Compilation JIT des termes de preuve

Quentin CARBONNEAUX
Encadré par Mathieu BOESPFLUG et Olivier HERMANT

1er septembre 2012

- **Le contexte général**
  Devant le besoin croissant d'interopérabilité entre les différents systèmes de preuves formelles il devient nécessaire de choisir un fragment logique commun. De plus, cela permet de factoriser la base de confiance qui est alors réduite à un unique vérificateur de types (et éventuellement, plusieurs traducteurs).
  Les preuves par réflexion se font de plus en plus communes et il faut pouvoir les vérifier à une vitesse raisonnable, l'ajout de calcul dans le raisonnement rend la vérification de preuve plus difficile, de nouvelles techniques doivent être développées.

- **Le problème étudié**
  Nous cherchons à développer un vérificateur de types rapide pour le $\lambda\Pi$-calcul modulo tout en respectant une certaine séparation des tâches : nous cherchons à utiliser au maximum le potentiel des langages fonctionnels pour réaliser les parties calculatoires des preuves sans avoir à écrire un interpréteur optimisé.

- **La contribution proposée**
  Nous proposons un nouvel algorithme de vérification de types. Il résulte de la combinaison des techniques hors contexte développées dans la thèse de Boespflug [6] et des algorithmes bidirectionnels que l'on peut trouver chez Coquand [10] et Abel et Altenkirch [1]. Cette modification de l'algorithme permet d'avoir des termes de preuve plus courts.
  Nous prouvons sa correction et en donnons une implémentation entièrement nouvelle, courte, simple et digne de confiance : DEDUKTI. Cette implémentation utilise un compilateur JIT pour être flexible quant à la quantité de calcul dans les preuves que nous vérifions.

- **Les arguments en faveur de sa validité**
  Les tests de performance que nous utilisons sont la bibliothèque standard de COQ et celle de HOL, des preuves reconnues et réalistes. Nous arrivons à nous comparer avec des résultats satisfaisants à des implémentations de référence. Dans le cas des preuves calculatoires, nous espérons de très bonnes performances, ces espérances se basent sur des micro tests qui révèlent une supériorité du test de conversion implémenté par compilation JIT par rapport aux approches basées sur des techniques d'interprétation.

- **Le bilan et ses perspectives**
  Les résultats sur les preuves disponibles sont satisfaisants mais nous souhaitons les étendre à des preuves plus calculatoires comme celle du théorème des quatre couleurs. Il reste également à prouver que l'interaction entre bidirectionnel et hors contexte donne un système correct, toutefois, des preuves existantes nous poussent à croire en la validité de l'approche.

Table des matières

1 Introduction

Le succès des méthodes de démonstrations formelles, qui trouvent leurs origines dans des systèmes tels qu'AUTOMATH et LCF a donné naissance à une variété impressionnante d'outils supportant ces techniques de raisonnement. Malgré des succès académiques et quelques percées dans des domaines industriels tels que l'aérospatiale ou le transport de masse, la puissance de ces méthodes reste largement sous-exploitée. L'une des explications à ce constat peut être le manque de standards et les difficultés à faire interagir plusieurs systèmes logiques ensemble.

Pour pallier ce manque de standards et avancer une solution au problème de l'échange de preuves, l'approche DEDUKTI propose un format universel d'expression des théories et des théorèmes. Le problème de l'échange de preuve est sujet à controverse, en effet, les systèmes logiques existants possèdent un très grand nombre de caractéristiques qui les rendent uniques et parfois même incompatibles : certains sont prédicatifs, d'autres non, certains sont constructifs, d'autres non, certains sont du premier ordre, d'autres non. Nous prétendons que, malgré ces subtilités, le λΠ-calcul modulo, une extension minimale mais très expressive du λΠ-calcul permet assez de souplesse pour devenir ce langage de preuve universel.

Le cadre logique fourni par le λΠ-calcul modulo permet l'expression de théories par plongement *superficiel*, à mettre en contraste avec les plongements *profonds* [1] qui sont le standard dans des systèmes comme Edinburgh Logical Framework, TWELF ou ISABELLE. Les plongements profonds utilisent une logique minimale et font du système à encoder l'objet du discours en traitant ses preuves et ses théorèmes comme des données. Lorsque des *Systèmes de Types Purs* (PTS) qui contiennent la *règle de conversion* (calcul des constructions, LF) :

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \equiv B}{\Gamma \vdash M : B}$$

1. Shallow et deep embeddings dans la littérature anglaise.

sont exprimés en plongement profond, les applications de cette règle sont traduites par l'exhibition complète de la preuve que A et B sont convertibles. Les propriétés calculatoires des théorèmes (ici A et B) sont perdues et il faut expliciter le calcul qui était implicite dans le système source. Le problème que l'on rencontre vient du fait que la notion d'égalité dans le méta langage n'est pas extensible. A contrario, le $\lambda\Pi$-calcul modulo étend le $\lambda\Pi$-calcul en permettant l'extension de la règle de conversion par un ensemble $\mathscr{R}$ de règles de réécriture défini par l'utilisateur. Intuitivement, cette nouvelle possibilité permet d'intégrer l'égalité du système à encoder dans le $\lambda\Pi$-calcul modulo et donc de préserver les propriétés calculatoires des théorèmes.

Le $\lambda\Pi$-calcul modulo, qui est exposé dans la section 2.2, propose un cadre minimal pour l'expression des théorèmes. Au contraire des systèmes LCF qui déclarent un type abstrait des théorèmes, cette approche repose sur la correspondance entre programmes et preuves et transpose le problème de vérification de preuve en un problème de typage d'un programme fonctionnel. Le noyau d'un tel système doit être digne de la plus grande confiance car la cohérence du cadre logique repose entièrement sur sa correction. Les techniques décrites dans ce rapport (systèmes de types bidirectionnels, utilisation des fermetures, conversion par évaluation) permettent une implantation très proche du cadre théorique d'un noyau de vérification du $\lambda\Pi$-calcul modulo : DEDUKTI ("déduire" en Espéranto).

La thèse de Boespflug [6] décrit l'implémentation initiale de DEDUKTI. Pendant cinq mois j'ai changé l'algorithme de vérification de types anciennement utilisé pour un système bidirectionnel décrit dans la section 3.3.1 que j'ai implémenté. Ce système garde l'esprit hors contexte de l'algorithme tel que décrit dans la thèse mais permet d'avoir des termes de preuve plus courts grâce à l'exploitation des techniques de typage bidirectionnel. Il est décrit extensivement dans l'article [8] que nous avons écrit avec Boespflug et Hermant et qui a été publié dans les proceedings du workshop PxTP. La section 3.2.1 présente une preuve de correction d'un algorithme bidirectionnel avec contextes explicites, c'est la première étape pour montrer que le calcul implémenté par DEDUKTI est correct. La section 3.3.1 finit montrer ce résultat en prouvant que le calcul hors contexte bidirectionnel présenté est correct par rapport au calcul bidirectionnel simple.

La première implémentation en Haskell, bien que très concise et proche du modèle, se comportait de façon dégénérée lorsque les fichiers à vérifier dépassaient quelques centaines de kilo octets ; j'ai donc implémenté en repartant de zéro une nouvelle version de DEDUKTI. Ce nouveau programme en C est plus rapide de plusieurs ordres de grandeurs sur les exemples où les deux programmes réussissent. Afin de permettre plus de flexibilité, le générateur de code a été refait à neuf pour générer du Lua au lieu de générer du Haskell. De fait, le langage Lua possède un compilateur "juste à temps" comme décrit dans la section 5.1.2 ; cela permet de gagner encore un ordre de grandeur sur l'ancienne implémentation dans la majeure partie des essais que nous avons réalisé. Le changement de langage cible m'a également permis d'ajouter plus facilement des optimisations qui n'étaient pas présentes dans l'ancienne version, elles sont décrites dans la section 5.1.3.

Comme le nouveau DEDUKTI permet des fonctionnalités plus fines pour définir des règles de réécriture (*c.f.* section 5.1.4), j'ai aussi consacré des efforts non négligeables à rendre compatible le traducteur COQINE pour que sa sortie puisse être vérifiée.

Finalement, la section 5.2 décrit le comportement de notre vérificateur de type sur des tests réalistes : la bibliothèque standard de HOL ainsi que celle de COQ sont partiellement vérifiées et la performance de DEDUKTI est comparée à celle d'implémentations de référence.

2 Le $\lambda\Pi$-calcul modulo

Le $\lambda\Pi$-calcul modulo est une extension du $\lambda\Pi$-calcul qui libéralise la règle de conversion. Cette modification mineure permet d'obtenir un gain d'expressivité majeur, on peut par exemple simuler un système avec polymorphisme en utilisant seulement des règles de réécriture. On peut également encoder un grand nombre de théories comme le Système F, le Calcul des Constructions, et de manière plus générale tous les Systèmes de Types Purs (PTS) fonctionnels dans le $\lambda\Pi$-calcul modulo.

2.1 Le $\lambda\Pi$-calcul

Le $\lambda\Pi$-calcul est un calcul minimal avec types dépendants. Il peut être vu par les correspondances de Curry-de Bruijn-Howard et Brouwer-Heiting-Kolmogorov comme le langage qui exprime les preuves de la logique minimale des prédicats.

Le $\lambda\Pi$-calcul forme un des sommets du cube de Barendregt, on l'obtient en ajoutant au λ-calcul simplement typé des types dépendants. L'exemple commun de type dépendant est celui des tableaux dans les langages de

$\boxed{\Gamma \text{ wf}}$ Le contexte Γ est bien formé

$$(\textit{empty})\ \frac{}{\cdot \text{ wf}} \qquad (\textit{decl})\ \frac{\Gamma \text{ wf} \qquad \Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x : A \text{ wf}}\ s \in \{\mathbf{Type}, \mathbf{Kind}\}$$

$\boxed{\Gamma \vdash M : A}$ Le terme M a pour type A dans le contexte Γ

$$(\textit{sort})\ \frac{\Gamma \text{ wf}}{\Gamma \vdash \mathbf{Type} : \mathbf{Kind}} \qquad (\textit{var})\ \frac{\Gamma \text{ wf} \qquad x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$(\textit{prod})\ \frac{\Gamma \vdash A : \mathbf{Type} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A.\, B : s}\ s \in \{\mathbf{Type}, \mathbf{Kind}\}$$

$$(\textit{abs})\ \frac{\Gamma \vdash A : \mathbf{Type} \qquad \Gamma, x : A \vdash B : s \qquad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : \Pi x : A.\, B}\ s \in \{\mathbf{Type}, \mathbf{Kind}\}$$

$$(\textit{app})\ \frac{\Gamma \vdash M : \Pi x : A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : \{N/x\}B}$$

$$(\textit{conv})\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \equiv_\beta B}{\Gamma \vdash M : B}$$

FIGURE 1 – Règles de typage pour le $\lambda\Pi$-calcul

programmation. Bien souvent ce type est indexé par la taille des tableaux, il n'y a donc pas un unique type array mais une famille de types : array 0, array 1, array 2, etc. pour les tableaux de taille, 0, 1, 2, etc. La famille de types array peut donc être vue comme une fonction qui associe un type à un entier naturel, on écrit array : nat → **Type** pour le signifier.

Définition 1 (Pré termes du $\lambda\Pi$-calcul)**.** L'ensemble $\mathscr{T}$ des pré termes du $\lambda\Pi$-calcul est construit inductivement comme suit, on note $\mathscr{V}$ un ensemble de variables dénombrable :
- $\mathscr{V} \subset \mathscr{T}$
- $\mathbf{Type}, \mathbf{Kind} \in \mathscr{T}$
- Si $u, v, A, B \in \mathscr{T}$ et $x \in \mathscr{V}$ alors $\lambda x.\, u, \Pi x : A.\, B, u\ v \in \mathscr{T}$.

On peut par récurrence associer à tout terme M son ensemble de variables libres $\mathscr{FV}(M)$—les deux seuls lieurs du langage sont les symboles λ et Π. Dans toute la suite on suivra les conventions de Barendregt. En particulier, on considère que les pré termes α-équivalents sont identiques ; ainsi on identifie $\lambda x.\, x$ et $\lambda y.\, y$.

Définition 2 (Contextes)**.** L'ensemble des contextes de typage $\mathscr{C}$ est défini inductivement par les deux cas suivants :
- $. \in \mathscr{C}$
- Si $\Gamma \in \mathscr{C}$, $x \in \mathscr{V}$ et $A \in \mathscr{T}$ alors $\Gamma, x : A \in \mathscr{C}$.

Les pré termes du $\lambda\Pi$-calcul sont des expressions qui peuvent être mal typées tandis que les termes du $\lambda\Pi$-calcul doivent respecter des règles de typage. Plus formellement, un pré terme M est un *terme* si il est une *sorte* (**Type** ou **Kind**) ou bien si il existe un contexte Γ et un terme A tel que le *jugement* $\Gamma \vdash M : A$ soit dans l'ensemble inductivement défini par les règles de la figure 1.

2.2 Preuves modulo

Dans le calcul décrit précédemment, la règle de conversion permet de raisonner sur les types modulo la β-conversion [4] ; c'est l'unique procédure de calcul disponible. Cependant, il peut être souhaitable de vouloir donner un comportement calculatoire à une constante lorsque l'on encode un théorie dans le $\lambda\Pi$-calcul. Un exemple d'une telle situation survient lors de l'encodage des paires du Système T de Gödel. Dans ce calcul, on

spécifie le comportement des projections des paires (π_1 et π_2) avec des règles de réécriture du type

$$\pi_1\,(a,b) \longrightarrow a \qquad \pi_2\,(a,b) \longrightarrow b.$$

L'opérateur de point fixe sur les entiers naturels du Système T (R) peut également être exprimé par des règles de réécriture telles que

$$\mathsf{R}\,f\,z\,(\mathsf{S}\,n) \longrightarrow f\,(\mathsf{R}\,f\,z)\,n \qquad \mathsf{R}\,f\,z\,\mathsf{Z} \longrightarrow z.$$

Au lieu d'incorporer tous ces comportements calculatoires comme des cas particuliers d'une théorie plus générale des types inductifs (comme peut le faire le Calcul des Constructions Inductives), la déduction modulo [12] propose un formalisme dans lequel les théorèmes peuvent être prouvés modulo un ensemble de règles de réécriture.

Le $\lambda\Pi$-calcul modulo, qui est le langage dans lequel les preuves de la déduction modulo peuvent être exprimées par l'isomorphisme de Curry-de Bruijn-Howard incorpore cette possibilité de raisonner modulo en relaxant la règle de conversion du $\lambda\Pi$-calcul. Cette règle devient

$$(conv)\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \equiv_{\beta R} B}{\Gamma \vdash M : B}$$

où R est un ensemble de règles de réécriture qu'il est possible de choisir.

Le $\lambda\Pi$-calcul modulo, bien qu'étant une modification simple du $\lambda\Pi$-calcul se révèle être très expressif. Dowek et Cousineau [11] montrent qu'il permet d'encoder tous les autres PTS fonctionnels sans utiliser d'axiomes.

3 Vérification de types algorithmique

Avec la montée en puissance des systèmes logiques, le travail du vérificateur de preuves se fait de plus en plus difficile. Les relations de typage ne sont pas toujours décidables et dans le cas de la déduction modulo, la convertibilité de deux termes peut aussi être indécidable.

Dans ce chapitre nous présentons une relation de typage pour le $\lambda\Pi$-calcul modulo qui est algorithmique : il est possible d'extraire des règles de typage de manière simple un programme qui permet de vérifier qu'un terme du $\lambda\Pi$-calcul modulo est bien une preuve d'un théorème donné.

3.1 Le λ-calcul simplement typé

Dans le cas du λ-calcul simplement typé à la Church, il est assez simple d'extraire un algorithme de typage des règles de typage du calcul. Le programme Haskell suivant implémente cet algorithme.

```
typof (Γ,y:A) x = if y == x then A else typof Γ x
typof Γ (u v) =
    case typof Γ u of
      (σ → τ) | typof Γ v == σ -> τ
      _ -> throw TypeError
typof Γ (λx:σ.t) = σ → (typof (Γ,x:σ) t)
typof _ _ = throw TypeError
```

Cet exemple naïf montre comment à partir d'un ensemble de règles d'inférence on peut dériver un algorithme de synthèse de types. Mais dans le cas général il est plus compliqué d'effectuer cette dérivation car certaines règles peuvent être appliquées à tout instant, on dit que le système n'est pas dirigé par la syntaxe. Dans le cas du $\lambda\Pi$-calcul, la règle de conversion en est l'exemple. Toutefois, si un PTS est normalisant et fonctionnel, la vérification de types devient décidable [17].

La présentation que nous avons faite du typage du $\lambda\Pi$-calcul n'est pas algorithmique : son problème principal est que le calcul est présenté sans annotations sur les λ—c'est un présentation à la Curry. Cette présentation rend la règle *abs* impossible à implémenter car il faut "deviner" le type de la variable du λ, de plus plusieurs choix peuvent être valides pour ce type. La règle *conv* pose également un problème : on peut l'appliquer à tout instant dans la dérivation.

Toutefois, dans l'hypothèse où notre système a des annotations de type sur les λ, on peut, malgré la règle *conv*, extraire un algorithme de l'ensemble de règles de la figure 1 : il suffit de systématiquement mettre les types en forme normale de tête faible avant de les inspecter. Ce même algorithme traitera tout aussi bien le

$\lambda\Pi$-calcul modulo dans l'hypothèse où le système défini par $\beta + R$ est fortement normalisant. Nous avons donc une première version d'un algorithme de vérification, mais ce dernier ne fonctionne pas sur les termes comme nous les avons défini sans annotations pour le domaine des fonctions.

3.2 Systèmes bidirectionnels

L'algorithme exposé dans le paragraphe précédent, en plus de ne pas fonctionner sur les termes à la Curry, présente quelques redondances : par exemple, dans le traitement de l'application $(u\ v)$, les types de u et v sont *synthétisés* alors qu'il suffirait de *vérifier* que v a bien le type σ si v a synthétisé le type $\sigma \to \tau$. Nous présentons dans cette section une version du $\lambda\Pi$-calcul modulo définie par un système de types bidirectionnel et qui est correcte par rapport à celle présentée dans la section 2.2.

La distinction de phases entre vérification et synthèse est l'idée à la source des systèmes dits *bidirectionnels*, on peut en trouver une description dans [10]. Ces systèmes permettent d'avoir des termes de preuve concis car grâce à l'alternance des deux phases, les annotations de type sur les abstractions (λ) ne sont pas nécessaires.

Avoir des termes plus concis est un avantage de taille lorsque l'on travaille avec des preuves générées par une machine. Comme les systèmes bidirectionnels ne requièrent pas plus de calcul pour vérifier un terme que les systèmes classiques, ils sont très utilisés dans les vérificateurs de preuve et DEDUKTI en implémente un.

3.2.1 Le $\lambda\Pi^b$-modulo

Le $\lambda\Pi^b$-modulo est une version du $\lambda\Pi$-calcul modulo avec algorithme de typage bidirectionnel. La syntaxe des pré termes de ce calcul est exactement celle qui a été donnée pour le $\lambda\Pi$-calcul.

Les contextes du $\lambda\Pi^b$-modulo sont définis de la même manière que ceux du $\lambda\Pi$-calcul modulo. On dira que Γ *étend* Σ ou que Σ est un *préfixe* de Γ lorsque $\Gamma = \Sigma$ ou qu'il existe un contexte Δ dont Σ est un préfixe, $x \in \mathcal{V}$ et $A \in \mathcal{T}$ tels que $\Gamma = \Delta, x : A$.

Les termes du $\lambda\Pi^b$-modulo sont également définis de la même manière que ceux du $\lambda\Pi$-calcul modulo mais à la place d'imposer que le jugement $\Gamma \vdash M : A$ soit valide, c'est le jugement $\Gamma \vdash M \Leftarrow A$ qui doit l'être et A doit être un *type* du $\lambda\Pi^b$-modulo, *i.e.* il doit soit être une sorte, soit vérifier $\Gamma \vdash A \Leftarrow s$ où s est une sorte.

Les règles de la figure 2 décrivent la relation de typage du $\lambda\Pi^b$-modulo. Elles font usage de la relation $\longrightarrow_w^*$ qui est la clôture réflexive transitive de la relation de réduction de tête faible associée à la β-réduction et aux règles de R. Si $\Gamma \vdash M \Leftrightarrow A$ est un jugement ($\Leftrightarrow$ est soit $\Leftarrow$ soit $\Rightarrow$), on dira que M est en position de *sujet* et que A est en position de *classifieur*.

Définition 3 (Réduction de tête faible)**.** Si $\longrightarrow$ est une relation de réduction sur les pré termes, on définit la réduction de tête faible $\longrightarrow_w$ comme la clôture de la relation $\longrightarrow$ sous les contextes définis par la grammaire suivante :

$$C ::= [\,] \mid C\ M \mid M\ C$$

où M est un terme quelconque.

Dans les règles de la figure 2, les dérivations de bonne formation des contextes ne sont pas insérées aux feuilles de la dérivation de typage comme dans la présentation standard du $\lambda\Pi$-calcul. Au contraire, nous faisons le choix de présenter un système le plus proche possible de l'implémentation. Lors de la programmation d'un algorithme de vérification de types le contexte est fabriqué de façon progressive et la bonne formation de chacun des types insérés est vérifiée. Aussi, la majorité les théorèmes énoncés dans cette section seront gardés par une hypothèse du type $\Gamma\ \mathsf{wf}_b$ où la relation wf_b est définie inductivement par la figure 3.

Le $\lambda\Pi^b$-modulo vérifie quelques propriétés classiques pour les systèmes de types énoncées dans les lemmes suivants. Cependant, on peut remarquer que ce système n'est pas clos par réduction et qu'il est donc impossible de montrer un théorème de réduction du sujet dans notre version du $\lambda\Pi$-calcul modulo bidirectionnel. Il faudrait ajouter au calcul une construction dite de coercion $(T : A)$ avec la règle de typage suivante

$$\frac{\Gamma \vdash T \Leftarrow A}{\Gamma \vdash (T : A) \Rightarrow A}$$

pour pouvoir avoir un théorème de réduction du sujet. Néanmoins, ce n'est pas un problème car nous donnons une présentation algorithmique du typage, si une telle propriété est désirable on peut se ramener au $\lambda\Pi$-calcul modulo classique par le théorème de correction présenté plus loin.

$\boxed{\Gamma \vdash M \Rightarrow A}$ Le terme M synthétise le type A dans le contexte Γ

$$(sort)\ \frac{}{\Gamma \vdash \mathbf{Type} \Rightarrow \mathbf{Kind}} \qquad (var)\ \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A}$$

$$(app)\ \frac{\Gamma \vdash M \Rightarrow C \qquad C \longrightarrow_w^* \Pi x : A.\, B \qquad \Gamma \vdash N \Leftarrow A}{\Gamma \vdash M\ N \Rightarrow \{N/x\}B}$$

$\boxed{\Gamma \vdash M \Leftarrow A}$ Le terme M est vérifié contre le type A dans le contexte Γ

$$(prod)\ \frac{\Gamma \vdash A \Leftarrow \mathbf{Type} \qquad \Gamma, x : A \vdash B \Leftarrow s}{\Gamma \vdash \Pi x : A.\, B \Leftarrow s}\ s \in \{\mathbf{Type}, \mathbf{Kind}\}$$

$$(abs)\ \frac{C \longrightarrow_w^* \Pi x : A.\, B \qquad \Gamma, x : A \vdash M \Leftarrow B}{\Gamma \vdash \lambda x.\, M \Leftarrow C}$$

$$(conv)\ \frac{\Gamma \vdash M \Rightarrow A \qquad A \equiv_{\beta R} B}{\Gamma \vdash M \Leftarrow B}$$

FIGURE 2 – Règles de typage pour le $\lambda\Pi^b$-modulo

$$(empty)\ \frac{}{\cdot\ \mathsf{wf}_b} \qquad (decl)\ \frac{\Gamma\ \mathsf{wf}_b \qquad \Gamma \vdash A \Leftarrow s \qquad x \notin \Gamma}{\Gamma, x : A\ \mathsf{wf}_b}\ s \in \{\mathbf{Type}, \mathbf{Kind}\}$$

FIGURE 3 – Règles de bonne formation pour les contextes du $\lambda\Pi^b$-modulo

Lemme 1 (Affaiblissement)**.** Si $\Gamma \vdash M \Leftarrow A$, et si Γ est un préfixe de Σ un contexte bien formé, alors $\Sigma \vdash M \Leftarrow A$.

Lemme 2 (Substitution)**.** On a les deux propriétés suivantes.

i. Si $\Gamma, x : A \vdash M \Rightarrow B$ et $\Gamma \vdash N \Rightarrow A$, alors $\Gamma \vdash \{N/x\}M \Rightarrow \{N/x\}B$.
ii. Si $\Gamma, x : A \vdash M \Leftarrow B$ et $\Gamma \vdash N \Rightarrow A$, alors $\Gamma \vdash \{N/x\}M \Leftarrow \{N/x\}B$.

Nous souhaitons maintenant montrer la correction du $\lambda\Pi^b$-modulo par rapport au $\lambda\Pi$-calcul modulo, le théorème suivant énonce ce résultat.

Théorème 1 (Correction du $\lambda\Pi^b$-modulo)**.** Soit M un terme et A un type du $\lambda\Pi^b$-modulo,

i. si $\mathscr{D}$ prouve $\Gamma \vdash M \Rightarrow A$ et si Γ wf, on a $\Gamma \vdash M : A$;
ii. si $\mathscr{D}$ prouve $\Gamma \vdash M \Leftarrow A$, si Γ wf et si $\Gamma \vdash A : s$ ou $A = s$ avec s une sorte, on a $\Gamma \vdash M : A$.

Démonstration. Soit $\mathscr{D}$ une dérivation de typage, on raisonne par induction sur $\mathscr{D}$. On fait une analyse de cas sur la dernière règle de la dérivation :

- *sort* : On doit prouver i. Par hypothèse, on a Γ tel que Γ wf on peut donc appliquer *sort* du $\lambda\Pi$-calcul modulo pour obtenir un preuve de $\Gamma \vdash \mathbf{Type} : \mathbf{Kind}$.
- *var* : On doit prouver i. On procède comme dans le cas précédent.
- *app* : On doit prouver i. On a Γ wf, on applique i pour obtenir que $\Gamma \vdash M : C$. Remarquons maintenant que par propriété du $\lambda\Pi$-calcul modulo on a $\Gamma \vdash C : s$ ou bien $C = s$ (avec s une sorte). Le terme C ne peut être une sorte car il se réduit en un produit, donc $\Gamma \vdash C : s$. Par réduction du sujet $\Gamma \vdash \Pi x : A.\, B : s$. On peut donc appliquer *conv* du $\lambda\Pi$-calcul modulo pour avoir $\Gamma \vdash M : \Pi x : A.\, B$.
  Maintenant on applique ii à la preuve de $\Gamma \vdash N \Leftarrow A$, en remarquant que l'on a $\Gamma \vdash A : \mathbf{Type}$ par inversion. On obtient $\Gamma \vdash N : A$ et on peut conclure via *app* du $\lambda\Pi$-calcul modulo.

- *prod* : On doit prouver ii. Ici il suffit d'appliquer l'hypothèse de récurrence ii et la règle *prod* du $\lambda\Pi$-calcul modulo pour conclure.
- *abs* : On doit prouver ii. On a Γ wf et C tels que $\Gamma \vdash C : s$ pour s une sorte car C ne peut pas être une sorte. Alors par réduction du sujet, $\Gamma \vdash \Pi x : A.\, B : s$ et par inversion $\Gamma \vdash A :$ **Type** et $\Gamma, x : A \vdash B : s$. Donc, par hypothèse de récurrence $\Gamma, x : A \vdash M : B$.
  On peut maintenant appliquer *abs* du $\lambda\Pi$-calcul modulo pour obtenir un preuve de $\Gamma \vdash \lambda x.\, M : \Pi x : A.\, B$.
  On conclut en appliquant la règle *conv* du $\lambda\Pi$-calcul modulo pour obtenir la preuve de $\Gamma \vdash \lambda x.\, M : C$.
- *conv* : On doit prouver ii. Le cas dans lequel $B =$ **Kind** est facile. Dans le cas contraire, on a Γ wf et s tel que $\Gamma \vdash B : s$, on applique l'hypothèse de récurrence i pour obtenir que $\Gamma \vdash M : A$. Il reste à applique la règle *conv* du $\lambda\Pi$-calcul modulo pour conclure que $\Gamma \vdash M : B$.

□

Pour finir la preuve de correction, il est nécessaire d'avoir un lien entre les relations de bonne formation des contextes. De manière plus formelle, on prouve le théorème suivant.

Théorème 2 (Correction de la bonne formation des contextes)**.** Si $\Gamma\ \mathsf{wf}_b$ alors Γ wf.

Démonstration. Par induction sur la dérivation de $\Gamma\ \mathsf{wf}_b$.
- *empty* : Facile.
- *decl* : On applique l'hypothèse de récurrence et la partie ii du théorème de correction pour conclure avec *decl* du $\lambda\Pi$-calcul modulo.

□

3.2.2 Un nouvel algorithme de typage

Remarquons que le système donné dans la figure 2 est dirigé par la syntaxe (on applique la règle *conv* uniquement lorsque les deux autres règles de vérification ne s'appliquent plus).

Il donne donc naissance à un nouvel algorithme de vérification du $\lambda\Pi$-calcul modulo dans une version où les annotations de type sur les λ ont disparu. Le seul point à éclaircir reste la décision de la convertibilité de deux termes. Dans le cas où la relation $\longrightarrow^*$ est confluente et fortement normalisante, il est facile de tester la convertibilité de deux termes. En effet, deux termes seront convertibles si et seulement si leurs deux formes normales sont α-convertibles. Notons que $\longrightarrow^*_w$ peut ne pas être déterministe, dans ce cas il suffit de fixer une stratégie d'évaluation pour calculer les formes normales de tête faible.

Définition 4 (Relation standardisante)**.** Soit $\longrightarrow$, une relation confluente sur les pré termes et $\equiv$ la congruence associée. La relation $\longrightarrow$ est dite standardisante si :
- $M \equiv x\ N_1 \ldots N_k$ implique $M \longrightarrow^*_w x\ N'_1 \ldots N'_k$ et $N_1 \equiv N'_1 \ldots N_k \equiv N'_k$
- $M \equiv \lambda x.\, N$ implique $M \longrightarrow^*_w \lambda x.\, N'$ et $N \equiv N'$
- $M \equiv \Pi x : A.\, B$ implique $M \longrightarrow^*_w \Pi x : A'.\, B'$, $A \equiv A'$ et $B \equiv B'$
- $M \equiv$ **Type** implique $M \longrightarrow^*_w$ **Type**
- $M \equiv$ **Kind** implique $M \longrightarrow^*_w$ **Kind**

On peut également se poser la question de la complétude de l'algorithme obtenu. Pour l'instant aucune preuve n'est présentée pour le $\lambda\Pi^b$-modulo (typage bidirectionnel avec réécriture) mais, dans le cas où la relation de réécriture du système est confluente, standardisante et fortement normalisante la correction et la complétude de certains algorithmes très similaires sont prouvées par Abel et Altenkirch [1] et par Coquand [10].

3.3 Systèmes hors contexte

Un problème classique qui survient lors de l'implémentation des vérificateurs de preuves est la gestion des variables et en particulier, l'α-conversion et la substitution sans capture. Bien qu'élémentaire, la substitution sans capture n'est pas triviale à définir correctement, et comme Abelson et Sussman [2] le font remarquer, la littérature en logique et en sémantique des langages de programmation recèle une longue liste de définitions de la substitution erronées.

Dans le noyau de vérification DEDUKTI, afin d'appliquer au mieux le critère de de Bruijn et de garder l'implémentation la plus courte et la plus simple possible nous représentons les termes en syntaxe abstraite d'ordre supérieur (HOAS). Cette technique, dans laquelle les abstractions (λ et Π dans notre cas) sont représentées par

des fonctions, permet de réutiliser les mécanismes du méta langage pour avoir une substitution à coût zéro. Plus de détails seront donnés sur la HOAS dans la section 4.

En particulier, l'utilisation de HOAS force la distinction entre variable et paramètre. Les *variables* sont toujours liées et les *paramètres* sont les variables libres. Pour examiner une abstraction il faut l'appliquer à un terme T, ceci a pour effet de substituer toutes les occurrences de la variable liée par T. Mais rien n'empêche de substituer plus qu'un paramètre, on peut par exemple passer un paramètre accompagné de son type. Autrement dit, on substitue non plus un paramètre mais une *boîte* qui contient un nom et un type. Avec cette technique les paramètres sont toujours munis de leur type, il n'y a donc plus besoin de gérer un contexte. L'évolution dans cette direction mène aux systèmes dits *hors contexte*.

Les systèmes hors contexte sont décrits extensivement dans la thèse de Boespflug [6], il y présente une version des PTS hors contexte et montre la correction et la complétude de ces systèmes par rapport aux PTS classiques.

3.3.1 Le $\lambda\Pi^{cf}$-modulo

Le système de type qui résulte du mélange entre typage bidirectionnel et typage hors contexte est présenté dans la figure 4, il a été introduit par Boespflug, Carbonneaux et Hermant [8]. La syntaxe des pré termes est définie ci dessous.

Définition 5 (Syntaxe du $\lambda\Pi^{cf}$-modulo)**.** Si $\mathscr{P}$ est un ensemble infini dénombrable de paramètres et $\mathscr{V}$ un ensemble infini dénombrable de variables, l'ensemble des sujets $\mathscr{S}$ et l'ensemble des classifieurs $\mathscr{C}$ sont définis inductivement comme suit :

- $\mathscr{V} \subset \mathscr{S}$
- $\mathbf{Type} \in \mathscr{S}$
- Si $u, v, A, B \in \mathscr{S}$, $x \in \mathscr{V}$, $y \in \mathscr{P}$ et $T \in \mathscr{C}$ alors $[y : T], \lambda x.\, u, \Pi x : A.\, B, u\ v \in \mathscr{S}$
- $\mathscr{V} \subset \mathscr{C}$ et $\mathscr{P} \subset \mathscr{C}$.
- $\mathbf{Type}, \mathbf{Kind} \in \mathscr{C}$
- Si $u, v, A, B \in \mathscr{C}$ et $x \in \mathscr{V}$ alors $\lambda x.\, u, \Pi x : A.\, B, u\ v \in \mathscr{C}$

La différence essentielle entre les sujets et les classifieurs se trouve dans la gestion des paramètres, dans les classifieurs ils sont nus tandis que dans les sujets ils doivent venir avec leur type (qui est un classifieur). Par convention les paramètres seront notés y tandis que les variables seront notées x. Remarquons qu'il est impossible de lier les paramètres car ils représentent des variables libres.

Dans les règles de la figure 4 il n'est pas précisé quels termes sont les sujets, et quels termes sont les classifieurs. À l'instar de la section 3.2.1 les sujets sont à gauche dans les jugements tandis que les classifieurs sont à droite. Il faut cependant remarquer que dans certaines règles un sujet doit être converti en classifieur, on utilise alors la relation $\sim \subset \mathscr{S} \times \mathscr{C}$ ci dessous pour passer d'une représentation à une autre :

$$\frac{}{x \sim x} \qquad \frac{}{[y : A] \sim y} \qquad \frac{}{\mathbf{Type} \sim \mathbf{Type}} \qquad \frac{}{\mathbf{Kind} \sim \mathbf{Kind}}$$

$$\frac{M \sim M'}{\lambda x.\, M \sim \lambda x.\, M'} \qquad \frac{A \sim A' \qquad B \sim B'}{\Pi x{:}A.\, B \sim \Pi x{:}A'.\, B'} \qquad \frac{M \sim M' \qquad N \sim N'}{M\ N \sim M'\ N'}$$

Cette relation est totale à gauche et unique à droite on peut donc l'utiliser comme une fonction pour passer de sujet à classifieur dans les règles *prod* et *app* de la figure 4.

Nous cherchons maintenant à prouver la correction de ce système par rapport au $\lambda\Pi$-calcul modulo en utilisant le résultat de correction de la section précédente. Commençons par remarquer que l'ensemble des classifieurs de notre calcul peut être confondu avec l'ensemble des pré termes du $\lambda\Pi^{b}$-modulo. En effet, pour passer d'un ensemble à l'autre il suffit d'expliciter ou d'oublier la distinction entre variable libre (paramètre) et variable liée (variable). Cette conversion implicite simplifiera les notations dans la suite du développement. Nous définissons maintenant une fonction de restauration des annotations.

Définition 6 (Restauration des annotations)**.** Soient Γ est un contexte et M est un classifieur (ou un terme du $\lambda\Pi^{b}$-modulo) différent de **Kind** dont toutes les paramètres apparaissent dans le domaine de Γ, le sujet $\Gamma \downarrow M$ est

$\boxed{\vdash M \Rightarrow A}$ Le terme M synthétise le type A

$$(sort)\ \frac{}{\vdash \mathbf{Type} \Rightarrow \mathbf{Kind}} \qquad (var)\ \frac{}{\vdash [y:A] \Rightarrow A}$$

$$(app)\ \frac{\vdash M \Rightarrow C \qquad C \longrightarrow_w^* \Pi x{:}A.\, B \qquad \vdash N \Leftarrow A}{\vdash M\ N \Rightarrow \{N/x\}B}$$

$\boxed{\vdash M \Leftarrow A}$ Le terme M est vérifié contre le type A

$$(abs)\ \frac{C \longrightarrow_w^* \Pi x{:}A.\, B \qquad \vdash \{[y:A]/x\}M \Leftarrow \{y/x\}B}{\vdash \lambda x.\, M \Leftarrow C}$$

$$(prod)\ \frac{\vdash A \Leftarrow \mathbf{Type} \qquad \vdash \{[y:A]/x\}B \Leftarrow s}{\vdash \Pi x{:}A.\, B \Leftarrow s}\ s \in \{\mathbf{Type}, \mathbf{Kind}\}$$

$$(conv)\ \frac{\vdash N \Rightarrow B \qquad A \equiv B}{\vdash N \Leftarrow A}$$

FIGURE 4 – Règles de typage du $\lambda\Pi^{cf}$-modulo

défini par induction sur la structure de M :

$$\Gamma \downarrow x = x \qquad \Gamma \downarrow y = [y:A] \text{ si } y:A \in \Gamma \qquad \Gamma \downarrow \mathbf{Type} = \mathbf{Type}$$

$$\Gamma \downarrow \lambda x.\, M = \lambda x.\, \Gamma \downarrow M \qquad \Gamma \downarrow \Pi x:A.\, B = \Pi x:\Gamma \downarrow A.\, \Gamma \downarrow B \qquad \Gamma \downarrow (M\ N) = (\Gamma \downarrow M)\ (\Gamma \downarrow N)$$

Dans la suite, si M est un sujet, nous noterons M' l'unique classifieur tel que $M \sim M'$. Nous pouvons maintenant énoncer le théorème de correction du $\lambda\Pi^{cf}$-modulo.

Théorème 3 (Correction du $\lambda\Pi^{cf}$-modulo)**.** Soit M et A deux termes du $\lambda\Pi^{cf}$-modulo.

i. si $\mathscr{D}$ prouve $\vdash M \Rightarrow A$ alors pour tout Γ tel que $\Gamma \downarrow M' = M$ on a $\Gamma \vdash M' \Rightarrow A$;

ii. si $\mathscr{D}$ prouve $\vdash M \Leftarrow A$ alors pour tout Γ tel que $\Gamma \downarrow M' = M$ on a $\Gamma \vdash M' \Leftarrow A$.

Démonstration. Soit $\mathscr{D}$ une dérivation de typage, on raisonne par induction sur $\mathscr{D}$. On fait une analyse de cas sur la dernière règle de la dérivation :

- *sort* : On doit prouver i. On applique directement *sort* du $\lambda\Pi^{b}$-modulo.
- *var* : On doit prouver i. On a $M = [y:A]$ donc $M' = y$ et on sait que $\Gamma \downarrow y = [y:A]$ donc $y:A \in \Gamma$ et on peut appliquer *var* du $\lambda\Pi^{b}$-modulo pour conclure.
- *app* : On doit prouver i. Remarquons que si $\Gamma \downarrow M'\ N' = M\ N$ on a $\Gamma \downarrow M' = M$ et $\Gamma \downarrow N' = N$ par définition de $\downarrow$ et inversion de $\sim$; on peut maintenant conclure facilement avec les deux hypothèses de récurrence.
- *prod* : On doit prouver ii. Par définition de $\downarrow$ on a $\Gamma \downarrow A' = A$ donc on peut appliquer ii pour avoir $\Gamma \vdash A' \Leftarrow \mathbf{Type}$. Maintenant remarquons que $(\{[y:A]/x\}B)' = \{y/x\}B'$ et que $\Gamma, y:A \downarrow \{y/x\}B' = \{[y:A]/x\}B$, donc par ii on a $\Gamma, y:A \vdash \{y/x\}B' \Leftarrow s$. Enfin par *prod* du $\lambda\Pi^{b}$-modulo on obtient $\Gamma \vdash \Pi x:A'.\, B' \Leftarrow s$ car $\Pi y:A'.\, \{y/x\}B' = \Pi x:A'.\, B'$.
- *abs* : On doit prouver ii. On procède comme dans le cas précédent.
- *conv* : On doit prouver ii. On l'obtient par hypothèse de récurrence et par *conv* du $\lambda\Pi^{b}$-modulo.

□

Le théorème 3 n'est pas un théorème de correction dans toute sa généralité mais prouve seulement la correction dans le cas où M est l'image d'un terme du $\lambda\Pi$-calcul modulo par une transformation de "restauration des annotations" via un contexte bien formé. Toutefois, ce résultat n'est pas vain et on peut l'utiliser pour obtenir la correction d'un algorithme qui utiliserait le $\lambda\Pi^{cf}$-modulo comme système de types. En pratique, en se basant sur le fait qu'un fichier source est une suite de déclarations vérifiée incrémentalement il est simple de construire après chaque déclaration un contexte bien formé Γ et d'appliquer le théorème 3 à $\Gamma \downarrow M'$ où M' est le terme suivant à vérifier. L'algorithme utilisé dans DEDUKTI vérifie cette propriété mais les détails fastidieux ne sont pas exposés ici.

Il est encore une fois naturel de se poser la question de la complétude du système obtenu. Il n'existe pas encore de preuve de cette propriété pour le $\lambda\Pi^{cf}$-modulo mais elle est vraie pour les PTS hors contexte présentés dans la thèse [6]. Par ailleurs, le $\lambda\Pi^{cf}$-modulo est exactement le système implémenté par DEDUKTI qui est déjà capable de vérifier des preuves réalistes (bibliothèque standard de COQ, HOL, *c.f.* section 5.2), on a donc une sorte de résultat de complétude expérimentale. Ce système est, comme dans la section 3.2.1, algorithmique et la section 5 en décrit une implémentation qui interprète trivialement les règles d'inférence en un programme fonctionnel simple et court.

4 Conversion par évaluation

Dans la section 3.3.1 nous avons présenté une version abstraite d'un algorithme de typage. Cette présentation n'est pas du tout explicite en ce qui concerne l'implémentation des substitutions et la normalisation des types. Le programme que nous allons développer pour gérer ces transformations sur les termes doit idéalement être le plus court, le plus simple et le plus efficace possible. L'efficacité est primordiale car les termes qui résultent des traductions vers DEDUKTI utilisent copieusement le calcul dans les types. De plus, calculer les formes normales de tête faibles peut nécessiter un nombre arbitraire de pas de réduction. C'est par exemple le cas lorsque l'on essaie de vérifier une preuve qui utilise des techniques de réflexion. La réflexion permet de transformer un raisonnement statique lourd en un nombre réduit d'appels à des procédures de décision, et ainsi, elle fait diminuer drastiquement la taille des termes de preuve en tirant partie des possibilités calculatoires de la logique.

Pour résoudre le problème de la normalisation et de la conversion nous avons développé une technique qui s'appuie uniquement sur un évaluateur d'un langage fonctionnel existant. Utiliser un évaluateur existant garantit l'efficacité de notre méthode car ces programmes ont été constamment améliorés ces dernières années pour proposer des langages fonctionnels toujours plus rapides. Cela permet aussi de n'avoir à maintenir que le seul vérificateur de types et pas un évaluateur. Par contre, on peut reprocher à cette technique d'inclure dans la base de confiance un évaluateur complet alors que cette dernière est censée rester minimale. Cet argument est valide mais il faut le nuancer car les évaluateurs de langages de programmation sont testés dans des conditions bien plus extrêmes et par bien plus d'utilisateurs qu'un évaluateur ad hoc pour un λ-calcul.

Dans toute cette partie le méta langage—la cible de la compilation—utilisé sera un langage fonctionnel quelconque bénéficiant du filtrage, la syntaxe utilisée évoquera fortement les langages Haskell et OCaml.

4.1 Compilation du $\lambda\Pi$-calcul modulo

Pour pouvoir utiliser un évaluateur extérieur nous devons d'abord traduire les termes du $\lambda\Pi$-calcul modulo vers des programmes de notre méta langage. Pour éviter des interactions entre le système de types du $\lambda\Pi$-calcul modulo et celui du méta langage, nous devons réaliser un plongement non typé. Une manière classique de faire ce plongement est d'utiliser un type algébrique du méta langage qui est isomorphe à ses flèches, on peut alors représenter tout terme du $\lambda\Pi$-calcul modulo. Considérons le type suivant :

$$\begin{aligned}\mathtt{data}\ \mathsf{Code} = {} & \mathsf{Var}_C\ \mathsf{Int} \mid \mathsf{App}_C\ \mathsf{Code}\ \mathsf{Code} \\ & \mid \mathsf{Lam}_C\ (\mathsf{Code} \to \mathsf{Code}) \mid \mathsf{Pi}_C\ \mathsf{Code}\ (\mathsf{Code} \to \mathsf{Code}) \\ & \mid \mathsf{Type}_C \mid \mathsf{Kind}_C\end{aligned}$$

On dit que les valeurs de ce type sont du *code* car elles ne nous intéressent que pour leur comportement calculatoire, on veut trouver leur forme normale efficacement. Notons que cette représentation est en HOAS : on utilise des fonctions du méta langage pour représenter les lieurs de notre langage source.

L'identité des variables est en générale peu importante et cela explique que l'on utilise les conventions de Barendregt lors de développements formels sur les λ-calculs. L'unique opération importante pour les variables est la substitution : on veut être capable de substituer toutes les occurrences d'une variable dans un terme facilement. De manière duale, l'identité des paramètres est cruciale et on ne substitue jamais un paramètre par un

autre terme. Ce constat explique que notre fonction de traduction associe les variables du méta langage aux variables et des éléments d'un type comparables, dans notre cas Int , aux paramètres. Ainsi on va traduire le terme $\lambda x.\, xy$ par $\mathsf{Lam}_C\ (\lambda x \to \mathsf{App}_C\ \ x\ \ (\mathsf{Var}_C\ \ 0))$. En utilisant ce mécanisme il est possible de traduire naïvement un terme M en transportant sa syntaxe sur les constructeurs du type Code, on note $[M]$ cette traduction.

$$[x] = x \qquad [y] = \mathsf{Var}_C\ \ i_y \qquad [\mathbf{Type}] = \mathsf{Type}_C \qquad [\mathbf{Kind}] = \mathsf{Kind}_C$$

$$[\lambda x.\, u] = \mathsf{Lam}_C\ (\lambda x \to [u]) \qquad [\Pi x : A.\, B] = \mathsf{Pi}_C\ [A]\ (\lambda x \to [B]) \qquad [u\ v] = \mathsf{App}_C\ [u]\ [v]$$

Dans le même esprit de l'exploitation du méta langage, on souhaite réutiliser ses capacités calculatoires pour réduire les termes de notre calcul. Il est possible d'écrire un évaluateur de notre λ-calcul dans le méta langage mais ce n'est pas un solution optimale car il faut faire une inspection structurelle couteuse sur les termes traduits de notre λ-calcul pour les normaliser. Malgré tout, cette fonction d'évaluation n'est pas bonne à jeter, on peut, lors de la traduction d'un terme M, l'appliquer partiellement à $[M]$ et ainsi gagner un précieux temps de calcul— le temps qui est passé à localiser les radicaux. Cette optimisation par l'évaluation partielle permet de définir une nouvelle fonction de traduction sur les termes notée $[\![.]\!]$ qui est la composition de la fonction [.] et d'un évaluateur. Il est remarquable que la fonction $[\![.]\!]$ ne diffère qu'en un point de la première : les applications $[\![u\ v]\!]$ sont traduites par app $[\![u]\!]$ $[\![v]\!]$ où app est définie par les équations :

```
app :: Code → Code → Code
app (Lam_C f) b = f b
app a         b = App_C a b
```

Si le premier argument de la fonction app est une fonction, on peut réaliser l'application, autrement l'évaluation est bloquée et le résultat est un terme neutre. Notons que l'évaluation des termes suit maintenant la stratégie utilisée par l'évaluateur du méta langage. En particulier, comme les évaluateurs des langages fonctionnels usuels ne réduisent qu'en forme normale de tête faible un objet de type Code n'est pas en forme normale, il faut encore réduire sous les lieurs. C'est déjà suffisant pour appliquer les règles *abs* et *app* du $\lambda\Pi^{cf}$-modulo mais nous avons besoin d'une fonction spéciale pour tester la convertibilité de deux termes dans le cas de la règle *conv*. Il est possible de définir une telle fonction presque structurellement sur les termes :

```
conv :: Int → Code → Code → Bool
conv n (Var_C x) (Var_C x') = x == x'
conv n (Lam_C f) (Lam_C f') = conv (n+1) (f (Var_C n)) (f' (Var_C n))
conv n (Pi_C A f) (Pi_C A' f') = conv n A A' ∧
                                 conv (n+1) (f (Var_C n)) (f' (Var_C n))
conv n (App_C u v) (App_C u' v') = conv n u u' ∧ conv n v v'
conv n Type_C Type_C = True
conv n Kind_C Kind_C = True
conv n _ _ = False
```

Appliquer les fonctions de HOAS à des paramètres frais générés par le compteur en premier argument force le méta langage à évaluer le corps des fonctions. On compare donc bien les formes normales des deux termes.

Cette technique dite de *conversion par évaluation* présentée dans [6, 5] peut être mise en parallèle avec les techniques de normalisation par évaluation (NbE) comme décrites dans [9, 15]. Mais au contraire de ces dernières il n'y a pas d'étape de *réification* des objets du modèle sémantique : le test de conversion est réalisé directement sur les versions dynamiques des termes. Cette différence est due au fait que nous ne souhaitons pas obtenir une forme normale mais un booléen qui signifie si deux termes sont convertibles ou non. L'architecture de notre vérificateur de preuves DEDUKTI qui repose sur cette technique est détaillée dans la section 5.

4.2 Extension de l'égalité algorithmique

Le $\lambda\Pi$-calcul modulo offre la possibilité à ses utilisateurs d'ajouter des règles de réécriture pour enrichir l'égalité algorithmique, c'est à dire la congruence $\equiv_{\beta R}$ de la règle *conv*. Cependant le schéma présenté précédemment ne tient pas compte de cette possibilité. Malgré cela, l'approche décrite est très facilement extensible en remarquant que l'on peut voir les règles de réécriture comme des programmes fonctionnels sur les valeurs de type Code. Un ensemble de règles de réécriture concernant une variable donnée est alors traduit comme décrit dans la figure 5. De manière analogue à ce qui a été fait dans la section précédente, lorsqu'aucune règle ne peut être appliqué le programme fonctionnel généré produit un terme neutre.

$$||x||^{pat} = x$$
$$||c\ N_1\ \ldots\ N_n||^{pat} = \mathsf{App}_C\ (\ldots(\mathsf{App}_C\ (\mathsf{Var}_C\ c)\ ||N_1||^{pat})\ldots)\ ||N_n||^{pat}$$

$$\left|\left| \begin{array}{ccc} c\ N_{11}\ \ldots\ N_{1n} & \longrightarrow & M_1 \\ \vdots & & \vdots \\ c\ N_{m1}\ \ldots\ N_{mn} & \longrightarrow & M_m \end{array} \right|\right| = \begin{array}{ccccccc} c & ||N_{11}||^{pat} & \ldots & ||N_{1n}||^{pat} & = & ||M_1|| \\ & \vdots & & \vdots & & \vdots \\ c & ||N_{m1}||^{pat} & \ldots & ||N_{mn}||^{pat} & = & ||M_m|| \\ c & \_ & \ldots & \_ & = & ||c\ x_1\ \ldots\ x_n||^{pat} \end{array}$$

FIGURE 5 – Traduction des règles de réécriture en fonctions sur les classifieurs

5 Implémentation et performances

Un vérificateur de types du $\lambda\Pi$-calcul modulo basé sur le $\lambda\Pi^{cf}$-modulo doit réaliser deux actions sur les termes : il doit *vérifier que les sujets sont bien typés* par des classifieurs et pour cela il doit être capable de *normaliser les classifieurs*. Ce constat mène naturellement à l'utilisation de deux représentations différentes pour les termes du $\lambda\Pi$-calcul modulo. Ce besoin de deux représentations à rapprocher de la distinction faite entre sujet et classifieur dans le $\lambda\Pi^{cf}$-modulo. Afin d'éviter les conversions multiples entre sujets et classifieurs notre approche repose sur la compilation des termes du $\lambda\Pi$-calcul modulo en un vérificateur de types spécialisé écrit dans un méta langage. Ce dernier sera ensuite exécuté et décidera si les termes donnés sont bien typés.

Le programme DEDUKTI est un compilateur des termes du $\lambda\Pi$-calcul modulo vers un méta langage fonctionnel. Au contraire d'autres approches dont celle de COQ, la compilation des termes de preuve est faite une fois pour toute et il n'est pas nécessaire d'appeler le compilateur au cas par cas lors de la passe de vérification de types. Il est en effet appréciable de ne pas avoir à intercaler une multitude d'appels au compilateur pour des petits fragments de code lors du typage. Cette architecture est détaillée dans le schéma de la figure 6.

5.1 Le programme DEDUKTI

Le compilateur DEDUKTI a donc pour charge de traduire tous les termes d'un fichier source `.dk` en un vérificateur de type spécialisé. Ce vérificateur est implémenté dans le méta langage de notre système. Nous décrivons dans cette section les détails de la traduction vers ce langage, les différentes structures de données utilisées et les optimisations faites sur l'ancienne version de DEDUKTI.

5.1.1 Deux représentations pour deux fins

Dans la section 4 nous avons exposé une traduction efficace pour les classifieurs du $\lambda\Pi^{cf}$-modulo. Elle permet de tester la convertibilité de deux termes en utilisant au maximum les ressources calculatoires de notre langage hôte.

Cependant, les sujets de notre calcul doivent aussi être représentés dans le méta langage afin de les inspecter pour valider leur bonne formation. Cette représentation n'est pas soumise aux mêmes contraintes que celle des classifieurs. En effet, alors que l'on sait que les classifieurs sont bien formés, les sujets peuvent être des pré termes

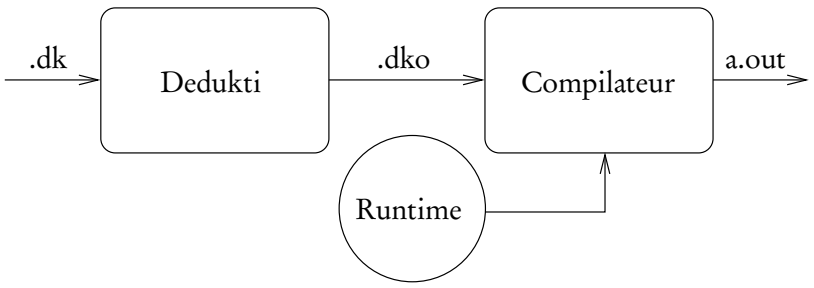

FIGURE 6 – Fonctionnement global de DEDUKTI

quelconques. Il est donc dangereux de les stocker en forme normale de tête faible car cette dernière peut très bien ne pas exister. On choisit donc une représentation statique en HOAS pour faciliter l'implémentation des substitutions. Le type de données suivant représentera les sujets.

$$\begin{aligned}\mathsf{data}\ \mathsf{Term} = {} & \mathsf{Box}_T\ \mathsf{Int}\ \mathsf{Code} \mid \mathsf{App}_T\ \mathsf{Term}\ \mathsf{Term} \\ & \mid \mathsf{Lam}_T\ (\mathsf{Term} \to \mathsf{Term}) \mid \mathsf{Pi}_T\ \mathsf{Term}\ (\mathsf{Term} \to \mathsf{Term}) \\ & \mid \mathsf{Type}_T\end{aligned}$$

Les constructeurs syntaxiques d'un terme M sont associés directement aux constructeurs du type Term pour obtenir la *représentation statique* $\ulcorner M \urcorner$.

$$\ulcorner x \urcorner = x \qquad \ulcorner [x : A] \urcorner = \mathsf{Box}_T\ i_x\ [\![A]\!] \qquad \ulcorner \mathbf{Type} \urcorner = \mathsf{Type}_T$$

$$\ulcorner \lambda x.\, u \urcorner = \mathsf{Lam}_T(\lambda x \to \ulcorner u \urcorner) \qquad \ulcorner \Pi x : A.\, B \urcorner = \mathsf{Pi}_T\ \ulcorner A \urcorner\ (\lambda x \to \ulcorner B \urcorner) \qquad \ulcorner u\ v \urcorner = \mathsf{App}_T\ \ulcorner u \urcorner\ \ulcorner v \urcorner$$

Combinée avec la représentation dynamique de la section 4, il devient alors très facile d'implémenter le vérificateur de type extrait du système de la figure 4. Il reste cependant à voir que le $\lambda\Pi^{cf}$-modulo fait un usage implicite d'une coercion des valeurs de type Term vers les valeurs de type Code, cette coercion peut être faite pendant l'exécution en implémentant la relation fonctionnelle $\sim$ de la section 3.3.1. Cependant faire la conversion à chaque application des règles *app* et *abs* peut se révéler coûteux, d'autant plus que, dans des cas pathologiques, la conversion d'un même terme devra être refaite un nombre de fois exponentiel. On choisit donc de changer légèrement la représentation statique des termes en ajoutant par un procédé d'*appariement* (*glueing* en anglais) les représentations dynamiques de certaines parties des termes. Les constructeurs Pi_T et App_T deviennent donc Pi_T Pair (Term $\to$ Term) et App_T Term Pair où le type Pair est défini par la déclaration suivante.

$$\mathsf{data}\ \mathsf{Pair} = \mathsf{Pair}\ \mathsf{Term}\ \mathsf{Code}$$

Pour obtenir le code associé à un terme il suffit d'ouvrir les *paires* (les objets de type Pair). On peut remarquer ici que si le méta langage a une stratégie d'évaluation en appel par valeurs, le membre de type Code d'une paire doit être protégé dans un *glaçon* (*thunk* en anglais) pour empêcher l'évaluateur d'exécuter du code qui n'est pas encore typé.

5.1.2 La compilation JIT des preuves

La première implémentation de DEDUKTI utilisait le langage Haskell comme méta langage, c'est un langage fonctionnel robuste pour lequel il existe un compilateur optimisant vers du code machine. Ce langage se prêtait extrêmement bien aux preuves très calculatoires car le temps passé à compiler les termes de preuve rendait la vérification de types très rapide. Il est aujourd'hui possible de trouver de plus en plus de preuves qui font usage du calcul, parmi les plus célèbres on compte la preuve du théorème des quatre couleurs et celle de la conjecture de Kepler. Pour vérifier ces preuves avoir un compilateur de termes de preuves est un avantage majeur.

Compilateurs JIT Nous remarquons cependant que la grande majorité des preuves disponibles aujourd'hui ne fait pas un usage aussi intense du calcul. Certains cadres logiques n'incluent même pas la possibilité de mêler raisonnement et calcul, c'est en particulier le cas de HOL, qu'Ali Assaf a traduit en DEDUKTI pendant son stage cette année. Pour être en mesure de traiter des preuves de ce type nous avons besoin d'un système plus flexible que la compilation systématique. Aussi nous proposons d'utiliser comme méta langage un langage disposant d'un compilateur *juste à temps* (JIT). Ce genre de compilateur implémente une approche de "tarification à l'usage" au sens où seul le code exécuté intensivement va être compilé en code machine, le reste est interprété par un interpréteur classique. Ainsi seuls les termes de preuve intensivement calculatoires vont bénéficier de la compilation sans pour autant pénaliser les parties de la preuve moins nécessiteuses en calcul. Un compilateur JIT offre donc une souplesse nécessaire pour pouvoir utiliser DEDUKTI comme vérificateur de types universel.

Le méta langage Lua Le langage Lua [14] est un langage de programmation minimal impératif développé au PUC-Rio. Destiné à être utilisé dans d'autres applications comme langage d'extension, il a eu beaucoup de succès dans le milieu des éditeurs de jeux vidéos. S'en est suivi un besoin croissant de performance qui culmine avec la création de `luajit`, un compilateur JIT pour ce langage. Tous langages confondus, ce compilateur JIT est

probablement le plus efficace actuellement disponible, nous l'avons donc choisi comme interpréteur de notre méta langage.

Bien que Lua soit un langage impératif, les dernière versions possèdent quelques traits fonctionnels. En particulier, les clôtures—indispensables pour utiliser la HOAS—sont disponibles. Il est donc possible d'utiliser les représentations exposées dans les section précédentes si Lua est notre langage cible. Il reste le problème du filtrage, en effet, pour traduire les règles de réécriture nous utilisons le filtrage par cas du méta langage et la fonction de traduction de la figure 5. Lua ne possède pas la fonctionnalité de filtrage par cas et nous devons donc modifier la fonction de traduction de la figure 5 pour compiler les filtrages en des séquences d'expressions conditionnelles (if .. then .. else).

Deux techniques majeures sont disponibles pour compiler le filtrage, on peut, soit compiler vers des automates avec retour sur trace comme présenté dans [3], soit compiler vers des bons arbres de décision comme présenté dans [16]. Les automates avec retour sur trace génèrent un code de taille linéaire qui peut inspecter un grand nombre de fois la même valeur ; tandis que le code généré par les arbres de décision peut être de taille exponentielle mais est plus efficace. Les motifs utilisés dans DEDUKTI sont une version simple de ceux que l'on peut trouver dans OCaml ou Haskell ; en pratique dans les motifs DEDUKTI les cas pathologiques où la génération de code par les bons arbres de décision explose ne surviennent pas. Nous avons donc retenu la compilation par arbres de décision dans notre compilateur, l'algorithme utilisé est exactement celui présenté dans [16].

5.1.3 Autres optimisations

Décurryfication Dans la section 4.1 nous avons présenté un schéma de compilation pour les classifieurs du $\lambda\Pi^{cf}$-modulo. Ce schéma permet l'écriture d'un test de conversion simple et efficace, cependant il reste quelques points possibles à améliorer. En effet, considérons la fonction fold sur les listes, cette dernière nécessite 3 arguments pour pouvoir calculer effectivement et renvoyer un résultat. Le schéma de compilation tel qu'exposé plus haut va générer un code analogue au suivant.

```
fold :: Code
fold = Lam_C (λf → Lam_C (λz → Lam_C (λl → go f z l)))
  where go = ...
```

Chaque fois que ce code est appliqué à un argument à l'aide de la fonction app, une clôture du méta langage est créée, or cette création de clôture est en général une opération coûteuse. De plus, cet encodage peut empêcher le compilateur du méta langage d'optimiser l'allocation de clôtures à très faible durée de vie. Il faut donc trouver un moyen de minimiser la création de clôtures inutiles. Une solution à ce problème, présentée par Boespflug dans [5], est l'utilisation de plusieurs contructeurs pour l'abstraction : on va associer aux fonctions n-aires du $\lambda\Pi^{cf}$-modulo les fonctions n-aires du langage hôte.

```
data Code = Lam_C (Code → Code)
          | Lam²_C ((Code * Code) → Code)
          | ...
```

On peut alors traduire de manière plus économe la fonction fold par Lam^3_C (λ (f,z,l) → go f z l). Enfin, il faut adapter la fonction app pour qu'elle traite ces différentes abstractions. Une manière de faire est d'utiliser une famille de fonctions app indexée par les entiers. On distingue alors trois cas, si une fonction n-aire est appliquée à n arguments on peut l'appliquer directement et retourner le résultat, si elle est appliquée à $m < n$ arguments, on crée une clôture, et si elle est appliquée à $m > n$ arguments, on l'applique d'abord aux n premiers puis on appelle la fonction d'application de $m - n$ arguments sur le résultat et les arguments restants.

Comme Lua est un langage qui n'est pas typé, nous avons implémenté cette optimisation en utilisant un constructeur Lam_C spécial qui stocke une fonction et son arité ainsi qu'un tableau contenant la liste des arguments déjà appliqués, une fois que le tableau est assez long, on peut appliquer la fonction et renvoyer ce résultat. Pour valider l'approche, nous avons aussi réalisé une une implémentation analogue et bien typée pour Haskell qui fait usage des types algébriques généralisés pour stocker une fonction et son arité de manière sûre.

Représentation des termes neutres On peut remarquer que la manière dont est compilé le filtrage dans la figure 5 entraine n déréférencements si un motif a n arguments. Ceci vient du fait que les termes neutres sont représentés comme des arbres binaires avec des nœuds App_C. Cette représentation entraine un surcoût de l'exécution du filtrage, DEDUKTI utilise donc une représentation plus efficace des termes neutres sous forme d'épine (*spine* en anglais) représentée par un tableau. Cela revient à changer le constructeur Var_C pour un nouveau

constructeur tel que $\mathsf{Neutral}_C$ Int [Code], on remarque alors que le constructeur App_C n'est plus nécessaire et qu'on peut le retirer de la définition du type Code car la fonction d'application est alors :

```
app :: Code → Code → Code
app ( Lam_C f)          b = f b
app ( Neutral_C c l) b = Neutral_C c (l, b)
```

Notons que la définition de app est partielle, mais comme les objets de type Code sont bien typés, tous les autres cas sont exclus. Cette nouvelle représentation des termes neutres rend leur inspection plus facile et améliore la qualité du code produit par compilation du filtrage.

Transformation ANF Le procédé d'appariement expliqué dans la section 5.1 permet d'éviter de faire les traductions entre sujets et classifieurs à l'exécution. Cependant, dans des cas pathologiques, il peut générer un code de taille exponentielle en fonction de l'entrée. Cette explosion de la taille du code produit peut survenir lorsque de nombreuses applications sont imbriquées à droite, dans ce cas il se peut qu'un même code soit répété un nombre de fois exponentiel. Pour optimiser la taille code produit il faut sortir les application imbriquées. Une technique bien connue des programmeurs de compilateurs est la *mise en forme A-normale* (ANF) : la forme A-normale [13] est un langage intermédiaire courant dans les compilateurs qui rend la compilation vers le code machine et les optimisations plus faciles, il combine les avantages de la transformation CPS tout en gardant les continuations implicites. Ce langage intermédiaire explicite tous les pas de calcul est ainsi ne compose que des termes simples entre eux : les gros termes composites qui posent problème lors de la traduction sont éliminés.

5.1.4 Réécriture et types dépendants

Il est possible de classifier les règles de réécriture données à DEDUKTI par leur symbole de tête. On peut alors lire l'ensemble de règles associé à un symbole comme un programme fonctionnel classique.

Cependant, au contraire des programmes fonctionnels classiques, lorsque l'on utilise les types dépendants, la forme d'un terme peut contraindre celle d'un autre. Ce genre de situation apparait usuellement lorsque l'on essaie de faire du filtrage sur un constructeur qui spécifie certains paramètres de l'index dans son type de retour—chose que les langages fonctionnels récents doivent également gérer au niveau des types avec l'avènement des types algébriques généralisés. Un exemple de ce genre de situation est la fonction head des vecteurs définie dans la signature suivante.

$$\begin{array}{lll} \mathsf{Nat} : \mathbf{Type} & \mathsf{Z} : \mathsf{Nat} & \mathsf{S} : \mathsf{Nat} \to \mathsf{Nat} \\ \mathsf{Vector} : \mathsf{Nat} \to \mathbf{Type} & \mathsf{Nil} : \mathsf{Vector\ Z} & \mathsf{Cons} : \Pi n : \mathsf{Nat}.\ \mathsf{Nat} \to \mathsf{Vector}\ n \to \mathsf{Vector}\ (\mathsf{S}\ n) \end{array}$$

La fonction head possède le type et la définition suivants.

$$\mathsf{head} : \Pi n : \mathsf{Nat}.\ \mathsf{Vector}\ n \to \mathsf{Nat} \qquad \mathsf{head}\ \{\mathsf{S}\ n\}\ (\mathsf{Cons}\ n\ h\ t\,l) \longrightarrow h$$

Il semble qu'un motif non linéaire (plusieurs occurrences de n) soit nécessaire dans la partie gauche de la définition, de plus il est impossible de généraliser le premier n, autrement la partie gauche est mal typée à cause de l'utilisation des types dépendants. Dans le cas général les indices des familles de types inductives peuvent contenir des termes d'ordre supérieur, cela signifie qu'il faudrait implémenter le filtrage d'ordre supérieur dans DEDUKTI pour rendre le système utilisable. Remarquons qu'un motif non linéaire classique traduit le fait que l'on veut tester explicitement la convertibilité des deux occurrences de n alors que ce que l'on souhaite exprimer ici c'est que toutes les instances de la variable n sont uniquement déterminées par la première occurrence. De fait, filtrer le constructeur Cons force la taille du vecteur passée comme premier argument à être $\mathsf{S}\ n$. Tel quel, le premier argument n'a aucun rôle opérationnel il est présent uniquement pour des fins de typage.

En s'inspirant d'Agda, DEDUKTI permet de marquer un paramètre de la partie gauche d'une règle de réécriture comme non nécessaire au calcul en l'entourant d'accolades. Les arguments marqués ne filtreront pas de valeurs dans le code dynamique produit par le compilateur DEDUKTI. Par ce biais nous pouvons également ne pas implémenter le filtrage d'ordre supérieur qui est trop compliqué pour entrer dans le noyau d'un vérificateur de preuves.

5.2 Performances de la vérification de types

Le système DEDUKTI est un noyau conçu pour être utilisé comme prouveur universel, ainsi son interaction avec l'utilisateur est minimale et aucun outil n'est encore développé pour rendre le développement de preuves

en DEDUKTI interactif. Pour faire fonctionner DEDUKTI sur des preuves existantes il faut encoder le système logique dans le lequel elles sont écrites au sein du $\lambda\Pi$-calcul modulo. Ces encodages qui doivent être prouvés corrects (et complets) sont également un sujet d'étude de notre groupe. Actuellement, deux traducteurs existent, COQINE [7], un traducteur des preuves COQ dans DEDUKTI et Holide, un traducteur des preuves HOL Light sauvegardées dans le système OpenTheory.

5.2.1 Preuves HOL Light

Il existe une grande variété de systèmes HOL tous basés sur la théorie des types simples. Cependant, les vérificateurs de preuve pour HOL sont basés sur l'architecture LCF, ce mode de fonctionnement définit un type abstrait des théorème et n'autorise la construction d'objets de ce type que par la composition de fonctions spéciales qui sont correctes par rapport à une théorie des types simples. Donc, dans l'architecture LCF, seuls les théorèmes sont stockés et les preuves sont perdues.

Afin de factoriser la base de confiance et de permettre l'échange de théories, le projet OpenTheory propose un standard pour stocker les théorèmes et preuves fabriquées par les implémentations d'HOL. Ce standard est le format de fichier d'entrée de Holide qui génère par la suite un fichier vérifiable par DEDUKTI. Toute la libraire standard d'HOL est disponible dans ce format. Du fait de l'intensité de mes échanges avec le programmeur de Holide, il constitue le test le plus important du programme DEDUKTI à ce jour.

Les preuves HOL stockées dans le format OpenTheory ont la particularité intéressante qu'elles n'utilisent aucunement les capacités de calcul du $\lambda\Pi$-calcul modulo, c'est donc la situation dans laquelle l'architecture que nous proposons doit se comporter le plus mal. En effet, il n'est pas nécessaire dans ce cas d'avoir un test de conversion rapide : il suffirait dans la majorité des cas de comparer syntaxiquement les termes. Nous avons pu comparer DEDUKTI à une implémentation avec interpréteur d'un vérificateur de preuves implémenté par Ali Assaf (Camelide), les résultats de la comparaison sont synthétisés dans la figure 7.

Modules	Camelide	DEDUKTI & `lua`	DEDUKTI & `luajit`
$<$ 3Mo	236	340	154
$<$ 20Mo et $\geq$ 3Mo	1441	1344	×
Total	1677	1684	×

FIGURE 7 – Temps de vérification des modules HOL en secondes

Bien que l'étape de traduction de DEDUKTI soit presque inutile sur ce genre de preuve, notre implémentation réussi à vérifier les modules de HOL à la même vitesse ou plus rapidement (selon l'interpréteur de Lua choisi) qu'un vérificateur naïf. C'est un progrès majeur par rapport à l'ancienne version de DEDUKTI qui peinait à vérifier les cinq premiers modules de la bibliothèque standard d'HOL et échouait par manque de mémoire—souvent lors de la compilation du code Haskell produit—sur tous les autres.

Actuellement Holide est capable de traduire l'intégralité de la bibliothèque standard d'HOL mais seuls 50% peuvent êtres vérifiés par Camelide, 35% par la combinaison de DEDUKTI et de l'interpréteur standard de Lua et 22% par DEDUKTI et `luajit`. Ce qui limite les trois approches est souvent la consommation de mémoire et l'implémentation des interpréteurs de Lua. De fait, certaines limites programmées en dur dans les interpréteurs de Lua sont dépassées lors de la vérification de modules de grosse taille (plusieurs méga octets).

5.2.2 Preuves COQ

Le traducteur de COQ vers DEDUKTI COQINE est capable de traduire 84% de la bibliothèque standard de COQ. Néanmoins, à cause de subtilités dans la traduction des modules, des points fixes et du sous typage, seule une faible partie peut être effectivement vérifiée par DEDUKTI. Actuellement, 3% (3 méga octets sur 90) de la bibliothèque de COQ peut être vérifié par DEDUKTI. Ce maigre chiffre s'explique par le fait que certains modules clés qui apparaissent très tôt dans la chaîne de dépendance des vérifications ne sont pas traduits.

Toutefois, sur le petit nombre de modules traduits, les performances de notre vérificateur sont comparables à celles de Chicken, le vérificateur de fichiers `.vo` optimisé distribué avec COQ. Notons encore une fois que même si le calcul des constructions inductives permet du calcul dans les types avec la règle de conversion, la bibliothèque standard de COQ n'en fait pas beaucoup usage, c'est donc encore un test qui n'est pas optimal pour notre architecture. Pour valider pleinement notre approche nous aurions aimé faire des tests sur des preuves calculatoires mais les limites de COQINE nous bloquent. Les temps mis pour vérifier des petites parties de la bibliothèque de COQ sont exposés dans la figure 8.

Sections	Chicken	DEDUKTI & `luajit`
Arith	1.21	1.51
Bool	0.64	0.67
Classes	1.51	0.91
Init	0.61	0.93
Logic	0.78	1.07
NArith	0.35	0.58
Numbers	0.15	0.10
Program	0.27	0.32
Relations	0.35	0.58
Setoids	0.12	0.08
Sets	1.25	1.50
Structures	1.14	0.16
Wellfounded	0.28	0.40
ZArith	1.94	5.93
Total	11	15

FIGURE 8 – Temps de vérification des modules COQ en secondes

6 Conclusion

Le $\lambda\Pi$-calcul modulo offre un cadre logique souple et extensible via l'enrichissement de l'égalité algorithmique, il est minimal mais suffisamment expressif pour être proposé comme langage de preuve universel. Son vérificateur, DEDUKTI, est un programme court (1285 lignes de code C et 100 de Lua) qui se repose principalement sur la compilation JIT de Lua pour garantir son efficacité. Il implémente également un algorithme original et extrêmement simple basé sur le nouveau calcul bidirectionnel et sans contexte que nous proposons : le $\lambda\Pi^{cf}$-modulo.

Le vérificateur attend en entrée un ensemble de règles de réécriture bien formé et suppose qu'il vérifie des conditions raisonnables (confluence, terminaison forte, etc.). C'est donc simplement une petite roue de l'engrenage complet et la pertinence de sa sortie dépend de la bonne formation de l'ensemble de règle fourni : DEDUKTI vérifie uniquement les types et non les règles de réécriture. Pour l'instant aucune preuve importante n'est développée directement dans DEDUKTI, mais plusieurs théories ont été traduites et dans ce cas, l'ensemble de règles de l'encodage a été prouvé bien formé. Si nous souhaitons développer des preuves directement dans DEDUKTI, il faudra intégrer à notre outil des mécanismes de vérification (semi-)automatiques pour aider l'utilisateur à vérifier des propriétés sur son système de règles.

Nous essayons actuellement d'améliorer la robustesse de notre système global qui dépend à la fois de la capacité de DEDUKTI à traduire les termes en des programmes et de la capacité à l'évaluateur de notre méta langage à exécuter ces programmes. De fait, sur certains exemples nous avons observé que l'évaluateur Lua butait sur des constantes mal dimensionnées pour du code généré mécaniquement (profondeur maximale des termes, nombre maximum de variables dans la portée courante, ...) ; comme nous ne souhaitons pas avoir à modifier cet évaluateur, DEDUKTI doit produire des programmes plus courts et plus facile à exécuter pour les interpréteur actuels. Nous travaillons également à l'amélioration des outils de traduction dont dépend DEDUKTI pour avoir des tests de performance solides et pouvoir vérifier des preuves concises et très calculatoires sur lesquelles DEDUKTI doit se démarquer.

Références

[1] Andreas ABEL et Thorsten ALTENKIRCH : A partial type checking algorithm for Type : Type. *Electr. Notes Theor. Comput. Sci.*, 229(5):3–17, 2011.

[2] Harold ABELSON et Gerald J. SUSSMAN : *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd édition, 1996.

[3] Lennart AUGUSTSSON : Compiling pattern matching. *In* Jean-Pierre JOUANNAUD, éditeur : *Functional Programming Languages and Computer Architecture*, volume 201 de *Lecture Notes in Computer Science*, pages 368–381. Springer Berlin / Heidelberg, 1985.

[4] Henk BARENDREGT, S. ABRAMSKY, D. M. GABBAY, T. S. E. MAIBAUM et H. P. BARENDREGT : Lambda calculi with types. *In Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[5] Mathieu BOESPFLUG : Conversion by evaluation. *In Proceedings of the Twelfth Internation Symposium on Pracical Aspects of Declarative Languages*, Madrid, Spain, janvier 2010. A preliminary version was presented at the NbE'09 workshop.

[6] Mathieu BOESPFLUG : *Conception d'un noyau de vérification de preuves pour le λΠ-calcul modulo*. Thèse de doctorat, École polytechnique, janvier 2011.

[7] Mathieu BOESPFLUG et Guillaume BUREL : CoqInE : Translating the calculus of inductive constructions into the λΠ-calculus modulo. *In* David PICHARDIE et Tjark WEBER, éditeurs : *PxTP*, 2012.

[8] Mathieu BOESPFLUG, Quentin CARBONNEAUX et Olivier HERMANT : The λΠ-calculus modulo as a universal proof language. *In* David PICHARDIE et Tjark WEBER, éditeurs : *PxTP*, 2012.

[9] Mathieu BOESPFLUG, Maxime DÉNÈS et Benjamin GRÉGOIRE : Full reduction at full throttle. *Lecture notes in computer science*, 2011.

[10] Thierry COQUAND : An algorithm for type-checking dependent types. *Sci. Comput. Program.*, 26(1-3):167–177, 1996.

[11] Denis COUSINEAU et Gilles DOWEK : Embedding pure type systems in the λΠ-calculus modulo. *In* Simona Ronchi Della ROCCA, éditeur : *TLCA*, volume 4583 de *LNCS*, pages 102–117. Springer, 2007.

[12] Gilles DOWEK, Thérèse HARDIN et Claude KIRCHNER : Theorem proving modulo. *Journal of Automated Reasoning*, 31:2003, 1998.

[13] Cormac FLANAGAN, Amr SABRY, Bruce F. DUBA et Matthias FELLEISEN : The essence of compiling with continuations. *SIGPLAN Not.*, 39(4):502–514, avril 2004.

[14] R. IERUSALIMSCHY, L. H. de FIGUEIREDO et W. CELES : *Lua 5.1 Reference Manual*, août 2006.

[15] S. LINDLEY : *Normalisation by evaluation in the compilation of typed functional programming languages*. Thèse de doctorat, 2005.

[16] Luc MARANGET : Compiling pattern matching to good decision trees. *In Proceedings of the 2008 ACM SIGPLAN workshop on ML*, ML '08, pages 35–46, New York, NY, USA, 2008. ACM.

[17] L. S. van BENTHEM JUTTING : Typing in pure type systems. *Inf. Comput.*, 105(1):30–41, juillet 1993.