Importer les preuves de Logipedia dans Agda

Tristan Delort

▶ To cite this version:

Tristan Delort. Importer les preuves de Logipedia dans Agda. Théorie et langage formel [cs.FL]. 2020. hal-02985530

HAL Id: hal-02985530
https://inria.hal.science/hal-02985530

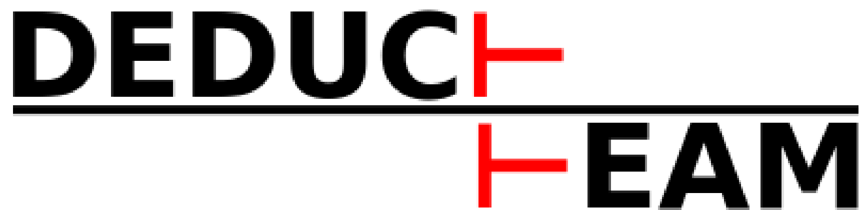
Submitted on 7 Jan 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Rapport de Stage

Importer les preuves de Logipedia dans Agda



Tristan Delort - Étudiant en première année à l'ENSIIE
Tuteurs Deducteam : Frederic Blanqui & Guillaume Genestier
Tuteur ENSIIE : Guillaume Burel

1er Juin 2020 - 31 Juillet 2020

Table des matières

1 Remerciements **3**

2 Introduction **4**

3 Contexte du stage **5**
- 3.1 Organisme d'accueil 5
- 3.2 Environnement de travail 5
- 3.3 Le projet Logipedia 5

4 Présentation du stage **7**
- 4.1 Notions importantes 7
- 4.2 Objectifs du stage 8
- 4.3 Outils mis en oeuvre 8

5 État de l'art **9**
- 5.1 Dedukti - Lambdapi 9
- 5.2 Agda 10

6 dk2agda - le traducteur d'encodages **11**
- 6.1 Introduction 11
- 6.2 Obtention d'une signature 11
- 6.3 Termes 11
- 6.4 Règles 12
- 6.5 Conclusion 14

7 STTfa2Agda - le traducteur de preuves **15**
- 7.1 Introduction 15
- 7.2 STTfa dans Logipedia 15
- 7.3 Traduction naïve 16
- 7.4 (Im)prédicativité 17
- 7.5 Niveaux d'univers 17
- 7.6 Conclusion 19

8 Conclusion **20**

A Développement Durable **21**

B dk2agda : Exemple de Traduction **22**

B.1 Dedukti 22

B.2 Agda 22

C Syntaxe de l'AST STTfa **23**

1 Remerciements

Je tiens tout d'abord à remercier Guillaume BUREL pour avoir transmis ma candidature à l'équipe Deducteam. Ceci m'a permis d'obtenir un stage malgré les conditions exceptionnelles de cette année 2020.

Je souhaite ensuite remercier toute l'équipe Deducteam, dans laquelle j'ai travaillé, pour son accueil et les conseils qu'elle m'a prodigués tout au long de ce stage.

J'adresse bien évidemment mes remerciements à mes tuteurs de stage, Frédéric BLANQUI et Guillaume GENESTIER, pour le temps qu'ils ont consacré au suivi de mon stage et au partage de leur expertise.

2 Introduction

"The realm of formal proofs is today a tower of Babel, just like the realm of theories was, before the design of predicate logic."[1]

Aujourd'hui, il existe un grand nombre d'assistants de preuves basés sur plusieurs logiques, et il est difficile d'utiliser dans un système des théorèmes prouvés dans un autre. Mon stage s'est déroulé dans l'équipe Deducteam qui, pour résoudre ce problème, a développé le framework Dedukti et la plateforme Logipedia.

Initialement, l'objet de mon stage était la création d'un traducteur entre Dedukti et Agda pour que les utilisateurs de Agda aient accès au fichiers présents dans Logipédia. Cependant, pour des raison techniques, j'ai été amené à développer deux traducteurs.

3 Contexte du stage

3.1 Organisme d'accueil

Créée en 2011, Deducteam est une équipe de recherche de l'INRIA sur le thème Preuves et vérification située au LSV (Laboratoire Spécification et Vérification). L'équipe est composée d'une vingtaine de personnes dont 6 membres permanents. Les chercheurs de Deducteam travaillent sur plusieurs sujets mais plus particulièrement :

— La théorie $\lambda\Pi$-calcul modulo réécriture et la déduction modulo,
— Dedukti [1], un framework pour plusieurs systèmes de preuve, basé sur le lambda-pi-modulo calcul,
— L'interopérabilité des systèmes de preuve grâce à Dedukti.

De plus, l'équipe développe une plateforme, Logipedia, permettant l'échange de preuves vers plusieurs systèmes de preuves.

3.2 Environnement de travail

Durant mon stage, j'ai été encadré par Frédéric Blanqui, membre permanent de l'équipe, et Guillaume Genestier, doctorant en troisième année. À cause de la situation exceptionnelle, l'entièreté de mon stage s'est effectuée à distance, à l'exception d'un séminaire rassemblant une grande partie de l'équipe, fin juillet.

Des réunions Skype hebdomadaires avec mes encadrants ont été mises en place, compte tenu de cette organisation du travail, afin de suivre l'avancement de mes travaux.

De plus, une réunion rassemblant toute l'équipe se tenait tous les jeudis via bbcollab.

3.3 Le projet Logipedia

Pour facilité l'interopérabilité entre assistants de preuves, la plateforme Deducteam a développé Logipedia. C'est une encyclopédie de preuves exprimées avec le framework logique Dedukti. Il est possible d'importer les preuves présentes dans Logipedia mais il est aussi possible de les exporter vers d'autres systèmes.

Aujourd'hui, 6 systèmes sont supportés :

— Coq
— Matita
— HOL Light
— Lean
— PVS
— Opentheory

Pour ce qui est de la traduction de systèmes vers Dedukti, un certain nombre de traducteurs ont été réalisés et sont répertoriés dans la Figure 1.

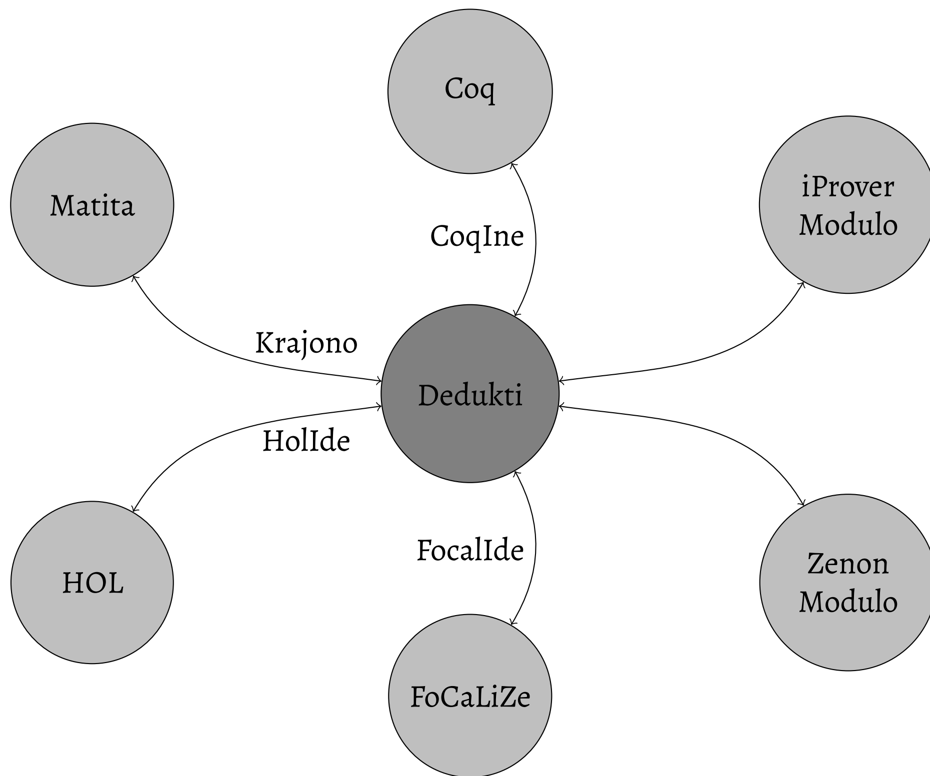


FIGURE 1 – Interactions entre Dedukti et des systèmes de preuve

4 Présentation du stage

L'objectif principal de mon stage est donc la mise en place d'un traducteur permettant aux utilisateurs de Agda d'avoir accès aux fichiers de Logipedia. Pour ce faire, nous avons choisi de développer 2 outils : un traducteur d'encodages de logiques, puis un traducteurs de preuves exprimées dans une de ces logiques.

4.1 Notions importantes

Plusieurs notions, termes, et acronymes sont utilisés dans ce rapport. Voici une liste des plus importants :

STTfa STTfa [3] (ou STTforall) est une extension de la théorie des types simples (STT) avec du polymorphisme prenex. C'est une logique puissante dans laquelle on peut exprimer un très grand nombre de théorèmes. Beaucoup de preuves dans Logipedia sont exprimées dans cette logique.

Réécriture Un système de réécriture est un ensemble de règles $lhs \to rhs$ réécrivant le terme lhs en rhs. Par exemple, la conjonction logique s'écrit de la façon suivante :

$$\bot \wedge _ \to \bot$$
$$\top \wedge x \to x$$

Il est important de noter l'utilisation du joker _ (qui fonctionne comme en OCaml), et d'une variable x.

$\lambda\Pi$-calcul modulo réécriture [1] C'est la théorie sur laquelle est basée Dedukti. C'est une extension du λ-calcul, développé par Alonzo Church en 1930, avec des types dépendants et des règles de réécriture.

Lambdapi C'est le nom de la version 3 de Dedukti qui se veut plus interactif. Une interface Visual Code a d'ailleurs été développée cette année par François Lefoulon, élève de l'ENSIIE en première année.

Univers Un univers est une notion proche d'un ensemble qui peut contenir ce que l'on veut. Il est donc possible d'avoir U_1 et U_2, 2 univers tel que $U_1 \in U_2$. Pour éviter le paradoxe suivant [1], certaines théories des types utilisent même une hiérarchie d'univers infinie.

$$\mathcal{U} = \{x, x \notin x\} \Rightarrow (\mathcal{U} \in \mathcal{U} \iff \mathcal{U} \notin \mathcal{U})$$

1. C'est le paradoxe de Russell. L'existence de l'ensemble de tous les ensembles ne se contenant pas eux même est contradictoire.

Notations Dans Dedukti, TYPE et KIND représentent les types. Dans Agda, pour représenter un type il y a Set, Set_n (avec $n \in \mathbb{N}$). $Setn$, $Set\ n$ et Set_n sont des notations équivalentes.

4.2 Objectifs du stage

Dedukti est un framework logique pour assistants de preuves et un type-checker pour la théorie du $\lambda\Pi$-calcul modulo. Il est donc possible d'y encoder une logique pour ensuite développer des théorèmes dans cette logique.

Le premier objectif de ce stage est donc la réalisation d'un traducteur d'encodages de logiques. Un tel traducteur pourrait donc permettre à n'importe quel utilisateur de Agda d'avoir accès à ces logiques.

Dans Logipedia, les preuves sont écrites dans des logiques. Un grand nombre des preuves sont pour l'instant écrites dans la logique STTfa.

Un deuxième traducteur expérimental a donc été conçu afin de traduire les preuves (et non les encodages) présentes dans Logipédia et exprimées dans la logique STTfa.

4.3 Outils mis en oeuvre

Les parties développement ont été réalisées entièrement sous *Vim* pour les 3 langages utilisés. Il a donc été nécessaire d'installer, sous Linux, un environnement de développement pour *OCaml* grâce aux ajouts automatiques de *opam*, et un environnement pour *Agda* grâce au plugin : agda-vim [2].

Ensuite, Visual Code à servi à la lecture de base de code, pour naviguer facilement entre définitions et déclarations dans le code source de Dedukti (v2.6), Lambdapi, et Logipedia.

Le système de gestion de version Git a été utilisé pour l'ensemble de mes développements. Un repository à part pour le premier traducteur et une Pull Request sur Logipedia pour le deuxième ont été créés.

La programmation des traducteurs ont été principalement élaborés en OCaml. Mais j'ai aussi développé dans les langages suivants :

— *Agda* pour créer des fichiers de test simples et pour modifier les fichiers traduits [2].

— *Script sh* pour manipuler les traducteurs de façon plus pratique.

— *Dedukti/Lambdapi* pour écrire des fichiers de test.

2. Pour résoudre les problèmes rencontrés à la main avant d'implémenter les changements dans les traducteurs.

5 État de l'art

La première semaine de stage a été consacré a de la formation, afin de connaître l'état de l'art. En outre, de la bibliographie me permettant de faire le pont entre la Logique du premier ordre, vu en cours, et des théories plus complexes (par exemple STTfa). Il aussi a été nécessaire de lire beaucoup de code pour comprendre les points de départ et d'arrivée de mes traducteurs.

5.1 Dedukti - Lambdapi

Le point de départ est Dedukti, un framework logique dans lequel beaucoup de logiques peuvent être exprimées. C'est possible notamment avec l'utilisation de règles de réécritures. Parmi les logiques encodées dans Logipedia, il y a :
- la logique intuitionniste propositionnelle,
- la logique du premier ordre,
- la théorie des types simples,
- le calcul des constructions inductives avec des univers,
- et le λ-calcul.

Il est donc possible d'encoder des logiques aussi simples que le calcul de prédicats, mais aussi des logiques très fortes avec des univers ou encore (depuis peu) la théorie des types cubiques.

Ensuite, une fois une logique encodée, il suffit de l'utiliser pour développer des théorèmes. Par exemple, dans Logipedia il y a un encodage de la logique *STTfa* ainsi qu'une preuve du petit théorème de Fermat exprimée dans cette logique.

Termes dans Dedukti Une notion importante dans Dedukti (mais aussi dans beaucoup d'assistants de preuves) est qu'un type peut avoir un type. Par exemple, le type des booléens :

$$\mathbb{B} : \text{TYPE} \quad \top : \mathbb{B} \quad \bot : \mathbb{B}$$

Dans Dedukti, $\top$ a le type $\mathbb{B}$, et $\mathbb{B}$ a le type TYPE. Mais TYPE a aussi un type, KIND. Dedukti a donc la hiérarchie de termes suivante :

3. KIND
2. Termes de type KIND, appelés les *kinds* (TYPE en fait partie mais $\mathbb{B} \to$ TYPE aussi)
1. Termes qui ont pour type un kind, appelés les types (ie $\mathbb{B}$)
0. Termes qui ont pour type un type (ie $\top$ ou $\bot$)

5.2 Agda

Parmi les logiques que l'on peu encoder dans Dedukti, il y a la théorie des types intuitionnistes de Per Martin-Löf sur laquelle Agda est fondé. Agda est un langage de programmation fonctionnelle développé en Haskell à l'université de Chalmers en Suède. C'est un langage fortement typé, qui inclus des types dépendants. C'est pour cela qu'il est souvent utilisé comme assistant de preuve.

Termes dans Agda Contrairement à Dedukti, la hiérarchie de termes dans Agda est infinie. Par exemple,

$$\mathbb{N} : Set \quad zero : \mathbb{N} \quad succ : \mathbb{N} \to \mathbb{N}$$

Ici, $\mathbb{N}$ a le type Set. Mais Set (qui est un raccourci pour Set_0) a le type Set_1. Il y a en fait la hiérarchie d'univers suivante :

- Set (ou Set_0) le type des petits types
- $\forall i \geq 0,\ Set_i : Set_{i+1}$

Et la règle du produit :

$$\frac{\Gamma \vdash A : Set_a, \quad \Gamma, x : A \vdash B : Set_b}{\Gamma \vdash \Pi x : A\ B : Set_{max(a,b)}} \Pi_{agda}$$

Type Dépendant C'est un type dont la définition dépend d'une valeur. Par exemple, le type des listes paramétrées par leur taille :

$$Vector : \mathbb{N} \to \text{Set}$$

Mais les types dépendants sont aussi utilisés pour représenter les quantifications. Par exemple, le produit dépendant $\Pi x : A\ B$ représente $\forall x : A \to B$.

6 dk2agda - le traducteur d'encodages

6.1 Introduction

Cette section explique les étapes de la réalisation d'un traducteur d'encodage de logique, de Dedukti/Lambdapi à Agda.

Soit $|\cdot|$ la traduction de Dedukti/Lambdapi vers Agda et $\langle x \rangle$ le nom de x. Malgré le peu de restrictions pour un nom dans Agda, la fonction $\langle \cdot \rangle$ effectue une modification des noms car dans Agda :

— Un nom contenant des _ est considéré comme opérateur mixfix.

— Un nom sous la forme `[0-9]`+ est considéré comme un entier.

— Certains mots clés sont interdits (`data`, `Set`, `infix`, . . .).

$\langle \cdot \rangle$ remplace donc chaque _ par ^ (sauf pour les jokers) et préfixe les noms rentrant dans les 2 dernières catégories par `dk^`.

6.2 Obtention d'une signature

Pour obtenir toutes les informations sur un fichier, il faut récupérer la signature. Après vérification d'un fichier .dk ou .lp par Lambdapi, un fichier .lpo est généré. C'est une signature qui contient toutes les informations nécessaires à la traduction : les symboles, les types, les règles de réécriture, les dépendances...

L'API de lambdapi fournit la fonction `compile_file` qui permet d'obtenir une signature à partir d'un chemin de fichier. Il faut seulement que le fichier soit un fichier valide (qu'il type-check et qu'il ait l'extension .dk ou .lp). Il est donc impossible de traduire un fichier erroné.

Il faut donc maintenant traduire les symboles un à un. Parmi une dizaine de champs, un symbole contient :

— un nom (= un chaîne de caractères),

— un type (= un terme),

— une définition (optionnelle) (= un terme),

— et une liste (possiblement vide) de règles de réécritures.

Contrairement au type et à la définition, les règles sont un type plus complexe qui sera abordé dans une partie ultérieure.

6.3 Termes

Un terme est un type somme qui peut être :

- une variable, *Vari*,
- un symbole, *Symb*,
- TYPE,
- KIND,
- une variable de pattern, *Patt*,
- un joker, *Wild*,
- un produit, $\Pi x : A\ B$
- une abstraction, $\lambda x : A.\ B$
- une application, $A\ B$
- une définition locale, `let x = y in z`

Il manque trois champs qui sont utilisés en interne (unification, scope checking, meta variables, subject réduction, . . .) et qui sont donc inutiles une fois la compilation finie.

Dans une représentation d'un terme comme un arbre, les éléments de la colonne de gauche sont les feuilles et ceux de la colonne de droite des noeuds.

Traduction Traduction des feuilles :

- $|Symb| = \langle Symb \rangle$
- $|Vari| = \langle Vari \rangle$
- $|Patt| = \langle Patt \rangle$ [3]
- $|\text{TYPE}| =$ `Set0`
- $|\text{KIND}| =$ `Set1`
- $|Wild| =$ `_`

Traduction des noeuds :

- $|\Pi x : A\ B| =$ `(`$\langle x \rangle$ `:` $|A|$`) ->` $|B|$
- $|\lambda x : A.\ B| = \lambda$`(`$\langle x \rangle$ `:` $|A|$`) ->` $|B|$
- $|AB| = |A|\,|B|$

Et pour ce qui est de la substitution `let x = y in z`, `x` est remplacé par `y` dans `z`.

6.4 Règles

La traduction des types et des définitions est faite. Pour les règles cependant, il existe quelques difficultés supplémentaires.

Une version simplifiée d'une règle s'obtient à travers l'API de Lambdapi :

- *nom*, le terme de tête (récupéré depuis *symbole*),
- *lhs*, une liste de termes,
- *rhs*, un terme [4],
- et *vars*, une liste de variables utilisées dans la règle.

Les termes de *lhs* et *rhs* sont traduits un à un.

3. Dans Lambdapi, il y avait une différence entre le nom d'une même variable si elle était utilisée à droite ou à gauche d'une règle de réécriture. Cette différence a donc été corrigée. D'autres détails seront précisés dans la partie suivante sur *Patt*.

4. Il faut utiliser la fonction `term_of_rhs` de l'API de Lambdapi pour avoir un version sous forme de terme du rhs.

Traduction avec pattern matching En utilisant le pattern matching de Agda, une règle s'écrit :

```
nom lhs = rhs
```

Par exemple, la fonction multiplication sur les entiers dans Agda ressemble à :

```
mult : Nat -> Nat -> Nat
mult zero _ = zero
mult (succ n) m = add (mult n m) m
```

Mais le pattern matching effectué par Agda a une limite. Ici Agda sait que `zero` est un constructeur du type Nat :

```
data Nat : Set where
  zero : Nat
  succ : Nat -> Nat
```

Mais dans la version actuelle de Lambdapi, tous les symboles (y compris les types comme `Nat`) sont représentés comme ceci par le traducteur (ici avec la syntaxe Agda) :

```
Nat : Set
zero : Nat
succ : Nat
```

Donc Agda ne sait pas si le symbole sur lequel il fait du matching est `mult` ou `zero` (`zero` pourrait être un opérateur infix par exemple).

Il faudrait donc pouvoir repérer quels ensembles de symbole forment un type a déclarer avec la syntaxe `data`. Ce n'est pas une tâche triviale mais Amélie Ledein, élève à l'ENSIIE en stage de troisième année dans l'équipe, est en train d'ajouter des types inductifs dans Lambdapi ce qui pourrait faciliter ce processus.

Mais il est possible que le pattern matching ait d'autres limites que les règles de réécriture n'ont pas.

Traduction avec réécriture Agda fournit une option et un pragma permettant d'utiliser une égalité comme règle de réécriture. Il est donc possible d'exprimer la fonction `mult` avec un type et un ensemble de règles :

```
{-# OPTIONS --rewriting #-}
open import Agda.Builtin.Equality
open import Agda.Builtin.Equality.Rewrite

mult : Nat -> Nat -> Nat
rule^mult^1 : mult zero _ ≡ zero
{-# REWRITE rule^mult^1 #-}
rule^mult^2 : ∀ n -> ∀ m -> (succ n) m ≡ add (mult n m) m
{-# REWRITE rule^mult^2 #-}
```

Agda peut donc être utilisé comme moteur de réécriture à la condition que le membre gauche type bien.

Mais un problème survient quand même. En effet, dans Lambdapi, les *Wild* utilisés pour laisser Lambdapi inférer leur valeur, sont remplacés par des variables fraîches après compilation. Il faut donc récupérer la valeur qui leur a été donnée avant qu'elle ne soit remplacée.

Pour résoudre ce problème, la méthode actuellement mise en oeuvre est de récupérer le membre de gauche *avant* que les valeurs ne soient effacées par Lambdapi.

Limitations Les définitions plus simples par pattern matching sont à exclure car les types inductifs ne sont toujours pas supportés.

Ensuite, les règles de réécritures sont une fonctionnalité nouvelle de Agda et ne supportent pas toutes les fonctionnalités de Lambdapi (par exemple la présence d'une lambda abstraction dans le membre de gauche).

6.5 Conclusion

Ce traducteur d'encodage supporte aujourd'hui 60% des encodages testés [5]. Un script shell permettant une utilisation plus intuitive du traducteur a aussi été élaboré. Mais les encodages ne sont pas des preuves. Il est par exemple possible de traduire l'encodage de la logique *STTfa* vers Agda mais il n'existe pas d'outils pour traduire les preuves de cette logique vers Agda.

5. Ces fichiers de test proviennent de lambdapi/test_files/. Ils utilisent donc une grande partie des fonctionnalités de Lambdapi/Dedukti. Certaines de ces fonctionnalités étant impossibles à traduire vers Agda.

7 STTfa2Agda - le traducteur de preuves

7.1 Introduction

Cette section explique les étapes de la réalisation d'un traducteur expérimental de preuves, de Dedukti[STTfa] vers Agda.

Soit Dedukti[x] l'encodage de la logique x dans Dedukti, et $|\cdot|$ la traduction de Dedukti[STTfa] vers Agda.

7.2 STTfa dans Logipedia

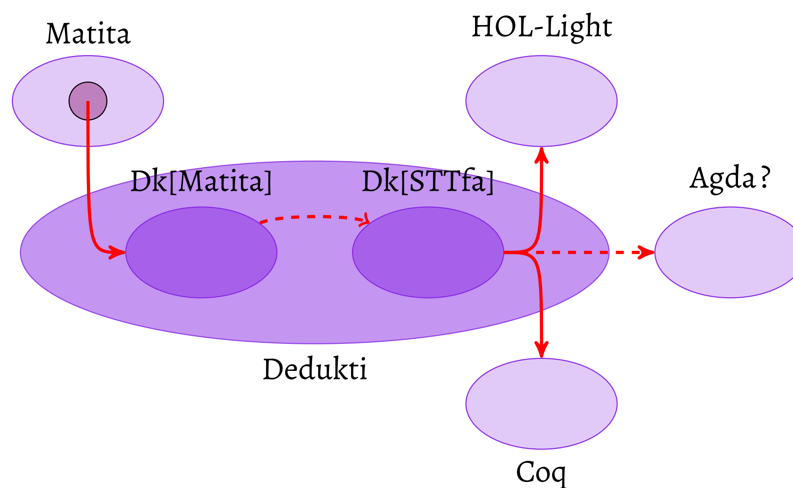


FIGURE 2 – Encodages de Logiques dans Logipedia

Dans Logipedia, les preuves sont exprimées dans plusieurs logiques. La bibliothèque sur laquelle ce traducteur a été testé est une preuve du petit théorème de Fermat (et la bibliothèque d'arithmétique de Matita) exprimé dans Dedukti[STTfa].

La Figure 2 illustre le cheminement de la preuve. Elle a été rédigée dans l'assistant Matita. Un traducteur (Krajono) s'est occupé de la traduction de Matita vers Dedukti[Matita] puis un autre traducteur de Dedukti[Matita] vers Dedukti[STTfa].

Il existe déjà un traducteur de Dedukti[STTfa] vers les 6 systèmes suivants :

— Coq
— Matita
— HOL Light
— Lean
— PVS
— Opentheory

Le passage d'un fichier dans Dedukti[STTfa] vers un AST est donc déjà fait. La première tâche est donc de repérer comment rajouter un système à la fonction d'export de Logipedia. Un guide pour l'ajout d'autres systèmes a été réalisé en parallèle.

7.3 Traduction naïve

Le point d'entrée du traducteur est une fonction qui, à un item, associe une chaîne de caractère. Un item peut être :
— *Parameter* = un nom et un type
— *Axiom* = un nom et un type
— *Definition* = un nom, un type, et un terme
— *Theorem* = un nom, un term, et une preuve
— *TypeDecl* = un nom et une arité
— *TypeDef* = un nom et des variables de type
L'annexe C fournit une version complète de l'AST écrit en EBNF qui a été rédigé lors de ce stage afin d'aider aux traductions.

En se basant sur le traducteur de Coq, il est possible d'obtenir la traduction suivante :
— $|Parameter|$ = `postulate nom : ` $|type|$
— $|Axiom|$ = `postulate nom : ` $|type|$
— $|Definition|$ = `nom : ` $|type|$ `\n nom = ` $|terme|$
— $|Theorem|$ = `nom : ` $|terme|$ `\n nom = ` $|preuve|$
— $|TypeDecl|$ = `nom : ` $|arité|$
— $|TypeDef|$ = non supporté [6]

Parameter et *Axiom* utilise la notation `postulate` car ils n'ont pas de définition. Et ensuite, pour traduire les termes, les types, et les quantifications c'est très similaire à dk2agda. La notion de Prop, est traduite par `Prop` de Agda [7].

La notion intéressante est la notion de preuves. Elle est constituée de 9 champs :

— Assume
— Lemma
— Conv
— ImplE
— ImplI
— ForallE
— ForallI
— ForallPE
— ForallPI

Pour les règles d'introduction j'ai utilisé une abstraction et pour l'élimination, une application. Pour ce qui est de *Assume*, *Lemma*, et *Conv*, j'ai utilisé respectivement, la variable, le nom, ou la preuve, qu'ils contiennent.

6. Cette option n'est supportée dans aucun des traducteurs de Dedukti[STTfa]. Elle n'a donc pas été prise en compte dans un premier temps (la résolution des problèmes dans les prochaines sections étant prioritaire).
7. Ici ce Prop n'est pas imprédicatif comme dans Coq. Pour les versions plus expérimentales du traducteur, Prop a été remplacé par Set.

Cependant cette traduction, bien que lisible, n'est pas utile car elle ne type check pas [8]. En effet Agda est un assistant de preuve **Prédicatif**. De plus, dans STTfa les niveaux d'univers sont implicites (non pris en compte dans cette traduction).

7.4 (Im)prédicativité

Definition 1. Prédicativité. Un objet ne peut pas "parler de lui même".

C'est une définition très vague car la prédicativité est une notion qui a été définie plusieurs fois au cours de l'histoire, et de manières différentes. Mais le commun de toutes ces définitions est l'idée d'introspection ou de définition de soi. Concrètement, une définition ne peut pas se faire référence.

Un exemple simple dans STTfa serait :

$$T : \forall p, p \Rightarrow p$$

Ici, p quantifie sur toutes les propositions. T est aussi une proposition. Donc cette définition de T est imprédicative et il est donc impossible d'exprimer T dans Agda.

La hiérarchie d'univers de Agda est d'ailleurs pensée pour éviter l'imprédicativité. Si $p : Set_i$ alors $\forall p$ sera de type Set_{i+1} et donc T de type Set_{i+1}. p ne quantifie donc plus sur un Univers où T est présent.

Il existe des preuves utilisant l'imprédicativité, sans pour autant en avoir besoin, car cela rend la preuve plus courte. Il est donc possible de traduire ces preuves. Pour ce qui est des autres preuves qui utilisent forcement l'imprédicativité, il est impossible de les traduire vers Agda.

7.5 Niveaux d'univers

Le second problème est (théoriquement) possible à résoudre. C'est le problème des niveaux d'univers qui sont implicites dans STTfa mais explicites dans Agda. Par exemple, dans Coq, les univers sont dit "flottants" et leurs niveaux sont implicites. C'est lors du type checking que les contraintes entre les niveaux sont évaluées et des niveaux sont inférés.

Prenons un exemple simple : Les 2 expressions suivantes sont valides dans STTfa :

$$eq : \forall A : type \rightarrow A \rightarrow A \rightarrow bool$$

$$refl : \forall A : type \rightarrow x : A \rightarrow eq\ A\ x\ x$$

Mais pour les traduire vers Agda, il faut savoir ce que $type$ représente.

Est-ce `Set` ? `Set1` ?

8. Sans utiliser `--type-in-type` qui rend Agda incohérent.

eq ou $refl$ seront-ils utilisés sur plusieurs niveaux dans les preuves [9] ?

L'utilisation de l'export vers Coq a permis de répondre à ces questions. Toutes les occurrences de Prop ont été remplacés par Type, Puis les niveaux d'univers et leurs contraintes ont été extraits avec *coqtop*. On peut remarquer que :

— Certaines expressions sont utilisées sur plusieurs niveaux.
— La bibliothèque pour le petit théorème de Fermat n'utilise pas l'imprédicativité [10].
— Cette bibliothèque utilise 3 niveaux d'univers.

Pour résoudre le problème des niveaux d'univers, j'ai exploré 3 méthodes :

1. Laisser Agda inférer les niveaux avec "_".
2. Utiliser le polymorphisme d'univers.
3. Utiliser la cumulativitée ou des lifts.

La solution 1 ne peut pas fonctionner seule car certaines expressions sont utilisées sur plusieurs niveaux.

Polymorphisme d'univers C'est un mécanisme qui permet à une expression d'être valide sur tout niveau. Mais si dans une expression, un niveau a_1 et a_2 sont utilisés et qu'une contrainte stipule que $a_1 > a_2$, il faut déduire lors de la traduction que cette contrainte existe pour pouvoir écrire l'expression.

Aucune traduction utilisant ce principe ne type check aujourd'hui mais c'est la méthode la plus prometteuse pour un traducteur 100% automatique. La version actuelle de la Pull Request sur Logipedia utilise ce principe.

Cumulativité et lift Un élément de type Lift peut être "lifté" vers des niveaux supérieurs. Ainsi, si il est valide sur Set_i, il pourra être "lifté" vers n'importe quel Set_j avec $j \geq i$.

La cumulativité peut être rajoutée dans Agda avec `--cumulativity` et elle permet de rendre les Lift implicites. Avec la cumulativité, non seulement $Set_i \in Set_{i+1}$ mais $Set_i \subset Set_{i+1}$. Donc, si une expression est de type Set_i, elle sera valide dans tout Set_j avec $j \geq i$.

Il est possible de faire type-checker les fichiers de façon semi-automatique, avec le traducteur, en ajoutant les niveaux de façon manuelle (en les prenant par exemple de *coqtop*) et en utilisant `--cumulativity`. La limite est qu'il faut "trouver" les niveaux d'univers et que `--cumulativity` est une option récente de Agda. Par exemple, il est impossible d'utiliser le polymorphisme d'univers et la cumulativité dans une même expression.

9. Les univers dans Matita sont cumulatifs.
10. Prop a été remplacé par Type car Type est prédicatif. Donc si la bibliothèque type check sans Prop, c'est qu'elle n'utilise pas l'imprédicativité.

7.6 Conclusion

Un traducteur expérimental qui fournit des preuves lisibles est maintenant disponible. En parallèle de ce traducteur, un guide pour implémenter d'autres traducteurs a été développé. Et une note répertoriant les tentatives de traductions, avec les erreurs qui en découlent, a été rédigée.

De plus, une version expérimentale utilisant le polymorphisme d'univers est accessible sur la Pull Request sur Logipedia.

Ce traducteur ouvre donc vers un projet : La résolution du problème de niveaux d'univers dans le traducteur STTfa2Agda.

8 Conclusion

Tout au long de ce stage ont été créés :

- Dans dk2agda :
  - un traducteur d'encodages,
  - et un utilitaire shell pour utiliser le traducteur.
- Dans STTfa2Agda :
  - un traducteur de preuves "lisibles",
  - une version expérimentale demandant la résolution du problème de niveaux d'univers,
  - une note décrivant les problèmes rencontrés,
  - et un guide pour implémenter un nouveau traducteur.

Aucun de ces traducteurs ne sont des fonctions totales. Une amélioration serait d'augmenter le pourcentage d'encodages que dk2agda peut traduire et d'ajouter les type inductifs quand ils seront disponibles.

Puis un nouveau projet serait de résoudre le problème du deuxième traducteur. En rajoutant par exemple un script extérieur afin d'obtenir les niveaux d'univers, ou alors en utilisant les fonctionnalités d'Agda.

A Développement Durable

Les démarches écologique de l'équipe s'inscrivent dans les directives de l'Inria. Depuis 2020, l'Inria présente annuellement lors de son conseil d'administration les actions mises en oeuvre pour réduire son impact carbone. Pour ce qui est des nouveaux locaux de l'ENS Paris-Saclay, le tri sélectif y est pratiqué et le caractère récent du bâtiment offre une meilleur isolation thermique.

Quant a l'égalité Homme-Femme, même si le femmes sont sous-représentées au sein de l'équipe (4 femmes pour une équipe de 30, dont 2 stagiaires et une membre associée), je n'ai été témoins d'aucune différence de traitement entre les sexes au cours de mon stage.

B dk2agda : Exemple de Traduction

B.1 Dedukti

```
Nat : Type.

0 : Nat.
S : Nat -> Nat.

def half : Nat -> Nat.

[]  half 0          --> 0.
[]  half (S 0)      --> 0.
[n] half (S (S n)) --> half n.
```

B.2 Agda

```
{-# OPTIONS -W noMissingDefinitions --rewriting #-}
module half where
open import Agda.Builtin.Equality
open import Agda.Builtin.Equality.Rewrite

Nat : Set

dk^0 : Nat
S : (Nat -> Nat)

half : (Nat -> Nat)

rule^half^0 : half dk^0 ≡ (dk^0)
{-# REWRITE rule^half^0 #-}
rule^half^1 : half (S (dk^0)) ≡ (dk^0)
{-# REWRITE rule^half^1 #-}
rule^half^2 : ∀ v0^n -> half (S ((S (v0^n)))) ≡ ((half (v0^n)))
{-# REWRITE rule^half^2 #-}
```

C Syntaxe de l'AST STTfa

```
<string> ::= [a-zA-Z0-9_-]+
   <int> ::= [0-9]+

 <ty_var> ::= <string>
 <te_var> ::= <string>
<hyp_var> ::= <string>
   <name> ::= <string> <string>
    <cst> ::= <name>

#Type
<arrow> ::= <_ty> <_ty>
<ty_op> ::= <name> <_ty>*

     <_ty> ::= <ty_var> | <arrow> | <ty_op> | <prop>
<forall_k> ::= <ty_var> <ty>
      <ty> ::= <forall_k> | <_ty>

#Term
   <abs> ::= <te_var> <_ty> <_te>
   <app> ::= <_te> <_te>
<forall> ::= <te_var> <_ty> <_te>
  <impl> ::= <_te> <_te>
<abs_ty> ::= <ty_var> <_te>
   <kst> ::= <cst> <_ty>*

     <_te> ::= <te_var> | <abs> | <app> | <forall>
             | <impl> | <abs_ty> | <kst>
<forall_p> ::= <ty_var> <te>
      <te> ::= <forall_p> | <_te>

#ctx
  <ty_ctx> ::= <ty_var>*
  <te_ctx> ::= (<te_var> <_ty>)*
<judgment> ::= <ty_ctx> <te_ctx> <hyp> <te>
```

```
#Proof
   <assume> ::= <judgment> <hyp_var>
    <lemma> ::= <name> <judgment>
     <conv> ::= <judgment> <proof> <trace>
   <impl_e> ::= <judgment> <proof> <proof>
   <impl_i> ::= <judgment> <proof> <hyp_var>
 <forall_e> ::= <judgment> <proof> <_te>
 <forall_i> ::= <judgment> <proof> <te_var>
<forall_pe> ::= <judgment> <proof> <_ty>
<forall_pi> ::= <judgment> <proof> <ty_var>

<proof> ::= <assume> | <lemma> | <conv> | <impl_e> | <impl_i>
          | <forall_e> | <forall_i> | <forall_pe> | <forall_pi>

#items
<arity> ::= <int>

 <parameter> ::= <name> <ty>
<definition> ::= <name> <ty> <te>
     <axiom> ::= <name> <te>
   <theorem> ::= <name> <te> <proof>
  <typeDecl> ::= <name> <arity>
   <typeDef> ::= <name> <ty_var>* <_ty>

<item> ::= <parameter> | <definition> | <axiom>
         | <theorem> | <type_decl> | <type_def>
```

Références

[1] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Expressing theories in the λΠ-calculus modulo theory and in the Dedukti system. In *22nd International Conference on Types for Proofs and Programs, TYPES 2016*, Novi SAd, Serbia, May 2016.

[2] Derek Elkins. agda-vim, 2020.

[3] François Thiré. Sharing a library between proof assistants : Reaching out to the HOL family. In *LFMTP@FSCD*, volume 274 of *EPTCS*, pages 57–71, 2018.