Master Parisien de Recherche en Informatique
Rapport de stage de M2

Chantal Keller
Université Paris VII

Importation de preuves HOL-Light en Coq

Chantal Keller
Encadrant: Benjamin Werner, équipe TypiCal, INRIA Saclay

17 février 2009 - 24 juillet 2009

Synthèse

Contexte général

Les assistants de preuve permettent d'élaborer et de vérifier formellement des preuves de théorèmes mathématiques. Chaque assistant de preuve a ses spécificités, qui lui permettent d'être plus efficaces (présence d'outils, automatisation...) dans certains domaines que d'autres assistants de preuve.

L'interaction entre assistants de preuves peut donc permettre d'utiliser des théorèmes "aisément" prouvés dans un assistant de preuve donné, dans un autre assistant de preuve dans lequel ils auraient été plus difficiles à prouver. Elle présente d'autres intérêts :

- prouver la correction d'un assistant de preuve, en en établissant un modèle ;
- étudier les traductions entre différents systèmes de preuve, comme la logique d'ordre supérieur et le calcul des constructions.

Différentes approches pour faire interagir différents assistants ont déjà été proposées. Ewen Denney [Den00] présente un moyen générique de traduction de preuves d'un système dans un autre, en utilisant une machine abstraite intermédiaire. Il utilise son système pour importer des preuves réalisées en HOL98 dans Coq. De manière moins générale, Steven Obua [OS06] décrit un système pour importer automatiquement des preuves de HOL-Light dans Isabelle/HOL et Freek Wiedijk [Wie07] montre comment traduire des preuves de HOL-Light dans Coq.

Problème étudié

On se propose ici de réaliser une interface permettant d'importer en Coq les preuves réalisées en HOL-Light. Ce problème particulier a déjà été étudié par Freek Wiedijk [Wie07]. À l'aide d'un encodage peu profond, il prouve en Coq les règles logiques de base de HOL-Light, puis montre comment traduire une preuve de HOL-Light en preuve Coq de manière non automatisée. Les preuves de HOL-Light pouvant utiliser jusqu'à plusieurs centaines de règles logiques, cela peut produire en Coq de très grandes preuves, ce qui induit les problèmes suivants :

– il est impossible de les produire de manière non automatisée ;
– l'encombrement mémoire des preuves en Coq est important ;
– les preuves sont très longues à vérifier en Coq.

Notre approche cherche à résoudre ces problèmes.

Contribution proposée

L'essentiel de notre travail consiste à établir un modèle de HOL-Light en Coq. Cette modélisation diffère de celle de [Wie07] par les points suivantes :

– nous définissons d'abord un plongement profond de HOL-Light dans Coq, puis traduisons ensuite les termes de notre encodage de HOL-Light vers des termes Coq ;
– nous utilisons une démarche réflexive.

Nous adaptons également le système d'exportation de preuves défini par Steven Obua pour HOL-Light [OS06].

Validité

Bien qu'il ne soit pas encore terminé et qu'il puisse être largement amélioré sur le plan de l'efficacité, notre travail permet déjà d'importer automatiquement en Coq le cœur logique de HOL-Light, ainsi que le début de sa bibliothèque (soit une soixantaine de théorèmes et une dizaine de définitions), avec les avantages suivants :

– le procédé est entièrement automatisé, avec la génération de code Coq et d'un Makefile ;
– grâce au partage de preuves et à la réflexion, la taille des fichiers Coq générés ainsi que le temps de compilation de ces fichiers sont assez faibles.

Ce travail ne passe cependant pas encore à l'échelle, pour des problèmes de mémoire et d'efficacité.

Notre technique est fortement dépendante des assistants de preuve source et cible ; cependant, elle offre une vision théorique de la manière de procéder, en se basant par exemple sur un encodage profond traduit vers un encodage peu profond.

Perspectives

Ce travail est toujours en cours, puisqu'il faut encore parvenir à le faire passer à l'échelle, tout en améliorant l'efficacité.

Il offre de nombreuses perspectives. À court terme, le modèle de la logique d'ordre supérieur en Coq n'est pas très dépendant de la présentation qu'en fait HOL-Light. À long terme, cela fournit une méthode pour interagir entre différents assistants de preuve, voire entre assistants de preuve et autres systèmes plus éloignés comme les outils de démonstration automatique.

Table des matières

Remerciements Je tiens à remercier tout particulièrement mon encadrant, Benjamin Werner, pour sa gentillesse et sa disponibilité. Je remercie toutes les personnes qui m'ont aidée durant mon stage, en particulier François Garillot, Assia Mahboubi, Bruno Barras, Arnaud Spiwack..., et plus généralement toute l'équipe INRIA TypiCal pour son accueil chaleureux.

1 Présentation

1.1 Introduction

Les assistants de preuve sont des logiciels permettant d'écrire et de vérifier des preuves de théorèmes mathématiques, de manière formelle. L'écriture et la recherche de preuves sont parfois fastidieuses :

- on doit détailler toutes les étapes, mêmes celles qui paraissent évidentes sur le papier ;
- un assistant de preuve donné n'est pas forcément bien équipé pour certains domaines des mathématiques.

Certains assistants de preuve intègrent des outils automatiques qui permettent de résoudre de manière plus ou moins efficace les problèmes liés au premier point, sans pour autant arriver à une automatisation complète.

Pour pallier cet inconvénient, il peut être utile de faire appel à des outils extérieurs pour résoudre les problèmes pour lesquels l'assistant de preuve n'est pas efficace. Par exemple, l'intégration de certains outils automatiques comme des solveurs SAT peut permettre d'augmenter l'automatisation de l'assistant de preuve.

Dans ce rapport, nous ne nous intéresserons pas à l'intégration d'outils automatiques à proprement parler, mais à la réalisation d'une interface entre assistants de preuve. L'intérêt de cet étude est multiple :

- utiliser des théorèmes qui ont été montrés dans un assistant de preuve donné dans d'autres assistants de preuve ;
- réaliser des modèles d'assistants de preuve et étudier les rapports entre deux systèmes de preuve ;
- prouver la cohérence des assistants de preuve ;
- à long terme, avoir une base de données de tous les théorèmes prouvés dans différents assistants de preuve, afin de faciliter la formalisation de nouveaux théorèmes.

Deux points sont essentiels dans l'intégration d'outils extérieurs à un assistant de preuve :

- il ne faut pas compromettre la cohérence de cet assistant de preuve : on ne doit pas ajouter d'axiome qui le rend inconsistant. Pour cela, la méthode que nous utilisons est de rejouer la preuve fournie par l'assistant source dans l'assistant cible, afin d'être sûr qu'elle est correcte pour l'assistant cible ;
- il faut que les théorèmes qui ont été automatiquement importés soient "utilisables" : par exemple, il est préférable qu'ils utilisent la syntaxe et les objets qui auraient été utilisés s'ils avaient été directement écrits dans cet assistant de preuve.

Nous étudierons le cas particulier de l'importation de preuves HOL-Light en Coq.

Ce travail a fait l'objet d'un exposé lors de la réunion Types 2009 [KW09].

1.2 Plan

Dans la partie 2, nous exposerons l'idée générale de notre approche, et introduirons les notions de HOL-Light et de Coq dont nous aurons besoin par la suite. La partie 3 sera consacrée à la réalisation d'un modèle de HOL-Light en Coq. Dans la partie 4, nous détaillerons notre implantation et ses résultats. Enfin, dans la partie 5, nous discuterons des perspectives de notre travail.

1.3 Notations

Dans toute la suite, les types simples du λ-calcul seront notés en majuscules, et les termes en minuscules. Nous utiliserons la notation de Krivine pour gérer les parenthèses lors de l'application. Ainsi, l'application de a à b sera notée $(a)\ b$, et on aura implicitement : $(a)\ b\ c\ d = (((a)\ b)\ c)\ d$.

Le code OCaml sera présenté sur fond jaune, le code Coq sur fond bleu, et le code shell sur fond rouge.

1.4 Tests

Dans ce qui suit, tous les tests ont été réalisés sur un ordinateur portable Samsung Q310 avec 3Go de mémoire vive et un processeur Intel Core 2 Duo P8400 / 2,26 GHz, sous le système d'exploitation Linux Debian Sid.

2 HOL-Light et Coq

L'idée générale de notre travail est de pouvoir importer et vérifier des preuves de HOL-Light en Coq, suivant le principe de la FIG. 1. Nous allons nous placer dans une version de HOL-Light incluant un système d'enregistrement de preuves. Lorsqu'un théorème est prouvé dans HOL-Light, sa preuve est enregistrée, et peut être exportée dans un fichier Coq (extension `.v`) (voir paragraphe 2.3). En se plaçant ensuite dans Coq auquel ont été ajouté des axiomes classiques ainsi que l'extensionalité, on peut, en même temps, générer une proposition Coq et la prouver à partir du terme de preuve qui a été importé (voir partie 3).

2.1 HOL-Light

Le langage source est HOL-Light, un assistant de preuve écrit par John Harrison et al. Il implante la logique classique d'ordre supérieur, présentée de manière équationnelle suivant le principe de la théorie des types basée sur l'égalité telle que décrite dans la partie II.2 de [LS88]. Contrairement à HOL, HOL-Light possède un noyau logique très succinct et très petit, ce qui le rend en même temps plus sûr (moins il y a de ligne de code, moins il y a

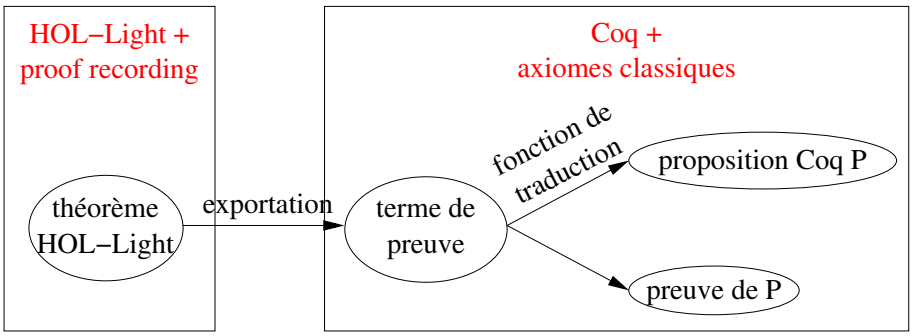

FIG. 1 – Principe

de bugs, et plus il sera facile de les vérifier) et plus accessible (on peut entrer très facilement dans ces quelques lignes de code bien commentées).

HOL-Light est écrit en OCaml [Ler98], mais n'est pas compilé : une session HOL-Light tourne dans un top-level OCaml, et tout le noyau de HOL-Light est réinterprété à chaque ouverture d'une session. Cela contribue à garantir sa cohérence (voir paragraphe 2.1.1).

Un des intérêts de HOL-Light pour l'utilisateur est d'être programmable en garantissant la cohérence du système : on peut créer de nouveaux termes et de nouveaux types en étant certain que cela ne rend pas HOL-Light incohérent. HOL-Light possède également de nombreux outils automatiques, et notamment des tactiques, qui améliorent sa facilité d'utilisation. De fait, il dispose d'une bibliothèque assez étendue, notamment en analyse.

HOL-Light possède la particularité de ne pas garder trace des preuves d'un théorème : lorsqu'une règle logique est appliquée, cela crée un nouveau théorème, dont on oublie la provenance.

2.1.1 Cohérence

La cohérence de HOL-Light est garantie notamment par deux caractéristiques :

- un théorème est un objet OCaml de type `thm`, qui est un type **abstrait**. Ainsi, un utilisateur ne peut construire un théorème qu'en appliquant les règles de HOL-Light, seules fonctions OCaml dont le type de retour est `thm` ; et ces règles sont cohérentes[1] ;
- les preuves ne sont pas enregistrées, donc ne sont pas vérifiables à chaque instant, mais cela est contrebalancé par le fait que HOL-Light est **réinterprété** à chaque ouverture de session, donc les théorèmes

[1] Exceptée la fonction `axiom : term -> thm`, qui permet d'ajouter à l'ensemble des théorèmes n'importe quelle proposition ; la cohérence peut donc être remise en jeu à chaque utilisation de cette fonction.

sont à chaque fois nécessairement construits à partir des règles comme indiqué dans le premier point.

2.1.2 Système logique

Les termes de HOL-Light sont ceux du λ-calcul simplement typé avec constantes de types et de termes, et variables de types. Les constantes peuvent être classifiées selon la TAB. 1, dans laquelle bool désigne le type des propositions, $\to$ celui des fonctions (en notation infixe), num celui des entiers ; $\mathbb{1}$ désigne l'ensemble singleton ; et les constantes de termes (dont la notation est standard, ε désignant l'opérateur de choix de Hilbert) ont le type suivant :

$=$	:	$A \to A \to \text{bool}$	$\Rightarrow$	:	$\text{bool} \to \text{bool} \to \text{bool}$	$\forall$	:	$(A \to \text{bool}) \to \text{bool}$
ε	:	$(A \to \text{bool}) \to A$	$\exists$	:	$(A \to \text{bool}) \to \text{bool}$	$\vee$	:	$\text{bool} \to \text{bool} \to \text{bool}$
$\wedge$	:	$\text{bool} \to \text{bool} \to \text{bool}$	$\bot$	:	bool	$\neg$	:	$\text{bool} \to \text{bool}$
$\top$	:	bool						

où A est une variable de type. Les constantes de base préexistent, alors que les constantes définies proviennent de définitions de types ou de termes.

Constante	de type	de terme
de base	bool, $\to$	$=$, ε
définie	$\mathbb{1}$, num...	$\top, \wedge, \Rightarrow, \forall, \exists, \vee, \bot, \neg \ldots$

TAB. 1 – Constantes de HOL-Light

Le polymorphisme provient des variables de type, qui sont implicitement universellement quantifiées. Ainsi, la constante $=$ est bien polymorphique, A pouvant être remplacé par n'importe quel type. Voici un exemple de terme : $(\forall)\ \lambda x : A.\ (\exists)\ \lambda y : A.\ (=)\ x\ y$, qui peut de manière usuelle se noter $\forall x : A.\ \exists y : A.\ x = y$, et qui a le type bool.

Toutes les constantes de types de HOL-Light sont habitées. En particulier, lorsqu'on définit une nouvelle constante de type, on doit donner un élément de ce type (voir paragraphe 2.2).

Un théorème est un terme de type bool sous l'hypothèse d'autres termes de type bool. Nous allons par la suite utiliser une présentation de déduction naturelle, et un théorème sera donc par exemple noté : $p, q \vdash p \wedge q$. Le système

logique de HOL-Light est défini à partir des dix règles d'inférence suivantes :

$$\frac{}{\vdash t = t}\ \text{REFL} \qquad \frac{}{\{p\} \vdash p}\ \text{AS} \qquad \frac{}{\vdash (\lambda x.t)\ x = t}\ \text{BETA} \qquad \frac{\Gamma \vdash s = t \qquad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\ \text{TRANS}$$

$$\frac{\Gamma \vdash p \Leftrightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}\ \text{EQ\_MP} \qquad \frac{\Gamma \vdash p \qquad \Delta \vdash q}{(\Gamma - \{q\}) \cup (\Delta - \{p\}) \vdash p \Leftrightarrow q}\ \text{DEDUCT\_ANTISYM}$$

$$\frac{\Gamma \vdash s = t}{\Gamma \vdash (\lambda x.s) = (\lambda x.t)}\ \text{ABS} \qquad \frac{\Gamma \vdash s = t \qquad \Delta \vdash u = v}{\Gamma \cup \Delta \vdash s\ u = t\ v}\ \text{MK\_COMB}$$

$$\frac{\Gamma(x_1, \ldots, x_n) \vdash p[x_1, \ldots, x_n]}{\Gamma(t_1, \ldots, t_n) \vdash p[t_1, \ldots, t_n]}\ \text{INST} \qquad \frac{\Gamma(\alpha_1, \ldots, \alpha_n) \vdash p[\alpha_1, \ldots, \alpha_n]}{\Gamma(\gamma_1, \ldots, \gamma_n) \vdash p[\gamma_1, \ldots, \gamma_n]}\ \text{INST\_TYPE}$$

où x ne doit pas apparaître libre dans Γ pour la règle ABS. La règle INST est une règle de substitution pour les variables de terme, et la règle INST_TYPE une règle de substitution pour les variables de type. L'extensionalité provient de l'axiome ETA et le côté classique de l'axiome SELECT :

$$\frac{}{\forall t : A \to B.\ \lambda x.(t)\ x = t}\ \text{ETA} \qquad \frac{}{\forall P.\ \forall x : A.\ (P)\ x \Rightarrow (P)\ (\varepsilon)\ P}\ \text{SELECT}$$

Les règles logiques traitent donc de l'égalité, et non des connecteurs logiques. Comme expliqué ci-dessus, ces derniers sont des constantes définies les unes à partir des autres ; voici quelques exemples :

$$\begin{array}{lcllcl} \top & \triangleq & (\lambda p.p) = (\lambda p.p) & \wedge & \triangleq & \lambda pq.(\lambda f.(f)\ p\ q) = (\lambda f.(f)\ \top\ \top) \\ \Rightarrow & \triangleq & \lambda pq.p \wedge q = p & & \ldots & \end{array}$$

Les règles usuelles d'introduction et d'élimination des connecteurs de la déduction naturelle sont donc dérivées à partir des définitions des connecteurs et des dix règles de base.

2.2 Définitions de termes et de types

Comme expliqué ci-dessus, un utilisateur peut définir ses termes et ses types sans changer la cohérence.

Le fonctionnement des définitions de termes est très simple : si l'utilisateur souhaite associer le λ-terme t (qui peut être arbitrairement grand) au nom "myT", alors le système va créer le nouveau théorème : $\vdash \text{myT} = t$ sous réserve que "myT" ne soit pas déjà un nom réservé.

On peut définir de nouveaux types en **compréhension** : étant donnés un type A et une propriété P sur ce type, on peut associer au nom "myType" l'ensemble $\{x : A \mid (P)\ x\}$. Comme expliqué au paragraphe 2.1.2, toutes les constantes de types doivent être habitées (afin d'assurer la cohérence), ce qui revient à demander l'existence d'un élément de A vérifiant la propriété P.

Lorsqu'on fait une telle définition de type, le système ajoute la nouvelle constante de type "myType" (sous réserve que le nom ne soit déjà réservé), deux nouvelles constantes de termes "mk_myType" et "dest_myType"[2] ayant les types suivants :

$$\text{mk\_myType} \;:\; A \to \text{myType} \qquad\qquad \text{dest\_myType} \;:\; \text{myType} \to A$$

et fournit deux nouveaux théorèmes :

$$\begin{array}{ll} \vdash & (\text{mk\_myType})\ (\text{dest\_myType})\ a = a \\ \vdash & ((P)\ r) = ((\text{dest\_myType})\ (\text{mk\_myType})\ r = r) \end{array}$$

où a et r sont des variables de termes.

2.3 Enregistrement et exportation des preuves HOL-Light

Comme expliqué ci-dessus, les termes de preuve ne sont pas conservés en HOL-Light. Cependant, nous avons besoin de les enregistrer puis d'en exporter une trace à vérifier dans Coq. Nous reprenons pour cela le système de Steven Obua [OS06] en le transformant légèrement pour exporter des traces non pas au format XML, mais comme objets Coq de type proof :

```
Inductive proof : Type :=
| Prefl : term → proof
| Pbeta : idV → type → term → proof
| Pinstt : proof → list (idT * type) → proof
...
```

qui est un inductif représentant les arbres de dérivation (pour plus de précisions, se rapporter à l'annexe A).

2.4 Coq

Le langage cible est Coq, un assistant de preuve implantant le calcul des constructions inductives [Wer94]. Il est écrit en OCaml et compilé.

Si Coq apporte plus de facilités que HOL-Light au niveau de la syntaxe et de l'écriture des théorèmes et des preuves, il est moins automatisé. Il possède une bibliothèque extrêmement étendue, mais cependant moins étendue quant à l'analyse. C'est une des raisons pour lesquelles il est très utile de pouvoir importer celle de HOL-Light.

Une particularité de Coq qui le rend efficace est le fait de distinguer calcul et preuve. En Coq, on peut définir des fonctions qui effectuent du calcul, comme par exemple l'addition :

$$\begin{array}{rcll} 0 & + & y & \longrightarrow\ y \\ (S)\ x & + & y & \longrightarrow\ (S)\ (x + y) \end{array}$$

[2] En pratique, ce sont les utilisateurs qui choisissent les noms de ces constantes, et le système vérifie qu'ils ne sont pas déjà attribués.

ici définie par récurrence sur les entiers de Peano. $\longrightarrow$ traduit le fait que cette opération est du calcul. Ainsi, comme $(S)\ (S)\ (S)\ 0\ +\ (S)\ (S)\ (S)\ (S)\ 0$ est **convertible** à $(S)\ (S)\ (S)\ (S)\ (S)\ (S)\ (S)\ 0$, une preuve que $(S)\ (S)\ (S)\ 0\ +\ (S)\ (S)\ (S)\ (S)\ 0\ =\ (S)\ (S)\ (S)\ (S)\ (S)\ (S)\ (S)\ 0$ est juste une preuve que $=$ est **réflexif**. On a donc une preuve très courte dont la taille ne dépend pas des entiers additionnés.

Ce principe est appelé *preuve par réflexion*.

2.4.1 Cohérence

À la différence de HOL-Light, les preuves des théorèmes sont enregistrées en Coq. C'est de là que provient la cohérence de Coq : lorsqu'une bibliothèque Coq est compilée, toutes les preuves sont gardées dans le fichier compilé, et peuvent donc être vérifiées à tout instant par n'importe quel outil, en particulier par des outils externes dont le noyau peut être assez simple.

2.4.2 Système logique

Le système logique de Coq est intuitionniste et non extensionnel, contrairement à celui de HOL-Light. Cependant, si on se donne certains axiomes comme l'opérateur de choix de Hilbert et l'extensionnalité, le système logique de Coq devient une extension de celui de HOL-Light. On peut donc y montrer la cohérence de HOL-Light.

2.4.3 Ssreflect

Dans notre développement, nous utilisons l'extension de Coq ssreflect [GM08], qui est une bibliothèque en cours de développement apportant des facilités pour gérer la réflexion, ainsi qu'au niveau des notations. Nous utiliserons en particulier les modules `ssreflect`, la base de l'extension ; `eqtype`, implantant les types ayant une égalité décidable ; et `ssrbool` et `ssrnat`, permettant de voir les booléens et les entiers comme des `eqType`.

3 Un modèle de HOL-Light en Coq

Nous allons maintenant exposer comment nous avons réalisé un modèle de HOL-Light en Coq, et comment celui-ci est relié à l'inductif proof ci-dessus défini. Nous allons donc modéliser les types et les termes de HOL-Light, ainsi que les règles de dérivation. Mais tout d'abord, nous devons définir comment construire notre modèle.

3.1 Plongements

Le but de notre travail est de plonger le système logique de HOL-Light dans celui de Coq. Deux possibilités s'offrent alors à nous :

– définir un plongement **profond**, dans lequel les types et les termes de HOL-Light sont représentés par des structures de données Coq (inductifs) ; par exemple :

```
Inductive type : Type := ...
Inductive term : Type := ...
```

Le plongement profond possède l'avantage que l'on a accès à la **structure** des objets que l'on manipule, ce qui permet par exemple de raisonner sur cette structure (raisonnement par induction) ;

– définir un plongement **peu profond**, dans lequel les types et les termes de HOL-Light sont représentés par des types et des termes Coq ; par exemple :

```
Definition type := Type.
Definition coq_bool : type := Prop.
Definition coq_fun (A B: type) : type := A → B.
```

Le plongement peu profond a l'avantage de pouvoir utiliser les facilités de Coq : par exemple, il n'y a pas besoin d'écrire un normaliseur pour les termes, car c'est celui de Coq qui les normalise.

Comme on souhaite obtenir *in fine* des propositions Coq, le plongement peu profond est nécessaire. Dans son développement [Wie07], Freek Wiedijk avait utilisé directement un plongement peu profond.

Cependant, dans notre travail, et en particulier pour pouvoir procéder par réflexion (voir paragraphe 2.4), il est important de pouvoir accéder à la structure des termes que l'on manipule. De plus, utiliser un plongement profond permet généralement d'avoir des termes plus compacts.

C'est pourquoi nous allons d'abord définir des types et des termes sous forme d'inductifs, puis traduire ces inductifs vers les types et les termes de Coq afin d'obtenir des propositions Coq. Pour cela, nous allons adapter la partie compilation (paragraphe 5) de la normalisation par évaluation de François Garillot et Benjamin Werner [GW07]. L'idée de reprendre ce développement pour réaliser le modèle de HOL-Light en Coq a été imaginée par Carlos Simpson.

3.2 Structures de données pour le plongement profond

3.2.1 Noms des variables et des constantes

Nous avons besoin de variables de types et de termes nommées, ainsi que de noms pour les définitions de types et de termes. Ces noms doivent appartenir à un ensemble infini dénombrable (pour pouvoir choisir des noms frais) ayant une égalité décidable (pour pouvoir décider l'égalité des types et des termes) : c'est pourquoi un eqType quelconque peut convenir (sur le plan théorique ; pour des raisons d'efficacité, il est préférable d'avoir un type sur lequel l'égalité est rapide à décider).

```
Definition  idT : eqType.    (*    variables de  types *)
Definition defT : eqType.    (* definitions de  types *)
Definition  idV : eqType.    (*    variables de termes *)
Definition defV : eqType.    (* definitions de termes *)
```

3.2.2 Types

Les types sont définis de la même manière que pour HOL-Light, en distinguant bool, num, et $\rightarrow$ des autres définitions de types. bool et $\rightarrow$ jouent un rôle particulier en étant des constantes de type de base, et num est à part car on souhaite le traduire vers le type des entiers de Coq (nat) quels que soient les choix de l'utilisateur.

Chaque définition de type ayant sa propre arité, à une définition de type est associée la liste de ses arguments. C'est pourquoi le type `type` est mutuellement récursif avec les listes de `type`. En revanche, une définition de type ne comporte pas sa version expansée : rappeler la définition de ce type à chaque utilisation serait une perte de temps et de mémoire, et on perd l'intérêt de pouvoir utiliser un nom court à la place d'une grande définition :

```
Inductive type : Type :=
| TVar : idT → type | TDef : defT → list_type → type
| Bool : type | Num : type | Arrow : type → type → type
with list_type : Type :=
| Tnil : list_type
| Tcons : type → list_type → list_type.
```

Traitement de l'égalité Lorsque l'on veut comparer deux types A et B, on ne souhaite pas seulement savoir s'ils sont égaux ou non, mais également avoir une fonction de coercition de l'un vers l'autre : en effet, lorsque $A == B$, il faudra pouvoir transformer un terme de type A en un terme de type B. C'est pourquoi nous avons repris le développement de F. Garillot et B. Werner [GW07] pour gérer l'égalité des types.

3.2.3 Termes

Comme pour les types, où nous avons distingué bool, num et $\rightarrow$ des autres constantes, nous allons distinguer certaines définitions de termes : c'est le rôle de l'inductif `cst`.

```
Inductive cst : Type :=
| Heq : type → cst | Heps : type → cst | Hand : cst
| Hor : cst | Hnot : cst | Himp : cst | Htrue : cst
| Hfalse : cst | Hforall : type → cst
| Hexists : type → cst.
```

Nous considérons comme constantes de base $=$ et ε, comme dans HOL-Light, ainsi que l'ensemble des connecteurs logiques.

En HOL-Light, les termes du λ-calcul avec constantes sont définis en utilisant des variables nommées. En Coq, il est plus simple de raisonner à l'aide d'une représentation "locally nameless" [ACP+08] :

```
Inductive term : Type :=
| Dbr : nat → term          | Var : idV → type → term
| Cst : cst → term          | Def : defV → type → term
| App : term → term → term  | Abs : type → term → term.
```

Les définitions de termes contiennent juste un nom et le type du λ-terme représenté par ce nom. Le type est nécessaire pour pouvoir typer aisément les termes. Ici encore, le λ-terme représenté par le nom peut être arbitrairement grand, et ce serait une grande perte d'efficacité que de le mettre dans la représentation de la définition de terme.

3.2.4 Typage

Les règles de typage sont les règles habituelles de typage pour le λ-calcul simplement typé. On a besoin d'un contexte pour typer les indices de De Bruijn, qui consiste en une liste de types. On définit tout d'abord une fonction de typage `infer`, qui renvoie `None` si le terme passé en argument est mal typé, et `Some ty` s'il est bien typé de type `ty`. Le prédicat `wt` représente les jugements de typage.

```
Definition context := list type.
Fixpoint infer (g: context) (t: term) : option type :=...
Definition wt (g: context) (te: term) (ty: type) : Prop
  := infer g te = Some ty.
```

3.3 Traduction du plongement profond vers le plongement peu profond

L'étape suivante est d'établir la **sémantique** des types et des termes (bien typés) ci-dessus définis, à l'aide de deux fonctions de traduction. Ce sont ces fonctions qui permettent de passer du plongement profond au plongement peu profond. Nous les noterons $|\bullet|_{\mathcal{I}}$, où $\mathcal{I}$ désigne un ensemble de fonctions d'interprétation (voir fin du paragraphe). Par extension, si l est la liste de types $[T_1; \ldots; T_n]$, alors $|l|_{\mathcal{I}} \triangleq [|T_1|_{\mathcal{I}}; \ldots; |T_n|_{\mathcal{I}}]$.

Comme évoqué au paragraphe 2.1.2, tous les types HOL-Light doivent être habités (pour pouvoir définir l'opérateur de choix de Hilbert). En conséquence, les types Coq images de types HOL-Light par $|\bullet|_{\mathcal{I}}$ doivent être habités. Le type de retour de $|\bullet|_{\mathcal{I}}$ n'est donc pas **Type**, mais un enregistrement contenant un **Type** `ttrans` ainsi qu'un élément de `ttrans` :

```
Record type_translation : Type :=
  mkTT { ttrans :> Type; tinhab : ttrans}.
```

Les fonctions d'interprétation servent à interpréter les variables et les définitions de types et de termes, ainsi que les indices de De Bruijn :

```
Definition type_context := idT → type_translation.
Definition type_definition_tr :=
  defT → list Type → type_translation.
Definition term_context :=
  idV → forall (A: type), tr_type tc tdt A.
Definition term_definition_tr := ...
Definition tr_context (g: context) : Type := ...
```

Elles sont nécessaires à la définition des fonctions de traduction. Celles qui concernent les définitions de types et de termes permettent également de favoriser la facilité d'utilisation des théorèmes HOL-Light importés en Coq. Par exemple, si un utilisateur définit la constante de type rat pour travailler sur les nombres rationnels, il peut choisir de la traduire par `Q`, l'ensemble des nombres rationnels de Coq défini par la bibliothèque `QArith`. Il pourra donc, en Coq, utiliser aussi bien les lemmes déjà prouvés en Coq que ceux importés de HOL-Light.

Dans toute la suite, la liste des fonctions d'interprétation sera abrégée par [...] .

3.3.1 Types

La traduction des types est immédiate par induction :

$$|A|_{\mathcal{I}} \triangleq \mathcal{I}(A) \qquad |\text{num}|_{\mathcal{I}} \triangleq \text{nat} \qquad |T \to S|_{\mathcal{I}} \triangleq |T|_{\mathcal{I}} \to |S|_{\mathcal{I}}$$
$$|C\ l|_{\mathcal{I}} \triangleq \mathcal{I}(C)\ |l|_{\mathcal{I}} \qquad |\text{bool}|_{\mathcal{I}} \triangleq \text{Prop}$$

où A désigne une variable de type, C une définition de type, l une liste de types, et T et S deux types.

```
Fixpoint tr_type [...] (ty: type) : Type := ...
```

3.3.2 Termes

Les types vont permettre de définir une interface entre syntaxe et sémantique pour la traduction des termes (bien typés). L'idée est la suivante : étant donnés un terme t et un type T, si t a le type T, alors la traduction de t est un élément de la traduction de T :

$$\text{si } \Gamma \vdash t : T, \text{ alors } |t|_{\mathcal{I}} \in |T|_{\mathcal{I}}$$

où Γ désigne le contexte de De Bruijn pour t.

Une telle fonction de traduction peut être définie grâce aux types dépendants : prenant en arguments Γ et t, elle renvoie la paire (dépendante) formée de T et d'un élément de $|T|_{\mathcal{I}}$. Elle se définit (non trivialement) de manière inductive sur t.

```
Fixpoint sem_term (g: context) (t: term) : option
  {ty: type & forall [...], tr_type [...] ty} := ...
```

3.4 Dérivations

Il faut maintenant définir les règles de dérivations de HOL-Light et prouver leur correction dans notre modèle.

3.4.1 Règles de dérivations

Pour cela, nous allons tout d'abord définir un inductif :

```
Inductive deriv : hyp_set → term → Prop := ...
```

où hyp_set est un type désignant les ensembles de termes. Cet inductif comprend un constructeur par règle de dérivation considérée (règles de base et règles d'introduction et d'élimination), avec les particularités suivantes :

- il n'y a pas de constructeur pour les règles INST et INST_TYPE (voir paragraphe 2.1.2), car elles sont dérivables dans notre modèle. Cela provient du fait que les substitutions sont des fonctions, définies de manière calculatoire et ayant un certain nombre de propriétés (comme le lemme de substitution), et non des objets de première classe comme en HOL-Light ;
- une règle d'affaiblissement est ajoutée ;
- les règles doivent être traduites pour une syntaxe locally nameless, notamment en utilisant la "cofinite quantification". Par exemple, la règle ABS se traduit de la manière suivante :

$$\frac{\forall x \notin L, \Gamma \vdash s^x = t^x}{\Gamma \vdash (\lambda s) = (\lambda t)} \text{ ABS'}$$

  où L est une liste finie de noms de variables et pour tous termes u et v, u^v désigne le terme u dans lequel l'indice de De Bruijn 0 a été remplacé par v.

Cela suffit à pouvoir traduire les termes de preuve provenant de l'exportation de HOL-Light (voir paragraphe 2.3) : pour tout objet de type proof **provenant d'une dérivation HOL-Light correcte**, on peut trouver h un ensemble de termes et t un terme tels que deriv h t est une proposition Coq correcte. Si l'objet de type proof ne provient pas d'une dérivation HOL-Light correcte (par exemple, si la condition de non liberté de la variable x pour la règle ABS au paragraphe 2.1.2 n'est pas vérifiée), alors ont ne peut pas déterminer de tels h et t.

Ceci est implanté par la fonction :

```
Fixpoint proof2deriv (p: proof) :
  option (hyp_set * term) := ...
```

qui :

- renvoie None si p ne provient pas d'une dérivation HOL-Light correct ;
- renvoie Some (h,t) dans le cas contraire. On peut alors prouver la propriété suivante :

```
Lemma proof2deriv_correct : forall p,
  match proof2deriv p with
    | Some (h, t) ⇒ deriv h t
    | None ⇒ True
  end.
```

3.4.2 Correction

Le point crucial de notre développement est d'énoncer et de prouver que ces règles de dérivations sont correctes vis-à-vis de notre sémantique. Pour cela, définissons tout d'abord ce qu'est une proposition HOL-Light correcte vis-à-vis de la sémantique : c'est un terme ayant les propriétés suivantes :

- il est localement clos (il n'y a pas d'indice de De Bruijn non lié) ;
- il est de type bool ;
- sa traduction (qui, grâce au point précédent, existe et est une proposition Coq) est correcte.

Cela se traduit en Coq par l'énoncé suivant :

```
Definition has_sem [...] (t: term) : Prop :=
  match sem_term nil t with
    | Some (existT Bool evT) ⇒ evT [...]
    | _ ⇒ False
  end.
```

On étend la fonction has_sem aux ensembles de termes en la fonction sem_hyp. On peut maintenant énoncer le théorème fondamental :

```
Theorem sem_deriv : forall (h: hyp_set) (t: term),
  deriv h t → forall [...], sem_hyp [...] h → has_sem
      [...] t.
```

Sa preuve est non triviale.

Conclusion des parties 2 et 3 Notre approche réalise le schéma de la FIG. 1. Les preuves de HOL-Light sont exportées dans des fichiers Coq sous forme d'objets de type `proof`. À partir de ces objets, on peut construire en même temps :

- son arbre de dérivation précis (à l'aide de la fonction proof2deriv) ;
- le théorème Coq qui correspond au théorème HOL-Light (à l'aide de la fonction has_sem) ;
- la preuve de ce dernier (grâce au lemme proof2deriv_correct et au théorème fondamental).

Nous avons également réalisé un modèle de HOL-Light en Coq et, plus généralement, de la logique d'ordre supérieur, prouvé correct.

4 Implantation

Les sources de notre développement sont disponibles en ligne [1]. Elles se présentent comme suit :

- un fichier `README`, contenant les instructions nécessaires à la compilation (rappelées dans l'annexe B) ;
- un répertoire `model`, contenant les fichiers Coq implantant le modèle de HOL-Light, ainsi qu'un Makefile pour les compiler (le détail du contenu de ces fichiers Coq est expliqué dans l'annexe C) ;
- un répertoire `hol_light`, contenant une partie du cœur de la version de développement de HOL-Light[3], pour pouvoir effectuer les tests.

4.1 Enregistrement et exportation

L'enregistrement et l'exportation des preuves HOL-Light sont implantés dans le fichier `hol_light/proofobject_trt.ml`, tiré de celui de S. Obua. Les preuves sont enregistrées en utilisant le type proof imaginé par ce dernier, à l'exception que les types et les termes utilisent la syntaxe du modèle Coq (en particulier, les termes sont en locally nameless). La fonction `make_dependencies` gère le partage entre les morceaux de preuve, et la fonction `export_proof` permet d'exporter les preuves dans les fichiers Coq.

Pour chaque théorème portant le nom THM, on génère le code source Coq suivant :

```
Definition hollight_THM : proof := ...

Definition hollight_THM_deriv :=
  match (proof2deriv hollight_THM) with
    | Some (h,t) ⇒ (h,t)
    | None ⇒ (hyp_empty, htrue)
  end.

Definition hollight_THM_lemma_type := Eval vm_compute in
    let (h,t) := hollight_THM_deriv in deriv h t.

Lemma hollight_THM_lemma : hollight_THM_lemma_type.
Proof. exact (proof2deriv_correct hollight_THM). Qed.

Definition hollight_THM_thm_type :=
  Eval vm_compute in let (h,t) := hollight_THM_deriv in
    if ssrfsets.Empty h then has_sem se sdt t else False.

Theorem hollight_THM_thm : hollight_THM_thm_type.
Proof. exact (sem_deriv hollight_THM_lemma (sem_empty se
    sdt)). Qed.
```

avec les noms suivants :

- hollight_THM désigne le terme de preuve tel qu'il est exporté de HOL-Light ;

[3] HOL-Light, version de développement du 25 février 2009.

- hollight_THM_deriv désigne l'arbre de dérivation associé (si hollight_THM provient d'une preuve HOL-Light correcte, on doit nécessairement être dans le cas Some) ;
- hollight_THM_lemma désigne la preuve que cette dérivation est correcte ;
- hollight_THM_thm désigne le théorème Coq qui a été importé.

Il est important de constater que les preuves de hollight_THM_lemma et hollight_THM_thm sont bien des preuves par réflexion, et que tout le calcul a été effectué par l'appel aux fonctions proof2deriv et has_sem.

4.2 Résultats

Nous avons testé notre implantation à deux échelles :

- à petite échelle, sur le cœur de HOL-Light (comme dans les sources disponibles en ligne [1]) ;
- à grande échelle, sur l'ensemble de la bibliothèque standard de HOL-Light.

Le Tab. 2 récapitule le nombre de théorèmes exportés, le nombre de lemmes intermédiaires générés par le partage, la taille des fichiers exportés, et les durées d'enregistrement, d'exportation et de compilation des théorèmes Coq générés. On constate que si l'enregistrement et l'exportation passent bien à l'échelle (il faudrait arriver à réduire la taille des fichiers Coq générés), la compilation n'y passe pas encore. En effet, cette étape est très lente et très coûteuse en mémoire : au bout de quelques heures de compilation, on obtient l'erreur *Out of memory*. Cela peut être dû à des problèmes d'efficacité, notamment pour les structures de données utilisées. Se pose également le problème du nombre de théorèmes à mettre par fichier Coq : il faut trouver le bon équilibre entre un théorème par fichier, ce qui fait de petits fichiers mais avec beaucoup d'appels à d'autres fichiers ; et beaucoup de théorèmes par fichier, ce qui risque de faire des trop gros fichiers que Coq ne peut compiler.

Échelle	Nb. th.	Nb. lemmes	Taille	D. enr.	D. exp.	D. comp.
Petite	69	1213	0,7 Mo	4 s	1 s	21 s
Grande	1694	191620	921 Mo	3 min	7 min 30	X

Tab. 2 – Résultats

L'objectif consistant à obtenir des théorèmes utilisables dans Coq est atteint. Par exemple, le théorème HOL-Light CONJ_SYM : $\forall\, t_1\ t_2.\ t_1 \wedge t_2 \Leftrightarrow t_2 \wedge t_1$ est traduit en Coq par **forall** x x0 : **Prop**, (x ∧x0) = (x0 ∧x).

5 Conclusion

5.1 Résultats

Nous avons implanté un système permettant l'importation en Coq de preuves réalisées au moyen de HOL-Light. Nous l'avons testé sur une partie du cœur de HOL-Light avec succès, mais sans parvenir à le faire passer à l'échelle. Notre travail se révèle être :

- efficace pour l'enregistrement et l'exportation des preuves (rapide et peu coûteux en mémoire) ;
- utilisable en pratique : les théorèmes générés sont parfaitement lisibles par un utilisateur de Coq, et utilisent les structures de données Coq ;
- basé sur un modèle simple.

L'idée d'utiliser deux plongements, l'un profond et l'autre peu profond, pour pouvoir importer plus efficacement des preuves, notamment grâce à la réflexion, est utilisable de manière tout à fait générale dans l'interaction entre outils externes et assistants de preuve. En particulier, si l'outil externe implante la logique d'ordre supérieur, peu de modifications seront à apporter à notre développement.

5.2 Perspectives

Notre travail n'est pas encore complet, et notamment les tests qui ont été effectués sur l'ensemble de la bibliothèque standard de HOL-Light n'ont pas encore abouti à ce jour. En outre, beaucoup d'améliorations peuvent être apportées :

- la réalisation d'une interface pour pouvoir plus facilement exporter les preuves, choisir comment traduire les définitions de types et de termes... ;
- la génération automatique de documentation, pour connaître la traduction en Coq des théorèmes qui ont été importés ;
- l'amélioration de l'efficacité de l'exportation, pour obtenir des fichiers plus petits et qui compilent plus vite (plus de partage, termes de preuve "à trou", utilisation de types Coq sur lesquels on peut tester rapidement l'égalité...) ;

Sur un plan théorique, on peut également améliorer l'axiomatisation : les axiomes classiques peuvent tous être prouvés en supposant uniquement l'existence de l'opérateur de choix de Hilbert, ce qui n'est pour l'instant pas le cas (il y a plusieurs axiomes classiques dans le fichier `axioms.v`).

À long terme, notre développement peut être réutilisé pour réaliser une interface entre Coq et d'autres assistants de preuve comme HOL4. L'idée peut s'adapter à d'autres outils externes, comme des outils de preuve automatique. Ce travail va déboucher sur une thèse portant sur l'interaction entre solveurs SMT et Coq.

Références

[1] Sources de notre développement. `http://perso.ens-lyon.fr/chantal.keller/Documents-etudes/Stage09/hollightcoq.tar.gz`.

[2] Sources de la version de développement de coq 8.2. `svn://scm.gforge.inria.fr/svn/coq/branches/v8.2`.

[3] Sources de la version de développement de ssreflect pour coq 8.2. `svn://scm.gforge.inria.fr/svn/coq-contribs/branches/v8.2/Saclay/Ssreflect`.

[ACP+08] B. Aydemir, A. Charguéraud, B.C. Pierce, R. Pollack et S. Weirich : Engineering formal metatheory. *POPL*, 2008.

[Den00] E. Denney : A Prototype Proof Translator from HOL to Coq. *Harrison and Aagaard [HA00]*, pages 108–125, 2000.

[GM08] G. Gonthier et A. Mahboubi : A small scale reflection extension for the Coq system. *Rapport de recherche INRIA*, 2008.

[GW07] F. Garillot et B. Werner : Simple types in type theory : deep and shallow encodings. *Lecture notes in computer science*, 4732:368–382, 2007.

[KW09] C. Keller et B. Werner : Importing hol-light into coq. Types 2009 meeting, mai 2009. `http://www.lama.univ-savoie.fr/~types09/slides/types09-slides-49.pdf`.

[Ler98] X. Leroy : The OCaml Programming Language, 1998.

[LS88] J. Lambek et PJ Scott : *Introduction to higher order categorical logic*. Cambridge Univ Pr, 1988.

[OS06] S. Obua et S. Skalberg : Importing hol into isabelle/hol. *Lecture Notes in Computer Science*, 4130:298, 2006.

[Ska04] S. Skalberg : Porting proofs between hol implementations. Types 2004 meeting, dec 2004. `http://types2004.lri.fr/SLIDES/skalberg.pdf`.

[Wer94] B. Werner : *Une theorie des constructions inductives*. Thèse de doctorat, Université Denis-Diderot, 1994.

[Wie07] F. Wiedijk : Encoding the HOL Light logic in Coq. *Unpublished notes*, 2007.

A Enregistrement et exportation des preuves HOL-Light

Une phase essentielle de notre développement consiste à pouvoir obtenir une trace des preuves faites en HOL-Light dans Coq. Cette étape se décompose en deux parties :

– enregistrer les preuves des théorèmes HOL-Light ;
– les exporter dans un format lisible par Coq et de manière à ce que ce dernier puisse les traduire en théorèmes et en leurs démonstrations.

A.1 Enregistrement des preuves

A.1.1 Principe

Pour enregistrer les preuves, nous reprenons en majeure partie le travail que Steven Obua [OS06] a réalisé dans le but d'interfacer HOL-Light et Isabelle/HOL. Inspiré du travail sur l'enregistrement de preuves pour HOL4 de Sebastian Kalberg [Ska04], il est inclus dans la version de développement actuelle de HOL-Light[4].

Lorsqu'un théorème est créé dans une session HOL-Light, sa preuve est enregistrée en même temps. Les enjeux de l'enregistrement des preuves sont les suivants :

– avoir un système rapide (rappelons que les théorèmes, et donc l'enregistrement de leurs preuves, sont reconstruits à chaque démarrage de HOL-Light) ;
– avoir un système peu coûteux en mémoire (on a potentiellement un nombre élevé de théorèmes ayant des arbres de preuves très grands).

La solution choisie par S. Obua pour satisfaire ces deux contraintes est de changer la granularité du système logique de HOL-Light. Comme nous l'avons constaté au paragraphe 2.1.2, les règles d'inférence de HOL-Light sont peu nombreuses et ne comportent pas les règles usuelles d'introduction et d'élimination des connecteurs de la déduction naturelle. Ces dernières sont seulement des règles dérivées à partir des définitions des connecteurs. Au contraire, un utilisateur aura plutôt tendance à employer les règles d'introduction et d'élimination que les définitions mêmes des connecteurs, qui ne sont pas toujours intuitives (par exemple, la définition suivante peut ne pas paraître naturelle à un utilisateur non averti : $\exists \triangleq \lambda P.\ \forall q.\ (\forall x.\ (P)\ x \Rightarrow q) \Rightarrow q$).

L'idée est donc de ne pas limiter les règles d'inférence aux dix règles de base de HOL-Light, mais d'inclure également les règles d'introduction et d'élimination des connecteurs. Sans cela, à chaque fois qu'un utilisateur utilise une règle d'introduction ou d'élimination, c'est tout l'arbre de la dérivabilité de cette règle qui serait enregistré, et non juste un nœud.

Cela se traduit en OCaml par le type proof dont voici un extrait :

```
type proof =
  | Proof of (proof_info * proof_content * (unit → unit))
and proof_content =
  | Prefl of nterm
  | Pbeta of int * ntype * nterm
  | Pinstt of proof * (int * ntype) list
```

[4] HOL-Light, version de développement du 25 février 2009.

```
...
| Pconj of proof * proof
| Pconjunct1 of proof
| Pconjunct2 of proof
...
```

On a donc les règles de base de HOL-Light (ici, une partie), et les règles concernant les connecteurs logiques (ici, le $\wedge$). On constate également dans le type proof qu'une preuve est un arbre de dérivation (proof_content) ainsi que des informations qui serviront à l'exportation et que l'on n'exposera pas ici.

Un autre point essentiel pour limiter l'encombrement mémoire est le **partage**. Lorsqu'un utilisateur fait appel à un théorème déjà existant, il est plus astucieux, dans l'arbre de preuve, d'appeler ce théorème plutôt que de recopier la preuve dudit théorème. La description des arbres de dérivation ci-dessus permet d'ores et déjà de faire du partage : par exemple, lorsqu'on appelle le constructeur Pconjunct1, son argument peut être le nom d'un objet de type proof déjà enregistré. Ce partage servira également lors de l'exportation (voir partie A.2.1).

Enfin, on peut définir de nouveau théorèmes grâce aux définitions de types et de termes (voir paragraphe 2.2) ainsi qu'en ajoutant des axiomes ; on ajoute donc les constructeurs suivants au type proof_content :

```
...
| Pdef of int * ntype * nterm
| Ptyintro of string * string * string * nterm * nterm
    * proof
| Paxm of string * nterm;;
```

A.1.2 Résultats

Au niveau de la contrainte de temps, l'enregistrement de toute la bibliothèque standard de HOL-Light, soit 1694 théorèmes, prend 3 min, ce qui est plus qu'acceptable ! Nous n'avons pas de valeurs quantitative concernant l'encombrement mémoire, mais il est suffisamment faible pour être transparent au niveau de l'utilisateur.

A.2 Exportation

A.2.1 Principe

Une fois que les théorèmes sont enregistrés avec leurs preuves, il faut les exporter dans des fichiers. Pour son implantation, S. Obua a choisi d'exporter vers un format XML. Cela ne posait pas de problème pour une importation dans Isabelle/HOL, qui autorise les entrées ; mais Coq ne les autorise pas, et passer par le format XML nécessiterait une étape intermédiaire de traduction de ce XML vers du Coq.

C'est pourquoi j'ai trouvé plus simple de modifier la fonction d'exportation de S. Obua pour extraire directement des fichiers Coq (extension .v), ainsi qu'un Makefile pour les compiler.

S. Obua génère exactement un fichier XML par morceau de preuve exporté. Coq ne pourrait pas gérer autant de fichiers, c'est pourquoi j'ai restreint le nombre de fichiers en insérant 10000 preuves par fichier.

Le reste de l'implantation de S. Obua, et en particulier le principe du partage, est conservé. L'enjeu principal de l'exportation est d'obtenir des fichiers source Coq de taille raisonnable et, de manière liée, un temps d'exportation le plus faible possible. Le partage joue là un rôle essentiel. En plus du partage suggéré par l'utilisateur et qui était déjà utilisé lors de l'enregistrement des preuves, on partage n'importe quel morceau de preuve qui apparaît au moins deux fois dans l'ensemble des théorèmes.

En Coq, les arbres de preuve exportés sont des objets de type proof, qui est un type très proche de celui de HOL-Light :

```
Inductive proof : Type :=
| Prefl : term → proof
| Pbeta : idV → type → term → proof
| Pinstt : proof → list (idT * type) → proof
...
| Pconj : proof → proof → proof
| Pconjunct1 : proof → proof
| Pconjunct2 : proof → proof
...
```

Il n'y a que deux différences essentielles :

- il n'y a pas de constructeurs pour les définitions de types et de termes et les axiomes : ceux-ci sont traduits directement vers un **Parameter** Coq, c'est-à-dire un axiome ;
- il y a un nouveau constructeur qui permet le partage :

  ```
  ...
  | Poracle : forall h t, deriv h t → proof.
  ```

  Il permet :
  - d'utiliser des preuves qui ont été faites en Coq sans nécessairement provenir de HOL-Light ;
  - de garder le partage qui a été fait entre les théorèmes provenant de HOL-Light : si THM2 souhaite appeler THM1, il suffit, dans l'arbre de preuve hollight_THM2, d'appeler le constructeur Poracle avec pour argument hollight_THM1_lemma (les arguments h et t de Poracle sont implicites, il suffit donc de fournir une preuve de deriv h t (voir paragraphe 4.1)).

A.2.2 Résultats

L'exportation de l'ensemble de la bibliothèque standard de HOL-Light (1694 théorèmes) produit 191620 "morceaux de preuves", prend 7 min 30 et

génère 921Mo de fichiers. Par rapport aux tests réalisés par S. Obua dans [OS06] :

- les fichiers générés sont quatre fois plus volumineux. Cela s'explique en grande partie par le fait que des fichiers source Coq sont plus verbeux que les fichiers XML générés par le système de S. Obua ;
- l'exportation est quatre fois plus rapide. Les raisons de cette différence de rapidité peuvent être :
  - la vitesse du processeur sur lequel les tests ont été réalisés. S. Obua ne précise pas les caractéristiques de celui qu'il a utilisé, et il est donc difficile de déterminer dans quelle mesure cela influe ;
  - le fait de n'avoir qu'une vingtaine de fichiers contre 200000 fichiers pour S. Obua : il y a moins d'ouvertures et de fermetures de fichiers.

B Compilation

La compilation a été testée sous Linux Debian Sid, avec les logiciels suivants :

- Coq, version 8.2 de développement [2] ;
- Ssreflect, version de développement pour Coq 8.2 [3] (la commande `ssrcoq` doit être dans la variable d'environnement $PATH) ;
- OCaml, version 3.11.1-2 ;
- ocamlgraph, version 1.1-1

L'exportation se déroule en 7 étapes :

1. compiler le modèle Coq :

```
cd model
make
```

2. définir la variable d'environnement $HOLPROOFEXPORTDIR, pour indiquer le répertoire d'exportation des termes de preuve HOL-Light, par exemple :

```
export HOLPROOFEXPORTDIR=/tmp
```

3. compiler un top-level OCaml chargeant les bibliothèques `ocamlgraph`, `my_dep` et `my_str` :

```
cd ../hol_light
ocamlmktop -o mytoplevel -I +ocamlgraph graph.cma
    my_dep.cma my_str.cma
```

4. compiler et lancer le top-level :

```
make
./mytoplevel
```

5. lancer HOL-Light et exporter les preuves :

```
#use "hol.ml";;
export_saved_proofs ();;
```

en choisissant les interprétations des définitions de termes demandées. Sur cette partie du cœur de HOL-Light, il s'agit de `?!` (opérateur d'existence et unicité), que l'on peut par exemple choisir de traduire par :

```
fun P ⇒ ex (unique P)
```

Pour les interprétations des définitions de termes, il faut mettre :
- une ligne vide, qui signifie l'ajout d'une variable représentant cette définition de terme ; ou
- un terme Coq correct et ayant le bon type.

Dans les autres cas, les fichiers Coq créés ne compileront pas correctement ;

6. copier les fichiers Coq compilés dans le répertoire d'exportation, puis s'y rendre :

```
cp ../model/*.vo
    $HOLPROOFEXPORTDIR/hollight/hollight/
cd $HOLPROOFEXPORTDIR/hollight/hollight
```

7. compiler les théorèmes Coq qui ont été générés :

```
make
```

C Implantation du modèle

Dans notre implantation [1], le répertoire `model` comprend les fichiers suivants :

- `list_ext.v`, qui prouve des propriétés de base sur les listes ;
- `axioms.v`, qui ajoute à Coq l'axiome d'extensionnalité ainsi que des axiomes de la logique classique ;
- `hol.v`, qui définit les structures de données de HOL-Light (paragraphe 3.2) ;
- `cast.v`, qui gère l'égalité des types HOL-Light (tel qu'expliqué au paragraphe 3.2.2) ainsi que des termes ;
- `typing.v`, qui définit les jugements de typage (paragraphe 3.2.4) ;
- `translation.v`, qui définit les fonctions de traduction ainsi que leurs propriétés (paragraphe 3.3) ;
- `subst_db.v` et `substitution.v`, qui définissent les substitutions de variables (ces substitutions étant définies exactement comme dans [ACP+08], je n'ai pas détaillé cette partie du modèle) ;
- `ssrfsets.v`, qui implante une bibliothèque d'ensembles finis semblable à `FSets` mais se basant sur un `eqType` ;

– `deriv.v`, qui implante les règles de dérivation et démontre le lemme fondamental (paragraphe 3.4) ;
– `proof.v`, qui implante le type proof (paragraphe 2.3).