Suivi mi-parcours

Certification de transformations logiques

Quentin Garchery[1,2]

[1] LRI, Univ. Paris-Sud, CNRS UMR8623, Orsay, Université Paris-Saclay
Quentin.Garchery@lri.fr
[2] Inria, Université Paris-Saclay, 91120 Palaiseau

1 Contexte général

Les outils de vérification de programmes ont pour rôle de garantir qu'un programme vérifie un *contrat* donné, portant par exemple sur des propriétés fonctionnelles du programme, sur l'absence d'erreur à l'exécution, etc. Ces outils se distinguent par l'expressivité des contrats possibles, le paradigme choisi pour établir le contrat, mais aussi par la taille de la *base de confiance*, le code auquel on doit faire confiance pour garantir que les contrats sur les programmes sont bien validés. Un objectif général de tels outils est d'être le plus automatisés possible, et, dans de nombreux cas, cette automatisation se fait à l'aide de code venant s'ajouter à la base de confiance.

Dans ce contexte, mon objectif est d'améliorer la confiance que l'on accorde au logiciel de vérification déductive de programmes Why3 [4]. En effet, en permettant l'utilisation de dizaines de prouveurs automatiques différents, Why3 offre une grande automatisation au prix d'une base de confiance considérablement grande.

Pour vérifier un programme, Why3 se base sur le paradigme suivant : (1) à partir du programme et de ses annotations (contrat et indications telles que les invariants de boucle), l'outil génère une *tâche de preuve*, énoncé logique dont la validité entraîne la correction du programme vis-à-vis du contrat ; (2) l'outil cherche à valider cette tâche de preuve, de manière automatique ou interactive.

Dans le cas général, la tâche de preuve générée est complexe et ne peut pas être directement validée par les prouveurs automatiques disponibles. Pour résoudre ce problème, Why3 permet l'utilisation de transformations logiques, celles-ci permettant de transformer la tâche initiale en une liste de tâches résultantes, en espérant que celles-ci soient plus simples à valider. Ces transformations logiques peuvent être utilisées de manière interactive, ce qui permet à l'utilisateur de décomposer le problème à sa convenance. Les transformations logiques sont aussi utilisées automatiquement lorsqu'un prouveur automatique est appelé afin de traduire la tâche de preuve considérée dans la logique du prouveur automatique.

Jusqu'à maintenant, mes travaux de thèse se sont focalisés sur la confiance que l'on peut obtenir dans les transformations logiques pour plusieurs raisons. D'une part, les transformations logiques sont omniprésentes car nécessaires à chaque appel de prouveur automatiques. D'autre part, leur implantation comporte déjà plus de 17000 lignes de code OCaml, code qui est dans la base de confiance de Why3 et qu'on aimerait vérifier. Enfin, contrairement à la génération de la tâche de preuve initiale, les transformations logiques restent dans un même contexte logique, celui des tâches de preuve, et sont donc de ce point de vue plus facilement abordables.

Mon approche est fondée sur l'utilisation de certificats vérifiables par une tierce partie, selon l'approche classiquement dénommée *sceptique*. L'idée est d'instrumenter l'outil afin qu'il génère un *certificat* de preuve, celui-ci permettant d'assurer *a posteriori* la validité de chaque exécution de l'outil. Ce certificat peut être vérifié par un outil externe qui, lui, peut avoir une

petite base de confiance. À l'inverse, une approche *autarcique* aurait cherché à prouver le code des transformations. L'approche sceptique est particulièrement indiquée lorsque l'on souhaite assurer la modularité de la certification et ajouter petit à petit des certificats. Je mettrai donc l'accent sur la modularité et la composabilité dans l'utilisation des certificats.

La partie 2 présente la structure des certificats et les propriétés de correction attendues, ainsi que les propriétés de composabilité. Ensuite, la partie 3 détaille mon implantation d'un vérificateur de certificats en OCaml qui forme ainsi la base de confiance du nouveau mécanisme de transformations certifiantes de Why3. Afin de réduire encore cette base de confiance, je propose, dans la partie 4, un deuxième vérificateur de certificats qui se base sur la plateforme Dedukti [2]. Ce rapport a pour objectif de donner une vision d'ensemble des problèmes rencontrés, une présentation plus détaillée [8] étant également disponible.

Les développements les plus récents de ce travail sont visibles à l'adresse `https://gitlab.inria.fr/why3/why3/tree/cert_split`. Une version stable est disponible à l'adresse `https://gitlab.inria.fr/why3/why3/tree/cert`, et des détails pour la compilation et l'utilisation sont donnés dans le fichier `README_JFLA.md` à la racine du dépôt.

2 Certificats modulaires pour les transformations logiques

Présentons ici la méthode que j'ai utilisée pour obtenir des transformations certifiantes. Un premier pas technique est nécessaire pour contourner les contraintes liées à l'API de Why3, décrite brièvement en section 2.1 : afin de s'abstraire de la représentation Why3 des tâches, la section 2.2 explique d'abord comment extraire le contenu logique de ces tâches. La section 2.3 présente le langage de certificats et leur sémantique attendue. Enfin, la section 2.4 s'intéresse à la composition des transformations certifiantes.

2.1 Tâches de preuves et transformations en Why3

Le formalisme logique de Why3 [3] est basé sur celui de la logique classique du premier ordre. Ce formalisme fournit un certain nombre de théories intégrées et d'extensions syntaxiques (par exemple les constructions `let` et `if`).

Les *tâches de preuve* dans Why3 sont composées d'une liste de prémisses et d'un but : un énoncé logique à démontrer dans ce contexte. La sémantique d'une tâche est simplement celle d'un séquent logique à un seul but écrit dans la signature correspondante.

Les *transformations logiques* sont essentiellement des fonctions OCaml implantées dans le code source de Why3 qui convertissent une tâche T en une liste des tâches $T_1, \ldots, T_n$. Une transformation est *correcte* si la validité des tâches générées implique la validité de la tâche initiale. Ainsi, les constructions étendues (comme `let` et `if`) peuvent être éliminées d'une tâche grâce à des transformations logiques appropriées. Cela permet par exemple d'appeler un prouveur qui ne supporte pas ces constructions.

2.2 Extraction du contenu logique des tâches

Pour représenter les tâches de preuve, je définis un nouveau type `ctask`. Par rapport à la représentation Why3 des tâches, les `ctask` expriment uniquement le contenu logique de celles-ci, sans mentionner les déclarations de types ni les signatures des symboles de fonction.

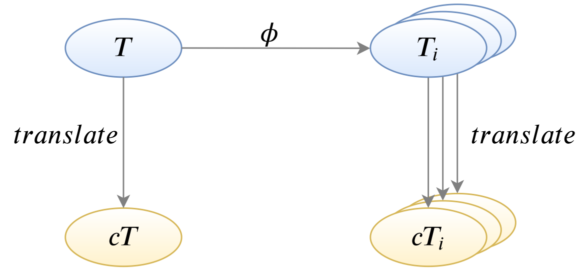
Les `ctask` sont ainsi représentées par des séquents, ce qu'on notera de la façon suivante :

$$H_1 : A_1, ..., H_m : A_m \vdash G_1 : B_1, ..., G_n : B_n$$

$$
\begin{array}{rclcl}
\textit{certif} & ::= & \mathsf{Hole} & \mid & \mathsf{Swap_neg}(\textit{ident}, \textit{certif}) \\
& \mid & \mathsf{Trivial}(\textit{ident}) & \mid & \mathsf{Destruct}(\textit{ident}, \textit{ident}, \textit{ident}, \textit{certif}) \\
& \mid & \mathsf{Axiom}(\textit{ident}, \textit{ident}) & \mid & \mathsf{Weakening}(\textit{ident}, \textit{certif}) \\
& \mid & \mathsf{Cut}(\textit{ident}, \textit{cterm}, \textit{certif}, \textit{certif}) & \mid & \mathsf{Intro_quant}(\textit{ident}, \textit{ident}, \textit{certif}) \\
& \mid & \mathsf{Split}(\textit{ident}, \textit{certif}, \textit{certif}) & \mid & \mathsf{Inst_quant}(\textit{ident}, \textit{ident}, \textit{cterm}, \textit{certif}) \\
& \mid & \mathsf{Unfold}(\textit{ident}, \textit{certif}) & \mid & \mathsf{Rewrite}(\textit{ident}, \textit{ident}, \textit{path}, \textit{bool}, \textit{certif}^{*})
\end{array}
$$

FIGURE 1 – Langage des certificats.

Je définis également une fonction de traduction *translate* qui, suivant l'approche sceptique, sera utilisée à chaque application d'une transformation ϕ, à la fois sur la tâche initiale T et sur les tâches résultantes T_i, comme indiqué sur la figure ci-contre.



On souhaite certifier que la validité de $T_1, \dots T_n$ entraîne celle de T, on doit donc vérifier que la validité de $cT_1, \dots, cT_n$ entraîne celle de cT et faire confiance à la traduction de tâches. Cette dernière apparaît dans les deux sens : la validité de cT doit entraîner celle de la tâche Why3 T et la validité de chaque tâche Why3 T_i doit entraîner celle de la tâche traduite cT_i. Il doit donc y avoir équi-validité entre une tâche et sa traduction. En ce sens, la traduction fait partie de la base de confiance. Si, par exemple, la traduction donne toujours $\vdash G : \top$, toutes les transformations seront automatiquement vérifiées. Heureusement, la traduction d'une tâche Why3 est un procédé simple qui revient essentiellement à retirer des champs dans un enregistrement.

Dans la suite, on supposera qu'une tâche Why3 ne diffère pas de sa traduction d'un point de vue logique. On peut alors, sans ambiguïté, utiliser la notation en séquents aussi bien pour les tâches Why3 que pour les *ctask*.

2.3 Certificats à trous

On suit l'approche sceptique : les transformations sont instrumentées afin de produire un certificat. La particularité de mon approche vient du fait que les transformations ne closent pas nécessairement la tâche courante et les certificats doivent refléter cette particularité. Un constructeur spécial, `Hole`, indique un « trou » dans un certificat, chaque trou devra correspondre à une tâche produite par cette transformation.

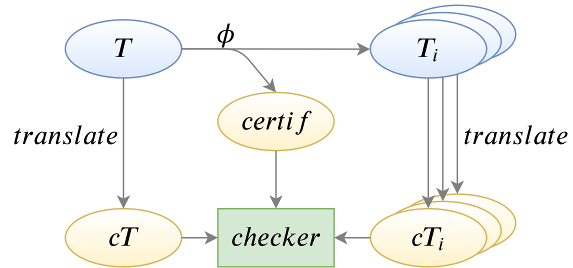
Definition 2.1. *Le langage des certificats est défini par la grammaire donnée figure 1, où le non-terminal cterm représente les termes et le non-terminal ident représente les noms de prémisses, de buts, ou de symboles logiques.*

La sémantique des certificats a été définie par un prédicat ternaire $T \stackrel{c}{\Leftarrow} S$, reliant une tâche initiale T, un certificat c et un ensemble de tâches S, qui affirme que le certificat c garantit que la validité des tâches de S entraîne celle de T. Par souci de concision, ce prédicat n'est pas défini formellement ici.

Voici néanmoins quelques exemples. Le certificat `Hole` est utilisé afin de garantir la validité des transformations qui ont des tâches résultantes. Celui-ci garantit notamment la validité d'une

transformation identité. Le certificat $\texttt{Axiom}(H, G)$ garantit la validité d'une transformation qui clôt une tâche ayant un but G identique à une prémisse H. Enfin, le certificat $\mathsf{Split}(P, c_1, c_2)$ garantit la validité d'une transformation qui commence par scinder une prémisse ou un but P et dont la validité du reste de la transformation sur ces deux tâches serait assurée par c_1 et c_2.

Vérificateurs de certificats. Un vérificateur de certificats est une procédure qui, étant donnés une tâche T, un certificat c et un ensemble de tâches S, tente de vérifier que $T \stackrel{c}{\Leftarrow} S$ est vrai. En pratique, un tel vérificateur *checker* s'applique à une tâche traduite cT, au certificat produit $certif$ et à l'ensemble des tâches résultantes traduites cT_i, comme indiqué sur la figure adjacente.



Definition 2.2. *Un tel vérificateur est* correct *si, lorsqu'il répond positivement, alors* $T \stackrel{c}{\Leftarrow} S$ *est dérivable.*

2.4 Composition des transformations certifiantes

Les certificats à trous permettent une certification *modulaire* selon deux sens : d'une part il est possible de certifier l'action d'une transformation sans devoir nécessairement certifier le traitement ultérieur des sous-tâches résultantes, d'autre part il est possible de composer les certificats pour certifier une transformation qui est obtenue par composition de transformations déjà certifiantes.

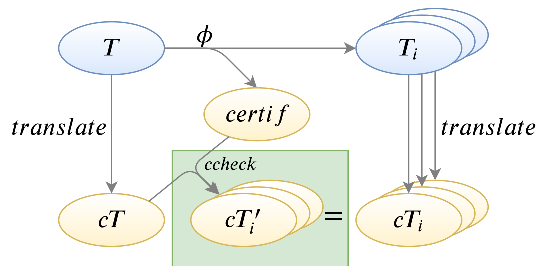
Une transformation certifiante produit un certificat en plus d'une liste de tâches résultantes. Pour composer de telles transformations il s'agit donc de définir la composition de certificats. On procède comme suit : l'application de la première transformation produit un certificat où chaque trou Hole correspond à une tache résultante à laquelle est appliquée la deuxième transformation, le certificat obtenu étant utilisé pour remplir le trou.

3 Vérificateur en OCaml

Le premier vérificateur proposé est implanté en OCaml. Dans cette première approche, j'ai implanté une version exécutable du jugement $T \stackrel{c}{\Leftarrow} S$ sous la forme d'une fonction OCaml ccheck qui, à partir d'une tâche T et d'un certificat c, reconstruit une *liste* de tâches L dont les éléments sont ceux de S.

Une transformation certifiante produit un certificat, ce qui permet de vérifier son résultat. Dans le cas du vérificateur en OCaml cette vérification se fait en appelant la fonction ccheck selon le schéma ci-après.

À partir de T, une tâche Why3, la transformation ϕ considérée donne *certif* et une liste de tâches T_i. La fonction $translate$ calcule alors les versions "abstraites" cT et cT_i des tâches de départ et d'arrivée, on appelle ccheck sur *certif* et cT pour obtenir une liste cT'_i et on vérifie point à point que les listes cT_i et cT'_i sont identiques.



La base de confiance de cette approche est constituée de la fonction d'abstraction des tâches $translate$ et de la fonction ccheck.

Réalisation de la fonction ccheck. L'implantation de ccheck procède naturellement par récurrence sur le certificat. Ce calcul vérifie les conditions d'application des règles, et lève des exceptions quand ce n'est pas le cas. Détaillons le fragment de code correspondant au constructeur $\mathsf{Split}(P, c_1, c_2)$. La fonction ccheck vérifie que la formule donnée par l'identifiant P est scindable, c'est-à-dire que c'est une conjonction si c'est un but ou une disjonction si c'est une prémisse. Si c'est le cas, ccheck s'appelle récursivement, par exemple dans le cas d'un but :

$$\mathsf{ccheck}\ (\mathsf{Split}(P, c_1, c_2))\ (\Gamma \vdash \Delta, P : A_1 \wedge A_2) \equiv \\ \mathsf{ccheck}\ c_1\ (\Gamma \vdash \Delta, P : A_1)\ @\ \mathsf{ccheck}\ c_2\ (\Gamma \vdash \Delta, P : A_2)$$

où @ est le symbole infixe pour la concaténation de listes.

Proposition 3.1 (Correction de ccheck)**.** *Pour tout certificat c et tâche T, si* ccheck c T *renvoie une liste L, alors il existe une dérivation de $T \overset{c}{\Leftarrow} S$, où S est l'ensemble des éléments de L.*

La preuve de cette proposition peut se faire par récurrence sur le certificat, en prenant tous les cas un à un. Il s'agit d'une preuve que l'on pourrait faire avec un assistant de preuve. Une telle preuve peut vite devenir fastidieuse en augmentant le nombre de certificats différents. J'ai donc décidé de me concentrer sur une deuxième approche, présentée ci-après.

4 Vérificateur basé sur Dedukti

La deuxième approche pour réaliser un vérificateur vise à se passer d'une preuve formalisée d'un théorème de correction comme la proposition 3.1. Mon implantation d'un tel vérificateur utilise à son cœur le vérificateur de preuve universel Dedukti. L'intérêt est de mettre à profit son mécanisme de règles de réécriture pour encoder simplement nos certificats. À chaque fois qu'une transformation Why3 est utilisée, une preuve Dedukti de correction est produite et peut être vérifiée par une implantation d'un vérificateur de types pour ce langage.

Plus précisément, on définit un encodage des tâches de preuves dans Dedukti, sous la forme d'un plongement superficiel : une tâche T est encodée en une formule Dedukti $\hat{T}$. Si sur une tâche T on applique une transformation certifiante produisant une liste de tâches T_i et un certificat c, alors on construit à partir de c un terme Dedukti qui est candidat pour une preuve de $\hat{T_1} \Rightarrow \cdots \Rightarrow \hat{T_n} \Rightarrow \hat{T}$, où $\Rightarrow$ est la flèche de Dedukti. Ce terme candidat est naturellement destiné à être transmis à Dedukti pour vérification.

Dans cette deuxième approche, la base de confiance est constituée de la fonction d'abstraction des tâches Why3, de la fonction de traduction des tâches abstraites vers Dedukti, et bien sûr de Dedukti lui-même et de l'encodage de la logique du premier ordre utilisé. En revanche, la fonction de traduction d'un certificat en un terme de preuve pour Dedukti n'est pas dans la base de confiance. En effet, si l'étape de traduction contient un *bug* alors la vérification risque d'échouer, mais, si elle réussit, la transformation Why3 initiale a bien fonctionné correctement sur la tâche considérée.

4.1 Traduction des tâches

Une tâche T est traduite en un type Dedukti $\hat{T}$ en utilisant un encodage superficiel. Si T est la tâche de preuve $H_1 : A_1, ..., H_m : A_m \vdash G_1 : B_1, ..., G_n : B_n$ alors le type $\hat{T}$ est $\hat{A_1} \Rightarrow ... \Rightarrow \hat{A_m} \Rightarrow \neg\hat{B_1} \Rightarrow ... \Rightarrow \neg\hat{B_n} \Rightarrow \bot$, où $\hat{A_i}$ est la traduction la formule A_i en Dedukti.

Grâce à cet encodage superficiel on peut se passer d'un traitement explicite du contexte qui s'appuierait par exemple sur des lemmes d'inversion et d'affaiblissement, ce qui rendrait la méthode inutilisable en pratique.

4.2 Traduction du certificat

4.2.1 Traduction des différentes étapes de certificat

Le terme de preuve dont on vérifiera qu'il a le type défini ci-dessus est obtenu à partir du certificat. Chaque étape de certificat est traduit en un terme Dedukti qui peut être vu comme une règle d'inférence. Par exemple, pour Split sur une prémisse qui est une disjonction, cette règle est l'introduction du $\vee$ à gauche :

$$\frac{P : A \vdash \qquad P : B \vdash}{P : A \vee B \vdash}$$

On définit donc une fois pour toutes en Dedukti un terme ayant pour type :

$$(A \Rightarrow \bot) \Rightarrow (B \Rightarrow \bot) \Rightarrow A \vee B \Rightarrow \bot$$

4.2.2 Traduction et élaboration du certificat

On souhaite obtenir un terme de preuve d'un type fourni par la traduction des tâches. Une première difficulté pour obtenir ce terme de preuve vient de la gestion des noms. Il faut d'une part faire attention à ce les noms des prémisses restent cohérents, ceux-ci apparaissant à la fois dans les tâches de preuve et dans le certificat. D'autre part, il faut généraliser suivant les variables libres qui apparaissent dans les tâches, l'ensemble de ces variables pouvant différer suivant la tâche considérée.

Par ailleurs, afin de faciliter la certification des transformations, le type des certificats est volontairement peu verbeux. En particulier, les étapes de certificat ne précisent pas quelles formules sont manipulées ni si ce sont des prémisses ou des buts. Cette ambiguïté demande un effort supplémentaire lors de la traduction des certificats vers des termes preuve Dedukti : c'est ce qu'on appelle l'élaboration des certificats. Les certificats sont reparcourus en partant de la tâche initiale, on peut alors en déduire le contexte d'application d'une étape de certificat. Celles-ci sont alors notamment enrichies des formules qu'elles manipulent, formules nécessaires pour définir des termes Dedukti clos.

5 Expérimentations

J'ai instrumenté une quinzaine nouvelles transformations de Why3 pour générer des certificats et les vérifier à la volée lors de chaque utilisation. La plus complexe d'entre elles étant une transformation nommée `blast`. Elle est obtenue par composition de transformations certifiantes plus élémentaires qui s'appuient uniquement sur la logique du premier ordre : décomposition de conjonctions, de disjonctions, d'implications et d'équivalences, résolution de tâches triviales, introduction d'hypothèses, etc. La méthode de composition de transformations certifiantes décrite dans la section 2.4 m'a permis de définir `blast` de façon à ce qu'elle s'appelle naturellement de manière récursive sur les sous-tâches qu'elle génère.

Par ailleurs, j'ai également cherché à certifier des transformations Why3 existantes. D'abord, une version légèrement simplifiée de la transformation `rewrite` permettant de réécrire dans les

termes a été rendue certifiante. Cette transformation implante un certain nombre de fonctionnalités de la transformation initiale pas encore les quantificateurs universels en tête de la formule à réécrire. Ensuite, dans certains cas, la transformation `split` a été rendue certifiante, les autres cas renvoient une erreur lorsqu'on cherche à les vérifier. Cette dernière transformation est complexe : de nombreux cas et paramètres sont à considérer et son comportement a été en partie défini afin que les tâches de preuves générées soient le plus facilement possible validées par les prouveurs automatiques.

Le vérificateur OCaml a été inclus dans Why3 et est disponible pour toutes les transformations déjà certifiantes. Il en est de même pour le mécanisme de vérification via Dedukti à l'exception de la transformation `rewrite` (à ce jour). À l'exception de cette dernière, nos expérimentations ont permis de montrer que les certificats générés par nos transformations certifiantes sont validés par Dedukti.

6 Conclusions et perspectives

Ce rapport a permis de présenter un cadre pour valider des transformations logiques « à petit pas » et composables. Cette validation se base sur une approche sceptique : génération d'un certificat pouvant être vérifié *a posteriori* par un outil externe. Ce travail se base sur une notion de certificat à trous, qui s'inspire naturellement fortement de notions analogues dans le contexte de termes-preuve comme les métavariables ou les variables existentielles. Dans le contexte de l'approche sceptique, l'utilisation de certificats à trous est nouvelle à ma connaissance. Elle permet de refléter, au sein des certificats, la modularité et la possibilité de chaîner les transformations : (i) les certificats eux-mêmes sont chaînables, grâce à la possibilité de laisser des trous qui seront comblés par la suite ; (ii) la vérification de ces certificats à trous nécessite de comparer les buts ouverts des certificats avec ceux produits par la transformation elle-même. Outre la flexibilité dans l'écriture des transformations apportée par l'approche sceptique, cela permet une très grande modularité puisque les transformations peuvent être instrumentées de manière totalement indépendante.

Cette approche a été implantée dans l'outil de preuve de programmes Why3 pour un ensemble de transformations en logique du premier ordre, afin de valider la démarche. Deux vérificateurs de certificats sont proposés, un non certifié développé en OCaml avec une approche calculatoire, et un développé dans le *framework* Dedukti à l'aide d'un plongement superficiel.

Perspectives. Un objectif à court terme est de poursuivre l'application à Why3, afin de rendre certifiantes les transformations `rewrite` et `split` et plus généralement les transformations les plus couramment utilisées. Cela nécessite d'étendre (a) le type `ctask` des tâches utilisé pour la validation, (b) le format de certificat et (c) nos vérificateurs. La certification des transformations d'élimination des types algébriques et des types polymorphes seront des défis importants. J'envisage notamment de me placer dans une logique typée et de passer à l'ordre supérieur.

Un passage à l'échelle va nécessairement poser des questions d'efficacité. D'un point de vue programmation, il s'agirait de se rapprocher de l'implantation concrète des tâches, notamment en étudiant la possibilité de conserver la mémoïsation (par une approche fonctionnelle). D'un point de vue plus théorique, la démarche présentée ici génère des certificats à petits grains et pouvant donc rapidement devenir volumineux lorsque plusieurs étapes sont combinées. Il conviendrait d'étudier la compression de ces certificats, et notamment comment elle pourrait être menée à la volée lors de la combinaison de certificats.

Remarquons que la méthode utilisée n'est pas propre à Why3 et peut s'appliquer à la certification d'encodages logiques en toute généralité. J'envisage de l'utiliser notamment pour certifier « à petits pas » des encodages de logiques d'assistants de preuve vers des prouveurs automatiques, afin de pouvoir combiner ces deux types de prouveurs.

Afin d'avoir une pleine confiance dans l'outil Why3, un objectif à plus long terme est de certifier de bout en bout la chaîne de Why3 : en aval, lier ce travail avec la vérification des preuves effectuées par les prouveurs automatiques, déjà proposée dans d'autres travaux [1, 5], et en amont, proposer une méthode de certification de la génération d'obligations de preuve. Pour aborder ce dernier point, j'envisage de proposer une traduction monadique [7, 6] étendue à WhyML, qui prenne donc en compte les régions et qui pourrait être dans la base de confiance dans un premier temps. Il faudrait ensuite montrer qu'on peut se ramener aux obligations alors obtenues à partir de celles produites par Why3.

Références

[1] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Thery, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *Certified Programs and Proofs*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2011.

[2] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Expressing theories in the $\lambda\Pi$-calculus modulo theory and in the Dedukti system. In *22nd International Conference on Types for Proofs and Programs*, 2016.

[3] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3 : Shepherd your herd of provers. In *International Workshop on Intermediate Verification Languages*, pages 53–64, 2011. https://hal.inria.fr/hal-00790310.

[4] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6) :709–727, 2015. See also http://toccata.lri.fr/gallery/fm2012comp.en.html.

[5] S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2010.

[6] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. PhD thesis, Paris 11, 1999.

[7] J.-C. Filliâtre, L. Gondelman, and A. Paskevich. A Pragmatic Type System for Deductive Verification. working paper or preprint, Feb. 2016.

[8] Q. Garchery, C. Keller, C. Marché, and A. Paskevich. Des transformations logiques passent leur certicat. In *JFLA 2020 - Journées Francophones des Langages Applicatifs*, Gruissan, France, Jan. 2020.