Un mécanisme de preuve par réflexion pour Why3 et son application aux algorithmes de GMP

Raphaël Rieu-Helft[1,2]

[1] TrustInSoft
[2] Inria

Résumé

Nous présentons un mécanisme de preuves par réflexion pour la plateforme de vérification Why3. L'utilisateur peut facilement écrire des procédures de décision dédiées, formellement vérifiées et qui utilisent pleinement les fonctionnalités impératives du langage WhyML. L'impact sur la base de confiance est minimal.

Nous avons évalué cette approche sur des preuves d'algorithmes de GMP. Elle permet de supprimer un très grand nombre d'indications de coupure qui devaient auparavant être fournies manuellement à Why3.

1 Introduction

La plateforme de vérification Why3 [1] fournit un langage programmation proche de ML, WhyML, qui permet d'écrire des programmes et de spécifier leur comportement fonctionnel à l'aide de pré- et postconditions et d'invariants de boucle [7]. L'outil transforme les programmes et spécifications en théorèmes logiques qui peuvent être envoyés à des prouveurs externes, tant automatiques (solveurs SMT ou TPTP) qu'interactifs (Coq, Isabelle/HOL, PVS). Une fois ces théorèmes prouvés, en supposant la correction de Why3 et des prouveurs externes, on sait que les programmes satisfont leur spécification.

Dans de précédents travaux, nous avons utilisé Why3 pour implémenter des algorithmes issus de la bibliothèque d'arithmétique en précision arbitraire GMP [2], prouver leur correction, et générer une bibliothèque performante écrite en C [13]. Les preuves ont été faites en utilisant exclusivement des prouveurs automatiques. Cependant, non seulement certains algorithmes sont très complexes (par exemple la division [12]), mais nous avons aussi dû fournir nettement plus d'indications de coupure que prévu, y compris pour prouver des théorèmes apparemment triviaux. En effet, la taille des contextes de preuve et la présence d'arithmétique non-linéaire est problématique pour les solveurs. Pour certains théorèmes, nous avons dû écrire plusieurs assertions de plus de 100 lignes, ce qui limite grandement l'intérêt d'utiliser des outils automatiques plutôt qu'un assistant de preuve interactif.

Une façon d'ajouter à un prouveur de nouvelles fonctionnalités, comme par exemple une règle d'inférence adaptée à un certain problème, est d'"incorporer un principe de réflexion, afin que l'utilisateur puisse vérifier à l'intérieur du système de preuve que le code qui implémente une nouvelle règle est correct, et d'ajouter ce code au système" [9]. Nous avons ajouté à Why3 un mécanisme qui permet à l'utilisateur d'écrire des procédures de décision comme des programmes Why3 classiques et de les utiliser pour prouver des théorèmes par le calcul. Cet article montre comment nous avons utilisé notre approche pour implémenter et vérifier une procédure de décision adaptée à la vérification d'algorithmes de GMP.

Une transformation Why3 présentée en section 2.1 génère automatiquement les termes à passer en argument à la procédure de décision à partir de sa spécification et du but courant.

1. http://why3.lri.fr
2. http://gmplib.org

Traditionnellement, les preuves par réflexion se font en évaluant des termes purs présents dans des propositions logiques. Cependant, le langage de programmation fourni par Why3 est beaucoup plus riche : variables mutables, tableaux, exceptions, boucles... Notre mécanisme permet à l'auteur de procédures de décision de bénéficier de toute l'expressivité de WhyML. Nous avons dû ajouter à Why3 un interpréteur de code WhyML (section 2.2). Ceci étend la base de confiance de Why3, mais de manière minimale. Nous discutons la correction de notre approche en section 2.3.

Étant capables d'écrire des procédures de décision en WhyML, de les vérifier avec Why3 et de les exécuter sur des propositions logiques réifiées, nous avons tous les outils pour vérifier des algorithmes de GMP à l'aide d'une procédure de décision dédiée, que nous présentons en section 3. Il s'agit d'une procédure de résolution de systèmes d'équations linéaires, mais les coefficients qu'elle manipule sont des produits de rationnels par des exposants symboliques, comme par exemple $-5/3 \cdot \beta^{i+j-2}$. En effet, les algorithmes que nous vérifions manipulent des sommes de monômes de forme $\sum a_i \beta^i$.

Ce travail fait partie de Why3 et les exemples présentés sont disponibles à l'adresse `http://toccata.lri.fr/gallery/multiprecision.en.html`.

Cet article reprend du travail déjà publié à la conférence IJCAR 2018 [11].

2 Preuves par réflexion

Le principe général de la preuve par réflexion est le suivant. Notons P la proposition que l'on cherche à prouver. D'abord, on trouve un plongement de P dans le langage logique du système formel. Notons $\ulcorner P \urcorner$ le terme résultant, par exemple l'arbre de syntaxe abstraite de P. Ensuite, on prouve que si $\ulcorner P \urcorner$ satisfait une certaine propriété φ, alors P est vraie. Ainsi, lorsque l'on veut prouver une proposition P, il suffit de prouver $\varphi(\ulcorner P \urcorner)$. Si φ est telle que $\varphi(\ulcorner P \urcorner)$ peut être validée par le calcul, il s'agit d'une procédure de preuve par réflexion. Cette approche a été utilisée dans de nombreux contextes [9, 8, 1, 3, 4, 6].

2.1 Réification

Une difficulté est que la fonction $\ulcorner \urcorner$ n'est pas exprimable dans le langage logique, et que le terme $\ulcorner P \urcorner$ peut être d'une telle taille qu'on ne peut pas s'attendre à ce qu'un utilisateur le fournisse manuellement. Pour qu'un mécanisme de preuve par réflexion soit utile, il est important de fournir un moyen d'automatiser l'obtention de $\ulcorner P \urcorner$ à partir de P. On appelle ce processus *réification*. Une approche classique est d'exprimer $\ulcorner \urcorner$ dans le méta-langage du système formel utilisé (comme Ltac pour Coq), mais ceci demanderait à l'utilisateur d'apprendre le fonctionnement interne du système.

Notons cependant que si $\ulcorner \urcorner$ n'est pas exprimable dans le langage logique, son inverse l'est. De plus, l'utilisateur doit nécessairement définir quelque chose s'en approchant afin de donner une spécification utile à φ. Dans l'exemple en figure 1, il s'agit de la fonction `interp`. Nous utilisons cet inverse pour générer $\ulcorner P \urcorner$ automatiquement. La réification utilise une approche similaire à la tactique Coq `quote`, avec quelques améliorations : la fonction d'interprétation peut être plus complexe, elle supporte les quantificateurs, le contexte logique peut être réifié, et l'utilisateur n'a pas besoin de spécifier quels termes peuvent être des constantes.

Étant donné un but logique P et une procédure de décision φ, l'utilisateur peut essayer de prouver P par réflexion en tapant la commande "`reflection_f` φ" dans l'IDE. Why3 utilise alors la spécification de φ pour inverser syntaxiquement `interp` et fournir un terme $\ulcorner P \urcorner$ approprié, puis évalue $\varphi(\ulcorner P \urcorner)$. Le contrat de φ est ensuite utilisé comme indication de coupure pour

```
type t = Var int | And t t | ...
type vars = int → bool

function interp (x:t) (y:vars) : bool =
match x with
  | Var n → y n
  | And x1 x2 → interp x1 y && interp x2 y
  ...
end

let φ (x:t) : bool
  ensures { result → forall y. interp x y }
```

FIGURE 1 – Exemple de spécification pour φ

peut-être prouver P. Dans l'exemple de la figure 1, si $\varphi(\ulcorner P \urcorner) = \texttt{True}$, on prouve facilement P car la formule $\ulcorner P \urcorner$ a été choisie de sorte que la postcondition de $\varphi(\ulcorner P \urcorner)$ implique P. Si $\varphi(\ulcorner P \urcorner) = \texttt{False}$, la tentative de preuve par réflexion échoue.

2.2 Interprétation

Traditionnellement, les procédures de décision sont des fonctions écrites dans la logique du système de preuve de manière à pouvoir être interprétées par des prouveurs automatiques ou par le moteur de réécriture du système. Ceci limite leur pouvoir d'expressivité. Par exemple, les fonctions logiques de Why3 ne peuvent pas avoir d'effet de bord et leur terminaison doit être garantie par la décroissance structurelle d'un paramètre.

À l'inverse, notre approche permet d'écrire des procédures de décision comme de simples programmes WhyML. Elles peuvent donc utiliser toutes les fonctionnalités impératives du langage, telles que les références, les tableaux, les boucles ou les exceptions. Leur correction peut être prouvée à l'aide de Why3 et de prouveurs automatiques, et leur contrat est instancié par réification et utilisé comme indication de coupure. Cependant, elles n'ont pas d'implémentation dans la logique de Why3 et ne peuvent donc pas être interprétées par le moteur de réécriture de Why3 ou par des prouveurs automatiques.

Nous avons donc ajouté à Why3 un interpréteur capable d'évaluer n'importe quelle procédure de décision. Il opère sur un langage intermédiaire proche de ML, qui correspond aux programmes WhyML dont les assertions logiques et le code *ghost* ont été effacés, en supposant que le programme a été prouvé au préalable et que ses préconditions sont satisfaites. Ce code intermédiaire est produit par le mécanisme d'extraction existant, qui produit du code OCaml et C à partir de programmes WhyML prouvés.

Peu de travaux antérieurs sur la preuve par réflexion utilisent des procédures de décision avec effets de bord. On peut citer Claret *et al* [4]. Ils utilisent un encodage monadique des calculs avec effet dans Coq (par exemple, la non-terminaison). Les procédures de décision monadiques sont transformées en des programmes impurs exécutés en-dehors de Coq. Le résultat de ces calculs externe est utilisé comme une "prophétie" pour simuler l'exécution de la procédure de décision à l'intérieur de Coq. Puisque nous travaillons avec Why3, qui supporte nativement les calculs impurs, nous n'avons pas besoin d'un tel mécanisme lourd de simulation.

2.3 Correction

L'implémentation de notre mécanisme de preuves par réflexion consiste en deux additions à Why3 : une transformation de réification et un interpréteur de programmes WhyML.

Le code de la réification est complexe et relativement long, car il comporte beaucoup d'heuristiques qui permettent de donner à l'utilisateur un maximum de liberté dans la syntaxe des fonctions d'interprétation. Heureusement, la procédure de réification ne fait pas partie de la base de confiance de Why3. En effet, la réification se contente de deviner des termes appropriés à interpréter et demande à Why3 d'instancier le contrat de la procédure de décision avec. En supposant que l'utilisateur a prouvé la correction de la procédure de décision, la proposition instanciée est vraie, que l'algorithme de réification soit correct ou non. Un échec de la réification empêcherait une instantiation bien typée de la postcondition, ou alors l'indication de coupure résultante serait insuffisante pour prouver le but, mais ne permettrait en aucun cas de prouver quelque chose de faux.

L'interpréteur, en revanche, fait bien partie de la base de confiance. Heureusement, il est très simple, car il ne manipule que des valeurs concrètes. Il n'y a pas besoin d'évaluation partielle, d'exécution symbolique ou d'égalité polymorphe, ce qui le rend bien plus simple que le moteur de réécriture de termes logiques existant. Une autre raison pour cette simplicité est que le langage intermédiaire interprété a relativement peu de constructions. En effet, les transformations de programme réalisées par le mécanisme d'extraction éliminent certains comportements complexes du langage de surface, comme par exemple l'assignation parallèle.

3 Application : GMP

Dans cette section, nous présentons brièvement notre bibliothèque vérifiée d'arithmétique en précision arbitraire [13], et nous montrons comment nous avons éliminé un grand nombre d'assertions à l'aide d'une procédure de décision dédiée.

3.1 Une fonction de GMP

Dans GMP, les entiers naturels sont représentant par des tableaux little-endian d'entiers machine non signés, appelés *limbs*. On pose une base β, typiquement 2^{64}. Tout entier naturel N a une décomposition unique en base β : $N = \sum_{k=0}^{n-1} a_k \beta^k$, représentée par le tableau $a_0 a_1 \dots a_{n-1}$.

Dans les fonctions de bas niveau, il y a très peu de gestion de la mémoire ; les opérandes sont représentées par un pointeur sur leur limb le moins significatif et une taille de type `int32`. Étant donné un tel pointeur `a` et une taille `n`, en supposant que le pointeur est valide sur une longueur `n`, on note

$$\texttt{value a n} = \overline{a_0 \dots a_{n-1}} = \sum_{k=0}^{n-1} a_k \beta^k.$$

La figure 2 montre la fonction qui additionne deux entiers naturels de même longueur. Pour des raisons de lisibilité, la plupart des invariants et une partie de la spécification ont été omis. Il s'agit de l'algorithme d'addition enseigné à l'école : on additionne les opérandes limb par limb, en commençant par le moins significatif, et en maintenant une retenue.

Malheureusement, même cet algorithme simple pose problème pour les prouveurs automatiques. Pour prouver l'invariant de boucle, nous avons eu besoin de l'assertion à la ligne 18. Sa preuve consiste en une suite d'une dizaine d'étapes relativement simples (réécriture d'une

```
let wmpn_add_n (r x y: ptr limb) (sz: int32) : limb
  ensures { 0 ≤ result ≤ 1 }
  ensures { value r sz + (power radix sz) * result =
            value x sz + value y sz }
=
  let i = ref 0 in
  let lx = ref 0 in
  let ly = ref 0 in
  let c = ref 0 in
  while !i < sz do
    invariant { value r !i + (power radix !i) * !c = value x !i + value y !i }
    lx := get_ofs x !i;
    ly := get_ofs y !i;
    let res, carry = add_with_carry !lx !ly !c in
    set_ofs r !i res;
    c := carry;
    assert { value r (!i+1) + (power radix (!i+1)) * !c
             = value x (!i+1) + value y (!i+1)
             by value r (!i+1) + (power radix (!i+1)) * !c
               = value r !i + (power radix !i) * res
                 + (power radix !i) * c
               = ... (* 10+ sous-buts *) };
    i := !i + 1;
  done;
  !c
```

FIGURE 2 – Addition de deux entiers

égalité dans le contexte, application de la distributivité...), mais la taille de l'espace de recherche empêche les prouveurs d'aboutir en un temps raisonnable. Nous avons donc dû fournir de nombreuses indications de coupure à la main avec la construction `by` [5].

Cependant, avec un choix judicieux de coefficients, ce but, ainsi que de nombreux autres buts difficiles dans les preuves de notre bibliothèque, peut être vu comme une combinaison linéaire d'égalités présentes dans le contexte logique (c'est l'exemple en section 3.3). On peut donc espérer le prouver avec une procédure de décision relativement simple.

3.2 Procédure de décision

Nous avons implémenté une procédure de décision pour les systèmes d'équations linéaires dans un corps arbitraire (extraits du code en figure 3). Étant donné des égalités linéaires supposées valides dans le contexte logique, cette procédure tente de prouver une égalité linéaire en montrant que c'est une combinaison linéaire du contexte.

Pour ce faire, on représente le contexte et le but dans une matrice et on effectue un pivot de Gauss (fonction `gauss_jordan`). En cas de succès, on obtient un vecteur de coefficients et on vérifie que la combinaison linéaire correspondante du contexte est égale au but (fonction `check_combination`). Si ce n'est pas le cas, la fonction renvoie `False` et on ne peut rien prouver d'intéressant, puisque la postcondition a `result = True` comme prémisse.

Comme pour les tactiques Coq `lia` et `lra` [1], il s'agit d'une preuve par certificat, le certificat étant le vecteur de coefficients renvoyé par `gauss_jordan`. Il n'y a pas besoin de prouver l'algorithme du pivot de Gauss, ni de définir une sémantique pour la matrice qu'on lui passe comme argument. En fait, on ne prouve rien sur le contenu d'aucune matrice dans le programme. La preuve de la correction de la procédure de décision est donc très simple par rapport à sa

```
clone LinearEquationsCoeffs as C with type t = coeff

type expr = Term coeff int | Add expr expr | Cst coeff
type equality = (expr, expr)

let linear_decision (l: list equality) (g: equality) : bool
  requires { valid_ctx l }
  requires { valid_eq g }
  ensures { forall y z. result = True → interp_ctx l g y z }
  raises  { C.Unknown → true }
= ...
  fill_ctx l 0;
  let (ex, d) = norm_eq g in
  fill_goal ex;
  let ab = m_append a b in
  let cd = v_append v d in
  match gauss_jordan (m_append (transpose ab) cd) with
    | Some r → check_combination l g (to_list r 0 ll)
    | None → False
  end
```

FIGURE 3 – Procédure de décision pour les systèmes d'équations linéaires

longueur et à sa complexité conceptuelle.

Considérons maintenant le contrat de la procédure de décision. La postcondition dit que si la procédure renvoie `True`, le but est vrai pour toutes les valuations `y` et `z` des variables telles que les égalités du contexte sont vraies. Les préconditions `valid_ctx` et `valid_eq` demandent que les entiers utilisés comme indices de variables dans le contexte et le but soient positifs, ce qui est nécessaire pour prouver la sûreté des accès dans les tableaux. La nature de la procédure de réification assure que ces préconditions seront toujours vérifiées en pratiques, mais comme on ne fait pas confiance à la réification, l'utilisateur doit prouver les préconditions explicitement. En pratique, ça ne pose aucun problème aux solveurs SMT. Enfin, la clause `raises` exprime que la procédure peut lever une exception non rattrapée (typiquement une erreur arithmétique, puisqu'on autorise les opérations du corps à être partielles). Dans ce cas, rien n'est prouvé.

On peut remarquer que la procédure de décision est indépendante de Why3, au sens où elle ne contient pas de méta-instructions pour la réification ou quoi que ce soit de lié au fonctionnement interne de Why3. On pourrait imaginer trouver du code similaire dans un prouveur automatique.

Enfin, le type `coeff` est laissé abstrait dans ce module. L'utilisateur peut l'instancier avec n'importe quel type qui vérifie certaines propriétés (données sous forme d'axiomes omis de la figure). Essentiellement, `coeff` doit implémenter les opérations élémentaires d'un corps, mais certaines de ces opérations peuvent être partielles et lever l'exception `Unknown`. Considérons maintenant comment choisir les coefficients dans le cas de GMP.

3.3 Coefficients

Voici une version simplifiée du contexte et du but donnés par Why3 au niveau de la longue assertion dans la boucle de `wmpn_add_n` (figure 2 ligne 18). Les variables `r1` et `c1` dénotent les valeurs de `r` et `c` au début du corps de la boucle (avant les modifications des lignes 13-14).

On remarque que la combinaison linéaire $\texttt{H5} - \texttt{H4} - \texttt{H3} + \texttt{H2} + \beta^{\texttt{i}} \cdot \texttt{H1} + \texttt{H}$ se simplifie en une égalité équivalente à `g`. Pour prouver cela, la procédure de décision a besoin d'inclure dans les coefficients des puissances de β (`radix` dans le code WhyML), et de permettre les exposants

```
axiom H:  value r1 i + (power radix i) * c1 = value x i + value y i
axiom H1: res + radix * c = lx + ly + c1
axiom H2: value r i = value r1 i
axiom H3: value x (i+1) = value x i + (power radix i) * lx
axiom H4: value y (i+1) = value y i + (power radix i) * ly
axiom H5: value r (i+1) = value r i + (power radix i) * res
goal g:   value r (i+1) + power radix (i+1) * c = value x (i+1) + value y (i+1)
```

symboliques (car `i` est une variable).

Plus précisément, les coefficients sont le produit d'un rationnel et d'une puissance (symbolique) de β. On peut définir sans problème une multiplication et un inverse multiplicatif sur ces coefficients. L'addition est partielle, car on ne peut additionner que des coefficients dont les exposants sont égaux. Si ce n'est pas le cas, l'addition lève une exception, ce qui est permis par la procédure de décision. Notons que les exposants n'ont pas besoin d'être structurellement égaux mais seulement égaux pour toutes valuations des variables présentes dans les exposants, ce qui peut être prouvé automatiquement par une procédure de décision secondaire très simple appelée par la procédure primaire.

4 Conclusion

Cet article présente deux contributions. Premièrement, nous avons développé un mécanisme pour les preuves par réflexion qui utilise des programmes WhyML impératifs comme procédures de décision. Deuxièmement, nous avons implémenté et vérifié une procédure de décision pour les systèmes d'équations linéaires avec coefficients symboliques. Nous avons utilisé cette procédure pour prouver de nombreux buts dans notre formalisation d'algorithmes de GMP. Instanciée avec un autre ensemble de coefficients, elle pourrait être réutilisée dans d'autres contextes.

Nous avons revisité toutes nos preuves existantes d'algorithmes d'addition, de soustraction et de multiplication, qui demandaient précédemment de nombreuses indications de coupure manuelles. La procédure de décision a été capable de décharger toutes les longues assertions (dans la même veine que celle de la figure 2). Cette section de notre bibliothèque faisait auparavant 1660 lignes. Les 660 lignes de code de programmes n'ont évidemment pas changé, mais les 1000 lignes de spécification et preuve ont été diminuées de moitié. Qui plus est, une grande partie des 500 lignes restantes est constituée de contrats de fonctions et d'invariants de boucle, essentiellement incompressibles.

Le but le plus difficile auquel nous avons pu appliquer notre procédure de décision (une assertion dans la preuve du cas général de la division longue) impliquait un pivot de Gauss sur une matrice de taille 150×90 environ, et le temps d'exécution était de 3 secondes environ, ce qui est acceptable du point de vue de l'expérience utilisateur. Si le temps d'exécution devient problématique pour de plus grandes matrices, une option pour améliorer les performances serait d'extraire la procédure de décision vers OCaml et d'exécuter le binaire résultant plutôt que notre interpréteur.

Bien que notre procédure de décision ne traite que des systèmes d'équations linéaires, nous l'avons utilisé avec succès dans les preuves d'algorithmes de multiplication, division et décalages logiques pour prouver des buts qui semblent complètement non-linéaires de prime abord. Dans ces cas, nous avons dû fournir une ou deux indications de coupure pour gérer les parties non-linéaires du raisonnement, mais c'est très acceptable puisque la plupart de ces buts demandaient plus de cinquante assertions auparavant. On peut penser que ce nouvel outil nous permettra de vérifier de nouveaux algorithmes de GMP bien plus efficacement.

L'approche présentée dans cet article n'est pas limitée à Why3 en principe. Pour développer un mécanisme similaire, il suffit de pouvoir spécifier et prouver des procédures de décision, et de pouvoir exécuter des programmes vérifiés. Il ne serait donc sans doute pas trop difficile d'adapter notre mécanisme pour des plateformes de vérification telles que Leon [2] ou Dafny [10].

Références

[1] Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In Thorsten Altenkirch and Conor McBride, editors, *International Workshop on Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 48–62, Nottingham, UK, 2007.

[2] Régis William Blanc, Etienne Kneuss, Viktor Kuncak, and Philippe Suter. An overview of the Leon verification system: Verification by translation to recursive functions. In *4th Annual Scala Workshop*, 2013.

[3] Amine Chaieb and Tobias Nipkow. Proof synthesis and reflection for linear arithmetic. *Journal of Automated Reasoning*, 41(1):33–59, 2008.

[4] Guillaume Claret, Lourdes del Carmen González Huesca, Yann Régis-Gianas, and Beta Ziliani. Lightweight proof by reflection using a posteriori simulation of effectful computation. In *4th International Conference on Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 67–83. Springer, July 2013.

[5] Martin Clochard. Preuves taillées en biseau. In *Vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA)*, Gourette, France, January 2017.

[6] Martin Clochard, Léon Gondelman, and Mário Pereira. The Matrix reproved. *Journal of Automated Reasoning*, 60(3):365–383, 2017.

[7] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[8] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In Joe Hurd and Tom Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics*, pages 98–113, Oxford, UK, August 2005.

[9] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Centre, 1995.

[10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[11] Guillaume Melquiond and Raphaël Rieu-Helft. A Why3 Framework for Reflection Proofs and its Application to GMP's Algorithms. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *9th International Joint Conference on Automated Reasoning*, number 10900 in Lecture Notes in Computer Science, pages 178–193, Oxford, United Kingdom, July 2018.

[12] Niels Moller and Torbjörn Granlund. Improved division by invariant integers. *IEEE Transactions on Computers*, 60(2):165–175, 2011.

[13] Raphaël Rieu-Helft, Claude Marché, and Guillaume Melquiond. How to get an efficient yet verified arbitrary-precision integer library. In *9th Working Conference on Verified Software: Theories, Tools, and Experiments*, volume 10712 of *Lecture Notes in Computer Science*, pages 84–101, Heidelberg, Germany, July 2017.