrait vérifier le critère à l'introduction des instances et implémenter un algorithme de résolution adéquat pour obtenir la même fonctionnalité (c.f. §10.2.1).

Structures canoniques

Les structures canoniques sont un autre outil pour le développement structuré en Coq, proche des classes de types. Essentiellement, une structure canonique permet de déclarer une implémentation par défaut pour une instance d'une signature donnée, et de l'inférer automatiquement lorsqu'un objet ayant pour type cette instanciation est requis.

Par exemple on peut déclarer que le monoïde canonique sur $\mathbb{N}$ est $(0, +)$. Supposons que l'on fait ensuite référence à l'élément vide ϵ ou à l'opération $\bullet$ d'un monoïde tout en imposant qu'il soit de type $\mathbb{N}$, par exemple dans l'énoncé : $\forall x : \mathbb{N}, \epsilon \bullet x = x$. Alors on inférera automatiquement qu'on parle du monoïde canonique $(0, +)$.

Techniquement, on se retrouve à l'unification avec des contraintes de conversion de la forme $p_i \overrightarrow{\tau}?_{\mathsf{S}\overrightarrow{\tau}} \equiv \tau$ où S est un type d'enregistrement déclaré comme structure canonique et p_i une projection de cet enregistrement. On va simplement chercher dans la base de déclarations l'enregistrement pour lequel cette équation est vérifiée et résoudre la métavariable avec cette solution.

Ce mécanisme diffère des classes de types sur deux points importants :

- Les structures canoniques s'appuient essentiellement sur les projections d'enregistrements, et donc ne fonctionnent pas très bien avec les enregistrements paramétriques où l'accès aux paramêtres ne passe pas par des projections mais est cette fois direct. *A contrario*, les classes ne peuvent être indexées que sur leurs paramètres et empêchent l'utilisation d'une contrainte d'unification sur une projection pour résoudre une classe. Il semblerait cependant raisonnable et utile de combiner les deux façons de présenter les problèmes d'unification.
- La résolution d'une contrainte de structure canonique n'utilise rien de plus que la convertibilité classique et doit être immédiate. Les classes permettent de faire une recherche de preuve arbitrairement complexe pour trouver une instance, mettant en jeu l'unification mais aussi la résolution grâce à des lemmes (et en fait même des tactiques arbitraires dans l'implémentation courante).

Les deux mécanismes peuvent cependant être utilisés simultanément, et il reste encore à faire des expériences pour savoir quels sont leurs domaines d'application idéaux.

10.2 Extensions et travaux futurs

De nombreuses directions sont ouvertes autour des classes, nous en détaillons les plus importantes.

10.2.1 Résolution

L'algorithme de résolution des instances est très fragile du fait que l'on n'a aucun critère sur la forme des lemmes utilisés et qu'on permet d'étendre l'algorithme en $\mathcal{L}_{\text{tac}}$ sans restrictions. C'est raisonnable pour les recherches d'instance de classes propositionnelles, où le contenu des objets-preuves nous importe moins que leur existence. Cependant, lorsqu'on programme ou qu'on manipule des structures mathématiques, il est essentiel de savoir quelle instance va être choisie et d'indiquer au programmeur les ambiguïtés éventuelles.

Il faut pour cela développer un algorithme de résolution qui retourne toutes les instances possibles d'une classe. Il serait donc souhaitable de développer un algorithme générique et paramétrable qu'on pourrait instancier pour obtenir un algorithme de recherche personnalisé, retournant la première ou toutes les solutions suivant une stratégie d'exploration donnée.

En complément, il faut aussi permettre d'énoncer des critères sur les instances qui garantissent la terminaison, le déterminisme ou la décidabilité de la recherche d'instance. On pourra s'inspirer des critères utilisés dans Haskell pour cela.

Enfin, on pourrait intégrer directement la résolution à l'unification en spécifiant qu'on ne lance une résolution que lorsque certaines variables sont instanciées, à la manière des structures canoniques.

10.2.2 Intégration

Les tactiques existantes ne fonctionnent pas directement avec les classes de types, il faut donc les adapter une à une pour utiliser la résolution au bon moment. Typiquement, la tactique `apply` type son argument sans contrainte de type et suppose qu'on lui retourne un type complètement résolu avant de l'unifier avec le but. On n'utilise donc pas l'information du but lorsqu'on applique une méthode de classe, ce qui produit soit un échec soit une résolution inattendue lorsqu'on type le lemme donné en argument. Un certain nombre de tactiques doivent donc être modifiées pour obtenir une intégration correcte des classes dans les développements.

Il est aussi intéressant d'utiliser les classes pour structurer la librairie standard de Coq. On pourrait définir une fois pour toutes les concepts de base tels que l'associativité d'une fonction par une classe puis l'utiliser partout où l'on a prouvé l'associativité d'une fonction particulière dans la librairie. Inutile alors de se souvenir du nom du lemme prouvant l'associativité de Zplus, il suffit de faire un appel à la résolution d'instances pour l'obtenir. Un travail de découverte automatique des preuves existantes dans la librairie de Coq a été entrepris par Matthias Puech (2008) en s'appuyant sur les classes, qui permet de retrouver automatiquement ces preuves disséminées dans les librairies.

10.2.3 Développements basés sur les classes

On a vu que les classes permettent de faire des développements génériques et abstraits facilement tout en faisant l'interface entre le système de tactiques et le code ML d'un côté et le langage fonctionnel Gallina de l'autre à travers l'algorithme de typage. D'autre tactiques peuvent bénéficier de cette architecture, comme par exemple la tactique `setoid_ring` qui implémente des algorithmes sur les anneaux (Grégoire et Mahboubi 2005) et qui utilise actuellement des enregistrements pour implémenter ces structures. Ils pourraient probablement être remplacés avantageusement par des classes.

Nous comptons travailler en collaboration avec des membres du projet conjoint composants mathématiques INRIA-Microsoft pour évaluer les classes de types dans ce cas et dans le cadre plus général de l'ingénierie des preuves dans Coq.

10.3 Conclusion

Pour résumer, nous avons présenté une implémentation d'un système de classes de types inspiré d'Haskell en Coq. Notre version est nettement plus expressive que celle de Haskell puisqu'elle supporte directement l'indexation d'une classe par des valeurs grâce au produit dépendant. Elle est aussi plus permissive, la recherche d'instance donnant lieu à une recherche de preuve générale. Nous avons montré la légèreté de l'implémentation qui s'appuie sur et étend des fonctionnalités existantes de l'assistant de preuve. Les exemples démontrent les bénéfices possibles pour la programmation générique et les développements mathématiques. Les classes peuvent enfin être utilisées de façon plus originale pour faire interface entre tactiques et développements utilisateurs comme dans la nouvelle tactique de réécriture présentée au chapitre 9.

Troisième partie

Expérimentations

Chapitre 11

Étude de cas : les Finger Trees

Sommaire

Les Finger Trees (Hinze et Paterson 2006) sont une structure de données de portée générale ayant de bonnes performances pour un grand nombre d'opérations. Leur généricité et leur généralité permet de développer un grand nombre de structures aussi diverses que des séquences ordonnées ou des arbres d'intervalles au-dessus d'une unique implémentation. Malheureusement, les systèmes de type utilisés dans les langages fonctionnels courants ne permettent pas de garantir la cohérence de la paramétrisation et la spécialisation de cette structure. D'autre part, ils sont très loin d'être capables de prouver la correction de leur implémentation. Ces deux problèmes, d'une part de modularité et d'autre part d'expressivité du système de type peuvent être résolus en utilisant un langage à types dépendants comme Coq. Pour réaliser un tel développement, nous avons utilisé l'extension Program qui rend l'écriture de telles spécifications bien plus aisée.

Nous allons non seulement implémenter les Finger Trees de façon certifiée mais aussi faire apparaître leur structure interne de telle manière qu'elle soit utilisable pour développer modulairement des structures certifiées au-dessus des Finger Trees. Ce chapitre est basé sur un article décrivant le développement (Sozeau 2007b) et correspond à la version 8.2 de la contribution Coq (les crochets des classes ont été remplacés par des accolades précédées d'une apostrophe `'{ }` dans cette version).

11.1 Introduction

Les Finger Trees sont basés sur une implémentation des arbres 2-3 (à branchement 2 ou 3 à chaque nœud) qui donne une complexité en temps amorti constante ou au pire cas logarithmique à l'ensemble des opérations sur les séquences. On peut ajouter ou enlever des éléments de chaque côté, découper à un endroit arbitraire et concaténer en temps logarithmique. Cette structure est suffisament versatile, robuste et efficace en pratique pour être utilisée comme implémentation pour le module Data.Sequence d'Haskell depuis la version 6.6.

Notre première contribution va être de montrer que l'implémentation d'Hinze & Paterson est correcte, c'est-à-dire que toutes les fonctions sont totales, terminent et que les invariants de la structure de Finger Tree sont préservés par celles-ci. En fait, comme les invariants font entièrement partie de notre structure, nous allons les propager vers les clients de la librairie, qui pourront donc en profiter. Notre deuxième contribution consiste en le développement d'une structure de séquences à accès aléatoire certifiée au dessus des Finger Trees utilisant cette information de façon modulaire. Ce développement (disponible sur le web (Sozeau 2007a)) a été rendu possible par l'utilisation de l'extension Program que nous avons développé dans la première partie et son extension aux familles inductives.

Nous allons tout d'abord introduire la structure des Finger Trees (§11.2) en Haskell puis décrirons incrémentalement l'implémentation des Finger Trees dépendants (§11.3) et les extensions de Russell nous permettant de traiter les familles inductives. On présentera les spécialisations possibles section 11.4, aboutissant à une implémentation des séquences certifiée. Enfin on discutera l'extraction section 11.5 et on s'intéressera en particulier au test que représente le sujet du concours de l'ICFP 2007 (ICFPC 2007).

11.2 Finger Trees

11.2.1 Introduction

Les Finger Trees sont une spécialisation des arbres 2-3 qui supporte essentiellement le découpage et la concaténation en temps logarithmique.

Avant d'entrer dans l'implémentation des arbres eux-mêmes, il nous faut développer des structures de base. La première est la structure de Digit. On fixe la taille des Digit de α à 4 valeurs de α. Cette taille a des répercussions sur la complexité des opérations sur les arbres qui est discutée dans l'article original (Hinze et Paterson 2006).

```
type Digit α =
    | One α
    | Two α α
    | Three α α α
    | Four α α α α
```

On suppose données les opérations partielles prenant la tête ou la queue d'un Digit à gauche ou à droite (digit_head, digit_last, digit_tail, digit_liat) ou ajoutant une valeur à

gauche ou à droite (add_digit_left, add_digit_right. Viennent ensuite les 2-3 nœuds : ce sont des "*buffers*" polymorphes à deux ou trois valeurs :

```
data Node α = Node2 α α | Node3 α α α
```

On définit enfin les Finger Trees :

```
data FingerTree α =
    | Empty
    | Single α
    | Deep (Digit α) (FingerTree (Node α)) (Digit α)
```

Un FingerTree α peut être soit vide (Empty), soit un singleton de α (Single), soit un arbre ayant pour racine un noeud de branchement (Deep). À chaque noeud Deep, on a à gauche et à droite des Digit α qui permettent d'atteindre rapidement les éléments en tête ou en queue de l'arbre. Au centre, on a un FingerTree non pas de α mais de nœuds de α. On a donc affaire à un type inductif *imbriqué*, où le paramètre polymorphe varie à chaque niveau : un FingerTree α peut contenir un FingerTree (Node α) qui peut contenir un FingerTree (Node (Node α)) et ainsi de suite.

Types algébriques imbriqués

De telles définitions de types algébriques imbriqués sont acceptées dans de nombreuses variantes de la famille ML et dans Coq (sous la forme de paramètres depuis la v8.1), mais l'écriture de fonctions sur ceux-ci est plus rarement supportée. En effet, les contraintes de typage générées à l'inférence de type lorsqu'on écrit une fonction polymorphe sur un type imbriqué n'entrent pas dans le cadre de l'algorithme d'Hindley-Milner. À chaque appel récursif, on instancie la variable polymorphe par un type différent. Prenons l'exemple des listes imbriquées :

```
data PowerList α =
    | PowerNil
    | PowerCons α (PowerList (α × α))
```

Un objet de type PowerList α est en fait une liste dont chaque noeud contient deux fois plus d'éléments que sont prédécesseur : l'instanciation de α dans le constructeur PowerCons force chaque sous-noeud à contenir deux fois plus d'information. Une PowerList α de taille n contient donc $2^n - 1$ objets de α.

On peut écrire des fonctions polymorphes sur PowerList, comme par exemple la fonction montrant que PowerList est un foncteur :

```
map :: (α → β) → PowerList α → PowerList β
map f PowerNil = PowerNil
map f (PowerCons a l) = PowerCons (f a) (map (λ(x, y), (f x, f y)) l)
```

On voit ici que non seulement le type change à l'appel récursif (on passe de α à $(\alpha \times \alpha)$), mais du coup les autres arguments polymorphes doivent être réinstanciés : ici on "*lifte*" la fonction polymorphe pour opérer sur des paires. Seul le compilateur GHC est capable d'accepter cette définition lorsqu'on lui donne la signature. Dans les autres systèmes, ou lorsqu'on omet la signature en Haskell, l'algorithme d'Hindley-Milner provoque une erreur d'"*occurs check*" puisqu'il va tenter d'unifier α et $(\alpha \times \alpha)$. On ne rencontre pas de tels problèmes dans Coq où le polymorphisme est explicite et ou l'on base l'inférence uniquement sur l'unification sans aucune généralisation implicite.

On peut voir les types imbriqués comme une sous-catégorie des familles inductives dont on va voir un exemple avec les Finger Trees. Intuitivement, on peut dire que la contrainte

PowerList $(\alpha \times \alpha)$ est poussée récursivement pour donner une forme particulière aux sous-structures. Les indices des familles inductives permettent dans un sens aussi de faire remonter de l'information des sous-nœuds aux structures qui les contiennent. Ghani et Johann (2007, 2008) formalisent d'ailleurs ces intuitions explicitement et catégoriquement en étudiant les algèbres initiales correpondant aux types imbriqués et aux "*Generalized Algebraic Data Types*" (GADT) de Haskell qui sont proches des familles inductives de la théorie des types.

11.2.2 Implémentation

On peut implémenter sur les Finger Trees un ensemble d'opérations qu'on développera plus précisément dans la suite : ajouter des éléments à gauche ou à droite, décomposer un FingerTree non vide en une tête et un arbre restant à l'aide d'une vue ou bien encore concaténer deux arbres. Par exemple, pour prendre la tête d'un arbre à droite on écrit :

```
last :: FingerTree α → Maybe α
last Empty = None
last (Single a) = Some a
last (Deep l m r) = Some (digit_last r)
```

Les fonctions sont assez simples à programmer excepté pour la concaténation de deux Finger Trees. La fonction de concaténation fait près de 200 lignes de code Haskell. Elle est en fait spécialisée pour toute combinaison de deux arbres et de 0 à 4 valeurs à insérer entre eux (le contenu d'un Digit). Dans le cas de base où l'on a 0 valeurs à insérer entre les deux arbres, on filtre sur les deux arbres. Le seul cas non-trivial est lorsque les deux arbres sont des nœuds Deep. On va alors appeler des fonctions de concaténation par cas sur le Digit à droite de l'arbre gauche et à gauche de l'arbre droit. En fonction de leurs tailles, on va avoir à insérer entre 0 et 4 valeurs entre les sous-arbres à gauche et à droite sur lesquels on se rappelle récursivement. Hinze et Paterson indiquent qu'ils ont même écrit un programme générant le code de cette fonction de peur de se tromper. Nous verrons qu'on peut la vérifier statiquement : les valeurs dans l'arbre résultant sont bien dans le même ordre que dans les arbres initiaux.

La complexité de ce code est surtout due à la structure simple mais très contrainte des Finger Trees. Cette structure est aussi leur grande force, la complexité de toute ces opérations est en effet logarithmique au pire cas en temps amorti (Okasaki 1996a,b). On obtient donc une structure très générale de séquences, purement fonctionnelle et efficace. L'implémentation est comparativement simple par rapport aux structures supportant le même ensemble d'opérations avec des complexités proches, notamment les "*deques*" de Kaplan et Tarjan (1999) récemment vérifiées par Yann Régis-Gianas dans le système Pangolin (Régis-Gianas et Pottier 2008).

11.2.3 Mesures

La deuxième idée géniale dans l'article de Hinze & Paterson est la généralisation des Finger Trees à ce qu'il appellent des *mesures*. On va annoter la structure des FingerTree α par des valeurs dans un type v sur lequel on a un monoïde $(\epsilon, \bullet)$ et une fonction de mesure d'un élément $|\_| : \alpha \to v$. Intuitivement, la fonction de mesure va donner une abstraction de la valeur d'un élément qu'on pourra concaténer $(\bullet)$ avec les mesures d'autres éléments pour donner la mesure d'un arbre les contenant. Le monoïde est la bonne abstraction de la suite ordonnée des éléments dans l'arbre, elle ignore leur emplacement précis qui peut être modifié sans changer la mesure par associativité du monoïde. Évidemment, l'arbre vide a pour mesure ϵ et le singleton la mesure de son unique élément.

On introduit donc deux classes pour les monoïdes et les mesures :

```
class Monoid α where
    mempty :: α
    mappend :: α → α → α


class (Monoid v) ⇒ Measured α v where
    |_| :: α → v
```

Grâce à ces deux classes on va pouvoir implicitement mesurer les doigts, les nœuds et les arbres.

On refait toute l'implémentation en calculant l'information sur les mesures parallèlement aux objets eux-mêmes. Les mesures vont être cachées (stockées) seulement à certains endoits de l'arbre. On redéfinit les structures de base ainsi :

- Les Digits sont inchangés. On implémente une instance de Measured sur les Digit qui attend une instance de Measured pour ses éléments et concatène les mesures de chaque élément.
- Les nœuds stockent la concaténations des mesures des éléments sous-jacents :

  ```
  data Node v α = Node2 v α α | Node3 v α α α
  ```

  L'instance de Measured projette simplement la mesure stockée sur un Node.
- Les FingerTree stockent la mesure de l'ensemble du sous-arbre à chaque noeud Deep :

  ```
  data FingerTree v α =
  | Empty
  | Single α
  | Deep v (Digit α) (FingerTree v (Node v α)) (Digit α)
  ```

On introduit des "*smart constructors*" node2, node3 et deep correspondant aux constructeurs sans mesure pour construire les structures correspondantes tout en calculant la mesure à stocker.

L'implémentation des Finger Trees mesurés ne pose pas davantage de problèmes, on utilise simplement les nouveaux constructeurs en lieu et place des anciens. Avec les mesures, on introduit une nouvelle opération sur les arbres : le découpage. On peut découper un FingerTree en deux parties à l'aide d'un prédicat sur les mesures $p : v \to \text{Bool}$. On parcourt l'arbre de gauche à droite et on s'arrête dès que le prédicat devient vrai sur la concaténation des mesures accumulée en ajoutant la mesure de chaque élément. On retourne alors l'arbre déjà parcouru, l'élément sur lequel on s'est arrêté et le reste de l'arbre :

```
split :: Measure α v ⇒ FingerTree v α → (v → Bool) → v →
  Maybe (FingerTree v α) × α × Maybe (FingerTree v α)
```

On n'a aucune hypothèse sur le prédicat, en particulier on ne sait pas s'il est monotone par rapport aux mesures, mais l'on aura quand même une propriété forte sur le résultat : le prédicat appliqué sur la mesure de l'arbre à gauche sera toujours faux et deviendra vrai en ajoutant l'élément trouvé. Cependant, l'implémentation suppose que si le prédicat n'est pas vrai pour la mesure d'un sous-arbre alors on peut ignorer ce sous-arbre entièrement (une forme d'incomplétude). Grâce à cette opération on peut implémenter un accès rapide à n'importe quelle partie de l'arbre. Nous verrons dans l'implémentation Coq comment cela se traduit dans les preuves sur des fonctions utilisant split.

11.2.4 Instantiations

L'intérêt de cette généralisation est dans la variété des spécialisations qu'on peut dériver à partir de cette unique implémentation. On peut déjà retrouver l'ancienne implémentation sans mesure en fixant le monoïde sur le type singleton : $((), \lambda\_\_, ())$ et la fonction de mesure des éléments $\lambda\_, ()$.

Une mesure peu intéressante en pratique mais utile en théorie est le monoïde des listes $([], +\!\!+)$ avec la mesure $\lambda x, [x]$. Dans ce cas la mesure d'un arbre est la liste de ses éléments dans l'ordre préfixe.

Hinze & Paterson développent un certain nombre d'instantiations intéressantes dans leur article, notamment :

- Des queues de priorité avec le monoïde des entiers et le maximum, la mesure d'un élément étant sa priorité.
- Des séquences à accès aléatoire avec le monoïde des naturels avec l'addition et la mesure constante à 1 pour tous les éléments.
- Des arbres d'intervalles avec le monoïde des intervalles et l'union.

Les possibilités offertes par cette généralisation semblent très nombreuses et le principe même d'annoter la structure par une abstraction des valeurs elle aussi a potentiellement d'autres applications, au moins dans d'autres structures algorithmiques. Nous développerons la spécialisation aux séquences de façon certifiée section 11.4.3.

11.3 Finger Trees dépendants

On a présenté l'implémentation des Finger Trees de Hinze & Paterson. On peut voir que certains points laissent à désirer dans cette implémentation :

- De nombreuses fonctions sont partielles : on peut essayer d'ajouter un élément à un Digit plein ou de découper un FingerTree vide.
- La cohérence des fonctions de mesure n'est assurée que parce qu'une seule instance de la classe Measure n'est autorisée par type dans l'environnement, ce qui empêche d'avoir deux spécialisations différentes des FingerTree sur le même type de mesure. On pourrait mélanger des FingerTree construits avec des mesures différentes sur le même type sans cette restriction.
- La cohérence des mesures cachées elles-même est mise en danger : on utilise des "*smart constructors*" pour s'assurer par construction que les mesures stockées dans les Node correspondent bien à la concaténation des mesures des sous-structures, mais rien n'empêche d'utiliser directement le constructeur avec une mesure érronée. En particulier, la sémantique des opérations sur les objets doit se refléter dans les mesures. Par exemple la concaténation de deux arbres doit donner un arbre dont la mesure est la concaténation des mesures de ces deux arbres. Rien ne permet de le garantir en Haskell.
- De même, Haskell ne permet pas de vérifier les invariants donnés par Hinze & Paterson sur les opérations telles que split.

Nous allons résoudre ces problèmes en donnant une implémentantation des Finger Trees dans Coq à l'aide de Program. Nous allons présenter de nouveau les structures de base des Finger Trees mesurés, mais cette fois-ci en nous attachant à certifier le développement en faisant apparaître les invariants dans les structures et les spécifications et en vérifiant la correction (totale) de notre implémentation.

11.3.1 Monoïdes

On va implémenter la classe des monoïdes en Coq par un enregistrement (ou "*record*") paramétré. On définit tout d'abord les lois du monoïde. On va utiliser le mécanisme de *sections* de Coq intensivement dans la suite. Les sections permettent d'écrire un ensemble de définitions paramétrées par un ensemble de variables de section (généralement des types, mais parfois aussi des opérations). Lorsqu'on clôt une section, toutes les définitions à l'intérieur de celle-ci sont automatiquement généralisées par rapport aux variables de section qu'elles utilisent.

```
Section Monoid_Laws.
```

Le support du monoïde m peut être n'importe quel type.

```
  Variable m : Type.
```

On suppose donnés l'élément neutre mempty et l'opération du monoïde mappend.

```
  Variables (mempty : m) (mappend : m → m → m).
```

On peut définir des notations appropriées pour ces deux variables.

```
  Notation "'ϵ" := mempty.
  Infix "•" := mappend (right associativity, at level 20).
```

Finalement, on définit les trois lois du monoïde.

```
  Definition monoid_id_l_t : Prop := ∀ x, ϵ • x = x.
  Definition monoid_id_r_t := ∀ x, x • ϵ = x.
  Definition monoid_assoc_t := ∀ x y z, (x • y) • z = x • y • z.
End Monoid_Laws.
```

Toutes les variables dans la section Monoid_Laws sont maintenant *déchargées*, on doit donc appliquer chaque définition de loi à des objets mempty et mappend particuliers. On définit la classe de type représentant un monoïde sur un type m :

```
Class Monoid (m : Type) : Type := {
  mempty : m;
  mappend : m → m → m;
  monoid_id_l : monoid_id_l_t m mempty mappend;
  monoid_id_r : monoid_id_r_t m mempty mappend;
  monoid_assoc : monoid_assoc_t m mappend }.

Infix "•" := mappend (right associativity, at level 20).
Notation ϵ := mempty.
```

On peut maintenant créer des instances de Monoid. Par exemple on peut déclarer le monoïde $([\,], +\!\!+)$ sur les listes. On peut utiliser Program pour créer des monoïdes en ne mentionnant explicitement que les parties informatives et en faisant de chaque preuve une obligation. Celles-ci peuvent être résolues automatiquement.

```
Require Import Coq.Lists.List.

Program Instance list_monoid A : Monoid (list A) :=
  { mempty := []; mappend := @app A }.

  Solve Obligations using red; auto with datatypes.
```

On définit aussi la classe Measured tout comme en Haskell qui va permettre de faire référence indifféremment à la mesure d'un doigt, un nœud ou un Finger Tree à l'aide de la fonction surchargée measure.

```
Class Measured v {m : Monoid v} (A : Type) := { measure : A → v }.
```

Par défaut, tout les arguments n'apparaissant pas à gauche de la flèche sont implicites. On veut ici que seule l'implémentation du monoïde le soit.

```
Implicit Arguments Measured [[m]].
```

On peut utiliser la notation "*mixfix*" ‖ _ ‖ pour se référer à la fonction measure. D'ores et déjà on peut déclarer une instance paramétrique pour prendre la mesure d'un objet de type option A si A est mesurable

```
Instance option_measure '(Measured v A) : Measured v (option A) :=
  { measure x := match x with Some x ⇒ ‖ x ‖ | None ⇒ ϵ end }.
```

11.3.2 "Doigts"

Les doigts de nos arbres sont simplement des buffers d'une à quatre valeurs d'un type polymorphe A.

```
Section Digit.
  Variable A : Type.
  Inductive digit : Type :=
  | One : A → digit
  | Two : A → A → digit
  | Three : A → A → A → digit
  | Four : A → A → A → A → digit.
```

On construit des prédicats simples sur les doigts qu'on va utiliser pour spécifier les fonctions partielles. Un digit est plein (full) lorsqu'il contient déjà quatre valeurs, et single lorsqu'il n'en contient qu'une. Notez qu'on utilise ici les constructueurs de propositions True et False et non pas les constructeurs du type booléen true et false.

```
  Definition full (x : digit) :=
    match x with Four _ _ _ _ ⇒ True | _ ⇒ False end.
  Definition single (x : digit) :=
    match x with One _ ⇒ True | _ ⇒ False end.
```

On définit l'addition d'un élément à gauche d'un digit.

```
  Program Definition add_digit_left (a : A) (d : digit | ¬ full d) : digit :=
    match d with
      | One x ⇒ Two a x
      | Two x y ⇒ Three a x y
      | Three x y z ⇒ Four a x y z
      | Four _ _ _ _ ⇒ !
    end.
```

C'est la première fonction *partielle* que l'on définit avec Program. On peut ajouter un élément à un doigt seulement si celui-ci n'est pas déjà plein. On requiert donc que l'argument d soit accompagné d'une preuve qu'il n'est pas plein. On utilise cette preuve pour montrer que le dernier cas est inaccessible, ce qu'on dénote par ! ('bang'). On peut toujours filtrer sur d comme sur n'importe quel autre digit : les propriétés n'ont aucune influence sur le code, uniquement dans les preuves. Au moment du filtrage, une projection est insérée implicitement au typage par l'algorithme de coercion. Ici, la compilation du filtrage de Program fait que l'obligation générée (figure 11.1) est facilement déchargée.

On définit de façon similaire l'addition à droite d'un digit et les divers accesseurs sur les digits : digit_head, digit_last, digit_tail, digit_liat. Ces deux dernières fonctions sont rendues totales en spécifiant que les digit en entrée ne doivent pas être des singletons

a : A
d : digit
Hd : ¬ full d
x, y, z, w : A
Heq_d : $d =$ Four $x\ y\ z\ w$

False

FIG. 11.1: Obligation générée pour add_digit_left.

(single). On définit aussi une instance paramétrique digit_measure qui calcule la mesure d'un doigt en fonction d'une mesure sur ses éléments.

On va maintenant définir la structure de Finger Tree sur un monoïde et une mesure donnée. On verra dans la section 11.4 comment diverses instantiations de ces paramètres permettent de dériver différentes structures.

```
Section DependentFingerTree.
  Context `{mono : Monoid v}.
```

La variable v représente le support du monoïde et *mono* est l'implémentation du monoide elle-même. Bien sûr, on a toujours accès au notations ϵ et $\bullet$ pour se référer aux méthodes de la classe Monoid.

11.3.3 Nodes

On a déjà défini les doigts, on définit maintenant les nœuds 2-3 paramétrés par une mesure. Les node cachent aussi une mesure qui contient la combinaison des mesures des sous-objets.

```
Section Nodes.
  Context `{ms : Measured v A}.
```

On dénote encore une fois la fonction measure par $\|\_\|$ dans la suite.

```
Inductive node : Type :=
| Node2 : ∀ x y, { s : v | s = ‖ x ‖ · ‖ y ‖ } → node
| Node3 : ∀ x y z, { s : v | s = ‖ x ‖ · ‖ y ‖ · ‖ z ‖ } → node.
```

On utilise un type sous-ensemble ici pour spécifier l'invariant sur la valeur cachée. Cet invariant ne peut pas être vérifié à l'aide de types simples puisqu'il dépend évidemment des valeurs portées par le constructeur de node. De plus, l'invariant fait référence à la *fonction* de mesure, qui va donc devenir un *paramètre* du type de donnée, ce qui requiert un mécanisme d'abstraction supplémentaire dans les langages fonctionnels courants, comme les classes de type de Haskell ou un système de module avec foncteurs à la ML. Ici, on utilise simplement le produit dépendant.

On peut aussi définir les "*smart constructors*" node2 et node3 qui calculent les mesures à la volée. On caste les expressions avec le type v pour désambiguer la surcharge qui sinon chercherait une instance de Monoid sur le type sous-ensemble $\{s : v \mid s = \| x \| \cdot \| y \| \}$.

```
Program Definition node2 (x y : A) : node :=
  Node2 x y (‖ x ‖ · ‖ y ‖ : v).
Program Definition node3 (x y z : A) : node :=
  Node3 x y z (‖ x ‖ · ‖ y ‖ · ‖ z ‖ : v).
```

Les obligations générées sont trivialement vraies, elles sont de la forme $x = x$. De façon correspondante, node_measure projette la mesure cachée. On a ici une projection implicite.

```
Program Definition node_measure n : v :=
  match n with Node2 _ _ s ⇒ s | Node3 _ _ _ s ⇒ s end.
```

On peut déclarer une instance globale (généralisée à la sortie de la section) pour la mesure d'un nœud.

```
Global Instance nodeMeasured : Measured v node := { measure := node_measure
}.
```

On peut toujours convertir un node en digit en oubliant la mesure.

```
Definition node_to_digit (n : node) : digit A :=
  match n with Node2 x y _ ⇒ Two x y | Node3 x y z _ ⇒ Three x y z end.
End Nodes.
```

Bien qu'il puisse sembler que la fonction node_measure est indépendante de la fonction measure, elle ne l'est pas. En effet le type de cette fonction après la clôture de la section devient : Π (m : Monoid v) (A : Type) (ms : Measured A), node → v

L'inductif node lui-même est paramétré par la fonction de mesure, donc toutes les opérations sur les nœuds la prennent comme argument implicite (à la manière d'une contrainte de classe sur un type de donnée en Haskell). On a donc gratuitement la propriété qu'on ne peut pas confondre deux objets de type node construits avec des mesures différentes sur le même type d'éléments puisque la mesure fait partie du type.

Si nous n'avions pas ajouté l'invariant au sein des constructeurs, nous n'aurions pas besoin de ce paramètre, mais nous ne pourrions pas prouver grand chose sur les mesures qui seraient des objets arbitraires de type v. On pourrait définir un prédicat inductif sur les node assurant que la mesure d'un nœud a bien été construite en utilisant la mesure de façon cohérente, mais on aurait à montrer la préservation de ce prédicat pour toutes les fonctions manipulant directement ou indirectement des nœuds. Ici, on garantit cette propriété par le typage et le système nous donnera automatiquement les obligations à montrer lorsque l'invariant pourrait être cassé.

11.3.4 Finger Trees

Avant de présenter la définition de fingertree en Coq, on rappelle le type Haskell original et on va justifier pourquoi une traduction directe serait insatisfaisante. Le type des Finger Trees mesurés est défini comme suit :

```
data FingerTree v a = Empty | Single a
    | Deep v (Digit a) (FingerTree v (Node v a)) (Digit a)
```

La figure 11.2 représente un exemple de FingerTree mesuré.

On pourrait directement définir la structure de Finger Tree en Coq en traduisant la définition Haskell. Seulement, en faisant cela on pourrait causer un certain nombre de difficultés. Premièrement, on aurait le même problème décrit auparavant d'identification de structures qui ont été construites par des mesures différentes si l'on ne paramétrait pas le type par celle-ci. Bien sûr, le fait de paramétrer par la mesure force à réinstancier celle-ci au nœud Deep puisque le FingerTree v (Node v α) sous-jacent attend une mesure sur des objets de type Node v α plutôt que α.

Un fingertree est donc paramétré par un type A et une fonction de mesure sur ce type. Chaque objet de type fingertree est aussi indexé par sa propre mesure :

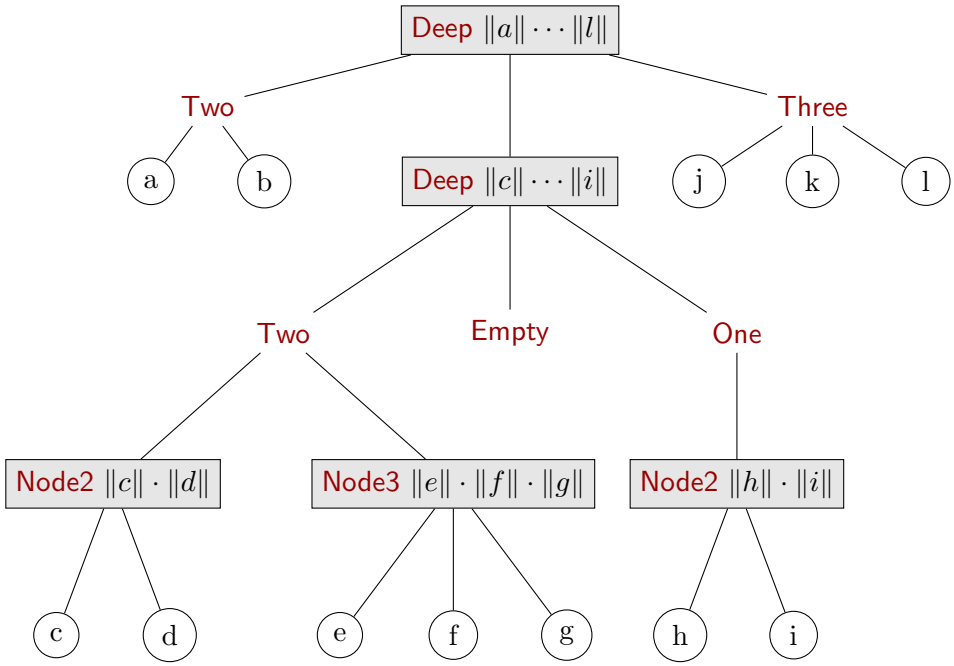


FIG. 11.2: Un exemple de Finger Tree. Au premier niveau on a un nœud Deep avec un 2-doigt de a à gauche et un 3-doigt à droite. L'arbre central est un FingerTree v (Node v a) consistant en un nœud Deep avec un arbre central vide. Il a un 2-doigt de Node v a à sa gauche et un 1-doigt à droite. Les nœuds ronds représentent les éléments de type a et les boîtes grisées indiquent les endroits où les mesures sont stockées.

- Empty construit l'arbre vide de mesure ϵ ;
- Single x construit l'arbre singleton de mesure $\|x\|$;
- Deep pr ms m sf construit un arbre de préfixe pr, sous-arbre m de mesure ms et suffixe sf. Sa mesure est calculée en combinant les mesures de ses sous-structures de gauche à droite. L'argument ms sera implicite lorsqu'on construira des nœuds Deep puisqu'on peut l'inférer à partir du type de m. On place cette mesure cachée juste avant l'arbre du milieu contrairement à la version originale où la mesure est le premier composant et stocke la mesure de l'*ensemble* de l'arbre. Pour nous, la mesure de l'arbre complet est donnée par l'index. On présente la définition sous forme de règles d'inférence pour une meilleure lisibilité, en omettant les paramètres A et measure dans les prémisses :

$$\frac{}{\text{Empty} : \text{fingertree}\ A\ \mathit{measure}\ \varepsilon}$$

$$\frac{x : A}{\text{Single}\ x : \text{fingertree}\ A\ \mathit{measure}\ (\mathit{measure}\ x)}$$

$$\frac{\begin{array}{c} pr, sf : \mathit{digit}\ A, ms : v \\ m : \text{fingertree}\ (\mathit{node}\ \mathit{measure}\ A)\ \text{measure}\ ms \end{array}}{\text{Deep}\ pr\ ms\ m\ sf : \text{fingertree}\ \ A\ \mathit{measure}\ (\text{measure}\ pr \cdot ms \cdot \text{measure}\ sf)}$$

Cette famille inductive est indexée par des valeurs de type v. Une simple observation nous a conduit à ce type dépendant : nous voulons avoir l'invariant que la mesure cachée sur le nœud Deep est bien celle du sous-arbre. Pour cela on doit avoir une façon de référer à cette mesure au moment même de la définition de l'arbre, alors qu'on ne peut pas écrire

une fonction récursive (polymorphe) sur l'arbre, à moins de se placer dans le cadre des définitions inductives-récursives (Dybjer 2000). La mesure de l'arbre fait ici partie de son type. On va donc faire apparaître les invariants sur les mesures contenues dans nos arbres directement dans les types des fonctions sur les fingertree.

Ajouter un élément On peut ajouter un élément a à gauche d'un arbre t de mesure s pour obtenir un arbre de mesure measure $a \cdot s$. Du fait de la récursion polymorphe, toutes nos fonctions récursives vont maintenant avoir des arguments A et measure puisqu'ils sont de *vrais* arguments qui changent lors des appels récursifs. Si t est vide, un singleton ou un arbre avec un préfixe gauche non plein, on pousse simplement a à la position la plus à gauche de l'arbre. Sinon, on doit réorganiser l'arbre pour faire de l'espace à gauche pour a. Cela requiert une récursion polymorphe pour ajouter un élément node3 measure c d e à gauche de t' : fingertree v (node_measure measure) st'.

```
Program Fixpoint add_left `{m :! Measured v A} (a : A) (s : v)
  (t : fingertree A s) {struct t} : fingertree A (measure a · s) :=
  match t in fingertree _ s return @fingertree A m (measure a · s) with
    | Empty ⇒ Single a
    | Single b ⇒ Deep (One a) Empty (One b)
    | Deep pr st' t' sf ⇒
      match pr with
        | Four b c d e ⇒ let sub := add_left (node3 c d e) t' in
          Deep (Two a b) sub sf
        | x ⇒ Deep (add_digit_left a x) t' sf
      end
  end.
```

La première expression `match` utilise l'élimination dépendante. Le sens des annotations est qu'à partir d'un fingertree d'une mesure s particulière, chaque branche doit construire un fingertree de mesure measure $a \cdot s$ ou s sera substitué par la mesure correspondant au motif de la branche. Par exemple, dans la première branche on doit construire un objet de type fingertree measure (measure $a \cdot \epsilon$). Seulement, la branche droite Single a a le type fingertree measure (measure a), on utilise donc la règle d'équivalence sur les familles inductives présentée figure 4.1 pour coercer l'objet dans le type attendu. L'application de cette règle génère une obligation $\vdash$ measure $a \bullet \varepsilon =$ measure a, facilement résolue en utilisant la loi d'identité à droite du monoïde. Les clauses `in` et `return` sont en général obligatoires en Coq à cause de l'indécidabilité de l'unification d'ordre-supérieur (il faudrait trouver le type le plus général unifiant les types des branches, en inférant les dépendances avec l'objet filtré). On peut cependant les omettre dans Russell, auquel cas la substitution utilisée par l'élimination dépendante est remplacée par du raisonnement équationnel. Si nous avions omis ces clauses, on aurait eu l'équation $s = \epsilon$ dans le contexte de la première branche et donc l'obligation $H : s = \epsilon \vdash$ measure $a \bullet s =$ measure a. Celle-ci serait résolue par substitution de s par ϵ dans le but puis application de l'identité à droite ; on a juste retardé la substitution.

Le filtrage imbriqué sur le préfixe de l'arbre utilise un *alias* x pour capturer les préfixes qui ne sont pas Four et applique la fonction "partielle" add_digit_left définie précédemment. On a une application de la règle d'équivalence sur les sous-ensembles ici, qui génère une obligation de montrer que x n'est pas plein. Celle ci peut être résolue parce que l'algorithme de compilation du filtrage ajoute une hypothèse $\forall b\ c\ d\ e, x \neq \mathit{Four}\ b\ c\ d\ e$ dans le contexte. On enrichit les contextes de typage des branches par ce genre d'inégalités lorsque leurs motifs sont en intersection avec des motifs précédents.

La préservation de la mesure est une propriété essentielle de cette fonction. Pour le voir, prenons l'instantiation des Finger Trees par le monoïde des listes avec la concaténation. On peut vérifier ici qu'ajouter un élément à gauche de l'arbre insère bien la mesure de l'élément devant la mesure de l'arbre. Cela revient donc à ajouter un élément en tête de la liste des éléments de l'arbre original avec ce monoïde des listes. Pour chacune des définitions suivantes, cette correspondance avec l'interprétation des listes aura toujours un sens évident. On définit la fonction add_right de la même façon.

Un exemple plus simple d'élimination dépendante nous est donné par la fonction de conversion digit_to_tree, qui transforme un "doigt" en un arbre de même mesure. Notons qu'ici on omet les annotations. Les versions récentes de Coq sont capables dans le cas où l'on filtre une variable qui apparaît dans le type de retour d'inférer automatiquement le prédicat d'élimination dépendante. On utilise aussi cette optimisation dans Program, qui permet d'utiliser la substitution le plus possible mais toujours en conservant les équations dans chaque branche.

```
Program Fixpoint digit_to_tree `{ma :! Measured v A} (d : digit A) {struct d} :
  fingertree A (measure d) :=
  match d with
    | One x ⇒ Single x
    | Two x y ⇒ Deep (One x) Empty (One y)
    | Three x y z ⇒ Deep (Two x y) Empty (One z)
    | Four x y z w ⇒ Deep (Two x y) Empty (Two z w)
  end.
```

11.3.5 La vue à gauche d'un Finger Tree

On va maintenant construire des vues (Wadler 1985; McBride et McKinna 2004) sur les Finger Trees qui permettent de décomposer un arbre en son premier élément (à gauche ou à droite) puis le reste de l'arbre. Cela permet de s'abstraire de l'implémentation et donner une interface similaire aux listes au type de donnée fingertree. On peut ainsi écrire des fonctions récursives sur les Finger Trees sans avoir à s'occuper des détails compliqués de la structure (voir par exemple la définition de *merge*).

Le type inductif de la vue à gauche View_L est un peu moins polymorphe que les autres, puisqu'il n'a pas besoin de contenir la fonction de mesure que les vues ignorent. En revanche on stocke la mesure s du reste de l'arbre dans le constructeur cons_L (s sera implicite). On abstrait donc View_L par le type de la séquence *seq* indexé par un objet de type v. On l'instanciera par fingertree A.

```
Inductive View_L {A : Type} {seq : v → Type} : Type :=
| nil_L : View_L
| cons_L : A → ∀ {s}, seq s → View_L.

Implicit Arguments View_L [ ].
```

Une telle vue sera produite par la fonction view_L, par récursion structurelle (polymorphe) sur le fingertree. On peut facilement utiliser la fonction partielle digit_tail qui n'accepte que les digit non-singleton et l'on n'a besoin d'aucune annotation de type à l'appel récursif de view_L. Notons qu'on utilise une *application partielle* de type (fingertree A) dans le type de retour, ce qui est parfaitement légal en Coq.

```
Program Fixpoint view_L `{ma :! Measured v A} (s : v) (t : fingertree A s) :
  View_L A (fingertree A) :=
  match t with
    | Empty ⇒ nil_L
```

```
  | Single x ⇒ cons_L x Empty
  | Deep pr st' t' sf ⇒
    match pr with
      | One x ⇒
        match view_L t' with
          | nil_L ⇒ cons_L x (digit_to_tree sf)
          | cons_L a st' t' ⇒ cons_L x (Deep (node_to_digit a) t' sf)
        end
      | y ⇒ cons_L (digit_head y) (Deep (digit_tail y) t' sf)
    end
end.
```

On peut montrer que view_L préserve la mesure de l'arbre. Si nous avions indexé View_L par la mesure de l'arbre en entrée, ces lemmes de génération seraient apparus comme obligations dans la définition de view_L.

```
Lemma view_L_nil : Π '{ma :! Measured v A} s (t : fingertree A s),
  view_L t = nil_L → s = ϵ.

Lemma view_L_cons : Π '{ma :! Measured v A} s (t : fingertree A s) x st' t',
  view_L t = cons_L x (s :=st') t' → s = measure x · st'.
```

Problèmes de dépendance

Nos vues sont utiles pour construire une interface de haut-niveau sur les Finger Trees, mais dans leur état courant elles sont très limitées puisqu'on ne peut écrire que des fonctions non-récursives sur ces vues. En effet, on ne peut pas convaincre Coq qu'une fonction définie par récursion sur la vue d'un arbre est aussi valide que par récursion sur l'arbre lui-même, à cause de la condition de garde. Pour ce faire, nous avons besoin d'une mesure sur le type fingertree, par exemple leur nombre d'éléments donné par la fonction tree_size. On peut dès lors créer une mesure trivialement sur les vues View_L_size. Les définitions de tree_size et donc View_L_size sont généralisées pour toute fonction de taille à cause de la récursion polymorphe.

```
Definition View_L_size' '{ma :! Measured v A} (size : A → nat)
  (view : View_L A (fingertree A)) :=
  match view with
    | nil_L ⇒ 0
    | cons_L x st' t' ⇒ size x + tree_size' size t'
  end.

Definition View_L_size '{ma :! Measured v A} (v : View_L A (fingertree A)) :=
  View_L_size' (λ _, 1) v.
```

Il n'y a plus qu'à montrer que pour tout arbre t, view_L t retourne une vue de taille tree_size t pour prouver qu'un appel récursif sur la queue d'une vue est correct (c.f. §11.4.2).

Seulement, faire cette preuve n'est pas si facile parce que view_L manipule des objets à type dépendant et raisonner sur ceux-ci est assez délicat. Une difficulté essentielle est que l'égalité de Leibniz n'est pas adaptée pour comparer des objets dans des types dépendants puisqu'ils peuvent être comparables mais dans des types différents. Par exemple la proposition sur les vecteurs $vnil = vcons\ x\ n\ v$ n'est pas bien typée puisque $vnil$ est de type $vector\ 0$ et $vcons\ x\ n\ v$ de type $vector\ (S\ n)$. Ces deux types n'étant pas convertibles, on ne peut même pas typer cet énoncé.

Dans notre cas, on veut montrer qu'un arbre t arbitraire de mesure s ayant pour vue nil_L est forcément l'arbre vide Empty, mais ces deux termes n'ont pas le même type. On applique la solution proposée par (McBride 1999) en utilisant une égalité hétérogène : on va alors pouvoir prouver que les termes doivent être dans le même type et retrouver l'égalité de Leibniz ensuite. L'inductif JMeq definit l'égalité hétérogène (denotée par $\simeq$) en Coq, de type : $\forall\ A\ (a : A)\ B\ (b : B)$, Type. Cette égalité permet de comparer des objets qui ne sont pas du même type (ou pas encore, avant simplifications dues à des réécritures, des éliminations...). Son unique constructeur est JMeq_refl, de type $\Pi\ A\ a$, JMeq $A\ a\ A\ a$. L'intérêt de cette notion d'égalité est de retarder le moment où l'on doit montrer que deux types sont égaux et donc que deux objets sont comparables. Lorsqu'on arrive à raffiner une hypothèse d'égalité dépendante pour que les deux types coincident, on peut appliquer l'axiome *JMeq_eq* de type $\Pi\ A\ x\ y$, JMeq $A\ x\ A\ y \to x = y$ pour récupérer une égalité de Leibniz usuelle entre les deux objets. L'axiome *JMeq_eq* est équivalent à l'axiome K de Streicher (Hofmann et Streicher 1998).

Dans le premier lemme, on compare t de mesure s avec Empty de mesure ϵ ; clairement, si l'on remplace JMeq par l'égalité de Leibniz on aura une erreur de typage.

```
Lemma view_L_nil_JMeq_Empty : ∀ `{ma :! Measured v A} s
  (t : fingertree A s), view_L t = nil_L → JMeq t Empty.
```

Une fois qu'on a montré l'égalité pour un index général s, on peut l'instancier sur un index particulier, ici ϵ. Sachant que t est maintenant de mesure ϵ, on peut utiliser l'égalité de Leibniz entre t et Empty.

```
Lemma view_L_nil_Empty : ∀ `{ma :! Measured v A}
  (t : fingertree A ϵ), view_L t = nil_L → t = Empty.
```

Ces lemmes auxiliaires sur nil_L et Empty vont nous permettre de construire la mesure. En effet, ils sont nécessaires pour prouver le lemme suivant :

```
Section view_L_measure.
  Lemma view_L_size : ∀ `{ma :! Measured v A} (s : v) (t : fingertree A s),
    View_L_size (view_L t) = tree_size t.
```

Cela nous donne une mesure décroissante sur les résultats de view_L. On l'utilisera plus tard, lorsqu'on programmera des instances.

```
  Lemma view_L_size_measure : ∀ `{ma :! Measured v A} (s : v)
    (t : fingertree A s) x st' (t' : fingertree A st'),
    view_L t = cons_L x t' → tree_size t' < tree_size t.
End view_L_measure.
```

On peut aussi définir le "*smart constructor*" deep_L, qui réarrange un arbre lorsqu'on lui donne un digit de préfixe potentiel et inversement pour deep_R. C'est une version dépendante de la fonction interne pour le cas Deep de view_L, qui est utilisée lorsqu'on découpe des Finger Trees. On peut noter que la spécification bénéficie de la surcharge qui permet de factoriser toutes les instances de measure sur les arbres, les doigts et les doigts optionnels dans ce cas.

```
Program Definition deep_L `{ma :! Measured v A}
  (d : option (digit A)) (s : v) (mid : fingertree (node A) s)
  (sf : digit A) : fingertree A (measure d · s · measure sf) :=
  match d with
    | Some pr ⇒ Deep pr mid sf
    | None ⇒
      match view_L mid with
```

```
      | nil_L ⇒ digit_to_tree sf
      | cons_L a sm' m' ⇒ Deep (node_to_digit a) m' sf
    end
  end.
```

De retour à la normale

À l'aide de ces vues, on peut maintenant implémenter facilement les opérations de "*deque*" sur le type fingertree. On n'a pas besoin de récursion ici, on peut donc définir ces opérations sur un type A, une mesure measure et un index s fixés dans une section.

```
Section View.
  Context `{ma :! Measured v A} (s : v).
```

On définit un prédicat isEmpty pour les fonctions définies seulement sur les arbres non vides. Cette fois-ci, on ne filtre pas directement sur l'objet mais sur la vue pour maintenir l'abstraction vis-à-vis de l'implémentation.

```
  Definition isEmpty (t : fingertree A s) :=
    match view_L t with nil_L ⇒ True | _ ⇒ False end.
```

On peut évidemment décider si un arbre est vide ou non.

```
  Definition isEmpty_dec (t : fingertree A s) : { isEmpty t } + { ¬ isEmpty t }.
```

Les opérations évidentes sont définissables, on montre la fonction liat duale de tail. Ici on retourne une mesure accompagnée d'un fingertree dans une paire dépendante de type $\{s : v \;\&\; \text{fingertree } A\ s\}$, qui correspond bien à la vue de la mesure comme une donnée et pas seulement un indice qui raffine le type de données. On note la construction d'une telle paire d'un arbre t avec sa mesure m par $m :| t$, qui se lit "m mesure t".

```
  Program Definition liat (t : fingertree A s | ¬ isEmpty t)
    : { s' : v & fingertree A s' } :=
    match view_R t with
      | nil_R ⇒ ! | cons_R st' t' last ⇒ st' :| t'
    end.
End View.
```

11.3.6 Concaténation et découpage dépendants.

On peut aussi définir la concaténation avec un type dépendant exprimant clairement sa spécification. L'implémentation est la même que celle d'Hinze & Paterson, excepté qu'on a prouvé les 100 obligations générées par Program concernant les mesures. Les cinq fonctions mutuellement récursives cachées ici qui définissent app ont la particularité d'être assez longues (près de 200 lignes au complet) puisqu'elles implémentent une spécialisation de la concaténation comme nous l'avons décrit plus haut (§11.2.2). On vérifie ici statiquement que la définition est correcte : le type indique qu'on a bien la propriété évidente sur les Finger Trees que la concaténation de deux arbres donne un arbre dont la mesure est la concaténation des mesures des deux arbres.

```
Definition app `{ma :! Measured v A}
  (xs : v) (x : fingertree A xs) (ys : v) (y : fingertree A ys) :
  fingertree A (xs · ys) := appendTree0 x y.

Notation " x +:+ y " := (app x y) (at level 20).
```

La dernière opération, qui a certainement la plus intéressante spéficication du développement, est le découpage d'un arbre par un prédicat sur sa mesure. On commence par découper un nœud.

```
Section Nodes.
  Context `{ma :! Measured v A}.
  Variables (p : v → bool) (i : v).
```

On découpe un nœud n par un prédicat p en cherchant où celui-ci s'évalue à true, en commencant par la mesure initiale i et en accumulant les mesures des sous-objets de gauche à droite. On a simplement copié l'invariant donné dans Hinze et Paterson (2006) sans rien y changer. On a simplement ajouté une propriété sur les mesures qui généralise l'équation sur *to_list*. Le code est aussi une traduction directe du code Haskell en Russell, on utilise juste un produit cartésien au lieu d'un nouveau type inductif `split`. On a aussi défini une opération split_digit avec une spécification similaire.

```
  Program Definition split_node (n : node A) :
  { (l, x, r) : option (digit A) × A × option (digit A) |
    let ls := measure l in let rs := measure r in
    measure n = ls · ‖ x ‖ · rs ∧
    node_to_list n = option_digit_to_list l ++ [x] ++ option_digit_to_list r ∧
    (l = None ∨ p (i · ls) = false) ∧
    (r = None ∨ p (i · ls · ‖ x ‖) = true)} :=
  match n with
    | Node2 x y _ ⇒
      let i' := i · ‖ x ‖ in
      if dec (p i') then (None, x, Some (One y))
      else (Some (One x), y, None)
    | Node3 x y z _ ⇒
      let i' := i · ‖ x ‖ in
      if dec (p i') then (None, x, Some (Two y z))
      else
        let i'':= i' · ‖ y ‖ in
        if dec (p i'') then
          (Some (One x), y, Some (One z))
        else
          (Some (Two x y), z, None)
  end.
End Nodes.
```

Le cas le plus intéressant est celui des arbres. Plutôt que de retourner un tuple raffiné dans un type sous-ensemble, on définit cette fois une famille inductive tree_split qui capture l'invariant qu'un découpage est une décomposition d'un fingertree préservant sa mesure. On ajoute aussi les invariants sur les arbres de gauche et droite vis-à-vis du prédicat. Ils pourront être utilisés par les clients pour prouver des propriétés sur leur code. L'inductif tree_split est en fait une *vue* dépendante d'un arbre. C'est une particularité de la programmation avec types dépendants et de ce développement en particulier : on dérive non seulement du code réutilisable mais aussi des preuves réutilisables en utilisant des types riches. En effet, la fonction de découpage peut être vue comme un combinateur de preuves, relevant une propriété sur une mesure et un monoïde en une propriété sur les mots de ce monoïde représentés par les fingertree.

```
Section Trees.
  Variable p : v → bool.
```

```
Inductive tree_split `{ma :! Measured v A} (i : v) : v → Type :=
  mkTreeSplit : ∀ (xsm : v)
    (xs : fingertree A xsm | isEmpty xs ∨ p (i · xsm) = false)
    (x : A) (ysm : v)
    (ys : fingertree A ysm | isEmpty ys ∨ p (i · xsm · measure x) = true),
    tree_split i (xsm · measure x · ysm).
```

Ce type inductif combine des types sous-ensemble et dépendants, mais nous pouvons toujours écrire notre code comme d'habitude. Approximativement 100 lignes de preuves sont nécessaires pour décharger les obligations générées par cette définition.

```
Program Fixpoint split_tree [ m :! Measured v A ]
(i s : v) (t : fingertree measure s | ¬ isEmpty t) : tree_split measure i s := ···
```

Ceci conclut notre implémentation des Finger Trees dépendants avec Program. Pour résumer, nous avons prouvé que :

- Toutes les fonctions sont totales une fois annotées par leurs préconditions.
- Toutes les fonctions respectent la sémantique des mesures qui est rendue explicite dans les types alors qu'elle était auparavant implicite dans le code.
- Toutes les fonctions respectent les invariants donnés dans le papier original par Hinze & Paterson.

Le fichier complet que nous venons de parcourir comprend 600 lignes de spécification (définitions et énoncés de lemmes) et 450 lignes de preuves réalisées par des tactiques. En comparaison, le fichier source Haskell donné par Hinze & Paterson fait 650 lignes incluant les commentaires. Cela montre que Russell est très proche d'un langage à portée générale malgré la verbosité et l'expressivité supplémentaire du système formel sous-jacent. On a bien avancé vers l'objectif de faire de Russell un langage de programmation utilisable en pratique. On peut aussi noter que la "charge de la preuve" n'est pas insurmontable, sachant que la plupart d'entre elles ont été résolues automatiquement à l'aide d'une tactique de normalisation des expressions monoïdales.

11.4 Instances

On peut maintenant instancier la structure de fingertree par différentes combinaisons de monoïdes et mesures pour obtenir des structures de plus haut niveau. Évidemment, on peut instancier les Finger Trees comme dans l'article original en utilisant une interface simplement typée et faiblement spécifiée et extraire ce code. On montre comment donner une interface non-dépendante à nos arbres et on l'instancie sur les séquences ordonnées section 11.4.1. On montrera ensuite (§11.4.3) comment utiliser l'interface dépendante directement pour dériver une implémentation des séquences à accès aléatoire certifiée.

11.4.1 Une interface non dépendante

Comme décrit précédemment, on peut agréger la mesure et le fingertree dans une paire dépendante pour donner une interface simplifiée aux Finger Trees dépendants.

```
Section FingerTree.
  Context `(ma : Measured v A).
  Definition FingerTree := { m : v & fingertree v A m }.
  Notation " ts :| t " := (existT _ ts t) (at level 20).
```

Cela nous donne un type FingerTree v {$mono$} A {ma}, qu'on peut comparer au type original FingerTree v a dans Haskell. On a ajouté l'implémentation *monoid* et la fonction de

mesure *ma* comme paramètres explicites alors qu'en Haskell ils sont passés implicitement à l'aide des classes de type à chaque fonction les manipulant.

On ne décrit pas le "repackaging" en détail, il s'agit just de décomposer un objet de type FingerTree, appeler la fonction appropriée sur les fingertree et recomposer la paire dépendante. En revanche un point important est que l'on peut exporter les vues sur les fingertree et retrouver la mesure associée :

```
Program Definition tree_size (s : FingerTree) : nat :=
  let '_ :| t := s in tree_size t.

Inductive left_view : Type :=
| nil_left : left_view | cons_left : A → FingerTree → left_view.

Lemma view_left_size : ∀ t x t', view_left t = cons_left x t' →
  tree_size t' < tree_size t.
```

On peut aussi refaire le découpage par rapport à un prédicat sur les FingerTree. On offre la même interface simplement typée qu'en Haskell en définissant la fonction split_with p x. Cette fonction découpe n'importe quel arbre en regardant tout d'abord s'il est vide et sinon en appelant la fonction fortement spécifiée `split` avec un accumulateur vide ϵ. On crée alors une paire avec l'arbre à gauche et l'arbre droit auquel on ajoute l'élément qui fait changer le prédicat de valeur. On oublie complètement les propriétés données par la fonction `split` sur le résultat.

```
Program Definition split_with (p : v → bool)
  (x : FingerTree) : FingerTree × FingerTree :=
  if is_empty_dec x then (empty, empty)
  else let (l, x, r) := split p ϵ x in (l, cons x r).

End FingerTree.
```

11.4.2 Séquences ordonnées

Un exemple d'utilisation de l'interface simplement typée est la construction de séquences ordonnées suivante. L'idée principale est d'utiliser comme mesure d'un élément l'élément lui-même et faire que le monoïde calcule l'élément le plus à droite de l'arbre. Alors, si les opérations maintiennent l'invariant que la séquence des éléments de l'arbre est ordonnée on a une séquence ordonnée ou l'élément maximal peut être récupéré en temps constant.

```
Module OrdSequence(AO : OrderedType).
```

On utilise des modules plutôt que des sections puisque cela nous permet d'instancier à "*compile-time*" ce qui est exactement ce dont nous avons besoin ici. Le système de modules de Coq est un surensemble de celui d'OCaml et l'extraction supporte la traduction de l'un à l'autre. On suppose donnée une implémentation d'un type ordonné AO.

```
Definition A : Type := AO.t.
```

On mesure un élément par lui même. On utilise un type option et l'arbre vide aura pour mesure None.

```
Module KM := KeyMonoid AO. Import KM.

Instance ord_measure : Measured key A :=
  { measure := Some }.
```

Le module *KeyMonoid* implémente le monoïde (None, keyop) sur le type option A, où l'opération keyop est définie par :

```
keyop x None          = x
keyop _ (Some _ as y) = y
```

Clairement, cela va donner l'élément le plus à droite de la structure s'il existe. Le module définit aussi une fonction key_gt qui "lifte" l'ordre sur le type ordonné vers le type du monoïde. On peut déclarer le type des séquences ordonnées comme spécialisation de FingerTree sur le monoïde et la mesure qu'on vient de présenter.

```
Definition OrdSeq := FingerTree ord_measure.
```

On peut prendre l'élément maximum en temps constant en récupérant la mesure à la racine du Finger Tree.

```
Definition max (x : OrdSeq) := tree_measure x.

Program Definition partition (k :A) (xs :OrdSeq) : OrdSeq × OrdSeq :=
  split_with (key_ltb (Some k)) xs.
```

On ne présente pas en détail les autres opérations définies dans l'article original qui permettent de partitionner une séquence (partition) ou d'insérer/supprimer un élément, elles ne sont pas plus compliquées à programmer. On se penche plus précisément sur la seule difficulté liée à la terminaison de ce développement qui apparaît lorsqu'on veut coder la fonction merge qui mélange deux séquences ordonnées. Cette fonction utilise view_left sur les séquences et fonctionne donc par récurrence sur la vue. Il est à noter que l'argument de terminaison est de plus compliqué par le fait qu'on interchange les arguments pour rendre la fonction adaptative. Elle dégénère à une concaténation dès que possible, donnant une complexité très bonne en pratique. On peut utiliser la mesure pair_size définie ci-dessous pour définir la fonction par récursion bien fondée sur la paire des séquences. Les obligations générées peuvent être résolues en utilisant le lemme view_left_size défini plus haut.

```
Definition pair_size (x : OrdSeq × OrdSeq) :=
  let (l, r) := x in tree_size l + tree_size r.

Program Fixpoint merge (x : OrdSeq × OrdSeq) {measure pair_size} : OrdSeq :=
  let '(xs, ys) := x in
  match view_left xs with
    | nil_left ⇒ ys
    | cons_left x xs' ⇒
      let '(l, r) := split_with (fun y ⇒ key_gt y (measure x)) ys
      in cat l (cons x (merge (r, xs')))
  end.
```

On obtient donc une implémentation des séquences ordonnées bâtie sur le code des *fingertree* qu'on peut extraire vers OCaml. Seulement, on s'appuie ici sur des invariants implicites de la structure et rien n'empêche de créer des objets de type OrdSeq non ordonnés. On pourrait bien sûr faire un raffinement du type forçant l'arbre à être ordonné, mais l'on aurait besoin pour cela de raisonner sur le code de *fingertree* pour montrer la préservation de cette propriété pour les différentes opérations que nous utilisons. On va plutôt profiter du pouvoir de spécification qui nous est donné par le monoïde et l'indexation des Finger Trees dépendants pour obtenir une structure certifiée modulairement, sans avoir à raisonner sur le code.

11.4.3 Séquences dépendantes

On va maintenant définir des séquences à accès aléatoire comme spécialisation des fingertree. Cette structure permet d'ajouter un élément à n'importe quel bout d'une séquence en temps constant amorti, de concaténer (app) et découper (`split`) des séquences en temps logarithmique et d'accéder (get) ou modifier (set) un élément de la séquence en temps logarithmique en la taille de la séquence aussi. On va créer une implémentation certifiée de cette structure en montrant que ces opérations respectent une spécification fonctionnelle très forte. Cette méthode diffère significativement de l'habituelle séparation entre opérations et preuve des invariants sur ces opérations. Ici, la préservation des propriétés va être prouvée simultanément à la définition des opérations elles-même.

Structure

On définit les séquences indexées par des naturels sur un type d'éléments A. On pourrait généraliser à d'autres représentations des indices, mais pour plus de simplicité on utilise ici les entiers de Peano.

```
Section DependentSequence.
  Variable A : Type.
```

Tout d'abord, une définition utile : le type below i (isomorphe au type bien connu *Fin* des ensembles finis) représente les entiers naturels plus petits que i, donc below 0 est un type vide.

```
  Definition below i := { x : nat | x < i }.
  Lemma not_below_0 : below 0 → False.
```

On utilise les entiers pour mesurer les séquences par leur longueur. Notre mesure est un peu particulière puisqu'elle contient aussi une fonction donnant la map *réalisée* par le Finger Tree. Cette fonction totale associe à toute position de la séquence une valeur de type A. Elle est utilisée seulement pour la spécification et pourrait être éliminée du code généré en utilisant une analyse de code mort ou même effacée à l'extraction en utilisant un système de type plus fin (c.f. §11.5).

```
  Definition v := { i : nat & (below i → A) }.
  Notation " x ≻ f " := (existT (fun i ⇒ below i → A) x f : v) (at level 80, no
associativity).
```

On définit une notation pour nos mesures : un objet $i \succ m$ est une paire dépendante composée d'une taille i et une "*map*" des naturels plus petits que i vers A. L'élément neutre ϵ représente la mesure d'une séquence vide. Comme elle ne contient aucun élément, aucune valeur n'est retournée par la fonction. On a une obligation de montrer que dans un contexte avec l'hypothèse below 0, on peut prouver False : on utilise juste le lemme not_below_0.

```
  Program Definition ε : v := 0 ≻ (fun _ ⇒ !).
```

Concaténer deux séquences requiert un peu de réindexation : la nouvelle map est formée par la projection du nouvel index sur l'une ou l'autre des maps.

```
  Program Definition append (xs ys : v) : v :=
    let (n, fx) := xs in let (m, fy) := ys in
    (n + m) ≻ (fun i ⇒ if less_than i n then fx i else fy (i - n)).
```

On peut construire le monoïde seqMonoid à partir de ces opérations. On saute les preuves qui sont relativement faciles. Il est juste à noter qu'on utilise l'axiome d'extensionalité des fonctions lorsqu'on a à comparer les fonctions de représentation des séquences

ainsi que l'indifférence aux preuves pour montrer que deux objets de type below i sont égaux si et seulement si leurs premières projections sont égales.

```
Program Instance seqMonoid : Monoid v :=
  { mempty := ε ; mappend := append }.
```

La mesure d'un singleton x est la map constante renvoyant x.

```
Definition single (x : A) : v := 1 ≻ const x.
Instance seq_measure : Measured v A := { measure := single }.
```

On définit l'abbréviation seq pour notre type de séquences. C'est une spécialisation de fingertree avec le monoïde seqMonoid et la mesure single définie ci-dessus.

```
Definition seq (x : v) := fingertree (v :=v) (A :=A) x.
```

Un objet de type seq $(i \succ m)$ est donc un Finger Tree représentant une séquence d'objets de longueur i donnée par la fonction m. On obtient trivialement la séquence vide et la séquence singleton.

```
Definition empty : seq ε := Empty.
Definition singleton x : seq (single x) := Single x.
```

Opérations

On peut prendre la longueur de la séquence en temps constant puisqu'elle fait partie de la mesure.

```
Program Definition length (s : v) (x : seq s) : nat :=
  let 'size ≻ _ := s in size.
```

Le constructeur make n x crée une séquence de longueur n avec la valeur x dans chaque cellule. Les obligations générées nous demandent de prouver par exemple que si make n x est de type seq $(n \succ \texttt{fun \_} \Rightarrow x)$ alors add_left x (make n x) est de type seq (S $n \succ \texttt{fun \_} \Rightarrow x$). On a la préservation de la sémantique de make directement par typage ici : tous les éléments de la séquence de longueur i contiennent x.

```
Program Fixpoint make (i : nat) (x : A) { struct i } :
  seq (i ≻ (fun _ ⇒ x)) :=
  match i with
    | 0 ⇒ empty
    | S n ⇒ add_left x (make n x)
  end.
```

On définit maintenant la fonction fortement spécifiée get qui retourne le $j$ème élément d'une séquence x de longueur i. On assure qu'aucun accès en dehors des bornes n'est possible puisque j est de type below i et on assure que l'élément retourné est bien m j. Le prédicat booléen lt_idx i x teste si un index i est inférieur ou égal à la première composante d'une mesure x. Pour récupérer un élément, on découpe donc l'arbre au $j$ème élément puisque chaque élément a pour mesure 1 et que l'accumulation additionne les mesures.

```
Program Definition get i m (x : seq (i ≻ m)) (j : below i)
   : { value : A | value = m j } :=
  let 'mkTreeSplit ls l x rs r := split_tree (lt_idx j) ε x in x.
```

Notons qu'on n'utilise pas m dans le code, seulement dans le type, on aurait sinon une implémentation triviale. Encore une fois, on génère des obligations pour montrer que le code respecte bien la spécification. Il est crucial que la fonction split_tree renvoie un objet

de type *tree_split* incluant des preuves reliant l'arbre donné en entrée aux résultats pour pouvoir résoudre ces obligations. Comme toujours, le code reste simple et compact. En revanche, on passe près de 50 lignes de preuves nécessaires pour décharger les obligations.

Le compagnon naturel de get est la fonction set qui étant donnée une séquence x de longueur i met sa $j$ème cellule à *value*. On a pour précondition que j est dans les bornes et comme postcondition que la "map" représentée par la séquence résultat est partout égale à l'ancienne séquence excepté en j où elle vaut désormais *value*. La fonction dénotée par =n décide l'égalité sur les naturels. Ici on découpe toujours l'arbre en j mais on ignore l'élément associé et on reconstruit un arbre en utilisant la nouvelle valeur.

```
Program Definition set i (m : below i → A) (x : seq (i ≻ m))
  (j : below i) (value : A) :
  seq (i ≻ (fun idx ⇒ if idx =n j then value else m idx)) :=
  let 'mkTreeSplit ls l _ rs r := split_tree (lt_idx j) ε x in
    add_right l value +:+ r.
```

Finalement, on définit le découpage d'une séquence en deux. Cela requiert deux projections dans les "*map*" correspondantes qui font une réindexation, d'où :

```
Program Definition split_l i (j : below i) (idx : below j) : below i := idx.
Program Definition split_r i (j : below i) (idx : below (i - j)) : below i := j +
idx.
```

On peut découper une séquence x à un index j en renvoyant deux séquences dont la deuxième contient l'élément à l'index j. Il faut approximativement une centaine de lignes de preuve pour décharger les trois obligations générées par Program, essentiellement sur l'arithmétique de la réindexation.

```
Program Definition split i m (x : seq (i ≻ m)) (j : below i) :
  seq (j ≻ (m ∘ split_l j)) × seq ((i - j) ≻ (m ∘ split_r j)) :=
  let 'mkTreeSplit ls l x rs r := split_tree (lt_idx j) ε x in
    (l, add_left x r).
```

Propriétés

Maintenant que nous avons défini nos opérations avec leurs invariants, il suffit d'utiliser l'information donnée par les types pour les relier. Ici on formalise comment get et set se comportent l'un vis-à-vis de l'autre. Les preuves utilisent uniquement les types et pas le code des deux opérations. On montre que nos séquences vérifient les deux axiomes des tableaux fonctionnels : si l'on récupère la valeur à l'index j qu'on vient de modifier on récupère la nouvelle valeur (get_set), sinon l'ancienne (get_set_diff).

```
Program Lemma get_set : ∀ i m (x : seq (i ≻ m))
  (j : below i) (value : A), value = get (set x j value) j.
```

Nous avons pu écrire ce lemme seulement parce qu'on a utilisé le "*type-checker*" de Russell, qui a automatiquement inséré une coercion à la droite de l'égalité pour aller d'un sous-ensemble sur A à A (le type implicite d'une égalité est fixé par son premier argument explicite, ici *value*). L'intégration transparente de Program dans Coq en tant qu'assistant de preuve et pas seulement langage de programmation est une amélioration par rapport aux systèmes antérieurs qui tentaient de faire de Coq un langage de programmation, comme la tactique `Program` de Parent (1995b). Il reste cependant du chemin à faire pour supporter les énoncés les plus naturels dans Russell. Dans la prochaine définition par exemple, on coerce explicitement les objets j et k du type below i vers leurs composantes de type nat et le résultat de get vers A. C'est dû au fait que l'égalité de Leibniz sur les objets de

type sous-ensemble ne fait du sens que sur leurs premières composantes, les témoins, tandis que la comparaison des preuves n'a pas vraiment de sens. Notre solution pragmatique est de laisser les utilisateurs insérer des "*casts*" pour obtenir le comportement souhaité, mais dans une théorie des types avec indifférence aux preuves on pourrait vraiment résoudre ce problème puisque la comparaison des preuves deviendrait triviale. On détaillera cette solution dans la section 11.6.1.

```
Program Lemma get_set_diff : ∀ i m (x : seq (i ≻ m))
  (j : below i) (value : A) (k : below i),
  (j : nat) ≠ k → (get x k : A) = get (set x j value) k.

End DependentSequence.
```

On a développé une implémentation certifiée des séquences à accès aléatoire de façon modulaire en moins de 400 lignes dont l'interface dépendante peut être utilisée pour écrire des programmes et prouver des propriétés sur ceux-ci en même temps.

11.5 Extraction

On peut extraire le code qu'on vient de certifier vers Haskell et OCaml (en passant outre le typeur pour la récursion polymorphe dans ce cas) et produire des versions certifiées des Finger Trees, des séquences ordonnées et des séquences à accès aléatoire. Nous avons la garantie que le code extrait aura la même complexité algorithmique que ce que l'on a écrit à l'origine à l'aide de Program. C'est un aspect important du développement dans Russell, puisque d'habitude quand on écrit du code avec des spécifications fortes en Coq on a plutôt tendance à utiliser le langage de tactique autant que le langage Gallina lui-même et cela peut avoir des effets dévastateurs sur le code extrait. Les tactiques peuvent en effet avoir des effets très profonds sur la forme des termes qui n'apparaissent qu'à l'extraction et sont alors parfois très difficiles à mitiger *a posteriori*, voir Letouzey (2004, chapitre 6) et Cruz-Filipe et Letouzey (2005). Notre méthode impose une distinction entre le code et la preuve très utile dans ce cadre.

Comme on peut s'y attendre, l'implémentation extraite des Finger Trees en Haskell a les mêmes performances que l'implémentation originale : c'est rigoureusement le même code. L'implémentation du module Data.Sequence dans la librairie standard d'Haskell est en fait une spécialisation de FingerTree sur un monoïde et une mesure particuliers (il s'agit de séquences à accès aléatoire), ce qui évite du "*boxing*" et les indirections dûes à l'abstraction sur ces structures. Nous avons développé une version à l'identique des Finger Trees utilisant cette fois-ci les modules de Coq pour la paramétrisation, ce qui nous donne un instanciation sans coût mais ne nous permet pas d'extraire vers Haskell. La version OCaml des Finger Trees a elle aussi de bonnes performances, comme on le verra dans la section suivante sur une instanciation de cette implémentation pour créer des "*ropes*" (ou cordes) (Boehm, Atkinson, et Plass 1995).

Malheureusement, la version extraite des séquences dépendantes (§11.4.3) ne donne pas de très bon résultats, puisque la mesure contient une fonction représentant la séquence complète des éléments et que celle-ci sera extraite même si elle n'apparaît jamais dans le code. Nous sommes dans un cas de figure ou seule une partie d'un index doit être stockée, contrairement à Brady, McBride, et McKinna (2003) par exemple. Une distinction plus fine que Prop/Type doit être trouvée pour distinguer le contenu algorithmique du contenu non-algorithmique même s'il n'est pas propositionnel. Les travaux entrepris dans Barras et Bernardo (2008), basés sur le Calcul des Constructions Implicite développé par Miquel (2001) devraient nous permettre d'exprimer ce genre de distinctions de phases.

Prendre la corde pour sauver Endo

Malgré les problèmes évoqués plus haut, il est possible d'extraire du code efficace des Finger Trees dépendants. Nous en avons fait l'essai en implémentant des "*ropes*" (Boehm et al. 1995) au-dessus des Finger Trees. La structure de corde est un arbre contenant des chaînes de caractères à taille fixe représentant des sous-chaînes de caractères à taille variable qu'on peut concaténer ou découper en temps logarithmique et où l'accès à un élément doit lui aussi être en temps logarithmique. Nous avons été amenés à développer cette structure pour résoudre le problème du concours de l'ICFP 2007 (ICFPC 2007). Le concours requiert en effet une implémentation des séquences très efficace car on doit s'en servir comme une sorte de ruban de machine de Turing réinscriptible qui doit contenir le code et les données d'un programme. On doit donc pouvoir faire des millions de concaténations et de découpages en sous-chaînes à la seconde.

Pour obtenir une structure de corde efficace, on peut simplement prendre nos Finger Trees et les instancier sur un type des sous-chaînes de caractères (un tuple contenant une `string` et deux entiers en OCaml). On utilise le monoïde des entiers avec l'addition pour pouvoir indexer les éléments dans l'arbre et on donne pour mesure d'une sous-chaîne sa longueur. On peut alors implémenter la concaténation et l'opération de découpage `substring` aisément à l'aide des opérations app et split. On utilise aussi le découpage pour coder l'accès à un élément. On pourrait aussi utiliser le découpage pour implémenter l'accès à un caractère. Évidemment, comme OCaml utilise l'appel par valeur, cela voudrait dire recréer inutilement un arbre représentant le préfixe avant l'élément accédé à chaque appel. C'est bien trop inefficace, on a donc implémenté une nouvelle primitive get similaire en tout point à split mais qui ne retourne que l'élément qui rend le prédicat vrai. On a aussi écrit la structure de corde comme un foncteur prenant en entrée une implémentation des indices pour permettre après extraction vers OCaml de l'instancier avec les entiers machine. En faisant ces deux changements, on obtient une structure de "*rope*" très compétitive, puisqu'on met environ 2 minutes à générer l'image test du concours, à comparer à la minute que prend une implémentation spécialisée des cordes développée par Jean-Christophe Filliâtre pour l'équipe des Caml Riders.

11.6 Discussion

11.6.1 "*Proof-Carrying Code*"

Comme on l'a vu, on finit toujours par mélanger preuves et code dans nos termes élaborés, puisque le typeur Coq a besoin de toute l'information pour vérifier les termes à n'importe quel moment. À la surface, on peut séparer les deux activités en utilisant Program, mais il y a des parties du système où l'on ne peut pas s'en sortir aussi bien. Le même problème apparaît lorsqu'on veut calculer avec ou faire des preuves sur les programmes générés. Lorsqu'on fait des preuves, on est confrontés à des termes contenant des preuves qu'on ne veut ni nommer ni manipuler d'aucune façon et lorsqu'on calcule (dans le cas où l'on est en appel par valeur en particulier) le système lui-même se perd en réductions inutiles dans des preuves ou est simplement bloqué par des preuves opaques. Évidemment, dans les deux cas les preuves apparaissent à des endroits où elles ne devraient pas être réduites puisqu'elles n'ont aucun contenu algorithmique et n'ont pas besoin d'être manipulées puisque leur structure n'a aucune importance, excepté pour les preuves correspondant aux coercions d'inductifs qui doivent se réduire vers le constructeur canonique. Cette idée est formalisée par l'indifférence aux preuves (PI), un principe extensionnel disant que deux preuves du même énoncé sont toujours égales. On a utilisé cet axiome dans le développe-

ment, mais on aimerait pouvoir avoir cette propriété gratuitement en intégrant ce principe dans la conversion (voir la discussion section 5.1.2).

11.6.2 Travaux connexes

D'autres certifications de structures d'arbres ont été faites dans les langages à types dépendants. Filliâtre et Letouzey (2004) certifient les Arbres binaires de recherche équilibrés (AVL) utilisés dans OCaml pour implémenter les ensembles en Coq. Le développement a été fait en définissant une interface dépendante tout d'abord puis en donnant une interface plus simple par dessus, comme dans notre cas. Le second auteur a adapté le développement qui utilisait au départ des tactiques pour utiliser Program (Letouzey 2008).

Dunfield (2007) utilise aussi un langage à types dépendants pour implémenter des arbres rouge-noir. Le système de type inclut des types union et intersection primitifs, des types raffinements et des types indexés qui peuvent être utilisés pour faire de la recherche de preuve au typage, dans la règle de conversion. Certaines des obligations que nous générons sont donc résolues par typage uniquement dans ce système. L'auteur indique que cela rend le système moins prédictible et les performances s'en ressentent. En effet les choix de branches laissés à l'utilisateur lorsqu'il résoud les obligations sont ici résolus par recherche de preuve avec "*backtracking*".

Finalement, Régis-Gianas a implémenté deux structures à base d'arbres dans le système Pangolin : les "*deques*" de Kaplan et Tarjan (1999) et les même arbres AVL dans sa thèse Régis-Gianas (2007, chapitre quinze). Cette dernière est presque complètement automatisée, un seul lemme nécessitant de l'induction étant prouvé en Coq, mais avec parfois des temps de recherche de preuve très longs (jusqu'à 10 minutes pour certaines obligations). En comparaison, notre développement prend à peu près 5 minutes pour être vérifié par Coq : les longues recherches seraient prohibitives pour le développement interactif, on utilise donc l'automatisation parcimonieusement.

11.7 Conclusion

La conclusion de cet exemple est qu'écrire les programmes en Russell n'est pas plus difficile qu'en Haskell. On a en sus un environnement complet pour prouver nos programmes avec un langage de spécification expressif et concis grâce aux classes. Program supporte déjà suffisament les constructions du langage pour spécifier des programmes riches et les certifier car il repose aussi sur l'outil mature qu'est Coq.

Nous avons vu aussi avec l'exemple des séquences dépendantes qu'il est très bénéfique de faire remonter les propriétés au travers des types dépendants des implémentations de bas-niveau aux structures de plus haut niveau. Program permet de faire ceci de façon organisée par la génération d'obligations, tout en conservant un haut niveau de fiabilité par la traduction vers Coq.

Quatrième partie

Conclusions et perspectives

Conclusion

Dans cette thèse, nous avons développé un ensemble d'outils théoriques et pratiques pour la programmation et la preuve avec types dépendants dans le système Coq. Ces contributions sont exportables directement dans d'autres théories des types.

Russell Dans un premier temps, nous avons développé un nouveau système de types nommé Russell correspondant à un Calcul des Constructions étendu par la notion de "*Predicate Subtyping*" venant du système PVS. On a développé en Coq une preuve des propriétés métathéoriques de base de ce système, et notamment une preuve d'autoréduction délicate. Cette preuve nous assure que la version algorithmique du système de typage est équivalente à sa version déclarative ou sémantique. On a ensuite démontré la décidabilité du système algorithmique, ce qui nous a donné un algorithme d'inférence et de typage pour Russell.

On a finalement décrit une traduction de Russell vers une variante du Calcul des Constructions enrichi par des variables existentielles et démontré que cette traduction était correcte : tout terme de Russell bien typé donne un terme de $\mathsf{CC}_?$ bien typé. La preuve très syntaxique de ce résultat nous a amené à découvrir des propriétés nécessaires restées implicites sur le système de coercion, notamment leur unicité et leur symétrie. De plus, on a vu que la théorie équationnelle du langage cible devait être étendue pour supporter l'indifférence aux preuves si l'on veut obtenir une traduction correcte.

Program On a implémenté cette traduction dans Coq sous le nom de Program. Cet outil permet de typer des termes de Russell et de les traduire dans Coq, tout en générant des obligations qui témoignent des coercions réalisées. Un système de gestion d'obligations permet alors de les résoudre pour obtenir une définition Coq classique. Program implémente des extensions de Russell pour gérer les types inductifs et les définitions récursives structurelles et bien fondées.

Nous avons testé cette implémentation sur un exemple non-trivial : les Finger Trees. On a montré comment les types sous-ensembles et les familles inductives permettaient de spécifier fortement l'implémentation originale écrite en Haskell. On a réussi à traduire ce code de façon transparente grâce à l'utilisation de Program et des classes de types. De plus, on a montré l'utilité de la notion de coercion sur les familles inductives pour dériver des propriétés de façon modulaire en implémentant des séquences à accès aléatoire certifiées au-dessus de notre variante des Finger Trees dépendants.

Classes de types dépendantes On s'est aussi intéressés à la gestion de la surcharge qui nous avait fait défaut durant le développement original des Finger Trees. En collaboration avec Nicolas Oury, nous avons développé un système de classes de type puissant

inspiré de Haskell. On a décrit une implémentation légère et très générale de ce système basée uniquement sur des constructions existantes du langage. Les classes sont naturellement étendues aux classes de types dépendantes qui permettent de faire de la résolution logique arbitraire au moment du typage. Cela facilite les développements structurés et la construction de procédures de décisions partielles sans compromettre les propriétés fortes du système. Les classes de type sont aussi un outil structurant pour les développements et permettent de mêler les définitions de l'utilisateur avec le système de tactiques.

On a construit une nouvelle tactique de réécriture généralisée puissante et bien intégrée à l'environnement en utilisant les classes. On a décomposé la tactique de réécriture en une partie de génération de contraintes et une partie de résolution s'appuyant directement sur la résolution des classes qui peut être contrôlée entièrement à partir du langage de tactiques. Notre tactique de réécriture est plus générale et plus performante que les solutions précédentes.

Conclusion Nous avons développé un ensemble d'outils au-dessus du Calcul des Constructions qui rendent son utilisation plus agréable sans restriction sur les constructions existantes du système et surtout sans sacrifier le critère de De Bruijn. Ces travaux nous ont amené à mettre au jour des manques dans les fondations du système qu'il faudra corriger pour obtenir un système complet. On a néanmoins démontré le bien fondé de ces extensions par le biais d'un exemple conséquent de programmation avec types dépendants et d'une nouvelle procédure de réécriture généralisée.

Travaux futurs

Nos travaux ouvrent des perspectives directes sur les fondements du système mais aussi plus largement autour de la programmation et la preuve en théorie des types.

Indifférence aux preuves On a vu que le principe d'indifférence aux preuves devait être internalisé pour pouvoir obtenir une traduction correcte de Russell vers Coq. Nous avons développé un prototype implémentant la proposition de Werner (2006), mais il reste des difficultés à résoudre pour en faire une alternative viable au noyau de Coq actuel.

Élimination dépendante Nous avons commencé des travaux autour de l'élimination dépendante. Essentiellement, nous avons réimplémenté les tactiques d'inversion de Coq développées par Cornes et Terrasse (1995) avec les techniques de McBride (1996) et McBride (2000); McBride, Goguen, et McKinna (2004) qui utilisent l'égalité hétérogène et l'axiome K pour traiter les problèmes posés par l'élimination dépendante. Encore une fois, une implémentation de l'indifférence aux preuves permettrait de se passer d'un tel axiome.

Ces techniques peuvent être utilisées pour simplifier la compilation du filtrage dépendant introduit par Coquand (1992). Le filtrage dépendant a été implémenté dans Alf (Magnusson et Nordström 1993) puis Agda 2 (Norell 2007) dans une approche dite « externe » où un nouvel algorithme vient étendre le noyau pour traiter la construction de filtrage. Grâce à la technique de Goguen et al. (2006) implémentée dans Epigram, on peut utiliser une approche dite « interne » où l'on compile le filtrage vers des constructions de plus bas niveau du langage, en supposant que l'on a K. Une extension du filtrage aux familles inductives au sein d'un système d'élaboration de programmes à partir d'équations a déjà été étudiée par Cornes (1997) dans le CCI mais elle n'a jamais vu le jour du fait de la complexité de sa solution (une partie de la construction se faisait dans une extension du système) et puisqu'elle s'appuyait toujours *in fine* sur l'axiome K.

Nous expérimentons actuellement une variante de cette construction utilisant la technique de Goguen et al., dans l'espoir que l'axiome K soit bientôt intégré au système ou que l'on puisse remplacer ses utilisations par des instances de l'axiome particulières pour des domaines décidables à l'aide de classes de type. En effet, l'architecture de compilation utilisée laisse une bonne part du travail à des tactiques qui peuvent utiliser ce genre de constructions.

Autour de Russell Comme on l'a déjà indiqué, l'extension de la preuve mécanisée d'équivalence des présentations déclarative et algorithmique de Russell doit être étendue par la preuve de la traduction qu'on a développé ici (chapitre 3). On est aussi mieux armés aujourd'hui pour reprendre cette preuve grâce à Program et aux nouvelles tactiques d'induction-inversion qu'on a développées.

Existentielles, recherche de preuve L'extension Program est maintenant robuste et a quelques utilisateurs. Cependant, une meilleure intégration avec le système de tactiques est toujours à l'ordre du jour, et cela passe par une meilleure gestion des existentielles dans Coq de façon générale. En effet, nous avons identifié la même difficulté lors du développement des classes de type, où la tactique de recherche de preuve est limitée par son traitement des variables existentielles.

Nous comptons poursuivre les travaux existants dans ce sens. En particulier, le développement d'un ensemble d'outils autour de la recherche de preuve semble une direction importante à poursuivre, notamment au vu des niveaux d'automatisation souhaités par les utilisateurs. Par exemple, on voudrait pouvoir obtenir une tactique de réécriture automatique efficace basée sur un réseau de discrimination des lemmes de réécriture à partir d'une telle boîte à outils.

Finger Trees et coercions On a repris le développement des Finger Trees en utilisant les classes de type pour représenter les monoïdes et les mesures. De façon plus ambitieuse, on pourrait essayer de gérer une égalité sétoïde sur les mesures et donc un notion de coercion particulière pour les indices des Finger Trees. On peut imaginer gérer le système de coercions directement à l'aide des classes. Par exemple on pourrait avoir la classe :

```
Class Coercion (from to : Type) :=
  coerce : from → to.
```

Lors du typage, au moment du test de conversion de deux types A, B, plutôt que d'échouer si A et B ne sont pas convertibles on pourrait lancer une résolution pour une instance `Coercion A B` et si elle réussit insérer automatiquement un appel à coerce sur l'objet t. On pourrait ainsi rendre le contrôle de la dérivation des coercions "totales" à l'utilisateur :

```
Instance subset_proj {A} {P : A → Prop} : Coercion (sig P) A :=
  coerce := @proj1_sig A P.

Definition sub_test (x : nat | x ≠ 0) : nat := coerce x.
```

Il devient alors possible de faire des spécialisations du typage en fonction des constructeurs de types utilisés. En étendant coerce pour générer optionellement une obligation lorsqu'on l'appele, on pourrait internaliser des coercions partielles comme la règle de coercion d'un type dans un sous-ensemble ou la coercion d'un Finger Tree en un autre s'ils ont des mesures équivalentes.

Réécriture généralisée On aimerait obtenir un résultat de complétude sur la réécriture sétoïde, ce qui nous permettrait probablement d'améliorer les messages d'erreur aux utilisateurs qui pour l'instant donnent simplement l'ensemble des contraintes générées par une réécriture en cas d'échec. Possiblement, une interprétation catégorique des constructions de base (flèche de respect, relations inverses et complémentaires, produits et sommes, notions de sous-relation et d'équivalence) nous permettrait de déterminer la théorie équationelle pour laquelle on cherche une procédure de (semi-)décision. Il reste aussi à explorer les différentes façons de travailler avec les relations partielles et les espaces fonctionnels à l'aide de cette tactique.

Classes de type La représentation des structures mathématiques par des records avec coercions et son interaction avec les classes doit elle aussi être poussée. L'unification des structures canoniques et des classes de types est notre objectif principal à atteindre pour

simplifier la présentation des deux systèmes et éviter une certaine redondance conceptuelle et pratique inutile.

Annexe A

Extraits de la librairie Coq

A.1 Library Coq.Init.Notations

These are the notations whose level and associativity are imposed by Coq

Notations for propositional connectives

```
Reserved Notation "x ↔ y" (at level 95, no associativity).
Reserved Notation "x ∧ y" (at level 80, right associativity).
Reserved Notation "x ∨ y" (at level 85, right associativity).
Reserved Notation "¬ x" (at level 75, right associativity).
```

Notations for equality and inequalities

```
Reserved Notation "x = y :> T"
  (at level 70, y at next level, no associativity).
Reserved Notation "x = y" (at level 70, no associativity).

Reserved Notation "x ≠ y" (at level 70, no associativity).

Reserved Notation "x ≤ y" (at level 70, no associativity).
Reserved Notation "x < y" (at level 70, no associativity).
Reserved Notation "x ≥ y" (at level 70, no associativity).
Reserved Notation "x > y" (at level 70, no associativity).
```

Arithmetical notations (also used for type constructors)

```
Reserved Notation "x + y" (at level 50, left associativity).
Reserved Notation "x - y" (at level 50, left associativity).
Reserved Notation "x × y" (at level 40, left associativity).
Reserved Notation "x / y" (at level 40, left associativity).
Reserved Notation "- x" (at level 35, right associativity).
Reserved Notation "/ x" (at level 35, right associativity).
Reserved Notation "x ^ y" (at level 30, right associativity).
```

Notations for booleans

```
Reserved Notation "x || y" (at level 50, left associativity).
Reserved Notation "x && y" (at level 40, left associativity).
```

Notations for pairs

```
Reserved Notation "( x , y , .. , z )" (at level 0).
```

A.2 Library Coq.Init.Logic

```
Require Import Coq.Init.Notations.
```

A.2.1 Propositional connectives

True is the always true proposition

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

False is the always false proposition

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

not A, written ~A, is the negation of A

```
Definition not (A : Prop) := A → False.
Notation "¬ x" := (not x) : type_scope.
```

and A B, written $A \wedge B$, is the conjunction of A and B
conj p q is a proof of $A \wedge B$ as soon as p is a proof of A and q a proof of B
proj1 and proj2 are first and second projections of a conjunction

```
Inductive and (A B : Prop) : Prop :=
  conj : A → B → A ∧ B
  where "A ∧ B" := (and A B) : type_scope.

Section Conjunction.
  Variables A B : Prop.
  Theorem proj1 : A ∧ B → A.
  Theorem proj2 : A ∧ B → B.
End Conjunction.
```

or A B, written $A \vee B$, is the disjunction of A and B

```
Inductive or (A B : Prop) : Prop :=
  | or_introl : A → A ∨ B
  | or_intror : B → A ∨ B
    where "A ∨ B" := (or A B) : type_scope.
```

iff A B, written $A \leftrightarrow B$, expresses the equivalence of A and B

```
Definition iff (A B : Prop) := (A → B) ∧ (B → A).
Notation "A ↔ B" := (iff A B) : type_scope.
```

A.2.2 First-order quantifiers

```
Inductive ex (A : Type) (P : A → Prop) : Prop :=
  ex_intro : ∀ x : A, P x → ex (A :=A) P.
Definition all (A : Type) (P : A → Prop) := ∀ x : A, P x.

Notation "'exists' x , p" := (ex (fun x ⇒ p))
  (at level 200, x ident, right associativity) : type_scope.
Notation "'exists' x : t , p" := (ex (fun x :t ⇒ p))
  (at level 200, x ident, right associativity, format "'[' 'exists' '/ ' x : t , '/ ' p ']'") :
type_scope.
```

Derived rules for universal quantification

```
Section universal_quantification.
  Variables (A : Type) (P : A → Prop).
```

```
  Theorem inst : ∀ x : A, all (fun x ⇒ P x) → P x.
  Theorem gen : ∀ (B : Prop) (f :forall y : A, B → P y), B → all P.
End universal_quantification.
```

A.2.3 Equality

```
Inductive eq (A : Type) (x : A) : A → Prop :=
    refl_equal : x = x :>A
  where "x = y :> A" := (@eq A x y) : type_scope.
Notation "x = y" := (x = y :>_) : type_scope.
Notation "x ≠ y" := (x ≠ y :>_) : type_scope.
```

A.3 Library Coq.Init.Specif

Basic specifications : sets that may contain logical information `Require Import` Coq.Init.Notations Coq.Init.Datatypes Coq.Init.Logic.

Subsets and Sigma-types

```
Inductive sig (A : Type) (P : A → Prop) : Type :=
    exist : ∀ x : A, P x → sig P.
Inductive sigT (A : Type) (P : A → Type) : Type :=
    existT : ∀ x : A, P x → sigT P.

Notation "{ x | P }" := (sig (fun x ⇒ P)) : type_scope.
Notation "{ x : A | P }" := (sig (fun x : A ⇒ P)) : type_scope.
Notation "{ x : A & P }" := (sigT (fun x : A ⇒ P)) : type_scope.
```

Projections of sig

```
Section Subset_projections.
  Variable A : Type.
  Variable P : A → Prop.
  Definition proj1_sig (e : sig P) := match e with
                                      | exist a b ⇒ a
                                      end.
  Definition proj2_sig (e : sig P) :=
    match e return P (proj1_sig e) with
    | exist a b ⇒ b
    end.
End Subset_projections.
```

Projections of sigT

```
Section Projections.
  Variable A : Type.
  Variable P : A → Type.
  Definition projT1 (x : sigT P) : A := match x with
                                        | existT a _ ⇒ a
                                        end.
  Definition projT2 (x : sigT P) : P (projT1 x) :=
    match x return P (projT1 x) with
    | existT _ h ⇒ h
    end.
End Projections.
```

A.4 Library Coq.Init.Datatypes

```
Require Import Coq.Init.Notations Coq.Init.Logic.
```

unit is a singleton datatype with sole inhabitant tt

```
Inductive unit : Set := tt : unit.
```

bool is the datatype of the boolean values true and false

```
Inductive bool : Set :=
| true : bool
| false : bool.
```

```
Definition negb (b :bool) := if b then false else true.
```

nat is the datatype of natural numbers built from O and successor S; note that the constructor name is the letter O. Numbers in nat can be denoted using a decimal notation; e.g. 3%nat abbreviates S (S (S O))

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

Empty_set has no inhabitant

```
Inductive Empty_set : Set :=.
```

option A is the extension of A with an extra element None

```
Inductive option (A :Type) : Type :=
  | Some : A → option A
  | None : option A.
Implicit Arguments None [A].
```

sum A B, written $A + B$, is the disjoint sum of A and B

```
Inductive sum (A B :Type) : Type :=
  | inl : A → sum A B
  | inr : B → sum A B.
Notation "x + y" := (sum x y) : type_scope.
```

prod A B, written $A \times B$, is the product of A and B; the pair pair A B a b of a and b is abbreviated (a,b)

```
Inductive prod (A B :Type) : Type :=
  pair : A → B → prod A B.
```

```
Notation "x × y" := (prod x y) : type_scope.
Notation "( x , y , .. , z )" := (pair ..
```

```
Section projections.
  Variables A B : Type.
  Definition fst (p :A × B) := match p with
                                 | (x, y) ⇒ x
                               end.
  Definition snd (p :A × B) := match p with
                                 | (x, y) ⇒ y
                               end.
End projections.
```

```
Definition prod_uncurry (A B C :Type) (f :prod A B → C)
  (x :A) (y :B) : C := f (pair x y).
```

```
Definition prod_curry (A B C :Type) (f :A → B → C)
  (p :prod A B) : C := match p with
                       | pair x y ⇒ f x y
                       end.
```

Identity

```
Definition ID := ∀ A :Type, A → A.
Definition id : ID := fun A x ⇒ x.
```

A.5 Library Coq.Lists.List

```
Require Import Le Gt Minus Min Bool.
Section Coq.Lists.Lists.
  Variable A : Type.

  Inductive list : Type :=
    | nil : list
    | cons : A → list → list.
  Infix " : :" := cons (at level 60, right associativity) : list_scope.
  Open Scope list_scope.
```

Head and tail

```
  Definition hd (default : A) (l : list) :=
   match l with
     | nil ⇒ default
     | x : : _ ⇒ x
   end.

  Definition tail (l : list) : list :=
    match l with
      | nil ⇒ nil
      | a : : m ⇒ m
    end.
```

Length of lists

```
  Fixpoint length (l : list) : nat :=
    match l with
      | nil ⇒ 0
      | _ : : m ⇒ S (length m)
    end.
```

Concatenation of two lists

```
  Fixpoint app (l m : list) {struct l} : list :=
    match l with
      | nil ⇒ m
      | a : : l1 ⇒ a : : app l1 m
    end.
End Coq.Lists.Lists.
```

Exporting list notations and tactics

```
Implicit Arguments nil [A].
Infix " : :" := cons (at level 60, right associativity) : list_scope.
Infix "++" := app (right associativity, at level 60) : list_scope.
```

A.6 Library Coq.Program.Basics

The polymorphic identity function is defined in *Datatypes*.

```
Implicit Arguments id [[A]].
```

Function composition.

```
Definition compose {A B C} (g : B → C) (f : A → B) :=
  fun x : A ⇒ g (f x).

Hint Unfold compose.
Notation " g ∘ f " := (compose g f)
  (at level 40, left associativity) : program_scope.
Open Local Scope program_scope.
```

The non-dependent function space between A and B.

```
Definition arrow (A B : Type) := A → B.
```

Logical implication.

```
Definition impl (A B : Prop) : Prop := A → B.
```

The constant function const a always returns a.

```
Definition const {A B} (a : A) := fun _ : B ⇒ a.
```

The flip combinator reverses the first two arguments of a function.

```
Definition flip {A B C} (f : A → B → C) x y := f y x.
```

Application as a combinator.

```
Definition apply {A B} (f : A → B) (x : A) := f x.
```

Curryfication of *prod* is defined in *Coq.Init.Logic.Datatypes*.

```
Implicit Arguments prod_curry [[A] [B] [C]].
Implicit Arguments prod_uncurry [[A] [B] [C]].
```

Index des définitions Coq

Acronymes

CC Calcul des Constructions – Théorie des types introduite par T. Coquand et G. Huet à la base des premières versions de Coq, caractérisée par une sorte `Set` imprédicative.

CCI Calcul des Constructions (Co-)Inductives – Extension du Calcul des Constructions par des types (co-)inductifs primitifs, introduit par C. Paulin et G. Huet à la base de Coq.

CCE Calcul des Constructions Étendu – Extension du Calcul des Constructions avec une hiérarchie d'univers, introduite par Z. Luo.

CCIp Calcul des Constructions (Co-)Inductives Prédicatif – Variante du Calcul des Constructions Inductives dont la sorte `Set` est prédicative, correspondant au noyau de Coq actuel.

OTT "*Observational Type Theory*" Théorie des types observationelle développée par T. Altenkirch, C. McBride et W. Swierstra

PCRI Pôle Commun de Recherche en Informatique – Regroupement d'instituts de recherche sur le plateau de Saclay

INRIA Institut National de Recherche en Informatique et Automatique

LRI Laboratoire de Recherche en Informatique de l'université Paris-Sud

LIX Laboratoire d'Informatique de l'X

X École Polytechnique située à Palaiseau

CNRS Centre National de la Recherche Scientifique

PTS Pure Type Systems Les Systèmes de Type Purs sont un formalisme introduit par H. G. Barendregt pour l'étude des lambda-calculs typés.

SR Subject Reduction Théorème de clôture du typage par réduction ou autoréduction.

CR Church-Rosser Théorème de confluence d'une relation de réduction.

TPOSR "*typed parrallel one-step reduction*".

GADT "*Generalized Algebraic Data Types*" ou types algébriques généralisés. Une variante plus faible des familles inductives implémentée dans Haskell et d'autres variantes de ML

PI "*Proof Irrelevance*" Principe d'indifférence aux preuves.

AVL Arbres binaires de recherche équilibrés du nom de leurs inventeurs G.M. Adelson-Velsky et E.M. Landis

TCC "*Type-Checking Condition*" Obligation de preuve générée au typage des fonctions dans PVS

Bibliographie

Robin Adams. Pure Type Systems with Judgemental Equality. *Journal of Functional Programming*, 16 :2 :219–246, 2006.

Thorsten Altenkirch, Conor McBride, et Wouter Swierstra. Observational Equality, Now ! In *PLPV'07 : Proceedings of the Programming Languages meets Program Verification Workshop*, 2007.

Clemens Ballarin. Interpretation of Locales in Isabelle : Theories and Proof Contexts. *Mathematical Knowledge Management*, pages 31–43, 2006.

Henk Barendregt. Typed lambda calculi. In Abramsky et al., editor, *Handbook of Logic in Computer Science*. Oxford Univ. Press, 1993.

Henk Barendregt et Tobias Nipkow, editors. Types for proofs and programs, international workshop types'93, nijmegen, the netherlands, may 24-28, 1993, selected papers, volume 806 of *Lecture Notes in Computer Science*, 1994. Springer.

Bruno Barras. Coq en Coq. Rapport de Recherche 3026, INRIA, October 1996.

Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, November 1999.

Bruno Barras et Bruno Bernardo. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In Roberto M. Amadio, editor, *FoSSaCS*, volume 4962 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2008.

Gilles Barthe. Implicit Coercions in Type Systems. In *TYPES*, pages 1–15, 1995.

Gilles Barthe. A computational view of implicit coercions in type theory. *Mathematical. Structures in Comp. Sci.*, 15(5) :839–874, 2005. ISSN 0960-1295.

Gilles Barthe, Venanzio Capretta, et Olivier Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2) :261–293, 2003.

Gilles Barthe et Olivier Pons. Type Isomorphisms and Proof Reuse in Dependent Type Theory. In Furio Honsell, editor, *FoSSaCS*, volume 2030 of *Lecture Notes in Computer Science*, pages 57–71, Genova, Italy, April 2001. Springer-Verlag.

David A. Basin. Generalized Rewriting in Type Theory. *Elektronische Informationsverarbeitung und Kybernetik*, 30(5/6) :249–259, 1994.

Nick Benton et Nicolas Tabareau. Compiling Functional Types to Relational Specifications for Low Level Imperative Code. In *TLDI*, 2009.

Frédéric Blanqui, Jean-Pierre Jouannaud, et Pierre-Yves Strub. Building Decision Procedures in the Calculus of Inductive Constructions. In Jacques Duparc et Thomas A. Henzinger, editors, *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2007.

Sandrine Blazy, Zaynah Dargaye, et Xavier Leroy. Formal Verification of a C Compiler Front-End. In Jayadev Misra, Tobias Nipkow, et Emil Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

Hans-Juergen Boehm, Russell R. Atkinson, et Michael F. Plass. Ropes : An Alternative to Strings. *Softw., Pract. Exper.*, 25(12) :1315–1330, 1995.

Samuel Boutin. *Réflexions sur les quotients*. thèse d'université, Paris 7, April 1997.

Edwin Brady, Conor McBride, et James McKinna. Inductive Families Need Not Store Their Indices. In Stefano Berardi, Mario Coppo, et Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2003.

Val Breazu-Tannen, Thierry Coquand, et Carl A. Gunter. Inheritance as implicit coercion. *INofrmation and computation*, 93(1) :172–221, 1991.

Venanzio Capretta. *Abstraction and Computation*. PhD thesis, University of Nijmegen, The Netherlands, April 2002.

Giuseppe Castagna. Covariance and contravariance : conflict without a cause. *ACM Trans. Program. Lang. Syst.*, 17(3) :431–447, 1995. ISSN 0164-0925.

Manuel M. T. Chakravarty, Gabriele Keller, Simon L. Peyton Jones, et Simon Marlow. Associated types with class. In Jens Palsberg et Martín Abadi, editors, *POPL*, pages 1–13. ACM, 2005.

Arthur Charguéraud et François Pottier. Functional Translation of a Calculus of Capabilities. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2008.

Chiyan Chen et Hongwei Xi. Combining Programming with Theorem Proving. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 66–77, Tallinn, Estonia, September 2005.

Gang Chen. *Sous-typage, Conversion de Types et Élimination de la Transitivité*. PhD thesis, Université Paris VII, Laboratoire d'Informatique de l'École Normale Supérieure, Paris, Décembre 1998.

Gang Chen. Coercive Subtyping for the Calculus of Constructions (extended abstract). In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, pages 150–159, 2003.

Adam Chlipala. *Implementing Certified Programming Language Tools in Dependent Type Theory*. Technical Report, University of California at Berkeley, 2007.

Claudio Sacerdoti Coen. A Semi-reflexive Tactic for (Sub-)Equational Reasoning. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, et Benjamin Werner, editors, *TYPES*, volume 3839 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 2004.

Sylvain Conchon et Evelyne Contejean. The Ergo automatic Theorem Prover. `http://ergo.lri.fr/`.

Thierry Coquand. Pattern Matching with Dependent Types, 1992. Proceedings of the Workshop on Logical Frameworks.

Thierry Coquand. Constructive Mathematics and Functional Programming (Abstract). *Programming Languages and Systems*, pages 146–147, 2008.

Thierry Coquand et Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2–3) :95–120, February/March 1988.

Pierre Corbineau. First-Order Reasoning in the Calculus of Inductive Constructions. *Types for Proofs and Programs*, pages 162–177, 2004.

Cristina Cornes. *Conception d'un langage de haut niveau de représentation de preuves : Récurrence par filtrage de motifs, Unification en présesence de types inductifs primitifs, Synthèse de lemmes d'inversion*. PhD thesis, Université Paris 7, Novembre 1997.

Cristina Cornes et Delphine Terrasse. Automating Inversion of Inductive Predicates in Coq. In Stefano Berardi et Mario Coppo, editors, *TYPES*, volume 1158 of *Lecture Notes in Computer Science*, pages 85–104. Springer, 1995.

Luís Cruz-Filipe, Herman Geuvers, et Freek Wiedijk. C-CoRN, the Constructive Coq Repository at Nijmegen. In Andrea Asperti, Grzegorz Bancerek, et Andrzej Trybulec, editors, *MKM*, volume 3119 of *Lecture Notes in Computer Science*, pages 88–103. Springer, 2004.

Luís Cruz-Filipe et Pierre Letouzey. A Large-Scale Experiment in Executing Extracted Programs. In *12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, Calculemus'2005*, 2005. To appear.

Nikolas. G. de Bruijn. Lambda Calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proc. of the Koninklijke Nederlands Akademie*, 75(5) :380–392, 1972.

David Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island (France)*, volume 1955 of *LNCS/LNAI*, pages 85–95. Springer-Verlag, November 2000.

David Detlefs, Greg Nelson, et James B. Saxe. Simplify : a theorem prover for program checking. *J. ACM*, 52(3) :365–473, 2005.

Gilles Dowek, Thérèse Hardin, et Claude Kirchner. Theorem Proving Modulo. *Journal of Automated Reasoning*, 31 :33–72, 2003.

Joshua Dunfield. Refined typechecking with Stardust. In *PLPV '07 : Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 21–32, New York, NY, USA, 2007. ACM.

Peter Dybjer. A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *J. Symb. Log.*, 65(2) :525–549, 2000.

Jean-Christophe Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4) :709–745, July 2003.

Jean-Christophe Filliâtre et Pierre Letouzey. Functors for Proofs and Programs. In D. Schmidt, editor, *European Symposium on Programing, ESOP'2004*, volume 2986 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

Tim Freeman et Frank Pfenning. Refinement Types for ML. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1991.

Diana Fulger. Reasoning about effects : tree labelling. Talk at the EffTT Small TYPES Workshop, December 2007.

Gallium, Marelle, CEDRIC, et PPS. The CompCert project. Compilers You Can *Formally* Trust, 2008.

Herman Geuvers. A short and flexible proof of Strong Normalization for the Calculus of Constructions. In Peter Dybjer, Bengt Nordström, et Jan M. Smith, editors, *TYPES*, volume 996 of *Lecture Notes in Computer Science*, pages 14–38. Springer, 1994.

Herman Geuvers et Benjamin Werner. On the Church-Rosser Property for Expressive Type Systems and its Consequences for their Metatheoretic Study. In *LICS*, pages 320–329. IEEE Computer Society, 1994.

Neil Ghani et Patricia Johann. Programming with Nested Types : A Principled Approach. Submitted to the Journal of Higher Order and Symbolic Computation, 2007.

Neil Ghani et Patricia Johann. Foundations for Structured Programming with GADTs. In *Proceedings of Principles and Programming Languages (POPL), 2008*, pages 297–308, 2008.

GHC. The Glasgow Haskell compiler.

Carlos Eduardo Giménez. *Un calcul de constructions infinies et son application à la vérification de systèmes communicants*. PhD thesis, Ecole Normale Supérieure de Lyon, Décembre 1996.

Jean-Yves Girard. *Le point aveugle : tome 1. Cours de Logique, Vers la perfection.*, volume 1 of *Visions des sciences*. Hermann, Juillet 2006.

Stéphane Glondu. Garantie formelle de correction pour l'extraction Coq. Master's thesis, Université Paris 7, 2007.

Healfdene Goguen, Conor McBride, et James McKinna. Eliminating Dependent Pattern Matching. In Kokichi Futatsugi, Jean-Pierre Jouannaud, et José Meseguer, editors, *Essays Dedicated to Joseph A. Goguen*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer, 2006.

Georges Gonthier et Benjamin Werner. A computer-checked proof of the four-coulour theorem, April 2005.

Benjamin Grégoire et Xavier Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pages 235–246. ACM Press, 2002.

Benjamin Grégoire et Assia Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq. *Theorem Proving in Higher Order Logics*, pages 98–113, 2005.

Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, et Cormac Flanagan. Sage : Hybrid Checking for Flexible Specifications. In *Scheme and Functional Programming*, September 2006.

Carl A. Gunter, Elsa L. Gunter, et David B. MacQueen. Computing ML equality kinds using abstract interpretation. *Inf. Comput.*, 107(2) :303–323, 1993. ISSN 0890-5401.

Florian Haftmann et Makarius Wenzel. Constructive Type Classes in Isabelle. In Thorsten Altenkirch et Conor Mc Bride, editors, *TYPES*, volume 4502 of *Lecture Notes in Computer Science*, pages 160–174. Springer-Verlag, 2006.

Haskell. The Haskell programming language.

Ralf Hinze et Ross Paterson. Finger Trees : A Simple General-purpose Data Structure. *J. Funct. Program.*, 16(2) :197–217, 2006.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) : 576–580, October 1969. ISSN 0001-0782.

Martin Hofmann. Elimination of Extensionality in Martin-Löf Type Theory. In Barendregt et Nipkow (1994), pages 166–190.

Martin Hofmann. *Extensional concepts in intensional type theory*. Phd thesis, Edinburgh university, 1995.

Martin Hofmann et Thomas Streicher. The groupoid interpretation of type theory. In *Twenty-Five Years of Constructive Type Theory*, pages 83–111. Oxford University Press, 1998.

Brian Huffman, John Matthews, et Peter White. Axiomatic Constructor Classes in Isabelle/HOLCF. In Hurd et Melham (2005), pages 147–162.

Hugs. Hugs 98.

Joe Hurd et Thomas F. Melham, editors. Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings, volume 3603 of *Lecture Notes in Computer Science*, 2005. Springer.

ICFPC. The 10th ICFP Programming Contest, juillet 2007. Organized by the Software Technology group at Utrecht University.

Bart Jacobs. *Categorical Logic and Type Theory*. Elsevier Science, 1999.

Mark P. Jones. A Theory of Qualified Types. In Bernd Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582, pages 287–306. Springer-Verlag, New York, N.Y., 1992.

Mark P. Jones. Type Classes with Functional Dependencies. In Gert Smolka, editor, *ESOP*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 2000.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, et Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP '06 : Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 50–61, New York, NY, USA, 2006. ACM.

Wolfram Kahl et Jan Scheffczyk. Named Instances for Haskell Type Classes. In Ralf Hinze, editor, *Proc. Haskell Workshop 2001*, volume 59 of *Electronic Notes in Theoretical Computer Science*, 2001.

Florian Kammüller, Markus Wenzel, et Lawrence C. Paulson. Locales - A Sectioning Concept for Isabelle. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, et Laurent Théry, editors, *TPHOLs*, volume 1690 of *Lecture Notes in Computer Science*, pages 149–166. Springer, 1999.

Haim Kaplan et Robert E. Tarjan. Purely functional, real-time deques with catenation. *Journal of the ACM*, 46(5) :577–603, 1999.

Butler W. Lampson et Rod M. Burstall. Pebble, a Kernel Language for Modules and Abstract Data Types. *Inf. Comput.*, 76(2/3) :278–346, 1988.

Xavier Leroy. Formal certification of a compiler back-end, or : programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

Pierre Letouzey. *Programmation fonctionnelle certifiée : l'extraction de programmes dans l'assistant Coq*. Thèse de doctorat, Université Paris-Sud, July 2004.

Pierre Letouzey. Coq Extraction, an Overview. In A. Beckmann, C. Dimitracopoulos, et B. Löwe, editors, *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, volume 5028 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

Daniel R. Licata et Robert Harper. An Extensible Theory of Indexed Types. Submitted to POPL'08, 2008.

William Lovas et Frank Pfenning. A Bidirectional Refinement Type System for LF. *Electr. Notes Theor. Comput. Sci.*, 196 :113–128, 2008.

Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, Department of Computer Science, University of Edinburgh, June 1990.

Zhaohui Luo. Coercive Subtyping in Type Theory. In Dirk van Dalen et Marc Bezem, editors, *CSL*, volume 1258 of *Lecture Notes in Computer Science*, pages 276–296. Springer, 1996.

Lena Magnusson et Bengt Nordström. The ALF Proof Editor and Its Proof Engine. In Barendregt et Nipkow (1994), pages 213–237.

Yitzhak Mandelbaum, David Walker, et Robert Harper. An Effective Theory of Type Refinements. In *ICFP '03 : Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, New York, NY, USA, 2003. ACM.

Per Martin-Löf. An intuitionistic theory of types, Predicative part. In *Logic Colloquium 1973*, pages 73–118, 1975.

Per Martin-Löf. Intuitionistic type theory. Bibliopolis, Naples, Italy, 1984.

Conor McBride. Inverting Inductively Defined Relations in LEGO. In Eduardo Giménez et Christine Paulin-Mohring, editors, *TYPES*, volume 1512 of *Lecture Notes in Computer Science*, pages 236–253. Springer, 1996.

Conor McBride. *Dependently Typed Functional Programs and Their Proofs*. PhD thesis, University of Edinburgh, 1999.

Conor McBride. Elimination with a Motive. In Paul Callaghan, Zhaohui Luo, James McKinna, et Robert Pollack, editors, *TYPES*, volume 2277 of *Lecture Notes in Computer Science*, pages 197–216. Springer, 2000.

Conor McBride. Epigram : Practical Programming with Dependent Types. *Advanced Functional Programming*, pages 130–170, 2005.

Conor McBride, Healfdene Goguen, et James McKinna. A Few Constructions on Constructors. *Types for Proofs and Programs*, pages 186–200, 2004.

Conor McBride et James McKinna. The view from the left. *J. Funct. Program.*, 14(1) : 69–111, 2004.

James McKinna. Why dependent types matter. In J. Gregory Morrisett et Simon L. Peyton Jones, editors, *POPL*, page 1. ACM, 2006.

James McKinna et Rod M. Burstall. Deliverables : A Categorial Approach to Program Development in Type Theory. In Andrzej M. Borzyszkowski et Stefan Sokolowski, editors, *MFCS*, volume 711 of *Lecture Notes in Computer Science*, pages 32–67. Springer, 1993.

James Hugh McKinna. *Deliverables : a categorical approach to program development in type theory*. PhD thesis, Edinburgh University, 1992.

John Meacham. JHC : John's Haskell Compiler, 2007.

Robin Milner, Mads Tofte, et Robert Harper. The Definition of Standard ML. MIT Press, 1990.

Alexandre Miquel. The Implicit Calculus of Constructions. In *TLCA*, volume 2044 of *Lecture Notes in Computer Science*, pages 344–359. Springer, 2001.

Alexandre Miquel et Benjamin Werner. The Not So Simple Proof-Irrelevant Model of CC. In Herman Geuvers et Freek Wiedijk, editors, *TYPES*, volume 2646 of *Lecture Notes in Computer Science*, pages 240–258. Springer, 2002.

Neil Mitchell et Colin Runciman. Uniform boilerplate and list processing. In *Haskell '07 : Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 49–60, New York, NY, USA, 2007. ACM.

Aleksandar Nanevski, Greg Morrisett, et Lars Birkedal. Hoare Type Theory, Polymorphism and Separation. *Journal of Functional Programming*, 2007. To appear.

Tobias Nipkow, Lawrence C. Paulson, et Markus Wenzel. Isabelle/HOL — A Proof Assistant for Higher-Order Logic, volume 2283 of *LNCS*. Springer, 2002.

Tobias Nipkow et Christian Prehofer. Type Checking Type Classes. In *POPL*, pages 409–418, 1993.

Tobias Nipkow et Gregor Snelting. Type Classes and Overloading Resolution via Order-Sorted Unification. In John Hughes, editor, *FPCA*, volume 523 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 1991.

Bengt Nordström, Kent Petersson, et Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Russell O'Connor et Bas Spitters. A computer verified, monadic, functional implementation of the integral. *CoRR*, abs/0809.1552, 2008.

Chris Okasaki. Purely Functional Data Structures. Technical Report CMU-CS-96-177, School of Computer Science, Carnegie Mellon University, September 1996a.

Chris Okasaki. The role of lazy evaluation in amortized data structures. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 62–72, May 1996b.

Nicolas Oury. Extensionality in the Calculus of Constructions. In Hurd et Melham (2005), pages 278–293.

Nicolas Oury et Thorsten Altenkirch. PiSigma. Prototype available on the web, 2008.

Nicolas Oury et Wouter Swierstra. The Power of Pi. Accepted for presentation at ICFP 2008, 2008.

Sam Owre et Natarajan Shankar. The Formal Semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.

Sam Owre, Natarajan Shankar, John M. Rushby, et D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

Catherine Parent. *Synthèse de preuves de programmes dans le Calcul des Constructions Inductives*. PhD thesis, ENS Lyon, 1995a.

Catherine Parent. Synthesizing Proofs from Programs in the Calculus of Inductive Constructions. In Bernhard Möller, editor, *MPC*, volume 947 of *Lecture Notes in Computer Science*, pages 351–379. Springer, 1995b.

Christine Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, 1993.

Christine Paulin-Mohring. Extraction de programmes et réalisabilité. Notes de cours du MPRI, 2008.

Lawrence C. Paulson. A Higher-Order Implementation of Rewriting. *Science of Computer Programming*, 3(2) :119–149 (or 119–150 ? ?), 1983.

Frank Pfenning. Logic Programming in the LF Logical Framework. In Gérard Huet et Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181, Cambridge, England, 1991. Cambridge University Press.

Robert Pollack. Dependently Typed Records for Representing Mathematical Structure. In Mark Aagaard et John Harrison, editors, *TPHOLs*, volume 1869 of *Lecture Notes in Computer Science*, pages 462–479. Springer, 2000.

François Pottier et Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL'06)*, pages 232–244, Charleston, South Carolina, January 2006.

Alain Prouté. Sur quelques liens entre théorie des topos et théorie de la démonstration. Conférence faite à l'Université de la Méditerrannée, mai 2007.

Matthias Puech. Reconnaissance automatique de structures mathématiques dans l'assistant de preuve Coq. Master's thesis, Université Paris 7, Septembre 2008.

Benoît Razet. Simulating Finite Eilenberg Machines with a Reactive Engine. In *Mathematically Structured Functional Programming*, Electronic Notes in Theoretical Computer Science, 2008. To appear.

Yann Régis-Gianas. *Des types aux assertions logiques : preuve automatique ou assistée de propriétés sur les programmes fonctionnels*. Doctorat, Université Paris 7, November 2007.

Yann Régis-Gianas et François Pottier. A Hoare Logic for Call-by-Value Functional Programs. In *Proceedings of the Ninth International Conference on Mathematics of Program Construction (MPC'08)*, volume 5133 of *Lecture Notes in Computer Science*. Springer-Verlag, July 2008.

John C. Reynolds. Using Category Theory to Design Implicit Conversions and Generic Operators. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258, Berlin, 1980. Springer-Verlag.

John C. Reynolds. Types, Abstraction, and Parametric Polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Paris, France*, pages 513–523. Elsevier, 1983.

Peter J. Robinson et John Staples. Formalizing a Hierarchical Structure of Practical Mathematical Reasoning. *Journal of Logic and Computation*, 3(1) :47–61, 1993.

John Rushby, Sam Owre, et Natarajan Shankar. Subtypes for Specifications : Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9) :709–720, September 1998.

Amokrane Saïbi. Typing Algorithm in Type Theory with Inheritance. In *24th Annual Symposium on Principles of Programming Languages*, pages 292–301, La Sorbonne, Paris, France, January 15-17 1997. ACM.

Anne Salvesen et Jan M. Smith. The Strength of the Subset Type in Martin-Löf's Type Theory. In *LICS*, pages 384–391. IEEE Computer Society, 1988.

Natarajan Shankar et Sam Owre. Principles and Pragmatics of Subtyping in PVS. In Didier Bert, Christine Choppy, et Peter Mosses, editors, *Recent Trends in Algebraic Development Techniques, WADT '99*, volume 1827 of *Lecture Notes in Computer Science*, pages 37–52, Toulouse, France, September 1999. Springer-Verlag.

Matthieu Sozeau. Russell Metatheoretic Study in Coq, experimental development, 2006. `http://mattam.org/research/russell/meta.en.html`.

Matthieu Sozeau. Dependent Finger Trees in Coq, 2007a. http://mattam.org/research/russell/fingertrees.en.html.

Matthieu Sozeau. Program-ing Finger Trees in Coq. In *ICFP'07 : Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming*, pages 13–24. ACM Press, 2007b.

Matthieu Sozeau. *Coq 8.2 Reference Manual*, chapter Program. INRIA TypiCal, 2008a.

Matthieu Sozeau. *Coq 8.2 Reference Manual*, chapter Type Classes. INRIA TypiCal, 2008b.

Matthieu Sozeau. *Coq 8.2 Reference Manual*, chapter User defined equalities and relations. INRIA TypiCal, 2008c.

Matthieu Sozeau. Order theory using type classes in Coq. Coq development, 2008d. http://mattam.org/research/coq.

Matthieu Sozeau et Nicolas Oury. First-Class Type Classes. In César Muñoz Otmane Ait Mohamed et Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21th International Conference*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, August 2008.

Martin Sulzmann, Gregory J. Duck, Simon Peyton-Jones, et Peter J. Stuckey. Understanding functional dependencies via constraint handling rules. *J. Funct. Program.*, 17 (1) :83–129, 2007. ISSN 0956-7968.

Jean van Heijenoort. *From Frege to Gödel : A Source Book in Mathematical Logic, 1879-1931 (Source Books in the History of the Sciences)*. Harvard University Press, January 2002.

Dimitrios Vytiniotis et Stephanie Weirich. Free Theorems and Runtime Type Representations. *Electron. Notes Theor. Comput. Sci.*, 173 :357–373, 2007. ISSN 1571-0661.

Philip Wadler. Views . A Way for elegant definitions and efficient representations to coexist. Oxford University, January 1985.

Philip Wadler. Theorems for Free ! In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 347–359, September 1989.

Philip Wadler et Stephen Blott. How To Make *ad-hoc* Polymorphism Less *ad hoc*. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 60–76, 1989.

Stephanie Weirich. RepLib : a library for derivable type classes. In *Haskell '06 : Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 1–12, New York, NY, USA, 2006. ACM.

Markus Wenzel. Using axiomatic type classes in Isabelle, 1995.

Markus Wenzel. Type Classes and Overloading in Higher-Order Logic. In Elsa L. Gunter et Amy P. Felty, editors, *TPHOLs*, volume 1275 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 1997.

Benjamin Werner. On the strength of proof-irrelevant type theories. *3rd International Joint Conference on Automated Reasoning*, 2006.

Sean Wilson. Supporting the Development of Dependently Typed Functional Programs. Talk given at DTP'08, February 2008.

Hongwei Xi. Applied Type Systems, 2005.

Hongwei Xi et Frank Pfenning. Dependent types in practical programming. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 214–227, January 1999.

Résumé

Les systèmes basés sur la Théorie des Types prennent une importance considérable tant pour la vérification de programmes qu'en tant qu'outils permettant la preuve formelle de théorèmes mettant en jeu des calculs complexes.

Ces systèmes nécessitent aujourd'hui une grande expertise pour être utilisés efficacement. Nous développons des constructions de haut niveau permettant d'utiliser les langages basés sur les théories des types dépendants aussi simplement que les langages de programmation fonctionnels usuels, sans sacrifier pour autant la richesse des constructions disponibles dans les premiers.

Nous étudions un nouveau langage permettant l'écriture de programmes certifiés en ne donnant que leur squelette algorithmique et leur spécification. Le typage dans ce système donne lieu à la génération automatique d'obligations de preuve pouvant être résolues a posteriori. Nous démontrons les propriétés métathéoriques essentielles du système, dont les preuves sont partiellement mécanisées, et détaillons son implémentation dans l'assistant de preuve Coq.

D'autre part, nous décrivons l'intégration et l'extension d'un système de "*Type Classes*" venu d'Haskell à Coq via une simple interprétation des constructions liées aux classes dans la théorie des types sous-jacente. Nous démontrons l'utilité des classes de types dépendantes pour la spécification et la preuve et présentons une implémentation économique et puissante d'une tactique de réécriture généralisée basée sur les classes.

Nous concluons par la mise en œuvre de l'ensemble de ces contributions lors du développement d'une bibliothèque certifiée de manipulation d'une structure de données complexe, les "*Finger Trees* ".

Abstract

Systems based on dependent type theory are getting considerable attention for the verification of computer programs as well as a practical tool for developing formal mathematical proofs involving complex and expensive computations.

These systems still require considerable expertise from the users to be used efficiently. We design high-level constructs permitting to use languages based on dependent type theory as easily as modern functional programming languages, without sacrificing the powerful constructs of the former.

We study a new language allowing to build certified programs while writing only their algorithmical squeleton and their specification. Typing in this system gives rise to proof obligations that can be handled interactively a posteriori. We demonstrate the main metatheoretical results on this system, whose proofs are partially mechanized, and present its implementation in the Coq proof assistant.

Then we describe an integration and extension of the type classes concept à la Haskell into Coq, providing a simple interpretation of the constructs linked with type classes into the underlying dependent type theory. We demonstrate the usefulness of these dependent type classes for specifications and proofs and present an economical yet powerful implementation of a generalized rewriting tactic based on them.

We conclude by employing these contributions in the development of a certified library of a complex data structure, Finger Trees.