

Tactiques de preuve dans Dedukti

Antoine Defourné

▶ **To cite this version:**

Antoine Defourné. Tactiques de preuve dans Dedukti. Logique en informatique [cs.LO]. 2017. hal-01661872

HAL Id: hal-01661872
https://inria.hal.science/hal-01661872

Submitted on 4 Jan 2018

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.







Grenoble INP – ENSIMAG
École Nationale Supérieure d'Informatique et de Mathématiques Appliquées

Rapport de projet de fin d'études

Effectué chez INRIA Paris-Saclay

Tactiques de preuve dans Dedukti

Antoine Defourné
3e année – Option MMIS

20 février 2017 – 04 août 2017

INRIA Paris-Saclay
1 Rue Honoré d'Estienne d'Orves
91 120 Palaiseau

Responsable de stage
Frédéric Blanqui
Tuteur de l'école
Thierry Boy-de-la-Tour

Note : Le théorème 2, page 26, n'est vrai que sous certaines conditions. Voir la note en fin de document pour plus de détails.

Table des matières

1 Introduction . . . 4
2 Présentation du stage . . . 4
2.1 Contexte . . . 4
2.2 Objectifs et Planning . . . 5
3 Dossier : Théories des types et Assistants de preuve . . . 6
3.1 Notions de base . . . 6
3.1.1 Théories des types . . . 6
3.1.2 Un exemple : le λ-calcul simplement typé . . . 6
3.1.3 La Correspondance de Curry-Howard . . . 7
3.2 Présentation de Dedukti . . . 8
3.2.1 Concepts . . . 9
3.2.2 Preuve en Dedukti . . . 9
4 Appel à prouveurs externes avec Why3 . . . 10
4.1 Présentation de Why3 . . . 11
4.2 Stratégie . . . 11
4.3 Implémentation . . . 12
4.3.1 Traduction minimale . . . 13
4.3.2 Traduction de formules du premier ordre . . . 15
4.3.3 Traduction de formules de l'arithmétique . . . 19
4.4 Commentaire et Perspectives . . . 21
5 Réécriture de buts . . . 22
5.1 Énoncé du problème . . . 22
5.1.1 Exemple introductif . . . 22
5.1.2 Notion d'égalité . . . 23
5.2 Terme de preuve de réécriture . . . 24
5.3 Filtrage . . . 27
6 Conclusion . . . 30
6.1 Impact sociétal et environnemental . . . 30
6.2 Bilan technique . . . 30
6.3 Bilan personnel . . . 31
A L'Égalité dans Dedukti . . . 32

Introduction

Les premiers travaux en preuve formelle remontent au début du XX^{e} siècle, quand il a été question de réviser les bases sur lesquelles étaient fondées les mathématiques. Si le programme de Hilbert, qui voulait prouver la cohérence des mathématiques, s'est soldé par un échec, ces développements ont plus tard pu trouver une résonance dans le domaine de l'informatique. Aujourd'hui, les preuves informatiques sont une donnée précieuse quand il s'agit de garantir qu'un véhicule automatique, appareil médical, ou logiciel de contrôle d'une centrale nucléaire, ne peut pas faire d'erreur.

Ce stage de recherche s'inscrit dans la thématique de la logique mathématique et de la théorie de la preuve. Il s'agit d'étendre un assistant de preuve basé sur le système Dedukti, développé par l'équipe INRIA Deducteam, avec de nouvelles tactiques de preuve.

On commence par une présentation du contexte et des objectifs du stage. La seconde partie de ce rapport est une synthèse des notions de théorie de la preuve qui sont nécessaires à la compréhension de la suite. Les deux dernières sections sont dédiées aux tactiques de preuve auxquelles on s'est intéressé : la première est une tactique d'appel à un prouveur externe, et la seconde une tactique de réécriture.

Présentation du stage

Contexte

Ce stage s'est déroulé au sein de Deducteam, équipe de l'INRIA Paris-Saclay actuellement basée sur le site de l'ENS de Cachan au *Laboratoire Spécification et Vérification* (LSV). Deducteam fait partie du thème *Programmes, Vérification et Preuves* de l'INRIA, ses activités portent sur les systèmes de preuve en logique, et surtout le développement de l'outil **Dedukti**.

Dedukti est un *type-checker*, un logiciel de vérification de types sur les termes d'un langage spécifique. Dedukti joue un rôle important dans le développement de **preuves formelles** comme celles produites par les assistants de preuve.

Un **assistant de preuve** est un logiciel permettant d'exprimer des théorèmes et des preuves de ces théorèmes dans un certain système logique. L'utilisateur est invité à rédiger des définitions et des formules dans le langage du logiciel (qui correspond assez au langage mathématique en général), et peut également rédiger des *preuves* de ces formules. Cette étape prend souvent la forme d'une interaction entre l'utilisateur et le système, au cours de laquelle les règles que choisit le premier sont vérifiées formellement par le second. Des exemples d'assistants à la preuve sont Coq, Isabelle, ou PVS.

Dedukti n'est cependant pas un assistant de preuve, bien qu'il soit possible d'exprimer des formules et des preuves en Dedukti. Ce logiciel a principalement été conçu pour *vérifier* les preuves importées d'autres systèmes. Il existe quelques systèmes déjà capables de produire une sortie pour Dedukti : Zenon Modulo, iProverModulo. Pour d'autres, des

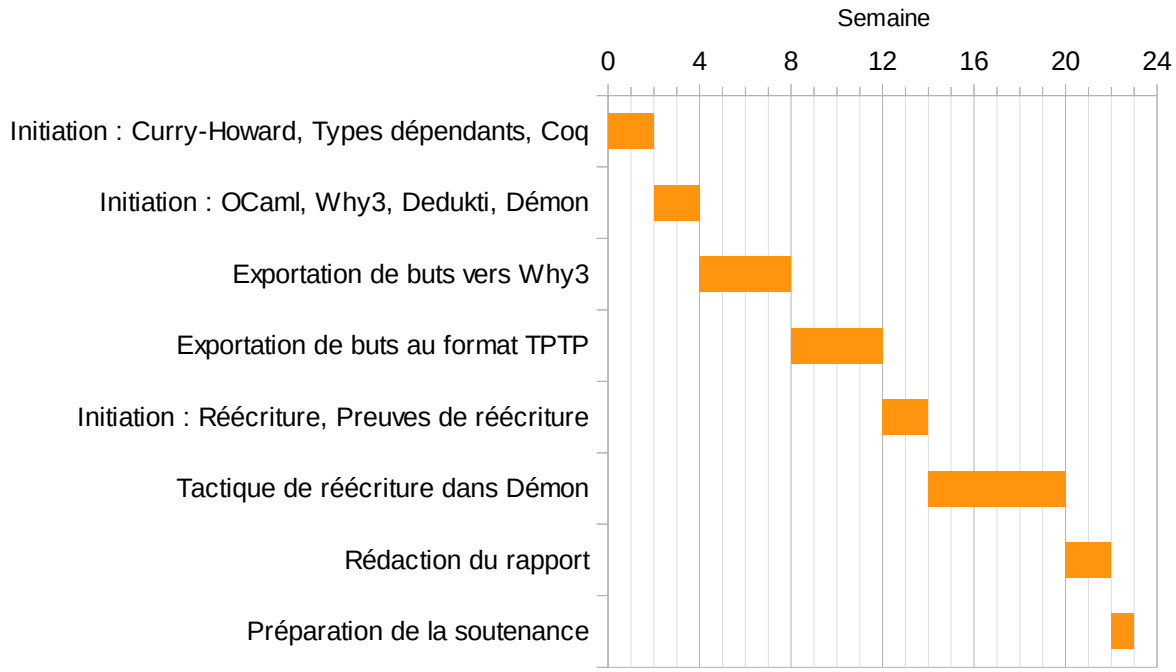


FIGURE 1 – Planning prévisionnel

outils externes ont été développés pour faire une traduction vers Dedukti : CoqInE traduit des preuves de Coq, Krajono traduit des preuves de Matita...

Objectifs et Planning

Ce stage de recherche ne porte pas directement sur Dedukti, mais sur l'outil Demon, un assistant de preuve basé sur le même langage que Dedukti. En tant qu'assistant de preuve, Demon propose plusieurs commandes de haut niveau, appelées **tactiques de preuve**, que l'utilisateur peut utiliser pour tenter de prouver les résultats qu'il énonce. Ces tactiques sont généralement conçues pour ressembler aux étapes d'un raisonnement mathématique, comme "prouver par récurrence sur n" ou "prouver par l'absurde".

Le développement de Demon n'a commencé que très récemment et le logiciel est à l'état de prototype. Il ne comporte que quelques tactiques de preuve basiques. L'objectif initial du stage était de développer deux nouvelles tactiques de preuve pour Demon. La première est une tactique d'*appel à un prouveur externe*, et la seconde une tactique de *réécriture de termes*.

Le diagramme 1 est le planning qui a été établi au début du stage. On peut le découper en quatre étapes :

1. (4 semaines) Initiation au domaine de recherche ;
2. (8 semaines) Travail sur la première tactique : appel à des prouveurs externes ;
3. (8 semaines) Travail sur la seconde tactique : réécriture de termes ;

4. (3 semaines) Préparation du rapport et de la soutenance.

Dossier : Théories des types et Assistants de preuve

On commence par une courte présentation de la partie de la théorie de la preuve qui nous intéresse. Cette présentation aboutit sur une introduction à la *Correspondance de Curry-Howard*, qui lie la notion de *preuve mathématique* à celle de *programme informatique* et joue un rôle central dans certains prouveurs.

On se focalise ensuite sur Dedukti, qui intègre toutes les notions précédentes. L'intérêt est de montrer comment il est possible de faire de la démonstration dans un outil tel que Dedukti.

Notions de base

Théories des types

Les *théories des types* sont des cadres dans lesquels les mots d'un certain langage, qu'on appelle *termes* en général, sont *typés*. En informatique, la notion de type est présente dans la plupart des langages ; en mathématique, on s'en sert parfois de façon implicite. Par exemple, la proposition $n \in \mathbb{N}$ pourrait en quelque sorte se comprendre comme "n a le type des entiers naturels".

Dans une théorie des types, il est possible de définir des types, comme celui des entiers naturels par exemple, qu'on pourrait appeler *Nat*. La déclaration de n se noterait $n : Nat$. Le plus souvent, le typage d'un terme est dépendant du *contexte*. Par exemple, un terme de la forme $m + n$ est typé *Nat* dans un contexte où m et n sont aussi typés *Nat*, ce qui s'écrit $m : Nat, n : Nat \vdash m + n : Nat$.

Définition 1. *On appelle* jugement *toute proposition de la forme $t : A$, où t représente un terme et A représente un type.*

On appelle contexte *toute suite de jugements (a priori ordonnée). On appelle* séquent *toute proposition de la forme $\Gamma \vdash t : A$, où Γ est un contexte.*

On a laissé libre pour le moment les définitions des langages dans lesquels écrire ces termes et ces types, mais la section suivante fournit un cas d'étude simple, qui se trouve être à la base du formalisme de Dedukti.

Un exemple : le λ-calcul simplement typé

À l'origine, le λ-calcul est un modèle de calcul théorique inventé par Alonzo Church dans les années 1930. C'est un langage minimaliste dont tous les termes représentent des fonctions. On peut combiner ces fonctions pour créer d'autres fonctions plus complexes en respectant la syntaxe du langage, qui ne comporte que trois règles.

$$\frac{x : A \in \Gamma}{\Gamma \vdash_{\mathsf{s}} x : A}\ \mathsf{Var} \qquad \frac{\Gamma \vdash_{\mathsf{s}} u : A \to B \qquad \Gamma \vdash_{\mathsf{s}} v : A}{\Gamma \vdash_{\mathsf{s}} u\ v : B}\ \mathsf{App}$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{s}} t : B}{\Gamma \vdash_{\mathsf{s}} \lambda x^A.\, t : A \to B}\ \mathsf{Abs}$$

TABLE 1 – Règles de typage du λ-calcul simplement typé

Le λ-calcul connaît de très nombreuses extensions, et c'est la plus simple d'entre toutes qui va nous intéresser ici : il s'agit du *λ-calcul simplement typé*. L'idée principale est d'affecter un type aux fonctions. Ainsi, étant donné deux types A et B, le type des fonctions prenant un objet de type A en argument et retournant un objet de type B s'écrit simplement $A \to B$.

Les règles de typage des fonctions sont également au nombre de trois, elles figurent sur la table 1. La notation $\frac{p_1 \ \cdots \ p_n}{c}$ signifie que la proposition c (la conclusion) est dérivable sous les hypothèses $p_1, \ldots, p_n$ (les prémisses). On définit la *relation de typage* comme la plus petite relation vérifiant ces règles de typage.

Ces trois règles peuvent se comprendre de la façon suivante :

Var x est un terme de type A si le jugement $x : A$ figure dans le contexte courant ;

App Si u est un terme de type $A \to B$ et v un terme de type A, alors l'application $u\ v$ est un terme de type B ;

Abs Si t est un terme de type B et que t dépend de la *variable* x de type A, alors l'asbtraction $\lambda x^A.\, t$ est un terme de type $A \to B$.

À titre d'exemple, la fonction identité peut s'écrire $\lambda x^A.\, x$. Son type est $A \to A$. Plus précisément, on écrit $\vdash_{\mathsf{s}} \lambda x^A.\, x : A \to A$ pour signifier que ce terme est doté de ce type en dehors de tout contexte. C'est de plus un séquent *dérivable*, car on peut le construire en appliquant deux règles :

$$\dfrac{\dfrac{}{x : A \vdash_{\mathsf{s}} x : A}\ \mathsf{Var}}{\vdash_{\mathsf{s}} \lambda x^A.\, x : A \to A}\ \mathsf{Abs}$$

La Correspondance de Curry-Howard

La *Correspondance de Curry-Howard* (qu'on raccourcira en "Correspondance CH") est une série de résultats montrant les liens entre différents systèmes logiques et modèles de calcul. On l'énonce souvent de manière générale : *les propositions sont des types, et leurs preuves sont les termes.*

Illustrons avec le λ-calcul simplement typé qu'on vient de décrire. On considère la *logique propositionnelle minimale* ; il s'agit du système logique dont le langage est construit sur les variables propositionnelles A, B,... et un seul opérateur, l'implication $\Rightarrow$. Les règles de déduction de ce système sont au nombre de trois :

$$\frac{A \in \Gamma}{\Gamma \vdash_{\mathcal{P}} A} \qquad \frac{\Gamma \vdash_{\mathcal{P}} A \qquad \Gamma \vdash_{\mathcal{P}} A \Rightarrow B}{\Gamma \vdash_{\mathcal{P}} B} \qquad \frac{\Gamma, A \vdash_{\mathcal{P}} B}{\Gamma \vdash_{\mathcal{P}} A \Rightarrow B}$$

où le contexte Γ ne représente qu'un ensemble d'hypothèses, c'est-à-dire de formules.

On peut d'ores et déjà remarquer les similitudes entre le système formé par ces trois règles et le modèle présenté sur la table 1 : il "suffit" d'échanger les flèches $\to$ pour des implications $\Rightarrow$, et d'ignorer les λ-termes. Il s'avère que ces deux systèmes sont en correspondance au sens de CH. C'est-à-dire qu'on a le résultat suivant :

Théorème 1. *Il existe une bijection ϕ des types du λ-calcul simplement typé vers les formules de la logique propositionnelle telle que :*

1. *Pour tout type T, s'il existe un terme t tel que $\vdash_{\mathsf{s}} t : T$ alors $\vdash_{\mathcal{P}} \phi(T)$;*
2. *Pour toute proposition P, si $\vdash_{\mathcal{P}} P$ alors il existe un terme t tel que $\vdash_{\mathsf{s}} t : \phi^{-1}(P)$.*

La signification de ce théorème est que tout terme t de type T est en quelque sorte une *preuve* de la formule $\phi(T)$ correspondant à ce type. Inversement, si P est une formule qu'on peut prouver en logique propositionnelle, alors son type correspondant $\phi^{-1}(P)$ doit être habité, c'est-à-dire qu'il doit exister un terme de ce type.

Ce résultat a des implications importantes, puisqu'il montre qu'on peut *représenter* des preuves de formules logiques par des fonctions qui calculent. Le lien est si fort qu'on peut maintenant confondre type et formule, fonction et preuve. On peut ainsi vérifier que la fonction identité $\lambda x^A.\, x$ *est* une preuve de l'implication $A \Rightarrow A$. De même, une preuve de la formule $A \Rightarrow B \Rightarrow A$[1] existe : il s'agit de la fonction $\lambda x^A.\, \lambda y^B.\, x$, qui retourne le premier de ses deux arguments.

La présentation originale de ce cas de correspondance peut être retrouvé dans [4]. Il existe bien d'autres interprétations pour des logiques et des modèles de calcul plus expressifs. Pour la suite, il est seulement important de comprendre que prouver une formule peut se ramener à considérer le type qui lui correspond, puis construire une fonction de ce type ; c'est comme cela que fonctionnent certains assistants de preuve, comme Coq.

Présentation de Dedukti

Dedukti est un vérificateur de types basé sur une extension du λ-calcul simplement typé : le $\lambda\Pi$-*calcul modulo théorie*. On se propose maintenant de présenter ce formalisme dans les grandes lignes, pour ensuite montrer comment Dedukti peut être utilisé pour représenter des preuves.

Pour une présentation plus complète de Dedukti, voir [1].

1. La convention veut que $A \Rightarrow B \Rightarrow C$ équivaille à $A \Rightarrow (B \Rightarrow C)$. De même pour la flèche $\to$.

Concepts

Le $\lambda\Pi$-calcul modulo théorie intègre deux concepts supplémentaires : les types dépendants, et la réécriture. Combinés, ces deux ajouts rendent très facile l'encodage de logiques plus expressives dans Dedukti.

L'idée des *types dépendants* est de généraliser le type $A \to B$ par le produit dépendant $\prod x^A.\, B$, dans lequel la variable x est susceptible d'apparaître libre dans B. Dans le cas où B ne dépend pas de x, on pourra utiliser la notation avec la flèche $\to$.

La *réécriture* dans Dedukti consiste à fournir au système des règles de réécriture sur les termes ; les termes sont ensuite équivalents modulo ces règles de réécriture. Par exemple, une manière courante de définir l'addition sur les entiers naturels consiste à déclarer les deux règles $0 + n \longrightarrow_{\mathcal{R}} n$ et $S(m) + n \longrightarrow_{\mathcal{R}} S(m + n)$, où $S(n)$ est le successeur de n.

Le code ci-dessous est un exemple de fichier Dedukti.

```
1  #NAME example.
2
3  Nat : Type.
4  0 : Nat.
5  S : Nat → Nat.
6
7  def plus : Nat → Nat → Nat.
8  [n] plus 0 n ⟶ℛ n.
9  [m,n] plus (S m) n ⟶ℛ S (plus m n).
10
11 A : Type.
12 Vec : Nat → Type.
13 Nil : Vec 0.
14 Cons : n:Nat → A → Vec n → Vec (S n).
```

On définit le type des entiers naturels *Nat*, puis les deux constructeurs 0 et S, respectivement le premier entier et la fonction successeur. On déclare ensuite la fonction plus, mais on la définit par des règles de réécriture.

Les dernières lignes illustrent un cas de type dépendant : pour tout n de type *Nat*, Vec n est *le type des listes de taille* n. Les deux constructeurs Nil et Cons servent à construire de telles listes ; si a, b et c sont des objets de type A, alors la liste $[a; b; c]$ est représentée par le terme Cons 2 a (Cons 1 b (Cons 0 c Nil)), et son type est Vec 3.

Preuve en Dedukti

Au sens de la Correspondance CH, les types de Dedukti peuvent être considérés comme des propositions, et leurs fonctions comme des preuves de ces propositions. La flèche $\to$ correspond toujours à l'implication $\Rightarrow$, mais on a ajouté au langage le produit dépendant $\prod$, qui correspond à la quantification universelle $\forall$. Il est montré dans [1] comment faire correspondre une logique du premier ordre intuitioniste minimale avec le $\lambda\Pi$-calcul.

Mais il est possible d'aller plus loin avec la réécriture, grâce à laquelle on peut définir des *encodages* de logiques plus expressives. Le code ci-dessous est un cas standard d'encodage d'un nouveau symbole, en l'occurrence l'opérateur de négation ¬.

```
1  #NAME logic.
2
3  Prop : Type.
4  def ε : Prop → Type.
5
6  neg : Prop → Prop.
7  [P] ε (neg P) ⟶ℛ Q:Prop → ε P → ε Q.
8
9  def neg_neg_intro : P:Prop → ε P → ε (neg (neg P)) :=
10         P:Prop => h1:ε P =>
11                 Q:Prop => h2:ε (neg P) => h2 Q h1.
```

Dans un premier temps, on déclare un nouveau type *Prop*. Les habitants de ce type seront les nouvelles propositions. Ce ne sont plus des propositions au sens de CH, puisque ce ne sont plus des types, il faudrait donc plutôt parler de *codes* de propositions.

Pour représenter des preuves de ces propositions, on doit les traduire en types, d'où l'opérateur ε. Cet opérateur affecte un type unique à chaque code de proposition, si bien qu'une preuve de la proposition encodée par P sera en fait un terme de type $\varepsilon(P)$. Il faut toutefois définir correctement ε, ce qui est fait habituellement par des règles de réécriture.

Par exemple, une définition possible de la négation est $\neg P := \forall Q, P \Rightarrow Q$. Le type $\varepsilon(\neg P)$ devrait donc être défini comme $\prod Q^{Prop}.\, \varepsilon(P) \to \varepsilon(Q)$, d'où la règle de réécriture qu'on a choisie.

La dernière instruction est la déclaration et la preuve d'un théorème, qui correspond à la proposition $\forall P, P \Rightarrow \neg\neg P$. Comme on peut le voir, la preuve de ce résultat est la fonction $\lambda P^{Prop}.\, \lambda {h_1}^{\varepsilon(P)}.\, \lambda Q^{Prop}.\, \lambda {h_2}^{\varepsilon(\neg P)}.\, h_2\ Q\ h_1$. Elle a été entrée à la main, ce qui paraît déjà fastidieux, et qui pourrait s'avérer pratiquement infaisable pour des théorèmes plus complexes ; d'où la nécessité d'assistants comme Demon qui implémentent des tactiques pour faciliter la construction des preuves.

Appel à prouveurs externes avec Why3

Le concept de cette tactique est de tenter de résoudre certains buts de Demon en appelant des prouveurs externes. L'essentiel du travail fait par Demon se trouve dans la traduction de ses buts dans un autre langage. Une telle tactique a été implémentée durant ce stage, elle est capable de traduire des problèmes de logique du premier ordre et d'arithmétique.

Présentation de Why3

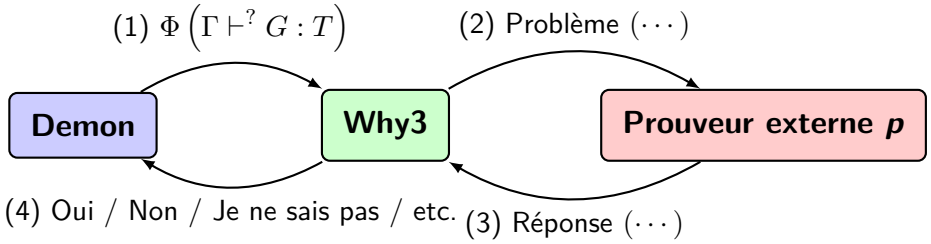
Why3 est un logiciel conçu pour la certification de programmes et une plateforme entre plusieurs prouveurs externes. Il dispose de son propre langage, WhyML, dans lequel il est possible d'écrire des programmes avec des annotations logiques, les *spécifications*, qui décrivent ce que doivent faire les programmes.

La particularité de Why3 est de reposer entièrement sur des *prouveurs externes* pour résoudre des buts. Ces prouveurs sont très divers, certains sont des assistants de preuve comme Coq, Isabelle ou PVS, conçus pour être utilisés de façon interactive ; d'autres sont des *SAT/SMT solvers* plutôt conçus pour résoudre des tâches efficacement.

Comme on souhaite seulement utiliser Why3 pour résoudre des formules logiques, on ignore la partie programmation de WhyML pour ne se concentrer que sur la partie logique.

Stratégie

Le processus général est décrit par la figure ci-dessous :



1. Traduire le but $\Gamma \vdash^? G : T$ (modélisé par la fonction Φ). En pratique, on dispose de l'AST Demon et des informations de typage de chaque symbole. On traduit vers WhyML, qu'on va assimiler à la logique du premier ordre typée.
2. Demander à Why3 de faire appel à un prouveur p pour résoudre le but.
3. Why3 obtient une réponse de p.
4. Why3 transmet la réponse à Demon. En cas de succès, on résout effectivement le but dans Demon. Dans tous les autres cas, on ne fait rien.

La stratégie adoptée à la dernière étape doit être justifiée : on part du principe que la traduction est **correcte**, ce qui signifie que si $\Phi\left(\Gamma \vdash^? G : T\right)$ est dérivable dans la logique du premier ordre, alors il existe bien G tel que $\Gamma \vdash G : T$ l'est dans Demon (*ie.* il existe une preuve de T dans le contexte Γ). La propriété réciproque est appelée *complétude*, mais elle n'est pas assurée par notre algorithme[2].

On a étendu le code de Demon pour implémenter cette tactique, en mettant à profit l'API de Why3 mise à disposition sous la forme d'une librairie pour OCaml.

2. Si toutefois on disposait de la complétude, cela nous autoriserait à abandonner un but dans Demon dans le cas où Why3 répondrait que ce but n'est pas valide. Comme on n'a pas la complétude, on ne fait rien dans ces cas de figure.

Dans la suite, on décrit le comportement de la tactique et on fournit les idées principales de l'implémentation. En particulier, on propose un formalisme pour Φ, assez imprécis toutefois, puisqu'il a été défini après l'implémentation et à partir de celle-ci.

Implémentation

L'API de Why3 fournit les fonctions nécessaires pour construire les termes de WhyML sous la forme d'arbres syntaxiques, et de créer les diverses déclarations qui forment les *théories* (qui sont en fait les buts de Why3).

Le fragment de WhyML qui nous intéresse correspond à la logique du premier ordre typée. On va donc les confondre à partir de maintenant.

La logique du premier ordre typée est la logique usuelle en mathématiques, à laquelle on ajoute la notion de type. Dans cette logique, chaque terme possède un type, ou *genre*, et tous les genres forment un ensemble $\mathcal{S}$. De plus, chaque fonction (resp. prédicat) possède une *signature* qui indique les types attendus de ses arguments et son type de retour (sauf les prédicats, qui n'ont pas de type de retour) ; seuls les termes bien typés sont autorisés par le langage. La notation $f :< s_1, \ldots, s_n; s >$ signifie que f est un symbole de fonction dont l'argument i est de genre s_i et s est le genre de retour. $p :< s_1, \ldots, s_n >$ est la même chose pour le symbole de prédicat p ; un prédicat n'a pas de genre de retour.

Les preuves en logique du premier ordre peuvent être construites à partir d'un petit ensemble de règles, qui forment la *déduction naturelle*. L'une de ces règles possède un statut spécial, il s'agit généralement du *tiers exclu* : $\dfrac{}{P \vee \neg P}\ \mathsf{EM}$. Si l'on retire cette règle, certains théorèmes ne sont plus démontrables, et on parle de *logique intuitioniste*.

Notre logique d'arrivée est donc la logique du premier ordre typée. L'objectif est de traduire le plus de buts possibles dans cette logique. Pour cela, on a procédé par étapes, en partant d'un fragment basique du langage de Dedukti puis en l'étendant à des cas plus complexes.

Avant de détailler ces étapes, qui sont au nombre de trois, on peut décrire le processus général de traduction d'un but $\Gamma \vdash^? G : T$.

1. En premier lieu, il faut définir les ensembles $\mathcal{S}$ (genres), Σ^F (symboles de fonction) et Σ^P (symboles de prédicats). Chaque fonction ou prédicat déclaré doit être associé à une *signature*. Appelons *signature du premier ordre* et notons Σ la donnée de tous ces ensembles.
2. Ensuite, pour chaque jugement $t : A$ dans le contexte Γ, on traduit A en une formule du premier ordre $\|A\|$. La suite de ces formules placées dans le même ordre est un contexte $\|\Gamma\|$ de la logique du premier ordre.
3. Enfin, on traduit la formule T en $\|T\|$.
   On définit alors $\Phi(\Gamma \vdash^? G : T) := \|\Gamma\| \vdash_1^\Sigma \|T\|$, où $\vdash_1^\Sigma$ est le symbole de dérivation dans la logique du premier ordre pour la signature qu'on a définie.

La majeure différence entre ce formalisme et la réalité de l'implémentation est que l'étape 1 n'est pas séparée du reste. En effet, on déclare les nouveaux genres et symboles à

la volée au cours de la traduction, au fur et à mesure qu'on les rencontre. Cela évite bien sûr de parcourir l'ensemble du contexte Γ, ce qui est potentiellement coûteux.

Par ailleurs, tout Γ n'est en réalité pas parcouru : on ignore le contexte global pour se contenter du contexte local, qui contient les termes introduits depuis le début de la preuve courante. Cela nous limite dans la mesure où, si un lemme précédemment démontré dans Demon est nécessaire pour démontrer le but, le prouveur externe n'y aura pas accès.

Il convient maintenant de détailler la façon dont est construite la signature du premier ordre Σ, ainsi que les formules $\|A\|$. On commence par un cas de base : les termes qui correspondent à la logique du premier ordre minimale, celle ne comportant que les connecteurs $\Rightarrow$ et $\forall$.

Traduction minimale

On a déjà abordé la correspondance entre la logique de Dedukti et le fragment minimal de la logique du premier ordre : $\to$ correspond à $\Rightarrow$, et $\prod$ correspond à $\forall$. La syntaxe de Demon intègre d'ailleurs cette idée, puisque les opérateurs de Dedukti y sont remplacés par les opérateurs mathématiques $\Rightarrow$ et $\forall$.

On procède ainsi pour obtenir une signature du premier ordre à partir d'un contexte Γ dans Demon :

— Pour chaque symbole s tel que $\Gamma \vdash \mathsf{s} : \mathit{Type}$, on déclare $s \in \mathcal{S}$;

— Pour chaque symbole f tel que $\Gamma \vdash \mathsf{f} : A_1 \to \cdots \to A_n \to B$, où $\Gamma \vdash A_i : \mathit{Type}$ et $\Gamma \vdash B : \mathit{Type}$ (les A_i et B sont des genres), on déclare $f \in \Sigma^F$ (symbole de fonction) de signature $< A_1, \ldots, A_n; B >$;

— Pour chaque symbole P tel que $\Gamma \vdash \mathsf{P} : A_1 \to \cdots \to A_n \to \mathit{Type}$, où $\Gamma \vdash A_i : \mathit{Type}$, on déclare $P \in \Sigma^P$ (symbole de prédicat) de signature $< A_1, \ldots, A_n >$.

La traduction d'un type de Demon en proposition du premier ordre est alors définie ainsi :

— $\| \prod \mathsf{x}^{\mathsf{s}}. B\| := \forall x^s \, \|b\|$

— $\|A \to B\| := \|A\| \Rightarrow \|B\|$

— $\|\mathsf{P}\ t_1 \cdots t_n\| := P(|t_1|, \ldots, |t_n|)$

— $|\mathsf{f}\ t_1 \cdots t_n| := f(|t_1|, \ldots, |t_n|)$

On s'est implicitement placé dans les cas où s, P et f appartiennent aux cas décrits plus haut, donc où s, P et f sont bien définis dans $\mathcal{S}$, Σ^P et Σ^F. Dans tous les autres cas, la traduction échoue ; la fonction n'est donc bien sûr pas *totale* sur tous les termes de Demon.

Les cas non couverts correspondent en général à des quantifications d'ordre supérieur, quand on quantifie sur des objets dont les types ne correspondent plus à des genres. Par exemple, le type $\prod P^{A \to \mathit{Type}}. \prod x^A. P\ x$ n'est pas traduisible, car $A \to \mathit{Type}$ ne peut pas être associé à un genre.

Les symboles de variables x sont traduits en x à la volée (avec les bonnes informations de type) et placés dans un contexte local (seulement visibles dans les sous-termes dans lesquels

ils sont liés). On ne détaille pas cette partie de l'implémentation par manque d'intérêt et par souci de simplification.

Le code ci-dessous est une démonstration dans Demon de la tactique implémentée sur un cas de théorème dans la logique minimale.

```
// Initialize a Why3 environment.
$yinit

// Load driver for the prover Alt-Ergo.
$yprover altergo

// Define a first-order signature:
// - three sorts s, s' and s'';
// - four function symbols a, b, f and g;
// - one predicate symbol P.

s : Type
s' : Type
s'' : Type

a : s
b : s'
f : s ⇒ s ⇒ s''
g : s' ⇒ s

P : s'' ⇒ Type

// Use demon terms to represent a theorem in first-order logic,
// and use Alt-Ergo through Why3 to prove it.

$theorem_1 : (∀ x y, P (f x (g y))) ⇒ P (f a (g b))
        $intro H
        $y altergo
```

La commande $\texttt{\$yinit}$ charge l'environement Why3, elle doit être invoquée une fois ; la commande $\texttt{\$yprover}$ charge le *driver* pour le prouveur donné en argument, elle doit être invoquée une fois pour utiliser ce prouveur.

Après les déclarations des symboles qu'on va utiliser en tant que genres, fonctions et prédicats, on déclare un nouveau *théorème*, en préfixant son identifiant par $ pour entrer en *mode preuve*, d'où l'on pourra utiliser des tactiques pour construire le terme de preuve.

La première tactique $\texttt{\$intro}$ déplace une hypothèse dans le contexte en la nommant H, de sorte que le but à prouver devienne $\Gamma, \forall x^{s}\, \forall y^{s'}\, \mathsf{P}\ (\mathsf{f}\ \mathsf{x}\ (\mathsf{g}\ \mathsf{y})) \vdash^{?} \mathsf{P}\ (\mathsf{f}\ \mathsf{a}\ (\mathsf{g}\ \mathsf{b}))$. On montre ainsi comment les hypothèses du contextes sont traduites en tant qu'*axiomes*.

La seconde tactique $\texttt{\$y altergo}$ réalise la traduction en WhyML du but courant, qui est

ensuite résolu par Alt-Ergo. La preuve est ensuite achevée dans Demon. Ci-dessous figure le fichier WhyML généré par la tactique[3].

```
theory Task
  type s

  type s'

  type s''

  constant a : s

  constant b : s'

  function f s s : s''

  function g s' : s

  predicate P s''

  axiom H :
    forall x:s.
     forall y:s'. P (f x (g y))

  goal theorem_1 :
    P (f a (g b))
end
```

La logique minimale est limitante ; pour accéder aux ordre connecteurs de la logique classique, on utilise des encodages dans Dedukti. Voyons maintenant comment il est possible de traduire des buts faisant intervenir ces encodages.

Traduction de formules du premier ordre

On se base ici sur un encodage de la logique du premier ordre déjà connu, dont on ne détaillera pas l'implémentation.

Cet encodage définit un type *Prop* dont les éléments sont des codes de propositions. L'opérateur ε associe à tout code P le type $\varepsilon(P)$. Un nouvel opérateur est déclaré pour chaque connecteur logique : top, bot, not, imp, and, or, forall, exists.

Le code suivant présente ces déclarations, ainsi que les théorèmes de la déduction naturelle, qui doivent être prouvables à partir des définitions des symboles. On a toutefois masqué ces définitions ainsi que les preuves des théorèmes pour alléger la présentation.

3. On a simplement modifié les identifiants à la main pour améliorer la lisibilité. La tactique génère des identifiants plus longs de façon à éviter les collisions de noms.

```
Prop : Type
Set : Type
t : Prop ⇒ Type

top : Prop
bot : Prop
not : Prop ⇒ Prop
imp : Prop ⇒ Prop ⇒ Prop
and : Prop ⇒ Prop ⇒ Prop
or  : Prop ⇒ Prop ⇒ Prop
fa  : (Set ⇒ Prop) ⇒ Prop
ex  : (Set ⇒ Prop) ⇒ Prop

$trivial : t top
$I_bot : ∀ p, t p ⇒ t (not p) ⇒ t bot
$E_bot : ∀ p, t bot ⇒ t p
$I_imp : ∀ p q, (t p ⇒ t q) ⇒ t (imp p q)
$E_imp : ∀ p q, t p ⇒ t (imp p q) ⇒ t q
$I_and : ∀ p q, t p ⇒ t q ⇒ t (and p q)
$E_and_l : ∀ p q, t (and p q) ⇒ t p
$E_and_r : ∀ p q, t (and p q) ⇒ t q
$I_or_l : ∀ p q, t p ⇒ t (or p q)
$I_or_r : ∀ p q, t q ⇒ t (or p q)
$E_or : ∀ p q r, t (or p q) ⇒ (t p ⇒ t r) ⇒ (t q ⇒ t r) ⇒ t r
$I_fa : ∀ p, (∀ x, t (p x)) ⇒ t (fa p)
$E_fa : ∀ p a, t (fa p) ⇒ t (p a)
$I_ex : ∀ p a, t (p a) ⇒ t (ex p)
$E_ex : ∀ p q, t (ex p) ⇒ (∀ x, t (p x) ⇒ t q) ⇒ t q
$EM : ∀ p, t (or p (not p))
```

Ces énoncés correspondent à la déduction naturelle, leur validité entraîne la correction de la traduction. Cette implémentation n'est pas la plus générale, puisqu'elle ne permet de quantifier que sur un seul genre, nommé Set. Il est possible de définir des quantificateurs forall et exists sur plusieurs genres avec des encodages supplémentaires.

On étend à présent la traduction en définissant un comportement spécial pour ces opérateurs.

— $\|\varepsilon\ A\| := \|A\|$
— $\|\mathsf{top}\| := \top$
— $\|\mathsf{bot}\| := \bot$
— $\|\mathsf{not}\ A\| := \neg\|A\|$
— $\|\mathsf{imp}\ A\ B\| := \|A\| \Rightarrow \|B\|$
— $\|\mathsf{and}\ A\ B\| := \|A\| \wedge \|B\|$

— $\|\mathsf{or}\ A\ B\| := \|A\| \vee \|B\|$
— $\|\mathsf{forall}\ (\lambda \mathsf{x}^{\mathsf{s}}.\ A)\| := \forall x^s\ \|A\|$
— $\|\mathsf{exists}\ (\lambda \mathsf{x}^{\mathsf{s}}.\ A)\| := \exists x^s\ \|A\|$

Ci-dessous le prolongement du fichier Demon précédent, qui présente deux démonstrations de théorèmes de la logique du premier ordre. L'opérateur y est nommé `t`.

```
// logic.dem implements first-order logic
$o logic

// Declare logical connectives.

$ydef Prop logic.Prop

$ydef prf  logic.t
$ydef top  logic.top
$ydef bot  logic.bot
$ydef not  logic.not
$ydef imp  logic.imp
$ydef and  logic.and
$ydef or   logic.or
$ydef fa   logic.fa
$ydef ex   logic.ex


c : Set
h : Set ⇒ Set ⇒ Set

Q : Set ⇒ Prop


$theorem_2 : t (imp (not (ex (fun x -> Q (h x x)))) (fa (fun x -> not (Q (h x x)))))
        $y altergo

$theorem_3 : ∀ x, t (imp (and (Q (h c x)) (Q (h x c))) (or (Q (h c x)) (Q (h x c))))
        $intro x
```

La commande `$o logic` importe les déclarations et définitions d'un fichier `logic.dem` (après compilation de ce dernier). Ce fichier est une implémentation de la logique du premier ordre telle qu'on l'a présentée plus haut.

Les commandes commençant par `$ydef` permettent d'indiquer à la tactique Why3 quels symboles dans Demon représentent les connecteurs logiques.

Enfin, on prouve deux buts avec Alt-Ergo : $\Gamma \vdash \left(\neg\exists x^{\mathsf{Set}}\, Q(h(x,x))\right) \Rightarrow \left(\forall x^{\mathsf{Set}}\, \neg Q(h(x,x))\right)$ et $\Gamma, x : \mathsf{Set}, Q(h(c,x)) \wedge Q(h(x,c)) \vdash Q(h(c,x)) \vee Q(h(x,c))$. Ci-dessous figurent les deux traductions WhyML de ces buts.

```
theory Task
  type Set

  function h Set Set : Set

  predicate Q Set

  goal theorem_2 :
    not (exists x:Set. Q (h x x)) ->
     (forall y:Set. not Q (h y y))
end
```

```
theory Task
  type Set

  constant x : Set

  constant c : Set

  function h Set Set : Set

  predicate Q Set

  predicate Set

  axiom x : Set

  goal theorem_3 :
    Q (h c x) /\
     Q (h x c) ->
     Q (h c x) \/
      Q (h x c)
end
```

On remarque que Set est déclaré deux fois : en tant que type, et en tant que prédicat sans argument ; de même, x est à la fois déclaré en tant que constante de type Set et axiome pour la proposition Set. Il ne s'agit pas d'un bug, mais d'une ambiguïté sur la dualité type-proposition : impossible en effet de déterminer si x : Set doit être interprété comme 'x est de type Set" ou "on a l'axiome Set identifié par x". En général, pour un type donné, on ne peut pas savoir si ce type représente un type-objet ou un type-proposition. On peut même imaginer des buts dans lesquels un type sera les deux à la fois, par exemple $A : \mathit{Type}, x : A, \varepsilon : A \to \mathit{Type} \vdash^{?} G : A \to \varepsilon\ x$. C'est pourquoi on a choisi de réaliser cette double-déclaration lorsque c'est nécessaire.

En mathématiques, la logique du premier ordre n'est qu'un cadre ; en général, on disposera de fonctions et prédicats dans Σ et d'axiomes supplémentaires pour décrire leur comportement. L'exemple le plus simple est l'*arithmétique*, qui définit les entiers naturels, et les fonctions d'addition et de multiplication. Voyons maintenant comment étendre encore la traduction à de telles logiques.

Traduction de formules de l'arithmétique

On peut étendre la traduction à des formules arithmétiques assez simplement, si l'on dispose d'une implémentation de l'arithmétique dans Demon. Cette implémentation peut prendre la forme suivante, basée sur l'arithmétique de Peano :

```
1  Nat : Type
2  O : Nat
3  S : Nat ⇒ Nat
4
5  plus : Nat ⇒ Nat ⇒ Nat
6  plus O m -> m
7  plus (S n) m -> S (plus n m)
8
9  mult : Nat ⇒ Nat ⇒ Nat
10 mult O m -> O
11 mult (S n) m -> plus n (mult n m)
12
13
14 eq : Nat ⇒ Nat ⇒ Type
15 eq_tt : eq O O
16 eq (S n) (S m) -> eq n m
17
18 le : Nat ⇒ Nat ⇒ Type
19 le_tt : le O O
20 le (S m) (S n) -> le m n
21 le m (S n) -> le m n
22
23 lt : Nat ⇒ Nat ⇒ Type
24 lt m n -> le (S m) n
25
26 ge : Nat ⇒ Nat ⇒ Type
27 ge m n -> le n m
28
29 gt : Nat ⇒ Nat ⇒ Type
30 gt m n -> lt n m
```

On y définit deux constructeurs pour les entiers naturels, le zéro O et la fonction successeur S ; puis les deux opérations de l'addition et de la multiplication ; et enfin cinq

prédicats, l'égalité sur les entiers et les relations d'ordre usuelles.

On aurait pu aussi déclarer les prédicats comme des fonctions vers l'univers *Prop* précédent. La tactique implémentée est capable de gérer ce cas de figure.

Why3 ne dispose pas d'une implémentation des entiers naturels, mais des entiers relatifs, avec un type *Int*. L'extension de la tactique passe par plusieurs points : identifier le type *Nat* de Demon avec *Int* dans Why3 ; pour chaque fonction f de signature $< s_1, \ldots, s_n; Int >$ que l'on déclare, ajouter l'axiome $\forall {x_1}^{s_1} \ldots \forall {x_n}^{s_n} f(x_1, \ldots, x_n) \geq 0$; associer chaque opérateur arithmétique dans Demon à son opérateur correspondant dans Why3, à l'exception de S, qui doit être défini autrement. Cette dernière étape est résumée par :

- $|\mathsf{O}| := 0$
- $|\mathsf{S}\ n| := |n| + 1$
- $|\mathsf{plus}\ m\ n| := |m| + |n|$
- $|\mathsf{mult}\ m\ n| := |m| \times |n|$
- $\|\mathsf{eq}\ m\ n\| := |m| = |n|$
- $\|\mathsf{le}\ m\ n\| := |m| \leq |n|$
- $\|\mathsf{lt}\ m\ n\| := |m| < |n|$
- $\|\mathsf{ge}\ m\ n\| := |m| \geq |n|$
- $\|\mathsf{gt}\ m\ n\| := |m| > |n|$

Un exemple de théorème dans l'arithmétique :

```
$o arith

$yd Nat  arith.Nat

$yd neq  arith.eq
$yd lt   arith.lt
$yd le   arith.le
$yd gt   arith.gt
$yd ge   arith.ge

$yd zero arith.O
$yd succ arith.S
$yd plus arith.plus
$yd mult arith.mult

$theorem_4 : ∀ m n, arith.le (arith.S m) n ⇒ arith.gt n m
        $i m p
        $y altergo
```

```
theory Task
  use int.Int

  constant p : int

  constant m : int

  predicate Nat

  axiom m : Nat

  axiom p : Nat

  goal theorem_4 : (1 + m) <= p -> p > m
end
```

Commentaire et Perspectives

La tactique `$y` a été implémentée sans qu'on ait formalisé son fonctionnement au préalable. Sa correction n'a donc pas été réellement prouvée, on dispose seulement d'*arguments* en sa faveur, qui pour la plupart reviennent au fait qu'on s'est basé sur des encodages de logiques dans Dedukti déjà connus (tous présentés dans [1]). On justifie cette démarche par le fait que la priorité du stage était d'obtenir une tactique fonctionnelle, tandis que la preuve de sa correction était secondaire.

La tactique comporte quelques failles, par exemple liées au fait que la logique des prouveurs est souvent classique. Il est ainsi possible de prouver la *loi de Peirce* en faisant appel à un de ces prouveurs, alors que cela est censé être impossible dans Dedukti :

```
$yi
$yp altergo

P : Type
Q : Type

$peirce : ((P ⇒ Q) ⇒ P) ⇒ P
        $y altergo
```

Un problème important est celui de la définition d'un terme de preuve dans Demon. Why3 ne retournant aucune *trace de preuve* avec sa réponse, il n'est actuellement pas possible de constuire un tel terme. Pour compléter tout de même les preuves en cas de réponse positive, on se contente donc d'ajouter le but résolu en tant qu'*axiome* dans Demon.

Le problème des traces de preuve est inhérent aux prouveurs que l'on utilise, car tous ne se soucient pas forcément de la preuve d'un résultat. C'est en particulier le cas des

SAT/SMT solvers, qui sont avant tout conçus pour être efficaces et s'appuient sur des heuristiques plutôt que des méthodes de déduction rigoureuses.

Une solution possible est de contourner le problème en passant par des prouveurs capables de retourner des preuves. *iProverModulo* et *ZenonModulo* peuvent produire des sorties Dedukti et acceptent le format TPTP (format standard de problème logique). En corollaire au développement de la tactique `$y`, on a fait en sorte de produire une sortie des buts en TPTP : il suffit en fait de récupérer la sortie WhyML, que Why3 est ensuite capable de traduire au format TPTP.

Enfin, l'extension à l'arithmétique n'est pas sans défauts ; il manque notamment l'ajout des axiomes $\forall {x_1}^{s_1} \ldots \forall {x_n}^{s_n}\, f(x_1, \ldots, x_n) \geq 0$ permettant de se restreindre aux entiers naturels. On peut cependant se demander si l'approche consistant à se servir de la librairie *Int* est bien raisonnable. Elle est en effet difficilement compatible avec la volonté d'extraire des preuves de Why3, puisqu'une preuve dans l'arithmétique de Peano n'a pas grand-chose à voir avec une preuve dans l'algèbre des entiers relatifs telle que Why3 l'implémente.

Réécriture de buts

La réécriture dont il est question ici n'a rien à voir avec les *systèmes de réécriture* que Dedukti utilise. Le principe est de modifier l'expression de termes sous des hypothèses d'égalité. Cette méthode de raisonnement est extrêmement courante en mathématiques et paraît naturelle, comme on le verra dans l'exemple introductif. En logique, cependant, elle est plus difficile à mettre en place, car on doit définir la notion d'*égalité* dans un premier temps.

Au cours de ce stage, on a pu définir un cadre formel dans lequel exprimer des preuves de réécriture, ce qui pourra servir de base à une tactique de réécriture. La tactique en elle-même n'a cependant pas été implémentée, car une partie du problème a soulevé des problèmes qui n'avaient pas été anticipés, ce qui fera l'objet de la fin de cette section.

Énoncé du problème

Exemple introductif

On se place dans un contexte Γ dans lequel on dispose de la logique du premier ordre et de l'arithmétique (pour l'exemple). Supposons qu'on essaie de prouver le but suivant :

$$\Gamma, H : \forall n, n + 0 = n \vdash^{?} G_1 : (m + 0) \times p = m \times p$$

Pour finir la preuve, on aimerait pouvoir utiliser l'hypothèse H pour *réécrire* le sous-terme $(m + 0)$ en m, puisque H exprime que $n + 0 = n$ pour tout entier naturel n.

On obtiendrait le but

$$\Gamma, H : \forall n, n + 0 = n \vdash^{?} G_2 : m \times p = m \times p$$

qui permettrait de conclure par réflexivité de $=$.

En mathématiques, cela paraitraît naturel, mais notre tâche est de le formaliser dans un système logique. Or, on peut déjà décomposer ce problème en deux sous-problèmes :

1. S'il existe, trouver un sous-terme dans le but qui puisse être réécrit. Plus formellement, il s'agit de trouver une *instance* du terme $n+0$, dans lequel n est une variable, dans le terme-but ; ce terme est $m+0$. En même temps, on doit déterminer la *substitution* des variables correspondante, c'est-à-dire la fonction $\{n \mapsto m\}$ dans l'exemple.
2. Déterminer un *terme de preuve* permettant de passer de G_2 à G_1[4], c'est-à-dire un λ-terme dans le langage de Demon qu'on note F et tel que $F\,G_2 = G_1$.

Il s'avère que la première moitié du problème est difficile dans le cadre de la théorie des types sur laquelle sont basés Dedukti et Demon. La seconde moitié a toutefois été résolue pour un certain formalisme dans lequel l'égalité est définie. C'est ce qu'on se propose de développer à présent, en commençant par présenter la définition de l'égalité.

Notion d'égalité

Une définition classique de l'égalité mathématique est celle provenant du *principe d'égalité* de Leibniz : deux objets sont égaux s'ils vérifient les mêmes propriétés (on dit qu'ils sont *indiscernables*). Mathématiquement, cela peut s'exprimer en logique du second ordre par l'équation :

$$\forall P\,\forall x\,\forall y\; x = y \iff \Big(P(y) \iff P(x)\Big)$$

Pour le problème qui nous concerne, on choisit une forme alternative de cette définition, qui est la suivante :

$$\forall x\; x = x \tag{Refl.}$$
$$\forall P\,\forall x\,\forall y\; x = y \Rightarrow P(y) \Rightarrow P(x) \tag{Ind.}$$

Ces definitions sont bien équivalentes ; pour s'en convaincre, on peut démontrer les propriétés de symétrie et transitivité de $=$ à partir de (Refl.) et (Ind.). Quant aux motivations derrière ce choix, elles tiennent au fait que les termes dont on fera correspondre le type à ces formules seront plus simples à utiliser.

Développons à présent une interprétation de ces formules dans Dedukti.

Soit un contexte Γ. On se donne un *encodage* quelconque, c'est-à-dire un type U et un opérateur ε sur U qui donne des types. Formellement, $\Gamma \vdash U : \mathit{Type}$ et $\Gamma \vdash \varepsilon : U \to \mathit{Type}$.

U est le type des codes d'*autres* types, qu'on peut définir par des règles de réécriture par exemple. On utiliser ces types pour représenter en même temps les *types d'objets* et les *propositions*, ce qui n'est pas gênant (puisque "les types sont des propositions").

Enfin, on définit le prédicat d'égalité :

4. Et non pas l'inverse. Pour rappel, on cherche à prouver $(m+0)\times p = m \times p$, c'est-à-dire à construire un terme de ce type. G_1 dénote ce terme que l'on cherche à construire. G_2 dénote le terme d'un type différent, mais à partir duquel on sait construire G_1. En résumé, c'est bien G_1 qui est fonction de G_2.

$$\Gamma \vdash \mathsf{Eq} : \prod A^U.\, \varepsilon(A) \to \varepsilon(A) \to U$$

Eq est une fonction prenant pour premier argument un *code de type* A, pour deuxième et troisième arguments deux objets de type $\varepsilon(A)$, et retournant un *code de type*, qu'on interprètera en général comme un code de proposition.

Enfin, pour que Eq se comporte comme le prédicat d'égalité, on impose que deux termes dont les types puissent être interprétés comme les principes (Refl.) et (Ind.) soient dérivables dans Γ. Cela s'énonce :

$$\Gamma \vdash \mathsf{eqRefl} : \prod A^U.\, \prod x^{\varepsilon(A)}.\, \varepsilon(\mathsf{Eq}\ A\ x\ x) \tag{EqRefl.}$$

$$\Gamma \vdash \mathsf{eqInd} : \prod A^U.\, \prod P^{\varepsilon(A)\to U}.\, \prod x^{\varepsilon(A)}.\, \prod y^{\varepsilon(A)}.\, \varepsilon(\mathsf{Eq}\ A\ x\ y) \to \varepsilon(P\ y) \to \varepsilon(P\ x) \tag{EqInd.}$$

Les correspondances avec (Refl.) et (Ind.) sont assez directes. On a quantifié en plus sur le *code de type* A, et on utilise $\varepsilon(A)$ pour accéder au "vrai" type correspondant à A. Le prédicat unaire P est une fonction sur $\varepsilon(A)$ retournant un *code de type*, qu'on interprétera comme un code de proposition en général. Si x est de type $\varepsilon(A)$, alors un objet de type $\varepsilon(P\ x)$ est une *preuve* de la proposition correspondant à $P\ x$.

Terme de preuve de réécriture

On va maintenant énoncer le problème de réécriture dans le cas général. On se place dans un contexte Γ dans lequel les termes Eq, eqRefl et eqInd tels qu'ils sont définis plus haut sont dérivables.

Quelques définitions préliminaires sont nécessaires pour introduire l'énoncé du problème.

Définition 2. *$\mathcal{V}$ est un ensemble de variables notées $x_1, \ldots, x_n$ et tel que*

$$\Gamma, \Delta_{i-1} \vdash x_i : A_i : \mathit{Type}^{5} \tag{CondVar.}$$

pour tout i, où $\Delta_i := x_1 : A_1, \ldots, x_i : A_i$. On note $\Delta := \Delta_n$.

L'ensemble $\mathcal{V}$ est la suite de variables sur lesquelles on va quantifier l'hypothèse de réécriture, qui correspondra à la formule $\forall {x_1}^{A_1} \ldots \forall {x_n}^{A_n}\, \ell = r$. Δ est donc le contexte *local* dans lequel la formule $\ell = r$ a un sens.

Un point important : on ne suppose aucune restriction sur les types $A_1, \ldots, A_n$: *chaque A_i peut dépendre des x_j pour $j < i$.*

On suppose à présent acquises les notion classique de *substitution*, comme il est trop long d'introduire tous ses résultats basiques. Une substitution σ sur un ensemble $\mathcal{V}$ est une fonction de domaine fini inclus dans $\mathcal{V}$ vers les termes du language. Si t est un terme, $t\sigma$ est

5. "$\Gamma \vdash t : A : B$" est une notation pour "$\Gamma \vdash t : A$ et $\Gamma \vdash A : B$".

le terme obtenu en remplaçant toutes les occurrences libres de chaque variable x par $\sigma(x)$. On note parfois $t\,[u\backslash x]$ si le domaine est réduit à $\{x\}$ et $\sigma(x) = u$.

On introduit aussi la notion de *position* : une position p dans un terme t est un pointeur sur un sous-terme de t, que l'on note $t|_p$. On note $t[u]_p$ le terme obtenu en remplaçant $t|_p$ par u dans t.

Ces notions sont définies de façon plus complète dans [2], au chapitre 3.

Le lemme qui suit se montrera utile pour la démonstration principale :

Lemme 1. *Soit σ une substitution sur $\mathcal{V}$ telle que*

$$\Gamma \vdash \sigma(x_i) : A_i\sigma \qquad \text{(CondType)}$$

pour tout i. Alors pour tout terme t de type A dans le contexte Γ, on a $\Gamma \vdash t\sigma : A\sigma$.

Une substitution qui respecte la condition (CondType) est "de bonne qualité", puisqu'elle substitue chaque variable de son domaine par un terme de même type – mais dans lequel on a aussi substitué les éventuelles occurrences d'autres variables par la même substitution. Chaque A_i peut en effet dépendre des x_j déclarés avant, voir (CondVar.).

On admet le lemme (1). Sa démonstration est possible par une induction sur la structure des termes de types de $\lambda\Pi$.

On énonce à présent le problème de réécriture :

Définition 3. *Soient A, ℓ et r tels que :*

$$\Gamma, \Delta \vdash A : U$$
$$\Gamma, \Delta \vdash \ell : \varepsilon(A)$$
$$\Gamma, \Delta \vdash r : \varepsilon(A)$$

Soit un terme C tel que $\Gamma \vdash C : U$. On considère le but $\Gamma \vdash^? G : \varepsilon(C)$.
On fait les hypothèses suivantes :

i. Il existe un terme H tel que $\Gamma, \Delta \vdash H : \varepsilon(\mathsf{Eq}\ A\ \ell\ r)$;

ii. Il existe une position p et une substitution σ sur $\mathcal{V}$ tel que $C|_p = \ell\sigma$;

iii. La substitution σ vérifie (CondType).

On énonce alors le problème de réécriture comme suit : sous les hypothèses précédentes, trouver un terme F tel que $\Gamma \vdash F(G') : \varepsilon(C)$, où G' est le terme de preuve du but réécrit : $\Gamma \vdash G' : \varepsilon\,(C[r\sigma]_p)$.

ℓ et r sont les deux expressions égales d'après l'hypothèse i., les variables de $\mathcal{V}$ y apparaissent libres. Toutes deux ont le type encodé par A, qui peut aussi dépendre des variables de $\mathcal{V}$.

On considère un but de la forme $\varepsilon(C)$ où C est un code de type, car la définition de l'égalité qu'on s'est donnée ne permet de travailler que sur des propositions encodées de la sorte.

L'hypothèse ii. exprime qu'on a trouvé une *instance* du terme gauche ℓ dans le but C. L'hypothèse iii. assure que l'instance trouvée est cohérente vis-à-vis des types.

$C[r\sigma]_p$ est le but qu'on souhaite obtenir après application de la tactique : il s'agit de C dans lequel on a remplacé l'instance $\ell\sigma$ par $r\sigma$. On peut définir $G := F(G')$ pour avoir $\Gamma \vdash G : \varepsilon(C)$ comme souhaité – reste encore à définir F, ce qui est l'objet du théorème suivant.

Théorème 2. *Soit le problème 3. Sous les hypothèses énoncées, il existe toujours un terme F solution du problème. Il s'agit du terme*

$$F := \mathsf{eqInd}\ A\sigma\ \left(\lambda z^{\varepsilon(A\sigma)}.\, C[z]_p\right)\ \ell\sigma\ r\sigma\ H\sigma$$

Démonstration. Il convient de vérifier le type de chaque argument donné à eqInd ; voir la définition (EqInd.).

(On néglige certains jugements dans les contextes quand ils ne sont pas nécessaires.)

⋆ $\Gamma \vdash A\sigma : U$

Évident à partir du lemme (1), sachant que $U\sigma = U$. En effet, $\Gamma \vdash U : \mathit{Type}$, donc U ne dépend d'aucune variable de $\mathcal{V}$.

⋆ $\Gamma, A\sigma : U \vdash \ell\sigma : \varepsilon(A\sigma)$

(On prouve pour le troisième argument avant le deuxième pour l'utiliser ensuite. Ce n'est pas gênant dans la mesure où le troisième argument est indépendant du deuxième.)

Évident à partir du lemme (1), sachant que $\big(\varepsilon(A)\big)\sigma = \varepsilon(A\sigma)$.

⋆ $\Gamma, A\sigma : U \vdash \left(\lambda z^{\varepsilon(A\sigma)}.\, C[z]_p\right) : \varepsilon(A\sigma) \to U$

D'après ii., $C = C[\ell\sigma]_p$, d'où $\Gamma \vdash C[\ell\sigma]_p : U$.

Il est clair qu'échanger un sous-terme pour un autre sous-terme de type égal ne modifie pas le type du terme entier, donc $\Gamma, z : \varepsilon(A\sigma) \vdash C[z]_p : U$, puis $\Gamma \vdash \lambda z^{\varepsilon(A\sigma)}.\, C[z]_p : \varepsilon(A\sigma) \to U$.

⋆ $\Gamma, A\sigma : U \vdash r\sigma : \varepsilon(A\sigma)$

Idem que $\ell\sigma$.

⋆ $\Gamma, A\sigma : U, \ell\sigma : \varepsilon(A\sigma), r\sigma : \varepsilon(A\sigma) \vdash H\sigma : \varepsilon(\mathsf{Eq}\ A\sigma\ \ell\sigma\ r\sigma)$

Il est clair que les arguments de Eq sont bien typés. Le reste est évident à partir du lemme (1), sachant que $\big(\varepsilon(\mathsf{Eq}\ A\ \ell\ r)\big)\sigma = \varepsilon(\mathsf{Eq}\ A\sigma\ \ell\sigma\ r\sigma)$.

Par applications successives de la règle App de $\lambda\Pi$, on obtient alors que le terme F a le type

$$\Gamma \vdash F : \varepsilon\left(\left(\lambda z^{\varepsilon(A\sigma)}.\, C[z]_p\right)\ (\ell\sigma)\right) \to \varepsilon\left(\left(\lambda z^{\varepsilon(A\sigma)}.\, C[z]_p\right)\ (r\sigma)\right)$$

On peut ensuite beta-réduire[6] ce type et invoquer la règle Conv pour obtenir

$$\Gamma \vdash F : \varepsilon\,(C[r\sigma]_p) \to \varepsilon\,(C[\ell\sigma]_p)$$

Si $\Gamma \vdash G' : \varepsilon\,(C[r\sigma]_p)$, alors on a bien $\Gamma \vdash F(G') : \varepsilon(C)$ puisque $C[\ell\sigma]_p = C$ par hypothèse.

□

Le terme de preuve pour la réécriture peut donc toujours être défini, sous réserve que toutes les conditions de 3 sont remplies. En particulier, ii. et iii. exigent de la part d'une implémentation d'une tactique de réécriture que de tels p et σ soient trouvés, ce qui nous amène au problème du *filtrage*.

Filtrage

On présente une première approche au problème de filtrage, qui reste très peu satisfaisante. Elle est en fait défectueuse dans certains cas ; on concluera par un exemple.

Cette approche est directement inspirée des algorithmes de filtrage du premier ordre. On pourra se référer à [2] pour une présentation plus développée. Le filtrage du premier ordre est un problème résolu, dont l'algorithme standard est coûteux par rapport à la taille du terme, mais efficace en pratique sur les cas d'utilisation dont a le plus souvent besoin.

Le table 2 présente un système, qu'on peut voir comme un algorithme de filtrage : la notation "P ; σ" dénote un ensemble de problèmes de filtrage P avec pour solution partielle σ. On note $t \ll^? t'$ les éléments de P, et $x \mapsto t$ les éléments de σ.

Il est clair que tous les ensembles σ construits à partir de ce système sont bien des substitutions sur $\mathcal{V}$, c'est-à-dire des fonctions de domaine fini inclus dans $\mathcal{V}$ (noté $\mathrm{Dom}(\sigma)$). La première condition de la règle Elim l'assure. Les deux conditions supplémentaires assurent que σ respecte aussi la condition souhaitée (CondType).

Définition 4. *Soit θ une substitution sur $\mathcal{V}$. θ est solution de P ; σ si et seulement si*

- *$\sigma \subset \theta$, c'est-à-dire que pour tout $x \in \mathrm{Dom}(\sigma)$, $x \in \mathrm{Dom}(\theta)$ et $\theta(x) = \sigma(x)$;*
- *$\forall (t \ll^? t') \in P, t\theta = t'$;*

On note "$\bot$" un problème n'ayant pas de solution. Pour tout σ, il est clair que σ est solution du problème $\emptyset$; σ. Une implémentation de la recherche d'une instance de ℓ dans le but C basée serait conçue sur le modèle du code suivant.

6. En λ-calcul pur, la *beta-réduction* est la règle modélisant l'exécution d'une fonction à laquelle on a fourni un argument : $(\lambda x.\, t)\ u \longrightarrow_\beta t\,[u\backslash x]$.

$$\frac{\left\{f\ t_1\ \cdots\ t_n \ll^? f\ u_1\ \cdots\ u_n\right\} \uplus P\ ;\ \sigma}{\left\{t_1 \ll^? u_1,\ \ldots,\ t_n \ll^? u_n\right\} \cup P\ ;\ \sigma}\ \textsf{Decomp}$$

$$p+n=m \quad \frac{\left\{x\ t_1\ \cdots\ t_n \ll^? f\ u_1\ \cdots\ u_p\ u_{p+1}\ \cdots\ u_m\right\} \uplus P\ ;\ \sigma}{\left\{x \ll^? f\ u_1\ \cdots\ u_p,\ t_1 \ll^? u_{p+1},\ \ldots,\ t_n \ll^? u_{p+n}\right\} \cup P\ ;\ \sigma}\ \textsf{Match}$$

$$f \neq g \quad \frac{\left\{f\ t_1\ \cdots\ t_n \ll^? g\ u_1\ \cdots\ u_m\right\} \uplus P\ ;\ \sigma}{\bot}\ \textsf{Symb-Clash}$$

$$t \neq t' \quad \frac{\left\{x \ll^? t\right\} \cup P\ ;\ \{x \mapsto t'\} \cup \sigma}{\bot}\ \textsf{Var-Clash}$$

$$\begin{array}{c} x_i \in \mathrm{Dom}(\sigma) \Rightarrow \sigma(x_i) = t \\ A_i\sigma \text{ est défini} \\ \Gamma \vdash t : A_i\sigma \end{array} \quad \frac{\left\{x_i \ll^? t\right\} \uplus P\ ;\ \sigma}{P\ ;\ \{x_i \mapsto t\} \cup \sigma}\ \textsf{Elim}$$

TABLE 2 – Présentation de l'algorithme de filtrage

```
for (p ∈ Pos(C)) {
        P := {ℓ ≪? C|_p};
        σ := ∅;
        do {
                if (P = ∅)
                        return (σ, p);
        } while (applyRule(P, σ));
}
return ⊥;
```

Pour tous P et σ, si l'ensemble des règles de 2 applicables à $P\ ;\ \sigma$ n'est pas vide, la fonction applyRule en choisit une – selon une *stratégie* à définir – et l'applique à $P\ ;\ \sigma$ avant de retourner true ; si cet ensemble est vide, la fonction retourne false.

La stratégie pour parcourir l'ensemble $\mathrm{Pos}(C)$, c'est-à-dire l'ordre dans lequel on évalue

les sous-termes de C, est également à définir.

On peut se convaincre simplement que l'algorithme est correct : il retourne une solution σ quand le problème est de la forme σ, et repose entièrement sur la correction de 2.

Théorème 3. *Le système 2 est correct, c'est-à-dire que pour toute règle* $H \dfrac{P_1 \,;\, \sigma_1}{P_2 \,;\, \sigma_2}\,\mathsf{R}$ *, si θ est solution de $P_2 \,;\, \sigma_2$ sous l'hypothèse H, alors θ est aussi solution de $P_1 \,;\, \sigma_1$.*

Démonstration. La preuve est triviale pour les règles Symb-Clash et Var-Clash, puisque $\bot$ n'a pas de solution.

⋆ Decomp

Soit θ solution de $\left\{t_1 \ll^? u_1, \ldots, t_n \ll^? u_n\right\} \cup P \,;\, \sigma$.

$\sigma \subset \theta$ par hypothèse. Pour tout problème $t \ll^? t'$ dans P, $t\theta = t'$ par hypothèse. De plus, $\Big(f\ t_1\ \cdots\ t_n\Big)\theta = f\ u_1\ \cdots\ u_n$. En effet :

$$\begin{aligned}\Big(f\ t_1\ \cdots\ t_n\Big)\theta &= f\ t_1\theta\ \cdots\ t_n\theta \\ &= f\ u_1\ \cdots\ u_n \qquad \text{par hypothèse}\end{aligned}$$

Donc θ est solution de $\left\{f\ t_1\ \cdots\ t_n \ll^? f\ u_1\ \cdots\ u_n\right\} \uplus P \,;\, \sigma$.

⋆ Match

La preuve est analogue à la preuve pour Decomp. On vérifie simplement :

$$\begin{aligned}\Big(x\ t_1\ \cdots\ t_n\Big)\theta &= x\theta\ t_1\theta\ \cdots\ t_n\theta \\ &= \Big(f\ u_1\ \cdots\ u_p\Big)\ u_{p+1}\ \cdots\ u_{p+n} \qquad \text{par hypothèse} \\ &= f\ u_1\ \cdots\ u_p\ u_{p+1}\ \cdots\ u_m \qquad \text{car } p+n=m \text{ par hypothèse}\end{aligned}$$

⋆ Elim

Soit θ solution de $P \ ; \ \{x_i \mapsto t\} \cup \sigma$. Il est clair que $\sigma \subset \theta$ et que $t\theta = t'$ pour tout $(t \ll^? t') \in P$.

Puisque $\{x_i \mapsto t\} \cup \sigma \subset \theta$ par hypothèse, on a $\theta(x_i) = t$, soit $x_i\theta = t$.

□

Le programme termine s'il parvient à dériver $\emptyset \,;\, \sigma$ à partir du problème initial, auquel cas la solution triviale σ est retournée. Par le théorème 3, σ est aussi solution du problème initial $\left\{\ell \ll^? C|_p\right\} \,;\, \emptyset$. Donc $\ell\sigma = C|_p$ par la définition 4.

Le problème majeur de ce système est qu'il n'étend pas le filtrage aux types. On présente un cas simple sur lequel le système ne peut pas terminer : la fonction d'identité, et sa relation d'égalité caractéristique :

$$\Gamma \vdash \forall A^U\, \forall x^{\varepsilon(A)}\, x =_{\varepsilon(A)} \mathsf{Id}\ A\ x$$

où $\Gamma \vdash \mathsf{Id} : \forall A^U\, \forall x^{\varepsilon(A)}\, \varepsilon(A)$. "$\ell =_{\varepsilon(A)} r$" est une notation pour "$\varepsilon\,(\mathsf{Eq}\ A\ \ell\ r)$".

On a volontairement choisi cet ordre pour les membres gauche et droite. Dans le cas d'un filtrage purement *syntaxique* (c'est-à-dire ne cherchant à satisfaire que la condition ii.), le problème $x \ll^? t$ est trivial, puisque $\sigma := \{x \mapsto t\}$ convient. Cependant, on a aussi besoin que le type de t soit cohérent par rapport à celui de x (la condition iii.). Plus précisément, dans cet exemple, on a besoin que le type de t soit de la forme $\varepsilon(A)$, où A est une autre variable du problème.

Or, le système que l'on a présenté est incapable d'observer le type de t ; il se borne à vérifier que le type de x, soit $\varepsilon(A)$, est bien défini avant d'appliquer la règle Elim sur $\left\{x \ll^? t\right\}$. Mais comme A ne peut être instancié dans cet exemple, le système n'aboutira jamais à une solution.

Il semble maintenant absolument nécessaire pour un algorithme de filtrage de comparer les types, et de les comparer *avant les termes*. Cette approche offre aussi un avantage d'efficacité, puisque le système actuel semble capable de commencer à s'exécuter entre des termes dont les types sont incompatibles ; en comparant d'abord les types, la réponse négative sera plus immédiate.

Conclusion

Impact sociétal et environnemental

La consommation énergétique durant le PFE se résume essentiellement en ces points : utilisation quotidienne d'un ordinateur (de 9 heures à 17 heures environ), climatisation du bureau dès fin mai, et un déplacement à Eindhoven d'une semaine pour l'*International School on Rewriting* (ISR). Ce déplacement s'est fait par avion jusqu'à Amsterdam, puis en bus jusqu'à Eindhoven ; de même pour le retour, dans le sens inverse.

Il est difficile de parler de l'impact sur la société qu'aura le travail réalisé durant le PFE, étant donné que cette thématique appartient à la recherche fondamentale. À l'échelle de la vérification en général, cependant, l'enjeu de ces travaux est important dans tous les domaines où la sécurité informatique est cruciale : en particulier l'énergie, le transport, le médical ; on peut aussi mentionner la vérification formelle des protocoles de sécurité. L'objectif à long terme est de garantir la sûreté des programmes.

Dans le cas de Dedukti, toutefois, cet enjeu est lointain, d'une part parce que l'approche "Curry-Howard" ne représente qu'une toute petite partie de la vérification formelle (l'interprétation abstraite, par exemple, semble plus adaptée à la vérification de programmes impératifs), et d'autre part parce que Deducteam étudie des questions plus abstraites et fondamentales.

Bilan technique

L'objectif initial était de développer deux nouvelles tactiques de preuve dans Demon.

La première tactique a été implémentée. Sa correction n'a pas été prouvée formellement, mais on dispose toutefois d'arguments en sa faveur (des travaux antérieurs sur l'expression de preuves dans Dedukti). Une étude plus théorique de la traduction de buts Demon en WhyML serait néanmoins intéressante, surtout si on souhaite l'étendre à des buts plus complexes. Dans sa dernière version, la tactique peut traduire des expressions de la logique du premier ordre classique et de l'arithmétique, pour un certain cas d'utilisation cependant.

Les recherches sur la seconde tactique se sont heurtés à des problèmes qui n'avaient pas été anticipés, au sujet de la difficulté du problème de filtrage sur les termes typés comme dans $\lambda\Pi$. La direction la plus logique à présent serait d'étudier ce problème et les travaux qui ont dû être réalisés dessus par le passé. La structure générale d'un terme de preuve a toutefois pu être définie de façon rigoureuse avant la fin du stage.

Bilan personnel

Mon objectif personnel pour ce PFE était de confirmer mon intérêt pour la recherche et pour la logique. C'est maintenant chose faite.

Outre mon sujet, ce stage m'a permis d'apprendre une quantité de choses sur des thèmes proches, à travers des lectures ou simplement des discussions entre collègues, sans oublier ma participation à l'ISR. Toutes ces nouvelles connaissances m'offrent une meilleure visibilité sur le domaine dans lequel j'aimerais travailler.

J'ai jugé toutefois qu'il était encore tôt pour m'engager dans une thèse, la logique en informatique étant une discipline vaste et exigeante. Je n'avais pas choisi ce sujet de stage dans l'objectif de le poursuivre en thèse l'année prochaine. C'est pourquoi j'ai décidé de m'inscrire au Master Parisien de Recherche en Informatique (MPRI) en M2 pour me spécialiser en informatique théorique l'année prochaine, avant de faire une thèse.

L'Égalité dans Dedukti

Le code ci-dessous est la définition de l'égalité qu'on a proposée dans la section 5.1.2. On y démontre la symétrie et la transitivité de cette opération à l'aide du principe d'induction.

```
#NAME equality.

U : Type.
ε : U → Type.

Eq : A:U → ε A → ε A → U.

(; Reflexivity ;)
eqRefl : A:U → x:(ε A) → ε (Eq A x x).

(; Induction Principle ;)
eqInd : A:U → P:(ε A → U) → x:(ε A) → y:(ε A) →
        ε (Eq A x y) → ε (P y) → ε (P x).


(; Symetry ;)
def eqSym : A:U → x:(ε A) → y:(ε A) →
                ε (Eq A x y) → ε (Eq A y x) :=
        A:U => x:ε A => y:ε A => H:ε (Eq A x y) =>
                eqInd A (z:ε A => Eq A y z) x y H (eqRefl A y).

(; Transitivity ;)
def eqTrans : A:U → x:(ε A) → y:(ε A) → z:(ε A) →
                ε (Eq A x y) → ε (Eq A y z) → ε (Eq A x z) :=
        A:U => x:ε A => y:ε A => z:ε A =>
                H1:ε (Eq A x y) => H2:ε (Eq A y z) =>
                        eqInd A (u:ε A => Eq A u z) x y H1 H2.
```

Références

[1] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti : a logical framework based on the $\lambda\pi$-calculus modulo theory. http://www.lsv.fr/~dowek/Publi/expressing.pdf.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[3] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 Platform*, Septembre 2014.

[4] William Alvin Howard. The formulæ-as-types notions of construction. In J. P. Seldin and J. R. Hindley, editors, *Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pages 479–490. Academic Press, 1980.

Note

Les quelques pages qui suivent ne figurent pas dans le rapport original, mais ont été ajoutées pour corriger une erreur importante qui y figure en page 26 dans la preuve du théorème 2.

On peut y lire : “Il est clair qu'échanger un sous-terme pour un autre sous-terme de type égal ne modifie pas le type du terme entier”, cette phrase ayant pour but de justifier la proposition $\Gamma, z : \varepsilon(A\sigma) \vdash C[z]_p : U$ sachant $\Gamma \vdash C : U$ et $\Gamma \vdash C|_p : \varepsilon(A\sigma)$. Il se trouve que l'affirmation est fausse dans le cas général, ce qu'on illustrera par deux exemples, mais que le résultat est tout de même vrai pour une certaine classe de sous-termes de C, ce qui fera l'objet d'une preuve.

Reformulons d'abord le résultat à démontrer sous une forme plus générale. Étant donné un terme t et une position p dans t, sachant qu'on a les typages $\Gamma \vdash t : A$ et $\Gamma, \Delta \vdash t|_p : B$, on se demande si le séquent $\Gamma, x : B \vdash t[x]_p : A$ est dérivable. On se retreint au cas où t est un “objet”, ce qui signifie que A est un type : $\Gamma \vdash A : \mathit{Type}$. Cela réduit le nombre de cas à traiter et se trouve être suffisant pour nos besoins.

Deux contre-exemples

Le premier cas problématique est celui où le type A du terme dépend du sous-terme considéré. Posons $\Gamma = \mathit{Nat} : \mathit{Type}, O : \mathit{Nat}, S : \mathit{Nat} \to \mathit{Nat}, \mathit{Vec} : \mathit{Nat} \to \mathit{Type}, f : \Pi n^{\mathit{Nat}}.\mathit{Vec}\, n$. On a les typages $\Gamma \vdash f\, O : \mathit{Vec}\, O$ et $\Gamma \vdash O : \mathit{Nat}$, mais pas $\Gamma, x : \mathit{Nat} \vdash f\, x : \mathit{Vec}\, O$. Le typage attendu serait $\Gamma, x : \mathit{Nat} \vdash f\, x : \mathit{Vec}\, x$, ce qui suggère une recherche du sous-terme supplémentaire dans A ; le problème semble cependant loin d'être trivial, surtout en présence de réécriture sur les types.

Le deuxième cas est celui où le type B du sous-terme dépend d'un terme abstrait dans t. Posons $\Gamma = U : \mathit{Type}, \varepsilon : U \to \mathit{Type}$. On a les typages $\Gamma \vdash \lambda\alpha^U.\lambda x^{\varepsilon(\alpha)}.x : \Pi\alpha^U.\varepsilon(\alpha) \to \varepsilon(\alpha)$ et $\Gamma, \alpha : U \vdash \lambda x^{\varepsilon(\alpha)}.x : \varepsilon(\alpha) \to \varepsilon(\alpha)$, mais pas $\Gamma, x : \varepsilon(\alpha) \to \varepsilon(\alpha) \vdash \lambda\alpha^U.x : \Pi\alpha^U.\varepsilon(\alpha) \to \varepsilon(\alpha)$. Un moyen d'arranger le problème serait de remplacer le sous-terme non pas par une variable de type B, mais une variable de type $\Pi\bar{x}^{\bar{A}}.B$ où les variables $\bar{x}$ et types $\bar{A}$ sont ceux du contexte local Δ du sous-terme, et d'appliquer les termes de dépendance après remplacement : $\Gamma, x : \Pi\alpha^U.\varepsilon(\alpha) \to \varepsilon(\alpha) \vdash \lambda\alpha^U.x\,\alpha : \Pi\alpha^U.\varepsilon(\alpha) \to \varepsilon(\alpha)$.

Théorème affaibli

Si u est un sous-terme de t (terme-objet), on notera $u = t|_p$ et on dira que “p est une position dans t”. La relation de sous-terme est définie inductivement ; si p est une position dans t, on a l'un des quatre cas suivants :

— $t|_p = t$
— $t = t_1\, t_2$ et $t|_p = t_1|_{p'}$
— $t = t_1\, t_2$ et $t|_p = t_2|_{p'}$
— $t = \lambda x.u$ et $t|_p = u|_{p'}$

Si t pouvait aussi représenter un type, on devrait considérer le cas du produit dépendant.

On va admettre deux lemmes pour faciliter la preuve.

Lemme. *Si $\Gamma \vdash t : A : Type$, alors pour toute position p dans t il existe Δ et B tels que $\Gamma, \Delta \vdash t|_p : B$.*

Cela est prouvable par induction sur la relation de sous-terme, en utilisant le lemme d'inversion quand t est une application ou une abstraction.

Lemme. *Si $\Gamma, x : A, y : B, \Delta \vdash t : C$ et que x n'apparaît pas libre dans B, alors on peut dériver $\Gamma, y : B, x : A, \Delta \vdash t : C$.*

La dérivation du nouveau séquent est triviale par induction sur t, mais le point important est que le nouveau contexte est toujours bien formé.

Théorème. *On suppose $\Gamma \vdash t : A : Type$. Soit p une position dans t. Soient Δ et B tels que $\Gamma, \Delta \vdash t|_p : B$. On suppose les deux conditions suivantes respectées :*

i. Si $t|_p$ est à droite d'une application, le type du terme à sa gauche est un produit non-dépendant (une flèche) ;
ii. Si $t|_p$ est sous une abstraction, son type B ne dépend pas du terme abstrait.

Alors $\Gamma, x : B \vdash t[x]_p : A$.

Démonstration. Fixons t et p et raisonnons par induction sur la relation de sous-terme.

Cas n° 1 : $t|_p = t$

Alors Δ est vide et $A \equiv B$ par unicité des types. De plus, $t[x]_p = x$. On a la dérivation suivante :

$$\dfrac{\dfrac{}{\Gamma, x : B \vdash x : B}\ Var \qquad A \equiv B}{\Gamma, x : B \vdash t[x]_p : A}\ Conv$$

Cas n° 2 : $t = t_1\, t_2$ et $t|_p = t_1|_{p'}$

Alors $t[x]_p = t_1[x]_{p'}\, t_2$. Par inversion, il existe C et D tels que $\Gamma \vdash t_1 : \Pi x^C.D$ et $\Gamma \vdash t_2 : C$ sont dérivables et $A \equiv D\,[x \backslash t_2]$. Soient Δ et B tels que $\Gamma, \Delta \vdash t_1|_{p'} : B$. Par hypothèse d'induction, on a une dérivation de $\Gamma, x : B \vdash t_1[x]_{p'} : \Pi x^C.D$. On complète en la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{\pi_1}{\Gamma, x : B \vdash t_1[x]_{p'} : \Pi x^C.D} \quad \dfrac{\dfrac{\pi_2}{\Gamma \vdash t_2 : C}}{\Gamma, x : B \vdash t_2 : C}\,\mathit{Weak}}{\Gamma, x : B \vdash t_1[x]_{p'}\, t_2 : D\,[x \backslash t_2]}\,\mathit{App} \qquad A \equiv D\,[x \backslash t_2]}{\Gamma, x : B \vdash t[x]_p : A}\,\mathit{Conv}$$

Cas n° 3 : $t = t_1\, t_2$ et $t|_p = t_2|_{p'}$

Alors $t[x]_p = t_1\, t_2[x]_{p'}$. Par inversion et sous l'hypothèse i., il existe C et D tels que $\Gamma \vdash t_1 : C \to D$ et $\Gamma \vdash t_2 : C$ sont dérivables et $A \equiv D$. Soient Δ et B tels que $\Gamma, \Delta \vdash t_1|_{p'} : B$. Par hypothèse d'induction, on a une dérivation de $\Gamma, x : B \vdash t_2[x]_{p'} : C$. On complète en la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{\pi_1}{\Gamma \vdash t_1 : C \to D}}{\Gamma, x : B \vdash t_1 : C \to D}\,\mathit{Weak} \qquad \dfrac{\pi_2}{\Gamma, x : B \vdash t_2[x]_{p'} : C}}{\Gamma, x : B \vdash t_1\, t_2[x]_{p'} : D}\,\mathit{App} \qquad A \equiv D}{\Gamma, x : B \vdash t[x]_p : A}\,\mathit{Conv}$$

Cas n° 4 : $t = \lambda y^C.u$ et $t|_p = u|_{p'}$

Alors $t[x]_p = \lambda y^C.u[x]_{p'}$. Par inversion, il existe D tel que $\Gamma, y : C \vdash u : D$ est dérivable et $A \equiv \Pi y^C.D$. Soient Δ et B tels que $\Gamma, y : C, \Delta \vdash u|_{p'} : B$. Par hypothèse d'induction, on a une dérivation de $\Gamma, y : C, x : B \vdash u[x]_{p'} : D$. Par l'hypothèse ii., y n'apparaît pas libre dans B, donc par le dernier lemme on a aussi une dérivation de $\Gamma, x : B, y : C \vdash u[x]_{p'} : D$. On complète en la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma, x : B, y : C \vdash u[x]_{p'} : D}}{\Gamma, x : B \vdash \lambda y^C.u[x]_{p'} : \Pi y^C.D}\,\mathit{Abs} \qquad A \equiv \Pi y^C.D}{\Gamma, x : B \vdash t[x]_p : A}\,\mathit{Conv}$$

□

En résumé, le résultat tel qu'on l'a énoncé n'est valide que sur une classe restreinte de sous-termes, telle que les effets des types dépendants n'interfèrent pas. Les deux contre-exemples qu'on a présentés suggèrent que l'énoncé doit être reformulé pour prendre en compte ces effets afin d'être peut-être valide.