

## WP 4.1

# Intégration du raisonnement sur l'arithmétique flottante dans Alt-Ergo

## 1. Introduction

L'intégration d'un raisonnement sur l'arithmétique flottante dans les démonstrateurs SMT commence à susciter un intérêt croissant ces dernières années. Ceci fait suite aux demandes et aux besoins des utilisateurs de ces démonstrateurs quant à la vérification de formules mathématiques contenant de l'arithmétique en virgule flottante. Considérée comme une théories « difficile », on s'est contenté dans le passé de l'approximer par l'arithmétique rationnelle/réelle, ou d'utiliser des logiciels de preuve spécialisés comme l'outil Gappa.

Il y a quelques années, une première expérience visant à intégrer la technologie de l'outil Gappa dans le démonstrateur SMT Alt-Ergo a été initiée. L'objectif était de prouver la validité de formules mathématiques raisonnablement petites issues de suites de tests fournis par Airbus. L'expérience était enrichissante. Néanmoins, le prototype avait quelques limitations (problèmes de correction, non-prise en compte de quelques axiomes importants sur les flottants, ...) et ne passait pas à l'échelle sur des exemples plus conséquents issus de l'outil SPARK et fournis par AdaCore dans le cadre de SOPRANO.

Nous allons voir ci-après les quelques pistes envisagées pour une meilleure gestion des flottants dans Alt-Ergo.

## 2. Nouvelle syntaxe

Contrairement à l'ancienne tentative d'intégrer Gappa dans Alt-Ergo, la nouvelle intégration apporte très peu de modifications au niveau du frontend du démonstrateur (lexer + parser + typechecker). En effet, lorsque le raisonnement sur l'arithmétique flottantes est activé, ce qui est actuellement réalisé via une option en ligne de commandes « **-use-fpa** » :

- un prélude déclarant différents types et symboles de fonctions et introduisant éventuellement des axiomes sur ces symboles est chargé. Ces types et ces symboles de fonctions deviennent de ce fait « pré-définis » pour l'utilisateur, qui n'a plus qu'à écrire ses formules à prouver en utilisant ces constructions. Le chargement du prélude (donné dans le tableau ci-dessous) va ainsi permettre de typer les formules données en entrée.

```
type fpa_rounding_mode =  
  (* five standard/why3 fpa rounding modes *)  
  NearestTiesToEven (*ne in Gappa: to nearest, tie breaking to even mantissas*)  
  | ToZero (* zr in Gappa: toward zero *)  
  | Up (* up in Gappa: toward plus infinity *)  
  | Down (* dn in Gappa: toward minus infinity *)  
  | NearestTiesToAway (* na : to nearest, tie breaking away from zero *)  
  
  (* additional Gappa rounding modes *)  
  | Aw (* aw in Gappa: away from zero **)  
  | Od (* od in Gappa: to odd mantissas *)  
  | No (* no in Gappa: to nearest, tie breaking to odd mantissas *)  
  | Nz (* nz in Gappa: to nearest, tie breaking toward zero *)
```

| Nd (\* nd in Gappa: to nearest, tie breaking toward minus infinity \*)  
| Nu (\* nu in Gappa: to nearest, tie breaking toward plus infinity \*)

(\* the first argument is mantissas' size (including the implicit bit),  
the second one is the exp of the min representable normalized number,  
the third one is the rounding mode, and the last one is the real to  
be rounded \*)

**logic** [float](#): int, int, fpa\_rounding\_mode, real -> real

(\* syntactic sugar for simple precision floats:  
mantissas size = 24, min exp = 149 \*)

**logic** [float32](#): fpa\_rounding\_mode, real -> real

(\* syntactic sugar for simple precision floats with default rounding mode  
(i.e. NearestTiesToEven)\*)

**logic** [float32d](#): real -> real

(\* syntactic sugar for double precision floats:  
mantissas size = 53, min exp = 1074 \*)

**logic** [float64](#): fpa\_rounding\_mode, real -> real

(\* syntactic sugar for double precision floats with default rounding mode  
(i.e. NearestTiesToEven) \*)

**logic** [float64d](#): real -> real

(\* type cast: from int to real \*)

**logic** [real\\_of\\_int](#) : int -> real

(\* abs value of a real \*)

**logic** [abs\\_real](#) : real -> real

(\* sqrt value of a real \*)

**logic** [sqrt\\_real](#) : real -> real

(\* abs value of an int \*)

**logic** [abs\\_int](#) : int -> int

(\* there are FPA axioms that are not inlined in the reasoning  
part. They are thus managed by the generic matching mechanism  
of Alt-Ergo \*)

(\* Remarque: should add semantic trigger 'x <= y'  
or maybe also 'float(m,p,md,x) > float(m,p,md,y)' in future  
version \*)

**axiom** [float\\_is\\_monotonic](#):

forall m, p : int.

forall md : fpa\_rounding\_mode.

forall x, y : real [float(m,p,md,x), float(m,p,md,y)].

x <= y -> float(m,p,md,x) <= float(m,p,md,y)

- Dans le back-end, les symboles de l'arithmétique linéaire (module Symbols.ml) a été étendu avec de nouvelles constructions relatives à l'arithmétique flottantes. Ainsi, lorsque l'option « **-use-fpa** » est activée, les constructions introduites par le prélude ci-dessus sont capturées et traduites vers des construites « pré-définis » du module Symbols.ml. Cela est réalisé lors de la traduction de l'AST typé vers une structure de données hashconsée mieux adaptée pour le moteur de raisonnement du démonstrateur.

### 3. Ré-intégration de Gappa dans Alt-Ergo

En plus du chargement du prélude de l'arithmétique flottante, l'option « **-use-fpa** » permet également d'activer le module Gappa.ml, qui fournit l'essentiel du raisonnement flottants (soit directement, soit en faisant appel à d'autres modules). La première étape pour ré-intégrer l'ancien prototype était d'adapter la signature des modules et l'implémentation aux versions récentes d'Alt-Ergo. Après cela, il fallait adapter les anciens benches à la nouvelle syntaxe.

Après cela, la prochaine étape était de reviewer le code et de tenter de comprendre quelles améliorations il fallait apporter et quelles extensions effectuer afin de pouvoir prouver les benches sur les flottants issus de SPARK.

**Passage à l'échelle :** de grosses améliorations ont été apportées à l'algorithme de Fourier-Motzkin, utilisé au coeur d'Alt-Ergo pour la déduction de bornes pour des polynômes et des monômes de l'arithmétique linéaire et non-linéaire. Cela a été notamment rendu possible en découpant les problèmes à résoudre en petits sous-problèmes indépendants, en détectant les subsumptions et les redondances d'inégalités, et en associant un « indice de fraîcheur » aux inégalités pour détecter quand il est vraiment nécessaire de relancer Fourier-Motzkin sur un sous-problème donné.

**Amélioration de raisonnement non-linéaire :** Lors de nos investigations, nous avons constaté qu'un peu plus de raisonnement non-linéaire était nécessaire pour prouver la validité de certains buts, comme factoriser  $a*a - a$  en  $a*(a - 1)$  ou déduire un intervalle pour  $a - b$  à partir de celui de  $a/b$  et de la contrainte  $b > 0$ . Nous avons donc effectué les extensions nécessaire pour intégrer ce genre de raisonnement.

**Bugs de correction dans intervals.ml :** Bien que gérant à la fois des unions d'intervalles sur les entiers et sur les rationnels, le module de représentation de et de manipulation d'intervalles dans Alt-Ergo (union, intersection, division, ...) a été surtout testé et utilisé dans le cadre de l'arithmétique entière. Comme l'arithmétique l'intervalles sur les rationnels est au coeur de la technologie « Gappa », nos investigations nous ont permis de découvrir près d'une dizaine de bugs de correction dans ce module, et de les fixer.

**Ajout des nouveaux termes introduits par le matching de Gappa :** Comme pour toute technique basée sur l'instantiation d'axiomes, la technologie « Gappa » introduisait de nouveaux termes lors de la déduction de nouveaux faits clos. Le souci est que ces termes ne sont pas remontés au niveau de la structure Union-Find d'Alt-Ergo, qui est sensée être initialisée avec tout les termes clos du problèmes et associer à chacun de ces termes sont représentant modulo théories. Ceci fait qu'on manque des déductions importantes pour prouver certains buts.

**Quels axiomes de Gappa sont gérés par le prototype « Alt-Ergo + Gappa » ?** Les axiomes (théorèmes) de Gappa sont représentés dans l'ancien prototype « Alt-Ergo+Gappa » en utilisant un DSL OCaml. En plus d'être assez verbeux, ce DSL rend la lecture de cette base d'axiomes et son éventuelle extension très difficile. Par exemple, il aurait fallu du temps pour comprendre que certains axiomes manquaient pour pouvoir montrer certains buts. C'est notamment le cas pour l'axiome d'idempotence de l'opération d'arrondi (i.e. **round(round(x)) = round(x)**), du cas d'une multiplication par une puissance de 2 positive (i.e. si  $n \geq 0$  alors **round (2<sup>n</sup> \* round(x)) = 2<sup>n</sup> \* round(x)**). Nous avons fini par « inliner » ces deux axiomes dans le module Gappa.

Pour d'autres formules, il fallait rajouter d'autres axiomes manquants. Par exemple, l'un d'eux nécessitait l'axiome de monotonie du symbole **float**. Nous avons fini par rajouter cet axiome directement dans le prélude des flottants (axiom [float\\_is\\_monotonic](#)) pour profiter du mécanisme de branchement du SAT. Pour d'autres encore, il fallait mettre en place une analyse par cas pour pouvoir conclure (eg. Lorsque **abs(a)** apparaît dans le contexte, il faudrait se demander si  $a \geq 0$  ou si  $a \leq 0$ ).

**Mélange arithmétique entière et arithmétique rationnelle :** Pour pouvoir « caster » des entiers en rationnels, l'utilisation de la fonction « **real\_of\_int** » était nécessaire en langage de surface pour pouvoir typer les formules données à Alt-Ergo. Or, dans le coeur du démonstrateur, cette fonction était tout simplement « effacée » forçant le mélange d'entiers et de rationnels dans les mêmes formules. Par exemple, la formule suivante en langage de surface :

$$\text{real\_of\_int}(a : \text{int}) : \text{real} = b : \text{real}$$

devient dans le coeur du démonstrateur :

$$a : \text{int} = b : \text{real}$$

Alt-Ergo ne gérant pas le mélange entiers/rationnels, cette transformation dans l'ancien prototype était fautive et passait totalement inaperçue. Une fois détecté, cet « effacement » a été désactivé et remplacé par des réécritures du style :

$$\begin{aligned} \text{real\_of\_int}(a + b) &= \text{real\_of\_int}(a) + \text{real\_of\_int}(b) \\ \text{real\_of\_int}(a * b) &= \text{real\_of\_int}(a) * \text{real\_of\_int}(b) \\ \dots \end{aligned}$$

Bien sûr, la solution idéale serait de gérer le mélange entiers/rationnels nativement dans Alt-Ergo. Mais ceci demanderait des modifications très conséquentes dans le démonstrateur.

#### 4. La suite

Lors de nos investigations, il nous était très difficile de déboguer chaque formule, de comprendre le raisonnement effectué et de déduire ce qui manque pour pouvoir faire la preuve. Dans les investigations en cours, nous essayons de réutiliser au maximum les composants déjà existants d'Alt-Ergo, qui sont bien testés et totalement maîtrisés, au lieu d'en développer de nouveaux. Cela passe notamment par :

1. la possibilité de décrire les axiomes de Gappa directement dans la syntaxe d'entrée d'Alt-Ergo au lieu d'utiliser un DSL. Ceci augmente la lisibilité et rend la modification et l'extension de l'axiomatisation plus immédiate.
2. Grâce à (1), la partie « instantiation des axiomes de Gappa » peut se faire, en partie,

en utilisant directement le module de matching d'Alt-Ergo. L'autre partie consiste à aller chercher des bornes dans les tables d'intervalles d'Alt-Ergo pour valider les hypothèse des axiomes avant de déduire leurs conclusions.

3. On peut, au choix, décider de traiter les instances générées localement dans la théorie, ou les remonter au SAT solver pour les gérer comme des instances d'axiomes classiques. La dernière solution pourrait être un peu lente en pratique Mais, elle nous assure la bonne initialisation des différents environnements des théories avec les termes frais créés par instantiation.

- **Release**

Les sources sont dans le répertoire :

**alt-ergo-gappa-5f59cf7c417f6b3245a70b63a9fd77108400a248**

Il suffit de faire « **./configure** » suivi de « **make** » dans ce répertoire pour générer le binaire.  
Il faut ensuite utiliser l'option « **-use-fpa** » pour activer les flottants.