Des transformations logiques passent leur certicat

Quentin Garchery, Chantal Keller, Claude Marché, Andrei Paskevich

Des transformations logiques passent leur certificat

Quentin Garchery[1,2], Chantal Keller[1], Claude Marché[2,1], et Andrei Paskevich[1,2]

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

Résumé

Dans un contexte de vérification formelle de programmes, utilisant des démonstrateurs automatiques, la *base de confiance* des environnements de vérification est typiquement très large. Ainsi, un outil de vérification de programmes tel que Why3 comporte de nombreuses procédures complexes : génération de conditions de vérification, transformations logiques de tâches de preuve et interactions avec des démonstrateurs externes. En ne considérant que les transformations logiques dans Why3, leur implantation comporte déjà plus de 17000 lignes de code OCaml. Afin d'augmenter notre confiance dans la correction d'un tel outil de vérification, nous proposons un mécanisme de *transformations certifiantes*, produisant des certificats pouvant être validés par un outil externe, selon l'approche *sceptique*. Nous présentons ce mécanisme de génération de certificats et explorons deux méthodes pour les valider : une fondée sur un vérificateur dédié développé en OCaml, l'autre reposant sur le vérificateur de preuves universel Dedukti. Une spécificité de nos certificats est d'être « à petits grains » et composables, ce qui rend notre approche incrémentale, permettant d'ajouter graduellement de nouvelles transformations certifiantes.

1 Introduction

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*, c'est-à-dire la quantité de 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. Un extrême est représenté par Why3 [6], qui permet l'utilisation de dizaines de prouveurs automatiques différents, et leur fait confiance.

Pour réduire la base de confiance de ces outils, une famille d'entre eux se base sur des environnements généralistes d'assistance à la preuve, comme Coq ou Isabelle. Cela reporte la confiance sur celle que l'on a dans l'assistant, qui est lui-même basé sur un noyau que l'on essaye de garder le plus petit possible. Le revers de la médaille est que les preuves doivent se faire obligatoirement dans la logique fournie par l'environnement, et souvent de manière interactive. Pour éviter cet inconvénient, nous présentons dans cet article une approche *sceptique*, qui ne nécessite pas de plonger l'outil de preuve de programmes dans un assistant de preuve. L'idée est d'instrumenter l'outil pour générer un *certificat* de preuve, qui permet de convaincre *a posteriori* de 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.

Nous avons étudié cette approche dans le cas de l'outil Why3, qui se base sur le paradigme suivant : (1) à partir d'un programme annoté par un contrat et des indications (comme des invariants de boucle), l'outil génère des *obligations de preuve*, à savoir des formules logiques dont la validité entraîne la correction du programme vis-à-vis du contrat ; (2) l'outil cherche à valider ces obligations de preuve, de manière automatique et/ou interactive.

Nous rappelons ci-dessous le processus de preuve de programmes. Il se décompose en quatre grandes étapes. Considérons l'exemple suivant [10] annoté avec des spécifications formelles :

```
let f (a:array int) (x:int) : int
  requires { a.length ≥ 1000 ∧ 0 ≤ x ≤ 10 }
  requires { forall i. 0 ≤ 4*i+1 < a.length → a[4*i+1] ≥ 0 }
  ensures  { result ≥ 0 }
  = let y = 2*x+1 in a[y*y]
```

1. Le générateur d'obligations produit un ensemble d'énoncés de théorèmes à prouver, appelés *tâches de preuve*. Sur l'exemple, on obtient la formule

```
forall a:array int, x:int.
 length a ≥ 1000 ∧ 0 ≤ x ≤ 10 ∧ (forall i:int.
                    0 ≤ 4 * i + 1 < length a → a[4 * i + 1] ≥ 0) →
 let y = 2 * x + 1 in (0 ≤ y * y < length a) ∧ a[y * y] ≥ 0
```

2. Chaque tâche de preuve peut être soumise à une ou plusieurs *transformations*, qui produisent un ensemble de sous-tâches à prouver. Sur l'exemple, la transformation `split` produit deux sous-tâches qui sont formées par les deux buts

```
  goal VC1 (* index in array bounds *) : 0 ≤ (y * y) < length a
  goal VC2 (* postcondition *) : a[y * y] ≥ 0
```

dans le contexte commun :

```
a : array int
x : int
Requires1 : length a ≥ 1000 ∧ 0 ≤ x ≤ 10
Requires2 : forall i:int. 0 ≤ 4 * i + 1 < length a → a[4 * i + 1] ≥ 0
y : int = 2 * x + 1
```

Les transformations sont typiquement déclenchées de manière interactive [10] et sont implantées par du code OCaml interne à Why3. Ainsi, le but `VC2` ci-dessus n'est prouvé par aucun prouveur SMT, à cause de la difficulté d'instancier l'hypothèse `Requires2` avec la bonne valeur de `i`. Un appel manuel de transformation « `instantiate Requires2 x*x+x` » produit la sous-tâche

```
[...] (* meme contexte que ci-dessus *)
Hinst :
  0 ≤ 4 * (x * x + x) + 1 < length a → a[4 * (x * x + x) + 1] ≥ 0

goal VC2 : a[y * y] ≥ 0
```

3. La tâche résultante ci-dessus est maintenant prouvable par un prouveur SMT. Mais afin de faire appel à un prouveur externe donné sur une sous-tâche donnée, d'autres transformations sont appliquées de manière transparente à l'utilisateur afin d'adapter cette sous-tâche à la logique supportée par ce prouveur. Par exemple le type polymorphe `array 'a` est typiquement encodé dans une logique simplement typée.

4. Les sous-tâches obtenues sont transmises aux prouveurs externes et les résultats sont collectés en faisant confiance à la réponse de ces prouveurs.

Toutes ces étapes sont dans la base de confiance de Why3. Le problème auquel on s'attaque est donc le suivant : comment réduire la taille de cette base de confiance ? Dans cet article nous nous focalisons sur la confiance que l'on peut obtenir dans les transformations des étapes 2 et 3. Notre approche est fondée sur l'utilisation de certificats vérifiables par une tierce partie, selon l'approche classiquement dénommée *sceptique*, à l'inverse d'une approche *autarcique* qui chercherait à prouver le code OCaml des transformations. Notre approche s'attache particulièrement à la *modularité* de la certification : nous ne visons pas d'emblée à certifier toutes les étapes ci-dessus, ce qui serait trop ambitieux, mais à ajouter petit à petit des certificats. Nous devons donc insister sur la modularité et la composabilité de nos certificats.

Dans la partie 2 nous présentons la structure de certificats que nous proposons et les propriétés de correction attendues, ainsi que les propriétés de composabilité. La partie 3 présente notre implantation OCaml d'un vérificateur de certificats. Cette implantation OCaml forme ainsi la base de confiance du nouveau mécanisme de transformations certifiantes de Why3. Afin de réduire encore cette base de confiance, nous proposons dans la partie 4 un deuxième vérificateur de certificats qui se base sur la plateforme Dedukti [4]. Dans la section 5 nous présentons des résultats expérimentaux. Le code source associé à ce travail est distribué avec Why3.

2 Certificats modulaires pour les transformations logiques

Nous présentons ici noter méthode 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 nous abstraire de la représentation Why3 des tâches, nous cherchons d'abord, en section 2.2, à extraire le contenu logique de ces tâches. En section 2.3 nous présentons notre 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 [5] est basé sur celui de la logique classique du premier ordre. Les termes sont typés, les types étant possiblement polymorphes et les variables de type étant implicitement quantifiées de façon prénexe sur chaque déclaration globale. Ce formalisme fournit un certain nombre d'extensions syntaxiques et de théories intégrées (entre autres, l'arithmétique des entiers et des réels). Les extensions les plus notables sont les types algébriques et filtrage par motif, les définitions locales (`let tau = 2.0 * pi in ...`) et les expressions conditionnelles (`if c then ... else ...`), celles-ci étant admises aussi bien dans les formules que dans les termes. Why3 fournit des éléments de logique d'ordre supérieur *via* une théorie prédéfinie qui introduit le type « flèche » et le symbole d'application.

Les *tâches de preuve* dans Why3 sont composées d'une liste de prémisses (axiomes et lemmes, déclarations et définitions de types, de fonctions et de prédicats) 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 (ou ajoutées avec un greffon) 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 avec `let` ou `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. De même, c'est grâce à une autre transformation logique que les

constructions d'ordre supérieur peuvent être éliminées, les λ-termes étant extraits des formules et convertis en définitions de symboles logiques de type « flèche ».

L'implantation OCaml de Why3 n'expose pas à l'auteur d'une transformation un accès libre aux structures de termes et de tâches. Au contraire, Why3 fournit une API défensive basée sur des constructeurs et destructeurs malins, qui permettent par exemple de manipuler les lieurs en protégeant contre les problèmes de capture de noms de variables.

2.2 Extraction du contenu logique des tâches

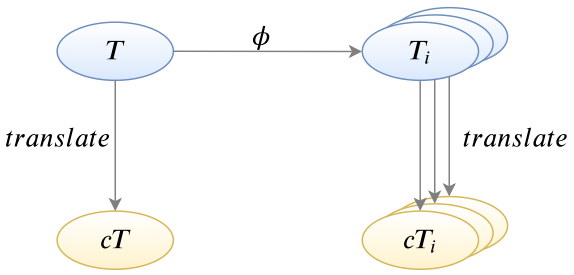
Nous introduisons un nouveau type `ctask` pour représenter les tâches de preuve. Par rapport à la représentation Why3 des tâches, les `ctask` expriment uniquement le contenu logique de celles-ci, sans mentionner les déclarations de types ni les signatures des symboles de fonction. De plus, nous avons choisi de permettre plusieurs buts dans ces `ctask`, comme en calcul des séquents. Les buts et les hypothèses sont alors traités de manière symétrique, nous verrons dans cette section que cela permet de limiter le nombre de certificats différents pour deux raisons. D'une part, un même certificat peut être utilisé pour deux règles duales (voir la FIGURE 2 en section 2.3). D'autre part, les certificats sont alors naturellement plus expressifs. Par exemple la règle d'introduction d'une implication à droite est dérivée à partir celle de l'introduction de la disjonction à droite (voir l'exemple 2.4).

Les `ctask` sont ainsi représentées par des séquents, ce que nous notons de la façon suivante :

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

En l'absence de signature explicite, la validité d'une `ctask` exprime implicitement une validité pour toute interprétation possible des symboles qui y apparaissent librement.

Nous définissons 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.



Notons que la traduction d'une tâche passe aussi par la traduction des formules qui la composent. On a choisit une représentation *locally nameless* afin de simplifier le traitement des transformations qui effectuent des réécritures dans les formules.

On souhaite certifier que la validité de $T_1, \ldots T_n$ entraîne celle de T, on doit donc vérifier que la validité de $cT_1, \ldots, cT_n$ 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 notre 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, nous supposerons 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*.

$$
\begin{array}{rcll}
\mathit{certif} & ::= & \mathsf{Hole} & \mid\ \mathsf{Swap\_neg}(\mathit{ident}, \mathit{certif}) \\
& \mid & \mathsf{Trivial}(\mathit{ident}) & \mid\ \mathsf{Destruct}(\mathit{ident}, \mathit{ident}, \mathit{ident}, \mathit{certif}) \\
& \mid & \mathsf{Axiom}(\mathit{ident}, \mathit{ident}) & \mid\ \mathsf{Weakening}(\mathit{ident}, \mathit{certif}) \\
& \mid & \mathsf{Cut}(\mathit{ident}, \mathit{cterm}, \mathit{certif}, \mathit{certif}) & \mid\ \mathsf{Intro\_quant}(\mathit{ident}, \mathit{ident}, \mathit{certif}) \\
& \mid & \mathsf{Split}(\mathit{ident}, \mathit{certif}, \mathit{certif}) & \mid\ \mathsf{Inst\_quant}(\mathit{ident}, \mathit{ident}, \mathit{cterm}, \mathit{certif}) \\
& \mid & \mathsf{Unfold}(\mathit{ident}, \mathit{certif}) & \mid\ \mathsf{Rewrite}(\mathit{ident}, \mathit{ident}, \mathit{path}, \mathit{bool}, \mathit{certif}^*)
\end{array}
$$

FIGURE 1 – Langage des certificats.

$$
\frac{}{\Gamma \vdash \Delta \overset{\mathtt{Hole}}{\Longleftarrow} \{\Gamma \vdash \Delta\}}
\qquad
\frac{}{\Gamma, H : A \vdash \Delta, G : A \overset{\mathtt{Axiom}(H,G)}{\Longleftarrow} \emptyset}
$$

$$
\frac{\Gamma, P : A \vdash \Delta \overset{c_1}{\Leftarrow} S_1 \qquad \Gamma, P : B \vdash \Delta \overset{c_2}{\Leftarrow} S_2}{\Gamma, P : A \vee B \vdash \Delta \overset{\mathsf{Split}(P,c_1,c_2)}{\Longleftarrow} S_1 \cup S_2}
\qquad
\frac{\Gamma \vdash \Delta, P : A \overset{c_1}{\Leftarrow} S_1 \qquad \Gamma \vdash \Delta, P : B \overset{c_2}{\Leftarrow} S_2}{\Gamma \vdash \Delta, P : A \wedge B \overset{\mathsf{Split}(P,c_1,c_2)}{\Longleftarrow} S_1 \cup S_2}
$$

$$
\frac{\Gamma, H_1 : A, H_2 : B \vdash \Delta \overset{c}{\Leftarrow} S}{\Gamma, P : A \wedge B \vdash \Delta \overset{\mathsf{Destruct}(P,H_1,H_2,c)}{\Longleftarrow} S}
\qquad
\frac{\Gamma \vdash \Delta, H_1 : A, H_2 : B \overset{c}{\Leftarrow} S}{\Gamma \vdash \Delta, P : A \vee B \overset{\mathsf{Destruct}(P,H_1,H_2,c)}{\Longleftarrow} S}
$$

$$
\frac{\Gamma, P : Q\ y \vdash \Delta \overset{c}{\Leftarrow} S \qquad y \text{ fresh}}{\Gamma, P : \exists x.Q\ x \vdash \Delta \overset{\mathsf{Intro\_quant}(P,y,c)}{\Longleftarrow} S}
\qquad
\frac{\Gamma \vdash \Delta, P : Q\ y \overset{c}{\Leftarrow} S \qquad y \text{ fresh}}{\Gamma \vdash \Delta, P : \forall x.Q\ x \overset{\mathsf{Intro\_quant}(P,y,c)}{\Longleftarrow} S}
$$

$$
\frac{\Gamma \vdash \Delta, P : \exists x.Q\ x, G : Q\ t \overset{c}{\Leftarrow} S}{\Gamma \vdash \Delta, P : \exists x.Q\ x \overset{\mathsf{Inst\_quant}(P,G,t,c)}{\Longleftarrow} S}
\qquad
\frac{\Gamma, P : \forall x.Q\ x, H : Q\ t \vdash \Delta \overset{c}{\Leftarrow} S}{\Gamma, P : \forall x.Q\ x \vdash \Delta \overset{\mathsf{Inst\_quant}(P,H,t,c)}{\Longleftarrow} S}
$$

$$
\frac{\Gamma \vdash \Delta, P : \neg A \vee B \overset{c}{\Leftarrow} S}{\Gamma \vdash \Delta, P : A \Rightarrow B \overset{\mathsf{Unfold}(P,c)}{\Longleftarrow} S}
\qquad
\frac{\Gamma, P : A \vdash \Delta \overset{c}{\Leftarrow} S}{\Gamma \vdash \Delta, P : \neg A \overset{\mathsf{Swap\_neg}(P,c)}{\Longleftarrow} S}
$$

FIGURE 2 – Règles de dérivation du jugement $T \overset{c}{\Leftarrow} S$ (extrait).

2.3 Certificats à trous

On suit l'approche sceptique : les transformations sont instrumentées afin de produire un certificat. La particularité de notre 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 est définie par un prédicat ternaire $T \overset{c}{\Leftarrow} S$, 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. La figure 2 donne un extrait de la définition inductive de ce prédicat.

Ces règles sont similaires à celles du calcul des séquents, nous pouvons en particulier énoncer la propriété suivante qui se démontre par récurrence sur l'arbre de dérivation de $T \overset{\mathcal{C}}{\Leftarrow} S$.

Proposition 2.2. *Pour tout certificat c, tâche T et ensemble de tâches S, si $T \overset{\mathcal{C}}{\Leftarrow} S$ alors la validité de l'ensemble des tâches de S entraîne la validité de T.*

Voici 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. Notons que le même certificat, Split, est utilisé pour deux règles duales.

Exemple 2.3. *Considérons la transformation* `split_all` *qui détruit les conjonctions tant que possible et tente d'éliminer à la volée les sous-tâches triviales. En notant $\Gamma := H_1 : A_1,\ H_3 : A_3$, cette transformation appliquée à $\Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$ donne la tâche $\Gamma \vdash G : A_2$ et le certificat certif $:= \mathsf{Split}(G, \mathsf{Axiom}(H_1, G), \mathsf{Split}(G, \mathsf{Hole}, \mathsf{Axiom}(H_3, G)))$ garantit la validité de cette application. En effet, on a la dérivation :*

$$\dfrac{\dfrac{}{\Gamma \vdash G : A_1 \overset{\mathsf{Axiom}(H_1,G)}{\Longleftarrow} \emptyset} \qquad \dfrac{\dfrac{}{\Gamma \vdash G : A_2 \overset{\mathsf{Hole}}{\Longleftarrow} \{\Gamma \vdash G : A_2\}} \qquad \dfrac{}{\Gamma \vdash G : A_3 \overset{\mathsf{Axiom}(H_3,G)}{\Longleftarrow} \emptyset}}{\Gamma \vdash G : A_2 \wedge A_3 \overset{\mathsf{Split}(G,\mathsf{Hole},\mathsf{Axiom}(H_3,G))}{\Longleftarrow} \{\Gamma \vdash G : A_2\}}}{\Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3) \overset{certif}{\Longleftarrow} \{\Gamma \vdash G : A_2\}}$$

Exemple 2.4 (Paradoxe du buveur)**.** *Donnons-nous un type A habité par a et posons dr la formule $\exists x.(P\,x \Rightarrow \forall y.P\,y)$. Il existe un certificat c tel que $(\vdash G_1 : dr) \overset{\mathcal{C}}{\Leftarrow} \emptyset$.*

En effet, posons $\mathsf{Intro}(G, P, c) := \mathsf{Unfold}(G, \mathsf{Destruct}(G, P, G, \mathsf{Swap\_neg}(P, c)))$. Pour deux formules A et B données, la dérivation du schéma de gauche suivant est abrégée par celle du schéma de droite (notation avec double barre) :

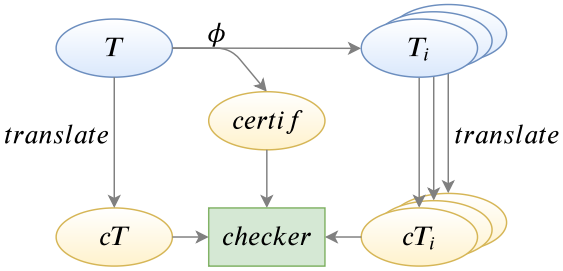
$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, P : A \vdash \Delta, G : B \overset{\mathcal{C}}{\Leftarrow} S}{\Gamma \vdash \Delta, P : \neg A, G : B \overset{\mathsf{Swap\_neg}(P,c)}{\Longleftarrow} S}}{\Gamma \vdash \Delta, G : \neg A \vee B \overset{\mathsf{Destruct}(G,P,G,\ldots)}{\Longleftarrow} S}}{\Gamma \vdash \Delta, G : A \Rightarrow B \overset{\mathsf{Unfold}(G,\ldots)}{\Longleftarrow} S}}{\qquad}$$

$$\dfrac{\Gamma, P : A \vdash \Delta, G : B \overset{\mathcal{C}}{\Leftarrow} S}{\Gamma \vdash \Delta, G : A \Rightarrow B \overset{\mathsf{Intro}(G,P,c)}{\Longleftarrow} S}$$

où les ... abrègent le certificat du jugement situé au-dessus. On a alors la dérivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{H_1 : P\,a, H_2 : P\,y \vdash G_1 : dr, G_2 : P\,y, G_3 : \forall y.P\,y \overset{\mathsf{Axiom}(H_2,G_2)}{\Longleftarrow} \emptyset}}{H_1 : P\,a \vdash G_1 : dr, G_2 : P\,y, G_3 : P\,y \Rightarrow \forall y.P\,y \overset{\mathsf{Intro}(G_3,H_2,\ldots)}{\Longleftarrow} \emptyset}}{H_1 : P\,a \vdash G_1 : dr, G_2 : P\,y \overset{\mathsf{Inst\_quant}(G_1,G_3,y,\ldots)}{\Longleftarrow} \emptyset}}{H_1 : P\,a \vdash G_1 : dr, G_2 : \forall y.P\,y \overset{\mathsf{Intro\_quant}(G_2,y,\ldots)}{\Longleftarrow} \emptyset}}{\vdash G_1 : dr, G_2 : P\,a \Rightarrow \forall y.P\,y \overset{\mathsf{Intro}(G_2,H_1,\ldots)}{\Longleftarrow} \emptyset}}{\vdash G_1 : dr \overset{\mathsf{Inst\_quant}(G_1,G_2,a,\ldots)}{\Longleftarrow} \emptyset}$$

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 $T \overset{c}{\Leftarrow} S$ 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 2.5. *Un tel vérificateur est* correct *si, lorsqu'il répond positivement, alors* $T \overset{c}{\Leftarrow} S$ *est dérivable.*

2.4 Composition des transformations certifiantes

Notre définition de certificats à trous permet 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. Nous supposerons ici que la composition de deux transformations ϕ_1 et ϕ_2 donne la transformation qui applique d'abord ϕ_1 à la tâche initiale et ensuite ϕ_2 à toutes les tâches résultant de l'application de ϕ_1.

Pour obtenir le certificat de l'application de ϕ_1 suivie de ϕ_2, ϕ_1 et ϕ_2 étant des transformations certifiantes, on procède comme suit : (1) on applique ϕ_1 à la tâche initiale, on obtient un certificat c et une liste de tâches lt, où chaque Hole de c correspond exactement à une tâche de lt ; (2) on remplace chacun de ces Hole de c par le certificat renvoyé par l'application de ϕ_2 sur la tâche correspondant à ce Hole. Le certificat c avec ces remplacements est le certificat recherché.

Exemple 2.6 (Suite de l'exemple 2.3)**.** *La transformation* `split_all` *peut être vue comme l'itération d'une transformation* ϕ *qui clôt la tâche initiale si elle est triviale et qui, sinon, décompose au plus une fois une conjonction dans un but. L'application de* ϕ *sur* $\Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$ *donne*

$$[\Gamma \vdash G : A_1; \Gamma \vdash G : A_2 \wedge A_3],\ \mathsf{Split}(G, \mathsf{Hole}, \mathsf{Hole})$$

On peut réappliquer ϕ *sur chacune des deux tâches résultantes :*

$$\phi(\Gamma \vdash G : A_1) = [],\ \mathsf{Axiom}(H_1, G)$$
$$\phi(\Gamma \vdash G : A_2 \wedge A_3) = [\Gamma \vdash G : A_2; \Gamma \vdash G : A_3],\ \mathsf{Split}(G, \mathsf{Hole}, \mathsf{Hole})$$

À partir de $\phi(\Gamma \vdash G : A_2) = [\Gamma \vdash G : A_2],\ \mathsf{Hole}$ *et de* $\phi(\Gamma \vdash G : A_3) = [],\ \mathsf{Axiom}(H_3, G)$*, on retrouve le certificat pour l'itération de* ϕ *en composant les certificats comme indiqué ci-dessus :*

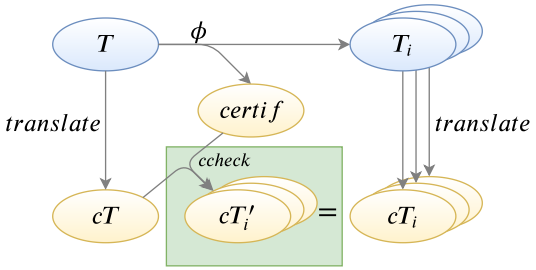
$$\mathsf{Split}(G, \mathsf{Axiom}(H_1, G), \mathsf{Split}(G, \mathsf{Hole}, \mathsf{Axiom}(H_3, G)))$$

3 Vérificateur codé en OCaml

Le premier vérificateur que nous proposons est implanté en OCaml. Dans cette première approche, nous définissons une version exécutable du jugement $T \overset{c}{\Leftarrow} S$ 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.

Notons que dans cette implantation, nous avons délibérément choisi de produire une liste de tâches et de ne pas chercher à autoriser les permutations dans cette liste. Ce vérificateur est donc en ce sens incomplet (il ne « décide » pas tous les $T \overset{certif}{\Longleftarrow} S$) mais a le mérite d'être calculable efficacement. En pratique, cette restriction n'est pas prohibitive car il est toujours possible de modifier le certificat afin de faire correspondre ses « trous » avec la liste de tâches renvoyée.

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, et suivant les règles qui définissent le jugement $T \overset{c}{\Leftarrow} S$. 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 $\mathsf{Split}(P, c_1, c_2)$: 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.

Exemple 3.1 (Suite de l'exemple 2.3)**.** *Pour vérifier que l'application de* `split_all` *à la tâche* $\Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$ *est bien certifiée par* $c := \mathsf{Split}(G, \mathsf{Axiom}(H_1, G), \mathsf{Split}(G, \mathsf{Hole}, \mathsf{Axiom}(H_3, G)))$ *on calcule :*

$$\begin{aligned} &\mathit{ccheck}\ c\ (\Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)) \\ &\quad = \mathit{ccheck}\ (\mathsf{Axiom}(H_1, G))\ (\Gamma \vdash G : A_1)\ @ \\ &\qquad \mathit{ccheck}\ (\mathsf{Split}(G, \mathsf{Hole}, \mathsf{Axiom}(H_3, G)))\ (\Gamma \vdash G : A_2 \wedge A_3) \\ &\quad = [] \ @\ \mathit{ccheck}\ \mathsf{Hole}\ (\Gamma \vdash G : A_2)\ @\ \mathit{ccheck}\ (\mathsf{Axiom}(H_3, G))\ (\Gamma \vdash G : A_3) \\ &\quad = [\Gamma \vdash G : A_2] \end{aligned}$$

qui est bien la tâche résultante attendue.

Proposition 3.2 (Correction de ccheck)**.** *Pour tout certificat c et tâche T, si* ccheck c T *renvoie une liste L, alors il existe une dérivation de $T \overset{\mathcal{L}}{\Leftarrow} S$, 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 règles de certificats. Nous nous sommes donc plutôt concentrés sur une deuxième approche, présentée ci-après.

4 Vérificateur basé sur Dedukti

Notre deuxième approche pour réaliser un vérificateur vise à se passer d'une preuve formalisée d'un théorème de correction comme la proposition 3.2. Cette autre approche utilise le vérificateur de preuve universel Dedukti. L'intérêt est d'utiliser 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 [1].

Plus précisément, on propose 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 nous construisons un terme Dedukti qui est candidat pour une preuve de $\hat{T_1}$`->`$\cdots$`->`$\hat{T_n}$`->`$\hat{T}$, où `->` est la flèche de Dedukti. Ce terme candidat est naturellement destiné à être transmis à Dedukti pour vérification.

Dans un premier temps nous rappelons un encodage existant de la logique du premier ordre en Dedukti, puis nous présentons notre plongement superficiel.

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 Logique du premier ordre en Dedukti

Dedukti [3] est un vérificateur de types pour le $\lambda\Pi$-calcul modulo, un formalisme logique basé sur un $\lambda\Pi$-calcul extensible avec des règles de réécriture. Il possède trois constructions : l'abstraction $\lambda x : A.u$, notée [2] `(x:A) => u` ; le produit dépendant $\Pi x : A.B$, noté `(x:A) -> B` ; et la possibilité de définir des règles de réécriture qui s'ajoutent à la conversion usuelle du $\lambda\Pi$-calcul, notées `[vars] l --> r`, signifiant « `l` se réécrit en `r` », où `[vars]` liste les variables libres apparaissant dans `l` et `r`.

La possibilité d'ajouter des règles de calcul arbitraires permet en toute généralité de construire des contextes incohérents, de la même façon que l'on peut ajouter n'importe quel axiome en Coq. Mais des encodages de la plupart des systèmes de preuve dans ce formalisme ont été étudiés et montrés corrects. Notre vérificateur se base actuellement sur un encodage de

1. À notre connaissance, deux implantations sont disponibles, `dkcheck` et `lambdapi`. Nous utilisons actuellement le vérificateur `dkcheck` pour des raisons techniques (voir https://github.com/Deducteam/lambdapi/issues/243).
2. Dans la suite, le type `A` sera omis lorsque l'on peut le déduire aisément du contexte.

la logique du premier ordre intuitioniste dans ce système, défini dans la bibliothèque de Dedukti et que nous rappelons ci-dessous.

Deux types sont introduits pour représenter les termes et les propositions de la logique du premier ordre, puis le type `Prop` est construit par les connecteurs logiques :

```
Term : Type.
Prop : Type.
true   : Prop.
false  : Prop.
not    : Prop -> Prop.
and    : Prop -> Prop -> Prop.
or     : Prop -> Prop -> Prop.
imp    : Prop -> Prop -> Prop.
forall : (Term -> Prop) -> Prop.
exists : (Term -> Prop) -> Prop.
```

et le type `Term` par la signature de la théorie considérée (par exemple, les entiers). On notera que l'encodage des quantificateurs utilise une syntaxe abstraite d'ordre supérieur.

Le point clé de l'encodage est un plongement `prf` des preuves de la logique du premier ordre dans la logique sous-jacente de Dedukti, qui donne leur sémantique aux constantes à travers des règles de réécriture (et une preuve directe dans le cas de `true`) :

```
prf : Prop -> Type.
tt:   prf true.
[]    prf false       --> (C:Prop) -> prf C
[A]   prf (not A)     --> prf A -> prf false
[A,B] prf (and A B)   --> (C:Prop) -> (prf A -> prf B -> prf C) -> prf C
[A,B] prf (or  A B)   --> (C:Prop) -> (prf A -> prf C)
                                   -> (prf B -> prf C) -> prf C
[A,B] prf (imp A B)   --> prf A -> prf B
[P]   prf (forall P)  --> (x:Term) -> prf (P x)
[P]   prf (exists P)  --> (A:Prop) -> ((x:Term) -> prf (P x) -> prf A)
                                   -> prf A
tnd: (A:Prop) -> prf (or A (not A))
```

Ainsi, la constante `false` est plongée dans un terme de $\lambda\Pi$ encodant la contradiction (`(C:Prop) -> prf C`, indiquant que toute proposition est vraie), et ainsi de suite. Notons également que comme nous travaillons en logique classique, nous ajoutons l'axiome `tnd`

Exemple 4.1. *Si on fixe A de type* `Prop` *et P de type* `Term -> Prop` *alors donner une preuve de $\forall x.\,(A \wedge (\forall y.\,P\,y)) \Rightarrow ((P\,x) \wedge A)$ consiste à donner un terme de type*

```
prf (forall ((x:Term) =>
     imp (and A (forall ((y : Term) => P y)) (and (P x) A))).
```

Par les règles ci-dessus et β-réduction, ce terme est convertible à

```
(x:Term) ->
 ((Q:Prop) -> (prf A -> ((y:Term) -> prf (P y)) -> prf Q) -> prf Q) ->
  ((Q:Prop) -> (prf (P x) -> prf A -> prf Q) -> prf Q)
```

et une preuve en est donc

```
x => hAP => Q => hQ => hAP Q (hA => hP => hQ (hP x) hA).
```

Notre *back-end* Dedukti traduit les tâches comme des types Dedukti qui s'appuient sur l'encodage ci-dessus. Le certificat est lui-même traduit vers un terme Dedukti, et la vérification consiste à s'assurer que ce terme a bien ce type, selon le principe de l'isomorphisme de Curry-Howard.

4.2 Traduction des tâches

Une tâche T est encodée en un type Dedukti $\hat{T}$ à l'aide de nouvelles constructions :

```
empty: Prop
hyp: Prop -> Prop -> Prop
goal: Prop -> Prop -> Prop
[] empty --> false
[H,S] hyp H S --> imp H S
[G,S] goal G S --> imp (not G) S
```

Si $T = H_1 : A_1, ..., H_m : A_m \vdash G_1 : B_1, ..., G_n : B_n$ alors $\hat{T}$ est

$$\texttt{prf}\ (\texttt{hyp}\ \ A_1\ \ (\texttt{hyp}\ \ A_2\ \ (\cdots\ \ (\texttt{hyp}\ \ A_m\ \ (\texttt{goal}\ \ B_1 (\cdots\ \ (\texttt{goal}\ \ B_n\ \ \texttt{empty}) \cdots))) \cdots)))$$

ce qui correspond à la formule $A_1 \Rightarrow ... \Rightarrow A_m \Rightarrow \neg B_1 \Rightarrow ... \Rightarrow \neg B_n \Rightarrow \bot$, notation que nous utiliserons dans la suite pour l'encodage des formules et des tâches Why3 en Dedukti.

Il faut remarquer que nous suivons l'approche usuelle d'un plongement superficiel, en utilisant l'implication de Dedukti. Une approche plus naïve serait de traduire la tâche $H_1 : A_1, ..., H_m : A_m \vdash G_1 : B_1, ..., G_n : B_n$ en $(A_1 \wedge ... \wedge A_m) \Rightarrow (B_1 \vee ... \vee B_n)$ et de procéder de manière similaire pour l'implication entre la conjonction des tâches résultantes et la tâche initiale. Cette approche naïve demande un traitement explicite du contexte qui devient rapidement très lourd et n'est donc en pratique pas utilisable.

Exemple 4.2. *Considérons la tâche $H_2 : B \vee C \vdash$ et une transformation ϕ qui scinde H_2. ϕ produit les sous-tâches $H_2 : B \vdash$ et $H_2 : C \vdash$. Le type Dedukti qui correspondrait à la traduction naïve serait $(B \Rightarrow \bot) \wedge (C \Rightarrow \bot) \Rightarrow B \vee C \Rightarrow \bot$. Supposons maintenant que l'on ait obtenu un terme* `t` *qui a ce type et considérons le nouveau problème où l'on ajoute l'hypothèse A. On applique ϕ à la tâche $H_1 : A, H_2 : B \vee C \vdash$ et on obtient les tâches $H_1 : A, H_2 : B \vdash$ et $H_1 : A, H_2 : C \vdash$. Il n'est pas facile d'obtenir, à partir de* `t`*, un terme de type $(A \wedge B \Rightarrow \bot) \wedge (A \wedge C \Rightarrow \bot) \Rightarrow A \wedge (B \vee C) \Rightarrow \bot$, car il faudrait construire et déconstruire plusieurs fois des conjonctions.*

Avec notre plongement superficiel, si on a construit le terme `t` *de type $(B \Rightarrow \bot) \Rightarrow (C \Rightarrow \bot) \Rightarrow B \vee C \Rightarrow \bot$ et que l'on cherche à obtenir un terme de type $(A \Rightarrow B \Rightarrow \bot) \Rightarrow (A \Rightarrow C \Rightarrow \bot) \Rightarrow A \Rightarrow B \vee C \Rightarrow \bot$, il suffit d'écrire :* `cT1 => cT2 => a => bc => t (cT1 a) (cT2 a) bc`.

4.3 Traduction et élaboration du certificat

4.3.1 Définition de termes de preuve

Le terme de preuve dont on vérifiera qu'il a le type défini ci-dessus est obtenu à partir du certificat. Pour ce faire, dans un premier temps, nous associons un terme Dedukti à chacune des règles d'inférence de $T \overset{\mathcal{L}}{\Leftarrow} S$. Le type de ce terme est obtenu en ignorant le contexte et les tâches résultantes de cette règle et en prenant sa formule logique équivalente.

Illustrons cette démarche sur la règle qui scinde une disjonction dans une prémisse :

$$\frac{\Gamma, P : A \vdash \Delta \overset{c_1}{\Leftarrow} S_1 \qquad \Gamma, P : B \vdash \Delta \overset{c_2}{\Leftarrow} S_2}{\Gamma, P : A \vee B \vdash \Delta \overset{\mathsf{Split}(P, c_1, c_2)}{\Longleftarrow} S_1 \cup S_2}$$

En lui retirant le contexte Γ et Δ, le certificat c et les tâches résultantes S, cette règle devient :

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

On définit donc une fois pour toutes en Dedukti le terme suivant

```
def split_hyp (A : Prop) (B : Prop) :
    prf (hyp A empty) -> prf (hyp B empty) ->
    prf (hyp (or A B) empty)
:= cT1 => cT2 => H => or_elim A B H false cT1 cT2.
```

À titre d'exemple, voici également le terme Dedukti introduit pour le cas d'une conjonction dans une prémisse :

```
def destruct_hyp (A : Prop) (B : Prop) :
    prf (hyp A (hyp B empty)) ->
    prf (hyp (and A B) empty)
:= cT1 => H => cT1 (and_elim_1 A B H) (and_elim_2 A B H).
```

4.3.2 Traduction du certificat

Remarquons que l'on souhaite obtenir un terme de preuve d'un type fourni par la traduction des tâches et qui a donc la forme suivante :

$$cT_1 \Rightarrow ... \Rightarrow cT_n \Rightarrow T$$

où les cT_i sont les tâches résultantes et T est la tâche initiale qui est elle-même de la forme $H_1 \Rightarrow ... \Rightarrow H_m \Rightarrow \bot$.

La principale difficulté pour obtenir ce terme de preuve vient de la gestion des noms. Premièrement, il faut générer et se souvenir de noms pour les tâches résultantes (le terme de preuve commence par `cT1 => ... => cTn =>`). La suite du terme de preuve est obtenue en introduisant les prémisses de la tâche initiale (`H1 => ... => Hm =>`). Enfin, le cœur du terme de preuve provient de la traduction du certificat, ce dernier indiquant les noms des hypothèses intermédiaires. La traduction d'un constructeur de certificat utilise le terme de preuve correspondant (ceux-ci étant définis en 4.3.1), à l'exception du constructeur Hole qui est traduit par la tâche correspondante appliquée aux identificateurs de ses prémisses.

Exemple 4.3. *Supposons qu'une application d'une certaine transformation sur la tâche* $H : A \vee (B \wedge C) \vdash$ *donne les tâches* $H : A \vdash$ *et* $H_b : B, H_c : C \vdash$ *et le certificat* Split (H, Hole, Destruct (H, H_b, H_c, Hole)). *On introduit les tâches résultantes (*`cT1 => cT2 => ...`*), la prémisse de la tâche initiale (*`H => ...`*) puis on suit le certificat qui fait d'abord un* Split *sur la prémisse* H *(*`split_hyp (H => c1) (H => c2) H`*) où* `c1` *suit le sous certificat gauche de* Split. *Celui-ci est le premier* Hole *rencontré et est donc remplacé par la première tâche appliquée aux ident de ses prémisses, c'est-à-dire par* `cT1 H`. *Pour* `c2`*, on suit le certificat* Destruct*(*H, H_b, H_c, *Hole), ce qui nous donne* `destruct_hyp (Hb => Hc => c3) H`. *Cette fois-ci* `c3` *suit le deuxième* Hole *rencontré et est donc remplacé par la deuxième tâche appliquée aux ident de ses prémisses, soit* `cT2 Hb Hc`. *En résumé, on vérifie avec Dedukti que le terme*

```
cT1 => cT2 => H =>
 split_hyp (H => cT1 H)
           (H => destruct_hyp (Hb => Hc => cT2 Hb Hc) H) H
```

a bien le type $(A \Rightarrow \bot) \Rightarrow (B \Rightarrow C \Rightarrow \bot) \Rightarrow A \vee (B \wedge C) \Rightarrow \bot$.

4.3.3 Élaboration du certificat

Le type de nos certificats est relativement peu verbeux. En particulier, les constructeurs 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 que l'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. Les types des termes de preuve correspondant aux règles sont généralisés en les variables libres qui y apparaissent. Du point de vue de la génération du terme de preuve, il s'agit de leur donner des paramètres qui explicitent ces formules.

Exemple 4.4. *Dans l'exemple précédent,* Split *a été traduit en* `split_hyp` *car la formule manipulée* `H` *est une prémisse et non un but.* `split_hyp` *prend* A *et* $B \wedge C$ *comme premiers arguments car* `H` *est la disjonction de* A *et de* $B \wedge C$*. De même, les deux premiers aguments de* `destruct_hyp` *sont* B *et* C *car son étape de certificat* Destruct *correspondante manipule l'hypothèse* $B \wedge C$.

5 Expérimentations

Des transformations de Why3, de l'ordre d'une quinzaine, ont été instrumentées pour générer des certificats et les vérifier à la volée lors de chaque utilisation. De nouvelles transformations ont été implantées, 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. Notre méthode de composition de transformations certifiantes décrite dans la section 2.4 nous 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, 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 mais ne supporte pas encore les quantificateurs universels en tête de la formule à réécrire. 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.

Pour évaluer l'efficacité des implantations de nos vérificateurs, nous avons testé la transformation `blast` sur une famille classique de problèmes pour évaluer les solveurs propositionnels : le principe des tiroirs de Dirichlet. Pour n donné, on ne peut pas mettre n chaussettes dans $n-1$ tiroirs sans qu'un tiroir contienne au moins une paire de chaussettes. La table suivante indique, pour un n donné : le nombre de variables booléennes, et le nombre total d'occurrences des variables dans le problème ; le temps que prend la transformation pour clore le problème et construire le certificat ; la taille de ce certificat (s'il est écrit dans un fichier) ; le temps de la vérification de ce certificat par le vérificateur OCaml ; le temps pris par l'élaboration du fichier Dedukti ; la taille de ce fichier ; et enfin le temps de vérification par Dedukti (appel de `dkcheck`).

nombre de chaussettes	3	4	5
nombre de variables	6	12	20
nombre d'occurrences de variables	18	48	100
temps d'exécution de la transformation (sec)	7×10^{-3}	12	-
taille du certificat (octet)	37×10^{3}	26×10^{6}	-
temps vérification OCaml (sec)	63×10^{-6}	55×10^{-3}	-
taille certificat Dedukti (octet)	89×10^{3}	46×10^{6}	-
temps élaboration Dedukti (sec)	8×10^{-3}	1,9	-
temps vérification Dedukti (sec)	24×10^{-3}	20	-

Le test sur cinq chaussettes ne se termine pas en un temps raisonnable, ce qui n'est pas surprenant car cette famille de problèmes nécessite des preuves sans coupures de taille exponentielle. Pour faire mieux, il faudrait que notre transformation `blast` recherche des preuves avec coupures. On remarque que le vérificateur OCaml est bien plus rapide que Dedukti. En effet, le vérificateur en OCaml utilise une approche réflexive, c'est-à-dire que la vérification est effectuée par un calcul dont les entrées sont les tâches et le certificat. Cela a l'avantage d'être très efficace dans le cas où le moteur de calcul est performant. *A contrario*, le vérificateur en Dedukti repose sur un plongement superficiel qui réduit la vérification à une question de typage dans un cadre où il faut appliquer des réécritures sur les types. On note aussi que la taille du fichier Dedukti est comparable à la taille du certificat. S'il est vrai que l'encodage des termes en Dedukti est linéaire, l'élaboration aurait pu accroître la taille des certificats de façon non-linéaire, ce que l'on n'observe pas ici, probablement parce la plupart des formules sur lesquelles on travaille sont petites.

6 Conclusions, travaux connexes et perspectives

Nous avons présenté 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. Notre 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 notre connaissance. Elle nous 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.

Travaux connexes. De nombreux outils pour la vérification déductive de programmes ont été développés, basés sur des assistants de preuve ou indépendants.

Dans le premier cas, la preuve de programmes nécessite la définition préalable de bibliothèques pour définir un langage de programmation particulier et une logique de programme

correspondante. Il s'agit par exemple de la bibliothèque Iris [14] au-dessus de Coq qui permet de raisonner sur des programmes impératifs avec une logique de séparation ; ou encore de la bibliothèque AutoCorres [12, 11] au-dessus d'Isabelle permettant de raisonner sur des programmes C. Dans un tel contexte, la preuve d'une propriété d'un programme se fait dans l'environnement de preuve assistée sous-jacent. La correction repose sur une sémantique formelle du langage et des preuves des règles de déduction établies une fois pour toutes, ce qui représente un effort de preuve très coûteux et est donc difficile à faire évoluer.

Dans le deuxième cas, les outils s'appuient sur un langage de programmation particulier équipé d'un langage de spécification associé, implantent un générateur d'obligations de preuve, et utilisent des prouveurs automatiques externes tels que les prouveurs SMT. Des exemples de tels environnements sont donnés par Why3, Dafny, Viper, et également des outils pour des langages *mainstream* comme Frama-C pour le C et SPARK pour Ada. Bien que reposant sur des bases théoriques solides, l'implantation des outils, elle-même, et certains aspects pratiques comme l'utilisation de prouveurs automatiques, n'ont pas été vérifiés mécaniquement et peuvent être sources de bugs. Une exception est l'outil F* [15], dont l'encodage de la logique vers SMT a été en partie établi en Coq [1], de manière globale.

L'originalité de notre approche est premièrement de procéder de manière modulaire, en considérant indépendemment des petites transformations, et deuxièmement d'utiliser l'approche sceptique. Cela offre une plus grande maniabilité, en permettant à la fois l'ajout de nouvelles transformations et l'évolution des transformations existantes à faible coût. Pour cela, nous avons proposé un cadre permettant de mêler transformations certifiantes et non certifiantes en ayant confiance dans les transformations certifiantes, grâce à la vérification de leur certificat combiné au contrôle des sous-buts générés, ce qui est à notre connaissance original.

Concernant la vérification des certificats elle-même, nous nous sommes basés sur deux approches classiques : l'approche réflexive, en OCaml, et un plongement superficiel dans un outil de preuve, en Dedukti. L'approche réflexive est connue pour être très efficace, et peut être utilisée de manière non certifiée ou certifiée dans des assistants de preuve intégrant une notion de calcul [13, 2]. Le plongement dans un outil de preuve est connu pour être moins efficace mais sa correction repose sur celle de l'outil de preuve et est donc beaucoup moins coûteuse [9, 8].

Perspectives. Un objectif à court terme est de poursuivre l'application à Why3, afin de rendre certifiantes la plupart de ses transformations les plus couramment utilisées et ainsi d'augmenter la confiance dans cet outil. 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. Nous envisageons notamment de nous placer dans une logique d'ordre supérieur.

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 [2, 7], et en amont, proposer une méthode de certification de la génération d'obligations de preuve. Ce dernier point nécessite de formaliser la sémantique du langage WhyML.

Un passage à l'échelle va nécessairement poser des questions d'efficacité. D'un point de vue programmation, nous souhaitons nous 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. Nous souhaitons étudier la compression de ces certificats, et notamment comment elle pourrait être menée à la volée lors de la combinaison de certificats.

Enfin, notre méthode n'est pas propre à Why3 et peut s'appliquer à la certification d'encodages logiques en toute généralité. Nous souhaitons 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.

Remerciements. Nous remercions les évaluateurs, ainsi que la présidente du comité de programme Zaynah Dargaye, pour leurs nombreux commentaires et suggestions durant la phase d'accompagnement pastoral.

Références

[1] A. Aguirre. Towards a provably correct encoding from F* to SMT. Master's thesis, Université Paris 7, 2016.

[2] 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.

[3] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Dedukti : a logical framework based on the λΠ-calculus modulo theory, 2016. Available at http://www.lsv.fr/~dowek/Publi/expressing.pdf.

[4] 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 λΠ-calculus modulo theory and in the Dedukti system. In *22nd International Conference on Types for Proofs and Programs*, 2016.

[5] 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.

[6] 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.

[7] 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.

[8] R. Cauderlier and P. Halmagrand. Checking Zenon modulo proofs in Dedukti. In *Proof eXchange for Theorem Proving*, volume 186 of *EPTCS*, pages 57–73, 2015.

[9] E. Contejean. Coccinelle, a Coq library for rewriting. In *Types*, 2008.

[10] S. Dailler, C. Marché, and Y. Moy. Lightweight interactive proving inside an automatic program verifier. In *Formal Integrated Development Environments*, 2018.

[11] D. Greenaway. *Automated proof-producing abstraction of C code*. PhD thesis, CSE, UNSW, 2015.

[12] D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't sweat the small stuff : Formal verification of C code without the pain. In *Programming Language Design and Implementation*, pages 429–439. ACM, 2014.

[13] B. Grégoire, L. Théry, and B. Werner. A computational approach to Pocklington certificates in type theory. In *Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 97–113. Springer, 2006.

[14] R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal. The essence of higher-order concurrent separation logic. In *Programming Languages and Systems*, volume 10201 of *Lecture Notes in Computer Science*, pages 696–723. Springer, 2017.

[15] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Z. Béguelin. Dependent types and multi-monadic effects in F*. In *Principles of Programming Languages*, pages 256–270. ACM, 2016.