Un Coq apprend à un bébé Colibri à flotter

Arthur Correnson, François Bobot

Un Coq apprend à un bébé Colibri à flotter

Arthur Correnson[1] and François Bobot[2]

[1] ENS de Rennes, Rennes, Bretagne, France
arthur.correnson@ens-rennes.fr
[2] CEA, Gif-sur-Yvette, France
francois.bobot@cea.fr

Résumé

L'arithmétique flottante est connue pour être un sujet difficile. Ses propriétés contre-intuitives rendent l'écriture d'algorithmes manipulant les nombres flottants propice à de nombreuses erreurs. Des outils automatiques pour la vérification de programmes flottants existent mais ces outils faisant eux-mêmes usage de calculs en arithmétique flottante, on peut se poser la question de leur propre fiabilité. Dans cet article, nous proposons de vérifier formellement l'implémentation de raisonnements sur les nombres flottants dans le solveur de contraintes Colibri2. En particulier, nous présentons une méthodologie pour mener la preuve de correction de propagateurs de contraintes en utilisant l'assistant de preuve Coq. Nous discutons également de l'intégration des raisonnements prouvés à un développement logiciel complet en OCaml.

1 Introduction

1.1 Solveurs et résolution de contraintes

Les outils de raisonnement automatique sont de plus en plus utilisés dans le contexte de la vérification formelle des logiciels. Parmi ces outils, les solveurs dits Satisfiabilité Modulo Théorie (SMT) sont certainement les plus utilisés car ils permettent d'effectuer de manière automatisée des preuves mobilisant plusieurs théories typiques des problèmes de vérification (théorie des tableaux, théorie des flottants, ...). Le standard SMT-LIB [4] propose un language commun pour interroger les solveur SMT ainsi qu'un ensemble de théories usuelles dans lesquelles les solveurs SMT peuvent raisonner. Si le standard SMT-LIB définit un langage commun pour les solveurs, il laisse en revanche complètement libre le choix des méthodes de résolution. Le solveur Colibri2, nouveau successeur de COLIBRI, propose d'utiliser des méthodes de programmation par contrainte (CP) pour la résolution de problème SMT. Un tel solveur interprète les problèmes SMT comme des ensembles de contraintes reliant plusieurs variables. On cherche ensuite une affectation de chaque variable satisfaisant l'ensemble des contraintes. Les algorithmes utilisés pour la résolution reposent sur l'idée de tenir compte de chacune des contraintes du problème pour restreindre progressivement l'espace de recherche associé à chaque variable. On parle alors de propagation de contraintes. Informellement, un propagateur de contrainte est une fonction qui pour toute contrainte répond à la question suivante : « si la contrainte est satisfaite, que puis-je en déduire sur la valeur de ses variables ». La correction d'un solveur CP est donc directement conditionnée par la correction des propagateurs de contraintes.

Après une présentation rapide de quelques difficultés notables liées à l'écriture correcte de propagateurs flottants, nous proposons une méthodologie pour la preuve de tels propagateurs à l'aide de l'assistant de preuve Coq. Nous détaillerons également comment extraire le code Coq vers du code OCaml fiable et prêt à être intégré aux sources de Colibri2.

1.2 L'arithmétique flottante et ses défauts

Représenter de manière exacte les nombres réels est impossible dans la mémoire finie d'un ordinateur. Les nombres flottants répondent à ce problème et propose une approximation finie des nombres réels permettant d'effectuer des calculs efficaces. De manière simplifiée, un nombre flottant peut-être vu comme un réel que l'on peut écrire sous la forme $m2^e$ avec $(m, e) \in \mathbb{Z}$. On contraint ensuite la taille de m et e pour assurer que tous les flottants ont une représentation finie. En imposant une précision (p) et des bornes sur l'exposant ($e_{\min}$ et $e_{\max}$), on peut alors décrire un ensemble des nombres flottants comme

$$\mathbb{F}_{p,e_{\min},e_{\max}} = \{m2^e \mid |m| < 2^p \wedge e_{\min} \leq e \leq e_{\max}\}$$

Notons qu'en fonction du choix des constantes p, $e_{\min}$ et $e_{\max}$ il existe plusieurs ensembles de flottants. D'une manière générale, une configuration $(p, e_{\min}, e_{\max})$ est appelée un format flottant. Cette représentation est une vue très mathématique et ne convient pas à l'implémentation de calculs effectifs sur les nombres flottants. Le standard IEEE-754 définit une représentation binaire des nombres flottants sur 16, 32 et 64 bits (cela revient à fixer un format) et formalise le comportement attendu des nombres flottants en machine. En particulier, ce standard définit la notion d'opérateur d'arrondi : des opérateurs permettant le passage d'un réel arbitraire au flottant le plus proche dans un format donné. Plusieurs opérateurs d'arrondi sont définis par le standard, le plus commun étant l'arrondi "au plus proche" aussi appelé RNE (*round to nearest, ties to even*). Pour simplifier la présentation dans la suite, on notera toujours $o : \mathbb{R} \to \mathbb{F}$ l'opérateur d'arrondi au plus proche.

Le standard IEEE-754 définit précisément les différentes opérations arithmétiques sur les flottants ainsi que leur spécification. Dans la suite on notera $\oplus_f^m$ pour l'addition flottante dans un mode d'arrondi donné (ou simplement $\oplus_f$ si le mode d'arrondi n'a pas d'importance) et $\leq_f$, $<_f$, $=_f$ pour les relations binaires. Nous noterons les opérations réelles en suivant les notations conventionnelles des mathématiques. Une propriété fondamentale du standard IEEE-754 est qu'il impose que les opérations arithmétiques correspondent exactement à l'arrondi de leur pendant réel. Par exemple pour deux flottants x et y on a $x \oplus_f y = o(x + y)$.

L'introduction de la notion d'arrondi entraîne des comportements tout à fait singuliers de l'arithmétique flottante qui la rendent fondamentalement plus difficile que l'arithmétique réelle. En particulier, selon l'ordre dans lequel les opérations binaires sont faites, on accumule les erreurs d'arrondis et on perd de ce fait l'associativité de l'addition et de la multiplication par exemple.

En plus des propriétés déjà soulignées, l'arithmétique des nombres flottants présente une autre difficulté de taille : la présence des valeurs spéciales et des zéros signés. Dans les nombres flottants, on s'autorise à manipuler les infinis comme des valeurs usuelles ce qui permet en particulier de donner un sens à la division par zéro. On admettra donc que $\pm 1/0 = \pm\infty$ dans les nombres flottants. L'introduction des infinis comme valeurs oblige à étendre l'arithmétique usuelle et à donner un sens à des expressions telles que $+\infty \oplus_f -\infty$. On introduit donc une valeur spéciale notée `NaN` pour désigner le résultat de toute opération résultant en une valeur non déterminée. Enfin, la présence d'un bit de signe dans la représentation mémoire des nombres flottants tels que décrits dans le standard IEEE-754 différencie un zéro positif noté 0^+ et un zéro négatif noté 0^-. L'égalité sur les flottants confond les deux zéros, mais $1/0^+ =_f +\infty$ alors que $1/0^- =_f -\infty$. Ces singularités engendrées par l'introduction des valeurs spéciales doivent être l'objet d'une attention toute particulière lorsque l'on écrit des algorithmes sur les nombres flottants.

1.3 Raisonnements corrects sur les programmes flottants

Nous venons d'énumérer quelques-unes des nombreuses difficultés de l'arithmétique flottante. Ces dernières rendent l'écriture d'algorithmes flottants particulièrement propices à des erreurs de programmation pouvant avoir des conséquences désastreuses (l'accumulation d'erreurs d'arrondis dans un calcul numérique de trajectoire de missile par exemple). Des outils de vérification formelle existent et permettent de valider la correction de programmes manipulant les nombres flottants [6, 10, 7]. Cependant, ces outils faisant eux mêmes l'usage de l'arithmétique flottante, on peut alors se demander ce qu'il en est de leur propre correction. Si ces derniers produisent des résultats incorrects, la sûreté des programmes vérifiés par leur intermédiaire peut-être compromise. L'effet est d'autant plus pervers que la confiance que nous donnons aux outils de vérification est grande.

Des initiatives pour prouver la correction des outils de vérification eux-mêmes ont déjà été initiées [11, 12]. Suivant cette idée, le solveur Colibri2 (développé en langage OCaml [2]) propose d'intégrer au sein même de son processus de développement la preuve formelle des composants les plus sensibles. Nous proposons dans cet article d'ajouter au solveur Colibri2 un support pour la théorie des nombres flottants. Cette extension consiste essentiellement en l'ajout de nouveaux propagateurs de contraintes spécifiques aux nombres flottants. Nous utiliserons l'assistant de preuve Coq [1] pour programmer et prouver ces propagateurs avant de les extraire pour les intégrer au solveur.

2 Résolution de contraintes, domaines et propagateurs

L'objectif final des travaux présentés dans cet article est l'intégration de raisonnements sur les flottants prouvés corrects dans un solveur CP. Si on se concentre uniquement sur des problèmes simples de contraintes numériques (arithmétique entière, réelle ou flottante par exemple), les solveurs CP raisonnent sur un ensemble de contraintes liant n variables $x_1, \ldots, x_n$ au moyens des relations binaires et opérateurs arithmétiques usuels ($=, \leq, <, +, -$). En fonction de l'arithmétique considérée, les variables prennent leurs valeurs dans un ensemble $E \in \{\mathbb{Z}, \mathbb{R}, \mathbb{F}_{32}, \ldots\}$ et les contraintes peuvent être vues comme des fonctions $c : E^n \to \texttt{bool}$. Dans ce contexte, décider de la satisfiabilité d'une formule φ revient à chercher une affectation $\lambda \in E^n$ des variables telle que pour toute contrainte c dans φ, $c(\lambda) = \texttt{true}$. L'ensemble de toutes les affectations possibles E^n est appelé l'espace de recherche et son sous-ensemble $S_\varphi = \{\lambda \in E^n \mid \forall c \in \varphi, c(\lambda) = \texttt{true}\}$ est appelé espace des solutions. Si $S_\varphi = \emptyset$, il n'existe aucune solution et on dit que φ est insatisfiable, dans le cas inverse φ est satisfiable et toute affectation de S_φ est appelée modèle de φ.

Pour résoudre un ensemble de contraintes numériques φ, les solveurs analysent et combinent les contraintes pour collecter des informations sur les valeurs que peuvent prendre chaque variable et ainsi calculer une sur-approximation $\bar{S}_\varphi \supseteq S_\varphi$ de l'espace des solutions. Pour chaque contrainte c, on élimine dans $\bar{S}$ les affectations qui invalide trivialement la contrainte c. On dit que l'on propage la contrainte c. Initialement, on prend $\bar{S} = E^n$, puis on propage toutes les contraintes dans φ jusqu'à stabilisation de S_φ. Si au cours du processus, S_φ devient $\emptyset$ alors φ est insatisfiable. Si le processus converge vers une approximation $\bar{S}_\varphi \neq \emptyset$, on essaye d'extraire de $\bar{S}$ un modèle en choisissant une valeur pour chaque variable dans l'ordre (d'abord pour x_1 puis pour x_2, etc.). À mesure que des choix de la forme $x_i = v$ sont faits, on peut propager ces décisions comme des contraintes. Si une décision provoque une contradiction, on l'annule ainsi que toutes les propagations qu'elle a engendré pour en faire une nouvelle : on parle de *backtracking*. On alterne les phases de décision, propagation, *backtracking* jusqu'à avoir affecté

une valeur à toutes les variables ou bien avoir testé toutes les décisions possibles sans succès.

En pratique on représente $\bar{S}$ comme une table qui associe à chaque expression e apparaissant dans φ des informations caractérisant une sur-approximation de son domaine d'appartenance. Ces informations peuvent être très simples (par exemple un intervalle d'appartenance) ou plus abstraites (par exemple une information de signe, un majorant, un bit connu, etc.) Intuitivement, toute donnée décrivant un ensemble de valeurs peut être utilisée. Toutefois, on souhaite *a minima* pouvoir propager ces informations à l'aide de propagateurs de contraintes.

Les travaux de Marie Pelleau et ses collaborateurs [13] ont déjà mis en avant un lien fort entre cette notion de domaine dans le contexte des solveurs CP et la notion de domaine abstrait issue de la discipline de l'interprétation abstraite [9]. Ce rapprochement donne un cadre théorique utile pour exprimer le comportement attendu des propagateurs de contraintes et ainsi élaborer leur preuve de correction. Nous rappelons à présent quelques définitions inspirées par ces travaux et qui nous serviront par la suite de support pour la preuve de propagateurs flottants.

Définition 2.1 (Domaine abstrait sur un ensemble E)**.** On appelle domaine abstrait sur l'ensemble E la donnée d'un quadruplet $(D, \top, \bot, \gamma)$ où :

- D est un ensemble et $\gamma : D \to \mathcal{P}(E)$ est une interprétation des éléments de D comme des parties de E.
- $\bot \in D$ représente l'ensemble vide ($\gamma(\bot) = \emptyset$) et permet de caractériser les contradictions.
- $\top \in D$ représente E ($\gamma(\top) = E$) et permet d'initialiser l'espace de recherche.

Dans le contexte de la résolution de contraintes, on équipe également les domaines abstraits avec les opérateurs suivants :

- Un opérateur d'intersection $\sqcap : D \times D \to D$ qui permet de combiner deux informations connues sur une expression.
- Pour chaque opérateur arithmétique $\circ : E \times E \to E$, un opérateur abstrait $\circ^\sharp : D \times D \to D$ qui permet d'assigner un domaine aux expressions arithmétiques.
- Pour chaque relation binaire $\sim \subseteq E \times E$, un propagateur de contrainte $\rho^\sim : D \times D \to D \times D$. Il permet de raffiner les informations connues sur deux expressions e_1 et e_2 sous l'hypothèse que $e_1 \sim e_2$.

Pour garantir la correction des solveurs, on impose que ces nouveaux opérateurs aient certaines propriétés.

Hypothèse 2.1 (Correction de $\sqcap$)**.** Soient d_1 et d_2 deux éléments de D et x un élément de E, on impose que $\gamma(d_1) \cap \gamma(d_2) \subseteq \gamma(d_1 \sqcap d_2)$.

Hypothèse 2.2 (Correction des opérateurs)**.** Soit $\circ : E \times E \to E$ et soient d_1 et d_2 deux éléments de D, on impose que $\{x \circ y \mid x \in \gamma(d_1) \wedge y \in \gamma(d_2)\} \subseteq \gamma(d_1 \circ^\sharp d_2)$.

Hypothèse 2.3 (Correction des propagateurs)**.** Soit $\sim \subseteq E \times E$ une relation binaire, soient d_1 et d_2 deux éléments de D, soit $(d_1', d_2') = \rho^\sim(d_1, d_2)$ et soient x et y deux éléments de E, on impose deux propriétés :

- $\rho^\sim$ améliore d_1 et d_2 : $\gamma(d_1') \subseteq \gamma(d_1) \wedge \gamma(d_2') \subseteq \gamma(d_2)$
- $\rho^\sim$ respecte $\sim$: Si $x \sim y$ alors $x \in \gamma(d_1') \wedge y \in \gamma(d_2')$

Les propriétés de correction des propagateurs et des opérateurs abstraits assurent que leur utilisation permet toujours de calculer une sur-approximation de l'espace de recherche. Le fait que les propagateurs améliorent les domaines permet de garantir la convergence des solveurs : on ne fait jamais grossir l'espace de recherche en propageant une contrainte.

2.1 Domaine d'intervalles

Pour illustrer la démarche adoptée pour le développement et la preuve d'un domaine abstrait, nous prenons l'exemple des intervalles flottants. En présence de contraintes d'inégalités, ce domaine abstrait s'avère très utile car il permet d'identifier rapidement des bornes inférieures et supérieures pour chaque variable du problème à résoudre. Par exemple pour le problème $\varphi := \{x \leq 3 \wedge x \geq 4\}$, l'utilisation d'un domaine d'intervalles révèle immédiatement que φ est insatisfiable après deux propagations :

1. On note dx le domaine de x et on initialise $dx := \top = \{-\infty, +\infty\}$
2. On propage la contrainte $x \leq 3 : dx$ devient $\texttt{fst}(\rho^{\leq}(dx, \{3, 3\})) = \{-\infty, 3\}$
3. On propage la contrainte $x \geq 4 : dx$ devient $\texttt{fst}(\rho^{\geq}(dx, \{4, 4\})) = \bot$
4. $dx = \bot$, sans possibilité de *backtracking* la résolution prend fin et φ est insatisfiable.

Dans les réels, il est d'usage de définir les intervalles comme des pairs $(a, b) \in \mathbb{R}^2$. L'ensemble dénoté par l'intervalle (a, b) est ensuite défini comme $\{x \in \mathbb{R} \mid a \leq x \leq b\}$. Pour permettre un meilleur découpage de $\mathbb{R}$ en intervalles, on prolonge généralement cette définition en autorisant des bornes dans $\bar{\mathbb{R}} := \mathbb{R} \cup \{-\infty, +\infty\}$ et on distingue les bornes d'intervalles strictes et non strictes. Dans les flottants, les infinis sont des valeurs usuelles et peuvent donc être traitées indistinctement des autres. De plus, la finitude des nombres flottants permet de définir les opérateurs unaire *pred* et *succ* qui à tout flottant associe respectivement son prédécesseur et son successeur autorisant ainsi à ramener les tests d'inégalités strictes à des tests d'inégalités larges. Les notions de bornes strictes et non strictes peuvent donc être ignorées dans le cas flottant.

La présence du `NaN` perturbe la définition des intervalles flottants. Rappelons que la valeur `NaN` n'est pas comparable, c'est à dire que $\forall x \in \mathbb{F}, \neg(x \sim \texttt{NaN} \vee \texttt{NaN} \sim x)$ (avec $\sim \in \{\leq_f, \geq_f, <_f, >_f, =_f\}$). Tout intervalle flottant ayant une borne `NaN` représente donc un intervalle vide et par conséquent il n'existe pas d'intervalle contenant `NaN`. Néanmoins, dans le contexte d'un solveur, modéliser la possibilité qu'une expression puisse prendre la valeur `NaN` est utile. En effet prendre en compte la valeur `NaN` peut permettre de détecter des contradictions supplémentaires et donc d'accroître les capacités de résolution des solveurs. Par exemple, toute contrainte de la forme $x \sim \texttt{NaN}$ est insatisfiable et il est souhaitable qu'un domaine d'intervalle soit en mesure de capturer cette propriété. Pour prendre en compte `NaN`, nous étendons la définition des intervalles flottants comme $\mathbb{I}(\mathbb{F}) = \mathbb{F}^2 \times \texttt{bool}$. Un triplet $(a, b, \text{nan}) \in \mathbb{F}^2 \times \texttt{bool}$ dénote l'ensemble $\{x \in \mathbb{F} \mid a \leq_f x \leq_f b \vee (x = \texttt{NaN} \wedge \text{nan})\}$.

Nous définissons à présent les opérateurs $\sqcap$, $+^\sharp$ et $\rho^{\leq}$ associés au domaine abstrait des intervalles et tels que spécifiés précédemment.

Définition 2.2 (Concrétisation)**.** Les éléments de $\mathbb{I}(\mathbb{F})$ peuvent être interprétés comme des parties de $\mathbb{F}$ par la fonction de concrétisation γ définie comme :

$$\gamma((a, b, \text{nan})) := \{x \in \mathbb{F} \mid a \leq x \leq b \vee (x = \texttt{NaN} \wedge \text{nan})\}$$

Définition 2.3 (Intersection)**.** L'intersection peut se définir de la façon suivante :

$$(a, b, \text{nan}) \sqcap (a', b', \text{nan'}) := (\max(a, a'), \min(b, b'), \text{nan} \wedge \text{nan'})$$

Théorème 2.1 (Correction et précision de l'intersection)**.** Soient d_1, d_2 deux intervalles de $\mathbb{I}(\mathbb{F})$, alors :

— correction : $\forall x \in \mathbb{F}, x \in \gamma(d_1) \wedge x \in \gamma(d_2) \Rightarrow x \in \gamma(d_1 \sqcap d_2)$

— précision : $\forall x \in \gamma(d_1 \sqcap d_2), x \in \gamma(d_1) \wedge x \in \gamma(d_2)$

Démonstration. Par analyse de cas sur les valeurs nan et nan', et à l'aide des propriétés du max et du min on obtient la précision et la correction de $\sqcap$. □

Définition 2.4 (Opérateurs abstraits)**.** Les opérateurs abstraits $\circ^\sharp$ s'obtiennent à partir des opérateurs binaires $\circ$. Nous prenons l'exemple de l'addition :

$$(a,b,n) +^\sharp (a',b',n') := (a \oplus_f a', b \oplus_f b', n \vee n' \vee \bigvee_{\substack{x\in\{a,a'\}\\ y\in\{b,b'\}}} \text{sum-to-nan}(x,y))$$

avec sum-to-nan(x,y) si et seulement si $x \oplus_f y =$ `NaN` c'est à dire si x et y sont deux infinis opposés ou si l'une des deux valeurs est `NaN`.

Théorème 2.2 (Correction de l'addition abstraite)**.** Soient d_1, d_2 deux intervalles de $\mathbb{I}(\mathbb{F})$, alors $\forall (x,y) \in \mathbb{F}^2, x \in \gamma(d_1) \wedge y \in \gamma(d_2) \implies x \oplus_f y \in \gamma(d_1 +^\sharp d_2)$

Démonstration. Soient $d_1 = (a, b, \text{nan})$ et $d_2 = (a', b', \text{nan'})$, on raisonne par analyse de cas selon que les bornes a, a', b, b' et la somme $x \oplus_f y$ soient `NaN`, $\pm\infty$, ou des valeurs finies. Dans les cas où les trois sommes $a \oplus_f a'$, $x \oplus_f y$ et $b \oplus_f b'$ sont différentes de `NaN`, la monotonie de l'addition flottante s'applique (voir 3.2) et on peut déduire des hypothèses que $a \oplus_f a' \leq_f x \oplus_f y \leq_f b \oplus_f b'$ et donc que $x \oplus_f y \in \gamma(d_1 +^\sharp d_2)$. Dans les autres cas, les règles de l'arithmétique flottante pour les valeurs spéciales s'appliquent et la preuve s'achève par simple calcul. □

Les propagateurs de contraintes $\rho^\sim$ peuvent être définis en deux temps. Dans un premier temps nous définissons un opérateur $\kappa^\sim : D \to D$ qui calcule le domaine des candidats à la satisfaction de la relation $\sim$ par rapport à un domaine donné. Formellement on souhaite que $\kappa^\sim(d) \supseteq \{y \in \mathbb{F} \mid \exists x \in \gamma(d), y \sim x\}$. Par exemple pour les comparaisons flottantes $\leq_f$ et $\geq_f$ on a :

$$\kappa^{\leq}(a,b,\text{nan}) := \texttt{if } b <_f a \texttt{ then } (a,b,\texttt{false}) \texttt{ else } (-\infty,b,\texttt{false})$$

$$\kappa^{\geq}(a,b,\text{nan}) := \texttt{if } b <_f a \texttt{ then } (a,b,\texttt{false}) \texttt{ else } (a,+\infty,\texttt{false})$$

Définition 2.5 (Propagateurs)**.** Le propagateur $\rho^\sim$ se construit ensuite à l'aide de l'intersection.

$$\rho^{\leq}(d_1,d_2) := (\kappa^{\leq}(d_2) \sqcap d_1, \kappa^{\geq}(d_1) \sqcap d_2)$$

Théorème 2.3 (Correction du propagateur $\rho^{\leq}$)**.** Soient d_1 et d_2 deux intervalles flottants et soit $(d_1', d_2') = \rho^{\leq}(d_1, d_2)$, on a $\forall x \in \gamma(d_1), \forall y \in \gamma(d_2), x \leq_f y \implies x \in \gamma(d_1') \wedge y \in \gamma(d_2')$.

Démonstration. Par minimalité de $-\infty$ et maximalité de $+\infty$ pour les flottants et par correction de $\sqcap$. □

Nous montrons également que $\rho^{\leq}(d_1, d_2)$ ne dégrade pas la précision de d_1 et d_2.

Théorème 2.4 ($\rho^{\leq}$ améliore les domaines ou stagne)**.** Soient d_1 et d_2 deux intervalles flottants et $(d_1', d_2') = \rho^{\leq}(d_1, d_2)$, on a $\gamma(d_1') \subseteq \gamma(d_1) \wedge \gamma(d_2') \subseteq \gamma(d_2)$.

Démonstration. Ce résultat est un corollaire de la précision de $\sqcap$. □

Une implémentation Coq de ce domaine d'intervalles accompagnée des preuves de correction est disponible en ligne à l'adresse https://colibri.frama-c.com/html/doc/farith/doc/F.All.html. La méthodologie employée pour les preuves est présentée en détail dans la partie 3. D'autres développements proposent également de prouver formellement des domaines flottants en utilisant l'assistant de preuve Coq. C'est le cas notamment du projet Verasco [11] qui propose une implémentation en Coq de plusieurs domaines abstraits dont des domaines flottants. Les domaines flottants de Verasco sont développés dans le contexte de la vérification par interprétation abstraite et ne traitent que des flottants de format 32 ou 64 bits. L'implémentation que nous proposons est générique pour tout format flottant.

2.2 Correction complète d'un solveur CP

La correction des propagateurs ne suffit pas à assurer la correction complète d'un solveur CP. En effet, d'autres paramètres sont à considérer dont notamment la terminaison. De même, il faut s'assurer que l'algorithme qui orchestre la résolution fait une utilisation raisonnable des propagateurs. La démarche de prouver formellement l'algorithme de résolution de Colibri2 a déjà été amorcée mais nous ne détaillerons pas ici ces aspects. Notons toutefois que la preuve de correction et de terminaison d'un algorithme de résolution de contraintes requiert que les propagateurs utilisés soient corrects et, à ce titre, notre démarche constitue déjà une étape importante dans l'élaboration d'une telle preuve.

Prouver formellement un algorithme de résolution de contraintes dans un assistant à la démonstration ou tout autre outil de vérification formelle (comme Why3 par exemple [5]) représente un coût de développement conséquent. De ce fait, les fragments entièrement prouvés de Colibri2 sont modestes et fournissent un moteur de résolution correct mais peu puissant. En prouvant les propagateurs séparément, les mêmes propagateurs prouvés peuvent-être utilisés aussi bien dans un algorithme de résolution entièrement formalisé (auquel cas, les preuves de correction des propagateurs jouent un rôle dans la preuve de correction du solveur entier), que dans un algorithme de résolution plus avancé pour lequel on tolère l'absence d'une preuve de correction complète (dans ce cas, la simple correction des propagateurs donne déjà des garanties fortes).

3 Formalisation Coq des nombres flottants

3.1 La bibliothèque Flocq

Nous souhaitons intégrer au solveur Colibri2 un support minimal pour la théorie des nombres flottants. Nous proposons à cet effet une implémentation Coq du domaine d'intervalles flottants présenté dans la partie 2.1. Cette implémentation est ensuite traduite en code exécutable afin d'être intégrée dans les sources du solveur. Pour atteindre ces objectifs, nous avons besoin d'une modélisation des nombres flottants au sein de l'assistant de preuve Coq. Cette même modélisation sera utilisée dans les preuves mais également comme implémentation de référence de l'arithmétique flottante. Utiliser la même modélisation des flottants pour la preuve et pour le calcul permet d'assurer que les propriétés établies en Coq sont préservées après extraction (sous l'hypothèse que le mécanisme d'extraction vers OCaml de Coq est correcte).

Une modélisation des flottants. Le standard IEEE-754 donne une définition très complète des nombres flottants qui convient parfaitement comme guide pour l'implémentation d'une arithmétique flottante en machine. Néanmoins une représentation binaire est assez peu adaptée

pour formaliser la notion de nombre flottant dans un assistant à la démonstration. Pour permettre à la fois des raisonnements formels plus simples sur les nombres flottants et des calculs effectifs au sein d'un assistant à la démonstration, la bibliothèque Coq Flocq [8] propose un compromis et fournit une implémentation des flottants comme des nombres de la forme $m2^e$. Cette modélisation des nombres flottants est prouvée correcte vis à vis du standard IEEE-754 et peut être extraite vers un module OCaml fiable pour le calcul flottant. Ces propriétés font donc de Flocq un outil de choix pour la suite des travaux présentés dans cet article.

Opérateurs flottants. Flocq fournit un type flottant `binary_float` paramétré par une précision et un exposant maximal (l'exposant minimal est déduit à partir de l'exposant maximal et de la précision tel que décrit dans le standard IEEE-754). Le type `binary_float` est défini comme un type algébrique avec un constructeur pour chacune des valeurs spéciales (`NaN`, $0^{\pm}$, $\pm\infty$) et un constructeur pour tous les flottants finis de la forme $m2^e$. Cette modélisation permet de réduire les démonstrations en arithmétique flottante à une analyse de cas sur la forme des flottants considérés.

Les opérateurs usuels paramétrés par un mode d'arrondi sont également fournis. Par soucis de simplicité, nous omettrons les paramètres correspondant au format dans les échantillons de code Coq et nous noterons simplement `float` pour désigner le type des flottants dans un format fixé. En suivant cette convention, on considère les opérateurs suivants :

relations binaires	`Beqb, Bleb, Bltb : float -> float -> bool`
addition flottante	`Bplus : mode -> float -> float -> float`
constante `NaN`	`NaN : float`
constantes infinies	`B754_infinity : bool -> float`
projection dans les réels	`B2R : float -> R`
opérateur d'arrondi générique [1]	`round : mode -> R -> R`

3.2 Monotonie : un réel problème de sémantique

Établir la preuve formelle d'un théorème en Coq est souvent une tâche longue et difficile, en particulier lorsque les preuves impliquent de mobiliser des résultats d'arithmétique et *a fortiori* d'arithmétique flottante. La preuve de correction d'opérateurs sur les domaine abstraits est très consommatrice de ce type de résultats et en particulier de théorèmes de monotonie qui assurent la préservation de la relation `Bleb` à travers les opérations arithmétiques. Ces théorèmes sont triviaux dans $\mathbb{R}$ mais plus difficiles à formuler voire même faux dans les flottants. La monotonie de l'addition réelle par exemple assure que $\forall (x, y, z) \in \mathbb{R}^3, x \leq y \rightarrow x + z \leq y + z$, mais la valeur `NaN` ne pouvant être comparée, il s'ensuit que ce résultat est faux dans les flottants. En effet, pour $x = -\infty$, $y = +\infty$, $z = +\infty$, on a bien $x \leq_f y$ mais $x \oplus_f z = \texttt{NaN}$ et donc $\neg(x \oplus_f z \leq_f y \oplus_f z)$. Pour des raisons similaires, la plupart des résultats d'arithmétique flottante sont conditionnés par des hypothèses spécifiques sur les opérandes. Ainsi la monotonie de l'addition dans $\mathbb{F}$ se ré-exprime de la façon suivante :

Théorème 3.1 (Monotonie de l'addition flottante)**.** Soit m un mode d'arrondi et x, y, x', y' quatre flottants. Si $x \oplus_f^m x' \neq \texttt{NaN}$, $y \oplus_f^m y' \neq \texttt{NaN}$, $x \leq_f y$ et $x' \leq_f y'$ alors $x \oplus_f^m x' \leq_f y \oplus_f^m y'$.

1. On pourrait s'attendre à ce qu'un opérateur d'arrondi soit de signature $\mathbb{R} \rightarrow \mathbb{F}$, mais une telle définition ne serait pas calculable car les réels ne sont pas représentables en machine. En revanche, une axiomatisation des réels suffit pour l'écriture de spécifications.

Pour établir la preuve de ce théorème en Coq on introduit deux résultats intermédiaires plus simples `Bplus_le_compat_l` et `Bplus_le_compat_r`. La monotonie de l'addition découle directement de ces deux résultats.

```
Theorem Bplus_le_compat_l :
  forall (m : mode) (x y z : float),
    is_nan (Bplus m z x) = false ->
    is_nan (Bplus m z y) = false ->
    x <= y ->
    Bplus m z x <= Bplus m z y
```

```
Theorem Bplus_le_compat_r :
  forall (m : mode) (x y z : float),
    is_nan (Bplus m x z) = false ->
    is_nan (Bplus m y z) = false ->
    x <= y ->
    Bplus m x z <= Bplus m y z
```

FIGURE 1 – Monotonie de l'addition flottante

Pour établir les preuves de ce type de résultats en arithmétique flottante, une première approche pourrait être de dérouler les définitions des opérations flottantes impliquées et de raisonner directement au niveau de la représentation des flottants. Raisonner à si bas niveau est généralement assez difficile. Une autre approche consiste à raisonner à un niveau d'abstraction plus élevé en plongeant les flottants dans les réels. Cette méthode présente de nombreux avantages. D'une part nous avons l'habitude de manipuler l'arithmétique réelle et les raisonnements en sont d'autant plus faciles à mener. D'autre part, Coq propose des outils pour automatiser les preuves en arithmétique réelle linéaire et non-linéaire. La bibliothèque Flocq fourni une fonction `B2R : float -> R` qui donne l'interprétation réelle des flottants ainsi que de multiples théorèmes définissant la sémantique des opérateurs flottants dans $\mathbb{R}$. Une fois que les propriétés que nous souhaitons démontrer sont établies dans $\mathbb{R}$, il faut pouvoir garantir que les résultats sont transposables dans les flottants. Pour se faire, on introduit des lemmes de préservation garantissant que certaines propriétés réelles sont toujours vraies dans les flottants.

Un premier résultat qui permet le passage de flottant à réel est la correction de l'addition flottante. Ce théorème donne la sémantique de l'addition flottante dans $\mathbb{R}$ et énonce que la somme de deux *petits* flottants finis est égale à l'arrondi de leur somme réelle. Dans le cas où la somme réelle provoque un débordement de capacité même une fois arrondi en un flottant de précision arbitraire, l'addition flottante renvoie une valeur d'*overflow* qui peut varier selon le mode d'arrondi et le signe des opérandes. Les dépassements de capacité peuvent produire la valeur $\pm\infty$ et donc n'ont pas de représentation réelle ; ce qui oblige à formuler le théorème de correction en prenant en compte deux cas selon que la somme déborde ou non. En supposant donnée une fonction `overflow : R -> bool` qui teste les dépassements de capacité pour un format flottant particulier et une fonction `binary_overflow : mode -> bool -> float` produisant la valeur de débordement en fonction d'un signe et d'un mode d'arrondi, le théorème de correction de l'addition flottante peut s'exprimer de la façon suivante :

```
Theorem Bplus_correct :
  forall (m : mode) (x y : float), is_finite x -> is_finite y ->
    if overflow (round m (B2R x + B2R y)) then
      Bplus m x y = binary_overflow m (Bsign x)
    else B2R (Bplus m x y) = round m (B2R x + B2R y)
```

FIGURE 2 – Sémantique réelle de l'addition flottante dans un mode d'arrondi donné

À chaque fois que nous souhaitons utiliser le théorème de correction de l'addition sans

hypothèses sur les opérandes flottants, nous sommes contraints de raisonner par analyse de cas et d'envisager la possibilité que les opérandes soient infinies ou que la somme flottante déborde. Cela rend les preuves longues et fastidieuses. Pour éviter d'avoir à traiter séparément tous ces cas, nous proposons de donner une nouvelle interprétation des flottants dans $\bar{\mathbb{R}} = \mathbb{R} \cup \{+\infty, -\infty\}$.

```
Inductive Rx : Type := Inf : bool -> Rx | Real : R -> Rx.

Definition B2Rx (f : float) : Rx :=
  match x with
  | B754_infinity b => Inf b
  | _ => Real (B2R f)
  end.
```

FIGURE 3 – Plongement des flottants dans $\bar{\mathbb{R}}$

Nous étendons le type `R` de Coq par un type `Rx` qui comprend les constantes $\pm\infty$. La fonction `B2Rx : float -> Rx` donne la sémantique des flottants dans $\bar{\mathbb{R}}$. Pour spécifier les opérateurs flottants dans $\bar{\mathbb{R}}$, nous redéfinissons également un arrondi `roundx` dans $\bar{\mathbb{R}}$. Contrairement à l'arrondi générique de Flocq, `roundx` donne soit une constante infinie (si le réel arrondi est trop grand) soit un réel de la forme $m2^e$ respectant strictement le format flottant imposé ($m < 2^{prec}$ et $e_{\min} \leq e \leq e_{\max}$).

Nous dotons également `Rx` des opérations usuelles $+, -, *, \ldots$ et des relations $\leq, \geq, <, >$ définies comme des extensions naturelles de leurs équivalents réels. Notons qu'il reste des zones d'ombres sur la valeur à attribuer à des expressions telles que `(B2Rx NaN)` ou `(Inf true + Inf false)`. Nous choisissons d'attribuer des valeurs arbitraires et nous montrons formellement qu'en dépit de ces choix, la nouvelle sémantique des flottants dans `Rx` se comporte comme la sémantique habituelle dans `R`. En particulier, nous montrons la monotonie de l'arrondi qui est une propriété fondamentale des nombres flottants.

```
Lemma roundx_mono :
  forall (m : mode) (r1 r2 : Rx), r1 <= r2 -> roundx m r1 <= roundx m r2.
```

FIGURE 4 – Monotonie de l'arrondi dans $\bar{\mathbb{R}}$

Ce passage de `R` à `Rx` nous permet de donner une expression plus générale du théorème de correction de l'addition (5).

```
Theorem Bplus_correct' :
  forall (m : mode) (x y : float),
    is_nan (Bplus m x y) = false ->
    B2Rx (Bplus m x y) = roundx m (B2Rx x + B2Rx y).
```

FIGURE 5 – Sémantique de l'addition flottante dans $\bar{\mathbb{R}}$

L'énoncé du théorème de correction de l'addition flottante dans $\bar{\mathbb{R}}$ est plus générique et l'hypothèse `is_nan (Bplus m x y) = false` est strictement plus faible que de supposer la finitude des opérandes. Dans sa version initiale, si l'on souhaite utiliser le théorème de correction

```
Lemma le_B2Rx :
  forall (x y : float), x <= y -> B2Rx x <= B2Rx y.

Lemma B2Rx_le :
  forall (x y : float), is_nan x = false -> is_nan y = false ->
  B2Rx x <= B2Rx y -> x <= y.
```

FIGURE 6 – Lemmes de préservation de l'ordre par passage de $\bar{\mathbb{R}}$ à $\mathbb{F}$

de l'addition sans avoir aucune hypothèse sur x et y, cela oblige à considérer 8 cas selon que x et y soient finis ou non et selon que leur somme déborde ou non. Le nouveau théorème que nous proposons limite cette analyse à deux cas seulement : la somme flottante vaut `NaN` ou non. Les preuves sont de ce fait plus courtes, plus simples à comprendre et plus faciles à développer. Pour illustrer cela, prenons l'exemple de la preuve du théorème `Bplus_le_compat_r`. Comme décrit précédemment, on commence par introduire des lemmes de préservation (voir figure 6) puis nous établissons la preuve du théorème `Bplus_le_compat_r`.

Démonstration. Soient x, y, z trois flottants et m un mode d'arrondi quelconque. On suppose que $x \oplus_f^m z$ et $y \oplus_f^m z$ sont différents de `NaN` et que $x \leq_f y$. Par le théorème `le_B2Rx`, comme $x \leq_f y$ on a également $\texttt{B2Rx}(x) \leq_f \texttt{B2Rx}(y)$. Montrons à présent que $x \oplus_f^m z \leq_f y \oplus_f^m z$. Par `B2Rx_le` cela équivaut à montrer que $\texttt{B2Rx}(x \oplus_f^m z)$ est plus petit que $\texttt{B2Rx}(y \oplus_f^m z)$.
En application du théorème de correction de l'addition flottante `Bplus_correct'`, cela revient à montrer $\texttt{roundx}_m(\texttt{B2Rx}(x) + \texttt{B2Rx}(z)) \leq_f \texttt{roundx}_m(\texttt{B2Rx}(y) + \texttt{B2Rx}(z))$. Par monotonie de l'arrondi (lemme `roundx_mono`), et monotonie de l'addition réelle cette propriété s'établit trivialement sous l'hypothèse que $\texttt{B2Rx}(x) \leq_f \texttt{B2Rx}(y)$. □

En pratique, cette stratégie de preuve s'est généralisée à tous les résultats d'arithmétique flottante dont nous avons eu besoin d'établir la preuve. Rappelons que l'objectif final de ces travaux est de montrer la correction de propagateurs flottants. Par essence, les propagateurs ont pour but, étant donnée une hypothèse P faite sur deux nombres flottants, d'en déduire une conséquence Q plus précise. Leur preuve de correction se réduit donc à prouver des théorèmes de la forme $P \implies Q$ avec P, Q deux assertions sur les flottants. La méthode de preuve suivante s'applique alors de manière assez systématique.

1. On souhaite prouver des énoncés de la forme $P \implies Q$ où P et Q sont des assertions sur les nombres flottants.
2. On suppose P et on exploite la sémantique réelle des flottants pour en déduire P', une réécriture de P dans $\bar{\mathbb{R}}$ obtenue en remplaçant tous les opérateurs et relations flottants par leurs équivalents dans $\bar{\mathbb{R}}$.
3. En utilisant des résultats simples d'arithmétique réelle on prouve $P' \implies Q'$ avec Q' une réécriture dans $\bar{\mathbb{R}}$ de Q.
4. On développe la preuve du lemme de préservation $Q' \implies Q$ qui assure que la propriété réelle Q' est préservée dans les flottants.
5. On a P, $P \implies P'$, $P' \implies Q'$ et $Q' \implies Q$ donc par transitivité $P \implies Q$ ce qui achève la preuve.

Cette méthode nous a permis de formuler un nombre important de résultats utiles sur les nombres flottants dont notamment des résultats de monotonie rendant possible la preuve

formelle de correction des propagateurs pour le domaine abstrait des intervalles. L'opérateur abstrait $+^\sharp$, l'intersection $\sqcap$ ainsi que les propagateurs pour les contraintes $\rho^=$, $\rho^\leq$, $\rho^\geq$, $\rho^<$ et $\rho^>$ ont été prouvés grâce à cette méthode. Ces propagateurs couvrent exactement le fragment linéaire de l'arithmétique flottante ce qui permet déjà de résoudre les problèmes SMT les plus simples.

4 Extraction vers OCaml : les flottants découvrent le monde réel

4.1 Colibri2

Colibri2 est un solveur de contraintes écrit en OCaml. En son cœur il propose un graphe d'égalité avec domaines. Une telle structure permet de garder à jour des informations sur les valeurs et les domaines associés à chaque sous-terme d'un problème tout en tenant compte des égalités potentielles entre termes. Un nœud du graphe (Node.t) représente une classe d'équivalence entre des termes liés par une contrainte d'égalité. Les domaines permettent d'attacher une information à toute une classe d'équivalence. Les valeurs (Value.t) permettent de représenter les constantes. Pour obtenir un modèle, le solveur cherche à associer à chaque nœud une valeur avec la règle qu'une unique valeur peut être associée à une classe d'équivalence.

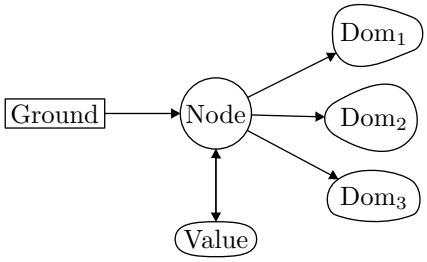

FIGURE 7 – Vision simplifiée des différents objets composant le Egraph dans Colibri2

L'architecture de Colibri2 est modulaire et permet d'enregistrer des nouveaux domaines abstraits. Tout module présentant au moins un type `t` et une fonction d'intersection peut être intégré comme un nouveau module de domaines. Les modules de domaines doivent également exposer une fonction `is_singleton` qui permet de détecter lorsqu'un domaine ne contient qu'une unique valeur et retourne cette valeur le cas échéant. Additionnellement, les modules de domaines peuvent fournir des propagateurs de contraintes.

Cette architecture permet d'implémenter les domaines abstraits séparément du cœur du solveur. En particulier, nous pouvons implémenter un domaine abstrait en Coq, l'extraire en OCaml et l'intégrer directement aux sources de Colibri2. Le domaine des intervalles flottants présenté dans cet article ainsi qu'une extraction vers OCaml de la bibliothèque Flocq sont ainsi intégrés à Colibri2 sous la forme d'un module nommé Farith2.

4.2 Le(s) problème(s) de l'extraction

Pour intégrer la théorie des nombres flottants au solveur Colibri2, deux éléments principaux sont requis. D'une part un module de valeurs qui fournit un type pour les nombres flottants ainsi que ses opérations arithmétiques élémentaires. Celui-ci permettra au Egraph d'associer des

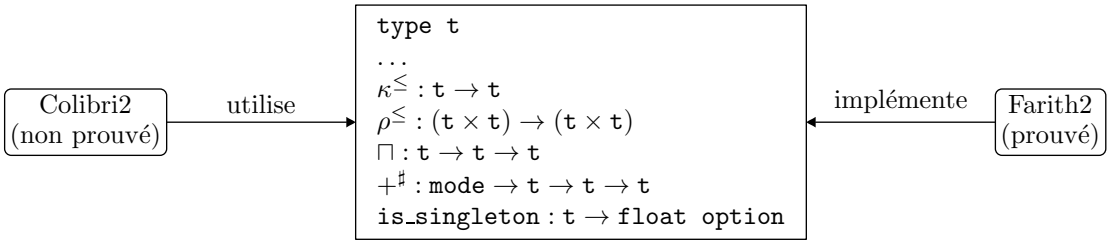

FIGURE 8 – Schéma de collaboration entre Colibri2 et Farith2

valeurs (Value.t) représentant un nombre flottant à un nœud. D'autre part, nous souhaitons enregistrer des nouveaux domaines flottants et leurs propagateurs associés.

Comme décrit plus tôt (voir section 3), la modélisation Coq que nous donnons des nombres flottants expose principalement un type `binary_float` paramétré par deux entiers relatifs (dénotant une précision et un exposant maximale). De plus, les opérateurs sur le type `binary_float` imposent comme préconditions que la précision soit strictement positive et que l'exposant maximal soit supérieure à la précision.

L'extraction naïve du type `binary_float` vers OCaml donne un type simple et incorrect : la partie dépendante du type est effacée et les flottants des différents formats sont confondus. De même, l'extraction d'un opérateur tel que `Bplus` ignore les préconditions. Ces conditions sont nécessaires pour établir la correction des opérateurs et le code extrait n'est donc pas nécessairement correct puisqu'il ne garantit pas leur satisfaction.

4.3 Effacement des types dépendants par instanciation de modules

Une solution partielle à ce problème est de proposer une ou plusieurs instanciations du type `binary_float` et des opérateurs associés. Ces instanciations sont réalisées en Coq de manière à garantir leur utilisation correcte. En particulier on pourra prouver pour chaque opérateur que les paramètres de précision et d'exposant choisis pour l'instanciation vérifient les pré-conditions `0 < prec` et `prec < emax`. En suivant cette démarche, nous pouvons par exemple exposer un module spécifique pour l'arithmétique flottante IEEE-754 sur 32 bits (le type `float32` des flottants 32 bits se réécrit `binary_float 24 128` dans le formalisme de la bibliothèque Flocq).

Un tel module ne présente plus que des types simples directement traduisible en OCaml sans perte d'information. Notons toutefois que les valeurs de type `binary_float prec emax` sont susceptibles de contenir une preuve d'appartenance au bon format flottant. Ces informations sont également effacées par l'extraction. Pour éviter de produire des valeurs flottantes incorrectes, on rend donc privé le type `t`. Les fonctions produisant des valeurs de type `t` ayant été écrites en Coq, on a l'assurance que toutes les valeurs produites par leur intermédiaire sont bien formées. Des constructeurs intelligents de la forme `A -> t` peuvent également être ajoutés afin de permettre la création de nouvelles valeurs de type `t`. La même démarche s'applique aux modules de domaines flottants.

4.4 Un Type flottant pour les gouverner tous

L'effacement des types dépendants par instanciation systématique n'est qu'une solution partielle au problème de l'extraction de code correcte. En effet, l'utilisation de cette méthode force à passer de familles potentiellement infinies de types (`binary_float` par exemple) à un ensemble restreint de types simples (typiquement `float32` et `float64`). De plus, pour chaque

```
Record float : Type := {
  prec := Z;
  emax := Z;
  Hprec := (0 < prec)%Z;
  Hemax := (prec < emax)%Z;
  value := binary_float prec emax;
}
```

FIGURE 9 – Type des flottants de format arbitraire

instance souhaitée, du code Coq doit être écrit rendant difficile l'ajout de nouvelles instances *a posteriori*. Dans le cadre du développement d'un solveur cela est d'autant plus handicapant que la théorie des flottants telle que décrite dans le standard SMT-LIB [3] manipule des flottants de format arbitraire. Pour respecter au mieux le standard et proposer un solveur le plus générique possible, il est donc souhaitable de pouvoir calculer sur des flottants de n'importe format. Nous proposons pour cela une approche expérimentale en Coq permettant de modéliser tous les formats flottants comme un unique type simple.

On commence par définir un type `float` des flottants de format arbitraire. Ce type est un enregistrement à trois champs : une précision `prec`, un exposant `emax` et un nombre flottant de type `binary_float prec emax`. On demande également que les invariants `0 < f.prec` et `f.prec < f.emax` soient vérifiés pour tout enregistrement `f : float`.

Pour définir les opérations binaires sur ce nouveau type flottant, il est nécessaire de vérifier à priori que les deux opérandes flottants partagent le même format. Cette vérification peut-être faite par le calcul. Bien sûr, si les formats flottants des deux opérandes ne coïncident pas, l'opération ne peut aboutir. Dans un langage de programmation comme OCaml, on souhaiterai utiliser une fonction telle que `assert` pour assurer que les deux formats correspondent et lever une exception sinon. À défaut de pouvoir lever une exception en Coq, nous proposons d'introduire une fonction `Assert` définie comme suit :

```
Program Definition Assert {t : Type} (condition : bool)
  (f : condition = true -> t) (default : t) : t :=
  match condition with
  | true => f _
  | false => default
  end.
```

Pour un booléen `b`, une valeur par défaut `d` et une valeur de retour `r`, le programme `assert b x (fun _ => r) d` renvoit `r` si la condition est vraie ou `d` sinon. Notons que le troisième paramètre de la fonction `assert` est une fonction qui prend en paramètre une preuve que la condition est vraie. Cela permet de se servir de cet hypothèse dans la définition de la valeur `r`.

Cette fonction peut-être extraite vers du code OCaml efficace qui effectue une vérification dynamique de la condition et lève une exception si celle-ci n'est pas satisfaite. On ajoute pour cela une directive d'extraction.

```
Extract Inlined Constant assert => "(fun x f -> assert x; f ())".
```

Cette directive d'extraction ne compromet pas la correction dans le cas où l'assertion est satisfaite. En effet dans ce cas la fonction OCaml a la même sémantique que la définition Coq.

Dans le cas contraire, le mécanisme d'exception d'OCaml interrompt l'exécution du programme. Cet argument de correction suppose que le code extrait ne contient pas de `try_with` permettant de rattraper ces exceptions et que les fonctions Coq extraites ne prennent pas en paramètre des fonctions d'ordre supérieur en argument (celles-ci pourraient éventuellement rattraper une exception provoquée par un `assert false`).

L'usage de la fonction `Assert` rend possible la définition des opérations sur le type `float`. Pour deux arguments flottants donnés, on vérifie la compatibilité de leurs formats avec un simple test d'égalité. Cette condition est vérifiée par un `Assert` et dans le cas où la condition est satisfaite, puisque les types des deux flottants sont égaux, on peut utiliser l'addition flottante. Cette démarche nécessite l'introduction d'opérateurs de conversion de type qui servent simplement à guider le vérificateur de type de Coq. En particulier, étant donnée une preuve que deux formats flottants (p, e) et (p', e') sont égaux et un flottant f de format (p', e'), la fonction `same_format_cast` retourne ce même flottant mais explicitement typé comme étant de format (p, e). Pour définir cette fonction on utilise l'outil `Program` de Coq qui permet de simplifier les éventuelles preuves requises pour établir le bon typage d'un terme.

```
Definition same_format (x y : float) : bool :=
  (prec x =? prec y)%Z && (emax x =? emax y)%Z.

Program Definition same_format_cast {p e p' e' : Z}
  (H : ((p =? p') && (e =? e') = true)%Z)
  (f : binary_float p' e') : binary_float p e := f.
```

FIGURE 10 – Opérateurs de conversion de type

```
Definition add (m : mode) (x y : float) : float :=
    Assert (same_format x y) (fun H => {|
      prec := prec x;
      emax := emax x;
      Hprec := Hprec x;
      Hemax := Hemax x;
      value := @Bplus _ _ (Hprec x) (Hemax x) m (value x)
        (same_format_cast H (value y));
    |}) NaN.
```

FIGURE 11 – Addition sur les flottants de format arbtitraire

5 Conclusion

Grâce à l'utilisation de l'assistant de preuve Coq, nous proposons une implémentation formellement prouvée d'une bibliothèque de manipulation d'intervalles flottants accompagnés de propagateurs de contraintes simples. Cette bibliothèque a été conçue dès son développement pour pouvoir être extraite en OCaml. A l'aide de cette bibliothèque, nous proposons une extension au solveur Colibri2 permettant un support minimal et prouvé correct pour la théorie des nombres flottants telle que définie dans le standard SMT-LIB. Par exemple, le problème SMT suivant est maintenant résolu par Colibri2 (qui déclare le problème insatisfiable) :

```
(set-logic ALL)
(declare-const x Float32)
(assert (fp.leq x ((_ to_fp 8 24) RNE 1.0)))
(assert (fp.geq x ((_ to_fp 8 24) RNE 3.0)))
(check-sat)
```

Il reste des propagateurs à implémenter et à prouver. Par exemple nous pourrions ajouter des opérateurs arithmétiques inverses qui, étant donné le domaine d'appartenance d'une expression arithmétique, permettent d'inférer les domaines d'appartenance des variables apparaissant dans cette expression. La preuve de tels propageteurs sera simplifiée par le choix des définitions et des stratégies de preuves que nous avons présentées.

D'autres domaines comme par exemple des domaines de congruences ou des domaines de bits connus ont prouvé leur efficacité pour la résolution de problèmes de contraintes sur les nombres flottants. Un support pour de nouveaux domaines pourrait également être ajoutés dans la continuation de ces travaux.

Références

[1] The coq proof assistant. https://coq.inria.fr/.

[2] The ocaml programming language. https://ocaml.org/.

[3] The smt-lib floatingpoint theory. http://smtlib.cs.uiowa.edu/theories-FloatingPoint.shtml.

[4] The smt-lib standard. http://smtlib.cs.uiowa.edu/.

[5] Why3. http://why3.lri.fr/.

[6] Ali Ayad and Claude Marché. Multi-prover verification of floating-point programs. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning*, pages 127–141, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[7] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining coq and gappa for certifying floating-point programs. In Jacques Carette, Lucas Dixon, Claudio Sacerdoti Coen, and Stephen M. Watt, editors, *Intelligent Computer Mathematics*, pages 59–74, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[8] Sylvie Boldo and Guillaume Melquiond. Flocq : A Unified Library for Proving Floating-point Algorithms in Coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, pages 243–252, Tübingen, Germany, July 2011.

[9] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[10] Alwyn E. Goodloe, César Muñoz, Florent Kirchner, and Loïc Correnson. Verification of numerical programs : From real numbers to floating point numbers. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods*, pages 441–446, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[11] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified c static analyzer. *SIGPLAN Not.*, 50(1) :247–259, January 2015.

[12] Stéphane Lescuyer and Sylvain Conchon. A Reflexive Formalization of a SAT Solver in Coq. In *TPHOLs 2008 : In Emerging Trends of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2008.

[13] Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. A Constraint Solver based on Abstract Domains. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, edi-

tors, *VMCAI 2013 - 14th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 7737 of *Lecture Notes in Computer Science*, pages 434–454, Rome, Italy, January 2013. Springer-Verlag.