Conservatoire National des Arts et Métiers

Thèse de doctorat présentée pour l'obtention du titre de

DOCTEUR EN SCIENCES DU CNAM
Spécialité : **INFORMATIQUE**

par
Olivier BOITE

Titre :

Une aide à la réutilisation de preuves formelles
Application aux preuves de propriétés sémantiques

Soutenue le 13 juin 2005 devant le jury composé de :

Président :	M. Roberto Di Cosmo
Rapporteurs :	M. Yves Bertot M. Jean-François Monin
Directrices de thèse :	Mme Catherine Dubois Mme Véronique Donzeau-Gouge
Examinateur :	M. Mathieu Jaume

Centre d'Études et De Recherche en Informatique du Cnam

Remerciements

Cette partie de la thèse apparait toujours en premier et pourtant c'est celle toujours écrite en dernier. Commençons la liste des remerciements, qui seront peut-être avec la première page d'introduction la seule partie lue par ma famille.

Je tiens à remercier Catherine Dubois ma co-directrice de thèse. Tout au long de cette thèse, Catherine m'a guidé, aidé et conseillé. Sa gentillesse et son attachement pour le sujet de cette thèse m'ont permis de travailler dans une ambiance agréable. Ses idées toujours bienvenues et ses relectures ont contribué à l'avancée de ma thèse.

Je remercie également Véronique Donzeau-Gouge, ma co-directrice de thèse, de m'avoir accueillie dans l'équipe "Conception et Programmation Raisonnée". Elle a porté un grand intérêt à mon travail. Ses remarques, questions et relectures ont permis d'améliorer la qualité de ce rapport.

Jean-François Monin et Yves Bertot m'ont fait l'honneur et le plaisir d'être rapporteurs de cette thèse. Je tiens à les remercier pour l'intérêt qu'ils ont apporté à mes travaux et pour leurs remarques constructives, qui ont permis d'améliorer ce manuscrit.

Je remercie aussi Mathieu Jaume et Roberto Di Cosmo pour l'honneur qu'ils me font de participer à mon jury de thèse.

Je me dois de dire merci à Olivier Pons, David Delahaye et Xavier Urbain qui m'ont écouté, conseillé et même donné des idées.

Je remercie également toutes les personnes que j'ai côtoyées à l'IIE : doctorants, enseignants, ingénieurs, administratifs et techniques.

Enfin, je termine par la partie que ma famille attendait : merci à mes parents et à mon frère qui m'ont soutenu et motivé et je remercie ma fiancée qui m'a supporté et encouragé.

Sommaire

Table des figures

Chapitre 1

Introduction

Les méthodes formelles sont le plus souvent utilisées pour valider un cahier des charges ou vérifier des propriétés, en particulier de sécurité. Elles sont aujourd'hui essentiellement utilisées pour les applications sécuritaires. Dès que des vies humaines ou que de lourds enjeux économiques sont impliqués, ces applications nécessitent de satisfaire des normes imposées. Les critères de sécurité à respecter deviennent vite complexes et nombreux. Il s'agit d'exprimer les propriétés requises par le futur système, de décrire un modèle de ce système et de montrer par la preuve que le système satisfait les propriétés.

Un frein pour une utilisation plus générale des méthodes formelles dans l'industrie vient de la difficulté de faire des preuves, et donc de leur coût. En effet, les preuves peuvent être longues et fastidieuses, et peuvent être difficiles d'un point de vue mathématique.

Pour palier à ce frein de l'utilisation de la preuve, les travaux actuels proposent toute une gamme d'outils qui vont des prouveurs ou procédures de décision spécialisés à un type de problème [16, 64], aux outils d'ingénierie de la preuve [75]. Les premiers permettent d'automatiser tout ou une partie de la preuve, mais ne peuvent se placer que dans un cadre restreint. Les seconds apporteront une aide au développement et à la maintenance des preuves. A côté de ceux-ci, on peut mettre en avant des techniques et outils permettant de réutiliser des spécifications et preuves.

Dans la plupart des prouveurs, on trouve des outils pour structurer et paramétrer les spécifications et les preuves : modules et foncteurs en Coq [5], espèces et héritage en Focal [79], machines abstraites et importations en B [1]. Il s'agit là d'un pas important vers la réutilisation des spécifications et des preuves permettant une réutilisation à grande échelle.

Mais peu de travaux concernent la réutilisation à petite échelle. Quels sont les impacts sur les preuves quand on modifie par exemple la définition d'un type ou d'une fonction à l'intérieur d'un composant ? À la moindre modification, il faut s'assurer que tout le développement formel reste correct. Autrement dit, même si les changements sont mineurs, toutes les preuves du composant doivent être refaites, même si elles sont identiques.

C'est précisément ce point que la thèse vise à traiter. Nous envisageons quelques modifications typiques dans un développement formel et proposons les outils permettant de mettre à jour et adapter les spécifications et les preuves touchées par ces modifications.

De nombreux formalismes mathématiques existent. Historiquement, on peut citer deux tentatives de formalisation des mathématiques. La première est la théorie des ensembles, où tout objet mathématique est considéré comme un ensemble. La seconde est le λ-calcul typé initié par Alonzo Church . Chacune est utilisée de nos jours dans des prouveurs : Z et la méthode B pour la théorie des ensembles, les assistants à la preuve Coq, Lego, PVS, HOL, Alfa... pour le λ-calcul. Nous nous intéresserons dans cette thèse aux méthodes formelles fondées sur la preuve dans le formalisme du λ-calcul typé.

Les techniques de réutilisation que nous présentons ici s'appliquent d'une manière générale dans des assistants à la preuve offrant des types inductifs, comme dans Isabelle [67], Alfa [56], ou comme ceux reposant sur le Calcul des Constructions Inductives (CCI) [70] par exemple les systèmes Coq [5] ou Lego [50]. Un deuxième prérequis nécessite que ces assistants à la preuve construisent une preuve sous la forme d'arbre de tactiques ou bien sous forme de λ-termes.

Comme le problème de la réutilisation est vaste et difficile, nous nous plaçons dans un cadre de réutilisation de preuves suite à des modifications bien définies dans les spécifications. Nous nous intéresserons à l'ajout de constructeurs, de paramètres, ou d'arguments dans un type inductif. Étendre des types inductifs en ajoutant de nouveaux cas, donc de nouveaux constructeurs, dans les spécifications d'un problème, est une démarche assez commune. Comme nous l'avons dit, il faut alors mettre à jour l'ensemble du développement, sans faire de copier-coller fastidieux. En effet, la plupart du temps lorsqu'on ajoute un ou plusieurs constructeurs à un type inductif, toutes les propriétés prouvées par induction se montrent de la même manière qu'auparavant et seule la preuve des nouveaux cas est à inventer.

Notre méthode de réutilisation concerne tout problème pouvant se formaliser à l'aide de spécifications inductives. Nous choisissons cependant d'appliquer la réutilisation de preuves à la conception des langages de programmation et plus particulièrement à la vérification de propriétés sémantiques. C'est en effet un domaine où il est fréquent de partir d'un noyau de langage et d'ajouter pas à pas des constructions. Il s'agit alors souvent de vérifier que cette nouvelle construction préserve une propriété sémantique donnée, comme le fait par exemple Benjamin Pierce dans [72]. L'exemple le plus étudié et le plus important en sémantique est certainement la conservation du typage par réduction, appelée *subject-reduction theorem* (SRT). Elle assure qu'un programme bien typé ne rencontrera pas d'erreur de typage en cours de réduction. Dans les preuves sur papier, il est fréquent de lire que la preuve est similaire à celle de Luis Damas et Robin Milner [28]. Pour les preuves qui procèdent à cette démarche par étapes en partant d'un noyau de langage et en l'enrichissant de plus en plus, il est fréquent de lire pour passer d'une étape à une autre que

la preuve est similaire au cas précédent pour les constructions communes du langage. Mais évidemment, avec une preuve assistée sur ordinateur comme celles de [66, 34, 89, 92], il n'est pas possible de faire une telle ellipse.

Les techniques d'extension que nous proposons se font de manière syntaxique. Il ne s'agit pas ici d'étendre le Calcul des Constructions Inductives avec une notion de type extensible, mais de fournir des outils générant des types inductifs étendus avec de nouveaux contructeurs, et de mettre à jour les preuves associées. Le mécanisme de réutilisation que nous proposons permet, en s'appuyant sur un calcul de dépendances, de réutiliser automatiquement les preuves après avoir ajouté des constructeurs, paramètres ou arguments dans des types inductifs. Une preuve partielle est ainsi créée et les obligations de preuves restantes correspondent aux nouveaux constructeurs. Cette réutilisation peut se faire soit en réutilisant l'arbre des tactiques appliquées pour créer l'ancienne preuve, soit en modifiant le terme de preuve. La seconde méthode est plus robuste et permet d'envisager plus de modifications. En effet, manipuler les termes permet d'envisager des modifications plus complexes tout en étant assuré de la correction de la transformation par le typage du terme produit.

Nous avons développé un prototype s'intégrant à l'assistant à la preuve dans le système Coq sans avoir à modifier son code source. C'est pourquoi les exemples que nous donnerons pour illustrer nos propos seront donné dans la syntaxe de Coq.

Cette thèse est organisée en 13 chapitres. Le chapitre 2 expose de manière détaillée le Calcul des Constructions Inductives. Ce chapitre est nécessaire à la compréhension du chapitre 4 et de certains exemples du chapitre 3. Avant tout, ce chapitre servira de référence pour les chapitres 6, 7, 8, 9 et 10.

Le chapitre 3 présente un état de l'art, non exhaustif, mais cependant assez riche, sur les différentes méthodes de réutilisation de preuves formelles. Nous donnerons quelques exemples pour chacune de ces méthodes.

Le chapitre 4 décrit un ensemble de commandes de base permettant de modifier des spécifications et réutiliser des preuves. Ces commandes forment l'environnement que nous souhaitons fournir, permettant de faire évoluer et maintenir un développement formel.

Le chapitre 5 formule le besoin et la manière de calculer les dépendances entre les spécifications et les preuves, au sein d'un développement formel.

Le chapitre 6 décrit une première méthode de réutilisation de preuves, qui exploite le script de tactiques, dans le cas d'ajout de constructeurs à un type inductif. Cette méthode présente l'avantage d'être utilisable dans tout système d'aide à la preuve utilisant des tactiques.

Le chapitre 7 montre une autre méthode de réutilisation de preuves dans le cas d'ajout de constructeurs : celle-ci exploite les termes de preuve. Une preuve de correction assure que la méthode fournit bien la preuve attendue.

Les chapitres 8, 9 et 10 suivent le même schéma que le chapitre 7, pour respec-

tivement la suppression d'un constructeur, l'ajout de paramètres à un type inductif et l'ajout d'arguments à un constructeur ou à un type inductif.

Le chapitre 11 décrit l'implémentation d'un prototype permettant de tester les transformations présentées dans les chapitres précédents.

Le chapitre 12 applique ces méthodes de réutilisation de spécifications et de preuves dans le contexte de la sémantique des langages. Ces exemples, qui ont motivé cette thèse, illustrent l'intérêt de notre environnement de réutilisation.

Enfin, nous terminons par la conclusion sur notre contribution et esquissons les principales perspectives.

Chapitre 2

Le Calcul des Constructions Inductives et Coq

Le Calcul des Constructions (CC) [25] est le λ-calcul le plus puissant sur le cube de Barendregt [3]. Le CC étend le système F de Girard avec les types dépendants et l'ordre supérieur. Le Calcul des Constructions Inductives (CCI) est une extension du Calcul des Constructions, où les définitions de types inductifs et de fonctions récursives sont primitives. Cette extension dûe à Christine Paulin [70], Thierry Coquand [26] et Benjamin Werner [94] permet d'avoir un meilleur aspect calculatoire que le CC pour les types inductifs, comme par exemple la possibilité de montrer que $O \neq 1$. Le système autorise des définitions récursives à la ML et des prédicats inductifs à la PROLOG, ce qui permet d'avoir un système puissant tout en étant convivial.

Dans cette partie, nous présentons le Calcul des Constructions Inductives et l'assistant à la preuve Coq développé par l'INRIA Rocquencourt, l'ENS Lyon et le CNRS depuis 1984. Nous décrivons d'abord les types inductifs, puis donnons une définition formelle du CCI. Les exemples seront donnés dans la syntaxe de Coq v7.x.

2.1 Les types inductifs

2.1.1 Définitions inductives

La notion de type inductif permet de représenter des structures récursives de manière naturelle. Par exemple, les listes, les arbres, les syntaxes abstraites se définissent très facilement. Il suffit d'en donner les différents constructeurs et les types de ces derniers.

Un type inductif est spécifié par son nom, son type, et le type de ses constructeurs. Un élément d'un type inductif est construit par applications successives des constructeurs du type inductif et uniquement ceux-là. Plus précisemment, un type inductif est le plus petit ensemble clos par application de ses constructeurs.

Les types des entiers naturels `nat` et des listes d'entiers naturels `liste` sont donnés ici dans la syntaxe de Coq :

```
Inductive nat : Set :=
  O : nat
| S : nat→nat
```

Les entiers naturels représentés par le type `nat` sont construits à partir du constructeur constant `O` et du constructeur `S` (pour successeur) qui prend en argument un autre entier de type `nat`. Le type des listes d'entiers se définit de même :

```
Inductive liste : Set :=
  Nil : liste
| Cons : nat → liste → liste
```

Ainsi le nombre 3 se représente par le terme de type `nat (S (S (S O)))`. La liste [2 ;1 ;0] se représente par `(Cons (S (S O)) (Cons (S O) (Cons O nil)))`.

Le type d'un type inductif peut être fonctionnel, cela permet de définir des prédicats inductifs. Par exemple `is_pair` qui s'applique à un argument `n` de type `nat`, est vrai lorsque `n` est pair.

```
Inductive is_pair : nat → Prop :=
  O_is_pair : (is_pair O)
| SSis_pair : ∀n:nat.(is_pair n) → (is_pair (S (S n)))
```

La liste des constructeurs indique les différentes manières de construire un objet de type `(is_pair n)`. Pour un entier `n1` de type `nat`, `(is_pair n1)` est de type `Prop` et a le statut de prédicat. Ce prédicat ne peut être établi que s'il est construit à l'aide du constructeur `O_is_pair` ou du constructeur `SSis_pair`. Ce second constructeur se lit intuitivement "pour tout `n` de type `nat`, si `n` est pair alors `(S (S n))` est pair". Donc dans une preuve, si nous avons `is_pair n1` en hypothèse, nous pouvons en déduire que soit `n1` est `O` (cas `O_is_pair`), soit `n1` est de la forme `(S (S n0))` et `is_pair n0` (cas `SSis_pair`).

2.1.2 Filtrage et ι-réduction

Nous avons adopté ici, comme dans [5] ou [70], le choix de prendre les constructions de filtrage et de point-fixe de manière primitive. Les schémas d'induction sont définis à partir de ceux-ci. L'autre possibilité aurait été de prendre des schémas d'élimination primitifs.

Notre CCI dispose donc d'une construction d'analyse par cas `Cases ... end` qui est l'analogue au filtrage de OCaml. Si `l` est une liste du type `liste` précédent, une analyse par cas sur `l` doit donner de manière exhaustive, la liste de motifs sur lesquels `l` peut être construite. Par exemple, la fonction suivante procède à une analyse par cas sur `l`.

```
Definition tail [l:liste] : liste :=
 Cases l of Nil => Nil
         | (Cons a l') => l'
 end
```

Dans une analyse par cas, les paramètres du motif qui a été filtré sont instanciés, et l'expression correspondante est renvoyée. Cette règle de réduction d'un `Cases ... end` s'appelle une ι-réduction (*iota*-réduction).

2.1.3 Preuves par induction

La plupart du temps, une propriété portant sur un type inductif se montre par induction. Imaginons que nous voulions prouver une propriété sur une liste quelconque `∀l :liste.(Q l)`, par induction sur cette liste. Dans ce cas, on est amené à prouver `(Q Nil)` et `(Q l)` implique `(Q (Cons n l))` pour tout `n :nat` et tout `l :liste`. Ce principe d'induction n'est pas primitif mais il est automatiquement généré par Coq au moment de la définition du type `liste`. Il porte le nom `liste_ind` et a le type

```
∀P:liste→Prop.
(P Nil)→
(∀n:nat.∀l:liste.(P l)→(P (Cons n l)))→
∀l:liste.(P l)
```

Ce principe s'applique sur notre but `∀l :liste (Q l)`, et engendre la preuve par induction que nous voulions faire, en générant les sous-buts `(Q Nil)` et `∀n :nat ∀l :liste (Q l)→(Q (Cons n l))`.

Pour le type `liste`, en plus du principe d'induction `liste_ind`, deux autres principes d'élimination sont générés, respectivement sur `Set` et `Type`.

```
liste_rec :
     ∀P:(liste→Set).
     (P Nil)→
     (∀n:nat.∀l:liste.(P l)→(P (Cons n l)))→
     ∀l:liste.(P l)
```

et

```
liste_rect :
     ∀P:(liste→Type).
     (P Nil)→
     (∀n:nat.∀l:liste.(P l)→(P (Cons n l)))→
     ∀l:liste.(P l)
```

Le principe d'élimination `liste_rect` appelé élimination forte permet de faire une preuve par induction sur le type `Type`. Nous en reparlerons à la fin de ce chapitre.

Ces trois principes d'élimination ne sont pas primitifs, ils sont générés de manière automatique à partir des constructions primitives de point fixe et d'analyse par cas. Dans la pratique, ce sont ces principes d'élimination qui sont utilisés, et non pas leur définition à l'aide des constructions primitives.

2.1.4 Preuves par inversion

L'idée de l'inversion sur un type inductif est que tout habitant de ce type doit être construit à partir d'un des constructeurs du type inductif, vu comme règles d'inférence. C. Cornes et D. Terrasse [27] exposent l'automatisation de l'inversion en Coq.

Inverser une hypothèse H de type inductif I permet d'ajouter en hypothèse toutes les prémisses qui ont permis d'établir H, accompagnées des contraintes sur les arguments de H, contenues dans les règles d'inférence. Si plusieurs règles sont applicables, le but courant est divisé en autant de sous-but. Ainsi, une inversion correspond à une analyse par cas accompagnée de contraintes sur les arguments.

Par exemple, considérons le type inductif `le` correspondant à la relation $\leqslant$ sur les entiers naturels.

```
Inductive le : nat→nat→Prop :=
   le_O : ∀n:nat.(le O n)
 | le_S : ∀n,m:nat.(le n m)→(le (S n) (S m))
```

Supposons avoir l'hypothèse `H1 :(le a O)`. Une inversion sur `H1` permet de déduire que `a=O`. En effet, la seule règle qui a pu construire `H1` est `le_n`. Or les contraintes de `le_n` imposent d'avoir `a=O`.

De même, une hypothèse `H2 :(le (S a) b)` n'a pu être construite que par `le_S`. Une inversion sur `H2` ajoute donc les hypothèses `m :nat`, `b=(S m)` et `(le a m)`.

Enfin, une hypothèse `H3 :(le (S a) O)` n'a pu être construite avec aucune règle. Une inversion sur `H3` déduit que l'hypothèse `H3` est absurde, donc le but est déchargé.

2.1.5 Type inductif paramétré

Un type inductif peut être fonctionnel s'il prend un argument. Par exemple `is_pair` prend un argument de type `nat`. Un paramètre au sens des types inductifs tient un rôle légèrement différent. Un paramètre d'un type inductif est constant dans la définition du type, sa portée s'étend à toute la définition. Cela permet par exemple de représenter des listes d'objets de type `A`, où `A` est le paramètre. On peut considérer qu'un type inductif paramètré par un type `A` est polymorphe. Le paramètre pourra être instancié par tout type dont le type est celui du paramètre.

```
Inductive list [A:Set] : Set :=
  Nil : (list A)
| Cons : A → (list A) → (list A).
```

Le paramètre `A` peut être instancié par tout objet de type `Set`, comme `nat` par exemple. Ainsi le type `(list nat)` a le type `Set`. En revanche, pour représenter des listes de longueur n, nous ne pouvons pas mettre n en paramètre car il faut précisément faire varier n dans la définition : la liste vide est de longueur O, une liste non-vide est de longueur $(S\ n)$ si son reste est de longueur n. La solution est d'utiliser un argument au lieu d'un paramètre, car un argument peut varier dans la définition.

Nous définissons ci-dessous le type des listes d'entiers de longueur n :

```
Inductive list2 : nat → Set :=
  Nil : (list2 O)
| Cons : ∀n:nat.nat → (list2 n) → (list2 (S n)).
```

L'utilisation des paramètres dans les types inductifs est plus contrainte que les arguments au niveau de la définition, et permet une forme de polymorphisme quand les paramètres sont de type `Set`.

La différence entre paramètre et argument se retrouve au niveau des principes d'induction générés. Dans le cas du paramètre, le principe d'induction est plus simple, il est simplement quantifié sur ce paramètre, comme illustré ci-dessous avec le principe d'induction sur `list`.

```
list_ind: ∀A:Set.∀P:((list A)→Prop).
          (P (Nil A)) →
          (∀a:A.∀l:(list A).(P l)→(P (Cons A a l)))→
          ∀l:(list A).(P l)
```

Dans le cas d'un argument, la quantification porte sur le type de la propriété `P` sur laquelle le principe d'induction sera appliqué, mais une autre quantification apparait sur la conclusion du principe d'induction :

```
list2_ind: ∀P:(∀n:nat. (list2 n)→Prop).
        (P O Nil) →
        (∀n,n0:nat.∀l:(list2 n).(P n l)→(P (S n) (Cons n n0 l)))→
        ∀n:nat.∀l:(list2 n).(P n l)
```

Lorsqu'un type inductif possède à la fois des paramètres et des arguments nous retrouvons à la fois les quantifications des paramètres sur tout le principe d'induction, et les quantifications des arguments sur le type de la propriété `P`, comme par exemple dans `list3` :

```
Inductive list3 [A:Set] : nat → Set :=
  Nil : (list3 A O)
| Cons : ∀n:nat. A → (list3 A n) → (list3 A (S n))

list3_ind: ∀A:Set.∀P:(∀n:nat.(list3 A n)→Prop).
  (P O (Nil A))→
```

```
(∀n:nat.∀a:A.∀l:(list3 A n).(P n l)→(P (S n) (Cons A n a l)))→
∀n:nat.∀l:(list3 A n).(P n l)
```

Une autre différence entre paramètre et argument se retrouve dans la forme des analyses par cas. Nous nous en discuterons plus en détails dans les chapitres 9 et 10.

2.1.6 Types inductifs particuliers

Nous décrivons ici trois types inductifs particuliers qui ont un rôle important au niveau logique et que l'on retrouve donc dans la plupart des preuves.

Le type qui n'a aucun constructeur représente la proposition toujours fausse :

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

Ce type n'a aucun habitant car les seuls habitants d'un type inductif sont ceux construits avec les constructeurs. On ne pourra donc jamais fournir de preuve de ce type. C'est exactement ce qu'on attend du Faux en logique.

L'égalité en Coq est également définie par un type inductif.

```
Inductive eq [A:Set; x:A] : A→Prop :=
  refl_equal : (eq A x x)
```

On notera `(eq A x x)` par `x=x`. Il s'agit de l'égalité de Leibniz. Le principe d'élimination

```
eq_ind : ∀A:Set ∀x:A ∀P:(A→Prop) (P x)→ ∀y:A x=y →(P y)
```

permet de remplacer dans une preuve un terme par un autre, quand celui-ci lui est égal au sens de `eq`.

2.2 Le Calcul des Constructions Inductives

Dans cette section, nous présentons une version simplifiée du CCI, inspirée de la description donnée dans le manuel de référence de Coq [5]. D'autres présentations peuvent être trouvées par exemple dans la thèse d'habilitation à diriger des recherches de Christine Paulin [70], dans la thèse de doctorat de Benjamin Werner [94]. Bruno Barras dans sa thèse de doctorat [4] donne aussi une autre définition du CCI en tant que *Pure Type System* (PTS) avec opérateurs.

2.2.1 Les termes du CCI

Pour simplifier, nous ne considérons pas les fonctions et types mutuellement inductifs, les définitions locales (`let in`) et les types coinductifs. En plus de ces simplifications la différence majeure entre le manuel de référence et notre présentation

concerne le nommage des constructeurs d'un type inductif. Au lieu de les nommer, nous les représentons par un couple (n, I) où I est le type inductif d'où provient le constructeur, et n est le numéro du constructeur. Cela suppose évidemment que l'ordre dans lequel les constructeurs sont donnés dans la définition de I est significatif et conservé.

Le CCI permet de spécifier des définitions à l'aide de termes et des propriétés à l'aide de types. La preuve d'une propriété est également un objet du formalisme, représentée par un terme. En vertu de l'isomorphisme de Curry-Howard, lorsqu'un terme Λ est typé par τ (selon les règles décrites en figure 2.2), nous sommes en présence d'une preuve Λ de la propriété τ.

Les sortes

Les termes et les types sont définis dans la même algèbre, autrement dit un type est un terme, comme dans la théorie des types de Martin-Löf. Cela se distingue de la théorie utilisée par PVS, où les preuves ne sont pas des objets du formalisme. Cela se distingue également des théories des types des langages de programmation où les types ont leur propre catégorie syntaxique.

Ainsi dans le CCI un type, qui est aussi un terme, doit pouvoir être typé. Le type d'un type est appelé une *sorte*. On distingue deux sortes de base, `Set` et `Prop` : `Set` est le type des données, autrement dit des objets ayant un contenu calculatoire, `Prop` est le type des propositions logiques. En tant que termes du langage, `Set` et `Prop` sont typés par la sorte `Type(0)` [1]. La propriété $\texttt{Type}(i) : \texttt{Type}(i+1)$ permet de typer toute une hiérarchie de sortes. On note S l'ensemble des sortes :

$$S = \{Prop, Set, Type(i) | i \in \mathbb{N}\}$$

Dans la suite, lorsqu'il ne sera pas nécessaire de faire apparaître l'index i, nous écrirons simplement $Type$.

Les constantes

Les sortes peuvent être considérées comme des constantes. Mais nous réserverons cette qualification pour deux autres situations. Une constante sera déclarée dans un environnement de déclarations, et désignera un objet précédemment défini.

Dans notre présentation des termes du CCI, comme dans [5] ou [70], le choix est fait de placer également dans cet environnement de déclarations les types inductifs, au lieu de les considérer comme des termes de première classe comme les autres, comme dans [69] ou [94]. Cela présente d'abord l'avantage d'être proche de l'implémentation. Mais surtout, cela permet de pouvoir partager la définition d'un type inductif, au lieu de devoir tout écrire à chaque occurrence de ce type inductif.

[1] Dans la pratique, l'utilisateur écrit `Type`, et le système infère seul le niveau dans la hiérarchie des sortes

L'algèbre de termes du CCI

Nous donnons à la figure 2.1 une définition des termes du CCI par l'algèbre de termes M.

s	$:=$	$Set \mid Prop \mid Type(i)$	sortes
Γ	$:=$	$\emptyset \mid \Gamma;\ x : M$	contexte
E	$:=$	$\emptyset \mid E; Ind(c : M)[\Gamma]\{\overrightarrow{M}\} \mid E;\ c := M : M \mid E;\ c : M$	environnement
M	$:=$	$s \mid x \mid c \mid \forall x : M.M \mid \lambda x : M.M \mid (M\ \overrightarrow{M})$	termes
		$\mid Fix(x/i : M)\{M\} \mid Case(M, M, \overrightarrow{M}) \mid Constr(i, c)$	

Fig. 2.1 – Algèbre simplifiée des termes du CCI

La lettre x désigne une variable, c une constante, i un entier et s une sorte.

Un contexte Γ est une liste de couples formés d'un identificateur et d'un terme. Chaque identificateur dans un contexte est différent. $\emptyset$ dénote un contexte vide.

Le produit $\forall x : M.M$ est un produit dépendant, encore noté $(x : M)M$ ou $\Pi x : M.M$. Dans $\forall x : A.B$, le type B peut dépendre de x, lorsque ce n'est pas le cas, c'est-à-dire quand x n'a pas d'occurence libre dans B, l'expression $\forall x : A.B$ pourra s'écrire $A \rightarrow B$.

La notation $\overrightarrow{M}$ désigne une liste de termes $M_1, M_2, ..., M_n$. L'application $(M\ M_1, M_2, ..., M_n)$ représente les applications successives $(...(M\ M_1)\ M_2)...M_n)...)$.

L'environnement de déclarations E contient des types inductifs, des déclarations de constantes $c := M : M'$ déclarant que c est une définition dont la valeur est M et le type M', et des axiomes $c : M$ de nom c et de type M. Nous utiliserons la notation $(c : T) \in E$ lorsque c est une définition de type T déclarée dans E ou bien un axiome de type T déclaré dans E.

Un type inductif $Ind(c : M)[\Gamma_1]\{M_1, ..., M_n\}$ déclaré dans E a pour nom c et pour type M, Γ_1 est la liste des paramètres et $M_1, ..., M_n$ est la liste ordonnée des types des constructeurs. Par exemple, le type des entiers naturels est représenté par le terme $Ind(nat : Set)[]\{nat, nat \rightarrow nat\}$.

Le terme $Constr(i, I)$ désigne le $i^{ème}$ constructeur du type inductif de nom I. Dans notre exemple, O est représenté par $Constr(1, nat)$ et S par $Constr(2, nat)$.

La construction $Case(e, P, M_1...M_n)$ permet de faire une analyse par cas sur le terme e d'un type inductif. Le terme P est le prédicat d'élimination. Il s'agit d'un terme qui permettra de typer chacune des branches de l'analyse par cas. Les différents cas examinés dans une analyse par cas concernent la liste exhaustive des constructeurs de I avec leurs arguments, autrement dit des motifs. En syntaxe concrète, si C est un constructeur binaire de I, nous aurions `(C x y) => u`. Les termes $M_1...M_n$ correspondent aux différentes branches de l'analyse par cas, mises sous forme fonctionnelle : les arguments du constructeur sont considérés en tant qu'abstractions dans les termes M_i. En syntaxe abstraite, l'exemple devient $\lambda x.\lambda y.u$.

Enfin, la construction $Fix(f/n : T)\{M\}$ définit une fonction récursive f de

corps M et de type T. Ce type commence par au moins n produits représentant les arguments. L'entier n indique le rang de l'argument décroissant.

Les variables libres et l'opération de substitution qui substitue les occurences libres d'une variable x dans un terme u par un terme t, notée u{x/t}, se définissent de manière usuelle, en prenant soin de renommer pour éviter toute capture de variables.

Critère de terminaison

Pour préserver la normalisation forte du calcul, les fonctions, en particulier les fonctions récursives, doivent toujours terminer. Un critère syntaxique de terminaison permet de l'assurer : on impose que chaque appel récursif dans la définition d'une fonction f par Fix, porte sur un sous-terme strict. Dans $Fix(f/n : T)\{M\}$, cette décroissance bien-fondée par sous-terme doit porter sur l'argument de rang n.

2.2.2 Réductions

La réduction élémentaire comme dans tout λ-calcul, est la β-réduction qui réduit l'application d'une abstraction et d'un terme :

(β) $(\lambda x : T.u)\ v \rhd_\beta u\{x/v\}$

Le terme réduit est le terme u dans lequel on substitue les occurences libres de x par v.

La δ-réduction permet de remplacer une constante par sa définition, contenue dans l'environnement E. Cette réduction dépend donc des déclarations, et on dira qu'elle est contextuelle par rapport à l'environnement.

(δ) $E \vdash c \rhd_\delta t$, si $(c := t : T)\ \in E$

La ι-réduction réduit les appels récursifs de fonction et les analyses par cas.

(Φ) $(Fix(f/n : T)\{M\})\ u_1..u_n \rhd_\Phi (M\{f/Fix(f/n : T)\{M\}\})\ u_1..u_n$
si u_n, l'argument décroissant, commence par un constructeur.

Une fonction récursive $(Fix(f/n : T)\{M\})$ appliquée à n arguments $u_1..u_n$ se réduit en M appliqué à ces n arguments, où les occurences de f sont remplacées par sa définition $(Fix(f/n : T)\{M\})$.

(ι) $Case((Constr(i, I))\ p_1...p_r\ a_1...a_m, P, M_1...M_n) \rhd_\iota M_i\ a_1...a_m$
où I est un type inductif à r paramètres

Une analyse par cas sur un terme commençant par le constructeur $Constr(i, I)$ appliqué à ses paramètres et arguments, où les différentes branches de l'analyse sont

$M_1 ... M_n$, se réduit en M_i appliqué aux arguments, car on se trouve dans le cas du $i$ème constructeur, appliqué aux arguments.

Définition 2.2.1 *On notera $\triangleright$ la fermeture réflexive et transitive de $\triangleright_\beta \cup \triangleright_\delta \cup \triangleright_\Phi \cup \triangleright_\iota$.*

Cette réduction est contextuelle par rapport à l'environnement de déclarations.

Définition 2.2.2 *convertibilité*
Deux termes t et u sont convertibles, noté $E[\Gamma] \vdash t =_{\beta\delta\Phi\iota} u$, si ils peuvent être réduits en un même terme v, c'est-à-dire si $E[\Gamma] \vdash t \triangleright v$ et $E[\Gamma] \vdash u \triangleright v$. On dira aussi qu'ils sont intentionnellement égaux.

La convertibilité est étendue en un ordre dit de cumulativité, noté $\leq_{\beta\delta\Phi\iota}$, en ajoutant la propriété qu'un terme de type $Type(i)$ est aussi de type $Type(i+1)$.

Définition 2.2.3 *cumulativité*
L'ordre de cumulativité $E[\Gamma] \vdash t \leq_{\beta\delta\Phi\iota} u$ est défini par :
- *si $E[\Gamma] \vdash t =_{\beta\delta\Phi\iota} u$ alors $E[\Gamma] \vdash t \leq_{\beta\delta\Phi\iota} u$*
- *si $i \leq j$ alors $E[\Gamma] \vdash Type(i) \leq_{\beta\delta\Phi\iota} Type(j)$*
- *$E[\Gamma] \vdash Set \leq_{\beta\delta\Phi\iota} Type$*
- *$E[\Gamma] \vdash Prop \leq_{\beta\delta\Phi\iota} Type$*
- *si $E[\Gamma] \vdash t =_{\beta\delta\Phi\iota} u$ et $E[\Gamma; (x : t)] \vdash t' \leq_{\beta\delta\Phi\iota} u'$ alors $E[\Gamma] \vdash \forall x : t.t' \leq_{\beta\delta\Phi\iota} \forall x : u.u'$*

Définition 2.2.4 *Normalisation forte*
La normalisation forte signifie que toute réduction d'un terme bien typé converge vers une forme normale.

Le CCI possède la propriété de normalisation forte, comme tous les λ-calculs du cube de Barendregt. Il est important de garantir cette propriété, qui peut être comparée à l'élimination des coupures, afin de s'assurer que toute réduction termine.

2.2.3 Condition de positivité

Afin de ne pas aboutir à une réduction infinie ou à une preuve de $False$, la forme des constructeurs d'un type inductif est limitée.

Par exemple, supposons que le type inductif suivant soit accepté :

```
Inductive absurd : set :=
 c : (absurd → absurd) → absurd.
```

En définissant la fonction `app` :

```
Definition app [a1:absurd] : absurd → absurd := [a2:absurd]
 Cases a1 of (c f) => f a2
 end.
```

la réduction suivante s'obtient en une étape de ι-réduction :
`(app (c f) a)` $\triangleright$ `(f a)`.
En utilisant cette réduction, si on considère la constante `delta`

```
Definition delta : absurd :=
  (c ([x:absurd] (app x x))).
```

alors l'expression `(app delta delta)` se réduit en elle-même. L'expression est bien typée mais pourtant non normalisable, ce qui contredirait la propriété de normalisation forte.

Dans l'exemple suivant, il est même possible de donner une preuve de `False`.

```
Inductive absurd2 : Type :=
  C : (absurd2 → False) → absurd2.

Definition Delta: absurd2 → False :=  [a:absurd2]
  Cases a of (C f) => (f (C f))
  end.

Definition omega : False := (Delta (C Delta)).
```

Pour éviter de tels paradoxes et garantir la normalisation forte, toute définition d'un type inductif I est restreinte de manière à ce que les constructeurs de I respectent la *condition de positivité* pour I. Une définition formelle de la *condition de positivité* pourra se trouver dans [5].

Pour le type inductif $Ind(I : A)[p_1 : T_{p_1} ... p_k : T_{p_k}]\{T_1 ... T_n\}$ dont le nombre d'arguments (le nombre de produits dans A) est g, les constructeurs sont de la forme
$\forall x_1 : X_1 .... \forall x_q : X_q.(I\ \overrightarrow{p}\ t_1..t_g)$, où les types X_i peuvent être récursifs, c'est-à-dire contenir I. La condition de positivité impose que les X_i qui sont récursifs soient de la forme $\forall z_1 : D_1 ... z_u : D_u, (I\ \overrightarrow{p}\ t'_1..t'_g)$ où I n'apparaît ni dans les D_j ni dans les t'_j.

Un constructeur qui aurait pour type `((A→B→I) → (C→I) → I)` satisferait la condition de positivité pour `I`.

En revanche, le constructeur `C : (absurd2 → False) → absurd2`
ne satisfait pas la condition de positivité pour `absurd2`, car X_1=`(absurd2 → False)` n'est pas de la bonne forme.

De même, `c : (absurd → absurd) → absurd` ne satisfait pas la condition de positivité pour `absurd`.

2.2.4 Règles de typage

Cette partie détaille les règles de typage du CCI où la sorte `Set` est prédicative. Ce modèle est appelé le pCIC, que la version v8.0 de Coq respecte. Par rapport au CCI imprédicatif, seule une restriction est faite dans la règle de typage du produit de type `Set`. Ces différences n'auront pas d'influence sur nos travaux.

Dans toute la suite, s dénotera une sorte, autrement dit $s \in S$.

Définition 2.2.5 *Une arité de sorte s est un terme convertible en s ou en un produit $\forall x : T.U$ où U est une arité de sorte.*

Dans les règles de typage en figure 2.2, un jugement de typage est noté

$$E[\Gamma] \vdash M : T$$

et se lit : le terme M a le type T dans l'environnement de typage Γ et l'environnement de déclarations E.

Nous utiliserons également un jugement de bonne formation des environnements de typage et de déclarations $\mathcal{WF}$ (pour *well-formed*).

$$\mathcal{WF}(E)[\Gamma]$$

La règle (Conv) permet de juger si deux termes sont intentionnellement égaux (convertibles).

Dans ces règles nous utilisons des abus de notation : $x \notin \Gamma$ signifie que x n'appartient pas au domaine de Γ. La notation $\{\overrightarrow{x}/\overrightarrow{u}\}$ représente une substitution $\{x_1/u_1 ... x_r/u_r\}$. Dans la règle (App), $\overrightarrow{\forall x : U}$ représente une liste de quantifications $\forall x_1 : U_1 ... \forall x_r : U_r$. De même $E[\Gamma] \vdash \overrightarrow{u} : \overrightarrow{U}$ est un abus d'écriture qui représente une liste de jugement de typage $E[\Gamma] \vdash u_1 : U_1, ..., E[\Gamma] \vdash u_r : U_r$.

2.2.5 Règle de l'analyse par cas

Nous ne détaillerons pas chacune des règles de typage, car elles se trouvent dans le manuel de référence de Coq. Nous détaillons seulement la règle (Case) qui est la plus complexe et également la plus fondamentale dans la manipulation des constructions inductives.

Schématiquement, l'expression $Case(e, P, f_1 ... fn)$ est bien typée si e a un type inductif $Ind(I : A)[\overrightarrow{p : T_p}]\{T_1..T_n\}$ à g arguments (donc A est de la forme $\forall a_1 : A_1 ... \forall a_g : A_g.A_I$ où A_I ne contient pas de produit), et si chacune des branches f_i de la forme $\lambda x_1 : X_1 ... x_{f_i} : X_{f_i}.e_i$ a un type de la forme $\forall x_1 : X_1 ... x_{f_i} : X_{f_i}.(P\ t_1 ... t_g\ e_i)$. L'expression $Case(e, P, f_1 ... fn)$ a alors le type $(P\ t_1 ... t_g\ e)$.

$$\frac{}{\mathcal{WF}(\emptyset)[\emptyset]}(\text{W-E}) \qquad \frac{E[\Gamma]\vdash T:s \quad x\notin\Gamma}{\mathcal{WF}(E)[\Gamma;x:T]}(\text{W-S})$$

$$\frac{E[\Gamma]\vdash t:T \quad c\notin\Gamma\cup E}{\mathcal{WF}(E;c:=t:T)[\Gamma]}(\text{W-D}) \qquad \frac{E[\Gamma]\vdash T:s \quad c\notin\Gamma\cup E}{\mathcal{WF}(E;c:T)[\Gamma]}(\text{W-A})$$

$$\frac{E[\Gamma;\overrightarrow{p:T_p}]\vdash A:s \quad I\notin\Gamma\cup E \quad (E[\Gamma;\overrightarrow{p:T_p};(I:A)]\vdash T_i:s_i)_{i=1\ldots n}}{\mathcal{WF}(E;(Ind(I:A)[\overrightarrow{p:T_p}]\{T_1\ldots T_n\}))[\Gamma]}(\text{W-Ind})$$

tel que les occurences de I dans les T_i sont appliquées aux paramètres $\overrightarrow{p}$

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma]\vdash Prop:Type}(\text{Ax1}) \qquad \frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma]\vdash Set:Type}(\text{Ax2}) \qquad \frac{\mathcal{WF}(E)[\Gamma] \quad i<j}{E[\Gamma]\vdash Type(i):Type(j)}(\text{Ax3})$$

$$\frac{\mathcal{WF}(E)[\Gamma] \quad (x:T)\in\Gamma}{E[\Gamma]\vdash x:T}(\text{Var}) \qquad \frac{\mathcal{WF}(E)[\Gamma] \quad (c:T)\in E}{E[\Gamma]\vdash c:T}(\text{Const})$$

$$\frac{E[\Gamma]\vdash T:s \quad E[\Gamma;x:T]\vdash U:Prop}{E[\Gamma]\vdash\forall x:T.U:Prop}(\text{Prod1})$$

$$\frac{E[\Gamma]\vdash T:s \quad s\in\{Prop,Set\} \quad E[\Gamma;x:T]\vdash U:Set}{E[\Gamma]\vdash\forall x:T.U:Set}(\text{Prod2})$$

$$\frac{E[\Gamma]\vdash T:Type(i) \quad i\leqslant k \quad E[\Gamma;x:T]\vdash U:Type(j) \quad j\leqslant k}{E[\Gamma]\vdash\forall x:T.U:Type(k)}(\text{Prod3})$$

$$\frac{E[\Gamma]\vdash\forall x:T.U:s \quad E[\Gamma;x:T]\vdash t:U}{E[\Gamma]\vdash\lambda x:T.t\ :\ \forall x:T.U}(\text{Lam}) \qquad \frac{E[\Gamma]\vdash t:\overrightarrow{\forall x:U.T} \quad E[\Gamma]\vdash\overrightarrow{u}:\overrightarrow{U}}{E[\Gamma]\vdash(t\ \ \overrightarrow{u}):T\{\overrightarrow{x}/\overrightarrow{u}\}}(\text{App})$$

$$\frac{\mathcal{WF}(E)[\Gamma] \quad Ind(I:A)[\overrightarrow{p:T_p}]\{T_1\ldots T_n\}\in E \quad 1\leqslant i\leqslant n}{E[\Gamma]\vdash Constr(i,I):\overrightarrow{\forall p:T_p}.T_i}(\text{Ind-Const})$$

$$\frac{\mathcal{WF}(E)[\Gamma] \quad Ind(I:A)[\overrightarrow{p:T_p}]\{T_1\ldots T_n\}\in E}{E[\Gamma]\vdash I:\overrightarrow{\forall p:T_p}.A}(\text{Ind-Type})$$

$$\frac{\begin{array}{c}Ind(I:A)[\overrightarrow{p:T_p}]\{T_1..T_n\}\in E \qquad E[\Gamma]\vdash e:(I\ \ \overrightarrow{q}\ \ t_1\ldots t_g) \qquad \sigma=\{\overrightarrow{p}/\overrightarrow{q}\}\\ E[\Gamma]\vdash P:B \qquad [(I\ \ \overrightarrow{q}):A\sigma\mid B] \qquad (E[\Gamma]\vdash f_i:\{Constr(i,I)\ \ \overrightarrow{q}\}^P)_{i=1..n}\end{array}}{E[\Gamma]\vdash Case(e,P,f_1\ldots f_n):(P\ \ t_1\ldots t_g\ \ e)}(\text{Case})$$

$$\frac{E[\Gamma]\vdash A:s \quad E[\Gamma;f:A]\vdash M:A}{E[\Gamma]\vdash Fix(f/n:A)\{M\}:A}(\text{Fix})$$

$$\frac{E[\Gamma]\vdash U:s \quad E[\Gamma]\vdash t:T \quad E\vdash T\leq_{\beta\delta\Phi\iota}U}{E[\Gamma]\vdash t:U}(\text{Conv})$$

FIG. 2.2 – Règles de typage du CCI

Examinons maintenant de manière plus formelle la règle (Case). Celle-ci nécessite une relation $[I : A \mid B]$, où I est un nom de type inductif, A et B sont des types. Elle sert à vérifier qu'un terme filtré de type I peut être éliminé grâce à une propriété P de type B. Cette relation se définit par les quatre règles de la figure 2.3.

$$\frac{[(I\ x){:}A \mid B]}{[I{:}\forall a{:}T.A \mid \forall a{:}T.B]}$$

$$\frac{s_1,s_2 \in S \qquad s_1 \neq Prop}{[I{:}s_1 \mid I \to s_2]}$$

$$\frac{}{[I{:}Prop \mid I \to Prop]}$$

$$\frac{I \text{ est vide ou singleton}}{[I{:}Prop \mid I \to s]}$$

Fig. 2.3 – Règles de bonne formation d'un schéma d'élimination.

Un type inductif est dit singleton s'il n'a qu'un constructeur et que tous les arguments de ce constructeur ont le type *Prop*.

Les paramètres dans la définition de I sont $\overrightarrow{p}$ mais les paramètres effectifs utilisés par e sont $\overrightarrow{q}$. La substitution $\sigma = \{\overrightarrow{p}/\overrightarrow{q}\}$ permet de faire ce remplacement.

La condition $[(I\ \overrightarrow{q}) : A\sigma \mid B]$ dans la règle (Case) où B est le type du schéma d'élimination P, s'assure que P commence par les mêmes produits $\forall a_1 : A_1 ... a_g : A_g$ que $A\sigma$, puis que P possède un argument de type $(I\ \overrightarrow{q}\ a_1..a_g)$.
Autrement dit le type B de P est de la forme $\forall a_1 : A_1 ... a_g : A_g.(I\ \overrightarrow{q}\ a_1 ... a_g) \to s$, où s est une sorte qui doit vérifier certaines conditions en fonction de la sorte de $(I\ \overrightarrow{q}\ a_1..a_g)$.

Le terme e doit avoir le type $(I\ \overrightarrow{q}\ t_1 ... t_g)$. Cela impose qu'une analyse par cas ne peut se faire que sur un terme inductif complètement instancié.

Il ne reste qu'à typer chaque branche de l'analyse. On introduit pour cela en figure 2.4 la notation $\{f : T\}^P$, abrégée en $\{f\}^P$, pour typer une branche d'analyse par cas f.

$$\{f : \forall x : U.T\}^P = \forall x : U.\{(f\ x) : T\}^P$$

$$\{f : (I\ \overrightarrow{p}\ t_1 ... t_g)\}^P = (P\ t_1 ... t_g\ f)$$

Fig. 2.4 – Type d'une branche d'analyse par cas.

Ainsi dans la règle (Case), la condition $(E[\Gamma] \vdash f_i : \{Constr(i, I)\ \overrightarrow{q}\}^P)_{i=1..n}$ est équivalente à :

pour tout constructeur $Constr(i,I)$ de type $\overrightarrow{\forall p : T_p}.\forall x_1 : X_1 ... x_q : X_q.(I\ \overrightarrow{p}\ t_1 ... t_g)$, $E[\Gamma] \vdash f_i : \forall x_1 : X_1\sigma ... x_q : X_q\sigma.(P\ t_1\sigma ... t_g\sigma\ (Constr(i,I)\ \overrightarrow{q}\ x_1 ... x_q))$.

2.2.6 Propriétés du CCI

- Une propriété attendue et vérifiée du CCI est la confluence (ou propriété de Church-Rosser).
  Si $t1 \rhd t2$ et $t1 \rhd t3$, alors il existe un terme $t4$ tel que $t2 \rhd t4$ et $t3 \rhd t4$.

- Nous avons énoncé dans la définition 2.2.4 la propriété de normalisation forte, que possède le CCI.

- Nous retrouvons la propriété d'affaiblissement du contexte de typage, que l'on retrouve dans tout système de types : si x n'a pas d'occurence libre ni dans t ni dans T et que $E[\Gamma; x : X] \vdash t : T$, alors $E[\Gamma] \vdash t : T$.

- Une autre propriété est que le type d'un terme bien typé admet au moins une sorte.
  Si $E[\Gamma] \vdash t : T$ alors il existe $s \in \{Set, Prop, Type\}$ tel que $E[\Gamma] \vdash T : s$.

- Une élimination d'un objet de sorte `Set` pour construire un objet de type `Type` est qualifiée d'élimination forte. C'est un des avantages du CCI par rapport au Calcul des Constructions. Dans le système Coq ce schéma est généré à partir de la définition du type inductif, et porte le suffixe `_rect`.

  L'élimination forte permet par exemple de montrer que $0 \neq 1$.
  Si on construit :

```
Definition disc :=  [n:nat]
    <[n:nat]Type>
    Cases n of  O => True
              | (S n) => False
    end
```

  on a `(disc 0)` convertible à `True` donc prouvable. Si on suppose $0 = 1$, alors de `(disc 0)` on en déduit par réécriture `(disc 1)` qui est convertible à `False`. Ainsi $0 = 1 \Rightarrow$ `False`, c'est-à-dire $0 \neq 1$.

  Cependant autoriser une élimination forte sur n'importe quel type inductif mène à un paradoxe (une version typée du paradoxe de Burali-Forti [24]). Ce paradoxe s'obtient dès qu'il est possible d'encapsuler un type dans un terme, puis de défaire l'encapsulation.
  Le type suivant permet d'encapsuler tous les autres :

```
Inductive U : Set :=
  encap : Set → U
```

Si l'élimination forte est autorisée sur ce type, on obtient le paradoxe de Burali-Forti.
La solution adoptée est de limiter l'élimination forte aux types inductifs dont les constructeurs sont dits *petits*.

Définition 2.2.6
Un constructeur est petit *si il est de la forme* $\overrightarrow{\forall p : T_p}.\forall x_1 : X_1 ... \forall x_q : X_q.(I\ \overrightarrow{p}\ t_1 ... t_g)$*, avec les* X_i *de type* Set.

Par exemple, le constructeur `Cons` de `list3` est petit car il est de type
`∀n:nat.A → (list3 A n) → (list3 A (S n))`
et on a bien `nat : Set`, `A : Set` et `(list3 A n) : Set`.
Par contre le constructeur `SSis_pair` de type : `∀n :nat.(is_pair n) → (is_pair (S (S n)))` n'est pas petit car `(is_pair n)` est de type `Prop`.

Ainsi, l'élimination forte est autorisée sur `list3`, mais pas sur `is_pair`.

- La propriété de stabilité du typage *subject-reduction*, assure que lorsqu'un terme u se réduit en un terme t, alors tout type de u est aussi un type de t. De plus, si U est le type de u et T type de t alors $T \leq_{\beta\delta\Phi\iota} U$.

Chapitre 3

Etat de l'art sur la réutilisation de spécifications et de preuves

La réutilisation de preuves a été largement abordée dans la littérature, le plus souvent dans le domaine de l'intelligence artificielle, ou bien dans le domaine de la preuve automatique. Plusieurs méthodes ou techniques ont été étudiées. Certaines nécessitent la modification du système de types sous-jacent, pour permettre la réutilisation, d'autres non. Nous allons présenter les principales approches que nous avons classées ainsi :

- *réutilisation par isomorphisme*
- *réutilisation par analogie*
- *réutilisation par généralisation*
- *réutilisation par sous-typage*
- *réutilisation par transformation*

Ce découpage peut paraître arbitraire, car la plupart des travaux que nous allons citer pourraient être classés dans plusieurs catégories. Par exemple, une méthode de réutilisation par transformation peut être basée sur un isomorphisme. De même, une méthode de réutilisation par généralisation peut être considérée comme une forme d'analogie.

3.1 Réutilisation par isomorphisme

Deux types A et B sont isomorphes si il existe deux fonctions $f : A \to B$ et $g : B \to A$, telles que $f^{-1} = g$ et $g^{-1} = f$. Un isomorphisme de type définit une équivalence entre types, permettant d'utiliser un type à la place d'un autre. Par exemple, le type $A \times B \to C$ et le type $A \to B \to C$ sont isomorphes. Les fonctions permettant de passer de l'un à l'autre étant les fonctions `curry` et `uncurry` ci-dessous. Ces deux fonctions sont inverses l'une de l'autre.

```
let curry f = function x → function y → f(x,y)
```

```
let uncurry f = function (x,y) → f x y
```

Les principales applications de la notion d'isomorphisme entre types concernent la recherche de composants logiciels (fonctions, spécifications, lemmes etc) dans une bibliothèque [30, 32, 82, 83].

Lorsque deux types sont isomorphes, une preuve portant sur le premier type peut être adaptée, donc réutilisée, pour porter sur le deuxième type.

Mathieu Jaume dans [44] propose une réutilisation d'une preuve de la décidabilité de l'unification sur les quasi-termes [84] pour prouver la décidabilité de l'unification sur des termes. L'idée est de définir l'isomorphisme (et son inverse) entre les termes et un sous-ensemble des quasi-termes appelés quasi-termes compatibles, caractérisé par un prédicat. Le fait que la propriété d'unification des quasi-termes, restreints aux quasi-termes compatibles est préservée par l'isomorphisme, fournit la preuve de décidabilité de l'unification sur des termes. Les preuves sur les quasi-termes sont réutilisées ici de manière "boîte noire", c'est-à-dire sans rien modifier ni rien savoir sur les preuves sur les quasi-termes. Cette méthode de réutilisation ne nécessite pas d'étendre le système de types.

Dans un cadre plus général, Gilles Barthe et Olivier Pons dans [9] explorent l'utilisation d'isomorphismes de types pour la réutilisation de preuves dans le CCI. Pour cela la théorie des types sous jacente est enrichie avec les isomorphismes sous la forme de règles de conversion. Ainsi étendu, le système autorise la réutilisation de preuves : pour prouver un lemme de type A, on pourra réutiliser un lemme ayant un type B isomorphe à A. Cette méthode permet par exemple le changement de représentation de données. L'exemple suivant illustre le passage d'une représentation des entiers à la Peano N, vers une représentation binaire B.

$$\begin{array}{ll} \mathit{Inductive}\ N : \mathit{Set} := & \mathit{Inductive}\ B : \mathit{Set} := \\ \quad O : N & \quad Bh : B \\ \mid S : N \to N. & \mid Bo : B \to B \\ & \mid Bi : B \to B. \end{array}$$

Le but est d'ajouter au système de types des règles de conversion entre les types N et B, notées $N =_\Sigma B$. La règle de typage suivante permet alors la conversion de type :

$$\frac{\Gamma \vdash e{:}N \quad \Gamma \vdash B{:}s}{\Gamma \vdash e{:}B}\ si\ N =_\Sigma B$$

où s est la sorte de B. Ainsi $S\ b$ avec b de type B sera correct du point de vue du typage. D'un point de vue calculatoire, le constructeur S sur N doit avoir son

équivalent sur le type B. Il faut donc définir une fonction successeur sur les binaires :

$$\begin{array}{rl} S^B : B \to B := & \\ [b : B]\ case\ b & of \quad Bh \Rightarrow Bo\ Bh \\ & \mid \quad Bo\ c \Rightarrow Bi\ c \\ & \mid \quad Bi\ c \Rightarrow Bo\ (S^B\ c) \end{array}$$

De même les constructeurs Bo et Bi sur B doivent avoir leur équivalent sur N :

$$Bo^N : N \to N = [n : N]\ (twice\ n)$$
$$Bi^N : N \to N = [n : N]\ S\ (twice\ n)$$

où $twice : N \to N$ est la multiplication par 2. Ainsi, il est possible d'écrire les fonctions de conversion entre chaque type :

$$\begin{array}{rl} N2B : N \to B := & \\ [n : N]\ case\ n & of \quad O \Rightarrow Bh \\ & \mid \quad S\ m \Rightarrow S^B\ (N2B\ m) \\ B2N : B \to N := & \\ [b : B]\ case\ b & of \quad Bh \Rightarrow O \\ & \mid \quad Bo\ c \Rightarrow Bo^N\ (B2N\ c) \\ & \mid \quad Bi\ c \Rightarrow Bi^N\ (B2N\ c) \end{array}$$

Pour pouvoir faire par exemple une analyse par cas sur $(Bo\ y)$ qui est de type B, en utilisant dans le filtrage les constructeurs de N, il faut ajouter une règle de réduction concernant le fitrage d'une expression de type B avec des filtres construits à l'aide de constructeurs de N et vice-et-versa :

$$case\ F\ of\ E_N \to_\sigma case\ (B2N\ F)\ of\ E_N$$
$$case\ G\ of\ E_B \to_\sigma case\ (N2B\ G)\ of\ E_B$$

où F est B_H, $(Bo\ e)$ ou $(Bi\ e)$, G est O ou $(S\ e)$, et où E_N et E_B sont des filtrages (associant des termes fO, fS, fh, fo, fi) respectivement de la forme :

$$O \Rightarrow fO \ \mid\ S\ x \Rightarrow\ fS$$
$$Bh \Rightarrow fh \ \mid\ Bo\ x \Rightarrow\ fo \ \mid\ Bi\ x \Rightarrow\ fi$$

Cette méthode de réutilisation est concluante, mais elle nécessite des preuves lourdes pour établir la confluence de la relation de réduction, le théorème de *Subject Reduction*, la décidabilité du typage, et la consistance du modèle. De plus, une mise en pratique de cette méthode dans Coq par exemple nécessiterait de modifier en profondeur les sources de celui-ci.

Dans notre contexte, ajouter ou supprimer un constructeur dans un type inductif ne permet pas en général de définir un isomorphisme entre le nouveau type et l'ancien. Donc la méthode de réutilisation par isomorphisme n'est pas adaptée à notre situation.

3.2 Réutilisation par analogie

La preuve par analogie consiste à trouver une preuve source qui permet d'établir un guide pour une preuve cible. Cette technique se décompose en deux aspects : la découverte d'analogies et l'exploitation des similarités pour transformer la preuve connue en la preuve recherchée. Les notions d'abstraction et de généralisation de théorème sont souvent associées à la preuve par analogie.

La difficulté est de trouver la preuve source qui serait similaire au problème que l'on veut traiter. Cela requiert du filtrage et de l'unification d'ordre supérieur, mais également une part de généralisation.

Erica Melis et Jon Whittle [59, 60] ont implémenté dans le système CLAM une technique de réutilisation par analogie appelée ABALONE. Le travail se place dans un context de *proof planing* où il s'agit de rejouer des plans de preuves. Ce type d'analogie est appelé analogie externe dans le sens où la preuve réutilisée provient d'une preuve d'un lemme déjà prouvé, donc externe. On parle d'analogie interne lorsqu'on réutilise la preuve d'un sous-but déjà prouvé, à l'intérieur d'une même preuve.

Amy Felty et Douglas Howe [36] proposent une technique de preuve par analogie, et de réutilisation de tactiques de preuves. Dans leur système, écrit en λ-prolog, une preuve est un arbre de séquents où chaque branche correspond à l'application d'une tactique. En généralisant chaque application de tactiques par l'ajout de métavariables, ils obtiennent une méthode de réutilisation après modification de l'énoncé du théorème. Par exemple, partant du fait qu'une preuve de $H \vdash A \wedge B$ commence par faire une preuve de $H \vdash A$ puis $H \vdash B$, pour réutiliser par analogie cette preuve pour démontrer $H \vdash A \wedge B \wedge C$, on commencera par découper le séquent en $H \vdash A$ et $H \vdash B \wedge C$. Le système sera capable de continuer la preuve de $H \vdash A$ en utilisant la généralisation faite dans la preuve de départ. Ensuite, quand on découpera $H \vdash B \wedge C$ en $H \vdash B$ et $H \vdash C$, le système utilisera la généralisation faite dans la preuve de départ pour prouver $H \vdash B$.

Thomas Kolbe et Jürgen Brauburger ont également implémenté un prouveur appelé PLAGIATOR [47] en Common Lisp au dessus du prouveur INKA [43]. Le système enrichit un dictionnaire de schémas de preuve au fur et à mesure que des preuves sont réalisées. Ce dictionnaire est interrogé pour essayer de faire la preuve en cours et donne la main s'il n'y parvient pas. La preuve est alors interactive, puis ajoutée dans le dictionnaire une fois terminée. Pour obtenir le schéma de preuve, la preuve est analysée puis généralisée. Pour instancier le schéma de preuve sur la nouvelle preuve, l'unification d'ordre supérieur sur les noms libres de fonctions est nécessaire.

3.3 Réutilisation par généralisation

Généraliser une preuve signifie pouvoir l'utiliser dans un cadre plus large. Le principe est d'abstaire la preuve pour la rendre plus générale et plus simple, puis de la réutiliser par instanciation dans un cadre précis.

Dans la sous-section précédente, nous avons déjà mentionné une méthode de réutisation par généralisation [36]. D'une certaine manière généralisation et analogie sont des méthodes de réutilisation qui se recoupent. Faire une preuve par analogie avec une preuve existante, c'est en quelque sorte généraliser la preuve existante.

Adolfo Villafiorita, Roberto Sebastiani et Fausto Giunchiglia [40] ont implémenté un prouveur interactif appelé ABSFOL permettant d'abstraire facilement une preuve et de la réutiliser en l'instanciant. Un contexte est formé d'un langage, d'axiomes et de règles d'inférence. Dans ce système, il faut définir un contexte de départ et un contexte d'arrivée. Puis on définit une fonction d'abstraction, qui fait le lien entre les symboles du contexte de départ et ceux d'arrivée. Un contexte abstrait est alors généré automatiquement. Partant d'une preuve faite en déduction naturelle dans le contexte de départ, un schéma de preuve est généré dans le contexte abstrait. Il suffit alors d'instancier les paramètres de ce schéma pour former la preuve dans le contexte d'arrivée.

Olivier Pons dans [76] décrit une méthode de réutilisation par généralisation et instanciation de théorèmes en théorie des types, implémentée dans COQ. Dans ce cadre, pour généraliser un théorème, il ne suffit pas d'abstraire les opérateurs que l'on veut généraliser. L'exemple donné est la permutativité de la multiplication :

$$\forall n, m, p : nat,\ n \times (m \times p) = m \times (n \times p) \quad (1)$$

Abstraire l'opérateur de multiplication est insuffisant, car l'énoncé obtenu est faux en général :

$$\forall f : nat \to nat \to nat,\ \forall n, m, p : nat,\ (f\ n\ (f\ m\ p)) = (f\ m\ (f\ n\ p))$$

Il faut en effet ajouter les hypothèses de commutativité et d'associativité de l'opérateur abstrait. L'auteur donne un algorithme permettant de calculer les propriétés à ajouter dans le théorème cible. L'idée est d'examiner le terme de preuve pour calculer les dépendances, à la manière de ce que nous faisons dans le chapitre 5. Si le type d'une des dépendances fait référence à l'identificateur que nous voulons abstraire, cette dépendance fait partie des hypothèses à ajouter. Cependant, cette technique peut produire des termes de preuve mal typés, lorsque la règle de ι-réduction est utilisée pour construire la preuve du théorème source. En effet, aucune trace de la règle de calcul utilisée (par exemple $(O \times m \to_\iota O)$ n'est laissée dans le terme de preuve. On peut néanmoins retrouver son utilisation en comparant le type attendu et le type inféré. La réutilisation du théorème généralisé, par exemple pour prouver la permutativité de l'addition,

$$\forall n, m, p : nat,\ n + (m + p) = m + (n + p) \quad (2)$$

s'effectue en instanciant le théorème généralisé par le nouvel opérateur (l'addition dans l'exemple) et par les propriétés correspondantes (commutativité et associativité de l'addition).

Plus récemment Christoph Lüth et Einar Broch Johnsen [45] ont implémenté dans Isabelle le même genre d'outil. Pour permettre la généralisation, les dépendances sont ajoutées en prémisses du théorème généralisé. Le calcul de ces dépendances utilise les termes produits à partir du script de preuve [11]. Les symboles de fonction et les types sont généralisés. Un lemme généralisé peut alors être instancié pour prouver un nouveau lemme.

Mathieu Jaume, Catherine Dubois et Jérôme Grandguillot dans [35] proposent une réutilisation de preuves formelles dans le cadre de Focal [79]. Focal est un environnement de calcul formel permettant de spécifier des propriétés, écrire des programmes et prouver la correction des programmes par rapport aux spécifications. Les preuves de propriétés peuvent être réalisées à l'aide de Coq ou à l'aide d'un prouveur spécialisé appelé Zenon [33]. Dans le cadre de l'algèbre classique, les propriétés sont souvent des théorèmes bien connus, éventuellement déjà formalisés dans le système Coq ou dans un autre système de preuve. Le premier écueil rencontré dans cette étude de la réutilisation de spécifications et de preuve Coq au sein de Focal concerne l'égalité : les spécifications Focal spécifient et utilisent une égalité définie comme étant une relation d'égalité sur un ensemble non vide, les spécifications Coq utilisent en général l'égalité prédéfinie de Leibniz. Les auteurs proposent un outil permettant de généraliser un développement Coq se fondant sur l'égalité de Leibniz en un développement générique paramétré par une relation d'égalité. Les propriétés de compatibilité des opérations qui sont des instances de l'égalité de Leibniz, deviennent des obligations de preuve à démontrer dans la preuve généralisée. Les preuves ainsi transformées sont réutilisables dans l'environnement Focal qui utilise des relations d'équivalences à la place des égalités.

Les techniques précédentes de généralisation pourraient être utilisées pour généraliser un type défini inductivement, par exemple *nat* dans (1). Si le terme de preuve utilise une expression de filtrage sur ce type ou plus généralement un récurseur, la preuve généralisée sera paramétrée entre autres par le type, ses constructeurs et le récurseur. Comme le mentionne Olivier Pons [76], ceci est lourd et peu souvent utile sauf dans la situation de réutilisation avec un type inductif isomorphe au type inductif du théorème source, ce qui n'est pas notre cas de figure.

3.4 Réutilisation par sous-typage

La motivation principale liée à l'ajout d'une relation de sous-typage, notée $<$, dans une théorie ou un langage est la volonté d'utiliser un objet d'un certain type là où un autre type est attendu. Ainsi la règle de typage d'intérêt dans ce cadre est la règle dite de *subsumption* :

$$\frac{\Gamma \vdash a{:}A \quad A{<}B}{\Gamma \vdash a{:}B}$$

La littérature sur ce sujet est très vaste dans le domaine de la programmation. La communauté "preuve" s'intéresse également à introduire ce mécanisme dans les assistants à la preuve. Par exemple, on trouve des implémentations du sous-typage dans Lego [50], Plastic [22] et Coq [85]. Il s'agit plus précisemment de sous-typage coercif où A est un sous-type de B si et seulement si une fonction de conversion est définie de A vers B.

Ainsi si C est la fonction de conversion de A vers B, A est donc un sous-type de B, et si f est une fonction de B vers D alors f peut être appliquée à tout objet a de type A. Par définition, $f\ a$ vaut $f\ (C\ a)$. Ainsi $f\ a$ est une abréviation pour $f\ (C\ a)$.

Dans [7], Gilles Barthe définit également un système de types (PTS) étendu avec des coercions implicites, où les jugements de typage sont accompagnés d'un contexte de coercions Δ. Ce système, partiellement implémenté dans Lego, possède les bonnes propriétés de conservativité, normalisation, et décidabilité du typage.

Etendre un type inductif I avec de nouveaux constructeurs établit une relation de sous-typage. En effet, tout élément construit avec les constructeurs de I peut aussi être défini avec les constructeurs du type étendu. Gilles Barthe dans [10] dans le cadre du Calcul des Constructions Inductives appelle ce sous-typage par constructeur *constructor subtyping*. Il montre que le CCI étendu avec le sous-typage par constructeur possède les bonnes propriétés : confluence, subject reduction, terminaison et décidabilité. Il n'y a cependant pas d'implémentation de ce calcul.

Erik Poll dans [74] définit une notion similaire de sous-typage entre types inductifs. L'ajout de constructeur peut être comparé à l'ajout de méthode dans une classe. Erik Poll montre qu'ajouter un constructeur et ajouter une méthode sont deux notions duales. En effet, ajouter une méthode produit un sous-type, mais ajouter un constructeur produit un super-type. De la relation de sous-typage $A < B$ entre A et B, on déduit que pour tout type C, on a $B \to C < A \to C$ à cause de la contravariance de $\to$. Donc une fonction f de B vers C ne pourra pas être utilisée à la place d'une fonction de A vers C, car la règle de subsumption ne s'applique pas. Par contre, on a $C \to A < C \to B$. Cet article se focalise exclusivement sur les programmes et non sur les preuves. Dans notre contexte de réutilisation de preuve, nous sommes malheureusement la plupart du temps dans la situation où $A < B$ et où nous voulons réutiliser une preuve de type $A \to C$. Donc le sous-typage ne nous permet pas de réutiliser nos preuves.

3.5 Réutilisation par transformation

L'objectif de ce type de méthode est de transformer l'objet preuve afin d'obtenir une seconde preuve dans un contexte différent. Ce nouveau contexte peut contenir de nouvelles définitions, de nouveaux lemmes ou de nouveaux types. Il peut également s'agir d'un contexte où l'on change la représentation d'un type de

données. Si dans une preuve un type est redéfini, il faut adapter cette preuve à la nouvelle représentation du type.

Une motivation peut être l'optimisation d'un programme extrait [2, 81] à partir de cette preuve, où la nouvelle représentation du type de données conduirait à une implémentation plus efficace. En se basant sur l'isomorphisme de Curry-Howard où preuve et programme sont mis en parallèles, [51] montre que transformer un programme est équivalent à transformer la preuve associée. Par exemple, un programme qui calcule y à partir de x est une preuve de $\forall x : u\ \exists y : t\ P(x, y)$ où P représente les spécifications du programme. Si $f : u \to t$ est la fonction extraite de la preuve, on a la propriété : $\forall x : u\ P(x, f(x))$. Un changement de représentation de données de u en u' et t en t' permet alors d'extraire une fonction $f' : u' \to t'$ portant sur les nouveaux types de données.

Une autre motivation pour un changement de représentation est de vouloir réutiliser un théorème portant sur un premier type de données, pour prouver un deuxième théorème portant sur un deuxième type de données. Nicolas Magaud et Yves Bertot [53, 54, 52] ont formalisé et implémenté un outil de réutilisation de preuves lors de changements de représentations de données en Coq. Cette réutilisation est basée sur le calcul d'un nouveau terme de preuve à partir du terme de preuve sur l'ancienne représentation. L'exemple suivant illustre comment adapter les preuves en passant d'une représentation *à la Peano* des entiers naturels *nat*, à une représentation binaire. Son outil permet de convertir toute la bibliothèque arithmétique de la distribution de Coq, basée sur le type *nat*.

Inductive nat : *Set* :=
  O : *nat*
| *S* : *nat* → *nat*.

La représentation binaire utilisée dans l'exemple utilise les types *pos* et *bin* :

Inductive pos : *Set* :=
  one : *pos*
| *pI* : *pos* → *pos* (*impair*)
| *pO* : *pos* → *pos*. (*pair*)

Inductive bin : *Set* :=
  zero : *bin*
| *bP* : *pos* → *bin*.

Le but est de transformer une preuve sur *nat* en une preuve sur *bin*. Les preuves par induction sur le type de données *bin* nécessiteraient de faire une induction aussi sur le type *pos*. Le nombre de cas ne serait donc pas le même que pour le type *nat*. Utiliser le principe d'induction classique sur *bin* changerait la preuve en profondeur. Un principe d'induction non structurel sur les entiers binaires est donc généré en utilisant une fonction successeur sur le type *bin*. La génération de ce principe

d'induction ainsi que sa preuve sont automatiques. L'outil remplace ainsi l'ancien principe d'induction dans l'ancien terme de preuve par une abstraction faisant usage du nouveau principe d'induction non structurel. Les fonctions utilisées dans le terme devront également subir une adaptation. Il y a cependant une difficulté à surmonter. En effet, comme nous l'avions mentionné en évoquant les travaux d'Olivier Pons [76], le système Coq applique des règles de simplification de manière transparente, sur les termes. Comme par exemple, des simplifications de fonctions ou des analyses par cas. L'outil de Nicolas Magaud génère des équations équivalentes à ces règles de simplification et les applique par des réécritures lorsqu'il faut dans le nouveau terme de preuve.

Contrairement à l'approche présentée dans [9], cette technique de réutilisation ne nécessite pas d'étendre le système de types de Coq, mais utilise simplement des transformations sur le terme de preuve.

3.6 Discussion

Malgré la diversité des techniques de réutilisation, nous pouvons constater que très peu d'outils implémentent ou automatisent réellement ces réutilisations. D'autre part, la réutilisation que nous proposons dans cette thèse ne se classe pas facilement parmi l'une ou l'autre des techniques présentées précédemment.

Il ne s'agit pas d'isomorphisme, car bien au contraire, après avoir ajouté un constructeur dans un type inductif, le nouveau type n'est pas isomorphe à l'ancien. Il ne s'agit pas de sous-typage car nous avons fait remarquer que nous créons un super-type et non un sous-type, lorsque nous ajoutons un constructeur. Il ne s'agit pas de généralisation, car l'ancienne preuve n'est pas abstraite puis réinstanciée. Nous pourrions classer notre méthode de réutilisation parmi les réutilisations par analogie. En effet, dans notre technique, nous avons bien une notion de "preuve similaire" ou "analogue". La différence entre les deux preuves, est qu'il faut complèter de nouveaux sous-buts. Nous pourrions également nous classer parmi les transformations, vu qu'il s'agit de transformer une ancienne preuve en une preuve nouvelle mais partielle.

Parmi les travaux qui se rapprochent le plus de notre contribution dans la réutilisation de preuves formelles, nous pouvons citer ceux d'Axel Schairer et Dieter Hutter [86]. Cette approche permet de faire évoluer un développement formel en appliquant des règles de transformation de base sur les spécifications. Parmi ces règles figurent l'ajout ou le retrait de types, de fonctions, d'axiomes et de lemmes, l'ajout ou le retrait de constructeurs dans un type, et l'ajout d'arguments dans le type des constructeurs d'un type. Ces changements se répercutent sur les preuves en créant des obligations de preuve sous la forme de nouveaux buts ouverts. Cette approche travaille sur la structure des arbres de preuve, et est très proche de notre travail en ce qui concerne l'ajout de constructeurs.

Nous avons également des points communs avec les travaux de Nicolas Magaud [52]. Nous avons la même approche consistant à modifier le terme de preuve Coq par un outil *ad hoc* sans modifier le système de type, en incluant cet outil dans le moteur du système Coq. Citons aussi [8] où Gilles Barthe et Pierre Courtieu proposent une tactique incorporée à Coq, pour générer un principe d'induction à partir d'une fonction récursive. Cela permet d'automatiser le raisonnement sur des spécifications exécutables car écrites avec des fonctions. Il ne s'agit pas de réutilisation mais le point commun avec notre approche est la transformation de λ-termes dans le moteur de Coq, et l'ajout de métavariables afin de générer les sous-buts désirés.

Chapitre 4

Un environnement de réutilisation de preuves

Ce chapitre décrit les différentes extensions ou modifications que nous souhaitons offrir à travers des commandes de base et décrit le comportement des outils de réutilisation, d'un point de vue utilisateur. Ainsi ces commandes forment un environnement de réutilisation de spécifications et de preuves.

4.1 Les commandes de base

L'environnement de réutilisation de preuves que nous voulons construire comporte différentes commandes de base permettant de modifier les spécifications et réutiliser les preuves. Voici la liste des commandes proposées, implémentées dans un prototype s'intégrant au système Coq :

- ajouter un constructeur à un type inductif
- supprimer un constructeur d'un type inductif
- paramétrer un type inductif
- ajouter un argument à un constructeur
- ajouter un argument à un type inductif
- mettre à jour un type inductif ou une définition
- réutiliser une preuve après une des modifications de base

L'environnement ainsi créé se rapproche des travaux de Axel Schairer et Dieter Hutter [86]. Ils proposent en effet des transformations de base pour faire évoluer un développement formel, dont l'ajout ou le retrait de constructeurs, ou l'ajout d'arguments dans un constructeur.

En combinant ces modifications élémentaires, il est possible de faire subir différentes modifications consécutives à un type inductif et par la suite de réutiliser les preuves en conséquence.

4.2 Ajouter un constructeur à un type inductif

4.2.1 La commande `Extend`

Un type inductif se caractérise par son nom, ses paramètres, son type, et la liste de ses constructeurs accompagnés de leur type. Le lecteur se reportera au chapitre 2 pour une présentation formelle d'un type inductif.

Ajouter des constructeurs consiste d'un point de vue de l'utilisateur à étendre la liste des constructeurs.

Lorsque `I` est défini par

```
Inductive I [p1:U1;...;pl:Ul] : T :=
  d1 : t1 | ... | dn : tn.
```

la syntaxe d'extension de `I` est de la forme :

```
Extend I as J
 with c1 : t'1 | ... | ck : t'k.
```

qui ajoute à l'environnement le type `I`, et les principes d'induction associés. Les noms `c1,...,ck` sont de nouveaux noms de constructeurs tous distincts, respectivement de type `t'1,...,t'k`. Ces types devront être bien typés et respecter la condition de positivité pour créer un type inductif bien formé.

Utiliser cette commande serait équivalent à définir manuellement le type `J` suivant :

```
Inductive J [p1:U1;...;pl:Ul] : T :=
  d1 : t1[I := J] | ... | dn : tn[I := J]
| c1 : t'1       | ... | ck : t'k.
```

où la substitution notée $[I := J]$ remplace les occurrences libres de `I` dans les types `t1,...,tn` par `J`.

Il est possible d'étendre `I` à la fois en `J` et en `K` de deux manières différentes. Les types `I`, `J` et `K` coexistent alors dans l'environnement.

Remarque Dans un système où la surcharge des constructeurs est autorisée, `I` et `J` pourront partager les mêmes noms de constructeurs. Dans le cas contraire, ce qui est le cas de Coq, la commande `Extend` se charge de renommer dans le type étendu les constructeurs communs, de manière à assurer leur unicité.

4.2.2 Réutiliser une preuve après ajout de constructeurs

Soit `L : ∀x :I.(P x)` un lemme portant sur des objets de type inductif `I`. On suppose que `I` est étendu en `J` avec k nouveaux constructeurs. L'objectif est de prouver le lemme `L` lorsque le type étendu `J` vient remplacer le type `I`. Notre méthode

de réutilisation consiste à réutiliser l'ancienne preuve de `L` afin de ne générer que les obligations de preuve correspondant aux k nouveaux constructeurs.

Ainsi, pour prouver le nouveau lemme `L' : ∀x :J.(P' x)`, où `P'` est la la mise à jour de `P` pour tenir compte du changement de `I` en `J`, la commande

```
Reuse L on J.
```

permet d'adapter la preuve de `L` afin d'obtenir une preuve partielle de `L'`. Le morceau "...`on J`" de la commande permet de déterminer les adaptations à réaliser. Par exemple, toutes les occurences de `I` dans l'ancienne preuve sont remplacées par `J` dans la nouvelle preuve. Les parties manquantes de la nouvelle preuve se retrouvent sous forme d'obligations de preuves, produites automatiquement par la commande `Reuse`. La preuve de ces buts générés terminera la preuve de `L'` de manière automatique.

Nous dirons alors que `L'` est une extension de `L` par rapport à l'extension de `I` en `J`, et nous le noterons $\texttt{L} \underset{I,J}{\rightsquigarrow} \texttt{L'}$.

4.3 Supprimer un constructeur d'un type inductif

4.3.1 La commande `Suppress`

Soit `I` le type inductif :

```
Inductive I [p1:U1;...;pl:Ul] : t :=
  d1 : t1 | ... | dn : tn.
```

L'utilisateur peut supprimer un constructeur avec la commande :

```
Suppress dj in I as J.
```

Celle-ci produit le type `J` équivalent au type `I` mais où le constructeur `dj` a disparu. Nous parlerons alors de type *restreint*. Cette commande serait équivalente à définir le type :

```
Inductive J [p1:U1;...;pl:Ul] : t :=
   d1 : t1[I := J] | ... | dj-1 : tj-1[I := J]
   dj+1 : tj+1[I := J] | ... | dn : tn[I := J]
```

où la substitution notée $[I := J]$ remplace les occurrences libres de `I` dans les types `t1,...,tn` par `J`.

Comme pour la commande `Extend`, les principes d'induction sur `J` sont automatiquement générés. Les contructeurs devront également être renommés pour que les types `I` et `J` puissent coexister.

4.3.2 Réutiliser une preuve après suppression

Pour prouver un lemme nommé `L'` sur un type restreint, la commande

```
Reuse L on J.
```

réutilise la preuve de `L`, en faisant les adaptations nécessaires par rapport à `J`.

Cette fois, aucune obligation de preuve n'a besoin d'être générée, et la preuve par réutilisation est complètement automatique.

Nous dirons que `L'` est une mise à jour de `L` par rapport à la restriction de `I` en `J`, ce que nous noterons encore `L` $\underset{I,J}{\rightsquigarrow}$ `L'`.

4.4 Paramétrer un type inductif

4.4.1 La commande `Param`

Un type inductif `I` peut avoir un ou plusieurs paramètres `p : u`. Dans ce cas, chaque occurrence de `I` dans le type des constructeurs de `I` est paramétrée par `p`.

Soit `I` le type inductif :

```
Inductive I [p1:U1;...;pl:Ul] : t :=
  d1 : t1 | ... | dn : tn.
```

Nous donnons ici la syntaxe de la commande permettant d'ajouter des paramètres supplémentaires à `I` :

```
Param I as J with k1:N1;...;kq:Nq.
```

ce qui ajoute dans l'environnement le type `J` qui aurait pu être obtenu directement par :

```
Inductive J [p1:U1;...;pl:Ul;k1:N1;...;kq:Nq] : t :=
  d1: t1[(I p1..pl) := (J p1..pl k1..kq)] | ...
| dn: tn[(I p1..pl) := (J p1..pl k1..kq)].
```

où la substitution $[(I\ p1..pl) := (J\ p1..pl\ k1..kq)]$ dans les types `ti`, renomme les occurrences de `I` par `J` et complète la liste des paramètres de `J`. Ainsi, chaque occurrence de `J` se retrouve avec une liste de paramètres supplémentaires $k1..kq$ qui n'ont aucune incidence sur le reste. Les principes d'induction sur `J` sont automatiquement générés par Coq.

Comme toujours, les constructeurs devront être renommés. Les noms des nouveaux paramètres `ki` doivent être frais et distincts de manière à ne pas capturer de variables libres dans la définition de `I`.

4.4.2 Réutiliser une preuve après paramétrisation

Pour prouver un lemme `L'` portant sur un type `J` que l'on vient de paramétrer, la commande

```
Reuse L on J.
```

adapte et réutilise la preuve de `L` en ajoutant les paramètres là où il faut. La preuve est complètement automatique, aucune obligation de preuve n'est générée.

Le lemme `L'` est une mise à jour de `L` par rapport à la paramétrisation de `I` en `J`. Nous le noterons également par `L` $\underset{I,J}{\rightsquigarrow}$ `L'`.

4.5 Ajouter des arguments

Rappelons que pour un type inductif la notion d'argument est moins restrictive que celle de paramètre. Les arguments d'un type inductif peuvent varier dans la définition du type, tandis que les paramètres sont constants. Par exemple dans le type `list3` donné au chapitre 2, le paramètre `A` est constant tandis que l'argument de type `nat` varie.

```
Inductive list3 [A:Set] : nat → Set :=
  Nil : (list3 A O)
| Cons: ∀n:nat.A → (list3 A n) → (list3 A (S n)).
```

4.5.1 Ajout d'arguments sur un constructeur avec `ArgumConstr`

Soit `I` le type inductif :

```
Inductive I [p1:U1;...;pl:Ul] : t :=
  d1 : t1 | ... | dn : tn.
```

La commande

```
ArgumConstr dj in I as J with a1:A1;...;aq:Aq.
```

permet de générer le type `J` équivalent à :

```
Inductive J [p1:U1;...;pl:Ul] : t :=
  d1: t1[I := J]
| ...
| dj: ∀a1:A1...∀aq:Aq. tj[I := J]
| ...
| dn: tn[I := J].
```

Le constructeur `dj` commence maintenant par `q` arguments respectivement de type `A1,...,Aq`. Il peut y avoir des dépendances entres les `Ai`, comme par exemple dans `∀n :nat.(l :(liste nat n))`. La seule contrainte est que les `ai` sont frais et distincts. Cette transformation de base est une des transformations proposées par Axel Schairer et Dieter Hutter dans [86].

4.5.2 Ajout d'arguments au type inductif avec `Argum`

Nous proposons maintenant un autre type d'ajout d'arguments. Il s'agit d'ajouter des arguments en tête du type d'un type inductif.

Soit `I` le type inductif :

```
Inductive I [p1:U1;...;pl:Ul] : t :=
  d1 : t1 | ... | dn : tn.
```

La commande

```
Argum I as J with a1:A1;...;aq:Aq.
```

génére le type `J` équivalent à :

```
Inductive J [p1:U1;...;pl:Ul] : ∀a1:A1...∀aq:Aq.t :=
  d1: ∀a1:A1...∀aq:Aq. t1[(I p1..pl) := (J p1..pl a1..aq)]
| ...
| dn: ∀a1:A1...∀aq:Aq. tn[(I p1..pl) := (J p1..pl a1..aq)].
```

où les substitutions $[(I\ p1..pl) := (J\ p1..pl\ a1..aq)]$ permettent d'ajouter les arguments `a1..aq` aux occurrences de `J`.

Les types des constructeurs commencent maintenant par les produits (ou quantifications) `∀a1 :A1...∀aq :Aq`, où les `ai` doivent être des noms frais et distincts, n'ayant aucune incidence sur les `ti`.

La différence par rapport à la transformation précédente est que chaque constructeur possédera les nouveaux arguments.

Remarque Le passage de `list` à `list3` (donnés au chapitre 2) correspond à l'ajout en tête d'un argument de type `nat`. Cet ajout ne correspond pas à notre commande, car nous avons fait le choix d'ajouter les arguments de manière constante en quantifiant le type des constructeurs.

La différence avec l'ajout de paramètres, est que si plus tard ce type est étendu par la commande `Extend`, chacun des nouveaux constructeurs pourra comporter des arguments différents, ce qui ne serait pas possible avec les paramètres.

4.5.3 Réutiliser une preuve après ajout d'arguments

Pour prouver un lemme portant sur un type auquel on a ajouté des arguments au type lui-même ou à un de ses constructeurs, la commande

```
Reuse L on J.
```

réutilise la preuve de `L` en l'adaptant pour tenir compte des modifications à faire pour changer `I` en `J`. Les quantifications et les nouveaux arguments sont ajoutés où il faut. Ainsi, lors de l'ajout d'arguments à un type inductif, la preuve est complètement automatique.

Pour des raisons techniques détaillée ultérieurement, lors de l'ajout d'arguments à un constructeur, il y a des obligations de preuves à décharger. Ces preuves se réduisent à montrer que les types des nouveaux arguments sont non vides.

Le lemme `L'` est une mise à jour de `L` par rapport à la modification de `I` en `J`, ce qui sera noté $\texttt{L} \underset{I,J}{\rightsquigarrow} \texttt{L'}$.

4.6 Mettre à jour types, définitions et axiomes

Supposons qu'un type inductif `I` ait été modifié par une commande de base en `J`. Supposons également qu'un type inductif `K` et une fonction `f` utilisent `I`. Pour réutiliser une preuve utilisant `f` ou `K`, on doit pouvoir être en mesure de mettre à jour `f` et `K` pour tenir compte du changement de `I` en `J`.

4.6.1 La commande `Update`

Voici un exemple d'utilisation de la commande `Update` qui illustre l'intérêt de cette commande. Etant donné les types inductifs `I` et `K` définis par :

```
Inductive I : Set :=
 c1 : I.

Inductive K : I → Prop :=
 c2 : (K c1).
```

les commandes suivantes

```
Extend I as J with d1:J.

Update K as K' on J.

Extend K' as K'' with d2:(K'' d1).
```

sont équivalentes respectivement aux définitions de :

```
Inductive J : Set :=
   c1' : J
 | d1  : J.

Inductive K' : J → Prop :=
  c2' : (K' c1').

Inductive K'' : J → Prop :=
  c2'' : (K'' c1')
| d2   : (K'' d1).
```

Pour obtenir `K''` "par extension" à partir de `K`, il est nécessaire d'utiliser la commande `Update K as K' on J` pour mettre à jour le type `I` → `Prop` en `J` → `Prop`.

Ainsi, pour mettre à jour un type inductif `K`

```
Inductive K [p1:U1;...;pl:Ul] : t :=
  d1 : t1 | ... | dn : tn.
```

la commande

```
Update K as K' on F1,...,Fn.
```

permet d'indiquer la liste des extensions `F1,..,Fn` à prendre en compte dans la mise à jour. Les `Fi` désignent des types ou définitions obtenus à partir d'une commande de base appliquée sur un objet `Ei` (un type inductif ou une définition). Notre commande `Update` serait équivalente à définir le type suivant :

```
Inductive K' [p1:U1[Fi]_i;...;pl:Ul[Fi]_i] : t[Fi]_i :=
  d1 : t1[K := K'][Fi]_i
| ...
| dn : tn[K := K'][Fi]_i.
```

où les constructeurs `d1...dn` seront automatiquement renommés. La notation $[Fi]_i$ représente la séquence de substitutions pour $i = 1..n$ de Ei par Fi, ou de (Ei $p1..pl$) par (Fi $p1..pl$ $a1..aq$) lorsque Fi est obtenu à partir d'un type inductif Ei à l'aide de la commande `Param Ei as Fi with a1 :A1...aq :Aq` ou de la commande `Argum Ei as Fi with a1 :A1...aq :Aq`.

Comme le type `K'` est une mise à jour de `K` par rapport aux extensions de `Ei` en `Fi`, pour chaque couple (`Ei`,`Fi`) nous déclarons que `K` $\underset{Ei,Fi}{\rightsquigarrow}$ `K'`.

4.6.2 Mettre à jour une définition

La commande `Update` s'applique également sur une définition, comme par exemple une fonction.

```
Update d as d' on F1,...,Fn.
```

permet d'ajouter à l'environnement la définition de nom `d'`, obtenue à partir du corps de la définition `d`, ayant subi les substitutions $[Fi]_i$.

Si la définition `d` ne contient aucune définition par cas sur un type inductif `Ei` étendu avec de nouveaux constructeurs, la définition de `d'` par la commande `Update` est automatique et complète.

Dans le cas contraire, les définitions par cas sur `Ei` présentes dans `d`, sont remplacées par des définitions par cas sur `Fi`. Comme `Fi` possède plus de constructeurs

et comme le filtrage d'une définition par cas doit être exhaustif, l'utilisateur sera invité à compléter les morceaux de définition manquants.

La définition `d'` est une mise à jour de `d` par rapport aux extensions de `Ei` en `Fi`. Pour chaque couple (`Ei`,`Fi`) nous déclarons que `d` $\underset{Ei,Fi}{\rightsquigarrow}$ `d'`.

Considérons par exemple une fonction `f : I` $\rightarrow$ `nat` contenant une définition par cas sur un type inductif `I`.

```
λx:I. Cases x of  c1   => n1
                | c2 y => Cases y of c1   => n2
                                   | c2 z => n3
                           end
      end
```

Puis supposons que `I` est étendu en `J` avec un constructeur `c3` de type `J` $\rightarrow$ `J`. La fonction `f` a besoin d'être mise à jour. En effet, d'une part il faut faire le renommage de `I` par `J`, et d'autre part les filtres dans la définition par cas doivent être complétés pour tenir compte du nouveau contructeur.

La commande :

```
Update f as f_J on J.
```

permet de définir une nouvelle fonction `f_J : J` $\rightarrow$ `nat` dans l'environnement à partir de `f`. La commande propose alors la définition :

```
λx:I. Cases x of  c1   => n1
                | c2 y => Cases y of c1   => n2
                                   | c2 z => n3
                                   | c3 x0 => ?
                           end
                | c3 x1 => ?
      end
```

L'utilisateur doit compléter localement la définition de la nouvelle fonction, afin que celle-ci soit acceptée.

4.6.3 Mettre à jour un axiome

La commande `Update` s'applique également sur un axiome :

```
Update a as a' on F1,...,Fn.
```

permet d'ajouter à l'environnement l'axiome de nom `a'`, dont le type est celui de `a`, ayant subi les substitutions $[Fi]_i$.

L'axiome `a'` est une mise à jour de `a` par rapport aux extensions de `Ei` en `Fi`, donc pour chaque couple (`Ei`,`Fi`) nous déclarons que `a` $\underset{Ei,Fi}{\rightsquigarrow}$ `a'`.

4.7 Combiner les différentes commandes et réutiliser

4.7.1 Combiner les modifications

Les commandes de base peuvent être combinées entre elles. Il est possible par exemple à la fois d'ajouter des constructeurs et d'ajouter des paramètres ou des arguments. Etant donné le type :

```
Inductive I [p1:U1;...;pl:Ul] : t :=
  d1 : t1 | ... | dn : tn.
```

La syntaxe

```
Argum I as K with a1:A1;...;aq:Aq
  and extend with c1 : t'1 | ... | ck : t'k.
```

est équivalente à la définition du type :

```
Inductive K [p1:U1;...;pl:Ul] : (a1:A1;...;aq:Aq) t :=
   d1 : (a1:A1;...;aq:Aq) t1[(I p1..pl) := (K p1..pl a1..aq)]
 | ...
 | dn : (a1:A1;...;aq:Aq) tn[(I p1..pl) := (K p1..pl a1..aq)]
 | c1 : t'1 | ... | ck : t'k.
```

Toutes les combinaisons possibles sont envisageables, il suffit pour cela d'utiliser plusieurs commandes de base l'une à la suite des autres. Par exemple la commande `Argum ... and extend ....` peut être simulée par l'utilisation de la commande `Argum` puis de la commande `Extend` :

```
Argum I as K' with (a1:A1;...;aq:Aq).
```

équivalent à :

```
Inductive K' [p1:U1;...;pl:Ul] : (a1:A1;...aq:Aq) t :=
   d1 : (a1:A1;...;aq:Aq) t1[(I p1..pl) := (K' p1..pl a1..aq)] | ...
 | dn : (a1:A1;...;aq:Aq) tn[(I p1..pl) := (K' p1..pl a1..aq)]
```

puis

```
Extend K' as K with c1 : t'1 | ... | ck : t'k.
```

équivalent au même type inductif que la commande `Argum ... and extend ....`

4.7.2 La commande Reuse après plusieurs modifications

Chaque type de modification de base implique un type de réutilisation de preuve. Lorsque n modifications sont effectuées à la suite, la réutilisation de preuve à travers la commande `Reuse` doit opèrer les n types de réutilisation correspondants.

Imaginons un cas où 5 modifications sont réalisées avant d'entammer la réutilisation des lemmes `M` et `L`. Soit `I` un type inductif, `K` un type inductif utilisant les constructeurs de `I`, et `f` une fonction définie par cas sur `I`.

```
Lemma M : ∀x:I.(P x).

Lemma L : ∀x:I.(M x) → (K x) → (f x) = x.
```

- Le type inductif `I` est paramétré en `I'` avec `Param`.
- Le type `I'` est étendu en `J0` avec `Extend`.
- Le type inductif `K` est mis à jour en `K'` avec `Update`.
- La fonction `f` est mise à jour en `f'` avec `Update`.
- Le type `J0` est étendu en `J1` avec `Extend`.

La nouvelle version de `M` peut être prouvée en utilisant `Reuse on J1`.

```
Lemma M' : ∀x:J1.(P x).
Reuse M on J1.
```

La nouvelle version de `L` se montre aussi en utilisant la commande `Reuse` :

```
Lemma L' : ∀x:J1.(M' x) → (K' x) →(f' x) = x.
Reuse L on J1,M',K',f'.
```

La commande `Reuse` a besoin de connaître la liste des types et définitions `J1,M',K',f'` à prendre en compte pour adapter la preuve de `L`. En effet, le type `J` doit être remplacé par`J1`, `M` par `M'`, `K` par `K'`, `f` par `f'`.

Préciser quels types et définitions doivent être pris en compte est nécessaire, car un type (ou définition) peut être étendu de plusieurs manières. Par exemple, la fonction `f` est ici mise à jour en `f'`, mais pourrait également être mise à jour en `f''`. Donc il y a un choix à décider dans la mise à jour de `f`.

La commande `Reuse` entame alors la preuve de `L'` en effectuant les réutilisations de base correspondant aux cinq modifications de base.

Le lemme `L'` doit avoir le même type que `L` modulo le renommage des types et définitions par leur version étendue données par la liste `J1,M',K',f'`. Cela requiert donc que l'utilisateur énonce correctement le nouveau lemme avant de lancer la commande de réutilisation.

Pour éviter toute erreur et pour apporter encore un peu plus d'aide dans la réutilisation, nous fournissons la commande

```
Reuse L as L' on J1,M',K',f'.
```

Cette commande énonce elle-même le lemme `L'` à prouver, en utilisant le type de `L` dans lequel les types et définitions sont remplacés par leur version étendue. Puis la méthode de réutilisation est appliquée comme précédemment.

Chapitre 5

Dépendances

Dans ce chapitre, nous présentons d'abord la notion de dépendance entre définitions et preuves d'un développement formel. Cette notion nous servira dans les chapitres suivants.

L'ensemble des dépendances peut être visualisé sous forme d'un graphe. Ce graphe aide l'utilisateur à comprendre le schéma global de la preuve, à prévoir les preuves futures, et enfin à gérer l'ordre dans lequel mener ses actions d'extensions ou de modifications. En distinguant une notion de dépendance opaque et de dépendance transparente, nous proposons un outil d'aide à la réutilisation qui permet de visualiser le graphe de dépendances d'un développement formel.

5.1 Notion de Dépendance

Lorsqu'on modifie un type dans un développement formel, par exemple en introduisant de nouveaux constructeurs, on veut pouvoir réutiliser les constantes, les fonctions, les preuves et les autres types du développement formel, en complétant éventuellement certaines preuves et définitions.

Les définitions et les lemmes à compléter pour réutiliser la preuve d'un lemme L sont ceux qui *dépendent* de L. Intuitivement, un lemme L dépend d'une définition ou d'un lemme L' si la définition ou le lemme L' est utilisé pour prouver L.

Le système Coq a la particularité de construire un terme de preuve, à partir d'un script de preuve. Ce terme est un λ-terme du CCI, au même titre qu'une définition. Un λ-terme peut contenir des identificateurs libres, représentant des objets définis auparavant, et figurant dans l'environnement de déclarations (fonctions, constantes, lemmes intermédiaires, types inductifs ou principes d'élimination). Ces identificateurs correspondent à la notion de dépendance que l'on cherche. Comme dans [77], nous calculons l'ensemble des dépendances d'un lemme L, en établissant la liste des identificateurs libres du terme de preuve de L. Dans la suite, nous identifierons un lemme avec son λ-terme.

Définition 5.1.1 *Si L_1 et L_2 sont deux identificateurs, nous dirons que L_2 dépend*

de L_1 lorsque L_1 a une occurrence libre dans le λ-terme de L_2.

Définition 5.1.2 *L'ensemble des dépendances d'un lemme L, noté $Dep(L)$, est l'ensemble des identificateurs libres dans le λ-terme L.*

La fonction *Dep* est étendue de manière à s'appliquer aussi sur un type inductif I. Les dépendances de I sont l'ensemble des dépendances de ses constructeurs. Autrement dit, si I est défini par $Ind(I : A)[\overrightarrow{p : T_p}]\{T_1 ... T_n\}$, alors $Dep(I) = Dep(T_1) \cup ... \cup Dep(T_n) \setminus \{I\}$.

Le calcul des dépendances est néanmoins possible dans un système où les preuves ne sont pas représentées par un λ-terme, mais uniquement par un script comme HOL, PVS, ou Isabelle (notons que pour Isabelle, il est possible de construire des termes de preuve grâce aux travaux de Stefan Berghofer et Tobias Nipkow [11]). Il suffit de disposer d'un script expansé, c'est-à-dire un script où uniquement des tactiques de base apparaissent. Olivier Pons dans [75] propose un algorithme d'expansion. Ainsi si le système permet une certaine automatisation, il est nécessaire de disposer des règles utilisées. Une analyse du script expansé d'un lemme permet alors de calculer l'ensemble des dépendances du lemme.

La relation $\underset{I,J}{\rightsquigarrow}$ a été introduite informellement dans le chapitre 4. Nous donnons ici une définition un peu plus formelle.

Définition 5.1.3 *Nous dirons que A est mis à jour (ou étendu) en B, noté $A \underset{I,J}{\rightsquigarrow} B$, si B est obtenu à partir des commandes* `Reuse` A `as` B `on` $J\ldots$ *ou* `Update` A `as` B `on` $J\ldots$ *telles que J est obtenu à partir des commandes* `Extend` I `as` $J\ldots$, `Suppress` I `as` $J\ldots$, `Param` I `as` $J\ldots$, `ArgumConstr` I `as` $J\ldots$ *ou* `Argum` I `as` $J\ldots$.

5.2 Dépendances Opaques et Transparentes

Dans un développement formel, si un type I est étendu et que le lemme L que l'on veut réutiliser dépend de I, une mise à jour de la preuve s'impose. Celle-ci devra éventuellement être complétée. Nous utilisons l'adverbe "éventuellement", car si un lemme L dépend d'un type étendu I, il est possible que la nouvelle preuve de L soit identique à la précédente. En effet, si la preuve de L utilise seulement le type de I et d'autres définitions et lemmes intermédiaires, le fait que I ait plus de constructeurs n'influe pas sur la preuve de L, excepté le renommage. Donc L dépend d'un type étendu I et pourtant la réutilisation de L ne génère pas d'obligations de preuve.

Ainsi, la notion de dépendance simple n'est pas assez fine pour déterminer s'il y a besoin de compléter une preuve ou non après une extension. Nous allons maintenant distinguer les notions de dépendances opaques et dépendances transparentes.

5.2.1 Définition

Définition 5.2.1 *Un objet L dépend d'un type inductif I de manière transparente, si L dépend d'un principe d'induction sur I ou si une analyse par cas sur un terme de type I a lieu dans le terme de preuve L.*

Quand la dépendance d'un lemme (ou d'une définition) L envers I est transparente, la preuve (ou la définition) de L après extension de I aura besoin d'être complétée. En effet, si la dépendance est transparente, une preuve par induction ou une analyse par cas est utilisée, donc la liste des constructeurs influe sur le terme.

Définition 5.2.2 *Un objet L dépend d'un type inductif I de manière opaque, si L dépend de I et que aucune dépendance envers I n'est transparente.*

Par définition, si L a une dépendance opaque envers I, la preuve (ou la définition) de L est insensible à la liste des constructeurs, donc identique à la preuve avant extension de I en J, modulo le renommage de I en J.

On retrouve cette famille de dépendances dans le langage Focal [78]. Un composant Focal contient des déclarations, de définitions, d'énoncés et de preuves. Une déclaration est la version abstraite d'une définition car on ne connait pas encore son implémentation. De même un énoncé est la version abstraite d'une preuve, c'est-à-dire une propriété non encore prouvée. Par un mécanisme d'héritage, les déclarations peuvent être raffinées en définitions, et les énoncés en preuves. Lorsqu'un composant B hérite de A, un calcul de dépendances détermine si les preuves de A peuvent être réutilisées dans B. Une *decl-dépendance* apparaît quand une preuve dépend d'une déclaration. Une *def-dépendance* apparaît quand une preuve dépend d'une définition. Si une preuve a une *def-dépendance* avec une définition d dans A, et que d est redéfinie dans B, la preuve a besoin d'être refaite dans B. Un parallèle peut donc être fait entre définitions opaque/transparente et *decl-dépendance/def-dépendance*.

Le système Coq [5] fait également une distinction entre définition opaque et transparente. Une définition d est dite transparente lorsque le corps de la définition est dépliable, permettant ainsi de substituer les occurrences de d par son corps. Des réductions peuvent alors être appliquées par le système comme la β-réduction, δ-réduction, la Φ-réduction et la ι-réduction. Une définition est dite opaque si son corps est caché. Par défaut, une définition est transparente tandis qu'un lemme est opaque. Cela respecte un principe habituel en mathématique, dénommé par l'anglicisme *proof irrelevance*, selon lequel la preuve d'un théorème n'influe pas sur les autres, seul son énoncé (son type) compte. A l'inverse, le terme d'une définition (comme une fonction par exemple) est important pour pouvoir faire du calcul (des réductions).

Dans notre calcul de dépendances, nous cherchons les occurrences des identificateurs libres dans le terme associé à un identificateur, même si ce dernier est opaque au sens de Coq. En effet, même si une définition (ou un lemme) est opaque, le

terme qui le définit doit être mis à jour afin d'avoir une définition correcte après une modification.

5.2.2 Prendre en compte toutes les preuves par induction

Notre calcul de dépendances transparentes prend en compte l'utilisation des principes d'induction créés automatiquement par le système Coq lors de la définition d'un type inductif, mais il doit également prendre en compte les principes d'induction définis par l'utilisateur. En effet, l'utilisateur peut générer un principe d'induction avec la commande `Scheme` ou `Functional Scheme`, ou même le définir lui-même, puis l'utiliser dans une preuve.

Voici un exemple tiré de [14]. Considérons le type inductif `ltree` des arbres formés de listes d'arbres :

```
Inductive ltree [A:Set] : Set :=
    lnode : A → (list (ltree A)) → (ltree A).
```

Le principe d'induction généré n'est pas satisfaisant car pour le constructeur `lnode`, nous n'avons pas les hypothèses d'induction sur le fait que la liste formant l'arbre contient des sous-termes de type `(ltree A)`.

Le principe d'induction que nous aimerions avoir est le suivant :

```
Definition ltree_ind :
 ∀A:Set.∀P:((ltree A)→Prop).∀Q:((list (ltree A))→Prop).
 (∀a:A.∀l:(list (ltree A)).(Q l)→(P (lnode A a l))) →
 (Q (nil (ltree A))) →
 (∀t:(ltree A).(P t)→∀l:(list (ltree A)).(Q l)→(Q (cons t l))) →
 ∀t:(ltree A).(P t).
```

Il est possible de donner directement un terme qui a ce type à l'aide de points fixes, comme le fait [14]. Une autre manière consiste à construire le terme en mode preuve. Voici un script qui permet cela :

```
Intros A P Q H H0 H1.
Fix 1.
Intro t; Elim t.
Intros a l; Apply H.
Induction l.
Apply H0.
Apply H1.
Apply ltree_ind.
Exact Hrecl.
Defined.
```

La dépendance calculée entre `ltree` et `ltree_ind` sera transparente. Il en est de même si la commande `Scheme` a été utilisée. A charge pour l'utilisateur de mettre à jour `ltree_ind` si `ltree` est modifié.

La commande `Functional Scheme`, implémentée par Gilles Barthe et Pierre Courtieu [8], permet de faire une preuve par induction fonctionnelle en générant un principe d'induction à partir d'une fonction récursive. A nouveau la dépendance calculée sur le principe d'induction généré par `Functional Scheme` sera transparente.

La tactique `Functional Induction` permet également de réaliser une preuve par induction fonctionnelle. Le schéma de preuve engendré par `Functional Induction f n` n'utilise pas de principe d'induction, mais le schéma suit la structure de la fonction récursive `f`. Prenons l'exemple de la fonction de division par 2.

```
Fixpoint div2 [n:nat] : nat :=
 Cases n of
  | O => O
  | (S n1) => Cases n1 with
               | O => O
               | (S n') => (S (div2 n'))
              end
  end.

Lemma div2_le : ∀n:nat.(div2 n) <= n.
Intro n.
Functional Induction div2 n; Auto.
```

le terme produit suit la structure de `div2` :

```
[n:nat]
 (Fix div2_ind [n0:nat] :  (div2 n0) <= n0 :=
   Cases n of
   | O      =>  ...
   | (S n1) => Cases n1 with
                | O      => ...
                | (S n') => ...
               end
   end) n.
```

Dans la mesure où la preuve de `div2_le` utilise une analyse par cas sur un terme de type `nat`, `div2_le` dépend de `nat` de manière transparente.

5.3 Graphe de Dépendances

Soit D l'ensemble des identificateurs déclarés dans l'environnement de déclaration d'un développement formel. Soit R la relation binaire induite par le calcul de dépendances sur D.

Le graphe de dépendances du développement formel est un graphe orienté par R, acyclique, dont les nœuds sont les éléments de D, et les arcs représentent les dépendances entre les éléments de D. Lorsqu'une dépendance entre un type inductif et une définition ou une preuve est transparente, l'arc est annoté par l'attribut "transparent".

Un tel graphe a une utilité intrinsèque car il contribue à la documentation du développement formel. Il est également utile pour la maintenance, en particulier pour la mise à jour des preuves après extension.

Le graphe peut être considéré comme un plan ou un schéma de preuve. Puisque les extensions de l'utilisateur sont supposées conservatives, le schéma de la preuve donné par le graphe avant extension est le même après extension. Ainsi avant de commencer ses réutilisations, l'utilisateur peut voir graphiquement l'ordre dans lequel les lemmes doivent être prouvés. L'ordre à respecter est l'ordre induit par le semi-treillis formé par le graphe.

Après la modification d'un type inductif, le graphe permet de connaître les définitions et les preuves complètement indépendantes du type modifié. L'utilisateur sait alors que dans son processus de réutilisation ces définitions et preuves pourront être réutilisées sans avoir besoin de les complèter ou de les renommer.

Après extension, nous pouvons à partir du graphe détecter quels nœuds, donc quelles définitions et preuves, seront identiques, quels nœuds devront seulement subir les renommages, et quels nœuds devront être complétés et dans quel ordre.

Nous ne faisons pas apparaître dans le graphe de liens entre A et B ni entre I et J lorsque $A \underset{I,J}{\rightsquigarrow} B$. De tels liens ne feraient que surcharger le graphe. De plus, cette relation d'extension est différente de la notion de dépendance que nous voulons faire apparaître dans le graphe.

Nous avons réalisé un prototype en OCaml s'intégrant à Coq : la commande `Dep` commentée dans le chapitre 11 lance le calcul de dépendances, puis une fenêtre affiche le graphe sous forme graphique.

Chapitre 6

Une première tentative de réutilisation de preuves

Dans ce chapitre nous proposons une première méthode de réutilisation de preuves après extension d'un type inductif, à l'aide du script de preuve. Nous commençons par montrer comment représenter une preuve partielle à l'aide d'arbres de tactiques, puis nous présentons la méthode de réutilisation.

6.1 Représentation d'une preuve partielle

Dans un processus de construction de preuve dirigée par les buts, interactive ou automatique, comme dans les systèmes d'aide à la preuve LCF [71], Coq [5], Lego [50], HOL [41], NuPRL [23], la construction se fait *ipso facto* pas-à-pas, de manière plus ou moins interactive. La preuve est obtenue par des raffinements successifs en appliquant des tactiques : une tactique, appliquée au but courant, le simplifie ou le découpe en plusieurs sous-buts plus simples à prouver. Une tactique peut être une tactique de base (telle que `Intro` ou `Case`), ou une tactique composite. Différents combinateurs sont fournis dans Coq, tels que les *tacticals* (`;` ou `orelse` par exemple), ou définis par l'utilisateur à l'aide du langage de tactiques Ltac [31], ou enfin des procédures de décision (`Omega` par exemple).

En cours de construction, le système ou l'utilisateur peut explorer une mauvaise voie, et donc décider de revenir en arrière pour en essayer une autre. À un instant donné quand la preuve est entamée, le système a besoin de représenter une preuve partielle, de pouvoir la faire évoluer vers une preuve complète et surtout de pouvoir revenir en arrière. Une preuve partielle est représentée par une structure d'arbre. Comme les branches entre les nœuds de l'arbre représentent l'application d'une tactique, l'arbre est appelé un arbre de tactiques, dont voici une définition possible :

Définition 6.1.1 *arbre de tactiques*

$$\begin{aligned} arbre &::= arbre; noeud \mid [] \\ noeud &::= (but, état, arbre) \\ état &::= ouvert \mid fermé(tactique) \end{aligned}$$

Un nœud $(\Gamma \vdash M : T, E, a)$ représente le but $\Gamma \vdash M : T$, dans un état E et a pour fils le sous-arbre a. Si l'état est *ouvert*, aucune tactique n'a été appliquée et le sous-arbre est vide (noté []). Si une tactique t a été appliquée sur le but, alors l'état E est $fermé(t)$ et le sous-arbre a correspond à la liste des sous-buts résultant de l'application de la tactique t sur le but. Un arbre de la forme

$$[(\Gamma \vdash M : T, E, a), fermé(t), [(\Gamma_1 \vdash M_1 : T_1, ouvert, []); ...; (\Gamma_n \vdash M_n : T_n, ouvert, [])]$$

se représente sous la forme d'un séquent de manière habituelle :

$$\frac{\Gamma_1 \vdash M_1 : T_1 \ .... \ \Gamma_n \vdash M_n : T_n}{\Gamma \vdash M : T} \ t$$

Définition 6.1.2 *Une preuve est partielle s'il existe un nœud dans l'arbre de tactiques dont le statut est ouvert.*

L'arbre de tactiques en Coq est construit et maintenu jusqu'à ce que la preuve soit complète. Un terme de preuve est calculé en parallèle et son type est vérifié une fois la preuve terminée. C'est ce terme qui est stocké dans un environnement, tandis que l'arbre est oublié.

6.2 Réutiliser l'arbre de tactiques

Nous proposons ici une première méthode de réutilisation de preuve basée sur la réutilisation des scripts de preuve [17], dont un prototype a été implémenté.

La première méthode à laquelle on pense pour réutiliser une preuve est de réutiliser son script de tactiques en l'adaptant et en le complétant. C'est ce que l'on fait habituellement "à la main" à l'aide d'un copier-coller. D'un point de vue utilisateur, une preuve est une liste de tactiques, mais bien que le script soit linéaire, il cache une structure d'arbre de tactiques comme nous l'avons fait observer.

6.2.1 Structure d'arbre de tactiques

La technique de réutilisation d'une preuve proposée ici requiert de garder trace de la structure de l'arbre de tactique. Pour cela nous annotons dans l'ancien script chaque tactique avec son chemin dans l'arbre de tactiques.

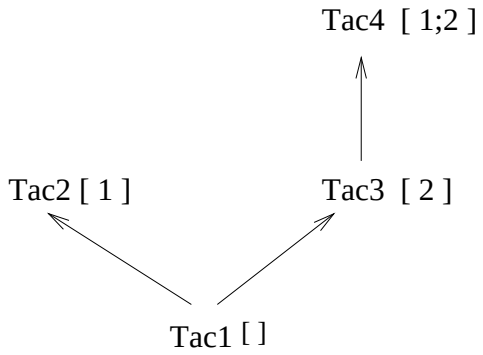
Définition 6.2.1 *Un chemin est une liste d'entiers qui repère la position d'un nœud dans un arbre. Nous définissons la fonction de chemin d'un nœud ν :*
Si ν est la racine de l'arbre, alors $chemin(\nu) = []$.
Si ν est le k-ième fils d'un nœud μ, alors $chemin(\nu) = k :: chemin(\mu)$.

Par exemple si le script suivant :

```
Tac1.
Tac2.
Tac3.
Tac4.
```

produit l'arbre $\dfrac{\dfrac{}{\Gamma_2 \vdash M_2{:}T_2}Tac2 \quad \dfrac{\dfrac{}{\Gamma_4 \vdash M_4{:}T_4}Tac4}{\Gamma_3 \vdash M_3{:}T_3}Tac3}{\Gamma_1 \vdash M_1{:}T_1}Tac1$

les chemins des tactiques `Tac1, Tac2, Tac3, Tac4` sont respectivement [], [1], [2], [1; 2]. Le script annoté par les informations de chemin permet de garder trace de la structure de l'arbre de tactiques :



6.2.2 Rejouer le script

Considérons une propriété L, conservative pour l'extension du type I. La structure de l'arbre de tactiques de L avant extension de I, est la même que celle de l'arbre de tactiques d'une preuve partielle de L après extension de I par de nouveaux constructeurs. Autrement dit les chemins sont préservés par extension.

En effet, quand une tactique concernant une définition inductive I, telle que `Case`, `Elim`, `Inversion` ou `Induction`, est appliquée à un but, les sous-buts générés sont présentés dans le même ordre que les constructeurs dans la définition de I. Quand on étend le type inductif I en ajoutant m nouveaux constructeurs, nous avons choisi d'ajouter ces nouveaux constructeurs *à la fin* du nouveau type inductif. Ceci est fondamental pour préserver les mêmes chemins lorsqu'on construit le nouvel arbre de tactiques.

Néanmoins, afin de préserver la structure de l'arbre, les tactiques utilisées doivent être monotones, comme l'explique Olivier Pons [75]. Une tactique monotone est une tactique dont le comportement est indépendant du contexte. Autrement dit, le résultat de l'application d'une tactique monotone est identique quelque soit l'environnement dans lequel elle est appliquée. En particulier, l'utilisation de tacticielles comme `Orelse` ou `Try` composées avec des tactiques automatiques comme `Auto` dépendent de la base de donnée des résultats utilisés par `Auto` (appelée liste de *Hints*). Donc si cette base est enrichie, les sous-buts produits après application d'une telle tactique peuvent être différents ou en moins grand nombre.

D'autre part, une tactique composée par la tacticielle ";" peut avoir un comportement différent après extension d'un type inductif. Par exemple, dans `Induction x ; Apply H` la tactique `Apply H` est appliquée à tous les sous-buts générés par `Induction x`. Quand le type de `x` est étendu, il y a de fortes chances pour que `Apply H` ne s'applique pas sur le nouveau sous-but généré par `Induction x`.

La solution à ces problèmes est de transformer toutes les tactiques composées en tactiques simples, en suivant l'algorithme d'expansion proposé dans [75]. Une tactique simple est une tactique qui ne comporte plus de tacticielle, ni de tactiques automatiques.

Les tactiques automatiques comme `Auto` doivent être expansées par les tactiques simples qui ont été utilisées. En effet, la tactique `Auto` peut très bien ne pas réussir à prouver un but après une extension.

De même, une tactique définie à l'aide du langage de tactiques *Ltac* devra pouvoir être remplacée par un ensemble de tactiques simples.

En manipulant le script de cette manière, nous nous assurons que chaque tactique de l'ancien arbre de tactiques se retrouvera au même chemin dans le nouvel arbre de tactiques.

Le nouvel arbre de tactiques est construit en appliquant successivement les anciennes tactiques annotées, précisément aux nœuds du nouvel arbre dont le chemin correspond à l'annotation. La seule différence entre l'ancien arbre de tactiques et le nouveau réside dans le fait que certains nœuds ont des fils supplémentaires de statut *ouvert*. Ces nouveaux sous-buts sont autant d'obligations de preuve que l'utilisateur aura à décharger en proposant les tactiques idoines.

Pour tous les sous-buts communs, comme la tactique à appliquer figure dans l'ancien arbre, celle-ci est appliquée et le statut du nœud est *fermé*.

Si la tactique appliquée ne concerne pas le type inductif étendu, elle produit exactement autant de sous-buts que dans l'ancien arbre, donc aucune obligation de preuve n'est générée.

La figure 6.1 est un exemple de structure d'arbre de tactiques d'une preuve L de $\forall e : I.(P\ e)$.

Le type de e est un type inductif I à 2 constructeurs que l'on étend en J avec un troisième constructeur. Ainsi pour prouver $\forall e : J.(P\ e)$, on rejoue les tactiques aux

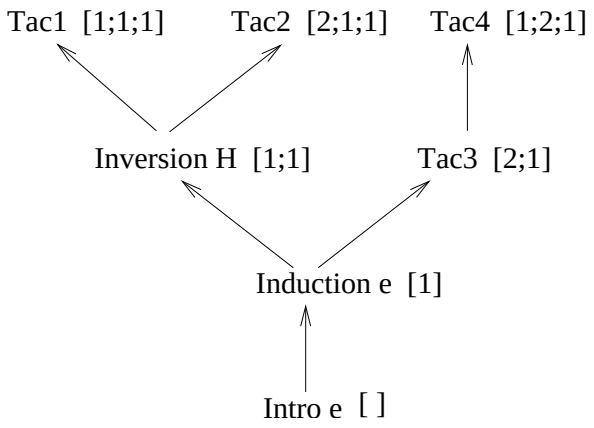


FIG. 6.1 – Exemple de structure d'arbre de tactiques

mêmes chemins que dans l'ancien arbre. La structure de l'arbre de tactiques obtenu se trouve en figure 6.2.

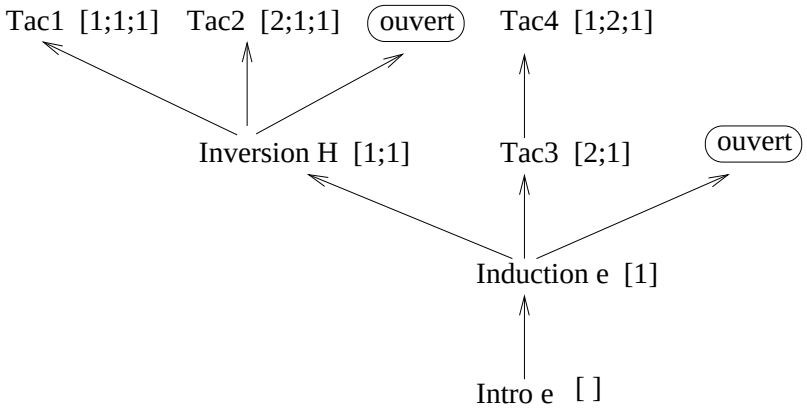


FIG. 6.2 – Structure d'arbre de tactiques après extension

Au nœud $[1;1]$ la tactique `Inversion` a produit un nouveau sous-but en $[3;1;1]$. De même, en $[1]$ `Induction` a produit un nouveau sous-but en $[3;1]$. Ces deux nouveaux sous-buts forment deux obligations de preuve.

6.3 Avantages et inconvénients de cette approche

Cette approche comporte quelques défauts. Le premier est que le script de l'ancienne preuve doit être joué pour être annoté par les chemins. Ensuite le script est rejoué pour construire le nouvel arbre de tactiques et pour obtenir la liste des obligations de preuve. Devoir jouer deux fois le script peut être coûteux en temps.

Un autre inconvénient est qu'il est nécessaire de transformer le script de tactiques avant de l'annoter, en expansant toutes les tacticielles.

Un dernier inconvénient concerne l'utilisation de définitions par cas sur un type étendu. Le succès de la réutilisation suppose que la fonction soit étendue convenablement. La méthode de réutilisation de script ne permet pas d'étendre une défintion, à moins que celle-ci soit définie en mode preuve à l'aide de tactiques.

Cette approche a l'avantage de pouvoir traiter facilement la réutilisation de preuve après des extensions de type. La suppression de constructeurs est également envisageable, mais demande un peu de travail sur les rangs des constructeurs. En effet, après avoir supprimé le $i^{ème}$ constructeur, une tactique produisant une branche par constructeur génère une branche de moins, et toutes les branches de rang j supérieur à i correspondent aux branches de rang $j+1$ de l'ancien arbre. Les chemins ne sont pas préservés après une suppression, mais il est possible de recalculer les nouveaux chemins en prenant en compte le décalage.

Cependant traiter d'autres genres de modification, comme paramétrer le type ou ajouter des arguments, nécessiterait une analyse beaucoup plus fine. En effet, en présence de nouvelles quantifications sur les arguments ou paramètres, la structure de l'arbre de preuve est modifiée, donc les chemins ne sont pas préservés. Il faudrait pouvoir calculer avec exactitude les modifications dans les chemins afin de trouver comment réutiliser les tactiques.

Pour remédier à ces défauts, nous proposons dans les chapitres suivants une autre méthode qui utilise directement les λ-termes représentant les preuves, appelés termes de preuve dans la suite.

Cette approche par le script permet néanmoins une réutilisation mécanique et sûre. C'est une manière simple de réutiliser des preuves, facile à implémenter et adaptable à tout assistant à la preuve orienté script comme PVS [87], NuPRL [23] ou HOL [41] où notre seconde méthode ne serait pas envisageable.

Chapitre 7

Réutiliser une preuve après extension d'un type inductif

Nous présentons dans ce chapitre notre approche de la réutilisation de preuve après extension d'un type inductif par de nouveaux constructeurs, basée sur la réutilisation de termes de preuve dans le CCI. Nous expliquons d'abord comment les métavariables permettent de représenter une preuve partielle. Nous expliquons ensuite notre principe de réutilisation avant de le formaliser.

7.1 λ-calcul avec métavariables

7.1.1 Construction de terme de preuve

Généralement une preuve est construite "de bas en haut", c'est-à-dire en partant de la formule à prouver. Par exemple, pour prouver $A \to B$, on applique la règle logique ($\to_{intro}$) de bas en haut. On se retrouve alors à prouver B sous l'hypothèse A.

$$\frac{\Gamma;A \vdash B}{\Gamma \vdash A \to B}(\to_{intro})$$

Afin de construire le terme de preuve en même temps que l'arbre de tactiques, et donc avoir un terme de preuve partielle, il faut pouvoir également construire le terme "de bas en haut". Ainsi pour prouver $A \to B$, on applique la règle du CCI (Abs) de bas en haut.

$$\frac{\Gamma;(x{:}A) \vdash t{:}B}{\Gamma \vdash \lambda x{:}A.t \ : \ A \to B}(\text{Abs})$$

Le terme t n'est pas encore connu donc on ne peut *a priori* pas l'instancier. Mais on a l'intention de l'instancier plus tard, donc en attendant on utilise un symbole ?. Le terme $\lambda x : A.? : B$ est alors une preuve incomplète de $A \to B$, où ? représente un terme de type B sous hypothèse que $x : A$ est dans le contexte. Ce symbole ? est

un véritable "trou" (appelé *place-holder* dans la littérature) dans la preuve, destiné à être rempli par la suite.

Pour représenter ces termes à trous dans le λ-calcul, il est nécessaire d'introduire une nouvelle notion de variables appelées *métavariables*.

Dans le système d'assistant à la preuve ALFA [56] la notion de métavariable est intégrée au λ-calcul sous-jacent. La notion d'arbre de tactiques n'existe pas, une définition ou la preuve d'un lemme est construite pas-à-pas en définissant un terme progressivement. Un but dans ce système correspond à une métavariable, que l'on remplace par un terme contenant éventuellement d'autres métavariables qui seront à leur tour remplacées, jusqu'à ce qu'il n'y ait plus de métavariables.

Par exemple, pour définir les fonctions booléennes *impair* et *pair* sur le type des entiers naturels Nat, on utilise des métavariables pour donner leur type et des métavariables pour donner leur définition :

$$impair \in ?_{10} = ?_{12}$$

$$pair \in ?_{11} = ?_{13}$$

Puis la métavariable $?_{10}$ est remplacée par $?_{14} \rightarrow ?_{15}$.
La métavariable $?_{14}$ est remplacée par $(a \in Nat)$.
La métavariable $?_{15}$ est remplacée par $Bool$.
On obtient ainsi

$$impair \in \ (a \in Nat) \rightarrow Bool = ?_{12}$$

On fait de même pour *pair* afin d'obtenir

$$pair \in \ (a \in Nat) \rightarrow Bool = ?_{13}$$

Pour donner la définition de la fonction *impair*, on remplace la métavariable $?_{12}$ par $\lambda\ a \rightarrow ?_{18}$ pour avoir

$$impair \in \ (a \in Nat) \rightarrow Bool = \ \lambda\ a \rightarrow ?_{18}$$

puis $?_{18}$ est remplacée par $case\ a\ of\ 0 \rightarrow ?_{19} \mid S\ n \rightarrow ?_{20}$.
$?_{19}$ est remplacée par $False$ et $?_{20}$ par $pair\ n$
On obtient

$$impair \in \ (a \in Nat) \rightarrow Bool = \ \lambda\ a \rightarrow case\ a\ of\ 0 \rightarrow False \mid S\ n \rightarrow pair\ n$$

On fait de même pour *pair* :

$$pair \in \ (a \in Nat) \rightarrow Bool = \ \lambda\ a \rightarrow case\ a\ of\ 0 \rightarrow True \mid S\ n \rightarrow impair\ n$$

Les termes de preuve se définissent de la même manière.

Quand une preuve est réalisée dans le système COQ à l'aide d'un arbre de preuve et de tactiques, un terme de preuve à trou est également construit à l'aide de métavariables. Lorsque l'arbre de preuve est complet, il n'y a plus de métavariables dans le terme.

7.1.2 Instantiation de métavariables

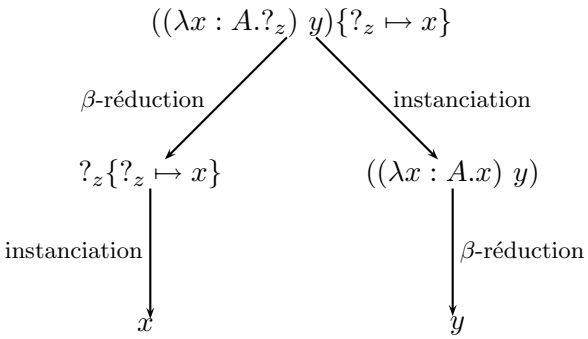
Dans [62], César Muñoz donne une formalisation générale de la notion de métavariables. En particulier, il montre pourquoi les variables ordinaires du λ-calcul sont insuffisantes. Supposons que le symbole ? soit une variable. Pour construire une preuve de $A \to A$ à partir de $\lambda x : A.?$, il faudrait substituer ? par x. Or cela est impossible car l'opération de substitution n'autorise pas la capture de variable. Il faut donc introduire un nouveau mécanisme : l'instantiation de métavariables.

Dans la suite, les métavariables seront dénotées $?_x$, et appartiennent à un ensemble dénombrable χ.

Définition 7.1.1 *Une instantiation de la métavariable $?_x \in \chi$ dans le terme t_1 par le terme t_2, notée $t_1\{?_x \mapsto t_2\}$, remplace toutes les occurrences de la métavariable $?_x$ par t_2 dans t_1.*

7.1.3 Instanciation et β-réduction

Si on ne prend pas de précautions la β-réduction ne commute pas avec l'instantiation. Par exemple, $((\lambda x : A.?_z)\ y)\{?_z \mapsto x\}$ se β-réduit en $?_z\{?_z \mapsto x\}$ puis se réduit par instanciation en x. Mais $((\lambda x : A.?_z)y)\{?_z \mapsto x\}$ se réduit par instanciation en $((\lambda x : A.x)\ y)$ qui se β-réduit en y.



Autrement dit la relation de réduction ne serait plus confluente. La solution consiste à ne pas autoriser de β-réduction sur un terme $(\lambda x : A.t)\ t'$ si t contient une métavariable.

Une autre solution hors de notre contexte, le calcul des substitutions explicites présenté par César Muñoz [63] ou Lena Magnusson [55], permet de gérer les substitutions en tant qu'opération syntaxique, interne au calcul. Le β-rédex $(\lambda x : A.M)\ N$ se réduit alors en $M[x := N]$ où la substitution n'est pas encore effectuée. Les règles

de réductions des substitutions peuvent alors être restreintes de manière à ne pas réduire une métavariable.

7.1.4 Signature

La conservation du type par instanciation, n'est pas garantie si aucune condition n'est ajoutée. Par exemple, supposons que $x : A \vdash (\lambda x : B.?_z) : B \to A$ avec $?_z : A$. Si on instancie $?_z$ par x, on obtient $x : A \vdash (\lambda x : B.x) : B \to B$ et donc le type n'est pas conservé.

Pour garantir la conservation du typage par instanciation, à chaque métavariable $?_z$ on associe son type T_z, le contexte Γ_z où $?_z$ a été définie et la règle de typage implicite

$$\frac{}{\Gamma_z \vdash ?_z : T_z}(Meta_z)$$

Définition 7.1.2 *Un raffinement de $?_x$ par t_1 dans t_2 transforme le terme t_2 en $t_2\{?_x \mapsto t_1\}$, à condition que $\Gamma_x \vdash t_1 : T_x$ et $\Gamma_x \vdash x : T_x$.*

Pour accéder à la liste des règles implicites de typage des métavariables, on définit la notion de signature.

Définition 7.1.3 *Une signature Σ est une liste de déclarations de métavariables $\Gamma_x \vdash ?_x : T_x$*

Pour typer un λ-terme t avec métavariables, on a besoin bien sûr du contexte de typage Γ pour les variables, mais aussi de la signature Σ de toutes les métavariables de t. On note alors le jugement de typage $\Sigma \triangleright \Gamma \vdash t : T$, mais par abus de notation, nous omettrons Σ quand elle n'a pas d'importance.

Proposition 7.1.1 *Préservation du type par raffinement*
Si $\Sigma \triangleright \Gamma_x \vdash t_1 : T_x$ et $(\Sigma; \Gamma_x \vdash ?_x : T_x) \triangleright \Gamma \vdash t_2 : T$, alors le raffinement de la métavariable $?_x$ par le terme t_1 donne $\Sigma \triangleright \Gamma \vdash t_2\{?_x \mapsto t_1\} : T$.

Autrement dit, instanciation et typage commutent, ou encore, un jugement de typage obtenu après raffinement d'un jugement de typage valide est valide.

Si on reprend l'exemple $x : A \vdash (\lambda x : B.?_z) : B \to A$, il faut ajouter la signature. Ici $?_z$ est utilisé dans le contexte $x : B$, donc la signature Σ est $(x : B \vdash ?_z : A)$. Le terme t par lequel on voudra instancier $?_z$ devra donc être typé ainsi $x : B \vdash t : A$.

7.2 Etendre un type inductif

Soit I un type inductif

$$Ind(I : A)[\Gamma_p]\{\overrightarrow{T}\}$$

défini dans l'environnement de déclarations E. Etendre le type I par une liste de m nouveaux constructeurs de type $\overrightarrow{M}$, consiste à ajouter dans l'environnement E le nouveau type :

$$Ind(J : A)[\Gamma_p]\{[\![\overrightarrow{T}]\!]_1\ \overrightarrow{M}\}$$

où l'opération $[\![.]\!]_1$ définie figure 7.1 permet de remplacer les occurences de I par J.

Les principes d'induction sur J (nommés `J_ind,J_rec,J_rect` en Coq) sont également générés et ajoutés dans E. Si le type I n'était formé que de constructeurs *petits*, alors l'élimination forte est autorisée sur le type I. Cela se traduit par la génération du principe d'induction nommé `I_rect`. Dans ce cas, nous imposons que les constructeurs ajoutés pour construire J soient également petits, de manière à ce que `J_rect` soit généré et ajouté à E.

7.3 Une approche fondée sur la réutilisation de λ-termes

Dans le chapitre 5, nous avons expliqué que l'extension d'un type inductif I a des incidences sur les objets qui dépendent de I. En particulier, les preuves par induction sur I ont besoin d'être complétées. La contribution de notre travail est d'automatiser la recherche des lemmes et définitions à compléter, et de générer les obligations de preuve nécessaires.

Nous nous intéresserons bien sûr à des extensions conservatives, de sorte que les propriétés prouvées dans le développement formel initial restent prouvables après extension selon le même schéma de preuve. Si tel n'était pas le cas, notre méthode de réutilisation créerait des sous-buts non prouvables.

Le principe de réutilisation d'un terme de preuve est simple. Il consiste d'une part à mettre à jour le terme de preuve en renommant les noms des objets étendus par le nom de leur extension.

D'autre part, comme le filtrage d'une définition par cas doit être exhaustif, on complète les filtrages sur un type étendu en ajoutant des métavariables qui tiennent lieu de trous, afin d'obtenir une preuve partielle du lemme. Il faut en effet ajouter les nouveaux cas introduits par l'ajout de constructeurs. Les expressions associées à ces nouveaux cas sont des métavariables fraîches toutes distinctes.

Cette preuve partielle est utilisée, par l'intermédiaire de la tactique `Refine` de Coq, de manière à ce que chaque trou génère un sous-but à prouver.

En plus du renommage des noms des types, des définitions et des lemmes qui ont été étendus, nous renommons les constructeurs et les principes d'induction associés aux types inductifs étendus. Pour un principe d'induction sur un type étendu I appliqué à une propriété P de type $I \rightarrow Prop$, nous ajoutons parmi les arguments de l'application de nouvelles métavariables correspondant aux preuves de P appliquées aux nouveaux constructeurs.

Ce principe de réutilisation de terme est aussi valable pour mettre à jour le terme d'une définition. En effet, une définition contenant une analyse par cas sur un objet de type I, utilise un filtrage exhaustif sur les constructeurs de I. Donc si I est étendu, il faut compléter le filtrage. Le graphe de dépendances indiquerait d'ailleurs une dépendance à compléter, car une telle dépendance est transparente.

Pour définir la définition étendue, nous procédons comme pour le terme d'une preuve. Reprenons l'exemple de la fonction `f :I -> nat` définie par :

```
λx:I. Cases x of  c1   => n1
                | c2 y => Cases y of c1   => n2
                                   | c2 z => n3
                           end
      end
```

Le type I est ensuite étendu en J avec un constructeur `c3` de type `J -> J`.

Nous construisons alors `f_J` à partir de `f` en mettant à jour les noms dans le λ-terme `f`, et en complétant le filtrage de `x` et de `y` en ajoutant des métavariables notées `?1` et `?2`.

Le λ-terme proposé pour la définition de `f_J :J -> nat` est le suivant :

```
λx:J. Cases x of   c1'   => n1
                 | c2' y => Cases y of c1' => n2
                                     | c2' z => n3
                                     | c3 x0 => ?1
                           end
                 | c3 x1 => ?2
      end
```

Le système génère alors des obligations de preuve pour chaque métavariable qui correspondent aux morceaux de définition manquants.

7.4 Formalisation de la modification de λ-termes

Nous formalisons ici les modifications sur les λ-termes, pour réutiliser des preuves après extension de la liste des constructeurs de types inductifs.

La définition formelle de l'opération d'extension (ou de modification) $[\![.]\!]_1$ des termes du CCI est donnée figure 7.1.

– La transformation $[\![.]\!]_1$ est implicitement paramétrée par I et J.

– $[\![\Lambda]\!]_1$ est le λ-terme Λ où les occurrences du type inductif I sont remplacées par J, et où les constructeurs de I sont remplacés par les constructeurs de J correspondants.

$[\![s]\!]_1$	$= s$	
$[\![x]\!]_1$	$= x$	
$[\![I]\!]_1$	$= J$	
$[\![c]\!]_1$	$= c'$	si $I \in Dep(c)$ et $c \underset{I,J}{\rightsquigarrow} c'$
	$= c$	sinon
$[\![\Pi x : M.N]\!]_1$	$= \Pi x : [\![M]\!]_1.[\![N]\!]_1$	
$[\![\lambda\ x : M.N]\!]_1$	$= \lambda\ x : [\![M]\!]_1.[\![N]\!]_1$	
$[\![I\_ind\ \ \overrightarrow{q}\ Q\ \overrightarrow{N}\ \overrightarrow{t}\ e]\!]_1$	$= J\_ind\ \ [\![\overrightarrow{q}]\!]_1\ [\![Q]\!]_1\ [\![\overrightarrow{N}]\!]_1\ \overrightarrow{X}\ [\![\overrightarrow{t}]\!]_1\ [\![e]\!]_1$	
$[\![I\_rec\ \ \overrightarrow{q}\ Q\ \overrightarrow{N}\ \overrightarrow{t}\ e]\!]_1$	$= J\_rec\ \ [\![\overrightarrow{q}]\!]_1\ [\![Q]\!]_1\ [\![\overrightarrow{N}]\!]_1\ \overrightarrow{X}\ [\![\overrightarrow{t}]\!]_1\ [\![e]\!]_1$	
$[\![I\_rect\ \overrightarrow{q}\ Q\ \overrightarrow{N}\ \overrightarrow{t}\ e]\!]_1$	$= J\_rect\ [\![\overrightarrow{q}]\!]_1\ [\![Q]\!]_1\ [\![\overrightarrow{N}]\!]_1\ \overrightarrow{X}\ [\![\overrightarrow{t}]\!]_1\ [\![e]\!]_1$	
$[\![M\ \overrightarrow{N}]\!]_1$	$= [\![M]\!]_1\ [\![\overrightarrow{N}]\!]_1$	si $M \neq I\_ind, I\_rec, I\_rect$
$[\![Constr(i,c)]\!]_1$	$=Constr(i, [\![c]\!]_1)$	
$[\![Case(e,P,\overrightarrow{N})]\!]_1$	$= Case([\![e]\!]_1, [\![P]\!]_1, [\![\overrightarrow{N}]\!]_1\ \overrightarrow{X})$	si $e : (I\ \overrightarrow{q}\ \overrightarrow{t})$
	$= Case([\![e]\!]_1, [\![P]\!]_1, [\![\overrightarrow{N}]\!]_1)$	sinon
$[\![Fix(f/n : A)\{M\}]\!]_1$	$= Fix(f/n : [\![A]\!]_1)\{[\![M]\!]_1\}$	

FIG. 7.1 – Définition de $[\![\Lambda]\!]_1$ quand I est étendu en J

– Dans $[\![\overrightarrow{N}]\!]_1$, l'opération de transformation $[\![.]\!]_1$ est appliquée sur tous les éléments de $\overrightarrow{N}$.

– Nous notons $\overrightarrow{X}$ une liste de métavariables fraîches (déclarées dans la signature), de longueur m, le nombre de nouveaux constructeurs de J.

– Pour toute constante c dont dépend L ($c \in Dep(L)$ [1]), telle que c dépend de I ($I \in Dep(c)$), c est remplacée par la constante c' telle que $c \underset{I,J}{\rightsquigarrow} c'$, supposée avoir été définie au préalable, et par conséquent présente dans l'environnement de déclarations. Une constante c peut être une définition, un axiome ou un type inductif. Le calcul du graphe de dépendances est donc nécessaire afin d'avoir étendu toutes les constantes dont L dépend avant d'étendre L.

– Puisque dans une analyse par cas le filtrage est exhaustif, $[\![.]\!]_1$ complète la liste des clauses de filtrage des expressions de type $(J\ \overrightarrow{q})$ en ajoutant les cas des nouveaux constructeurs de J. Les valeurs associées sont des métavariables fraîches et distinctes, déclarées dans la signature Σ par effet de bord. Lorsque les métavariables $?_1 ... ?_m$ sont ajoutées dans une analyse par cas sur e de type $(J\ \overrightarrow{q}\ \overrightarrow{t})$, typée par $E[\Gamma] \vdash Case(e, P, f1...fn) : (P\ \overrightarrow{t}\ e)$ avec une signature Σ, alors les métavariables $?_j$ sont déclarées dans Σ par :

$$E[\Gamma] \vdash ?_j : \{Constr(n+j, J)\ \overrightarrow{q}\}^P \text{ pour } j = 1..m.$$

[1] La définition de *Dep* est donnée dans la définition 5.1.2

La notation $\{Constr(n+j, J)\ \overrightarrow{q}\}^P$ définie figure 2.4 désigne le type d'une branche d'une analyse par cas.

– Pour un type inductif

$$Ind(I : \forall a_1 : A_1 ... \forall a_g : A_g . A_I)[p_1 : T_{p_1}; ...; p_k : T_{p_k}]\{T_1\ ...\ T_n\}$$

le type du principe d'induction généré sur la sorte $Prop$, noté T_{I_ind}, a la forme suivante :

$$\begin{array}{l}
\forall p_1 : T_{p_1} ... \forall p_k : T_{p_k}. \\
\forall P : (\forall a_1 : A_1 ... \forall a_g : A_g . (I\ p_1 ... p_k\ a_1 ... a_g) \rightarrow Prop). \\
(... \rightarrow (P\ a_1 ... a_g\ (Constr(I,1)\ p_1 ... p_k\ ...))) \rightarrow \\
... \\
(... \rightarrow (P\ a_1 ... a_g\ (Constr(I,n)\ p_1 ... p_k\ ...))) \rightarrow \\
\forall a_1 : A_1 ... \forall a_g : A_g. \\
\forall e : (I\ p_1 ... p_k\ a_1 ... a_g). \\
(P\ a_1 ... a_g\ e)
\end{array}$$

Définition 7.4.1
Dans le type T_{I_ind} du principe d'induction I_ind, nous noterons $Principe(I, i)$ le type $(... \rightarrow (P\ a_1 ... a_g\ (Constr(I, i)\ p_1 ... p_k\ ...)))$, type de la preuve à donner pour le constructeur numéro i.

Le type $Principe(I, i)$ dépend du type du constructeur $Constr(I, i)$: pour chaque occurence de I dans les arguments du type de $Constr(I, i)$, il y aura une prémisse dans $Principe(I, i)$.
Dans [14], les types $Principe(I, i)$ sont appelés *prémisses principales* du principe d'induction. Les produits $\forall a_1 : A_1 ... \forall a_g : A_g . \forall e : (I\ p_1 ... p_k\ a_1 ... a_g)$ sont appelés *épilogue* ou conclusion du principe d'induction. Ces quantifications sont là pour généraliser le principe d'induction à tous les habitants du type inductif.

Nous faisons l'hypothèse, comme c'est le cas dans la pratique, que les principes d'induction sont appliqués à tous leurs arguments. Nous décomposons alors la liste des arguments appliqués à un principe d'induction ainsi : $(I\_ind\ \overrightarrow{q}\ Q\ \overrightarrow{N}\ \overrightarrow{t}\ e)$, où $\overrightarrow{q}$ désigne les paramètres effectifs de I, Q est la propriété sur laquelle le principe d'induction est utilisé, les types $\overrightarrow{N}$ correspondent aux preuves de Q pour les n constructeurs de I (les étapes d'induction de type $Principe(I, i)$), et $\overrightarrow{t}$ et e sont les arguments correspondant à l'épilogue.
L'opération $[\![.]\!]_1$ remplace les principes d'induction I_ind, I_rec, I_rect respectivement par J_ind, J_rec, J_rect, et ajoute parmi la liste des arguments appliqués au principe d'induction, après les preuves correspondant aux n anciens constructeurs, m métavariables fraîches et distinctes $?_1 ... ?_m$.
En effet, si le type étendu J possède m nouveaux constructeurs, J_ind aura un type T_{J_ind} de la forme :

$\forall p_1 : T_{p_1} ... \forall p_k : T_{p_k}.$
$\forall P : (\forall a_1 : A_1 ... \forall a_g : A_g.(J\ p_1 ... p_k\ a_1 ... a_g) \rightarrow Prop).$
$Principe(J, 1) \rightarrow$
$...$
$Principe(J, n) \rightarrow$
$Principe(J, n+1) \rightarrow$
$...$
$Principe(J, n+m) \rightarrow$
$\forall a_1 : A_1 ... \forall a_g : A_g.$
$\forall e : (J\ p_1 ... p_k\ a_1 ... a_g).$
$(P\ a_1 ... a_g\ e)$

De plus, l'opération $[\![.]\!]_1$ ajoute par effet de bord dans la signature Σ, la déclaration de ces métavariables. Si $E[\Gamma] \vdash J\_ind : T_{J\_ind}$ alors $[\![.]\!]_1$ déclare dans Σ :

$$E[\Gamma] \vdash ?_j : Principe(J, n+j) \text{ pour } j = 1..m$$

Définition 7.4.2 *L'opération $[\![.]\!]_1$ est étendue aux contextes de typage de la manière suivante :*

$$\begin{array}{lcl} [\![\emptyset]\!]_1 & = & \emptyset \\ [\![\Gamma; x : T]\!]_1 & = & [\![\Gamma]\!]_1; x : [\![T]\!]_1 \end{array}$$

Autrement dit, $[\![.]\!]_1$ appliquée à Γ s'applique à tous les types des variables contenues dans Γ.

Définition 7.4.3 *L'opération $[\![.]\!]_1$ s'étend également aux environnements de déclarations lorsque I est étendu en J avec des constructeurs de types $\overrightarrow{M}$:*

$$\begin{array}{lcl}
[\![\emptyset]\!]_1 & = & \emptyset \\
[\![E; c : T]\!]_1 & = & [\![E]\!]_1;\ c : T;\ c' : [\![T]\!]_1 \qquad si\ I \in Dep(c)\ et\ c \underset{I,J}{\rightsquigarrow} c' \\
 & = & [\![E]\!]_1;\ c : T \qquad sinon \\
[\![E; c := d : T]\!]_1 & = & [\![E]\!]_1;\ c := d : T; \\
 & & c' := [\![d]\!]_1\{?_i \mapsto M_i\}_i : [\![T]\!]_1\ si\ I \in Dep(c)\ et\ c \underset{I,J}{\rightsquigarrow} c' \\
 & = & [\![E]\!]_1;\ c := d : T \qquad sinon \\
[\![E; Ind(I : A)[\overrightarrow{p : T_p}]\{\overrightarrow{T}\}]\!]_1 & = & [\![E]\!]_1;\ Ind(I : A)[\overrightarrow{p : T_p}]\{\overrightarrow{T}\}; \\
 & & Ind(J : A)[\overrightarrow{p : T_p}]\{[\![\overrightarrow{T}]\!]_1\ \overrightarrow{M}\}; \\
 & & J_ind := \Lambda_{J_ind} : T_{J_ind}; \\
 & & J_rec := \Lambda_{J_rec} : T_{J_rec}; \\
 & & J_rect := \Lambda_{J_rect} : T_{J_rect} \\
[\![E; Ind(K : u)[\overrightarrow{r : R}]\{\overrightarrow{U}\}]\!]_1 & = & [\![E]\!]_1;\ Ind(K' : [\![u]\!]_1)[\overrightarrow{r : [\![R]\!]_1}]\{[\![\overrightarrow{U}]\!]_1\} \\
 & & \qquad si\ I \in Dep(K)\ et\ K \underset{I,J}{\rightsquigarrow} K' \\
 & = & [\![E]\!]_1; Ind(K : u)[\overrightarrow{r : R}]\{\overrightarrow{U}\} \qquad sinon
\end{array}$$

La transformation d'un environnement de déclarations E correspond à un enrichissement de E. Nous ne détaillons pas le calcul des types $\Lambda_{J\_ind}, T_{J\_ind}, \Lambda_{J\_rec}, T_{J\_rec}, \Lambda_{J\_rect}$ et T_{J_rect} car ce n'est pas notre propos. Nous laissons le soin à Coq de les générer à partir de la définition de J. La notation $[\![d]\!]_1\{?_i \mapsto M_i\}_i$ signifie que toutes les métavariables apparaissant dans $[\![d]\!]_1$ ont été instantiées correctement.

7.5 Correction de la preuve par réutilisation de λ-termes modifiés

La proposition 7.5.1 établit la correction de la méthode de réutilisation vis-à-vis du typage. Nous supposons disposer d'une propriété ϕ prouvée par le terme Λ dans un environnement de déclaration E_I contenant le type inductif I, et un contexte de typage Γ. Ainsi on a $E_I[\Gamma] \vdash \Lambda : \phi$. Le type I est étendu en J, et nous supposons que les dépendances de Λ qui dépendent de I, $\{c \in Dep(\Lambda) \mid I \in Dep(c)\}$, sont déjà étendues pour le type J. Le nouveau lemme que nous voulons prouver aura le type $[\![\phi]\!]_1$. Nous montrons que l'application de l'opération de transformation $[\![.]\!]_1$ sur Λ, produit un terme qui a le type recherché à condition de pouvoir instancier convenablement les métavariables introduites par $[\![.]\!]_1$.

Pour rappel, la notation $M\{?_i \mapsto M_i\}_{i=1..z}$, ou plus simplement $M\{?_i \mapsto M_i\}_i$, désigne l'instantiation de toutes les métavariables apparaissant dans M.

Proposition 7.5.1

On suppose que

$$E_I[\Gamma] \vdash \Lambda : \phi$$

avec $Ind(I : A)[\overrightarrow{p : T_p}]\{\overrightarrow{T}\} \in E_I$. *On suppose que le type* I *est étendu en* J *par* m *constructeurs de type* $\overrightarrow{M}$ *et que* $\mathcal{WF}([\![E_I]\!]_1)[[\![\Gamma]\!]_1]$.
Si la signature produite par le calcul de $[\![\Lambda]\!]_1$ *est* $\Sigma = ([\![E_I]\!]_1[\Gamma_i] \vdash ?_i : \tau_i)_{i=1..z}$, *et si pour tout* $i \in 1..z$, $[\![E_I]\!]_1[\Gamma_i] \vdash M_i : \tau_i$ *alors on a*

$$[\![E_I]\!]_1[[\![\Gamma]\!]_1] \vdash [\![\Lambda]\!]_1\{?_i \mapsto M_i\}_{i=1..z} : [\![\phi]\!]_1$$

Dans la suite, nous noterons $[\![E_I]\!]_1$ par E_J. Nous avons donc

$$\begin{aligned} E_J \;=\; & E_I \;;\; Ind(J : A)[\overrightarrow{p : T_p}]\{[\![\overrightarrow{T}]\!]_1\ \overrightarrow{M}\}; \\ & J\_ind := \Lambda_{J\_ind} : T_{J\_ind};\ J\_rec := \Lambda_{J\_rec} : T_{J\_rec};\ J\_rect := \Lambda_{J\_rect} : T_{J\_rect}; \\ & c'_1 := [\![d_1]\!]_1\{?_i \mapsto M_i\}_i : [\![t_1]\!]_1;\ \ldots\ ;\ c'_w := [\![d_w]\!]_1\{?_i \mapsto M_i\}_i : [\![t_w]\!]_1\ ; \\ & c'_{w+1} : [\![t_{w+1}]\!]_1\ ; \ldots ;\ c_v :: [\![t_v]\!]_1\ ; \\ & Ind(K'_1 : [\![u_1]\!]_1)[\overrightarrow{r_1 : [\![R_1]\!]_1}]\{[\![\overrightarrow{U_1}]\!]_1\}; \ldots ; Ind(K'_h : [\![u_h]\!]_1)[\overrightarrow{r_h : [\![R_h]\!]_1}]\{[\![\overrightarrow{U_h}]\!]_1\} \end{aligned}$$

où pour tout $i \in 1...v$, $c_i \underset{I,J}{\rightsquigarrow} c'_i$, et tout $i \in 1...h$, $K_i \underset{I,J}{\rightsquigarrow} K'_i$
lorsque $\{c \in Dep(\Lambda) \mid I \in Dep(c)\} = \{c_1, ..., c_w, c_{w+1}, ..., c_v, K_1, ..., K_h\}$

et que $c_1 := d_1 : t_1; ...; c_w := d_w : t_w;\ c_{w+1} : t_{w+1}; ...; c_v : t_v \in E_I$, et
$Ind(K_1 : u_1)[\overrightarrow{r_1 : R_1}]\{\overrightarrow{U_1}\}; ...; Ind(K_h : u_h)[\overrightarrow{r_h : R_h}]\{\overrightarrow{U_h}\} \in E_I$.

Preuve

Il faut établir que $[\![\Lambda]\!]_1\{?_i \mapsto M_i\}_{i=1..z}$ est une preuve de $[\![\phi]\!]_1$ dans le contexte $[\![\Gamma]\!]_1$ et l'environnement E_J. La preuve de cette proposition se fait par induction sur les dérivations de typage de $E_I[\Gamma] \vdash \Lambda : \phi$. Les règles de typage se trouvent page 27, examinons chaque cas :

- (Ax1)
  Dans ce cas, on a $\Lambda = Prop$ et $\phi = Type$, donc $[\![\Lambda]\!]_1 = Prop$ et $[\![\phi]\!]_1 = Type$.
  Il n'y a pas de métavariables introduites dans $Prop$, donc il suffit d'établir :
  $E_J[[\![\Gamma]\!]_1] \vdash [\![\Lambda]\!]_1 : [\![\phi]\!]_1$.
  On l'obtient en appliquant la règle (Ax1), vu que l'on a $\mathcal{WF}(E_J)[[\![\Gamma]\!]_1]$ par hypothèse.

- (Ax2), (Ax3)
  Ces cas sont identiques au précédent.

- (Var)
  Dans ce cas $\Lambda = x$ et $\phi = T$ avec $(x : T) \in \Gamma$.
  Comme $[\![\Lambda]\!]_1 = x$, nous devons montrer que $E_J[[\![\Gamma]\!]_1] \vdash x : [\![T]\!]_1$.
  Pour cela, nous appliquons la règle (Var).
  En effet, on a $\mathcal{WF}(E_J)[[\![\Gamma]\!]_1]$ par hypothèse,
  et on a bien $(x : [\![T]\!]_1) \in [\![\Gamma]\!]_1$ puisque $(x : T) \in \Gamma$.

- (Const)
  Il faut distinguer 2 cas :
  - Supposons que Λ soit l'une des constantes $c_i \in \{c \in Dep(\Lambda) \mid I \in Dep(c)\}$ et $\phi = t_i$.
    Dans ce cas, $[\![\Lambda]\!]_1 = c'_i$. Or les constantes c'_i ajoutées dans l'environnement de déclarations ont précisément le type $[\![t_i]\!]_1$.
    Il suffit donc d'appliquer la règle (Const) pour avoir
    $E_J[[\![\Gamma]\!]_1] \vdash c'_i : [\![t_i]\!]_1$
    vu que $(c'_i : [\![t_i]\!]_1) \in E_J$ et $\mathcal{WF}(E_J)[[\![\Gamma]\!]_1]$.
  - Sinon Λ est une constante c telle que $c \notin \{c \in Dep(\Lambda) \mid I \in Dep(c)\}$.
    Dans ce cas, $[\![\Lambda]\!]_1 = c$.
    Comme cette constante c de type ϕ n'est pas dépendante de I, $[\![\phi]\!]_1 = \phi$.
    Pour établir $E_J[[\![\Gamma]\!]_1] \vdash c : \phi$, on utilise la règle (Const).
    On a bien l'hypothèse $(c : T) \in E_J$ car $(c : T) \in E_I$ et $E_I \subset E_J$,
    et on a bien l'hypothèse $\mathcal{WF}(E_J)[[\![\Gamma]\!]_1]$.

- (Prod)
  Une des règles (Prod1), (Prod2) ou (Prod3) a permis d'établir
  $E_I[\Gamma] \vdash \forall x : T.U : s'$

à partir de $E_I[\Gamma] \vdash T : s$ et $E_I[\Gamma; x : T] \vdash U : s'$.
Comme $[\![\forall x : T.U]\!]_1 = \forall x : [\![T]\!]_1.[\![U]\!]_1$ et $[\![s']\!]_1 = s'$,
il s'agit ici de montrer que
$E_J[[\![\Gamma]\!]_1] \vdash (\forall x : [\![T]\!]_1.[\![U]\!]_1)\{?_i \mapsto M_i\}_i : s'$.
Par hypothèse d'induction, on a
$E_J[[\![\Gamma]\!]_1] \vdash [\![T]\!]_1\{?_i \mapsto M_i\}_i : s$ et $E_J[[\![\Gamma; (x : T)]\!]_1] \vdash [\![U]\!]_1\{?_i \mapsto M_i\}_i : s'$.
Cette deuxième hypothèse revient à $E_J[[\![\Gamma]\!]_1; (x : [\![T]\!]_1)] \vdash [\![U]\!]_1\{?_i \mapsto M_i\}_i : s'$.
Il suffit donc d'appliquer la règle (Prod1), (Prod2) ou (Prod3) (en fonction de la sorte s')
pour avoir $E_J[[\![\Gamma]\!]_1] \vdash \forall x : [\![T]\!]_1\{?_i \mapsto M_i\}_i.[\![U]\!]_1 : s'$.

- (Lam)
  Nous sommes dans le cas où $\Lambda = \lambda x : T.t$ et $\phi = \forall x : T.U$.
  Il s'agit de montrer que
  $E_J[[\![\Gamma]\!]_1] \vdash [\![\lambda x : T.t]\!]_1\{?_i \mapsto M_i\}_i : [\![\forall x : T.U]\!]_1$,
  c'est-à-dire $E_J[[\![\Gamma]\!]_1] \vdash \lambda x : [\![T]\!]_1.[\![t]\!]_1\{?_i \mapsto M_i\}_i : \forall x : [\![T]\!]_1.[\![U]\!]_1$.
  Or par hypothèse d'induction on a $E_J[[\![\Gamma]\!]_1] \vdash (\forall x : [\![T]\!]_1.[\![U]\!]_1)\{?_i \mapsto M_i\}_i : s$
  et $E_J[[\![\Gamma; (x : T)]\!]_1] \vdash [\![t]\!]_1\{?_i \mapsto M_i\}_i : [\![U]\!]_1$.
  Cette deuxième hypothèse revient à
  $E_J[[\![\Gamma]\!]_1; (x : [\![T]\!]_1)] \vdash [\![t]\!]_1\{?_i \mapsto M_i\}_i : [\![U]\!]_1$.
  Il suffit donc d'appliquer la règle (Lam) pour avoir
  $E_J[[\![\Gamma]\!]_1] \vdash \lambda x : [\![T]\!]_1.[\![t]\!]_1\{?_i \mapsto M_i\}_i : \forall x : [\![T]\!]_1.[\![U]\!]_1$.

- (App)
  Nous sommes dans le cas où $\Lambda = M\ \overrightarrow{u}$ et $\phi = T\{\overrightarrow{x}/\overrightarrow{u}\}$.
  Il nous faut distinguer 2 cas.
  – Supposons que $M = I\_ind$ (ou $M = I\_rec$ ou $M = I\_rect$).
    Dans ce cas $\overrightarrow{u}$ est de la forme $\overrightarrow{q}\ Q\ N_1...N_n\ t_1...t_g\ e'$
    (car nous avons supposé que les applications des principes d'induction sont complètes),
    et T est le type T_{I_ind} du principe d'induction I_ind :
    $\forall \overrightarrow{p : T_p}$
    $\forall P : (\forall a_1 : A_1...\forall a_g : A_g.(I\ \overrightarrow{p}\ a_1...a_g) \to Prop)$.
    $Principe(I, 1) \to$
    ...
    $Principe(I, n) \to$
    $\forall a_1 : A_1...\forall a_g : A_g.$
    $\forall e : (I\ \overrightarrow{p}\ a_1...a_g).$
    $(P\ a_1...a_g\ e)$

    Comme la règle appliquée pour typer $I\_ind\ \overrightarrow{q}\ Q\ N_1...N_n\ t_1...t_g\ e'$ est (App), on en déduit que on a nécessairement :
    (1) $E_I[\Gamma] \vdash \overrightarrow{q : T_p}$

(2) $E_I[\Gamma] \vdash Q : \forall a_1 : A_1 ... \forall a_g : A_g.(I\ \overrightarrow{q}\ a_1 ... a_g) \rightarrow Prop$
(3) $E_I[\Gamma] \vdash N_j : Principe(I, j)$ pour $j = 1..n$
(4) $E_I[\Gamma] \vdash t_j : A_j$ pour $j = 1..g$
(5) $E_I[\Gamma] \vdash e' : (I\ \overrightarrow{q}\ t_1 ... t_g)$

Il nous faut montrer ici que
$E_J[[\![\Gamma]\!]_1] \vdash [\![I\_ind\ \overrightarrow{q}\ Q\ N_1 ... N_n\ t_1 ... t_g\ e']\!]_1\{?_i \mapsto M_i\}_i : [\![P\ a_1 ... a_g\ e]\!]_1\sigma$
avec $\sigma = \{\overrightarrow{p}/\overrightarrow{q},\ P/Q,\ a_j/t_j,\ e/e'\}$.
Ce qui revient à montrer que
$E_J[[\![\Gamma]\!]_1] \vdash (J\_ind\ [\![\overrightarrow{q}]\!]_1\ [\![Q]\!]_1\ [\![\overrightarrow{N}]\!]_1\ ?_1 ... ?_m\ [\![t_1]\!]_1 ... [\![t_g]\!]_1\ )\{?_i \mapsto M_i\}_i :$
$[\![Q]\!]_1\ [\![t_1]\!]_1 ... [\![t_g]\!]_1\ [\![e']\!]_1$

Après instanciation des métavariables $?_1 ... ?_m$ on obtient :
$E_J[[\![\Gamma]\!]_1] \vdash J\_ind\quad [\![\overrightarrow{q}]\!]_1\{?_i \mapsto M_i\}_i$
$[\![Q]\!]_1\{?_i \mapsto M_i\}_i$
$[\![\overrightarrow{N}]\!]_1\{?_i \mapsto M_i\}_i$
$M_1 ... M_m$
$[\![t_1]\!]_1\{?_i \mapsto M_i\}_i\ ... [\![t_g]\!]_1\{?_i \mapsto M_i\}_i$
$[\![e']\!]_1\{?_i \mapsto M_i\}_i\ :\ [\![Q]\!]_1\ [\![t_1]\!]_1 ... [\![t_g]\!]_1\ [\![e']\!]_1$
où M_j est de type $Principe(J, n + j)$ pour $j = 1..m$.

Pour établir cela, on applique la règle (App) :
comme le principe d'induction J_ind a le type
$\forall \overrightarrow{p : T_p}$
$\forall P : (\forall a_1 : A_1 ... \forall a_g : A_g.(J\ \overrightarrow{p}\ a_1 ... a_g) \rightarrow Prop).$
$Principe(J, 1) \rightarrow$
...
$Principe(J, n) \rightarrow$
$Principe(J, n + 1) \rightarrow$
...
$Principe(J, n + m) \rightarrow$
$\forall a_1 : A_1 ... \forall a_g : A_g.$
$\forall e : (J\ \overrightarrow{p}\ a_1 ... a_g).$
$(P\ a_1 ... a_g\ e)$
il faut montrer que
(i) $E_J[[\![\Gamma]\!]_1] \vdash [\![\overrightarrow{q}]\!]_1\{?_i \mapsto M_i\}_i : \overrightarrow{T_p}$
(ii) $E_J[[\![\Gamma]\!]_1] \vdash [\![Q]\!]_1\{?_i \mapsto M_i\}_i : ((a_1 : A_1) ... (a_g : A_g)(J\ \overrightarrow{q}\ a_1 ... a_g) \rightarrow Prop)$
(iii) $E_J[[\![\Gamma]\!]_1] \vdash [\![N_j]\!]_1\{?_i \mapsto M_i\}_i : Principe(J, j)$ pour $j = 1..n$
(iv) $E_J[[\![\Gamma]\!]_1] \vdash M_j : Principe(J, n + j)$ pour $j = 1..m.$
(v) $E_J[[\![\Gamma]\!]_1] \vdash [\![t_j]\!]_1\{?_i \mapsto M_i\}_i : A_j$ pour $j = 1..g.$
(vi) $E_J[[\![\Gamma]\!]_1] \vdash [\![e']\!]_1\{?_i \mapsto M_i\}_i : (J\ \overrightarrow{q}\ [\![t_1]\!]_1 ... [\![t_g]\!]_1)$
Notons que les métavariables ne peuvent pas apparaître dans $[\![\overrightarrow{q}]\!]_1$, $[\![Q]\!]_1$,

ni dans $[\![t_j]\!]_1$ pour $j = 1..g$.

On établit (i), (ii), (v) et (vi) par induction respectivement à partir de (1), (2), (4) et (5).
En effet, $[\![T_p]\!]_1 = T_p$ car $[\![T_p]\!]_1$ est le type des paramètres de I, donc $[\![T_p]\!]_1$ ne peut pas contenir d'occurences de I.
De même, $[\![A_j]\!]_1 = A_j$ pour $j = 1..g$ car les A_j sont les types des arguments de I, donc ne peuvent pas contenir d'occurences de I.
Comme les arguments N_j sont de type $Principe(I, j)$ par (3),
et que $[\![Principe(I, j)]\!]_1 = Principe(J, j)$, on a (iii) par induction.
Enfin, le (iv) est une hypothèse car M_j instancie la métavariable $?_j$ de type $Principe(J, n + j)$ pour $j = 1..m$.

– Sinon $M \neq I\_ind$ et $M \neq I\_rec$ et $M \neq I\_rect$.
Nous devons montrer que
$E_J[[\![\Gamma]\!]_1] \vdash [\![M\ \overrightarrow{u}]\!]_1\{?_i \mapsto M_i\}_i : [\![T\{\overrightarrow{x}/\overrightarrow{u}\}]\!]_1$.
Or nous savons par hypothèse d'induction que
$E_J[[\![\Gamma]\!]_1] \vdash [\![M]\!]_1\{?_i \mapsto M_i\}_i : \forall \overrightarrow{x : [\![U]\!]_1}.[\![T]\!]_1$ et que
$E_J[[\![\Gamma]\!]_1] \vdash [\![\overrightarrow{u}]\!]_1\{?_i \mapsto M_i\}_i : [\![\overrightarrow{U}]\!]_1$.
Il suffit donc d'appliquer la règle (App) sur ces deux hypothèses.

- (Ind-Const)
Distinguons 3 cas.
– Si $\Lambda = Constr(i, I)$ et $\phi = \forall \overrightarrow{p : T_p}.T_i$, alors nous devons prouver
$E_J[[\![\Gamma]\!]_1] \vdash [\![Constr(i, I)]\!]_1 : [\![\forall \overrightarrow{p : T_p}.T_i]\!]_1$
Autrement dit
$E_J[[\![\Gamma]\!]_1] \vdash Constr(i, J) : \forall \overrightarrow{p : [\![T_p]\!]_1}.[\![T_i]\!]_1$
De même que précédemment, on a $[\![T_p]\!]_1 = T_p$.
Partant de l'hypothèse $\mathcal{WF}(E_J)[[\![\Gamma]\!]_1]$ et du fait que
$Ind(J : t)[\overrightarrow{p : T_p}]\{[\![T_1]\!]_1 ... [\![T_n]\!]_1\ \overrightarrow{M}\} \in E_J$,
on applique la règle (Ind-Const) afin d'obtenir la propriété recherchée :
$E_J[[\![\Gamma]\!]_1] \vdash Constr(i, J) : \forall \overrightarrow{p : T_p}.[\![T_i]\!]_1$

– Dans le cas où la règle (Ind-Const) a été appliquée sur un constructeur d'un type inductif K_j dépendant de I, défini par $Ind(K_j : u_j)[\overrightarrow{r_j : R_j}]\{\overrightarrow{U_j}\}$,
on a $\Lambda = Constr(i, K_j)$ et nous devons montrer que
$E_J[[\![\Gamma]\!]_1] \vdash [\![Constr(i, K_j)]\!]_1 : [\![\forall \overrightarrow{r_j : R_j}.U_j]\!]_1$,
c'est-à-dire $E_J[[\![\Gamma]\!]_1] \vdash Constr(i, K'_j) : \forall \overrightarrow{r_j : [\![R_j]\!]_1}.[\![U_j]\!]_1$.
Cela résulte de la règle (Ind-Const) car $Ind(K'_j : [\![u_j]\!]_1)[\overrightarrow{r_j : [\![R_j]\!]_1}]\{\overrightarrow{[\![U_j]\!]_1}\} \in E_J$.

– Dans le dernier cas, $\Lambda = Constr(i, K)$ où K est un type inductif différent de I et ne dépendant pas de I. On a $Ind(K : A')[\overrightarrow{p' : T'_{p'}}]\{T'_1 ... T'_{n'}\} \in E_I$.
Comme $E_I \subset E_J$, on sait que ce type inductif K est dans E_J.

Comme K ne dépend pas de I, on a $[\![K]\!]_1 = K$, $[\![T'_{i'}]\!]_1 = T'_{i'}$ et $[\![\overrightarrow{T'_{p'}}]\!]_1 = \overrightarrow{T'_{p'}}$.
La propriété à établir : $E_J[[\![\Gamma]\!]_1] \vdash [\![Constr(i,K)]\!]_1 : [\![\overrightarrow{\forall p' : T'_{p'}}.T'_{i'}]\!]_1$,
se réécrit $E_J[[\![\Gamma]\!]_1] \vdash Constr(i,K) : \overrightarrow{\forall p' : T'_{p'}}.T'_{i'}$.
Elle découle de (Ind-Const).

- (Ind-Type)
  Il faut à nouveau distinguer 3 cas.
  - Si $\Lambda = I$ et $\phi = \overrightarrow{\forall p : T_p}.A$ Il s'agit de montrer que $E_J[[\![\Gamma]\!]_1] \vdash [\![I]\!]_1 : [\![\overrightarrow{\forall p : T_p}.A]\!]_1$
    Autrement dit $E_J[[\![\Gamma]\!]_1] \vdash J : \overrightarrow{\forall p : [\![T_p]\!]_1}.[\![A]\!]_1$
    Comme précédemment, on a $[\![T_p]\!]_1 = T_p$. De même $[\![A]\!]_1 = A$ car A est le type de I donc ne peut pas contenir d'occurences de I.
    Il faut donc simplement montrer $E_J[[\![\Gamma]\!]_1] \vdash J : \overrightarrow{\forall p : T_p}.A$
    ce qui s'obtient par la règle (Ind-Type) vu que
    $Ind(J : A)[\overrightarrow{p : T_p}]\{[\![T_1]\!]_1 ... [\![T_n]\!]_1\ \overrightarrow{M}\} \in E_J$ et que $\mathcal{WF}(E_J)[[\![\Gamma]\!]_1]$
  - Si la règle (Ind-Type) a été appliquée pour établir
    $E_I[\Gamma] \vdash K_j : \overrightarrow{\forall r_j : R_j}.u_j$ où $I \in Dep(K_J)$,
    alors $\Lambda = K_j$ et nous devons montrer que $E_J[[\![\Gamma]\!]_1] \vdash [\![K_j]\!]_1 : [\![\overrightarrow{\forall r_j : R_j}.u_j]\!]_1$,
    c'est-à-dire $E_J[[\![\Gamma]\!]_1] \vdash K'_j : \overrightarrow{\forall r_j : [\![R_j]\!]_1}.[\![u_j]\!]_1$
    On applique pour montrer cela la règle (Ind-Type).
    En effet on a $\mathcal{WF}(E_J)[[\![\Gamma]\!]_1]$ par hypothèse, et
    $Ind(K_j : [\![u_j]\!]_1)[\overrightarrow{r_j : [\![R_j]\!]_1}]\{[\![\overrightarrow{U_j}]\!]_1\} \in E_J$.
  - Dans le dernier cas, $\Lambda = K$ où K est un type inductif différent de I et ne dépendant pas de I.
    On a $Ind(K : A')[\overrightarrow{p' : T'_{p'}}]\{T'_1 ... T'_{n'}\} \in E_I$, donc K est dans E_J.
    Comme K ne dépend pas de I, on a $[\![K]\!]_1 = K$, $[\![A']\!]_1 = A'$ et $[\![\overrightarrow{T'_{p'}}]\!]_1 = \overrightarrow{T'_{p'}}$.
    La propriété à établir : $E_J[[\![\Gamma]\!]_1] \vdash [\![K]\!]_1 : [\![\overrightarrow{\forall p' : T'_{p'}}.A']\!]_1$,
    se réécrit $E_J[[\![\Gamma]\!]_1] \vdash Constr(i,K) : \overrightarrow{\forall p' : T'_{p'}}.A'$.
    Elle découle de (Ind-Type).

- (Case)
  Il nous faut distinguer 2 cas.
  - Supposons que la règle (Case) ait été utilisée pour typer une analyse par cas sur un terme e de type $(I\ \overrightarrow{q}\ t_1 ... t_g)$.

$$\frac{\begin{array}{c} Ind(I : A)[\overrightarrow{p : T_p}]\{T_1 .. T_n\} \in E_I \qquad E_I[\Gamma] \vdash e : (I\ \overrightarrow{q}\ t_1 ... t_g) \qquad \sigma = \{\overrightarrow{p}/\overrightarrow{q}\} \\ E_I[\Gamma] \vdash P : B \qquad [(I\ \overrightarrow{q}) : A\sigma \mid B] \qquad (E_I[\Gamma] \vdash f_i : \{Constr(i,I)\ \overrightarrow{q}\}^P)_{i=1..n} \end{array}}{E_I[\Gamma] \vdash Case(e,P,f_1 .. f_n) : (P\ t_1 ... t_g\ e)}$$

Par hypothèse on a (1) $Ind(J:A)[\overrightarrow{p:T_p}]\{[\![T_1]\!]_1..[\![T_n]\!]_1\ \overrightarrow{M}\} \in E_J$
Par hypothèse d'induction, on sait que :
(2) $E_J[[\![\Gamma]\!]_1] \vdash [\![e]\!]_1\{?_i \mapsto M_i\}_i : (J\ [\![\overrightarrow{q}]\!]_1\ [\![t_1]\!]_1..[\![t_g]\!]_1)$
(3) $E_J[[\![\Gamma]\!]_1] \vdash [\![P]\!]_1 : [\![B]\!]_1$
(4) $E_J[[\![\Gamma]\!]_1] \vdash [\![f_j]\!]_1\{?_i \mapsto M_i\}_i : \{Constr(j,J)\ [\![\overrightarrow{q}]\!]_1\}^{[\![P]\!]_1}$ pour $j = 1..n$
Par hypothèse, les termes M_i ont le type $\{Constr(i,J)\ [\![\overrightarrow{q}]\!]_1\}^{[\![P]\!]_1}$, pour $i = 1..m$, car les métavariables $?_i$ ont été déclarées par ce type dans la signature.
Sachant qu'on a nécessairement $[(I\ \overrightarrow{q}) : A\sigma \mid B]$,
et que $(I\ \overrightarrow{q})$ et $(J\ [\![\overrightarrow{q}]\!]_1)$ ont la même structure, alors on a
(5) $[(J\ [\![\overrightarrow{q}]\!]_1) : [\![A]\!]_1\sigma \mid [\![B]\!]_1]$.

Il s'agit ici de montrer que
$E_J[[\![\Gamma]\!]_1] \vdash [\![Case(e,P,f_1..f_n)]\!]_1\{?_i \mapsto M_i\}_i : [\![(P\ t_1...t_g\ e)]\!]_1$
Autrement dit
$E_J[[\![\Gamma]\!]_1] \vdash Case([\![e]\!]_1, [\![P]\!]_1, [\![f_1]\!]_1..[\![f_n]\!]_1\ ?_1...?_m)\{?_i \mapsto M_i\}_i : ([\![P]\!]_1\ [\![t_1]\!]_1...[\![t_g]\!]_1\ [\![e]\!]_1)$
Ce qui donne après instanciation des métavariables $?_1...?_m$:
$E_J[[\![\Gamma]\!]_1] \vdash Case([\![e]\!]_1\{?_i \mapsto M_i\}_i, [\![P]\!]_1, [\![f_1]\!]_1\{?_i \mapsto M_i\}_i...[\![f_n]\!]_1\{?_i \mapsto M_i\}_i\ M_1...M_m) : ([\![P]\!]_1\ [\![t_1]\!]_1...[\![t_g]\!]_1\ [\![e]\!]_1)$
Pour établir cela, on utilise la règle (Case), appliquée sur (1), (2), (3), (4) et (5).

– Si la règle porte sur une analyse par cas d'un terme e de type $(K\ \overrightarrow{q'}\ t'_1...t'_{g'})$ avec $K \neq I$

$$\frac{Ind(K:A')[\overrightarrow{p':T'_{p'}}]\{T'_1..T'_{n'}\} \in E_I \quad E_I[\Gamma] \vdash e : (K\ \overrightarrow{q'}\ t'_1...t'_{g'}) \quad \sigma = \{\overrightarrow{p'}/\overrightarrow{q'}\} \quad E_I[\Gamma] \vdash P' : B' \quad [(K\ \overrightarrow{q'}) : A'\sigma \mid B'] \quad (E_I[\Gamma] \vdash f'_i : \{Constr(i,K)\ \overrightarrow{q'}\}^{P'})_{i=1..n'}}{E_I[\Gamma] \vdash Case(e,P',f'_1..f'_{n'}) : (P'\ t'_1...t'_{g'}\ e)}$$

alors la propriété à montrer
$E_J[[\![\Gamma]\!]_1] \vdash [\![Case(e,P',f'_1..f'_{n'})]\!]_1\{?_i \mapsto M_i\}_i : [\![(P'\ t'_1...t'_{g'}\ e)]\!]_1$
devient
$E_J[[\![\Gamma]\!]_1] \vdash Case([\![e]\!]_1\{?_i \mapsto M_i\}_i, [\![P']\!]_1, [\![f'_1]\!]_1\{?_i \mapsto M_i\}_i..[\![f'_{n'}]\!]_1\{?_i \mapsto M_i\}_i) : ([\![P']\!]_1\ [\![t'_1]\!]_1...[\![t'_{g'}]\!]_1\ [\![e]\!]_1)$
Par hypothèse d'induction, nous avons toutes les prémisses nécessaires pour appliquer la règle (Case) et montrer cette propriété.

- (Fix)
Il nous faut maintenant montrer que
$E_J[[\![\Gamma]\!]_1] \vdash ([\![Fix(f/n:A)\{M\}]\!]_1)\{?_i \mapsto M_i\}_i : [\![A]\!]_1$.
C'est-à-dire $E_J[[\![\Gamma]\!]_1] \vdash Fix(f/n : [\![A]\!]_1)\{[\![M]\!]_1\{?_i \mapsto M_i\}_i\} : [\![A]\!]_1$.
Si on applique la règle (Fix) pour montrer cela, il nous faut établir
$E_J[[\![\Gamma]\!]_1] \vdash [\![A]\!]_1 : s$ et $E_J[[\![\Gamma]\!]_1; (f : [\![A]\!]_1)] \vdash [\![M]\!]_1\{?_i \mapsto M_i\}_i : [\![A]\!]_1$.
Comme $[\![\Gamma]\!]_1; (f : [\![A]\!]_1) = [\![\Gamma; (f:A)]\!]_1$ nous avons ces propriétés par hy-

pothèse d'induction.

- (Conv)
  Enfin, si la règle appliquée est $\dfrac{E_I[\Gamma]\vdash U:s \quad E_I[\Gamma]\vdash t:T \quad E_I\vdash T\leq_{\beta\delta\phi\iota} U}{E_I[\Gamma]\vdash t:U}$
  il nous reste à montrer que $E_J[[\![\Gamma]\!]_1] \vdash [\![t]\!]_1\{?_i \mapsto M_i\}_i : [\![U]\!]_1$.
  Il suffit d'appliquer la règle (Conv) et d'utiliser les hypothèses d'induction.

□

7.6 Discussion

Nous avons donc formalisé notre méthode de réutilisation de preuves et de définitions par transformation de λ-termes. Dans le cas de mise à jour d'une définition, l'instanciation des métavariables dans le terme modifié correspond à la donnée par l'utilisateur des morceaux de définitions manquants.

Dans le cas de la réutilisation d'une preuve, l'instanciation des métavariables dans le terme modifié est réalisée par l'utilisateur en appliquant des tactiques pour résoudre les obligations de preuve générées.

Notre méthode ne s'applique qu'à des propriétés conservatives, c'est-à-dire toujours prouvable pour le type étendu. Si la propriété réutilisée n'était pas conservative par l'extension, alors nous ne pourrions pas montrer la propriété pour le type étendu. Cela se traduirait par l'impossibilité de fournir des termes dont le type serait celui des métavariables à instancier. Autrement dit, la preuve des obligations de preuve générées serait impossible. Autre cas de figure, si un lemme est prouvé par vacuité sur un type vide, et si on étend le type vide avec un ou plusieurs constructeurs, alors le terme de preuve modifié générera des obligations de preuve impossibles à décharger.

Notre méthode a quelques contraintes. D'une part nous ne considérons pas les types et les fonctions mutuellement inductifs. D'autre part, pour que notre méthode de réutilisation soit utilisable sur une propriété, la propriété doit être conservée par l'extension considérée, mais le schéma de la preuve doit lui aussi est conservé. En particulier si le type vide est prouvé par vacuité, nous ne pourrons pas réutiliser la preuve après extension car le schéma de preuve ne sera pas identique. De même, si la preuve nécessite de passer d'une inversion sur une hypothèse à une induction, le schéma n'est plus le même et nous ne pouvons donc pas le réutiliser. De la même manière, si la preuve faite par induction sur un type a besoin d'être faite par induction sur un autre type, le schéma n'est pas conservé.

Enfin, nous avons la contrainte que les principes d'induction sont totalement appliqués. Cette condition n'est pas vraiment contraignante car dans la pratique les principes d'induction sont utilisés par l'intermédiaire de la tactique `Induction`,

et sont totalement appliqués.

Malgré cela notre méthode permet dans bien des cas de faire de la réutilisation dans un développement formel, de manière automatique et sûre. Une évolution incrémentale du développement peut ainsi être suivie pas-à-pas en ne traitant que le strict minimum à chaque étape.

Chapitre 8

Réutiliser une preuve après suppression d'un constructeur

Nous présentons ici notre approche de la réutilisation de termes de preuve dans le cas de la suppression d'un constructeur d'un type inductif.

Si ajouter un constructeur est utile et fréquent dans un esprit de construction de développement formel incrémental, supprimer un constructeur l'est tout autant. En effet, nous pouvons décider qu'un constructeur est inutile, redondant ou encore mal formulé. Il est ainsi possible de remplacer un constructeur en effectuant une suppression suivie d'un ajout.

8.1 Suppression d'un constructeur

Soit I un type inductif

$$Ind(I : A)[\Gamma_p]\{T_1 ... T_n\}$$

défini dans un environnement de déclarations E. Supprimer le $j^{ème}$ constructeur consiste à produire et ajouter dans E le type restreint suivant

$$Ind(J : A)[\Gamma_p]\{[\![T_1]\!]_2 ... [\![T_{j-1}]\!]_2 \ [\![T_{j+1}]\!]_2 ... [\![T_n]\!]_2\}$$

où le type T_j a disparu et où les T_i restants subissent l'opération $[\![.]\!]_2$ définie figure 8.1 afin de remplacer les occurences libres de I par J. Les principes d'induction sur J sont également générés automatiquement et ajoutés dans E.

8.2 Réutilisation de λ-termes

Le principe de réutilisation est le même qu'au chapitre 7 : l'idée consiste à modifier l'ancien λ-terme de la preuve d'un lemme (ou d'une définition), en supprimant cette fois les morceaux correspondant au constructeur effacé. Ici la preuve obtenue

n'est pas partielle mais complète, aucune obligation de preuve n'a besoin d'être générée, le procédé est complètement automatique.

Pour la modification du terme de preuve, le filtrage des analyses par cas est examiné, et les cas correspondant au constructeur effacé sont supprimés. Nous mettons ensuite à jour par renommage les noms des types, des définitions et des lemmes qui ont été mis à jour, ainsi que les constructeurs et les principes d'induction associés aux types inductifs modifiés. De plus, pour un principe d'induction sur un type modifié I appliqué à une propriété P de type $I \rightarrow Prop$, les arguments correspondant aux constructeurs effacés sont supprimés.

De même une définition par cas sur un objet de type I utilise un filtrage exhaustif sur les constructeurs de I. Donc si I est restreint, le filtrage doit lui aussi être restreint.

Par exemple, considérons le type inductif I avec deux constructeurs `c1` et `c2` dont les types respectifs sont `I` et `I` $\rightarrow$ `I`, et la fonction `f : I` $\rightarrow$ `nat` définie par :

```
λx:I. Cases x of  c1   => n1
                | c2 y => Cases y of c1   => n2
                                   | c2 z => n3
                           end
      end
```

Lorsque `I` est restreint en `J` en supprimant le constructeur `c1`, nous construisons `f_J` à partir de `f` en mettant à jour les noms dans le λ-terme de `f`, et en restreignant le filtrage de `x` et de `y`. Le λ-terme obtenu pour `f_J : J` $\rightarrow$ `nat` est le suivant :

```
λx:J. Cases x of
            c2' y => Cases y of
                           c2' z => n3
                     end
       end
```

La définition de `f_J` est complète et automatique.

La seule contrainte que nous devons imposer est que ni les définitions à mettre à jour ni les preuves à réutiliser ne font usage du $j^{ème}$ constructeur que l'on veut supprimer. Cette vérification peut être faite par une analyse des dépendances dans les termes, similaire à celle présentée au chapitre 5.

Par exemple, si la définition de `f` était

```
λx:I. Cases x of  c1   => n1
                | c2 y => c2 c1
      end
```

la mise à jour de cette définition donnerait le terme

```
λx:J. Cases x of
            c2' y => c2' c1
       end
```

qui n'est pas bien typé car dans `c2' c1`, `c2'` est de type `J→J` et `c1` est de type `I`.

De même, cela n'a aucun sens de vouloir réutiliser un lemme `L` de la forme `(P c1)` avec `P : I→Prop`, lorsque `I` est restreint en `J` en supprimant le constructeur `c1`.

Nous pourrions être un peu moins restrictif en autorisant les dépendances sur le constructeur numéro j, dans le cas où cette dépendance apparait dans la $j^{ème}$ branche d'une analyse par cas. En effet, cette branche est destinée à être supprimée, donc les éventuelles occurences du $j^{ème}$ constructeur disparaitront également. Notre analyse de dépendances ne permet pas de distinguer la provenance d'une dépendance. Pour remédier à cela, il suffit de marquer les dépendances qui proviennent d'une analyse par cas, par leur indice dans la liste de filtrages, afin de pouvoir faire le distingo.

Une autre solution, suggérée par Yves Bertot, consiste à ne pas restreindre *a priori* les cas où la mise à jour est autorisée. On vérifie alors que la transformation appliquée produit une expression bien typée. Dans le cas contraire, la fonction transformée contient une occurrence du constructeur supprimé. La fonction doit donc être refusée *a posteriori*. Cette méthode permet de ne pas faire un calcul de dépendances marquées par des indices comme nous le proposons.

8.3 Formalisation de la modification de λ-termes

Nous formalisons ici les modifications sur les λ-termes, pour réutiliser des preuves après suppression d'un constructeur d'un type inductif.

L'opération de transformation de λ-termes $[\![.]\!]_2$ implicitement paramétrée par I et J est définie formellement figure 8.1. Le principe est le même qu'au chapitre 7 :

- les occurrences du type inductif I, de ses constructeurs et de ses principes d'induction sont substituées par J, les constructeurs correspondants de J, et les principes d'induction de J.
- Les constantes c_i qui dépendent de I sont remplacées par les constantes c'_i telles que $c_i \underset{I,J}{\rightsquigarrow} c'_i$.
- Les constructeurs d'un type inductif sont désignés par leur numéro dans la liste des constructeurs. Comme notre nouveau type J ne possède plus le constructeur numéro j, les constructeurs de J se voient renumérotés : tout constructeur de numéro i avec $j < i$ devient le constructeur de numéro $i - 1$.
- Enfin, $[\![.]\!]_2$ supprime de la liste des filtrages d'une analyse par cas et de la liste des arguments d'un principe d'induction le cas du constructeur supprimé.

Contrairement à la situation du chapitre 7, aucune métavariable n'est ajoutée, donc aucune obligation de preuve n'est générée et la preuve est complètement automatique.

$$
\begin{array}{lll}
[\![s]\!]_2 & = s & \\
[\![x]\!]_2 & = x & \\
[\![I]\!]_2 & = J & \\
[\![c]\!]_2 & = c' & \text{si } I \in Dep(c) \text{ et } c \underset{I,J}{\rightsquigarrow} c' \\
 & = c & \text{sinon} \\
[\![\Pi x : M.N]\!]_2 & = \Pi x : [\![M]\!]_2.[\![N]\!]_2 & \\
[\![\lambda\ x : M.N]\!]_2 & = \lambda\ x : [\![M]\!]_2.[\![N]\!]_2 & \\
[\![I\_ind\ \overrightarrow{q}\ Q\ N_1..N_n\ \overrightarrow{t}\ e]\!]_2 & = J\_ind\ [\![\overrightarrow{q}]\!]_2\ [\![Q]\!]_2\ [\![N_1]\!]_2..[\![N_{j-1}]\!]_2\ [\![N_{j+1}]\!]_2..[\![N_n]\!]_2\ [\![\overrightarrow{t}]\!]_2\ [\![e]\!]_2 & \\
[\![I\_rec\ \overrightarrow{q}\ Q\ N_1..N_n\ \overrightarrow{t}\ e]\!]_2 & = J\_rec\ [\![\overrightarrow{q}]\!]_2\ [\![Q]\!]_2\ [\![N_1]\!]_2..[\![N_{j-1}]\!]_2\ [\![N_{j+1}]\!]_2..[\![N_n]\!]_2\ [\![\overrightarrow{t}]\!]_2\ [\![e]\!]_2 & \\
[\![I\_rect\overrightarrow{q}\ Q\ N_1..N_n\ \overrightarrow{t}\ e]\!]_2 & = J\_rect[\![\overrightarrow{q}]\!]_2\ [\![Q]\!]_2\ [\![N_1]\!]_2..[\![N_{j-1}]\!]_2\ [\![N_{j+1}]\!]_2..[\![N_n]\!]_2\ [\![\overrightarrow{t}]\!]_2\ [\![e]\!]_2 & \\
[\![M\ \overrightarrow{N}]\!]_2 & = [\![M]\!]_2\ [\![\overrightarrow{N}]\!]_2 & \text{si } M \neq I\_ind, I\_rec, I\_rect \\
[\![Constr(i,I)]\!]_2 & = Constr(i,J) & \text{si } i < j \\
 & = Constr(i-1,J) & \text{si } i > j \\
[\![Constr(i,c)]\!]_2 & = Constr(i,[\![c]\!]_2) & \text{si } c \neq I \\
[\![Case(e,P,N_1..N_n)]\!]_2 & = Case([\![e]\!]_2,[\![P]\!]_2,[\![N_1]\!]_2..[\![N_{i-1}]\!]_2\ [\![N_{i+1}]\!]_2..[\![N_n]\!]_2) & \\
 & & \text{si } e : (I\ \overrightarrow{q}\ t_1...t_g) \\
 & = Case([\![e]\!]_2,[\![P]\!]_2,[\![N_1]\!]_2..[\![N_n]\!]_2) & \text{sinon} \\
[\![Fix(f/n : A)\{M\}]\!]_2 & = Fix(f/n : [\![A]\!]_2)\{[\![M]\!]_2\} &
\end{array}
$$

FIG. 8.1 – Définition de $[\![\Lambda]\!]_2$ quand I est restreint en J en supprimant le constructeur de rang j

La définition de $[\![.]\!]_2$ s'étend aux contextes de typage comme dans la définition 7.4.2.

Définition 8.3.1 *L'opération $[\![.]\!]_2$ s'étend aux environnements de déclarations lorsque I est restreint en J en suprrimant le le constructeur de rang j de la manière suivante :*

$$
\begin{array}{lcl}
[\![\emptyset]\!]_2 & = & \emptyset \\
[\![E; c : T]\!]_2 & = & [\![E]\!]_2;\ c : T;\ c' : [\![T]\!]_2 \quad si\ I \in Dep(c)\ et\ c \underset{I,J}{\rightsquigarrow} c' \\
 & = & [\![E]\!]_2;\ c : T \qquad sinon \\
[\![E; c := d : T]\!]_2 & = & [\![E]\!]_2;\ c := d : T; \\
 & & c' := [\![d]\!]_2 : [\![T]\!]_2 \qquad si\ I \in Dep(c)\ et\ c \underset{I,J}{\rightsquigarrow} c' \\
 & = & [\![E]\!]_2;\ c := d : T \qquad sinon \\
[\![E; Ind(I : A)[\overrightarrow{p : T_p}]\{T_1 ... T_n\}]\!]_2 & = & [\![E]\!]_2;\ Ind(I : A)[\overrightarrow{p : T_p}]\{T_1 ... T_n\}; \\
 & & Ind(J : A)[\overrightarrow{p : T_p}]\{[\![T_1]\!]_2 ... [\![T_{j-1}]\!]_2 \\
 & & \qquad\qquad [\![T_{j+1}]\!]_2 ... [\![T_n]\!]_2\}; \\
 & & J\_ind := \Lambda_{J\_ind} : T_{J\_ind}; \\
 & & J\_rec := \Lambda_{J\_rec} : T_{J\_rec}; \\
 & & J\_rect := \Lambda_{J\_rect} : T_{J\_rect} \\
[\![E; Ind(K : u)[\overrightarrow{r : R}]\{\overrightarrow{U}\}]\!]_2 & = & [\![E]\!]_2;\ Ind(K' : [\![u]\!]_2)[\overrightarrow{r : [\![R]\!]_2}]\{[\![\overrightarrow{U}]\!]_2\} \\
 & & \qquad si\ I \in Dep(K)\ et\ K \underset{I,J}{\rightsquigarrow} K' \\
 & = & [\![E]\!]_2; Ind(K : u)[\overrightarrow{r : R}]\{\overrightarrow{U}\} \qquad sinon
\end{array}
$$

Comme dans la définition 7.4.3, nous ne détaillons pas le calcul de $\Lambda_{J\_ind}, T_{J\_ind}, \Lambda_{J\_rec}, T_{J\_rec}, \Lambda_{J\_rect}$ et T_{J_rect}. Nous laissons le système générer ces types.

8.4 Correction de la preuve par réutilisation de λ-termes modifiés

Pour prouver la correction de notre méthode de réutilisation, nous partons des mêmes hypothèses qu'au chapitre 7. Le terme de preuve Λ de type ϕ est transformé en $[\![\Lambda]\!]_2$ après avoir restreint I en J et après avoir mis à jour les constantes qui dépendent de Λ. On montre alors que $[\![\Lambda]\!]_2$ a le type $[\![\phi]\!]_2$.

Proposition 8.4.1

On suppose que

$$E_I[\Gamma] \vdash \Lambda : \phi$$

avec $Ind(I : A)[\overrightarrow{p : Tp}]\{T_1 ... T_n\} \in E_I$. *On suppose que le type* I *est restreint en* J *en supprimant le constructeur numéro* j, *que* Λ *n'a pas de dépendance avec* $Constr(j, I)$, *et que* $\mathcal{WF}([[\![E_I]\!]_2)[[\![\Gamma]\!]_2]$.
Alors on a

$$[\![E_I]\!]_2[[\![\Gamma]\!]_2] \vdash [\![\Lambda]\!]_2 : [\![\phi]\!]_2$$

Dans la suite, nous noterons $[\![E_I]\!]_1$ par E_J. Nous avons donc

$$\begin{array}{rl} E_J = & E_I \ ; \ Ind(J:A)[\overrightarrow{p:Tp}]\{[\![T_1]\!]_2...[\![T_{j-1}]\!]_2\ [\![T_{j+1}]\!]_2...[\![T_n]\!]_2\} \ ; \\ & J\_ind := \Lambda_{J\_ind} : T_{J\_ind};\ J\_rec := \Lambda_{J\_rec} : T_{J\_rec};\ J\_rect := \Lambda_{J\_rect} : T_{J\_rect}; \\ & c'_1 := [\![d_1]\!]_2 : [\![t_1]\!]_2; ...; c'_w := [\![d_w]\!]_2 : [\![t_w]\!]_2 \ ; \\ & c'_{w+1} : [\![t_{w+1}]\!]_1 \ ; ...;\ c_v :: [\![t_v]\!]_1 \ ; \\ & Ind(K'_1 : [\![u_1]\!]_2)[\overrightarrow{r_1 : [\![R_1]\!]_2}]\{\overrightarrow{[\![U_1]\!]_2}\}; ...; Ind(K'_h : [\![u_h]\!]_2)[\overrightarrow{r_h : [\![R_h]\!]_2}]\{\overrightarrow{[\![U_h]\!]_2}\} \end{array}$$

où pour tout $i \in 1...v$, $c_i \underset{I,J}{\rightsquigarrow} c'_i$, et tout $i \in 1...h$, $K_i \underset{I,J}{\rightsquigarrow} K'_i$
lorsque $\{c \in Dep(\Lambda) \mid I \in Dep(c)\} = \{c_1, ..., c_w, c_{w+1}, ..., c_v, K_1, ..., K_h\}$
et que $c_1 := d_1 : t_1; ...; c_w := d_w : t_w;\ c_{w+1} : t_{w+1}; ...; c_v : t_v \in E_I$, et
$Ind(K_1 : u_1)[\overrightarrow{r_1 : R_1}]\{\overrightarrow{U_1}\}; ...; Ind(K_h : u_h)[\overrightarrow{r_h : R_h}]\{\overrightarrow{U_h}\} \in E_I$.

Preuve

Il faut établir que $[\![\Lambda]\!]_2$ est une preuve de $[\![\phi]\!]_2$ dans le contexte $[\![\Gamma]\!]_2$ et l'environnement E_J. La preuve de cette proposition se fait par induction sur les dérivations de typage de $E_I[\Gamma] \vdash \Lambda : \phi$. Examinons chaque cas :

- Les cas des règles (Ax1), (Ax2), (Ax3), (Var), (Const), (Prod), (Lam), (IndType), (Fix) et (Conv) se montrent de manière analogue à ceux du chapitre 7.
- (App)

  La règle appliquée est $\dfrac{E_I[\Gamma]\vdash M:\overrightarrow{\forall x:U.T} \quad E_I[\Gamma]\vdash \overrightarrow{u}:\overrightarrow{U}}{E_I[\Gamma]\vdash (M\ \overrightarrow{u}):T\{\overrightarrow{x}/\overrightarrow{u}\}}$

  Il faut distinguer 2 cas.
  - Supposons que $M = I\_ind$ (ou $M = I\_rec$ ou $M = I\_rect$).
    Dans ce cas $\overrightarrow{u}$ est de la forme $\overrightarrow{q}\ Q\ N_1...N_n\ t_1...t_g\ e'$.
    Le type du principe d'induction I_ind est de la forme
    $$\begin{array}{l} \forall \overrightarrow{p:Tp} \\ \forall P : (\forall a_1 : A_1...a_g : A_g.(I\ \overrightarrow{p}\ a_1...a_g) \to Prop). \\ Principe(I,1) \to \\ ... \\ Principe(I,n) \to \\ \forall a_1 : A_1...\forall a_g : A_g. \\ \forall e : (I\ \overrightarrow{p}\ a_1...a_g). \\ (P\ a_1...a_g\ e) \end{array}$$
    Nous en déduisons que
    (1) $E_I[\Gamma] \vdash \overrightarrow{q} : \overrightarrow{Tp}$
    (2) $E_I[\Gamma] \vdash Q : \forall a_1 : A_1...\forall a_g : A_g.(I\ \overrightarrow{q}\ a_1...a_g) \to Prop$
    (3) $E_I[\Gamma] \vdash N_i : Principe(I,i)$ pour $i = 1..n$
    (4) $E_I[\Gamma] \vdash t_i : A_i$ pour $i = 1..g$
    (5) $E_I[\Gamma] \vdash e' : (I\ \overrightarrow{q}\ t_1...t_g)$

    Il nous faut montrer que
    $E_J[[\![\Gamma]\!]_2] \vdash [\![I\_ind\ \overrightarrow{q}\ Q\ N_1...N_n\ t_1...t_g\ e']\!]_2 : [\![(P\ a_1...a_g\ e)\sigma]\!]_2$
    avec $\sigma = \{\overrightarrow{p}/\overrightarrow{q},\ P/Q,\ a_i/t_i,\ e/e'\}$.

Ce qui revient à montrer que
$E_J[[\Gamma]_2] \vdash J\_ind\ [\![\overrightarrow{q}]\!]_2\ [\![Q]\!]_2\ [\![N_1]\!]_2 ... [\![N_{j-1}]\!]_2\ [\![N_{j+1}]\!]_2 ... [\![N_n]\!]_2\ [\![t_1]\!]_2 ... [\![t_g]\!]_2\ [\![e']\!]_2 :$
$[\![Q\ t_1...t_g\ e']\!]_2$

Pour établir cela, on applique la règle (App) :
comme le principe d'induction J_ind a le type

$\forall \overrightarrow{p : Tp}$
$\forall P : (\forall a_1 : A_1 ... \forall a_g : A_g.(J\ \overrightarrow{p}\ a_1...a_g) \to Prop).$
$Principe(J, 1) \to$
...
$Principe(J, j-1) \to$
$Principe(J, j+1) \to$
...
$Principe(J, n) \to$
$\forall a_1 : A_1 ... \forall a_g : A_g.$
$\forall e : (J\ \overrightarrow{p}\ a_1...a_g).$
$(P\ a_1...a_g\ e)$

il suffit de montrer que

(i) $E_J[[\Gamma]_2] \vdash [\![\overrightarrow{q}]\!]_2 : \overrightarrow{Tp}$
(ii) $E_J[[\Gamma]_2] \vdash [\![Q]\!]_2 : \forall a_1 : A_1 ... \forall a_g : A_g.(J\ \overrightarrow{p}\ a_1...a_g) \to Prop$
(iii) $E_J[[\Gamma]_2] \vdash [\![N_i]\!]_2 : Principe(J, i)$ pour $1 \leqslant i \leqslant n$ et $i \neq j$
(iv) $E_J[[\Gamma]_2] \vdash [\![t_i]\!]_1 : A_i$ pour $1 \leqslant i \leqslant g$
(v) $E_J[[\Gamma]_2] \vdash [\![e']\!]_1 : (J\ \overrightarrow{p}\ [\![t_1]\!]_1 ... [\![t_g]\!]_1)$

On établit (i), (ii), (iii), (iv) et (v) par induction respectivement à partir de (1), (2), (3), (4) et (5).
En effet, $[\![T_p]\!]_1 = T_p$ car $[\![T_p]\!]_1$ est le type des paramètres de I, donc $[\![T_p]\!]_1$ ne peut pas contenir d'occurences de I.
De même, $[\![A_i]\!]_1 = A_i$ pour $i = 1..g$ car les A_i sont les types des arguments de I, donc ne peuvent pas contenir d'occurences de I.
Comme les arguments N_i sont de type $Principe(I, i)$ par (3),
et que $[\![Principe(I, i)]\!]_1 = Principe(J, i)$ pour $i \neq j$, on a bien (iii) par hypothèse d'induction.

– Sinon $M \neq I\_ind$ et $M \neq I\_rec$ et $M \neq I\_rect$.
Nous devons alors montrer que $E_J[[\Gamma]_2] \vdash [\![(M\ \overrightarrow{u})]\!]_2 : [\![T\{\overrightarrow{x}/\overrightarrow{u}\}]\!]_2$.
Or nous savons par hypothèse d'induction que
$E_J[[\Gamma]_2] \vdash [\![M]\!]_2 : \overrightarrow{\forall x : U.T}$ et que $E_J[[\Gamma]_2] \vdash [\![\overrightarrow{u}]\!]_2 : [\![\overrightarrow{U}]\!]_2$.
Il suffit donc d'appliquer la règle (App) sur ces deux hypothèses.

• (Ind-Const)
Distinguons 2 cas.

– Si la règle appliquée est $\dfrac{\mathcal{WF}(E_I)[\Gamma] \quad Ind(I{:}A)[\overrightarrow{p{:}T_p}]\{T_1...T_n\} \in E_I}{E_I[\Gamma] \vdash Constr(i,I){:}\overrightarrow{\forall p{:}T_p.T_i}}$,
Nous devons montrer que $E_J[[\Gamma]_2] \vdash [\![Constr(i, I)]\!]_2 : [\![\overrightarrow{\forall p : T_p.T_i}]\!]_2$.

Rappelons que $[\![T_p]\!]_2 = T_p$ car $[\![T_p]\!]_2$ est le type des paramètres de I, donc $[\![T_p]\!]_2$ ne peut pas contenir d'occurences de I.

- Si $i < j$, nous devons donc montrer que $E_J[[\![\Gamma]\!]_2] \vdash Constr(i,J) : \overrightarrow{\forall p : T_p}.[\![T_i]\!]_2$
  Il suffit pour montrer cela d'utiliser la règle (Ind-Const).
  En effet, nous avons $\mathcal{WF}(E_J)[[\![\Gamma]\!]_2]$ et
  $Ind(J:A)[\overrightarrow{p:T_p}]\{[\![T_1]\!]_2...[\![T_{j-1}]\!]_2\ [\![T_{j+1}]\!]_2...[\![T_n]\!]_2\} \in E_J$ par hypothèse.
- le cas $i = j$ est impossible car le terme Λ aurait eu une dépendance avec $Constr(j,I)$.
- Si $i > j$ nous devons donc montrer que
  $E_J[[\![\Gamma]\!]_2] \vdash Constr(i-1,J) : \overrightarrow{\forall p : T_p}.[\![T_i]\!]_2$
  A nouveau la règle (Ind-Const) s'applique car dans
  $Ind(J:A)[\overrightarrow{p:T_p}]\{[\![T_1]\!]_2...[\![T_{j-1}]\!]_2\ [\![T_{j+1}]\!]_2...[\![T_n]\!]_2\}$, le $(i-1)^{eme}$ constructeur a le type $[\![T_i]\!]_2$ (pour $i > j$).

- Supposons maintenant que la règle (Ind-Const) a été appliquée sur un constructeur $Constr(i,K)$ d'un autre type inductif appartenant à E_I.
  Dans ce cas la preuve est analogue à celle du chapitre 7, en distinguant le cas où K est une dépendance sur I ou non.

- (Case)
  Nous distinguons 2 cas :
  - Supposons que la règle (Case) a été utilisée pour typer une analyse par cas sur un terme e de type $(I\ \overrightarrow{q}\ t_1..t_g)$.

$$\frac{\begin{array}{c} Ind(I:A)[\overrightarrow{p:T_p}]\{T_1...T_n\} \in E_I \qquad E_I[\Gamma] \vdash e : (I\ \overrightarrow{q}\ t_1...t_g) \qquad \sigma = \{\overrightarrow{p}/\overrightarrow{q}\} \\ E_I[\Gamma] \vdash P : B \qquad [(I\ \overrightarrow{q}) : A\sigma \mid B] \qquad (E_I[\Gamma] \vdash f_i : \{Constr(i,I)\ \overrightarrow{q}\}^P)_{i=1..n} \end{array}}{E_I[\Gamma] \vdash Case(e,P,f_1...f_n) : (P\ t_1...t_g\ e)}$$

    Il s'agit ici de montrer que
    $E_J[[\![\Gamma]\!]_2] \vdash [\![Case(e,P,f_1..f_n)]\!]_2 : [\![(P\ t_1..t_g\ e)]\!]_2$. Autrement dit
    $E_J[[\![\Gamma]\!]_2] \vdash$
    $Case([\![e]\!]_2, [\![P]\!]_2, [\![f_1]\!]_2...[\![f_{j-1}]\!]_2\ [\![f_{j+1}]\!]_2...[\![f_n]\!]_2) : ([\![P]\!]_2\ [\![t_1]\!]_2...[\![t_g]\!]_2\ [\![e]\!]_2)$.

    Par hypothèse on a :
    (1) $Ind(J:A)[\overrightarrow{p:T_p}]\{[\![T_1]\!]_2...[\![T_{j-1}]\!]_2\ [\![T_{j+1}]\!]_2...[\![T_n]\!]_2\} \in E_J$
    Par hypothèse d'induction, on sait que :
    (2) $E_J[[\![\Gamma]\!]_2] \vdash [\![e]\!]_2 : (J\ [\![\overrightarrow{q}]\!]_2\ [\![t_1]\!]_2...[\![t_g]\!]_2)$, que
    (3) $E_J[[\![\Gamma]\!]_2] \vdash [\![P]\!]_2 : [\![B]\!]_2$, et que
    (4a) pour $1 \leqslant i < j$, $E_J[[\![\Gamma]\!]_2] \vdash [\![f_i]\!]_2 : \{Constr(i,J)\ [\![\overrightarrow{q}]\!]_2\}^{[\![P]\!]_2}$
    (4b) pour $j < i \leqslant n-1$, $E_J[[\![\Gamma]\!]_2] \vdash [\![f_{i+1}]\!]_2 : \{Constr(i,J)\ [\![\overrightarrow{q}]\!]_2\}^{[\![P]\!]_2}$
    Enfin, on a
    (5) $[(J\ [\![\overrightarrow{q}]\!]_2) : [\![A]\!]_2\sigma \mid [\![B]\!]_2]$, car on sait que $[(I\ \overrightarrow{q}) : A\sigma \mid B]$.

Pour conclure, il suffit d'appliquer la règle (Case) sur (1), (2), (3), (4) et (5).

– Si la règle (Case) appliquée porte sur un terme e de type $(K\ \overrightarrow{q'}\ t'_1..t'_{g'})$ avec $K \neq I$ alors la preuve est analogue à celle du chapitre 7.

□

8.5 Discussion

La transformation que nous venons de présenter permet de réutiliser le terme de preuve d'un lemme ou mettre à jour une définition, après suppression d'un constructeur dans un type inductif.

Contrairement au chapitre précédent, aucune obligation de preuve n'est générée, donc cette réutilisation est totalement automatique.

Retirer un constructeur d'un type inductif I produit un sous-type J, comme le montre E. Poll [74]. En effet, un objet e de type J est construit avec les constructeurs de J, donc e peut être construit de manière analogue avec les constructeurs de I. Ce type de sous-typage est appelé *Constructor Subtyping* par G. Barthe [10], mais sa formalisation nécessite d'étendre le système de types et d'ajouter des règles de conversions afin d'autoriser l'utilisation d'un sous-type J de I à la place de I. Une autre possibilité, plus pragmatique consiste à définir de manière systématique des fonctions de coercions de I vers J. Dans le système Coq, les coercions implicites [85] permettent de rendre l'utilisation de ces fonctions de coercion transparente pour l'utilisateur.

Notre avons choisi de garder notre approche réutilisant les termes de preuve afin d'avoir une méthode homogène quelque soit les transformations appliquées. Cela permet en outre de pouvoir composer les opérations de transformation entre elles.

Chapitre 9

Réutiliser une preuve après paramétrisation

Dans ce chapitre nous formalisons l'ajout de paramètres dans un type inductif. Nous formalisons l'opération de modification des termes, et nous montrons la correction de la réutilisation des termes de preuve modifiés.

9.1 Ajouter des paramètres à un type inductif

Soit I un type inductif

$$Ind(I : \forall \overrightarrow{a : A}.A_I)[\overrightarrow{p : T_p}]\{\overrightarrow{T}\}$$

défini dans un environnement de déclarations E. Les arguments $a_1 : A_1 \ldots a_g : A_g$ de I sont notés $\overrightarrow{a : A}$. Donc le type A_I n'est pas fonctionnel, il ne contient pas de produit. Le type I est transformé en J en ajoutant une liste de nouveaux paramètres $\overrightarrow{u : T_u}$, ce qui ajoute dans E la déclaration de

$$Ind(J : \forall \overrightarrow{a : A}.A_I)[\overrightarrow{p : T_p}; \overrightarrow{\boldsymbol{u : T_u}}]\{[\![\overrightarrow{T}]\!]_3\}$$

et la déclaration des principes d'induction sur J, où $[\![.]\!]_3$ est l'opération de modification de λ-termes, définie figure 9.1. L'opération $[\![.]\!]_3$ est appliquée sur $\overrightarrow{T}$ pour effectuer le renommage des occurrences de I par J.

Ces nouveaux paramètres doivent être tous distincts et ne pas capturer de variables dans la définition du type inductif où ils sont ajoutés. On remarquera qu'ils sont ajoutés à la suite de ceux déjà présents.

9.2 Formalisation de la modification de λ-termes

Nous allons maintenant formaliser les modifications de λ-termes après paramétrisation d'un type inductif I en J, par $\overrightarrow{u : T_u}$. L'opération de modification

$[\![.]\!]_3$ implicitement paramétrée par I, J et $\overrightarrow{u : T_u}$ est définie formellement dans la figure 9.1, comme au chapitre 7. Dans le terme $[\![\Lambda]\!]_3$, les occurrences des paramètres $\overrightarrow{u}$ sont libres.

- La différence principale est que les occurences de I appliqué à ses paramètres effectifs $\overrightarrow{q}$ sont remplacées par J appliqué aux paramètres transformés $[\![\overrightarrow{q}]\!]_3$ et aux nouveaux paramètres $\overrightarrow{u}$.
- De plus, si le type de I_ind est

$$\begin{array}{rl} I\_ind: & \forall p_1 : T_{p_1} ... \forall p_k : T_{p_k}. \\ & \forall P : (\forall a_1 : A_1 ... \forall a_g : A_g.(I\ p_1 ... p_k\ a_1 ... a_g) \rightarrow Prop). \\ & Principe(I, 1) \rightarrow \\ & ... \\ & Principe(I, n) \rightarrow \\ & \forall a_1 : A_1 ... \forall a_g : A_g. \\ & \forall e : (I\ p_1 ... p_k\ a_1 ... a_g). \\ & (P\ a_1 ... a_g\ e) \end{array}$$

  alors le type de J_ind sera

$$\begin{array}{rl} J\_ind: & \forall p_1 : T_{p_1} ... \forall p_k : T_{p_k} \forall \boldsymbol{u_1 : T_{u_1} ... \forall u_r : T_{u_r}}. \\ & \forall P : (\forall a_1 : A_1 ... \forall a_g : A_g.(J\ p_1 ... p_k\ \boldsymbol{u_1 ... u_r}\ a_1 ... a_g) \rightarrow Prop). \\ & Principe(J, 1) \rightarrow \\ & ... \\ & Principe(J, n) \rightarrow \\ & \forall a_1 : A_1 ... \forall a_g : A_g. \\ & \forall e : (J\ p_1 ... p_k\ \boldsymbol{u_1 ... u_r}\ a_1 ... a_g). \\ & (P\ a_1 ... a_g\ e) \end{array}$$

  C'est pourquoi l'opération $[\![.]\!]_3$ ajoute aussi les nouveaux paramètres $u_1 ... u_r$ à J_ind, J_rec, J_rest et aux constructeurs $Constr(i, J)$.

Nous illustrons ci-dessous le schéma de modification à l'aide d'un exemple jouet.

Considérons le type inductif $Ind(I : Set)[\ ]\{I\}$ et le terme Λ suivant :

$$\begin{array}{ll} \lambda x : I.\ (I\_ind & (\lambda x : I.\ x = Constr(1, I)) \\ & (refl\_equal\ I\ Constr(1, I)) \\ & x), \end{array}$$

de type $\forall x : I.\ x = Constr(1, I)$.

On ajoute le paramètre $n : nat$ à I pour former $Ind(J : Set)[\boldsymbol{n : nat}]\{J\}$.
La transformation $[\![\Lambda]\!]_3$ donne

$$\begin{array}{ll} \lambda x : (J\ \boldsymbol{n}).\ (J\_ind & \boldsymbol{n} \\ & (\lambda x : (J\ \boldsymbol{n}).\ x = (Constr(1, J)\ \boldsymbol{n})) \\ & (refl\_equal\ (J\ \boldsymbol{n})\ (Constr(1, J)\ \boldsymbol{n})) \\ & x) \end{array}$$

Ainsi, $\boldsymbol{\lambda n : nat.}[\![\Lambda]\!]_3$ est de type $\forall \boldsymbol{n : nat.} \forall x : (J\ \boldsymbol{n}).\ x = (Constr(1, J)\ \boldsymbol{n})$.

L'opération de transformation $[\![.]\!]_3$ lorsque I est paramétré par $\overrightarrow{u : T_u}$ en J s'étend aux contextes de typage comme dans les chapitres précédents, et aux environnement

$$
\begin{array}{lll}
[\![s]\!]_3 & = s & \\
[\![x]\!]_3 & = x & \\
[\![I]\!]_3 & = \lambda \overrightarrow{p : T_p}.(J\ \overrightarrow{p}\ \overrightarrow{\boldsymbol{u}}) & \\
[\![c]\!]_3 & = c'\ \boldsymbol{u} & \text{si } I \in Dep(c) \text{ et } c \underset{I,J}{\rightsquigarrow} c' \\
 & = c & \text{sinon} \\
[\![\Pi x : M.N]\!]_3 & = \Pi x : [\![M]\!]_3.[\![N]\!]_3 & \\
[\![\lambda\ x : M.N]\!]_3 & = \lambda\ x : [\![M]\!]_3.[\![N]\!]_3 & \\
[\![I\ \overrightarrow{q}\ \overrightarrow{t}]\!]_3 & = J\ [\![\overrightarrow{q}]\!]_3\ \overrightarrow{\boldsymbol{u}}\ [\![\overrightarrow{t}]\!]_3 & \\
[\![I\_ind\ \overrightarrow{q}\ P\ \overrightarrow{N}\ \overrightarrow{t}\ e]\!]_3 & = J\_ind\ [\![\overrightarrow{q}]\!]_3\ \overrightarrow{\boldsymbol{u}}\ [\![P]\!]_3\ [\![\overrightarrow{N}]\!]_3\ [\![\overrightarrow{t}]\!]_3\ [\![e]\!]_3 & \\
[\![I\_rec\ \overrightarrow{q}\ P\ \overrightarrow{N}\ \overrightarrow{t}\ e]\!]_3 & = J\_rec\ [\![\overrightarrow{q}]\!]_3\ \overrightarrow{\boldsymbol{u}}\ [\![P]\!]_3\ [\![\overrightarrow{N}]\!]_3\ [\![\overrightarrow{t}]\!]_3\ [\![e]\!]_3 & \\
[\![I\_rect\ \overrightarrow{q}\ P\ \overrightarrow{N}\ \overrightarrow{t}\ e]\!]_3 & = J\_rect\ [\![\overrightarrow{q}]\!]_3\ \overrightarrow{\boldsymbol{u}}\ [\![P]\!]_3\ [\![\overrightarrow{N}]\!]_3\ [\![\overrightarrow{t}]\!]_3\ [\![e]\!]_3 & \\
[\![Constr(i,I)\ \overrightarrow{q}\ \overrightarrow{t}]\!]_3 & = Constr(i,J)\ [\![\overrightarrow{q}]\!]_3\ \overrightarrow{\boldsymbol{u}}\ [\![\overrightarrow{t}]\!]_3 & \\
[\![M\ \overrightarrow{N}]\!]_3 & = [\![M]\!]_3\ [\![\overrightarrow{N}]\!]_3 \text{ si } M \neq I\_ind, I\_rec, I\_rect, Constr(i,I) & \\
[\![Constr(i,I)]\!]_3 & = \lambda \overrightarrow{p : T_p}.(Constr(i,J)\ \overrightarrow{p}\ \overrightarrow{\boldsymbol{u}}) & \\
[\![Constr(i,c)]\!]_3 & = Constr(i, [\![c]\!]_3) & \text{si } c \neq I \\
[\![Case(e,P,\overrightarrow{f})]\!]_3 & = Case([\![e]\!]_3, [\![P]\!]_3, [\![\overrightarrow{f}]\!]_3) & \\
[\![Fix(f/n : A)\{M\}]\!]_3 & = Fix(f/n : [\![A]\!]_3)\{[\![M]\!]_3\} &
\end{array}
$$

FIG. 9.1 – Définition de $[\![\Lambda]\!]_3$ quand I est paramétré par $\overrightarrow{\boldsymbol{u}}$ pour former J

de déclarations de la manière suivante :

$$
\begin{array}{lll}
[\![\emptyset]\!]_3 & = \emptyset & \\
[\![E; c : T]\!]_3 & = [\![E]\!]_3;\ c : T;\ c' : \forall \overrightarrow{u : T_u}.[\![T]\!]_3 & \text{si } I \in Dep(c) \\
 & & \text{et } c \underset{I,J}{\rightsquigarrow} c' \\
 & = [\![E]\!]_3;\ c : T & \text{sinon} \\
[\![E; c := d : T]\!]_3 & = [\![E]\!]_3;\ c := d : T; & \\
 & \quad c' := \lambda \overrightarrow{u : T_u}.[\![d]\!]_3 : \forall \overrightarrow{u : T_u}.[\![T]\!]_3 & \text{si } I \in Dep(c) \\
 & & \text{et } c \underset{I,J}{\rightsquigarrow} c' \\
 & = [\![E]\!]_3;\ c := d : T & \text{sinon} \\
[\![E; Ind(I : A)[\overrightarrow{p : T_p}]\{\overrightarrow{T}\}]\!]_3 = & [\![E]\!]_3;\ Ind(I : A)[\overrightarrow{p : T_p}]\{\overrightarrow{T}\}; & \\
 & Ind(J : A)[\overrightarrow{p : T_p}; \overrightarrow{u : T_u}]\{[\![\overrightarrow{T}]\!]_3\}; & \\
 & J\_ind := \Lambda_{J\_ind} : T_{J\_ind}; & \\
 & J\_rec := \Lambda_{J\_rec} : T_{J\_rec}; & \\
 & J\_rect := \Lambda_{J\_rect} : T_{J\_rect} & \\
[\![E; Ind(K : u)[\overrightarrow{r : R}]\{\overrightarrow{U}\}]\!]_3 = & [\![E]\!]_3;\ Ind(K' : [\![u]\!]_3)[\overrightarrow{r : [\![R]\!]_3}]\{[\![\overrightarrow{U}]\!]_3\} & \\
 & & \text{si } I \in Dep(K) \text{ et } K \underset{I,J}{\rightsquigarrow} K' \\
 & = [\![E]\!]_3; Ind(K : u)[\overrightarrow{r : R}]\{\overrightarrow{U}\} & \text{sinon}
\end{array}
$$

9.3 Correction de la preuve par réutilisation de λ-termes modifiés

La correction de notre méthode de réutilisation de terme se montre à partir des mêmes hypothèses qu'au chapitre 7. Le terme Λ de type ϕ est transformé en $[\![\Lambda]\!]_3$ après avoir ajouté des paramètres $\overrightarrow{u : T_u}$ à I et après avoir étendu les dépendances.

Nous supposons que les noms $\overrightarrow{u}$ n'apparaissent pas libres dans Λ. Dans le cas contraire, une étape d'α-conversion est nécessaire. On montre alors que $\lambda\overrightarrow{u : T_u}.[\![\Lambda]\!]_3$ a le type $\forall\overrightarrow{u : T_u}.[\![\phi]\!]_3$. La formulation du théorème diffère donc légèrement par rapport aux deux précédents, en ajoutant la quantification sur $\overrightarrow{u}$.

Proposition 9.3.1

On suppose que

$$E_I[\Gamma] \vdash \Lambda : \phi$$

avec $Ind(I : \forall\overrightarrow{a : A}.A_I)[\overrightarrow{p : T_p}]\{\overrightarrow{T}\} \in E_I$. *On suppose que le type* I *est transformé en* J *par ajout des paramètres* $\overrightarrow{u : T_u}$ *non libres dans* Λ, *et que* $\mathcal{WF}([\![E_I]\!]_3)([\![\Gamma]\!]_3]$. *Alors on a*

$$[\![E_I]\!]_3[[\![\Gamma]\!]_3] \vdash \lambda\overrightarrow{u : T_u}.[\![\Lambda]\!]_3 : \forall\overrightarrow{u : T_u}.[\![\phi]\!]_3$$

Dans la suite, nous noterons $[\![E_I]\!]_1$ par E_J. Nous avons donc

$$\begin{aligned} E_J = \quad & E_I \ ; \ Ind(J : \forall\overrightarrow{a : A}.A_I)[\overrightarrow{p : T_p}; \overrightarrow{u : T_u}]\{[\![\overrightarrow{T}]\!]_3\} \ ; \\ & J\_ind := \Lambda_{J\_ind} : \ T_{J\_ind}; \ J\_rec := \Lambda_{J\_rec} : T_{J\_rec}; \ J\_rect := \Lambda_{J\_rect} : T_{J\_rect}; \\ & c'_1 := \lambda\overrightarrow{u : T_u}.[\![d_1]\!]_3 : \forall\overrightarrow{u : T_u}.[\![t_1]\!]_3; ...; c'_w := \lambda\overrightarrow{u : T_u}.[\![d_w]\!]_3 : \forall\overrightarrow{u : T_u}.[\![t_w]\!]_3 \ ; \\ & c'_{w+1} : \forall\overrightarrow{u : T_u}.[\![t_{w+1}]\!]_1 \ ; ...; \ c_v : \forall\overrightarrow{u : T_u}.[\![t_v]\!]_1 \ ; \\ & Ind(K'_1 : [\![u_1]\!]_3)[\overrightarrow{r_1 : [\![R_1]\!]_3}]\{[\![\overrightarrow{U_1}]\!]_3\}; ...; Ind(K'_h : [\![u_h]\!]_3)[\overrightarrow{r_h : [\![R_h]\!]_3}]\{[\![\overrightarrow{U_h}]\!]_3\} \end{aligned}$$

où pour tout $i \in 1...v$, $c_i \underset{I,J}{\rightsquigarrow} c'_i$, et tout $i \in 1...h$, $K_i \underset{I,J}{\rightsquigarrow} K'_i$
lorsque $\{c \in Dep(\Lambda) \mid I \in Dep(c)\} = \{c_1, ..., c_w, c_{w+1}, ..., c_v, K_1, ..., K_h\}$
et que $c_1 := d_1 : t_1; ...; c_w := d_w : t_w; \ c_{w+1} : t_{w+1}; ...; c_v : t_v \in E_I$, et
$Ind(K_1 : u_1)[\overrightarrow{r_1 : R_1}]\{\overrightarrow{U_1}\}; ...; Ind(K_h : u_h)[\overrightarrow{r_h : R_h}]\{\overrightarrow{U_h}\} \in E_I$.

Preuve

Par application de la règle (Lam), cette proposition revient à démontrer

$$E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash [\![\Lambda]\!]_3 : [\![\phi]\!]_3$$

La preuve se fait par induction sur les dérivations de typage de $E_I[\Gamma] \vdash \Lambda : \phi$. Examinons chaque cas :

- Les cas des règles (Ax1), (Ax2), (Ax3), (Var), (Prod), (Fix) et (Conv) se montrent de manière analogue à ceux du chapitre 7.

- (Const)
  Lorsque Λ est un axiome ou une définition c, nous distinguons le cas où c dépend de I.
  - si $I \in Dep(c)\}$ et $c \underset{I,J}{\rightsquigarrow} c'$, dans ce cas il nous faut montrer
    $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash c'\ \overrightarrow{u} : [\![\phi]\!]_3$.
    Par application de la règle (App), cela revient à montrer que
    (i) $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash c' : \forall \overrightarrow{u : T_u}.[\![\phi]\!]_3$
    (ii) $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash \overrightarrow{u} : \overrightarrow{T_u}$
    Le (i) s'obtient par la règle (Const) sachant que l'on a bien
    $(c' : \forall \overrightarrow{u : T_u}.[\![\phi]\!]_3) \in E_J$.
    Le (ii) est une simple conséquence de la règle (Var).
  - si $I \notin Dep(c)\}$ dans ce cas il nous faut montrer $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash [\![c]\!]_3 : [\![\phi]\!]_3$.
    On obtient la propriété recherchée en appliquant la règle (Const), sachant que l'on a bien $(c : \phi) \in E_J$.

- (Lam)
  Dans le cas où $\Lambda = \lambda x : T.t$ et $\phi = \forall x : T.U$, nous pouvons déduire que :
  (1) $E_I[\Gamma] \vdash \forall x : T.U : s$
  (2) $E_I[\Gamma; x : T] \vdash t : U$
  Il s'agit pour nous de montrer que $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash [\![\lambda x : T.t]\!]_3 : [\![\forall x : T.U]\!]_3$,
  c'est-à-dire $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash \lambda x : [\![T]\!]_3.[\![t]\!]_3 : \forall x : [\![T]\!]_3.[\![U]\!]_3$.
  Par hypothèse d'induction appliquée sur (1) et (2) on a
  (i) $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash \forall x : [\![T]\!]_3.[\![U]\!]_3\ : s$
  (ii) $E_J[[\![\Gamma; x : T]\!]_3; \overrightarrow{u : T_u}] \vdash [\![t]\!]_3 : [\![U]\!]_3$.
  Comme les noms de paramètres $\overrightarrow{u}$ sont frais, le contexte de typage dans (ii) est équivalent à $[\![\Gamma]\!]_3; \overrightarrow{u : T_u}; x : [\![T]\!]_3$.
  Pour conclure, il suffit d'appliquer la règle (Lam) sur (i) et (ii) pour avoir
  $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash \lambda x : [\![T]\!]_3.[\![t]\!]_3 : \forall x : [\![T]\!]_3.[\![U]\!]_3$.

- (App)
  Dans le cas où $\Lambda = (M\ \overrightarrow{N})$ nous distinguons quatre cas.
  - Si $\Lambda = (I\ \overrightarrow{q}\ t_1 ... t_g)$, alors nous devons montrer que
    $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash (J\ [\![\overrightarrow{q}]\!]_3\ \overrightarrow{u}\ [\![t_1]\!]_3 ... [\![t_g]\!]_3)\ :\ [\![A_I]\!]_3$,
    Nous utilisons pour cela la règle (App), ce qui nous amène à établir :
    (1) $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash J : \forall \overrightarrow{p : T_p}.\forall \overrightarrow{u : T_u}.\forall \overrightarrow{a : A}.[\![A_I]\!]_3$
    (2) $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash [\![\overrightarrow{q}]\!]_3 : \overrightarrow{T_p}$
    (3) $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash \overrightarrow{u} : \overrightarrow{T_u}$
    (4) $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash [\![t_i]\!]_3 : A_i$ pour $i = 1..g$
    Le (1) résulte de la règle (Ind-type),
    (2) et (4) résultent des hypothèses d'induction et du fait que $[\![\overrightarrow{T_p}]\!]_3 = \overrightarrow{T_p}$ et $[\![A_i]\!]_3 = A_i$,

(3) résulte de la règle (Var).

– Si $M = I\_ind$ (ou I_rec ou I_rect) alors $\Lambda = (I\_ind\ \overrightarrow{q}\ Q\ N_1...N_n\ t_1..t_g\ e')$.
On en déduit que $\phi = (Q\ t_1..t_g\ e')$ et que l'on a nécessairement :
(1) $E_I[\Gamma] \vdash \overrightarrow{q : Tp}$
(2) $E_I[\Gamma] \vdash Q : (\forall a_1 : A_1...\forall a_g : A_g.(I\ \overrightarrow{q}\ a_1...a_g) \rightarrow Prop)$
(3) $E_I[\Gamma] \vdash N_i : Principe(I, i)$ pour $i = 1..n$
(4) $E_I[\Gamma] \vdash t_i : A_i$ pour $i = 1..g$
(5) $E_I[\Gamma] \vdash e' : (I\ \overrightarrow{q}\ t_1...t_g)$

Il nous faut montrer que :
$E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash$
$J\_ind\ [\![\overrightarrow{q}]\!]_3\ \overrightarrow{u}\ [\![Q]\!]_3\ [\![N_1]\!]_3...[\![N_n]\!]_3\ [\![t_1]\!]_3...[\![t_g]\!]_3\ [\![e']\!]_3 : [\![(Q\ t_1..t_g\ e')]\!]_3$
Nous appliquons pour cela la règle (App).
Cela nous ramène à établir :
(i) $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash [\![\overrightarrow{q}]\!]_3 : \overrightarrow{Tp}$
(ii) $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash \overrightarrow{u} : \overrightarrow{T_u}$
(iii) $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash [\![Q]\!]_3 : \forall a_1 : A_1...\forall a_g : A_g.(J\ [\![\overrightarrow{q}]\!]_3\ \overrightarrow{u}\ a_1...a_g) \rightarrow Prop$
(iv) $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash [\![N_i]\!]_3 : Principe(J, i)$ pour $i = 1..n$
(v) $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash [\![t_i]\!]_3 : A_i$ pour $i = 1..g$
(vi) $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash [\![e']\!]_3 : (J\ [\![\overrightarrow{q}]\!]_3\ \overrightarrow{u}\ [\![t_1]\!]_3...[\![t_g]\!]_3)$
Notons que $T_p = [\![T_p]\!]_3$, $A_i = [\![A_i]\!]_3$ et que $(J\ [\![\overrightarrow{q}]\!]_3\ \overrightarrow{u}\ [\![t_1]\!]_3...[\![t_g]\!]_3) = [\![(I\ \overrightarrow{q}\ t_1..t_g)]\!]_3$.
Remarquons également que $Principe(J, i) = [\![Principe(I, i)]\!]_3$.
En tenant compte de ces remarques, les points (i), (iii), (iv), (v) et (vi) résultent respectivement des hypothèses d'induction appliquées à (1), (2), (3), (4) et (5). Le (ii) résulte de la règle (Var).

– Supposons que $M = (Constr(i, I)\ \overrightarrow{q}\ \overrightarrow{t})$.
Si les arguments du i^{eme} constructeur sont $\overrightarrow{x : X}$, alors le type du i^{eme} constructeur s'écrit $T_i = \forall \overrightarrow{x : X}.T'_i$ tel que T'_i n'est pas fonctionnel.
Dans ce cas, $\phi = T'_i$ et $E_I[\Gamma] \vdash \overrightarrow{t} : \overrightarrow{X}$.
Nous devons montrer ici que
$E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash Constr(i, J)\ [\![\overrightarrow{q}]\!]_3\ \overrightarrow{u}\ [\![\overrightarrow{t}]\!]_3 : [\![T'_i]\!]_3$.
Comme nous avons $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash [\![\overrightarrow{q}]\!]_3 : \overrightarrow{T_p}$ par hypothèse d'induction,
et
$E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash \overrightarrow{u} : \overrightarrow{T_u}$ par la règle (Var),
et $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash [\![\overrightarrow{t}]\!]_3 : [\![\overrightarrow{X}]\!]_3$ par hypothèse d'induction,
nous pouvons appliquer la règle (App), afin de nous ramener à :
$E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash Constr(i, J) : \forall \overrightarrow{p : T_p}.\forall \overrightarrow{u : T_u}.\forall \overrightarrow{x : [\![X]\!]_3}.[\![T'_i]\!]_3$.
Ceci résulte de la règle (Ind-Const) car on a

$Ind(J : \forall\overrightarrow{a : A}.A_I)[\overrightarrow{p : T_p}; \overrightarrow{u : T_u}]\{[\![T_1]\!]_3 ... [\![T_n]\!]_3\} \in E_J$,
et $\forall\overrightarrow{x : [\![X]\!]_3}.[\![T'_i]\!]_3 = [\![T_i]\!]_3$.

– Si $M \neq I, I\_ind, I\_rec,\ I\_rect,\ Constr(i, I)$ alors dans ce cas, il suffit d'appliquer la règle (App) et d'utiliser les hypothèses d'induction.

- (Ind-Const)
  Nous distinguons encore deux cas.
  – Si $\Lambda = Constr(i, I)$ et $\phi = \forall p : T_p.T_i$ où T_i est le type du i^{eme} constructeur, alors nous devons montrer que
  $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash \lambda\overrightarrow{p : T_p}.(Constr(i, J)\ \overrightarrow{p}\ \overrightarrow{u}) : \forall\overrightarrow{p : T_p}.[\![T_i]\!]_3$.
  Après avoir appliquée la règle (Lam), nous obtenons
  $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}; \overrightarrow{p : T_p}] \vdash (Constr(i, J)\ \overrightarrow{p}\ \overrightarrow{u}) : [\![T_i]\!]_3$.
  Cela nous permet d'appliquer la règle (App) pour obtenir
  $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}; \overrightarrow{p : T_p}] \vdash Constr(i, J) : \forall\overrightarrow{p : T_p}.\forall\overrightarrow{u : T_u}.[\![T_i]\!]_6$.
  Ce qui résulte de la règle (Ind-Const).
  – Si $\Lambda = Constr(i, K)$ avec $K \neq I$, la preuve est analogue à celle du chapitre 7.
- (Ind-Type)
  Nous distinguons toujours deux cas.
  – Si $\Lambda = I$ alors $\phi = \forall\overrightarrow{p : T_p}.\forall\overrightarrow{a : A}.A_I$ et nous devons montrer que
  $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash \lambda\overrightarrow{p : T_p}.(J\ \overrightarrow{p}\ \overrightarrow{u}) : \forall\overrightarrow{p : T_p}.\forall\overrightarrow{a : A}.A_I$.
  Après avoir appliquée la règle (Lam), nous obtenons
  $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}; \overrightarrow{p : T_p}] \vdash (J\ \overrightarrow{p}\ \overrightarrow{u}) : \forall\overrightarrow{a : A}.A_I$,
  et par application de la règle (App), nous obtenons :
  $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}; \overrightarrow{p : T_p}] \vdash J : \forall\overrightarrow{p : T_p}.\forall\overrightarrow{u : T_u}.\forall\overrightarrow{a : A}.A_I$.
  Ce qui résulte de la règle (Ind-Type).
  – Si $\Lambda = K$ avec $K \neq I$, la preuve est analogue au chapitre 7.
- (Case)
  Pour montrer que
  $E_J[[\![\Gamma]\!]_3; \overrightarrow{u : T_u}] \vdash Case([\![e]\!]_3, [\![P]\!]_3, [\![\overrightarrow{f}]\!]_3) : [\![P\ \overrightarrow{t}\ e]\!]_3$,
  il suffit d'appliquer la règle (Case) et d'utiliser les hypothèses d'induction.

□

9.4 Discussion

Ici encore, la transformation produit un terme correct de manière automatique, sans générer d'obligations de preuve.

Les paramètres que nous ajoutons n'interviennent en aucune manière dans les constructeurs existants, ils ne semblent qu'être une décoration. D'ailleurs, nous pourrions envisagez l'opération inverse, à savoir le retrait de paramètres inutiles. Nous laisserons ce sujet en perspectives.

Le véritable intérêt de cette transformation ne se fait sentir que lorsqu'elle est suivie de l'ajout de nouveaux constructeurs. En effet, ces nouveaux constructeurs pourront faire usage des paramètres ajoutés. Nous renvoyons au chapitre 12 pour un exemple concret.

Chapitre 10

Réutiliser une preuve après ajout d'arguments

Ce chapitre concerne l'ajout d'arguments soit à un constructeur d'un type inductif soit au type lui-même. Nous formalisons l'opération de modification des termes pour chacune de ces deux transformations. Dans le deuxième cas, nous proposons deux opérations de transformation, la deuxième étant plus forte et à même de prendre en compte l'inversion. Nous montrons la correction de la réutilisation des preuves dans le cas de l'ajout d'arguments à un constructeur et dans la cas de l'ajout au type pour la deuxième transformation.

10.1 Cas de l'ajout d'arguments à un constructeur

Nous allons nous intéresser à l'ajout d'aguments à un constructeur donné. Cet ajout se fera en tête du type du constructeur, mais nous pourrions adapter le problème pour un ajout à une place quelconque.

10.1.1 Ajouter une liste d'arguments à un constructeur

L'ajout de la liste d'arguments $\overrightarrow{b : D}$ au constructeur numéro j du type inductif suivant

$$Ind(I : A)[\Gamma_p]\{T_1 ... T_n\}$$

consiste à ajouter dans l'environnement de déclarations le type

$$Ind(J : A)[\Gamma_p]\{[\![T_1]\!]_4 ... [\![T_{j-1}]\!]_4 \ (\forall \overrightarrow{\boldsymbol{b : D}}.[\![T_j]\!]_4) \ \ [\![T_{j+1}]\!]_4 ... [\![T_n]\!]_4\}$$

et les principes d'inductions J_ind, J_rec, J_rect associés.

Les occurrences de I dans les types T_i des constructeurs sont remplacées par J grâce à l'opération $[\![.]\!]_4$ définie figure 10.1, qui ajoute également les arguments $\overrightarrow{\boldsymbol{b}}$.

Les noms de ces nouveaux arguments doivent être distincts et frais afin de ne pas capturer de variables dans $[\![T_j]\!]_4$. De plus, les types $\overrightarrow{D}$ de ces arguments ne devront pas contenir d'occurrence de J. Une nouvelle occurrence de J dans le constructeur numéro j modifierait le principe d'induction, en ajoutant des hypothèses. En particulier $Principe(J, j)$ serait différent de $\forall \overrightarrow{b : D}.[\![Principe(I, j)]\!]_4$. En effet, dans un constructeur $Constr(j, J)$, chaque argument $b : D$ induit un produit $\forall b : D$ dans $Principe(J, j)$, mais lorsque l'argument est récursif, il induit en plus une hypothèse d'induction supplémentaire dans $Principe(J, j)$. Considérons le type inductif $Ind(J : Set)[\,]\{\boldsymbol{J} \rightarrow J\}$. Le type de J_{Ind} est dans ce cas :

$$\begin{array}{l}\forall P : (J \rightarrow Prop).\\ Principe(J, 1) \rightarrow\\ \forall e : J.(P\ e)\end{array}$$

avec $Principe(J, 1) = (\forall \boldsymbol{b : J.(P\ b)} \rightarrow (P\ (Constr(1, J)\ \boldsymbol{b})))$, au lieu d'un $(\forall \boldsymbol{b : J.}(P\ (Constr(1, J)\ \boldsymbol{b})))$ attendu.

10.1.2 Formalisation de la modification de λ-termes

Nous formalisons ici la mise à jour d'un λ-terme Λ après ajout d'arguments $\overrightarrow{\boldsymbol{b : D}}$ non libres dans Λ, au $j^{ème}$ constructeur de I.

L'opération d'extension $[\![.]\!]_4$ est définie figure 10.1, selon le même principe qu'au chapitre 7. L'opération $[\![.]\!]_4$ consiste à ajouter des abstractions par rapport aux nouveaux arguments dans la $j^{ème}$ branche d'une analyse par cas, et dans la preuve correspondant au $j^{ème}$ constructeur apparaissant dans un principe d'induction. De plus chaque occurrence du constructeur numéro j se voit appliquer les nouveaux arguments $\overrightarrow{\boldsymbol{b}}$.

Les termes Λ sur lesquels nous appliquons $[\![.]\!]_4$ doivent être restreints : le constructeur $Constr(j, I)$ ne doit pas avoir d'occurrence dans le type ϕ de Λ.

Dans le cas contraire, lorsque les arguments $\overrightarrow{\boldsymbol{b}}$ sont ajoutés à I, l'occurrence de $Constr(j, J)$ dans $[\![\phi]\!]_4$ se trouve appliquée à $\overrightarrow{\boldsymbol{b}}$ où $\overrightarrow{\boldsymbol{b}}$ sont libres, donc $[\![\phi]\!]_4$ serait mal typé.

Par exemple, considérons le type inductif $Ind(I : Set)[\,]\{I\}$, et un terme Λ de type $\phi = \forall x : I.\ x = Constr(1, I)$.

Nous ajoutons un paramètre $n : nat$ au premier constructeur de I, pour obtenir le type $Ind(J : Set)[\,]\{nat \rightarrow J\}$.

Ainsi, $[\![\phi]\!]_4 = \forall x : J.\ x = (Constr(1, J)\ n)$, où n est libre.

Cette restriction étant faite, examinons un exemple de situation où la transformation est licite. Soit le type inductif K suivant : $Ind(K : nat \rightarrow Prop)[\,]\{(K\ O),\ (K\ O)\}$, où les deux constructeurs sont de type $(K\ O)$.

Supposons que la propriété $\forall n : nat.(K\ n) \rightarrow (K\ O)$ soit établie à partir du script suivant :

$$
\begin{array}{lll}
[\![s]\!]_4 & = s & \\
[\![x]\!]_4 & = x & \\
[\![I]\!]_4 & = J & \\
[\![c]\!]_4 & = c' & \text{si } I \in Dep(c) \text{ et } c \underset{I,J}{\rightsquigarrow} c' \\
 & = c & \text{sinon} \\
[\![\Pi x : M.N]\!]_4 & = \Pi x : [\![M]\!]_4.[\![N]\!]_4 & \\
[\![\lambda\ x : M.N]\!]_4 & = \lambda\ x : [\![M]\!]_4.[\![N]\!]_4 & \\
[\![I\_ind\ \ \overrightarrow{q}\ Q\ N_1..N_n\ \overrightarrow{t}\ e]\!]_4 & = & \\
\multicolumn{3}{l}{J\_ind\ [\![\overrightarrow{q}]\!]_4\ [\![Q]\!]_4\ [\![N_1]\!]_4..[\![N_{j-1}]\!]_4\ (\overrightarrow{\boldsymbol{\lambda b : D}}.[\![N_j]\!]_4)\ [\![N_{j+1}]\!]_4..[\![N_n]\!]_4\ [\![\overrightarrow{t}]\!]_4\ [\![e]\!]_4} \\
[\![I\_rec\ \ \overrightarrow{q}\ Q\ N_1..N_n\ \overrightarrow{t}\ e]\!]_4 & = & \\
\multicolumn{3}{l}{J\_rec\ [\![\overrightarrow{q}]\!]_4\ [\![Q]\!]_4\ [\![N_1]\!]_4..[\![N_{j-1}]\!]_4\ (\overrightarrow{\boldsymbol{\lambda b : D}}.[\![N_j]\!]_4)\ [\![N_{j+1}]\!]_4..[\![N_n]\!]_4\ [\![\overrightarrow{t}]\!]_4\ [\![e]\!]_4} \\
[\![I\_rect\ \overrightarrow{q}\ Q\ N_1..N_n\ \overrightarrow{t}\ e]\!]_4 & = & \\
\multicolumn{3}{l}{J\_rect\ [\![\overrightarrow{q}]\!]_4\ [\![Q]\!]_4\ [\![N_1]\!]_4..[\![N_{j-1}]\!]_4\ (\overrightarrow{\boldsymbol{\lambda b : D}}.[\![N_j]\!]_4)\ [\![N_{j+1}]\!]_4..[\![N_n]\!]_4\ [\![\overrightarrow{t}]\!]_4\ [\![e]\!]_4} \\
[\![Constr(j,I)\ \overrightarrow{q}\ \ \overrightarrow{t})]\!]_4 & = Constr(j,J)\ [\![\overrightarrow{q}]\!]_4\ \ \overrightarrow{\boldsymbol{b}}\ \ [\![\overrightarrow{t}]\!]_4 & \\
[\![M\ \overrightarrow{N}]\!]_4 & = [\![M]\!]_4\ [\![\overrightarrow{N}]\!]_4 & \\
 & & \text{si } M \neq Constr(j,I), I\_ind, I\_rec, I\_rect \\
[\![Constr(i,I)]\!]_4 & = Constr(i,J) & \text{si } i \neq j \\
[\![Constr(j,I)]\!]_4 & = \lambda\overrightarrow{p : T_p}.(Constr(j,J)\ \overrightarrow{p}\ \ \overrightarrow{\boldsymbol{b}}) & \\
[\![Constr(i,c)]\!]_4 & = Constr(i,[\![c]\!]_4) & \text{si } c \neq I \\
[\![Case(e,P,f_1...f_n)]\!]_4 & = & \\
\multicolumn{3}{l}{Case([\![e]\!]_4,[\![P]\!]_4,[\![f_1]\!]_4..[\![f_{j-1}]\!]_4\ (\overrightarrow{\boldsymbol{\lambda b : D}}.[\![f_j]\!]_4)\ [\![f_{j+1}]\!]_4..[\![f_n]\!]_4)} & \text{si } e : (I\ \overrightarrow{q}\ t_1..t_g) \\
 & = Case([\![e]\!]_4,[\![P]\!]_4,[\![f_1]\!]_4..[\![f_n]\!]_4) & \text{sinon} \\
[\![Fix(f/n : A)\{M\}]\!]_4 & = Fix(f/n : [\![A]\!]_4)\{[\![M]\!]_4\} &
\end{array}
$$

FIG. 10.1 – Définition de $[\![\Lambda]\!]_4$ lors de l'ajout d'arguments $\overrightarrow{\boldsymbol{b}}$ à un constructeur de I

```
Intros n H.
Case H.
Apply C2.
Apply C2.
Qed.
```

Le terme de preuve Λ généré est alors :

$$
\begin{array}{l}
\lambda n : nat.\lambda H : (K\ n).Case(\ H, \\
\qquad\qquad\qquad\qquad\quad \lambda n : nat.(K\ O), \\
\qquad\qquad\qquad\qquad\quad Constr(2,K), \\
\qquad\qquad\qquad\qquad\quad Constr(2,K) \\
\qquad\qquad\qquad\qquad)
\end{array}
$$

Nous ajoutons maintenant un argument b : *bool* au $2^{ème}$ constructeur de K. Nous obtenons donc $Ind(K2 : nat \to Prop)[\ ]\{(K2\ O),\ \boldsymbol{bool} \to (K2\ O)\}$

Le terme $[\![\Lambda]\!]_4$ obtenu est :

$$\begin{array}{l}\lambda n : nat.\lambda H : (K2\ n).Case(\ H,\\ \qquad\qquad\qquad\qquad\qquad \lambda n : nat.(K2\ O),\\ \qquad\qquad\qquad\qquad\qquad (Constr(2, K2)\ \boldsymbol{b}),\\ \qquad\qquad\qquad\qquad\qquad \boldsymbol{\lambda b : bool.}(Constr(2, K2)\ \boldsymbol{b})\\ \qquad\qquad\qquad\qquad)\end{array}$$

Dans la deuxième branche de l'analyse, l'occurrence de la variable $\boldsymbol{b}$ est capturée par $\boldsymbol{\lambda b : bool}$, mais dans la première branche, nous avons une occurrence libre de $\boldsymbol{b}$. Cette occurrence doit être instanciée par l'utilisateur, par un habitant du type *bool*. Cela requiert que le type des nouveaux arguments soit non vide.

Pour capturer les éventuelles occurrences libres des nouveaux arguments $\overrightarrow{b}$ après l'application de la transformation $[\![.]\!]_4$, le terme $[\![\Lambda]\!]_4$ est abstrait par rapport à la liste $\overrightarrow{b}$, puis appliqué à autant de métavariables $\overrightarrow{?_b}$ pour générer des obligations de preuves.

Pour prouver les obligations de preuve, il suffira de fournir des termes $\overrightarrow{b_0}$ de type $\overrightarrow{D}$. Ces obligations de preuve correspondent à la preuve que les types des nouveaux arguments sont non vides.

Ainsi, le terme $(\lambda\overrightarrow{b : D}.[\![\Lambda]\!]_4)\ \overrightarrow{?_b}$ n'a plus d'occurrence libre de $\overrightarrow{b}$. Lorsque l'utilisateur instancie les métavariables par $\overrightarrow{b_0}$ en déchargeant les obligations de preuve, le terme $((\lambda\overrightarrow{b : D}.[\![\Lambda]\!]_4)\ \overrightarrow{?_b})\{\overrightarrow{?_b} \mapsto b_0\}$ a le type $[\![\phi]\!]_4$.

Si Λ est typé dans l'environnement $E_I[\Gamma]$, les jugements de typage $[\![E_I]\!]_4[[\![\Gamma]\!]_4] \vdash \overrightarrow{?_b} : \overrightarrow{D}$ sont déclarés dans la signature Σ.

Dans notre exemple, si la métavariable $?_b$ ajoutée dans $(\lambda b : bool.[\![\Lambda]\!]_4)\ ?_b$ est instanciée par *true* nous obtenons le terme

$$\begin{array}{l}(\lambda b : bool.\lambda n : nat.\lambda H : (K2\ n).Case(\ H,\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \lambda n : nat.(K2\ O),\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (Constr(2, K2)\ \boldsymbol{b}),\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \boldsymbol{\lambda b : bool.}(Constr(2, K2)\ \boldsymbol{b})\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad)\\ )\ true\end{array}$$

qui se réduit en

$$\begin{array}{l}\lambda n : nat.\lambda H : (K2\ n).Case\ (\ H,\\ \qquad\qquad\qquad\qquad\qquad \lambda n : nat.(K2\ O),\\ \qquad\qquad\qquad\qquad\qquad (Constr(2, K2)\ \boldsymbol{true}),\\ \qquad\qquad\qquad\qquad\qquad \boldsymbol{\lambda b : bool.}(Constr(2, K2)\ \boldsymbol{b})\\ \qquad\qquad\qquad\qquad\qquad)\end{array}$$

et qui est bien de type $\forall n : nat.(K2\ n) \to (K2\ O)$.

L'opération $[\![.]\!]_4$ lorsque I est transformé en J par ajout d'argument $\overrightarrow{b : D}$ au $j^{ème}$ constructeur s'étend aux contextes de typage comme dans les chapitres précédents, et aux environnements de déclarations de la manière suivante :

$$
\begin{array}{lll}
[\![\emptyset]\!]_4 & = \emptyset & \\
[\![E; c : T]\!]_4 & = [\![E]\!]_4;\ c : T;\ c' : [\![T]\!]_4 & \text{si } I \in Dep(c) \text{ et } c \underset{I,J}{\rightsquigarrow} c' \\
 & = [\![E]\!]_4;\ c : T & \text{sinon} \\
[\![E; c := d : T]\!]_4 & = [\![E]\!]_4;\ c := d : T; & \\
 & \quad c' := (\lambda \overrightarrow{b : D}.[\![d]\!]_4\ \overrightarrow{?_b})\{\overrightarrow{?_b} \mapsto \overrightarrow{b_0}\} : [\![T]\!]_4 & \\
 & \qquad\qquad \text{si } I \in Dep(c) \text{ et } c \underset{I,J}{\rightsquigarrow} c' & \\
 & = [\![E]\!]_4;\ c := d : T & \text{sinon} \\
[\![E; Ind(I : A)[\overrightarrow{p : T_p}]\{T_1 ... T_n\}]\!]_4 & = [\![E]\!]_4;\ Ind(I : A)[\overrightarrow{p : T_p}]\{T_1 ... T_n\}; & \\
 & \quad Ind(J : A)[\overrightarrow{p : T_p}]\{[\![T_1]\!]_4 ... [\![T_{j-1}]\!]_4 & \\
 & \qquad\qquad (\forall \overrightarrow{b : D}.[\![T_j]\!]_4)\ [\![T_{j+1}]\!]_4 ... [\![T_n]\!]_4\}; & \\
 & \quad J\_ind := \Lambda_{J\_ind} : T_{J\_ind}; & \\
 & \quad J\_rec := \Lambda_{J\_rec} : T_{J\_rec}; & \\
 & \quad J\_rect := \Lambda_{J\_rect} : T_{J\_rect} & \\
[\![E; Ind(K : u)[\overrightarrow{r : R}]\{\overrightarrow{U}\}]\!]_4 & = [\![E]\!]_4;\ Ind(K' : [\![u]\!]_4)[\overrightarrow{r : [\![R]\!]_4}]\{\overrightarrow{[\![U]\!]_4}\} & \\
 & \qquad\qquad \text{si } I \in Dep(K) \text{ et } K \underset{I,J}{\rightsquigarrow} K' & \\
 & = [\![E]\!]_4;\ Ind(K : u)[\overrightarrow{r : R}]\{\overrightarrow{U}\} & \text{sinon}
\end{array}
$$

10.1.3 Correction de la preuve par réutilisation de λ-termes modifiés

Pour prouver la correction de notre méthode de réutilisation, nous partons des mêmes hypothèses qu'au chapitre 7. Le terme de preuve Λ de type ϕ est transformé en $[\![\Lambda]\!]_4$ après avoir ajouté les nouveaux arguments $\overrightarrow{b : D}$ à I pour former J et après avoir étendu les dépendances.

On montre alors que $((\lambda \overrightarrow{b : D}.[\![\Lambda]\!]_4)\ \overrightarrow{?_b})\{\overrightarrow{?_b} \mapsto \overrightarrow{b_0}\}$ a le type $[\![\phi]\!]_4$, où $\overrightarrow{b_0}$ sont les termes de type $\overrightarrow{D}$.

Proposition 10.1.1

On suppose que

$$E_I[\Gamma] \vdash \Lambda : \phi$$

avec $Ind(I:A)[\overrightarrow{p:T_p}]\{T_1...T_n\} \in E_I$. *On suppose que le type* I *est transformé en* J *en ajoutant des arguments* $\overrightarrow{b:D}$ *non libres dans* Λ, *au constructeur numéro* j, *que* ϕ *n'a pas de dépendance avec* $Constr(j,I)$, *et que* $\mathcal{WF}([\![E_I]\!]_4)[[\![\Gamma]\!]_4]$.
Si $\Sigma = ([\![E_I]\!]_4[[\![\Gamma]\!]_4] \vdash \overrightarrow{?_b} : \overrightarrow{D})$ *et si* $[\![E_I]\!]_4[[\![\Gamma]\!]_4] \vdash \overrightarrow{b_0} : \overrightarrow{D}$ *alors on a*

$$[\![E_I]\!]_4[[\![\Gamma]\!]_4] \vdash (\lambda\overrightarrow{b:D}.[\![\Lambda]\!]_4\ \overrightarrow{?_b})\{\overrightarrow{?_b} \mapsto \overrightarrow{b_0}\} : [\![\phi]\!]_4$$

Dans la suite, nous noterons $[\![E_I]\!]_4$ par E_J. Nous avons donc

$$\begin{aligned} E_J = \quad & E_I\ ;\ Ind(J:A)[\overrightarrow{p:T_p}]\{[\![T_1]\!]_4...[\![T_{j-1}]\!]_4\ (\forall\overrightarrow{b:D}.[\![T_j]\!]_4)\ [\![T_{j+1}]\!]_4...[\![T_n]\!]_4\}\ ; \\ & J\_ind := \Lambda_{J\_ind} : T_{J\_ind};\ J\_rec := \Lambda_{J\_rec} : T_{J\_rec};\ J\_rect := \Lambda_{J\_rect} : T_{J\_rect}; \\ & c'_1 := (\lambda\overrightarrow{b:D}.[\![d_1]\!]_4\ \overrightarrow{?_b})\{\overrightarrow{?_b} \mapsto \overrightarrow{b_0}\} : [\![t_1]\!]_4; ...; \\ & c'_w := (\lambda\overrightarrow{b:D}.[\![d_w]\!]_4\ \overrightarrow{?_b})\{\overrightarrow{?_b} \mapsto \overrightarrow{b_0}\} : [\![t_w]\!]_4\ ; \\ & c'_{w+1} : [\![t_{w+1}]\!]_1\ ; ...;\ c_v :: [\![t_v]\!]_1\ ; \\ & Ind(K'_1 : [\![u_1]\!]_4)[\overrightarrow{r_1 : [\![R_1]\!]_4}]\{[\![\overrightarrow{U_1}]\!]_4\}; ...; Ind(K'_h : [\![u_h]\!]_4)[\overrightarrow{r_h : [\![R_h]\!]_4}]\{[\![\overrightarrow{U_h}]\!]_4\} \end{aligned}$$

où pour tout $i \in 1...v$, $c_i \underset{I,J}{\rightsquigarrow} c'_i$, et tout $i \in 1...h$, $K_i \underset{I,J}{\rightsquigarrow} K'_i$
lorsque $\{c \in Dep(\Lambda) \mid I \in Dep(c)\} = \{c_1, ..., c_w, c_{w+1}, ..., c_v, K_1, ..., K_h\}$
et que $c_1 := d_1 : t_1; ...; c_w := d_w : t_w\ c_{w+1} : t_{w+1}; ...; c_v : t_v \in E_I$, et
$Ind(K_1 : u_1)[\overrightarrow{r_1 : R_1}]\{\overrightarrow{U_1}\}; ...; Ind(K_h : u_h)[\overrightarrow{r_h : R_h}]\{\overrightarrow{U_h}\} \in E_I$.

Preuve
Après instanciation des métavariables la proposition revient à démontrer

$$E_J[[\![\Gamma]\!]_4] \vdash ((\lambda\overrightarrow{b:D}.[\![\Lambda]\!]_4)\ \overrightarrow{b_0}) : [\![\phi]\!]_4$$

Par application de la règle (App), vu que $E_J[[\![\Gamma]\!]_4] \vdash \overrightarrow{b_0} : \overrightarrow{D}$, nous nous ramenons à

$$E_J[[\![\Gamma]\!]_4] \vdash \lambda\overrightarrow{b:D}.[\![\Lambda]\!]_4 : \forall\overrightarrow{b:D}.[\![\phi]\!]_4$$

ce qui revient, par application de la règle (Lam), à

$$E_J[[\![\Gamma]\!]_4; \overrightarrow{b:D}] \vdash [\![\Lambda]\!]_4 : [\![\phi]\!]_4$$

La preuve se fait par induction sur les dérivations de typage de $E_I[\Gamma] \vdash \Lambda : \phi$. Examinons chaque cas :

- Les cas des règles (Ax1), (Ax2), (Ax3), (Var), (Const), (Prod), (Lam), (Ind-Type) et (Fix) se montrent de manière analogue à ceux du chapitre 7.
- (App)
  Nous sommes dans la situation où $\Lambda = M\ \overrightarrow{u}$.

– Si $M = I\_ind$ (ou I_rec, ou I_rect), alors $\overrightarrow{u}$ est de la forme $\overrightarrow{q}\ Q\ N_1...N_n\ t_1...t_g\ e'$.
Le type T_{I_ind} du principe d'induction I_ind est de la forme :

$$\begin{array}{l}\forall \overrightarrow{p : T_p} \\ \forall P : (\forall a_1 : A_1 ... \forall a_g : A_g.(I\ \overrightarrow{p}\ a_1...a_g) \rightarrow Prop) \\ Principe(I,1) \rightarrow \\ ... \\ Principe(I,n) \rightarrow \\ \forall a_1 : A_1 ... \forall a_g : A_g. \\ \forall e : (I\ \overrightarrow{p}\ a_1...a_g). \\ (P\ a_1...a_g\ e)\end{array}$$

On en déduit que $\phi = (Q\ t_1..t_g\ e')$ et que l'on a nécessairement :
(1) $E_I[\Gamma] \vdash \overrightarrow{q : T_p}$
(2) $E_I[\Gamma] \vdash Q : \forall a_1 : A_1 ... \forall a_g : A_g.(I\ \overrightarrow{q}\ a_1...a_g) \rightarrow Prop$
(3) $E_I[\Gamma] \vdash N_i : Principe(I,i)$ pour $i = 1..n$
(4) $E_I[\Gamma] \vdash t_i : A_i$ pour $i = 1..g$
(5) $E_I[\Gamma] \vdash e' : (I\ \overrightarrow{q}\ t_1...t_g)$

Il nous faut montrer que :
$E_J[[\![\Gamma]\!]_4; \overrightarrow{b : D}] \vdash$
$(J\_ind\ [\![\overrightarrow{q}]\!]_4\ [\![Q]\!]_4\ [\![N_1]\!]_4..[\![N_{j-1}]\!]_4\ \lambda \overrightarrow{b : D}.[\![N_j]\!]_4\ [\![N_{j+1}]\!]_4..[\![N_n]\!]_4\ [\![t_1]\!]_4...[\![t_g]\!]_4\ [\![e']\!]_4) :$
$[\![(Q\ t_1..t_g\ e')]\!]_4$
Pour cela nous appliquons la règle (App). Comme le type T_{J_ind} du principe d'induction J_ind est de la forme :

$$\begin{array}{l}\forall \overrightarrow{p : T_p} \\ \forall P : (\forall a_1 : A_1 ... \forall a_g : A_g.(J\ \overrightarrow{p}\ a_1...a_g) \rightarrow Prop) \\ Principe(J,1) \rightarrow \\ ... \\ Principe(J,n) \rightarrow \\ \forall a_1 : A_1 ... \forall a_g : A_g. \\ \forall e : (J\ \overrightarrow{p}\ a_1...a_g). \\ (P\ a_1...a_g\ e)\end{array}$$

il nous faut maintenant montrer que :
(i) $E_J[[\![\Gamma]\!]_4; \overrightarrow{b : D}] \vdash [\![\overrightarrow{q}]\!]_4 : \overrightarrow{T_p}$
(ii) $E_J[[\![\Gamma]\!]_4; \overrightarrow{b : D}] \vdash [\![Q]\!]_4 : \forall a_1 : A_1 ... \forall a_g : A_g.(J\ [\![\overrightarrow{q}]\!]_4\ a_1...a_g) \rightarrow Prop$
(iii) $E_J[[\![\Gamma]\!]_4; \overrightarrow{b : D}] \vdash [\![N_i]\!]_4 : Principe(J,i)$ pour $i = 1..n$ et $i \neq j$
(iv) $E_J[[\![\Gamma]\!]_4; \overrightarrow{b : D}] \vdash \lambda \overrightarrow{b : D}.[\![N_j]\!]_4 : Principe(J,j)$
(v) $E_J[[\![\Gamma]\!]_4; \overrightarrow{b : D}] \vdash [\![t_i]\!]_4 : A_i$ pour $i = 1..g$
(vi) $E_J[[\![\Gamma]\!]_4; \overrightarrow{b : D}] \vdash [\![e']\!]_4 : (J\ [\![\overrightarrow{q}]\!]_4\ [\![t_1]\!]_4...[\![t_g]\!]_4)$

Notons que $T_p = [\![T_p]\!]_4$, $A_i = [\![A_i]\!]_4$ et que $(J\ [\![\overrightarrow{q}]\!]_4\ [\![t_1]\!]_4 ... [\![t_g]\!]_4) = [\![(I\ \overrightarrow{q}\ t_1..t_n)]\!]_4$.
Remarquons également que $Principe(J,i) = [\![Principe(I,i)]\!]_4$ si $i \neq j$.

En tenant compte de ces remarques, les points (i), (ii), (iii), (v) et (vi) résultent respectivement des hypothèses d'induction appliquées à (1), (2), (3), (4) et (5).

Comme les noms $\overrightarrow{b}$ n'ont pas d'occurrences libres ni dans $\lambda\overrightarrow{b:D}.[\![N_j]\!]_4$, ni dans $Principe(J,j)$, nous pouvons affaiblir le contexte de typage de (iv), afin de nous ramener à
$E_J[[\![\Gamma]\!]_4] \vdash \lambda\overrightarrow{b:D}.[\![N_j]\!]_4 : Principe(J,j)$
Or $Principe(J,j) = \forall\overrightarrow{b:D}.[\![Principe(I,j)]\!]_4$, donc
le point (iv) se réécrit par application de la règle (Lam) :
$E_J[[\![\Gamma]\!]_4; \overrightarrow{b:D}] \vdash [\![N_j]\!]_4 : [\![Principe(I,j)]\!]_4$
ce qui est une conséquence des hypothèses d'inductions appliquées à (3).

– Si $\Lambda = Constr(j,I)\ \overrightarrow{q}\ \overrightarrow{t}$, alors $\phi = T'_j$ tel que le type du $j^{ème}$ constructeur de I est $T_j = \forall\overrightarrow{x.X}.T'_j$ où T'_j n'est pas fonctionnel, et $E_I[\Gamma] \vdash \overrightarrow{t} : \overrightarrow{X}$.
Il nous faut montrer que
$E_J[[\![\Gamma]\!]_4; \overrightarrow{b:D}] \vdash Constr(j,J)\ [\![\overrightarrow{q}]\!]_4\ \overrightarrow{b}\ [\![\overrightarrow{t}]\!]_4 : [\![T'_j]\!]_4$.
Comme nous avons
$E_J[[\![\Gamma]\!]_4; \overrightarrow{b:D}] \vdash [\![\overrightarrow{q}]\!]_4 : \overrightarrow{T_p}$ et $E_J[[\![\Gamma]\!]_4; \overrightarrow{b:D}] \vdash [\![\overrightarrow{t}]\!]_4 : [\![\overrightarrow{X}]\!]_4$, par hypothèse d'induction,
et $E_J[[\![\Gamma]\!]_4; \overrightarrow{b:D}] \vdash \overrightarrow{b} : \overrightarrow{D}$, par la règle (Var),
nous pouvons appliquer la règle (App), afin de nous ramener à :
$E_J[[\![\Gamma]\!]_4; \overrightarrow{b:D}] \vdash Constr(j,J) : \forall\overrightarrow{p:T_p}.\forall\overrightarrow{b:D}.\forall\overrightarrow{x:[\![X]\!]_4}.[\![T'_j]\!]_4$,
qui est une conséquence de la règle (Ind-Const) car on a
$Ind(J:A)[\overrightarrow{p:T_p}]\{[\![T_1]\!]_4 ... [\![T_{j-1}]\!]_4\ \forall\overrightarrow{b:D}.[\![T_j]\!]_4\ [\![T_{j+1}]\!]_4 ... [\![T_n]\!]_4\} \in E_J$,
et $\forall\overrightarrow{x:[\![X]\!]_4}.[\![T'_j]\!]_4 = [\![T_j]\!]_4$.

– Si $M \neq Constr(j,I)$, I_ind, I_rec, I_rect, alors dans ce cas, il suffit d'appliquer la règle (App) et d'utiliser les hypothèses d'induction.

- (Ind-Const)
Lorsque Λ est un constructeur, nous distinguons trois cas.
– Si $\Lambda = Constr(j,I)$ alors il nous faut montrer que
$E_J[[\![\Gamma]\!]_4; \overrightarrow{b:D}] \vdash \lambda\overrightarrow{p:T_p}.(Constr(j,J)\ \overrightarrow{p}\ \overrightarrow{b}) : \forall\overrightarrow{p:T_p}.[\![T_j]\!]_4$.
Par application de la règle (Lam), nous nous ramenons à
$E_J[[\![\Gamma]\!]_4; \overrightarrow{b:D}; \overrightarrow{p:T_p}] \vdash (Constr(j,J)\ \overrightarrow{p}\ \overrightarrow{b}) : [\![T_j]\!]_4$.
Par application de la règle (App), nous devons établir
$E_J[[\![\Gamma]\!]_4; \overrightarrow{b:D}; \overrightarrow{p:T_p}] \vdash Constr(j,J) : \forall\overrightarrow{p:T_p}.\forall\overrightarrow{b:D}.[\![T_j]\!]_4$.
Ceci résulte de la règle (Ind-Const).

– Si $\Lambda = Constr(i, I)$ avec $i \neq j$ alors il nous faut montrer que
$E_J[[\Gamma]_4; \overrightarrow{b : D}] \vdash Constr(i, J) : \forall p : T_p.[T_i]_4$ pour $i \neq j$.
Ceci résulte de la règle (Ind-Const) car on a
$Ind(J : A)[\overrightarrow{p : T_p}]\{[T_1]_4...[T_{j-1}]_4\ \forall \overrightarrow{b : D}.[T_j]_4\ [T_{j+1}]_4...[T_n]_4\} \in E_J$.

– Si $\Lambda = Constr(i, K)$ avec $K \neq I$, la preuve est analogue à celle du chapitre 7.

- (Case)

– Supposons être dans la situation où $\Lambda = Case(e,\ P,\ f_1...f_n)$ avec $e : (I\ \overrightarrow{q}\ \overrightarrow{t})$. Donc $\phi = (P\ \overrightarrow{t}\ e)$.
Il s'agit par conséquent de montrer que

$$E_J[[\Gamma]_4; \overrightarrow{b : D}] \vdash Case(\begin{array}{ll} & [e]_4, \\ & [P]_4, \\ & [f_1]_4..[f_{j-1}]_4\ (\lambda\overrightarrow{b : D}.[f_j]_4)\ [f_{j+1}]_4..[f_n]_4) \\ : & ([P]_4\ [\overrightarrow{t}]_4\ [e]_4) \end{array}$$

Pour cela nous utilisons la règle (Case).
Les prémisses nécessaires à l'application de cette règle s'obtiennent toutes par hypothèse d'induction, sauf
$E_J[[\Gamma]_4; \overrightarrow{b : D}] \vdash \lambda\overrightarrow{b : D}.[f_j]_4 : \{Constr(j, J)\ [\overrightarrow{q}]_4\}^{[P]_4}$
qui nous reste donc à établir.

Comme le constructeur $Constr(j, J)$ est de type
$\forall\overrightarrow{p : T_p}.\forall\overrightarrow{b : D}.\forall\overrightarrow{x : [X]_4}.(J\ \overrightarrow{p}\ [t_1]_4..[t_g]_4)$,
la prémisse à établir est équivalente à
$E_J[[\Gamma]_4; \overrightarrow{b : D}] \vdash$
$\lambda\overrightarrow{b : D}.[f_j]_4 : \forall\overrightarrow{b : D}.\forall\overrightarrow{x : [X]_4\sigma'}.([P]_4\ [t_1]_4\sigma'..[t_g]_4\sigma'\ (Constr(j, J)\ [\overrightarrow{q}]_4\ \overrightarrow{b}\ \overrightarrow{x}))$,
avec $\sigma' = \{\overrightarrow{p}/[\overrightarrow{q}]_4\}$.

Nous pouvons affaiblir le contexte de typage, afin de nous ramener à
$E_J[[\Gamma]_4] \vdash$
$\lambda\overrightarrow{b : D}.[f_j]_4 : \forall\overrightarrow{b : D}.\forall\overrightarrow{x : [X]_4\sigma'}.([P]_4\ [t_1]_4\sigma'..[t_g]_4\sigma'\ (Constr(j, J)\ [\overrightarrow{q}]_4\ \overrightarrow{b}\ \overrightarrow{x}))$.
Une application de la règle (Lam) nous ramène à
$E_J[[\Gamma]_4; \overrightarrow{b : D}] \vdash$
$[f_j]_4 : \forall\overrightarrow{x : [X]_4\sigma'}.([P]_4\ [t_1]_4\sigma'..[t_g]_4\sigma'\ (Constr(j, J)\ [\overrightarrow{q}]_4\ \overrightarrow{b}\ \overrightarrow{x}))$,
qui est conséquence des hypothèses d'induction appliquées sur la prémisse
$E_I[\Gamma] \vdash f_j : \{(Constr(j, I))\ \overrightarrow{q}\}^P$
utilisée dans la règle (Case) pour typer Λ.

– Supposons maintenant être dans la situation où $\Lambda = Case(e, P, \overrightarrow{f})$ où e n'est pas de type $(I\ \overrightarrow{q}\ \overrightarrow{t})$.
Dans ce cas, pour montrer $E_J[[\Gamma]_4; \overrightarrow{b : D}] \vdash Case([e]_4, [P]_4, [\overrightarrow{f}]_4) : [\phi]_4$, il

suffit d'utiliser la règle (Case) et les hypothèses d'induction.

□

10.2 Cas de l'ajout d'arguments à un type inductif

Nous allons maintenant décrire l'opération de transformation de terme après l'ajout d'arguments en tête du type d'un type inductif. Une première transformation est proposée, mais nous utiliserons une deuxième version en expliquant en quoi elle est plus puissante.

10.2.1 Ajouter une liste d'arguments à un type inductif

L'ajout de la liste d'arguments $\overrightarrow{b : D}$ dans un type inductif

$$Ind(I : A)[\Gamma_p]\{\overrightarrow{T}\}$$

consiste à ajouter dans l'environnement de déclarations le type

$$Ind(J : \forall \overrightarrow{\boldsymbol{b : D}}.A)[\Gamma_p]\{\overrightarrow{\forall \overrightarrow{\boldsymbol{b : D}}.[\![T]\!]_5}\}$$

et les principes d'inductions J_ind, J_rec, J_rect associés.

Les occurrences de I dans les types $\overrightarrow{T}$ des constructeurs sont remplacées par J grâce à l'opération $[\![.]\!]_5$, qui ajoute également les arguments $\overrightarrow{b}$.

Nous faisons ici le choix de quantifier chaque constructeur par $\forall \overrightarrow{b : D}$. Ainsi l'ajout des arguments $\overrightarrow{b}$ par $[\![.]\!]_5$ donne des termes correctement typés.

Cela suppose que ces nouveaux arguments, tous nécessairement distincts, sont sans effet dans les constructeurs, en particulier ils ne capturent pas de variables. L'utilité de cette modification élémentaire n'est révélée que si on ajoute dans une étape ultérieure de nouveaux constructeurs, dans lesquels ces arguments supplémentaires joueront un rôle.

Prenons l'exemple d'un type inductif $Ind(I : nat \to Set)[\,]\{(I\ O)\}$ auquel nous ajoutons un nouvel argument $\boldsymbol{b : D}$, pour former
$Ind(J : \boldsymbol{D} \to nat \to Set)[\,]\{\forall \boldsymbol{b : D}.(J\ \boldsymbol{b}\ O)\}$.

Les types des principes d'induction I_ind et J_ind sont alors respectivement :

$$\begin{aligned} T_{I\_ind} = \quad & \forall P : (\forall n : nat.(I\ n) \to Prop). \\ & Principe(I, 1) \to \\ & \forall n : nat. \\ & \forall e : (I\ n). \\ & (P\ n\ e) \end{aligned}$$

$$
\begin{array}{rl}
T_{J\_ind} = & \forall P : (\forall \boldsymbol{b} : \boldsymbol{D}.\forall n : nat.(J\ \boldsymbol{b}\ n) \to Prop). \\
& Principe(J,1) \to \\
& \forall \boldsymbol{b} : \boldsymbol{D}.\forall n : nat. \\
& \forall e : (J\ \boldsymbol{b}\ n)) \\
& (P\ \boldsymbol{b}\ n\ e)
\end{array}
$$

avec $Principe(I,1) = (P\ O\ Constr(1,I))$
et $Principe(J,1) = (\forall \boldsymbol{b} : \boldsymbol{D}.(P\ \boldsymbol{b}\ O\ (Constr(1,J)\ \boldsymbol{b})))$

10.2.2 Formalisation de la modification de λ-termes

Nous formalisons ici la mise à jour d'un λ-terme Λ après ajout d'arguments $\overrightarrow{b : D}$ à un type inductif, non libre dans Λ.

L'opération d'extension $[\![.]\!]_5$ est définie figure 10.2, selon le même principe qu'au chapitre 7. La principale différence est qu'il faut ajouter des abstractions par rapport aux nouveaux arguments.

$$
\begin{array}{lll}
[\![s]\!]_5 & = s & \\
[\![x]\!]_5 & = x & \\
[\![I]\!]_5 & = \lambda \overrightarrow{p : T_p}.(J\ \overrightarrow{p}\ \overrightarrow{\boldsymbol{b}}) & \\
[\![c]\!]_5 & = c' & \text{si } I \in Dep(c) \text{ et } c \underset{I,J}{\rightsquigarrow} c' \\
 & = c & \text{sinon} \\
[\![\Pi x : M.N]\!]_5 & = \Pi x : [\![M]\!]_5.[\![N]\!]_5 & \\
[\![\lambda\ x : M.N]\!]_5 & = \lambda\ x : [\![M]\!]_5.[\![N]\!]_5 & \\
[\![I\ \overrightarrow{q}\ \overrightarrow{t}]\!]_5 & = J\ [\![\overrightarrow{q}]\!]_5\ \overrightarrow{\boldsymbol{b}}\ [\![\overrightarrow{t}]\!]_5 & \\
[\![I\_ind\ \overrightarrow{q}\ Q\ \overrightarrow{N}\ \overrightarrow{t}\ e]\!]_5 & = J\_ind\ [\![\overrightarrow{q}]\!]_5\ (\overrightarrow{\boldsymbol{\lambda b : D}.[\![Q]\!]_5})\ (\overrightarrow{\boldsymbol{\lambda b : D}.[\![N]\!]_5})\ \overrightarrow{\boldsymbol{b}}\ [\![\overrightarrow{t}]\!]_5\ [\![e]\!]_5 & \\
[\![I\_rec\ \overrightarrow{q}\ Q\ \overrightarrow{N}\ \overrightarrow{t}\ e]\!]_5 & = J\_rec\ [\![\overrightarrow{q}]\!]_5\ (\overrightarrow{\boldsymbol{\lambda b : D}.[\![Q]\!]_5})\ (\overrightarrow{\boldsymbol{\lambda b : D}.[\![N]\!]_5})\ \overrightarrow{\boldsymbol{b}}\ [\![\overrightarrow{t}]\!]_5\ [\![e]\!]_5 & \\
[\![I\_rect\ \overrightarrow{q}\ Q\ \overrightarrow{N}\ \overrightarrow{t}\ e]\!]_5 & = J\_rect\ [\![\overrightarrow{q}]\!]_5\ (\overrightarrow{\boldsymbol{\lambda b : D}.[\![Q]\!]_5})\ (\overrightarrow{\boldsymbol{\lambda b : D}.[\![N]\!]_5})\ \overrightarrow{\boldsymbol{b}}\ [\![\overrightarrow{t}]\!]_5\ [\![e]\!]_5 & \\
[\![Constr(i,I)\ \overrightarrow{q}\ \overrightarrow{t}]\!]_5 & = Constr(i,J)\ [\![\overrightarrow{q}]\!]_5\ \overrightarrow{\boldsymbol{b}}\ [\![\overrightarrow{t}]\!]_5 & \\
[\![M\ \overrightarrow{N}]\!]_5 & = [\![M]\!]_5\ [\![\overrightarrow{N}]\!]_5 & \text{si } M \neq I,\ I\_ind, I\_rec, I\_rect, Constr(i,I) \\
[\![Constr(i,I)]\!]_5 & = \lambda \overrightarrow{p : T_p}.(Constr(i,J)\ \overrightarrow{p}\ \overrightarrow{\boldsymbol{b}}) & \\
[\![Constr(i,c)]\!]_5 & = Constr(i,[\![c]\!]_5) & \text{si } c \neq I \\
[\![Case(e,P,\overrightarrow{f})]\!]_5 & = Case([\![e]\!]_5, \overrightarrow{\boldsymbol{\lambda b : D}.[\![P]\!]_5}, \overrightarrow{\boldsymbol{\lambda b : D}.[\![f]\!]_5}) & \text{si } e : (I\ \overrightarrow{q}\ \overrightarrow{t}) \\
 & = Case([\![e]\!]_5, [\![P]\!]_5, [\![\overrightarrow{f}]\!]_5) & \text{sinon} \\
[\![Fix(f/n : A)\{M\}]\!]_5 & = Fix(f/n : [\![A]\!]_5)\{[\![M]\!]_5\} &
\end{array}
$$

FIG. 10.2 – Définition de $[\![\Lambda]\!]_5$ lorsque de nouveaux arguments $\overrightarrow{\boldsymbol{b}}$ sont ajoutés à I

Partant d'une preuve Λ d'une propriété ϕ, la transformation $[\![.]\!]_5$ construirait une preuve correcte $\lambda \overrightarrow{b : D}.[\![\Lambda]\!]_5$ de $\forall \overrightarrow{b : D}.[\![\phi]\!]_5$. Cependant cette transformation peut parfois être insuffisante. En effet, si la preuve Λ a été construite par une tactique

d'inversion, la structure du terme de preuve $[\![\Lambda]\!]_5$ ne tient pas compte de l'inversion sur les nouveaux arguments. En effet, une inversion engendre une analyse par cas dans laquelle des hypothèses d'égalité sur chaque argument apparaissent dans le schéma d'élimination de l'analyse par cas, ainsi que dans chaque branche. Or la transformation $[\![.]\!]_5$ ne génère pas ces hypothèses pour les nouveaux arguments.
Reprenons l'exemple du type inductif I défini par :
$Ind(I : nat \to Set)[\,]\{(I\ O)\}$

Le lemme suivant peut se montrer par une inversion sur l'hypothèse `(I n)`.

```
Lemma L : ∀n:nat (I n) -> n=0.
```

Le terme de preuve produit Λ_L a alors la forme :

$$
\begin{array}{l}
(\lambda n : nat.\lambda e : (I\ n).Case\ (\ e, \\
\qquad\qquad\qquad\qquad \lambda n0 : nat.\lambda\_ : (I\ n0).n0 = n \to n = O, \\
\qquad\qquad\qquad\qquad \lambda H0 : (O = n).(eq\_ind\ ...) \\
\qquad\qquad\qquad\qquad)\\
)\ (refl\_equal\ nat\ n)
\end{array}
$$

L'inversion a généré une hypothèse `n0=n` pour l'argument `n` dans le schéma d'élimination. Supposons qu'un nouvel argument $b : D$ soit ajouté à I, pour former
$Ind(J : D \to nat \to Set)[\,]\{\forall b : D.(I\ b\ O)\}$.
Dans ce cas $\lambda b : D.[\![\Lambda_L]\!]_5 =$

$$
\begin{array}{l}
(\lambda b : D.\lambda n : nat.\lambda e : (J\ \boldsymbol{b}\ n).Case(e, \\
\qquad\qquad\qquad\qquad \lambda \boldsymbol{b} : \boldsymbol{D}.\lambda n0 : nat.\lambda\_ : (J\ \boldsymbol{b}\ n0).n0 = n \to n = O, \\
\qquad\qquad\qquad\qquad \lambda \boldsymbol{b} : \boldsymbol{D}.\lambda H0 : (O = n).(eq\_ind...) \\
\qquad\qquad\qquad\qquad)\\
)\ (refl\_equal\ nat\ n)
\end{array}
$$

L'hypothèse $b0 = b$ n'existe pas dans le schéma d'élimination.

Supposons que par la suite le type J soit étendu avec un nouveau constructeur, et que le morceau de preuve manquant nécessite l'inversion sur l'argument b. La preuve échouera car l'hypothèse $b0 = b$ n'existera pas dans le schéma d'élimination de $[\![[\![\Lambda_L]\!]_5]\!]_1$ après ajout du nouveau constructeur.

Nous avons précisé à plusieurs reprises que nous nous intéressons à des extensions conservatives, permettant de prouver la nouvelle version d'un lemme selon le même schéma que celui existant dans l'ancienne preuve. Nous pouvons ici nous demander dans le cas d'une inversion ce que signifie "même schéma". Puisque le schéma de preuve initial comporte des hypothèses pour chaque argument du type inductif, il serait légitime de considérer que le nouveau schéma comporte également les hypothèses d'égalité sur les nouveaux arguments.

La transformation $[\![.]\!]_6$ définie figure 10.3 génère ces hypothèses de manière systématique, dans le cas d'ajout d'arguments non-dépendants, afin de simuler

l'usage d'une inversion. Dans le cas où la preuve initiale n'avait pas fait usage d'inversion, nous générons tout de même les égalités, même si elles ne serviront à rien.

Par conséquent, de nouvelles abstractions correspondant aux égalités apparaissent dans les branches des analyses par cas. Ces abstractions n'ont aucun effet sur la preuve. Elles ne sont là que pour permettre au schéma d'élimination d'avoir la forme correspondant à l'usage d'une inversion qui pourrait être nécessaire lors d'une future extension.

Sur notre exemple Λ_L, $[\![\Lambda_L]\!]_6$ produit :

$(\lambda n : nat.\ \lambda e : (J\ b\ n).$
$\quad Case\ (\ e,$
$\quad\quad \lambda b0 : D.\lambda n0 : nat.\lambda\_ : (J\ b0\ n0).\boldsymbol{b0 = b} \rightarrow n0 = n \rightarrow n = O,$
$\quad\quad \lambda b0 : D.\lambda \boldsymbol{Hb0 : (b0 = b).}\lambda H0 : (O = n).(eq\_ind...)$
$\quad\quad)$
$)\ (refl\_equal\ nat\ n)$

Dans le cas d'ajout d'arguments dépendants, les contraintes devant être générées par une inversion ne sont plus de simples égalités. C. Cornes et D. Terrasse [27] montrent le besoin d'un mécanisme d'égalité dépendante. En particulier, l'approche de C. Murthy utilise le type existentiel

```
Inductive sigS [A :Set ;P :A→Set] : Set :=
existS : ∀p :A.(P p→)(sigS A P)
```

et la propriété d'injectivité :

```
∀A :Set.∀P :A→Set.∀p :A.∀x,y :(P p).
(existS A P p x)=(existS A P p y)→ x=y.
```

Ainsi les contraintes entre arguments dépendants générées par une inversion sont des égalités de la forme `(existS A P p x)=(existS A P p y)`. Comme la complexité augmente en fonction du nombre d'arguments dépendants, nous restreignons notre simulation de l'inversion au cas d'arguments non-dépendants.

La différence de $[\![.]\!]_6$ par rapport à l'opération $[\![.]\!]_5$ concerne la transformation d'une analyse par cas sur un terme e de type $(I\ \overrightarrow{q}\ \overrightarrow{a})$.

Le schéma d'élimination est décomposé ici sous la forme

$$\lambda\overrightarrow{a' : A}.\lambda e' : (I\ \overrightarrow{q}\ \overrightarrow{a'}).P$$

où $\overrightarrow{a'}$ est une liste de noms frais et distincts de même longueur que $\overrightarrow{a}$.

Cette décomposition est licite car dans la règle de typage (Case), la condition $[(I\ \overrightarrow{q}) : A\sigma \mid B]$ impose que le type B du schéma d'élimination soit de la forme

$$\forall\overrightarrow{a : A}.\forall\_ : (I\ \overrightarrow{q}\ \overrightarrow{a}).s$$

Chaque branche de l'analyse par cas est décomposée sous la forme

$$\lambda\overrightarrow{x : X}.f_i$$

$$
\begin{array}{lll}
[\![s]\!]_6 & = s & \\
[\![x]\!]_6 & = x & \\
[\![I]\!]_6 & = \lambda \overrightarrow{p : T_p}.(J\ \overrightarrow{p}\ \overrightarrow{\boldsymbol{b}}) & \\
[\![c]\!]_6 & = c'\ \overrightarrow{\boldsymbol{b}} & \text{si } I \in Dep(c) \text{ et } c \underset{I,J}{\rightsquigarrow} c' \\
 & = c & \text{sinon} \\
[\![\Pi x : M.N]\!]_6 & = \Pi x : [\![M]\!]_6.[\![N]\!]_6 & \\
[\![\lambda\ x : M.N]\!]_6 & = \lambda\ x : [\![M]\!]_6.[\![N]\!]_6 & \\
[\![I\ \overrightarrow{q}\ \overrightarrow{t}]\!]_6 & = J\ [\![\overrightarrow{q}]\!]_6\ \overrightarrow{\boldsymbol{b}}\ [\![\overrightarrow{t}]\!]_6 & \\
[\![I\_ind\ \overrightarrow{q}\ Q\ \overrightarrow{N}\ \overrightarrow{t}\ e]\!]_6 & = J\_ind\ [\![\overrightarrow{q}]\!]_6\ (\overrightarrow{\boldsymbol{\lambda b : D}.[\![Q]\!]_6})\ \overrightarrow{\boldsymbol{\lambda b : D}.[\![N]\!]_6}\ \overrightarrow{\boldsymbol{b}}\ [\![\overrightarrow{t}]\!]_6\ [\![e]\!]_6 & \\
[\![I\_rec\ \overrightarrow{q}\ Q\ \overrightarrow{N}\ \overrightarrow{t}\ e]\!]_6 & = J\_rec\ [\![\overrightarrow{q}]\!]_6\ (\overrightarrow{\boldsymbol{\lambda b : D}.[\![Q]\!]_6})\ \overrightarrow{\boldsymbol{\lambda b : D}.[\![N]\!]_6}\ \overrightarrow{\boldsymbol{b}}\ [\![\overrightarrow{t}]\!]_6\ [\![e]\!]_6 & \\
[\![I\_rect\ \overrightarrow{q}\ Q\ \overrightarrow{N}\ \overrightarrow{t}\ e]\!]_6 & = J\_rect\ [\![\overrightarrow{q}]\!]_6\ (\overrightarrow{\boldsymbol{\lambda b : D}.[\![Q]\!]_6})\ \overrightarrow{\boldsymbol{\lambda b : D}.[\![N]\!]_6}\ \overrightarrow{\boldsymbol{b}}\ [\![\overrightarrow{t}]\!]_6\ [\![e]\!]_6 & \\
[\![Constr(i,I)\ \overrightarrow{q}\ \overrightarrow{t}]\!]_6 & = Constr(i,J)\ [\![\overrightarrow{q}]\!]_6\ \overrightarrow{\boldsymbol{b}}\ [\![\overrightarrow{t}]\!]_5 & \\
[\![M\ \overrightarrow{N}]\!]_6 & = [\![M]\!]_6\ [\![\overrightarrow{N}]\!]_6 & \text{si } M \neq I,\ I\_ind, I\_rec, I\_rect, Constr(i,I) \\
[\![Constr(i,I)]\!]_5 & = \lambda \overrightarrow{p : T_p}.(Constr(i,J)\ \overrightarrow{p}\ \overrightarrow{\boldsymbol{b}}) & \\
[\![Constr(i,c)]\!]_6 & = Constr(i, [\![c]\!]_6) & \text{si } c \neq I \\
[\![Case(e,\ \overrightarrow{\lambda a' : A.\lambda e' : (I\ \overrightarrow{q}\ \overrightarrow{a'}).P},\ \overrightarrow{\lambda x : X.f})]\!]_6 & = & \\
\multicolumn{3}{l}{Case([\![e]\!]_6,} \\
\multicolumn{3}{l}{\quad \overrightarrow{\boldsymbol{\lambda b' : D}.\lambda a' : [\![A]\!]_6}.\lambda e' : (J\ [\![\overrightarrow{q}]\!]_6\ \overrightarrow{\boldsymbol{b'}}\ \overrightarrow{a'}).\overrightarrow{\forall \boldsymbol{H_{b'} : b' = b}.[\![P]\!]_6},} \\
\multicolumn{3}{l}{\quad \overrightarrow{\boldsymbol{\lambda b' : D}.\lambda x : [\![X]\!]_6}.\overrightarrow{\boldsymbol{\lambda H_{b'} : b' = b}.([\![f]\!]_6\{\overrightarrow{\boldsymbol{b/b'}}\})})} \\
\overrightarrow{(refl\_equal\ D\ \boldsymbol{b})} & & \text{si } e : (I\ \overrightarrow{q}\ \overrightarrow{t}) \\
[\![Case(e,P,\overrightarrow{f})]\!]_6 & = Case([\![e]\!]_6, [\![P]\!]_6, [\![\overrightarrow{f}]\!]_6) & \text{sinon} \\
[\![Fix(f/n : A)\{M\}]\!]_6 & = Fix(f/n : [\![A]\!]_6)\{[\![M]\!]_6\} &
\end{array}
$$

FIG. 10.3 – Définition de $[\![\Lambda]\!]_6$ lorsque de nouveaux arguments $\overrightarrow{\boldsymbol{b}}$ sont ajoutés à I

où $\overrightarrow{x : X}$ est la liste des arguments du constructeur numéro i.

L'opération $[\![.]\!]_6$ apporte les modifications suivantes :

- Des abstractions $\lambda \overrightarrow{b' : D}$ sont ajoutées au schéma d'élimination et à chaque branche de l'analyse. En effet, chaque constructeur de J est quantifié par les nouveaux arguments.
- Des produits $\forall \overrightarrow{H_{b'} : b' = b}$ sont insérés dans le schéma d'élimination, afin que chaque branche de l'analyse comporte les hypothèses d'égalités de type $\overrightarrow{b' = b}$.
- Des abstractions $\lambda \overrightarrow{H_{b'} : b' = b}$ sont ajoutées à chaque branche de l'analyse, après les arguments $\overrightarrow{x : [\![X]\!]_6}$ des constructeurs, de manière à être conforme au type du schéma d'élimination, et générer les égalités voulues.
- Si le terme f_i contient des occurrences de I, celles-ci deviennent dans $[\![f_i]\!]_6$

des occurrences de J appliquées aux nouveaux arguments $\overrightarrow{b}$. Or les nouveaux arguments sont introduits par les noms $\overrightarrow{b'}$. Par conséquent les occurrences de $\overrightarrow{b}$ introduites dans $[\![f_i]\!]_6$ doivent être renommées par $\overrightarrow{b'}$, ce qui est noté $[\overrightarrow{\{b/b'\}}]$.

– L'analyse par cas est maintenant appliquée à $\overrightarrow{(refl\_equal\ D\ b)}$, où $refl\_equal$ est l'unique constructeur de l'égalité (voir la section 2.1.6). Ainsi $(refl\_equal\ D\ b)$ est de type $b = b$.

L'opération $[\![.]\!]_6$ lorsque I est transformé en J par ajout d'argument $\overrightarrow{b : D}$ s'étend aux contextes de typage comme dans les chapitres précédents, et aux environnements de déclarations de la manière suivante :

$$
\begin{array}{lll}
[\![\emptyset]\!]_6 & = \emptyset & \\
[\![E; c : T]\!]_6 & = [\![E]\!]_6;\ c : T;\ c' : [\![T]\!]_6 & \text{si } I \in Dep(c) \text{ et } c \underset{I,J}{\rightsquigarrow} c' \\
 & = [\![E]\!]_6;\ c : T & \text{sinon} \\
[\![E; c := d : T]\!]_6 & = [\![E]\!]_6;\ c := d : T; & \\
 & \quad c' := \lambda\overrightarrow{b : D}.[\![d]\!]_6 : \forall\overrightarrow{b : D}.[\![T]\!]_6 & \\
 & \qquad\qquad \text{si } I \in Dep(c) \text{ et } c \underset{I,J}{\rightsquigarrow} c' & \\
 & = [\![E]\!]_6;\ c := d : T & \text{sinon} \\
[\![E; Ind(I : A)[\overrightarrow{p : T_p}]\{\overrightarrow{T}\}]\!]_6 & = [\![E]\!]_6;\ Ind(I : A)[\overrightarrow{p : T_p}]\{\overrightarrow{T}\}; & \\
 & \quad Ind(J : \forall\overrightarrow{b : D}.A)[\overrightarrow{p : T_p}]\{\overrightarrow{\forall\overrightarrow{b : D}.[\![T]\!]_6}\}; & \\
 & \quad J\_ind := \Lambda_{J\_ind} : T_{J\_ind}; & \\
 & \quad J\_rec := \Lambda_{J\_rec} : T_{J\_rec}; & \\
 & \quad J\_rect := \Lambda_{J\_rect} : T_{J\_rect} & \\
[\![E; Ind(K : u)[\overrightarrow{r : R}]\{\overrightarrow{U}\}]\!]_6 & = [\![E]\!]_6;\ Ind(K' : [\![u]\!]_6)[\overrightarrow{r : [\![R]\!]_6}]\{\overrightarrow{[\![U]\!]_6}\} & \\
 & \qquad\qquad \text{si } I \in Dep(K) \text{ et } K \underset{I,J}{\rightsquigarrow} K' & \\
 & = [\![E]\!]_6;\ Ind(K : u)[\overrightarrow{r : R}]\{\overrightarrow{U}\} & \text{sinon}
\end{array}
$$

10.2.3 Correction de la preuve par réutilisation de λ-termes modifiés

Proposition 10.2.1

On suppose que

$$E_I[\Gamma] \vdash \Lambda : \phi$$

avec $Ind(I : A)[\overrightarrow{p : T_p}]\{\overrightarrow{T}\} \in E_I$. *On suppose que le type inductif* I *est transformé en* J *en lui ajoutant les arguments* $\overrightarrow{b : D}$ *non libres dans* Λ, *et que* $\mathcal{WF}([\![E_I]\!]_6)[[\![\Gamma]\!]_6]$. *Alors on a*

$$[\![E_I]\!]_6[[\![\Gamma]\!]_6] \vdash \lambda\overrightarrow{b : D}.[\![\Lambda]\!]_6 : \forall\overrightarrow{b : D}.[\![\phi]\!]_6$$

Dans la suite, nous noterons $[\![E_I]\!]_4$ par E_J. Nous avons donc

$E_J \quad = E_I \ ; \ Ind(J : \forall \overrightarrow{b : D}.A)[\overrightarrow{p : T_p}]\{\overrightarrow{\forall b : D.[\![T]\!]_6}\};$
$J\_ind := \Lambda_{J\_ind} : T_{J\_ind}; \ J\_rec := \Lambda_{J\_rec} : T_{J\_rec}; \ J\_rect := \Lambda_{J\_rect} : T_{J\_rect};$
$c'_1 := \lambda \overrightarrow{b : D}.[\![d_1]\!]_6 : \forall \overrightarrow{b : D}.[\![t_1]\!]_6; ...; c'_w := \lambda \overrightarrow{b : D}.[\![d_w]\!]_6 : \forall \overrightarrow{b : D}.[\![t_w]\!]_6 \ ;$
$Ind(K'_1 : [\![u_1]\!]_6)[\overrightarrow{r_1 : [\![R_1]\!]_6}]\{[\![\overrightarrow{U_1}]\!]_6\}; ...; Ind(K'_h : [\![u_h]\!]_6)[\overrightarrow{r_h : [\![R_h]\!]_6}]\{[\![\overrightarrow{U_h}]\!]_6\}$

où pour tout $i \in 1...v$, $c_i \underset{I,J}{\rightsquigarrow} c'_i$, et tout $i \in 1...h$, $K_i \underset{I,J}{\rightsquigarrow} K'_i$
lorsque $\{c \in Dep(\Lambda) \mid I \in Dep(c)\} = \{c_1, ..., c_w, c_{w+1}, ..., c_v, K_1, ..., K_h\}$
et que $c_1 := d_1 : t_1; ...; c_w := d_w : t_w \ c_{w+1} : t_{w+1}; ...; c_v : t_v \in E_I$ et
$Ind(K_1 : u_1)[\overrightarrow{r_1 : R_1}]\{\overrightarrow{U_1}\}; ...; Ind(K_h : u_h)[\overrightarrow{r_h : R_h}]\{\overrightarrow{U_h}\} \in E_I$.

Preuve

Par application de la règle (Lam), cette proposition revient à démontrer

$$E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash [\![\Lambda]\!]_6 : [\![\phi]\!]_6$$

La preuve se fait par induction sur les dérivations de typage de $E_I[\Gamma] \vdash \Lambda : \phi$. Examinons chaque cas :

- Les cas des règles (Ax1), (Ax2), (Ax3), (Var), (Const), (Prod), (Lam) et (Fix) se montrent de manière analogue à celle du chapitre 7.

- (App)
  Quand $\Lambda = (M \ N)$ nous distinguons 4 cas :
  - Si $\Lambda = (I \ \overrightarrow{q} \ t_1...t_g)$, alors nous devons montrer que
    $E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash (J \ [\![\overrightarrow{q}]\!]_6 \ \overrightarrow{b} \ [\![t_1]\!]_6...[\![t_g]\!]_6) \ : \ [\![A_I]\!]_6$,
    si le type A se décompose en $\forall a_1 : A_1...\forall a_g : A_g.A_I$, tel que A_I ne contient pas de produit.
    Nous utilisons pour cela la règle (App), ce qui nous ramène à établir :
    (1) $E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash J : \forall \overrightarrow{p : T_p}.\forall \overrightarrow{b : D}.\forall a_1 : A_1...\forall a_g : A_g.[\![A_I]\!]_6$
    (2) $E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash [\![\overrightarrow{q}]\!]_6 : \overrightarrow{T_p}$
    (3) $E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash \overrightarrow{b} : \overrightarrow{D}$
    (4) $E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash [\![t_i]\!]_6 : A_i$ pour $i = 1..g$

    Or (1) résulte de la règle (Ind-type),
    (2) et (4) résultent des hypothèses d'induction et du fait que $[\![T_p]\!]_6 = T_p$ et $[\![A_i]\!]_6 = A_i$,
    (3) résulte de la règle (Var).

  - Si $M = I\_ind$ (ou I_rec ou I_rect) alors $\Lambda = (I\_ind \ \overrightarrow{q} \ Q \ N_1...N_n \ t_1..t_g \ e')$.
    Le type T_{I_ind} du principe d'induction I_ind est de la forme :

$\forall \overrightarrow{p : T_p}.$
$\forall P : (\forall a_1 : A_1 ... \forall a_g : A_g.(I\ \overrightarrow{p}\ a_1 ... a_g) \rightarrow Prop).$
$Principe(I, 1) \rightarrow$
...
$Principe(I, n) \rightarrow$
$\forall a_1 : A_1 ... \forall a_g : A_g.$
$\forall e : (I\ \overrightarrow{p}\ a_1 ... a_g).$
$(P\ a_1 ... a_g\ e)$

On en déduit que $\phi = (Q\ t_1..t_g\ e')$ et que l'on a nécessairement :
(1) $E_I[\Gamma] \vdash \overrightarrow{q : T_p}$
(2) $E_I[\Gamma] \vdash Q : \forall a_1 : A_1 ... \forall a_g : A_g.(I\ \overrightarrow{q}\ a_1 ... a_g) \rightarrow Prop$
(3) $E_I[\Gamma] \vdash N_i : Principe(I, i)$ pour $i = 1..n$
(4) $E_I[\Gamma] \vdash t_i : A_i$ pour $i = 1..g$
(5) $E_I[\Gamma] \vdash e' : (I\ \overrightarrow{q}\ t_1 ... t_g)$

Il nous faut montrer que :
$E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash$
$J\_ind\ [\![\overrightarrow{q}]\!]_6\ (\lambda \overrightarrow{b : D}.[\![Q]\!]_6)\ (\lambda \overrightarrow{b : D}.[\![N_1]\!]_6) ... (\lambda \overrightarrow{b : D}.[\![N_n]\!]_6)\ \overrightarrow{b}\ [\![t_1]\!]_6 ... [\![t_g]\!]_6\ [\![e']\!]_6 :$
$[\![(Q\ t_1..t_g\ e')]\!]_6$
Nous posons $Q' = \lambda \overrightarrow{b : D}.[\![Q]\!]_6$. Puis nous appliquons la règle (App).
Comme le type T_{J_ind} du principe d'induction J_ind est de la forme :

$\forall \overrightarrow{p : T_p}.$
$\forall P : (\forall \overrightarrow{b : D}.\forall a_1 : A_1 ... \forall a_g : A_g.(J\ \overrightarrow{p}\ \overrightarrow{b}\ a_1 ... a_g) \rightarrow Prop).$
$Principe(J, 1) \rightarrow$
...
$Principe(J, n) \rightarrow$
$\forall \overrightarrow{b : D}.$
$\forall a_1 : A_1 ... \forall a_g : A_g.$
$\forall e : (J\ \overrightarrow{p}\ \overrightarrow{b}\ a_1 ... a_g).$
$(P\ \overrightarrow{b}\ a_1 ... a_g\ e)$

il nous faut maintenant montrer que :
(i) $E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash [\![\overrightarrow{q}]\!]_6 : \overrightarrow{T_p}$
(ii) $E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash Q' : \forall \overrightarrow{b : D}.\forall a_1 : A_1 ... \forall a_g : A_g.(J\ [\![\overrightarrow{q}]\!]_6\ \overrightarrow{b}\ a_1 ... a_g) \rightarrow Prop$
(iii) $E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash \lambda \overrightarrow{b : D}.[\![N_i]\!]_6 : Principe(J, i)$ pour $i = 1..n$
(iv) $E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash \overrightarrow{b} : \overrightarrow{D}$
(v) $E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash [\![t_i]\!]_6 : A_i$ pour $i = 1..g$
(vi) $E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash [\![e']\!]_6 : (J\ [\![\overrightarrow{q}]\!]_6\ \overrightarrow{b}\ [\![t_1]\!]_6 ... [\![t_g]\!]_6)$

Au vu de la définition de Q', (ii) se "réécrit" par application de la règle (Lam) :
$E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash [\![Q]\!]_6 : \forall a_1 : A_1 ... \forall a_g : A_g.(J\ \overrightarrow{p}\ \overrightarrow{b}\ a_1 ... a_g) \rightarrow Prop$

Notons que $T_p = [\![T_p]\!]_6$, $A_i = [\![A_i]\!]_6$ et que $(J\ [\![\overrightarrow{q}]\!]_6\ \overrightarrow{b}\ [\![t_1]\!]_6 ... [\![t_g]\!]_6) = [\![(I\ \overrightarrow{q}\ t_1..t_n)]\!]_6$.
Remarquons également que $Principe(J,i) = \forall \overrightarrow{b:D}.[\![Principe(I,i)]\!]_6$, donc le (iii) se "réécrit" par application de la règle (Lam) :
$E_J[[\![\Gamma]\!]_6; \overrightarrow{b:D}] \vdash [\![N_i]\!]_6 : [\![Principe(I,i)]\!]_6$

En tenant compte de ces remarques, les points (i), (ii), (iii), (v) et (vi) résultent respectivement des hypothèses d'induction appliquées à (1), (2), (3), (4) et (5). Le (iv) résulte de la règle (Var).

– Si $\Lambda = (Constr(i,I)\ \overrightarrow{q}\ \overrightarrow{t})$,
alors $\phi = T'_i$ tel que le type du $i^{ème}$ constructeur de I est $T_i = \forall \overrightarrow{x:X}.T'_i$ où T'_j ne contient pas de produit, et $E_I[\Gamma] \vdash \overrightarrow{t} : \overrightarrow{X}$.
Nous devons montrer que
$E_J[[\![\Gamma]\!]_6; \overrightarrow{b:D}] \vdash Constr(i,J)\ [\![\overrightarrow{q}]\!]_6\ \overrightarrow{b}\ [\![\overrightarrow{t}]\!]_6 : [\![T'_i]\!]_6$.

Comme nous avons $E_J[[\![\Gamma]\!]_6; \overrightarrow{b:D}] \vdash [\![\overrightarrow{q}]\!]_6 : \overrightarrow{T_p}$ par hypothèse d'induction,
$E_J[[\![\Gamma]\!]_6; \overrightarrow{b:D}] \vdash \overrightarrow{b} : \overrightarrow{D}$ par la règle (Var),
et $E_J[[\![\Gamma]\!]_6; \overrightarrow{b:D}] \vdash [\![\overrightarrow{t}]\!]_6 : [\![\overrightarrow{X}]\!]_6$ par hypothèse d'induction,
nous pouvons appliquer la règle (App), afin de nous ramener à :
$E_J[[\![\Gamma]\!]_6; \overrightarrow{b:D}] \vdash Constr(i,J) : \forall \overrightarrow{p:T_p}.\forall \overrightarrow{b:D}.\forall \overrightarrow{x:[\![X]\!]_6}.[\![T'_i]\!]_6$.

Ceci résulte de la règle (Ind-Const) car on a
$Ind(J : \forall \overrightarrow{b:D}.A)[\overrightarrow{p:T_p}]\{\forall \overrightarrow{b:D}.[\![T_1]\!]_6 ... \forall \overrightarrow{b:D}.[\![T_n]\!]_6\} \in E_J$,
et $\forall \overrightarrow{x:[\![X]\!]_6}.[\![T'_i]\!]_6 = [\![T_i]\!]_6$.

– Si $M \neq I, I\_ind, I\_rec,\ I\_rect,\ Constr(i,I)$, alors dans ce cas, il suffit d'appliquer la règle (App) et d'utiliser les hypothèses d'induction.

• (Ind-Const)
Nous distinguons encore deux cas.
– Si $\Lambda = Constr(i,I)$ et $\phi = \forall p : T_p.T_i$, alors nous devons montrer que
$E_J[[\![\Gamma]\!]_6; \overrightarrow{b:D}] \vdash \lambda \overrightarrow{p:T_p}.(Constr(i,J)\ \overrightarrow{p}\ \overrightarrow{b}) : \forall \overrightarrow{p:T_p}.[\![T_i]\!]_6$.
Après avoir appliquée la règle (Lam), nous obtenons
$E_J[[\![\Gamma]\!]_6; \overrightarrow{b:D}; \overrightarrow{p:T_p}] \vdash (Constr(i,J)\ \overrightarrow{p}\ \overrightarrow{b}) : [\![T_i]\!]_6$.

Cela nous permet d'appliquer la règle (App) pour obtenir
$E_J[[\![\Gamma]\!]_6; \overrightarrow{b:D}; \overrightarrow{p:T_p}] \vdash Constr(i,J) : \forall \overrightarrow{p:T_p}.\forall \overrightarrow{b:D}.[\![T_i]\!]_6$,
ce qui résulte de la règle (Ind-Const).

– Si $\Lambda = Constr(i,K)$ avec $K \neq I$, la preuve est analogue à celle donnée au chapitre 7.

- (Ind-Type)
  Nous distinguons toujours deux cas.
  - Si $\Lambda = I$ alors $\phi = \forall \overrightarrow{p : T_p}.A$ et nous devons montrer que
    $E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash \lambda \overrightarrow{p : T_p}.(J\ \overrightarrow{p}\ \overrightarrow{b}) : \forall \overrightarrow{p : T_p}.[\![A]\!]_6$.

    Après avoir appliquée la règle (Lam), nous obtenons
    $E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}; \overrightarrow{p : T_p}] \vdash (J\ \overrightarrow{p}\ \overrightarrow{b}) : [\![A]\!]_6$.

    Après avoir appliquée la règle (App), nous obtenons
    $E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}; \overrightarrow{p : T_p}] \vdash J : \forall \overrightarrow{p : T_p}.\forall \overrightarrow{b : D}.[\![A]\!]_6$,
    ce qui résulte de la règle (Ind-Type).
  - Si $\Lambda = K$ avec $K \neq I$, la preuve est analogue à celle donnée au chapitre 7.

- (Case)
  Nous distinguons le cas où l'analyse par cas porte sur un terme de type $(I\ \overrightarrow{q}\ \overrightarrow{t})$ et le cas contraire. Nous décomposerons le type A en $\forall \overrightarrow{a : A}.A_I$, tel que le type A_I ne contienne pas de produit.

  - Supposons être dans la situation où

    $$\Lambda = Case(e,\ \lambda \overrightarrow{a' : A}.\lambda e' : (I\ \overrightarrow{q}\ \overrightarrow{a'}).P,\ \lambda \overrightarrow{x_1 : X_1}.f_1 ... \lambda \overrightarrow{x_n : X_n}.f_n)$$

    avec $e : (I\ \overrightarrow{q}\ \overrightarrow{t})$, et où $\overrightarrow{x_i}$ est la liste des arguments du constructeur numéro i.
    Dans ce cas la règle de typage qui fournit l'hypothèse est

    $$\frac{\begin{array}{c} Ind(I : \forall \overrightarrow{a : A}.A_I)[\overrightarrow{p : T_p}]\{T_1..T_n\} \in E_I \qquad E_I[\Gamma] \vdash e : (I\ \overrightarrow{q}\ \overrightarrow{t}) \qquad \sigma = \{\overrightarrow{p}/\overrightarrow{q}\} \\ E_I[\Gamma] \vdash P' : \forall \overrightarrow{a' : A}.\forall e' : (I\ \overrightarrow{q}\ \overrightarrow{a'}).B \\ [(I\ \overrightarrow{q}) : (\forall \overrightarrow{a : A}.A_I)\sigma \mid \forall \overrightarrow{a' : A}.\forall e' : (I\ \overrightarrow{q}\ \overrightarrow{a'}).B] \\ (E_I[\Gamma] \vdash \lambda \overrightarrow{x_i : X_i}.f_i : \{Constr(i, I)\ \overrightarrow{q}\}^{P'})_{i=1..n} \end{array}}{E_I[\Gamma] \vdash Case(e,\ P',\ \lambda \overrightarrow{x_1 : X_1}.f_1 ... \lambda \overrightarrow{x_n : X_n}.f_n) : (P'\ \overrightarrow{t}\ e)}$$

    où P' est une notation pour $\lambda \overrightarrow{a' : A}.\lambda e' : (I\ \overrightarrow{q}\ \overrightarrow{a'}).P$.

    Au vu de la définition donnée figure 2.3, la prémisse
    $[(I\ \overrightarrow{q}) : (\forall \overrightarrow{a : A}.A_I)\sigma \mid \forall \overrightarrow{a' : A}.\forall e' : (I\ \overrightarrow{q}\ \overrightarrow{a'}).B]$
    impose que B soit une sorte s avec les conditions de la figure 2.3.

    La propriété à démontrer ici est
    $E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash [\![\Lambda]\!]_6 : [\![(P'\ \overrightarrow{t}\ e)]\!]_6$,
    c'est-à-dire :

$$E_J[[\Gamma]_6; \overrightarrow{b : D}] \vdash Case(\quad [e]_6,$$
$$\overrightarrow{\lambda b' : D.\lambda a' : [A]_6}.\lambda e' : (J\ [\overrightarrow{q}]_6\ \overrightarrow{b'}\ \overrightarrow{a'}).\overrightarrow{\forall H_{b'} : b' = b.[P]_6},$$
$$\overrightarrow{\lambda b' : D.\lambda x : [X]_6}.\overrightarrow{\lambda H_{b'} : b' = b.([f]_6\{\overrightarrow{b/b'}\})}$$
$$) \quad \overrightarrow{(refl\_equal\ D\ b)}\ :\ [(P'\ \overrightarrow{t}\ e)]_6$$

Le terme $[(P'\ \overrightarrow{t}\ e)]_6$ se simplifie en $[P]_6\{\overrightarrow{a'}/[\overrightarrow{t}]_6\}\{e'/[e]_6\}$.

Comme $E_J[[\Gamma]_6; \overrightarrow{b : D}] \vdash \overrightarrow{(refl\_equal\ D\ b)} : \overrightarrow{b = b}$,
la règle (App) peut s'appliquer. Il nous reste maintenant à montrer que

$$E_J[[\Gamma]_6; \overrightarrow{b : D}] \vdash Case(\quad [e]_6,$$
$$\overrightarrow{\lambda b' : D.\lambda a' : [A]_6}.\lambda e' : (J\ [\overrightarrow{q}]_6\ \overrightarrow{b'}\ \overrightarrow{a'}).\overrightarrow{\forall H_{b'} : b' = b.[P]_6},$$
$$\overrightarrow{\lambda b' : D.\lambda x : [X]_6}.\overrightarrow{\lambda H_{b'} : b' = b.([f]_6\{\overrightarrow{b/b'}\})}$$
$$) \quad : \ \overrightarrow{\forall H_{b'} : (b = b).[P]_6\{\overrightarrow{a'}/[\overrightarrow{t}]_6\}\{e'/[e]_6\}}$$

Pour cela nous utilisons la règle (Case).
Par hypothèse d'induction, nous avons
$E_J[[\Gamma]_6; \overrightarrow{b : D}] \vdash [e]_6 : (J\ [\overrightarrow{q}]_6\ \overrightarrow{b}\ [\overrightarrow{t}]_6)$.

Si on note P'' le schéma d'élimination
$\overrightarrow{\lambda b' : D.\lambda a' : [A]_6}.\lambda e' : (J\ [\overrightarrow{q}]_6\ \overrightarrow{b'}\ \overrightarrow{a'}).\overrightarrow{\forall H_{b'} : b' = b.[P]_6}$, alors on a

$P'' : \overrightarrow{\forall b' : D.\forall a' : [A]_6}.\forall e' : (J\ [\overrightarrow{q}]_6\ \overrightarrow{b'}\ \overrightarrow{a'}).s$
(car $P' : \overrightarrow{\forall a' : A.\forall e' : (I\ \overrightarrow{q}\ \overrightarrow{a'}).s}$) et

$(P''\ \overrightarrow{b}\ [\overrightarrow{t}]_6\ [e]_6) =$
$(\overrightarrow{\forall H_{b'} : b' = b.[P]_6})\{\overrightarrow{b'}/\overrightarrow{b}\}\{\overrightarrow{a'}/[\overrightarrow{t}]_6\}\{e'/[e]_6\} =$
$\overrightarrow{\forall H_{b'} : b = b.[P]_6}\{\overrightarrow{a'}/[\overrightarrow{t}]_6\}\{e'/[e]_6\}$.

Donc le type de l'analyse par cas a bien la forme attendue $(P''\ \overrightarrow{b}\ [\overrightarrow{t}]_6\ [e]_6)$
pour l'application de la règle (Case).

$$\frac{\begin{array}{c} Ind(J : \forall\overrightarrow{b : D}.\forall\overrightarrow{a : A}.A_I)[\overrightarrow{p : T_p}]\{\overrightarrow{\forall b : D.[\![T]\!]_6}\} \in E_J \\ E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash [\![e]\!]_6 : (J\ [\![\overrightarrow{q}]\!]_6\ \overrightarrow{b}\ [\![\overrightarrow{t}]\!]_6) \qquad \sigma' = \{\overrightarrow{p}/[\![\overrightarrow{q}]\!]_6\} \\ E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash P'' : \forall\overrightarrow{b' : D}.\forall\overrightarrow{a' : [\![A]\!]_6}.\forall e' : (J\ [\![\overrightarrow{q}]\!]_6\ \overrightarrow{b'}\ \overrightarrow{a'}).s \\ [(J\ [\![\overrightarrow{q}]\!]_6) : (\forall\overrightarrow{b : D}.\forall\overrightarrow{a : A}.A_I)\sigma' \mid \forall\overrightarrow{b' : D}.\forall\overrightarrow{a' : [\![A]\!]_6}.\forall e' : (J\ [\![\overrightarrow{q}]\!]_6\ \overrightarrow{b'}\ \overrightarrow{a'}).s] \\ (E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash \lambda\overrightarrow{b' : D}.\lambda\overrightarrow{x_i : [\![X_i]\!]_6}.\lambda\overrightarrow{H_{b'} : b' = b}.([\![f_i]\!]_6\{\overrightarrow{b/b'}\}) \\ : \{Constr(i, J)\ \overrightarrow{q}\}^{P''})_{i=1..n} \end{array}}{\begin{array}{l} E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash Case(\ [\![e]\!]_6, \\ \qquad\qquad P'', \\ \qquad\qquad \lambda\overrightarrow{b' : D}.\lambda\overrightarrow{x_1 : [\![X_1]\!]_6}.\lambda\overrightarrow{H_{b'} : b' = b}.([\![f_1]\!]_6\{\overrightarrow{b/b'}\}) \\ \qquad\qquad ...\lambda\overrightarrow{b' : D}.\lambda\overrightarrow{x_n : [\![X_n]\!]_6}.\lambda\overrightarrow{H_{b'} : b' = b}.([\![f_n]\!]_6\{\overrightarrow{b/b'}\}) \\ \qquad) \ : \ (P''\ \overrightarrow{b}\ [\![\overrightarrow{t}]\!]_6\ [\![e]\!]_6) \end{array}}$$

Il nous reste à établir les propriétés (1) et (2) suivantes :
(1) $[(J\ [\![\overrightarrow{q}]\!]_6) : \forall\overrightarrow{b : D}.\forall\overrightarrow{a : A}.A_I\sigma' \mid \forall\overrightarrow{b' : D}.\forall\overrightarrow{a' : [\![A]\!]_6}.\forall e' : (J\ [\![\overrightarrow{q}]\!]_6\ \overrightarrow{b'}\ \overrightarrow{a'}).s]$
(2) $E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash \lambda\overrightarrow{b' : D}.\lambda\overrightarrow{x_i : [\![X_i]\!]_6}.\lambda\overrightarrow{H_{b'} : b' = b}.([\![f_i]\!]_6\{\overrightarrow{b/b'}\}) : \{Constr(i, J)\ \overrightarrow{q}\}^{P''}$ pour tout $i \in 1..n$.

En appliquant la première règle de la figure 2.3, (1) devient : $[(J\ [\![\overrightarrow{q}]\!]_6\ \overrightarrow{b}\ \overrightarrow{a}) : A_I\sigma' \mid \forall e' : (J\ [\![\overrightarrow{q}]\!]_6\ \overrightarrow{b}\ \overrightarrow{a}).s]$

Or nous savons que $[(I\ \overrightarrow{q}\ \overrightarrow{a}) : A_I\sigma \mid \forall e' : (I\ \overrightarrow{q}\ \overrightarrow{a}).s]$
donc A_I est une sorte s_1, et la propriété (1) se montre par l'une des trois dernières règles de la figure 2.3.

Le constructeur $Constr(i, J)$ s'écrit $\forall\overrightarrow{p : T_p}.\forall\overrightarrow{b : D}.\forall\overrightarrow{x_i : [\![X_i]\!]_6}.(J\ \overrightarrow{p}\ \overrightarrow{b}\ [\![\overrightarrow{t}]\!]_6)$.
D'après la figure 2.4, la propriété (2) se réécrit :
$E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash \lambda\overrightarrow{b' : D}.\lambda\overrightarrow{x_i : [\![X_i]\!]_6}.\lambda\overrightarrow{H_{b'} : b' = b}.([\![f_i]\!]_6\{\overrightarrow{b/b'}\}) : \forall\overrightarrow{b : D}.\forall\overrightarrow{x_i : [\![X_i]\!]_6}.(P''\ \overrightarrow{b}\ [\![\overrightarrow{t}]\!]_6\ (Constr(i, J)\ \overrightarrow{q}\ \overrightarrow{b}\ \overrightarrow{x_i}))$

Ceci devient en appliquant la règle (Lam) :
$E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}; \overrightarrow{b' : D}; \overrightarrow{x_i : [\![X_i]\!]_6}] \vdash \lambda\overrightarrow{H_{b'} : b' = b}.([\![f_i]\!]_6\{\overrightarrow{b/b'}\}) : (P''\ \overrightarrow{b'}\ [\![\overrightarrow{t}]\!]_6\ (Constr(i, J)\ \overrightarrow{q}\ \overrightarrow{b'}\ \overrightarrow{x_i}))$

En remplaçant P'' par sa définition, on obtient :
$E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}; \overrightarrow{b' : D}; \overrightarrow{x_i : [\![X_i]\!]_6}] \vdash \lambda\overrightarrow{H_{b'} : b' = b}.([\![f_i]\!]_6\{\overrightarrow{b/b'}\}) : (\forall\overrightarrow{H_{b'} : b' = b}.[\![P]\!]_6)\{\overrightarrow{b'}/\overrightarrow{b'}\}\{\overrightarrow{a'}/[\![\overrightarrow{t}]\!]_6\}\{e'/(Constr(i, J)\ \overrightarrow{q}\ \overrightarrow{b'}\ \overrightarrow{x_i})\}$

La règle (Lam) s'applique à nouveau :
$E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}; \overrightarrow{b' : D}; \overrightarrow{x_i : [\![X_i]\!]_6}; \overrightarrow{H_{b'} : b' = b}] \vdash ([\![f_i]\!]_6\{\overrightarrow{b/b'}\}) :$

$[\![P]\!]_6\{\overrightarrow{b'}/\overrightarrow{b'}\}\{\overrightarrow{a'}/[\![\overrightarrow{t}]\!]_6\}\{e'/(Constr(i,J)\ \overrightarrow{q}\ \overrightarrow{b'}\ \overrightarrow{x_i})\}$

Nous savons que les hypothèses $\overrightarrow{H_{b'} : b' = b}$ ne sont pas utilisées dans $[\![f_i]\!]_6$, et que les noms $\overrightarrow{b'}$ sont libres dans $[\![P]\!]_6$. Ainsi, par α-conversion nous pouvons réécrire (2) par :
$E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}; \overrightarrow{x_i : [\![X_i]\!]_6}] \vdash [\![f_i]\!]_6 : [\![P]\!]_6\{\overrightarrow{a'}/[\![\overrightarrow{t}]\!]_6\}\{e'/(Constr(i,J)\ \overrightarrow{q}\ \overrightarrow{b}\ \overrightarrow{x_i})\}$

D'après l'hypothèse de typage de l'analyse par cas de e, nous avons déduit que
$E_I[\Gamma] \vdash \lambda\overrightarrow{x_i : X_i}.f_i : \{Constr(i,I)\ \overrightarrow{q}\}^{P'}$.
Autrement dit
$E_I[\Gamma] \vdash \lambda\overrightarrow{x_i : X_i}.f_i : \forall\overrightarrow{x_i : X_i}.P\{\overrightarrow{a'}/\overrightarrow{t}\}\{e'/(Constr(i,I)\ \overrightarrow{q}\ \overrightarrow{x_i})\}$

En appliquant l'hypothèse d'induction sur cette propriété, nous obtenons
$E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash$
$\lambda\overrightarrow{x_i : [\![X_i]\!]_6}.[\![f_i]\!]_6 : \forall\overrightarrow{x_i : [\![X_i]\!]_6}.[\![P]\!]_6\{\overrightarrow{a'}/[\![\overrightarrow{t}]\!]_6\}\{e'/(Constr(i,J)\ \overrightarrow{q}\ \overrightarrow{b}\ \overrightarrow{x_i})\}$

Finalement, en appliquant (Lam) nous obtenons exactement la propriété (2) cherchée.

– Supposons maintenant être dans la situation où $\Lambda = Case(e, P, \overrightarrow{f})$ où e n'est pas de type $(I\ \overrightarrow{q}\ \overrightarrow{t})$.
Dans ce cas, pour montrer $E_J[[\![\Gamma]\!]_6; \overrightarrow{b : D}] \vdash Case([\![e]\!]_6, [\![P]\!]_6, [\![\overrightarrow{f}]\!]_6) : [\![\phi]\!]_6$, il suffit d'utiliser la règle (Case) et les hypothèses d'induction.

□

10.3 Discussion

La transformation $[\![.]\!]_4$ permet d'ajouter une liste d'arguments $\overrightarrow{b}$ à un constructeur donné d'un type inductif I. La transformation $[\![.]\!]_6$ permet quant à elle d'ajouter une liste d'arguments $\overrightarrow{b}$ au type inductif I.

Cette deuxième transformation n'est pas équivalente à l'utilisation de la première sur chaque constructeur. La différence est que dans le cas de $[\![.]\!]_6$, les nouveaux arguments apparaissent dans le nouveau type de I. La conséquence majeure se situe au niveau du schéma d'élimination d'une analyse par cas, qui est plus complexe dans le second cas.

Chapitre 11

Implémentation d'un prototype

Ce chapitre a pour objectif de décrire le prototype implémentant les commandes de base de notre environnement en OCaml. Ces commandes s'intègrent au système Coq (version 7). Le prototype nommé ExPaRe (pour Extend Parameter Reuse) est disponible à l'adresse `http://www.iie.cnam.fr/~boite/expare`.

11.1 Les commandes

De légères restrictions ou différences existent par rapport aux commandes présentées dans les chapitres précédents.

11.1.1 Commandes implémentées et ajoutées

Les commandes `Extend`, `Suppress`, `Param`, `Argum`, `Reuse` sont pour chacune d'elles implémentée en Ocaml, et ajoutée à l'environnement Coq grâce au *Pré-Processeur-Pretty-Printer* Camlp4 [29] qui permet d'étendre la grammaire du langage Vernacular. Nous gardons trace dans des structures appropriées de chaque application de commande, afin de pouvoir effectuer les transformations nécessaires lors de la réutilisation et pour le calcul des dépendances.

11.1.2 Commandes partiellement implémentées

La commande `ArgumConstr` n'a pas encore été implémentée, et la commande `Update d as d'` ne permet pas encore de mettre à jour un axiome, mais cela ne saurait tarder. Elle ne permet pas non plus de mettre à jour une définition car elle est équivalente dans ce cas à utiliser la commande `Reuse d as d'`. En effet, `Reuse d as d'` crée un nouveau terme, en mettant à jour tous les renommages possibles, et ajoute éventuellement des métavariables. Or une définition est un λ-terme qui peut être traité exactement de la même manière : renommer les mises à jour et ajouter des métavariables pour tenir compte des morceaux de définition manquants.

Il s'agit donc de la même opération vue sous deux angles différents, correspondant en fait aux deux aspects de l'isomorphisme de Curry-Howard. L'instanciation des

métavariables, sous l'angle de vue preuve, consiste à prouver les obligations de preuve générées, à l'aide de tactiques. L'instanciation des métavariables, sous l'angle de vue définition, consiste à fournir le morceau de terme manquant.

11.2 Renommer les constructeurs

Chaque commande crée un nouveau type inductif, une nouvelle définition, ou un nouveau lemme, dont le nom est donné par l'utilisateur. Dans le cas de la création d'un nouveau type inductif, le renommage des constructeurs communs est nécessaire pour ne pas avoir à gérer des phénomènes de surcharge qui n'est par ailleurs pas acceptée en Coq. L'ancien type inductif et le nouveau coexistent, rendant possible leur utilisation conjointe. Les principes d'induction sur le nouveau type inductif sont également générés et ajoutés à l'environnement automatiquement par Coq.

Par exemple, lorsque `I` est défini par

```
Inductive I [p1:U1;...;pl:Ul] : T :=
  d1 : t1 | ... | dn : tn.
```

la commande d'extension de `I` :

```
Extend I as J
 with c1 : t'1 | ... | ck : t'k.
```

génère en fait le type `J`

```
Inductive J [p1:U1;...;pl:Ul] : T :=
  d1_J : t1[I := J] | ... | dn_J : tn[I := J]
| c1   : t'1        | ... | ck  : t'k.
```

qui est ajouté à l'environnement, ainsi que les principes d'induction associés générés par Coq.

Les anciens constructeurs de `I` doivent donc être renommés par un nom frais pour assurer l'unicité car Coq ne permet pas la surcharge des noms de constructeurs. Ici, pour l'exemple, le renommage est réalisé en ajoutant un suffixe. Dans notre prototype, une fonction de génération de noms frais est utilisée.

11.3 Renommages à prendre en compte

Dans notre prototype nous gardons trace des extensions à l'aide d'une liste d'associations `lextassoc` sous forme de couples `(I,J)`. Ainsi la commande précédente

```
Extend I as J ...
```

ajoute dans `lextassoc` le couple `(I,J)`, mais également les couples `(d1,d1_J)...(dn,dn_J)` afin de connaître les substitutions à appliquer.

De même les commandes

```
Suppress I2 as J2 ...
Param I3 as J3 ...
ArgumConstr I4 as J4 ...
Argum I5 as J4 ...
```

ajoutent également dans `lextassoc` les associations entre ancien type inductif et nouveau, et les associations entre anciens constructeurs et nouveaux.

Enfin

```
Update d as d' ...
```

ajouterait le couple `(d,d')` dans `lextassoc`.

Il est possible d'étendre un type `I` en `J` puis d'étendre `I` en `K`. Les types `J` et `K` coexistent alors dans l'environnement. Cependant, nous ne gardons trace dans notre liste d'associations `lextassoc` que de la dernière extension réalisée, à savoir le couple `(I,K)`.

Cela permet de ne pas avoir à préciser quelles mises à jour prendre en compte lors de la réutilisation ou de la mise à jour de définition. Ainsi la commmande `Reuse`

```
Reuse L on J1...Jp.
```

se résume à

```
Reuse L.
```

La liste `J1...Jp` des nouveaux noms à utiliser pour la réutilisation n'a pas besoin d'être précisée. Nous appliquons tous les renommages apparaissant dans la liste d'associations `lextassoc`.

De même la commande `Update I as J` ne nécessite pas de liste de nouveaux noms à introduire.

11.4 `Refine` et les métavariables dans Coq

La commande `Reuse L` récupère le terme de `L`, applique tous les renommages correspondant à la liste `lextassoc`, complète éventuellement les trous avec des métavariables, et ajoute si nécessaire les nouveaux paramètres et arguments. Ce terme est ensuite utilisé par la tactique `Refine` [5]. Cette tactique permet de résoudre un but en donnant explicitement un terme de preuve partiel. Chaque partie manquante est identifiée par une métavariable ?. Pour chaque métavariable présente dans ce terme partiel, un sous-but est généré. La preuve de ces sous-buts terminera la preuve du sous-but initial.

Pour prouver par exemple que le successeur `S` sur les entiers naturels est une fonction croissante, on utilise ici la tactique `Refine` pour prouver le deuxième sous-but après l'étape d'induction.

```
Lemma S_croissant : (x:nat) (le x (S x)).
Intros x.
Induction x.
Auto.

  x : nat
  Hrecx : (le x (S x))
  ============================
   (le (S x) (S (S x)))
```

comme on dispose du lemme `le_n_S : (n,m :nat)(le n m)->(le (S n) (S m))` on peut fournir un terme de preuve partiel pour prouver notre but :

```
Refine (le_n_S x (S x) ?).

  x : nat
  Hrecx : (le x (S x))
  ============================
   (le x (S x))
```

Le troisième argument de `le_n_S` est une métavariable ? correspondant à un terme de type `(le x (S x))`.

Le système Coq possède la notion de métavariable, mais sous la terminologie de *variables existentielles*. Le terme *métavariable* est plutôt réservé à une classe de variables introduites par Jean-Christophe Filliâtre pour implémenter la tactique `Refine`.

Le terme de preuve incomplet est un terme contenant des variables existentielles. La tactique `Refine` fonctionne en trois temps. Le système remplace les variables existentielles par des métavariables, et essaie de calculer leur type. S'il échoue pour l'une d'elles, l'utilisateur devra fournir le type `T` de cette métavariable sous la forme `(? :T)` à la place de `?`.

Ensuite, le terme donné à `Refine` est mis à plat, de manière à ce que les métavariables se trouvent au premier niveau, c'est-à-dire avec une profondeur de 1. La profondeur d'un terme t est le nombre de règles de grammaire appliquées pour construire le terme. Par exemple dans $(f\ ?_1)$, $?_1$ a une profondeur de 1.

Pour faire la mise à plat d'un terme t, lorsqu'un sous-terme t_1 de niveau 1 contient une métavariable $?_i$, `Refine` remplace t_1 par une métavariable fraîche $?_j$, et garde trace que $?_j$ devra être instanciée par t_1, noté $?_j \Rightarrow t_1$.

Par exemple le terme $(f\ a\ ?_1\ [x : nat](e\ ?_2))$ est transformé en $(f\ a\ ?_1\ ?_3)$ avec $?_3 \Rightarrow [x : nat]?_4$ et $?4 \Rightarrow (e\ ?_5)$. Les métavariables $?_1$ et $?_5$ sont bien de niveau 1, et donneront les buts à prouver.

Enfin, la tactique calcule et applique un script de tactiques correspondant au terme transformé. Par exemple si le terme est une abstraction, `Refine` applique la tactique `Intro`.

Pour les métavariables aucune tactique n'est appliquée, donc le sous-but reste ouvert. Ainsi, un sous-but est engendré pour chaque trou dans le terme de départ.

11.5 Dépendances

La commande `Dep` lance le calcul de dépendances de tous les objets présents dans l'environnement, et affiche le graphe sous forme graphique. Un fichier de dépendances, dans un format de type `Makefile`, est généré : chaque identificateur est suivi de la liste de ces dépendances. Un outil autonome `dep2dot` écrit en Caml-lex, génère à partir de ce fichier, un fichier au format `dot`, affiché par l'outil `dotted` de la suite Graphviz [88]. Enfin un autre outil, `dotcolor` aussi écrit en Camllex, permet de transformer le fichier `dot` en ajoutant des couleurs.

En effet, pour marquer les dépendances opaques ou transparentes, les nœuds du graphe sont colorés. A titre de documentation pour les utilisateurs, la sémantique des couleurs attribuées est la suivante :

- jaune : type inductif étendu,
- rouge : définition ou lemme ayant une dépendance transparente avec un type étendu,
- vert : définition ou lemme ayant une dépendance transparente avec un type étendu, et une définition ou lemme pour ce type étendu a été défini,
- orange : définition ou lemme ayant une dépendance opaque avec un type étendu,
- vert clair : définition ou lemme ayant une dépendance opaque avec un type étendu, et une définition ou lemme pour ce type étendu a été défini.

Les lemmes non reliés à un type étendu sont incolores et ne doivent subir aucune modification.

La figure 11.1 représente un graphe de dépendances avec une dépendance opaque (`b` en orange) et une transparente (`c` en rouge) quand I est étendu en J.

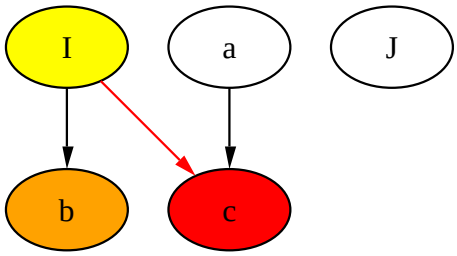


FIG. 11.1 – Exemple de graphe de dépendances

Le nœud c est rouge, il sera complété ou réutilisé pour former c' une fois les obligations de preuve prouvées. Le nœud passera alors au vert. Le nœud b est orange,

il suffit donc de définir b' en effectuant les renommages et aucune obligation de preuve n'est générée. Le nœud passe alors au vert clair.

En relançant la commande, le graphe est mis à jour, et se transforme comme figure 11.2.

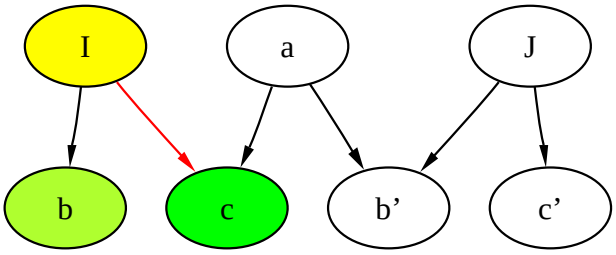


FIG. 11.2 – Graphe de dépendances mis à jour

Le calcul de dépendances cherche dans les termes l'utilisation des principes d'induction créés automatiquement par le système COQ lors de la définition d'un type inductif, afin de déterminer si une dépendance est transparente ou non.

La présence de principes d'induction est repéré par leur suffixe `_ind`, `_rec` ou `_rect`. Les principes d'induction définis par l'utilisateur avec la commande `Scheme` ou `Functional Scheme`, devront par convention être nommés avec un suffixe `_ind`, `_rec` ou `_rect` de manière à être pris en compte par notre calcul de dépendances transparentes.

11.6 Cohabitation avec COQ

Chaque commande de base écrite en Ocaml est ajoutée à l'environnement COQ en mode *bytecode* grâce au *Pré-Processeur-Pretty-Printer* Camlp4 [29] qui permet d'étendre la grammaire du langage Vernacular. Il serait possible de les intéger en mode natif à condition de recompiler COQ avec ces nouvelles commandes.

Les commandes utilisent beaucoup de fonctions des librairies se trouvant dans les sources de COQ. En particulier, nous laissons faire une partie du travail au système existant, comme par exemple la génération des principes d'induction à partir de la définition du type inductif. De même, la vérification que les nouveaux constructeurs respectent la condition de positivité, ou bien qu'un constructeur est petit, est laissé au système. Enfin, nous laissons à la tactique `Refine` le soin de générer les obligations de preuve à partir d'un terme contenant des métavariables.

Chapitre 12

Preuves de propriétés sémantiques

Dans ce chapitre, nous faisons un rapide état de l'art sur l'application des méthodes formelles dans le cadre de la sémantique des langages. Puis nous donnons un exemple d'utilisation de notre méthode de réutilisation, pour réutiliser des preuves de propriétés sémantiques, en nous appuyant sur notre prototype.

12.1 Sémantique et Méthodes Formelles

12.1.1 Historique

Dès 1967, R. W. Floyd [38], puis C.A.R. Hoare en 1969 [42] définissent un mode de programmation où les instructions sont préfixées et postfixées par des assertions logiques. La sémantique associée permet alors de prouver la correction du programme.

J. McCarthy et J. Painter [58] donnèrent la première preuve formelle de la correction d'un compilateur d'un langage d'expressions arithmétiques. Une version mécanisée de cette preuve fut réalisée en 1972 par R. Milner et R. Weyhrauch [61].

En 1971, D. Scott et C. Strachey formulent les premières sémantiques dénotationnelles de langages de programmation. Plus tard, G. D. Plotkin [73] et J. C. Reynolds [80] présentèrent leurs travaux sur le typage et la sémantique opérationnelle structurelle (SOS). Ces travaux font références, comme ceux de L. Damas et R. Milner [28] sur la preuve de sûreté de typage de langages fonctionnels. La propriété de sûreté de typage garantit qu'une expression bien typée n'occasionnera pas d'erreur de typage à l'exécution.

G. D. Plotkin fut l'un des premiers à donner un style *déduction naturelle* aux règles de sémantique et de typage. Ces travaux furent repris par G. Kahn pour un langage "à la ML" [46]. Par exemple, la règle de typage de l'abstraction se présente ainsi :

$$\frac{\Gamma \vdash x{:}\tau_1 \quad \Gamma \oplus x{:}\tau_1 \vdash y{:}\tau_2}{\Gamma \vdash \lambda x.y{:}\tau_1}(Abs)$$

Elle est vue comme une règle d'inférence entre jugements de typage dans un contexte donné. Une telle approche permet d'établir un jugement de typage par une succession d'applications de règles d'inférence.

Les sémantiques opérationnelles peuvent être de deux natures : à "grands pas" ou à "petits pas". Une sémantique à "grands pas" (*Big Step*) lie une expression à sa valeur. Une sémantique à réduction, ou dite à "petits pas"(*Small Step*) spécifie les règles de réduction élémentaire d'une expression. Plotkin introduisit cette sémantique à réduction dans [73] en 1981, puis A. K. Wright et M. Felleisen [95] en 1994 présentèrent une approche syntaxique de cette sémantique, vue alors comme un système de réécriture.

12.1.2 Usage des méthodes formelles

Dans le domaine de la sémantique, on distingue les propriétés du langage et les propriétés des programmes. On parle aussi de preuve dans le langage et sur le langage.

La preuve des propriétés de programmes (ou vérification), avec une technique comme la logique de Hoare par exemple, s'attache à établir qu'un programme particulier, lorsqu'on précise quelles sont les préconditions, respecte des postconditions. Pour chaque instruction du programme, des règles permettent de vérifier que les postconditions peuvent s'obtenir à partir des préconditions. L'outil Krakatoa [57] permet de certifier un programme Java annoté par des assertions au format JML *(Java Modeling Language)*. Ces assertions forment des *pre* et *post* conditions, et des invariants de classes. Krakatoa fournit une représentation de la sémantique du programme Java à Why [37] pour calculer des obligations de preuves. Ces preuves réalisées en Coq permettent d'établir que pour toute execution du programme, les *post*-conditions s'obtiennent bien en remontant jusqu'aux *pre*-conditions.

Les preuves de propriétés de langages consistent quant à elles à prouver qu'un langage de programmation respecte des propriétés comme la conservation du typage, de déterminisme de l'évaluation, de décidabilité, de confluence, de normalisation... La preuve ne concerne donc pas un programme particulier écrit dans ce langage, mais le langage dans son ensemble. C'est dans cette branche de la sémantique que nous envisageons l'application de notre méthode de réutilisation.

De nombreux assistants à la preuve généraux comme Coq [5], Lego [50], PVS [87], Isabelle [67], HOL [41] sont utilisés dans des travaux portant sur différents langages ou noyaux de langages comme ML et Java [12, 13, 18, 20, 21, 34, 89, 66, 91, 92, 65, 93]. Dans ces travaux, d'importantes propriétés sur le langage considéré ou bien la correction d'un compilateur sur le langage sont montrées. Mais d'une part les prouveurs utilisés ne sont pas spécialisés à des problèmes sémantiques, et d'autre part très peu de mécanismes permettent de réutiliser.

D'autres systèmes sont dédiés à la sémantique. L'outil TinkerType [48] de B. Pierce permet de construire des systèmes de types, en assemblant un choix de règles parmi des centaines et de *feature* comme le polymorphisme ou le sous-typage. L'outil

vérifie la consistence et produit un vérificateur de types. Mais nous ne pouvons pas faire de raisonnement avec cet outil.

D. Syme et A. D. Gordon [90] ont défini la notion de *réduction guidée*. Leur outil consiste à automatiser le plus possible les preuves de sûreté de typage. Ils utilisent une procédure de décision SVC [6] une fois que le problème a été rendu décidable par l'application de transformations spécifiques, permettant de traiter les cas non triviaux. L'automatisation permet ici de résoudre partiellement le problème de la réutilisation.

Le système CENTAUR [19] de l'INRIA Sophia Antipolis est un environnement interactif pour spécifier des langages. La définition de la syntaxe concrète et abstraite y est facilitée, ainsi que la sémantique et la transformation de programmes. Les spécifications sont écrites en Typol puis traduites en Prolog. D. Terrasse [91] a écrit un traducteur de Typol vers Coq pour relier CENTAUR à un assistant à la preuve, permettant ainsi d'exécuter des spécifications et de raisonner avec celles-ci, comme par exemple dans [15]. Cependant, le problème de la réutilisation reste entier : le moindre changement nécessite de tout vérifier à nouveau.

S. Gay [39] propose un cadre de formalisation de systèmes de types de Π-calcul en Isabelle/HOL. Une méthodologie et l'utilisation d'une théorie générale sur les environnements de types permet de réutiliser spécifications et preuves lors de variantes du système de type, mais cette réutilisation est finalement faite à main.

12.2 Preuves de propriétés sémantiques

Nous allons maintenant illustrer l'utilisation de notre outil de réutilisation de spécifications et de preuves, dans le cadre de preuves de propriétés sémantiques.

Dans un premier temps, nous donnons la description d'un premier développement formel contenant la sémantique d'un langage d'expressions, et la preuve que l'évaluation d'une expression est déterministe.

Puis, en utilisant notre environnement de réutilisation, nous enrichissons le langage de trois manières différentes, et montrons dans chacun des cas que l'évaluation reste déterministe en réutilisant la preuve initiale.

12.2.1 Le langage initial L_1

Les expressions

Cet exemple de langage, que nous appelerons L_1, est inspiré de [17]. Il s'agit d'un langage d'expressions arithmétiques, ne comportant que des constantes et des additions.

La grammaire suivante représente l'algèbre des expressions de L_1.

$$\begin{array}{rcll} e & ::= & Cst\ n & \text{constante entière } n \\ & | & Add(e,e) & \text{addition de deux expressions} \end{array}$$

Cette grammaire se code en Coq de manière naturelle en un type inductif :

```
Inductive L1 : Set :=
   Cst : nat → L1
 | Add : L1 → L1 → L1.
```

Evaluation d'une expression

L'évaluation d'une expression e de L_1 est un entier naturel n, ce que nous notons $e \longrightarrow n$. Les règles d'évaluation se limitent aux deux règles suivantes : (+ désigne l'addition sur les entiers naturels)

$$\frac{}{Cst\ n \longrightarrow n} \qquad \frac{e_1 \longrightarrow n_1 \quad e_2 \longrightarrow n_2}{Add(e_1,e_2) \longrightarrow n_1+n_2}$$

Plutôt que de formaliser l'évaluation par une fonction définie par cas, nous avons choisi d'utiliser un prédicat *eval* à deux arguments : l'expression à évaluer, et l'évaluation entière. Ainsi le prédicat (*eval e n*) est vrai lorsque e s'évalue en n.

Deux raisons motivent ce choix. D'une part, en Coq les fonctions doivent être totales. Or nos futures extensions pourraient induire une évaluation correspondant à une fonction partielle. D'autre part, la représentation de l'évaluation par un prédicat permet de coder un système de règles non guidées par la syntaxe. Par exemple, nous pourrions ajouter la règle

$$\frac{e_1 \longrightarrow n_1 \quad e_2 \longrightarrow n_2}{Add(e_1,e_2) \longrightarrow n_2+n_1}$$

ce qui ne pourrait pas être modélisé par une fonction.

La spécification Coq de la sémantique consiste à définir le type inductif `eval` dont chaque constructeur correspond à une règle d'évaluation.

```
Inductive eval : L1 → nat → Prop :=
  evalCst : ∀n:nat.(eval (Cst n) n)
| evalAdd : ∀e1,e2:L1.∀n1,n2:nat.
          (eval e1 n1) → (eval e2 n2) →
          (eval (Add e1 e2) n1+n2).
```

Preuve du déterminisme de l'évaluation

Nous établissons maintenant le déterminisme de l'évaluation : si une expression e s'évalue en n d'une part et si e s'évalue en m d'autre part, alors nous avons nécessairement $n = m$.

Voici l'énoncé et le script de tactiques Coq qui établit le déterminisme de l'évaluation :

```
Lemma eval_det : ∀e:L1.∀n,m:nat.(eval e n) → (eval e m) → n=m.
Proof.
 Intros e n m Hevaln.
 Generalize m. Clear m.
 Induction Hevaln.
 Intros m Hevalm. Apply CC.
 Assumption.
 Intros m Hevalm; Inversion Hevalm.
 Replace n0 with n1. Replace n3 with n2.
 Trivial.
 Apply HrecHevaln0; Assumption.
 Apply HrecHevaln1; Assumption.
Qed.
```

La preuve se fait par induction sur les règles de la sémantique. Nous utilisons au cours de cette preuve le lemme suivant [1] :

```
Lemma CC : ∀n,m:nat.(eval (Cst n) m) → n=m.
Proof.
 Intros n m H.
 Inversion H.
 Trivial.
Qed.
```

La preuve est réalisée en effectuant une inversion sur l'hypothèse `H` de type `(eval (Cst n) m)`. L'inversion induit une preuve par cas sur les règles de `eval`.

12.2.2 Ajouter une règle d'évaluation

Pour illustrer la méthode de réutilisation, nous ajoutons la règle d'évaluation suivante :

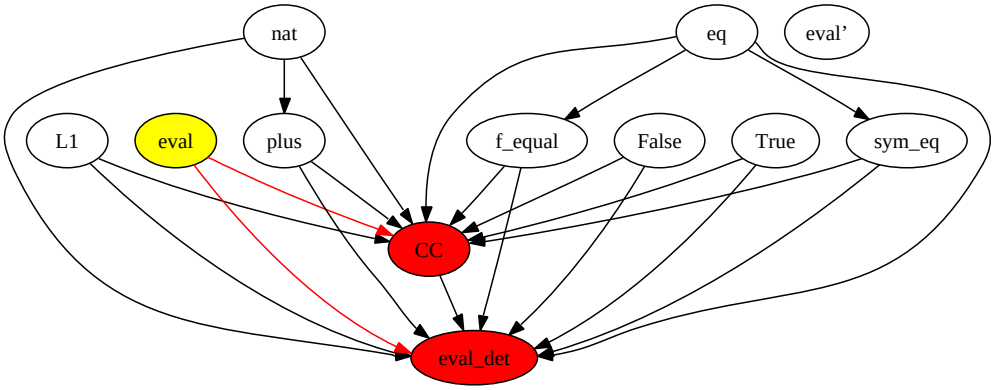
$$\frac{e \longrightarrow n}{Add(Cst\ O,e) \longrightarrow n}$$

Le langage L_1 reste le même mais le type inductif `eval` doit être étendu. Pour cela, nous utilisons la commande suivante :

```
Extend eval as eval' with
 evalAdd' : ∀e:L1.∀n:nat.
            (eval' e n) →(eval' (Add (Cst O) e) n).
```

L'utilisation de la commande `Dep` affiche le graphe de dépendances suivant :

[1] le lemme `CC` n'est absolument pas nécessaire, ni usuel, il n'est là que pour illustrer l'utilisation de lemmes.



Le type `eval` sera marqué en jaune pour rappeler qu'il a été modifié. Comme `CC` et `eval_det` ont une dépendance transparente avec ce type modifié, ils seront marqués en rouge pour souligner que ces lemmes doivent être mis à jour et que la réutilisation de ce lemme par la commande `Reuse` génèrera des obligations de preuve.

Comme `eval_det` dépend de `CC`, `CC` doit d'abord être mis à jour. Nous utilisons pour cela la commande `Reuse` :

```
Lemma CC2 : ∀n,m:nat.(eval' (Cst n) m) → n=m.
 Reuse CC.

1 subgoal

  n : nat
  m : nat
  H : (eval' (Cst n) m)
  ============================
   ∀e:L1;n0:nat.(eval' e n0)→(Add (Cst O) e)=(Cst n)→n0=m→n=m
```

Une seule obligation de preuve est générée, qui peut être montrée par le script suivant :

```
 Intros e n0 H0 Abs.
 Inversion Abs.
Qed.
```

La tactique `Reuse` modifie le terme de preuve du lemme `CC` de type `∀n,m :nat.(eval (Cst n) m) → n=m` par la fonction $[\![.]\!]_1$. Le script permettant de décharger l'obligation de preuve permet d'instancier la métavariable ajoutée par $[\![.]\!]_1$. Ainsi, comme le prédit la proposition 7.5.1, nous obtenons une preuve de type $[\![$`∀n,m :nat.(eval (Cst n) m) → n=m`$]\!]_1$=`∀n,m :nat.(eval' (Cst n) m) → n=m`.

Nous pouvons maintenant envisager de mettre à jour `eval_det` :

```
Lemma eval'_det : ∀e:L1.∀n,m:nat.(eval' e n) → (eval' e m) → n=m.
Reuse eval_det.

2 subgoals

  e : L1
  n : nat
  m : nat
  Hevaln : (eval' e n)
  e1 : L1
  e2 : L1
  n1 : nat
  n2 : nat
  Hevaln1 : (eval' e1 n1)
  HrecHevaln1 : ∀m:nat.(eval' e1 m)→n1=m
  Hevaln0 : (eval' e2 n2)
  HrecHevaln0 : ∀m:nat.(eval' e2 m)→n2=m
  m0 : nat
  Hevalm : (eval' (Add e1 e2) m0)
  ============================
   ∀e0:L1; n0:nat.
    (eval' e0 n0)
    →(Add (Cst O) e0)=(Add e1 e2)→n0=m0→n1+n2=m0

subgoal 2 is:
 ∀e0:L1; n0:nat.
  (eval' e0 n0) →(∀m:nat.(eval' e0 m)→n0=m)
   →∀m0:nat.(eval' (Add (Cst O) e0) m0)→n0=m0
```

Le deuxième sous-but était attendu puisque l'on a ajouté un nouveau constructeur dans `eval`, mais une autre obligation de preuve apparaît, provenant du morceau de terme de preuve correspondant à l'inversion sur `Hevalm`.

La preuve de ces obligations de preuve permet de terminer la preuve du déterminisme de la nouvelle évaluation. Le graphe de dépendances mis à jour permet de visualiser que tout est à jour.

La fonction $[\![.]\!]_1$ utilisée par la tactique `Reuse` modifie le terme de preuve du lemme `eval_det`. Une fois que les métavariables ajoutées sont instanciées obtenons une preuve `∀e :L1.∀n,m :nat.(eval' e n) → (eval' e m) → n=m`.

12.2.3 Ajouter la soustraction aux expressions

Nous décidons maintenant d'enrichir les expressions de L_1 avec la soustraction, pour former le langage L_2. La grammaire des expressions est enrichie par une

construction Sub :

$$\begin{array}{rcll} e & ::= & Cst\ n & \\ & | & Add(e,e) & \\ & | & \boldsymbol{Sub(e,e)} & \text{soustraction de deux expressions} \end{array}$$

Nous utilisons la commande `Extend` pour générer le type des expressions de L_2 à partir de L_1 :

```
Extend L1 as L2 with Sub : L2 → L2 → L2.
```

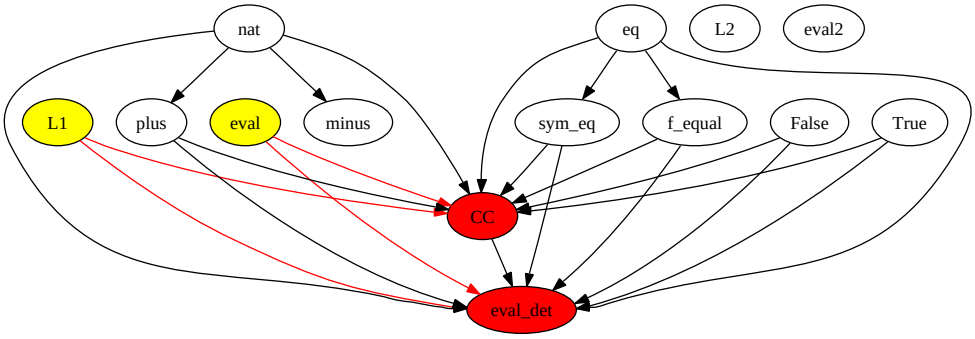
Maintenant que cette nouvelle construction existe, une règle d'évaluation supplémentaire doit être ajoutée pour compléter la sémantique de L_2 :

$$\frac{e_1 \longrightarrow n_1 \quad e_2 \longrightarrow n_2}{Sub(e_1,e_2) \longrightarrow n_1 - n_2}$$

Pour cela nous utilisons à nouveau la commande `Extend` :

```
Extend eval as eval2 with
 evalSub : ∀e1,e2:L2.∀n1,n2:nat.
          (eval2 e1 n1) → (eval2 e1 n2) →
          (eval2 (Sub e1 e2) n1-n2).
```

À ce niveau, l'utilisation de la commande `Dep` affiche le graphe de dépendances suivant :



On peut y lire (grâce au jeu des couleurs) que les types `L1` et `eval` ont été modifiés. Les lemmes `CC` et `eval_det` ont une dépendance transparente avec ces types modifiés et doivent donc être mis à jour. La commande `Reuse CC as CC2` permet de génèrer le nouveau lemme `CC2` à partir de `CC`, et applique la méthode de réutilisation.

```
Reuse CC as CC2.
```

génère donc l'énoncé

```
Lemma CC2 : ∀n,m:nat.(eval2 (Cst0 n) m) → n=m.
```

et applique la tactique `Reuse` qui génère à son tour une obligation de preuve.

```
  n : nat
  m : nat
  H : (eval2 (Cst0 n) m)
  ============================
    ∀e1,e2:L2.∀n1,n2:nat.
    (eval2 e1 n1)
    →(eval2 e2 n2)→(Sub e1 e2)=(Cst0 n)→(minus n1 n2)=m→n=m
```

Cette dernière obligation de preuve peut être montrée grâce au script suivant :

```
Intros e1 e2 n1 n2 Heval1 Heval2 Abs.
Inversion Abs.
Qed.
```

Nous pouvons alors mettre à jour `eval_det` :

```
Reuse eval_det as eval2_det.
```

qui génère les deux obligations de preuve :

```
  e : L2
  n : nat
  m : nat
  Hevaln : (eval2 e n)
  e1 : L2
  e2 : L2
  n1 : nat
  n2 : nat
  Hevaln1 : (eval2 e1 n1)
  HrecHevaln1 : ∀m:nat.(eval2 e1 m)→n1=m
  Hevaln0 : (eval2 e2 n2)
  HrecHevaln0 : ∀m:nat.(eval2 e2 m)→n2=m
  m0 : nat
  Hevalm : (eval2 (Add0 e1 e2) m0)
  ============================
   ∀e3,e4:L2.∀n3,n4:nat.
    (eval2 e3 n3)
    →(eval2 e4 n4)
      →(Sub e3 e4)=(Add0 e1 e2)→(minus n3 n4)=m0→n1+n2=m0
```

```
subgoal 2 is:
 ∀e1,e2:L2.∀n1,n2:nat.
  (eval2 e1 n1)
  →(∀m:nat.(eval2 e1 m)→n1=m)
    →(eval2 e2 n2)
      →(∀m:nat.(eval2 e2 m)→n2=m)
        →∀m0:nat.(eval2 (Sub e1 e2) m0)→(minus n1 n2)=m0
```

L'utilisateur décharge ces deux obligations de preuve avec le script suivant :

```
Intros e3 e4 n3 n4 He3 He4 Abs; Inversion Abs.

Intros.
Inversion H3.
Replace n0 with n1.
Replace n3 with n2.
Trivial.
Apply H2; Assumption.
Apply H0; Assumption.
```

Pour les lemmes `CC2` et `eval2_det`, la tactique `Reuse` utilise la fonction $[\![.]\!]_1$. La proposition 7.5.1 annonce que nous obtenons à chaque fois une preuve correcte, ce qui est confirmé ici par la vérification du typage de Coq lorsque la preuve est sauvegardée.

12.2.4 Ajouter des variables

Nous voulons maintenant ajouter le notion de variable dans le langage L_2. Nous nous donnons un ensemble abstrait `var` de noms de variables :

```
Parameter var : Set.
```

et nous ajoutons le constructeur `Var` dans le langage des expressions :

```
Extend L2 as L3 with Var : var → L3.
```

La règle d'évaluation à ajouter pour les variables nécessite l'ajout d'un environnement d'évaluation venant paramètrer les règles. Un environnement est ici modélisé par une fonction ρ associant à toute variable sa valeur entière.

$$\overline{Var(v) \longrightarrow_\rho \rho(v)}$$

Nous allons utiliser la commande `Param ... and extend with ...`, qui permet d'ajouter des paramètres, et des constructeurs en même temps. Il s'agit en fait de la composition des commandes élémentaires `Param` et `Extend`.

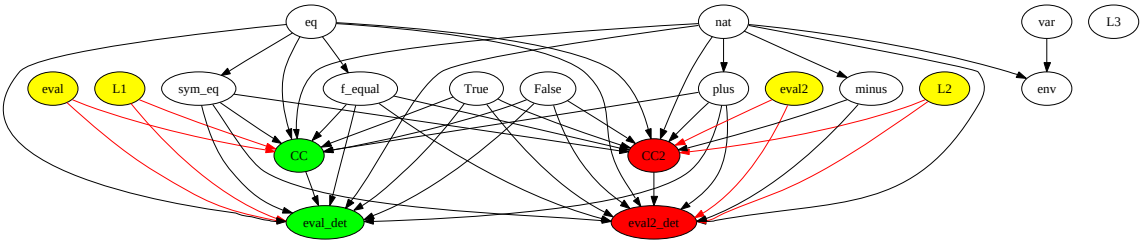
```
Definition  env := (var → nat).

Param eval2 as eval3 with rho:env
 and extend with
 evalVar:  ∀v:var.(eval3 rho (Var v) (rho v)).
```

Le type généré est bien paramétré par l'environnement `rho` :

```
Inductive eval3 [rho:env]  : L3→nat→Prop :=
      evalCst1 : ∀n:nat.(eval3 rho (Cst1 n) n)
    | evalAdd1 : ∀e1,e2:L3.∀n1,n2:nat.
                  (eval3 rho e1 n1)→(eval3 rho e2 n2)
                  →(eval3 rho (Add1 e1 e2) n1+n2)
    | evalSub0 : ∀e1,e2:L3.∀n1,n2:nat.
                  (eval3 rho e1 n1)→(eval3 rho e2 n2)
                    →(eval3 rho (Sub0 e1 e2) (minus n1 n2))
    | evalVar : ∀v:var.(eval3 rho (Var v) (rho v))
```

Le graphe de dépendances montre en rouge les lemmes `CC2` et `eval2_det` à mettre à jour.



La mise à jour de `CC2` s'effectue avec la commande `Reuse` à laquelle nous indiquons le paramètre à ajouter `rho`.

```
Lemma CC3 : ∀rho:env.∀n,m:nat.(eval3 rho (Cst1 n) m) → n=m.
Reuse CC2 with rho.

  rho : env
  n : nat
  m : nat
  H : (eval3 rho (Cst1 n) m)
  ============================
   ∀v:var.(Var v)=(Cst1 n)→(rho v)=m→n=m
```

L'obligation de preuve générée se montre grâce au script suivant :

```
Intros v Abs; Inversion Abs.
Qed.
```

Enfin, la mise à jour de `eval2_det` est réalisée par

```
Lemma eval3_det : ∀rho:env.∀e:L3.∀n,m:nat.(eval3 rho e n) →
                  (eval3 rho e m) → n=m.
Reuse eval2_det with rho.

3 subgoals

  rho : env
  e : L3
  n : nat
  m : nat
  Hevaln : (eval3 rho e n)
  e1 : L3
  e2 : L3
  n1 : nat
  n2 : nat
  Hevaln1 : (eval3 rho e1 n1)
  HrecHevaln1 : ∀m:nat.(eval3 rho e1 m)→n1=m
  Hevaln0 : (eval3 rho e2 n2)
  HrecHevaln0 : ∀m:nat.(eval3 rho e2 m)→n2=m
  m0 : nat
  Hevalm : (eval3 rho (Add1 e1 e2) m0)
  ============================
   ∀v:var.(Var v)=(Add1 e1 e2)→(rho v)=m0→n1+n2=m0

subgoal 2 is:
 ∀v:var.(Var v)=(Sub0 e1 e2)→(rho v)=m0→(minus n1 n2)=m0
subgoal 3 is:
 ∀v:var.∀m0:nat (eval3 rho (Var v) m0)→ (rho v)=m0
```

générant trois obligations de preuve, qui se déchargent par :

```
Intros v Abs; Inversion Abs.

Intros v Abs; Inversion Abs.

Intros v m0 Hevalm0.
Inversion Hevalm0.
Trivial.
Qed.
```

La tactique `Reuse` transforme le terme de preuve Λ_{CC2} (respectivement Λ_{eval2_det}) du lemme `CC2` (respectivement `eval2_det`) en $[\![[\![\Lambda_{CC2}]\!]_3]\!]_1$ (respectivement $[\![[\![\Lambda_{eval2\_det}]\!]_3]\!]_1$). Les propositions 7.5.1 et 9.3.1 annoncent que nous obtenons une preuve correcte, une fois les métavariables instanciées.

12.2.5 Ajouter un argument

Dans cet exemple, nous ajoutons un argument au prédicat d'évaluation, pour représenter un flux de sortie sous la forme d'une liste de valeurs. La construction `Print e` est ajoutée au langage, dont l'évaluation a pour effet d'ajouter la valeur d'évaluation de `e` sur le flux de sortie.

$$\frac{e \longrightarrow_{\rho}^{f} v}{Print(e) \longrightarrow_{\rho}^{v::f} v}$$

Nous ajoutons d'abord la construction `Print e` au langage `L3` :

```
Extend L3 as L4 with Print : L4 → L4.
```

Puis nous ajoutons la règle d'évaluation, nécessitant l'ajout de l'argument `f : flow`. Le type du flux de sortie `flow` désigne une liste d'entiers naturels.

```
Argum eval3 as eval4 with f:flow
 and extend with
 evalPrint: ∀f:flow.∀e:L4.∀v:nat.(eval4 rho f e v)→
            (eval4 rho (cons v f) (Print e) v).
```

Le flux `f` évolue dans la règle d'évaluation de `Print e`, donc nous n'aurions pas pu la représenter par un paramètre.

Maintenant que les spécifications sont modifiées, passons à la phase de mise à jour des preuves. Le lemme `CC3` est mis à jour par la commande `Reuse` :

```
Lemma CC4 : ∀f:flow.∀rho:env.∀n,m:nat.
            (eval4 rho f (Cst2 n) m) → n=m.
Reuse CC3 with f:flow.
```

La preuve initiale utilisait une inversion sur les règles, donc une obligation de preuve est générée due à l'ajout d'une règle. Le script suivant décharge cette obligation de preuve.

```
Intros f1 e v Heval Abs.
Inversion Abs.
Qed.
```

Enfin, nous pouvons mettre à jour la preuve du déterminisme de l'évaluation en réutilisant la preuve sur `L3` :

```
Lemma eval4_det : ∀f:flow.∀rho:env.∀e:L4.∀n,m:nat.
                  (eval4 rho f e n) → (eval4 rho f e m) → n=m.
Reuse eval3_det with f:flow.
```

Quatre obligations de preuve sont générées. Les trois premières proviennent des inversions dans les cas de l'addition, soustraction et variable, et se montrent par

l'absurde. La dernirère obligation de preuve correspond à l'étape d'induction sur le nouveau constructeur. Une simple inversion sur la règle permet de la résoudre :

```
  rho : env
  f : flow
  e : L4
  v : nat
  Hevaln : (eval4 rho f e v)
  HrecHevaln : ∀m:nat.(eval4 rho f e m)→v=m
  m : nat
  Hevalm : (eval4 rho (cons v f) (Print e) m)
  ============================
   v=m

Inversion Hevalm; Trivial.
Qed.
```

La tactique `Reuse` transforme le terme de preuve Λ_{CC3} (respectivement Λ_{eval3_det}) du lemme `CC3` (respectivement `eval3_det`) en $[\![\lambda f : flow.[\![\Lambda_{CC3}]\!]_6]\!]_1$ (respectivement $[\![\lambda f : flow.[\![\Lambda_{eval3\_det}]\!]_6]\!]_1$). Les propositions 7.5.1 et 10.2.1 annoncent que nous obtenons une preuve correcte, une fois que les métavariables introduites par $[\![.]\!]_1$ sont instanciées.

12.3 Conclusion

Le petit langage que nous venons d'étudier pourrait continuer à être enrichi, par exemple en ajoutant une séquence `Seq(e1,e2)` dont l'évaluation concatène les deux flux de sortie provenant respectivement de l'évaluation de `e1` et `e2` :

$$\frac{e1 \longrightarrow_{\rho}^{f1} v1 \qquad e2 \longrightarrow_{\rho}^{f2} v2}{Seq(e1,e2) \longrightarrow_{\rho}^{f1 \wedge f2} v2}$$

Nous pouvons ensuite envisager d'ajouter des instructions conditionnelles comme un *if then else*, une boucle *while*... A chaque fois, la commande `Reuse` permettra de n'avoir à prouver que le strict nécessaire.

L'utilisation des commandes de base présentées dans le chapitre 4 permet d'enrichir un développement formel, de manière réellement conviviale.

La réutilisation automatique permet ensuite de n'avoir à traiter que les nouveaux cas à travers la résolution d'obligations de preuve. Ainsi la frustration de devoir prouver à nouveau la même chose à chaque modification des spécifications est levée. Cependant, pour avoir un regard critique sur notre travail, l'application de notre méthode sur des études de cas nous amène à deux constatations. La première est que certaines obligations de preuve peuvent sembler inattendues à l'utilisateur. En effet, lorsque l'utilisateur réutilise une preuve par induction ou une preuve par cas

après ajout d'un constructeur, il s'attend à n'avoir comme obligation de preuve que le nouveau cas. Mais si une tactique d'inversion a été utilisée dans la preuve source, d'autres obligations de preuve apparaissent car une inversion induit une analyse par cas. La plupart du temps nous sommes en présence de cas absurdes, faciles à décharger. Afin d'éviter ce petit désagrément, il serait nécessaire d'automatiser la preuve de ces cas absurdes, ce qui semble envisageable.

La seconde constatation est que l'utilisateur doit avoir en tête de manière précise la structure de la preuve. En effet, le fait de ne générer que les obligations de preuve nécessaires, ce qui était notre l'objectif, peut faire perdre le fil de la preuve. C'est ce que l'on pourrait qualifier de "revers de la médaille".

Chapitre 13

Conclusion

Dans ce document, nous nous sommes attachés à décrire un ensemble de commandes de base, formant un environnement de modifications, pour modifier des spécifications et adapter en conséquence les preuves associées en vue de leur réutilisation. L'intérêt porte avant tout sur l'aspect formel et automatique de la réutilisation de preuves.

Le but de ce mémoire est de démontrer que la réutilisation formelle et automatique est réalisable, dans des systèmes complexes, et que cette réalisation peut effectivement être mise en pratique, afin de satisfaire un réel besoin de la part des utilisateurs.

Nous avons montré la correction de notre méthode de réutilisation de preuves après modifications, en décomposant les transformations en transformations élémentaires, et en montrant la correction de chacune d'entre elles.

D'une manière générale, pour que les transformations présentées dans les chapitres 7, 8, 9 et 10 puissent s'appliquer (en les adaptant), nous devons nous trouver dans un système représentant les objets preuve dans un λ-calcul typé avec types inductifs, avec la possibilité d'introduire des métavariables. Ce travail, exécuté dans le cadre du CCI et de l'outil Coq, peut être adapté pour permettre la réutilisation de preuves dans d'autres systèmes d'aide à la preuve. Le système Lego par exemple, implémente le ECC *(Extended Calculus of Constructions)* et la théorie des types UTT *(Unifying Theory of dependent Types)* [49], assez proche du CCI. En effet, le UTT est basé à la fois sur le calcul des constructions et sur la théorie des types de Martin-Löf [68], et fournit également des types inductifs. De plus Lego permet l'introduction de métavariables dans la construction d'une preuve. Ainsi la méthode de réutilisation que nous proposons est transposable à la théorie UTT et pourrait être implémenté dans Lego.

Dans le système Isabelle dans lequel il est maintenant possible de générer des termes de preuve grâce aux travaux de Berghofer et Nipkow [11], nous pouvons envisager d'appliquer notre méthode. D'autant plus que la notion de métavariable a

été introduite, sous le nom de variable d'unification.

L'assistant à la preuve Alfa [56] est également un cadre idéal pour l'implémentation de notre méthode de réutilisation. Dans ce système basé sur la théorie des types de Martin-Löf [68], l'utilisateur construit sa preuve à l'aide d'un λ-terme contenant des métavariables. La construction de la preuve consiste à instancier petit à petit les métavariables. Notre méthode s'adapte donc sans problèmes à Alfa.

Les perspectives à court terme de cette thèse sont d'inclure et formaliser d'autres types de modifications, et donc d'autres types de réutilisations. En parallèle, nous continuerons à implémenter le prototype pour Coq, dans la nouvelle syntaxe de la version 8.

Le sujet est vaste et cette thèse ouvre une voie vers l'intéressant problème de la réutilisation. Notre approche consistant à adapter les termes en fonction des modifications faites sur les spécifications permet d'envisager toute sorte de modifications.

Nous pourrions par exemple envisager d'ajouter un argument dépendant à un constructeur, pour exprimer une contrainte comme l'égalité entre deux autres arguments par exemple. Le morceau de terme de preuve manquant, correspondant à ce nouvel argument, pourrait être introduit par une métavariable. A charge pour l'utilisateur de fournir la preuve pour instancier cette métavariable.

L'exemple auquel nous pensons est l'ajout de références dans un noyau de langage fonctionnel. Outre l'ajout de nouvelles constructions (en gras dans la grammaire ci-dessous) et donc de nouveaux constructeurs, dans le langage, il convient d'ajouter dans la sémantique la notion de mémoire, plus précisément une mémoire qui évolue au cours de l'évaluation des expressions.

e ::=	c	constante	|	$\boldsymbol{e}$ **;** $\boldsymbol{e}$	**séquence**
|	x	identificateur	|	**while(**$\boldsymbol{e}$**)** $\boldsymbol{e}$	**boucle**
|	$e\ e$	application	|	**ref** $\boldsymbol{e}$	**référence**
|	fun x e	abstraction	|	**!**$\boldsymbol{e}$	**déférencement**
|	let x=e in e	définition locale	|	$\boldsymbol{e}$ **:=**$\boldsymbol{e}$	**affectation**
|	if e then e else e	conditionelle			

c : :=	n	entier
|	b	booléen
|	**unit**	**résultat d'un effet de bord**

Dans [18] nous montrons la sûreté du typage pour chacun de ces deux langages, vis à vis d'une réduction à petits-pas. Les règles de réduction de tête s'expriment également à l'aide d'un type inductif. Le passage de Mini-ML à Ref-ML incorporant les mutables consiste à ajouter des règles, donc des constructeurs, pour réduire les nouvelles expressions, mais il consiste aussi à ajouter un état mémoire m_1 avant réduction et un état mémoire m_2. Cette mémoire est ajoutée aux anciennes règles,

mais avec la particularité que la mémoire avant et la mémoire après réduction sont identiques.

$$
\begin{array}{rcll}
(\text{fun } x\ e)\ v\ /\ \boldsymbol{m} & \rightarrow_\epsilon & e[v/x]\ /\ \boldsymbol{m} & (1)\\
\text{let } x = v \text{ in } e\ /\ \boldsymbol{m} & \rightarrow_\epsilon & e[v/x]\ /\ \boldsymbol{m} & (2)\\
\text{if } \mathit{true} \text{ then } e \text{ else } e_1\ /\ \boldsymbol{m} & \rightarrow_\epsilon & e\ /\ \boldsymbol{m} & (3)\\
\text{if } \mathit{false} \text{ then } e \text{ else } e_1\ /\ \boldsymbol{m} & \rightarrow_\epsilon & e_1\ /\ \boldsymbol{m} & (4)\\
\textbf{while}(\boldsymbol{e})\ \boldsymbol{e}_1\ /\ \boldsymbol{m} & \rightarrow_\epsilon & \textbf{if } \boldsymbol{e} \textbf{ then } \boldsymbol{e}_1\textbf{;while}(\boldsymbol{e})\ \boldsymbol{e}_1 \textbf{ else unit}/\ \boldsymbol{m} & (5)\\
\textbf{ref } \boldsymbol{v}\ /\ \boldsymbol{m} & \rightarrow_\epsilon & \boldsymbol{l}\ /\ \boldsymbol{m} \oplus (\boldsymbol{l}, \boldsymbol{v})\ \textit{si}\ \boldsymbol{l} \notin \boldsymbol{dom}(\boldsymbol{m}) & (6)\\
\boldsymbol{l} := \boldsymbol{v}\ /\ \boldsymbol{m} & \rightarrow_\epsilon & \textbf{unit}\ /\ \boldsymbol{m} \oplus (\boldsymbol{l}, \boldsymbol{v})\ \textit{si}\ \boldsymbol{l} \in \boldsymbol{dom}(\boldsymbol{m}) & (7)\\
!\boldsymbol{l}\ /\boldsymbol{m} & \rightarrow_\epsilon & \boldsymbol{m}(\boldsymbol{l})\ /\ \boldsymbol{m}\ \textit{si}\ \boldsymbol{l} \in \boldsymbol{dom}(\boldsymbol{m}) & (8)\\
\boldsymbol{v}; \boldsymbol{e}/\ \boldsymbol{m} & \rightarrow_\epsilon & \boldsymbol{e}/\ \boldsymbol{m} & (9)
\end{array}
$$

Le type inductif des règles de réduction (1) à (4) se voit donc ajouter deux arguments m_1 et m_2, avec la contrainte $m_1 = m_2$. Le type inductif représentant les règles de réduction de Mini-ML commençant par :

```
Inductive reductionMiniML : expression → expression → Prop :=
 beta :  ∀ v,e:expression; x:identifier,
         (isvalue v) → (reductionMiniML ((fun x e) v)  e[v/x])
...
```

devient pour Ref-ML :

```
Inductive reductionRefML : memoire → memoire →
          expressionRefML → expressionRefML → Prop :=
 beta' : ∀ m1,m2:memoire; m1=m2 →
         ∀ v,e:expressionRefML; x:identifier,
         (isvalue v) → (reductionRefML m1 m2 ((fun x e) v)  e[v/x])
...
```

Cette transformation ressemble à la transformation permettant d'ajouter des arguments. La différence est qu'une contrainte sous la forme d'un argument supplémentaire de type `m1=m2` est ajouté à chaque constructeur. Si nous voulions appliquer notre méthode de réutilisation, nous définirions une transformation qui ajouterait cet argument supplémentaire à chaque constructeur. Dans le meilleur des cas, ces morceaux de terme pourront être calculés automatiquement. Sinon, nous utiliserions des métavariables afin de générer les obligations de preuve correspondantes, réclamant la preuve d'une égalité.

La transformation envisagée consiste à ajouter au type inductif un argument `m1` et une contrainte de type `m1=m2`, sur cet argument. La syntaxe d'une telle commande pourrait être la suivante :

```
Argum reductionRefML' as expressionRefML with m1:memoire
   and constraint m1=m2.
```

où `reductionRefML'` est obtenu à partir de `reductionMiniML` en ajoutant un argument `m2 : memoire`.

Remarque : Au lieu d'utiliser un argument supplémentaire de type `m1=m2`, nous aurions pu exprimer `beta'` de la manière suivante :

```
beta' : ∀ m:memoire;
        ∀ v,e:expressionRefML; x:identifier,
        (isvalue v) → (reductionRefML m m ((fun x e) v)  e[v/x])
```

La méthode de réutilisation par transformation devrait alors prendre en compte le fait que pour les anciens constructeurs les deux arguments à ajouter sont identiques. Cependant, la contrainte entre les deux arguments pourraient être différente d'une simple égalité. Par exemple, si nos deux arguments sont des entiers naturels n et m, la contrainte pourrait être $n = m + 1$, ou plus généralement $n = (f\ m)$ où f est une fonction définie auparavant. Nous aurions alors besoin d'une opération de transformation dédiée à chaque type de contrainte. Tandis qu'en exprimant la contrainte sous forme d'une preuve (d'égalité dans notre cas), il est possible de concevoir une commande générique, et non autant de commandes que de types de modifications envisagées.

Lors du passage de Mini-ML à Ref-ML, les règles de typage reçoivent également un nouvel argument correspondant au contexte de typage d'une location mémoire.

Ainsi, la propriété de *Subject-Reduction* et de sûreté du typage pour Ref-ML, ainsi que tous les lemmes qui en dépendent pourraient être montrés, modulo la preuve des obligations de preuve générées et de lemmes supplémentaires, par notre technique de réutilisation à partir des preuves faites pour Mini-ML.

La classe des modifications, qui peuvent supporter notre approche de la réutilisation, est l'ensemble des modifications d'un développement formel qui permettent en quelque sorte de "plonger" les anciennes spécifications dans les nouvelles. L'idée de plongement fait bien ressortir l'idée que l'on retrouve sous une certaine forme l'ancien terme, avec d'autres éléments en plus, comme d'autres morceaux de preuve ou bien simplement des décorations, commes des paramètres ou des arguments.

Néanmoins, vouloir réutiliser dans l'absolu n'est pas envisageable. En effet, chaque type de modification sur les spécifications induit une modification sur les termes. Ainsi pour être capable de réutiliser, il faut être en mesure de connaître avec précision les modifications effectuées sur les spécifications et d'être capable d'adapter les termes en conséquence.

Notre approche permet par conséquent de réutiliser un développement formel pour toute une classe de modifications de spécifications. Pour élargir encore le

contexte dans lequel il est permis de réutiliser, nous pourrions combiner les diverses techniques de réutilisation existantes comme celles présentées au chapitre 3, en particulier celle de Nicolas Magaud [52]. Un environnement de réutilisation beaucoup plus large pourrait ainsi être créé, dans lequel nous aurions une tactique de réutilisation prenant en paramètre la technique à utiliser.

Bibliographie

[1] Jean-Raymond Abrial. *The B-Book : Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] Penny Anderson. Representing Proof Transformations for Program Optimization. In *Proceedings of the 12th International Conference on Automated Deduction*, volume 814 of *LNAI*, pages 575–589. Springer-Verlag, June 1994.

[3] Henk P. Barendregt. Lambda Calculi with Types. *Handbook of Logic in Computer Science*, pages 117–309, 1992.

[4] Bruno Barras. *Auto-validation d'un systeme de preuves avec familles inductives*. PhD thesis, Université Paris VII, 1999.

[5] Bruno Barras et al. *The Coq Proof Assistant, Reference Manual (v7.4)*. INRIA Rocquencourt, 2003.

[6] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. Validity checking for combinations of theories with equality. In M. K. Srivas and A. Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (Palo Alto, CA)*, volume 1166, pages 187–201. Springer, 1996.

[7] Gilles Barthe. Implicit coercions in type systems. In *International Workshop TYPES'95*, volume 1158 of *LNCS*, pages 1–15. Springer, 1995.

[8] Gilles Barthe and Pierre Courtieu. Efficient reasoning about executable specifications in Coq. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 2410 of *LNCS*, pages 31–46. Spinger-Verlag, 2002.

[9] Gilles Barthe and Olivier Pons. Type Isomorphisms and Proof Reuse in Dependent Type Theory. In F. Honsell and M. Miculan, editors, *Proceedings of FOSSACS'01*, volume 2030 of *LNCS*, pages 57–71. Springer-Verlag, 2001.

[10] Gilles Barthe and Femke van Raamsdonk. Constructor Subtyping in the Calculus of Inductive Constructions. In J. Tuyrin, editor, *Proceedings of FOSSACS'00*, volume 1784 of *LNCS*, pages 17–34. Springer-Verlag, 2000.

[11] Stefan Berghofer and Tobias Nipkow. Proof terms for simply typed higher order logic. In *Theorem Proving in Higher Order Logics*, pages 38–52, 2000.

[12] Yves Bertot. A certified compiler for an imperative language. Technical Report RR-3488, INRIA, 1998.

[13] Yves Bertot, Venanzio Capretta, and Kuntal Das Barman. Type-theoretic functional sematics. In V. A. Carreno, C. A. Munoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 2410 of *LNCS*, pages 83–97. Springer, 2002. 15th International Conference, TPHOLs 2002, Hampton, VA, USA, August 20-23, 2002.

[14] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.

[15] Yves Bertot and Ranan Fraer. Reasoning with Executables Specification. In Mosses, Nielsen, and Schwartzbach, editors, *Proceedings of TAPSOFT'95*, volume 915 of *LNCS*, pages 531–545, Aarhus, Denmark, may 1995. Springer LNCS 915. Also appears as INRIA Research Report RR-2780, January 1996.

[16] Olivier Boite. Automatiser les preuves d'un sous-langage de la méthode B. *RSTI série TSI*, 21(8) :1099–1120, 2002.

[17] Olivier Boite. Proof Reuse with Extended Inductive Types. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Proceedings of TPHOLs'04*, volume 3223 of *LNCS*, pages 50–65, Park City (Utah), September 2004. Springer.

[18] Olivier Boite and Catherine Dubois. Proving Type Soundness of a Simply Typed ML-like Language with References. In R. Boulton and P. Jackson, editors, *Supplemental Proceedings of TPHOL'01, Informatics Research Report EDI-INF-RR-0046 of University of Edinburgh*, pages 69–84, 2001.

[19] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR : the system. In *Proceedings of SIGSOFT'88, Third Annual Symposium on Software Development Environments (SDE3)*, Boston, USA, 1988.

[20] Samuel Boutin. Preuve de correction de la compilation de mini-ml en code cam dans le syst'eme d'aide 'a la d'emonstration coq, 1995.

[21] Ana Bove. A Machine Assisted Proof that Well-Typed Expressions Cannot Go Wrong. *Technical report, Department of Computing Science, Chalmers University of Technology, Göteborg, Sweden.*, May 1998.

[22] Paul Callaghan. Coherence checking of coercions in plastic. In *From : Workshop on Subtyping and Dependent Types in Programming*, Ponte de Lima, July 2000.

[23] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[24] Thierry Coquand. An analysis of girard's paradox. In *In Symposium on Logic in Computer Science*, Cambridge, 1986. IEEE Computer Society Press.

[25] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76 :95–120, February 1988.

[26] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and editors G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *LNCS*. Springer-Verlag, 1990.

[27] Christina Cornes and Delphine Terrasse. Automatizing inversion of inductive predicates in coq. In *Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[28] Luis Damas and Robin Milner. Principal Type Schemes for Functional Programs. *ACM Symp. on Principles of Programming Languages (POPL)*, 1982.

[29] Daniel de Rauglaudre. Camlp4 - Reference Manual version 3.04. INRIA-Rocquencourt. `http://caml.inria.fr/camlp4/manual/`, December 2001.

[30] David Delahaye. Information Retrieval in a Coq Proof Library using Type Isomorphisms, 1999.

[31] David Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island*, volume 1955, pages 85–95. Springer-Verlag LNCS/LNAI, November 2000.

[32] David Delahaye, Roberto Di Cosmo, and Benjamin Werner. Recherche dans une bibliothèque de preuves Coq en utilisant le type et modulo isomorphismes. In *PRC/GDR de programmation, Pôle Preuves et Spécifications Algébriques*, Novembre 1997.

[33] Damien Doligez. Zenon, a first-order prover with coq-checkable output. 2nd Workshop on Coq and Rewriting, September 2004.

[34] Catherine Dubois. Proving ML Type Soundness Within Coq. *Theorem Proving in Higher Order Logics : 13th International Conference, TPHOLs 2000, Lecture Notes in Computer Science*, 1869 :Springer–Verlag, 126–144, 2000.

[35] Catherine Dubois, Jérôme Grandguillot, and Mathieu Jaume. Réutilisation de preuves formelles : Une étude pour le système FoC. In INRIA, editor, *14ème Journées Francophones des Langages Applicatifs, JFLA'2003*, pages 63–75, 2003.

[36] Amy P. Felty and Douglas J. Howe. Generalization and Reuse of Tactic Proofs. In Frank Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 1–15, Kiev, Ukraine, 1994. Springer-Verlag LNAI 822.

[37] Jean-Christophe Filliâtre. Why : a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[38] Robert W. Floyd. Assigning meaning to programs. In *Symposium on Applied Mathematics*, volume 1, pages 19–32. American Mathematical Society, 1967.

[39] Simon J. Gay. A Framework for the Formalisation of Pi Calculus Type Systems. In R.J. Boulton and P.B. Jackson, editors, *14th International Conference,*

TPHOL'01, volume 2152 of *LNCS*, pages 217–232, Edinburgh, Scotland, UK, September 2001. Springer.

[40] Fausto Giunchiglia and Adolfo Villafiorita. Absfol : a proof checker with abstraction. In Mcrobbie M. A. and Slaney J. K., editors, *13th international conference on automated deduction (CADE-13)*, New Brunswick, N.J., July 30 - August 3 1996.

[41] Michael J. C. Gordon and Thomas F. Melham, editors. *Introduction to HOL : A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[42] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–583, October 1969.

[43] Dieter Hutter and Claus Sengler. INKA : The next generation. In *Conference on Automated Deduction*, pages 288–292, 1996.

[44] Mathieu Jaume. Unification : a Case Study in Transposition of Formal Properties. In E.L. Gunter and A. Felty, editors, *Supplementary Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics : Poster session TPHOLs'97*, pages 79–93, Murray Hill, N.J., 1997.

[45] Einar Broch Johnsen and Christoph Lüth. Theorem Reuse by Proof Term Transformation. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Proceedings of TPHOLs 2004*, volume 3223 of *LNCS*, pages 152–167. Springer, 2004.

[46] Gilles Kahn. Natural Semantics. *In Lecture Notes in Computer Science*, 247 :22–39. Springer, 1987.

[47] Thomas Kolbe and Jurgen Brauburger. Plagiator - A Learning Prover. In *Conference on Automated Deduction*, pages 256–259, 1997.

[48] Michael Y. Levin and Benjamin C. Pierce. Tinkertype : A language for playing with formal systems. Technical Report MS-CIS-99-19, Dept of CIS, University of Pennsylvania, July 1999.

[49] Zhaohui Luo. *Computation and Reasoning : A Type Theory for Computer Science*. Oxford University Press, 1994.

[50] Zhaohui Luo and Randy Pollack. LEGO proof development system : User's manual. Technical Report ECS-LFCS-92-211, LFCS, The King's Buildings, Edinburgh EH9 3JZ, 1992.

[51] Peter Madden. The Specialization and Transformation of Constructive Existence Proofs. In *Proceeding of the Eleventh International Joint Conference on Artificial Inteligence*, pages 131–148. Morgan Kaufmann, 1989.

[52] Nicolas Magaud. Changing Data Representation within the Coq System. In *Proceedings of TPHOLs'03*, pages 87–102. Springer-Verlag LNCS 2758, September 2003.

[53] Nicolas Magaud and Yves Bertot. Changing data structures in type theory : a study of natural numbers. In Paul Callaghan, Zhaohui Luo, James McKinna, and Randy Pollack, editors, *TYPES'2000*, volume 2277. LNCS, Springer-Verlag, 2000. Springer-Verlag.

[54] Nicolas Magaud and Yves Bertot. Changement de Représentation des Structures de Données en Coq : le cas des entiers naturels. In *Proceedings of JFLA'2001*, 2001.

[55] Lena Magnusson. *The Implementation of ALF : A Proof Editor Based on Martin-Lof's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Chalmers University of Technology and Goteborg University, Jan. 1995.

[56] Lena Magnusson and Bengt Nordström. The alf proof editor and its proof engine. *Types for Proofs and Programs, Nijmegen*, LNCS 806 :213–237, 1994.

[57] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2) :89–106, 2004.

[58] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In J. T. Schwartz, editor, *Proceedings Symposium in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science*, pages 33–41. American Mathematical Society, Providence, RI, 1967.

[59] Erica Melis and Jon Whittle. Internal analogy in theorem proving. In *Conference on Automated Deduction*, pages 92–105, 1996.

[60] Erica Melis and Jon Whittle. Analogy in inductive theorem proving. *Journal of Automated Reasoning*, 22(2) :117–147, 1999.

[61] Robin Milner and Richard Weyhrauch. Proving compiler correctness in a mechanized logic. In *Machine Intelligence 7*, pages 51–71. John Wiley and Sons, New York, 1972.

[62] César Muñoz. Proof Representation in Type Theory : State of the Art. In *Proc. XXII Latinamerican Conference of Informatics CLEI Panel 96*, Santafé de Bogotá, Colombia, June 1996.

[63] César Muñoz. Proof-term synthesis on dependent-type systems via explicit substitutions. volume 266, pages 407–440, 2001. Also appears as report NASA/CR-1999-209730 ICASE No. 99-47.

[64] Julien Narboux. A Decision Procedure for Geometry in Coq. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Proceedings of TPHOLs'04*, volume 3223 of *LNCS*, pages 225–240, Park City (Utah), September 2004. Springer.

[65] Tobias Nipkow. Winskel is (almost) Right : Towards a Mechanized Semantics. *Formal Aspects of Computing*, 10(2) :171–186, 1998.

[66] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. Java : Embedding a Programming Language in a Theorem Prover. *In F.L. Bauer and R. Steinbruggen, editors, Foundations of Secure Computation. IOS Press*, 2000.

[67] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[68] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory : An Introduction*. Oxford University Press, 1990.

[69] Christine Paulin-Mohring. Inductive Definitions in the System Coq - Rules a nd Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, 1993. LIP research report 92-49.

[70] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[71] Lawrence C. Paulson. *Logic and Computation : Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.

[72] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2000.

[73] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, Denmark, 1981.

[74] Erik Poll. Subtyping and Inheritance for Inductive Types. *In Proceedings of TYPES'97 Workshop on Subtyping, inheritance and modular development of proofs*, September 1997.

[75] Olivier Pons. *Conception et réalisation d'outils d'aide au développement de grosses théories dans les systèmes de preuves interactifs*. PhD thesis, CNAM, 1999.

[76] Olivier Pons. Generalization in type theory based proof assistants. In *International Workshop on Types for Proofs and Programs, TYPES 2000, Durham, UK, December 8-12*, volume 2277 of *LNCS*, pages 217–232. Springer-Verlag, 2002.

[77] Olivier Pons, Yves Bertot, and Laurence Rideau. Notions of dependency in proof assistants. In *Electronic Proceedings of User Interfaces for Theorem Provers*, Sophia-Antipolis, France, 1998.

[78] Virgile Prevosto and Damien Doligez. Inheritance of algorithms and proofs in the computer algebra library foc. In *Journal of Automated Reasoning*, volume 29(3–4), pages 337–363, 2002. Special Issue on Mechanising and Automating Mathematics, In Honor of N.G de Bruijn.

[79] Virgile Prevosto, Damien Doligez, and Thérèse Hardin. Algebraic Structures and Dependent Records. In Sofiène Tahar César Muñoz and Víctor Carreño, editors, *Proceedings of TPHOLs 02*, Hampton, Virginia, USA, August 2002. Springer-Verlag.

[80] John C. Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development*, volume 185 of *LNCS*, pages 97–138. Springer Verlag, Mars 1985.

[81] Julian D.C. Richardson. Automating Changes of Data Type in Functional Programs. In *Proceedings of the 10th Conference on Knowledge-Based Software Engineering (KBSE)*, November 1995.

[82] Mikael Rittri. Using types as search keys in function libraries. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 174–183. ACM Press, 1989.

[83] Eugene J. Rollins and Jeannette M. Wing. Specifications as search keys for software libraries. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 173–187, Paris, France, 1991. The MIT Press.

[84] Joseph Rouyer. Développement de l'algorithme d'unification dans le calcul des constructions avec types inductifs. Technical Report RR-1795, INRIA-Lorraine, Novembre 1992.

[85] Amokrane Saïbi. Typing Algorithm in Type Theory with Inheritance. In *Symposium on POPL'97*, pages 292–301. ACM Press, January 1997.

[86] Axel Schairer and Dieter Hutter. Proof transformations for evolutionary formal software development. In *Algebraic Methodology And Software Technology (AMAST)*, volume 2422 of *LNCS*, pages 441–456. Springer-Verlag, 2002.

[87] Natarajan Shankar, Sam Owre, John M. Rushby, and David W. J. Stringer-Calvert. PVS prover guide - version 2.2, September 1999.

[88] Graph Visualization Software. `http://www.graphviz.org/`.

[89] Don Syme. Proving Java Type Sound. *In Jim Alves-Foss, editor, Formal Syntax and Semantics of Java, volume 1523 of LNCS. Springer*, 1999.

[90] Don Syme and Andy D. Gordon. Automating type soundness proofs via decision procedures and guided reductions. In *Proceedings of the 9th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, Tbilisi, Georgia., October 14-18 2002.

[91] Delphine Terrasse. Encoding natural semantics in Coq. In V. S. Alagar, editor, *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology*, pages 230–244, Montreal, Canada, 1995. Springer-Verlag LNCS 936.

[92] Myra VanInwegen. Towards Type Preservation in Core SML. *Technical report, Cambridge University*, 1997.

[93] David von Oheimb and Tobias Nipkow. Machine-checking the java specication : Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in LNCS, pages 119–156. Springer Verlag, 1999.

[94] Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris VII, Mai 1994.

[95] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach To Type Soundness. *Information and computation*, 115(1) :38–94, 1994.

Une aide à la réutilisation de preuves formelles
Application aux preuves de propriétés sémantiques

Résumé

Les preuves formelles sont longues et délicates. Puisque l'automatisation n'est pas toujours possible, il est capital de pouvoir réutiliser les preuves, même dans un contexte d'application légèrement différent. Nous proposons un ensemble de commandes élémentaires permettant la modification de spécifications inductives d'un développement formel, et permettant la réutilisation des preuves associées. Des obligations de preuve sont générées lorsque les preuves à réutiliser nécessitent d'être complétées. Un prototype implémente cet environnement dans l'assistant à la preuve Coq. Un outil de génération de graphe de dépendances permet également de prévoir les modifications à faire lors d'une réutilisation. A chaque commande est associée une méthode de réutilisation automatique, prouvée correcte vis à vis du typage.

Mots-clés

Réutilisation de preuves formelles, calcul de dépendances, systèmes d'aide à la preuve, assistant à la preuve Coq.

A help with the formal proof reuse
Application to semantic properties proof

Abstract

Formal proofs are long and difficult. As automatisation is not always possible, it is very usefull to reuse proofs, even in a different context. We propose in this PhD a set of elementary commands allowing to modify inductive specifications of a formal development, and allowing to reuse associated proofs. Proof obligations are generated when reused proof need to be completed. A prototype implements this environment in the Coq proof assistant. A dependancy graph allows to foresee the modifications when reusing. Each command is associated to an automatic reuse method which are proved sound in relation to the typing.

Keywords

Formal proof reuse, dependencies calculus, proof checker, Coq proof assistant.