Mémoire d'Habilitation à Diriger des Recherches

présentée à
L'UNIVERSITÉ PARIS 13

spécialité :
INFORMATIQUE

par

Micaela MAYERO
Laboratoire d'Informatique de Paris Nord

Sujet :
Problèmes critiques et preuves formelles

Présentée le 22 novembre 2012 après avis des rapporteurs :

MM.	**Thierry**	**COQUAND**	University of Gothenburg
	John	**HARRISON**	Intel Corporation
Mme	**Marie-Françoise**	**ROY**	Université de Rennes 1

Devant le jury composé de :

MM.	**Pierre-Louis**	**CURIEN**	CNRS - Université Paris 7
	Christophe	**FOUQUERÉ**	Université Paris 13
	Jean-Michel	**MULLER**	CNRS - ENS Lyon
Mmes	**Marie-Françoise**	**ROY**	Université de Rennes 1
	Jacqueline	**VAUZEILLES**	Université Paris 13

Mémoire d'Habilitation à Diriger des Recherches

présentée à

L'UNIVERSITÉ PARIS 13

spécialité :

INFORMATIQUE

par

Micaela MAYERO

Laboratoire d'Informatique de Paris Nord

Sujet :

Problèmes critiques et preuves formelles

Présentée le 22 novembre 2012 après avis des rapporteurs :

MM.	**Thierry**	**COQUAND**	University of Gothenburg
	John	**HARRISON**	Intel Corporation
Mme	**Marie-Françoise**	**ROY**	Université de Rennes 1

Devant le jury composé de :

MM.	**Pierre-Louis**	**CURIEN**	CNRS - Université Paris 7
	Christophe	**FOUQUERÉ**	Université Paris 13
	Jean-Michel	**MULLER**	CNRS - ENS Lyon
Mmes	**Marie-Françoise**	**ROY**	Université de Rennes 1
	Jacqueline	**VAUZEILLES**	Université Paris 13

Remerciements

Je tiens à remercier chaleureusement Thierry Coquand, John Harrison et Marie-françoise Roy d'avoir accepté d'être rapporteurs de cette habilitation à diriger les recherches. Thanks John for having accepted to read this manuscript despite that is written in french.

Je remercie également très sincèrement Pierre-Louis Curien, Christophe Fouqueré, Jean-Michel Muller, Marie-françoise Roy et Jacqueline Vauzeilles d'avoir accepté d'être membre de mon jury, avec une mention particulière à Jean-Michel et Jacqueline pour leurs conseils avisés.

Je tiens à remercier tout particulièrement les collaborateurs avec lesquels j'interagis ou j'ai interagi. La tâche n'est pas toujours simple, suivant les domaines, nous ne parlons pas toujours tous le même langage, certaines de nos motivations diffèrent parfois.

Merci aux membres de l'équipe LCR et plus généralement du LIPN pour m'avoir accueillie et permis toutes ces interactions et collaborations.

J'ai une pensée particulière pour les membres d'Aric de l'ENS de Lyon qui m'ont fait confiance à deux reprises. Merci pour l'accueil plus que chaleureux qui restera gravé dans ma mémoire. Merci pour m'avoir incitée à présenter cette habilitation. Merci.

Merci à mon entourage pour son soutien et pour avoir supporté mes absences hebdomadaires l'an dernier. Besitos grandes a todos.

Abstract : Works presented in this document are, for a large majority of them, at the inter-domain between formal proofs and other areas of computer science (floating point arithmetic, verification, algebraic combinatorics) or mathematics (numerical analysis, number theory, geometry). Two main motivations are at the origin of these works : first, formal proofs are relatively recent in terms of mathematics but also from the point of view of computing and logic. Second, we want to formally prove resolution of critical problems resolution, either in the sense of proving directly programs or proving correction of a problem resolution with respect to its specifications. Moreover, using and improving these techniques is useful for research communities involved in these interactions. For the formal proof community, this can improve development on mandatory theoretical concepts as well as ease of use. For the other domain, it may be of great interest to have a formal proof of a critical problem and to be able to eventually continue on using these reliable techniques.

The first four chapters are devoted to presentation of interaction examples between formal proofs and a critical application domain. We present works concerning numerical analysis, Petri nets, floating point arithmetic and formal computation respectively. These chapters present scientific and historical context linked with the resulting papers. Technical details that are present in the papers are not shown here since main papers are given in annex. The penultimate chapter is specifically dedicated to real numbers in Coq. Last chapter presents a conclusion and prospects.

Keywords : formal proofs, specification, certification, Coq, real numbers, proof of numerical programs, computer algebra, formalisation of mathematics, logic, types theory, verification, numerical analysis, computer arithmetic, refinement of Petri nets, proof automation

Table des matières

Table des matières **7**

Introduction générale **9**

1 Analyse numérique **11**
1.1 Contexte et résumé des résultats 12
1.2 Choix des problèmes 13
1.2.1 Le gradient analytique 13
1.2.2 L'équation des ondes 15
1.2.3 La méthode énergétique versus la méthode de Fourier . . 16
1.3 Choix des approches 16
1.4 Spécifications et preuves 19
1.5 Perspectives 20

2 Réseaux de Petri **21**
2.1 Contexte et résumé des résultats 21
2.2 Choix de formalisations 22
2.3 Difficultés 24
2.4 Perspectives possibles 25

3 Approximation polynomiale rigoureuse **27**
3.1 Contexte et résumé des résultats 27
3.2 La problématique du calcul 28
3.2.1 Les nombres en Coq 28
3.2.2 Réduire et Calculer 30
3.3 Choix techniques 31
3.3.1 Modularité par rapport à la base de polynômes 31
3.3.2 Modularité par rapport aux types de données 32
3.4 Les changements dus aux preuves 32
3.5 Conclusion et Perspectives 33

4 Calcul formel **35**
4.1 Preuve formelle de correction du logiciel SCHUR 35
4.1.1 Description de la problématique 37
4.1.2 Méthodologie 37
4.2 Procédure de décision pour les corps algébriquement clos 39
4.2.1 Description de la problématique 39
4.2.2 Méthodologie 40
4.3 Perspectives 41

5 Les nombres réels dans Coq **43**
5.1 Formalisations des nombres réels en Coq 45
5.1.1 La bibliothèque standard : Reals 45
5.1.2 La bibliothèque intuitionniste : CoRN 49
5.2 Réflexions pour une bibliothèque unique 53
5.2.1 Récapitulatif des points problématiques 53
5.2.2 Quelques options possibles 54
5.3 Perspectives 56

Bilan et perspectives **57**

Bibliographie **61**

Annexes **67**

Introduction générale

Dès les premières années de l'utilisation de l'informatique moderne et des premiers programmes, vers les années 1950, le besoin de sécuriser les actions effectuées par les ordinateurs est apparu. Les premières machines d'aide au calcul telle que, par exemple, la pascaline (Blaise Pascal, 1645) ne faisaient pas encore ressortir ce besoin. En effet, bien que ces machines puissent être considérées comme précurseurs des microprocesseurs (1971), elles n'étaient dédiées qu'aux calculs en permettant d'additionner et de soustraire deux nombres d'une façon directe et de faire des multiplications et des divisions par répétitions. C'est avec la notion de programme informatique, une séquence d'instructions qui spécifie étape par étape les opérations à effectuer pour obtenir un résultat, que le besoin de vérifier le bon déroulement de cette séquence et du résultat est apparu.

Plusieurs méthodes existent, ayant chacune leurs points forts et leurs points faibles. Nous pouvons citer par exemple les méthodes de tests, de *model checking*, de vérification par interprétation abstraite, par réseaux de Petri ou encore les preuves formelles. Nous nous intéressons ici à certaines de ces méthodes formelles et plus particulièrement à l'apport que peuvent avoir ces méthodes dans divers domaines scientifiques tels que le calcul scientifique (analyse numérique), le calcul formel, les mathématiques (analyse, géométrie, algèbre), le calcul numérique (arithmétique à virgule flottante), la certification d'algorithmes, etc.

Les travaux présentés dans ce document se situent, pour une grande majorité, à l'inter-domaine entre les preuves formelles et d'autres domaines de l'informatique (arithmétique à virgule flottante, vérification, combinatoire algébrique) ou des mathématiques (analyse numérique, théorie des nombres, géométrie). Deux motivations principales sont à l'origine de ces travaux : premièrement les preuves formelles sont relativement récentes au regard des mathématiques mais également au regard de l'informatique et de la logique, deuxièmement on veut prouver formellement la résolution de problèmes critiques, que cela soit au sens de prouver directement des programmes ou de prouver des algorithmes ou des propriétés de correction par rapport à des spécifications. L'utilisation de ces méthodes est bénéfique pour les communautés concernés par ces interactions. Pour celle des preuves formelles, cela permet d'accroître leur développement, aussi bien du point de vue des concepts théoriques nécessaires que pour la facilité d'utilisation. Pour celle du domaine considéré, de se prévaloir d'une preuve formelle de son problème critique et d'envisager éventuellement de poursuivre l'utilisation de telles méthodes, sûres.

Une des difficultés intrinsèques à certaines spécifications et preuves formelles est le besoin de nombres réels et flottants. Les méthodes formelles manipulent plutôt des nombres entiers, ou plus généralement des structures discrètes. Le développement de librairies spécifiant les nombres réels et flottants est récent.

Une partie des travaux présentés dans ce mémoire concerne particulièrement cette problématique.

Le mémoire s'articule de la façon suivante : Les chapitres 1, 2, 3 et 4 seront dédiés à la présentation d'exemples d'interaction entre les preuves formelles et un domaine critique d'application. Nous présenterons respectivement, les travaux concernant l'analyse numérique, les réseaux de Petri, l'arithmétique flottante, le calcul formel. Ces chapitres, liés aux articles écrits suite à ces travaux, présenteront les contextes scientifiques, historiques et des discussions sur les problématiques des preuves en interaction avec d'autres domaines, sans toutefois rentrer dans les détails techniques se trouvant dans les publications. Un article de référence pour chacun de ces quatre chapitres est mis en annexe. Le chapitre 5 est tout spécifiquement dédié à la problématique des nombres réels dans Coq. Le dernier chapitre présentera un bilan et des perspectives. Sont présentés, dans chacun de ces chapitres, des sections de rappels. Il s'agit généralement de rappels concernant les notions à connaître pour comprendre les tenants et les aboutissants des discussions et problématiques décrites dans ce mémoire. Le lecteur familier avec les domaines considérés pourra se passer des rappels. Ces sections de rappels commencent par **Rappels** et se terminent par un ▮.

Chapitre 1

Analyse numérique

[Ce chapitre se réfère aux articles [BCF+10, BCF+12], dont le [BCF+12] se trouve en annexe.]

Les travaux présentés dans ce chapitre résultent de la continuité de ma motivation première datant des années 1995-1997, qui précédait déjà ma thèse et juste postérieure à mon cursus d'analyse numérique. Les numériciens qui étudient des problèmes tels que les écoulements de fluide, la propagation des ondes, la prospection, etc., ont, en plus des résolutions mathématiques de ces problèmes (équations dont les solutions exactes ne sont pas connues), également à implanter des algorithmes qui tentent de résoudre ces équations. Ces programmes peuvent s'avérer critiques, par exemple lorsqu'il s'agit de la résistance d'un pont ou de l'écoulement de l'air autour d'une fusée... Ces programmes présentent deux difficultés majeures : d'une part, une programmation correcte (et efficace) est nécessaire et d'autre part, ces programmes étant des programmes numériques, ils utilisent intensivement les nombres à virgules flottantes, ce qui peut engendrer des erreurs de calculs et d'approximations conséquentes. Depuis environ 15 ans, je soutiens donc l'idée que les preuves formelles peuvent/doivent apporter une aide afin de garantir la sûreté de ces solutions (tant au niveau mathématique que informatique). Les premiers développements nécessaires et les premières expérimentations sont issues de ma thèse (2001) avec le développement de la bibliothèque standard de Coq des nombres réels et la preuve de correction (ou non) d'un outil de différentiation automatique. Je n'exposerai pas dans ce chapitre ces travaux. Au regard des objectifs à terme que représente une telle interaction, ce travail de thèse en est les balbutiements. La suite naturelle, de mon point de vue, a donc été de poursuivre dans la direction entamée. Ce chapitre est dédié à la présentation des résultats les plus récents. Nous mettrons principalement l'accent sur les points non discutés dans les articles.

La section 1.1 présente le contexte et un résumé des résultats depuis 2005. La section 1.2 tentera d'expliquer une des difficultés qu'il est nécessaire de surmonter, celle du choix des problèmes à étudier. En section 1.3, nous discuterons des différentes approches possibles pour aborder la preuve formelle de tels problèmes. Nous détaillerons ensuite en section 1.4 certaines spécifications et preuves choisies, puis en section 1.5 nous aborderons les perspectives concernant cette thématique.

1.1 Contexte et résumé des résultats

Le travail décrit ici a commencé dans le cadre du projet CerPAN [1] (Certification de Programmes d'Analyse Numérique), dont j'ai été porteuse, ANR 2005, commun aux laboratoires LRI (J.-C. Filliâtre, S. Boldo), INRIA-Rocquencourt (F. Clément) et CNAM (D. Delahaye), ainsi que du projet FOST [2] (Formal PrOofs about Scientific compuTation), porté par S. Boldo, ANR 2009, la suite de CerPAN.

Dans ces projets, nous nous sommes intéressés au développement et à la mise en application des méthodes permettant de démontrer formellement la correction de programmes issus du domaine de l'analyse numérique. Cela concerne plus particulièrement des programmes apparaissant de manière récurrente dans la résolution de problèmes critiques. Beaucoup de programmes critiques sont issus de ce domaine, mais les travaux traitant directement des applications des méthodes formelles aux programmes d'analyse numérique sont rares. La principale raison est l'utilisation intensive des nombres à virgule flottante, en guise de nombres réels, dans les programmes numériques, alors que les méthodes formelles manipulent plutôt des nombres entiers, ou plus généralement des structures discrètes. Ces projets ont exploré deux méthodes, toutes deux étant vouées à être mises en pratique sur une étude de cas : l'équation des ondes en une dimension. Je me suis surtout intéressée à la seconde méthode.

1. L'une consiste à utiliser des outils de vérification de programmes tels que Why et Caduceus [3] puis Frama-C [4] (dédié aux programmes C) développés au LRI. Il s'agit de travailler sur un programme déjà existant, qui va être annoté suivant les propriétés à préserver. Ces annotations engendrent alors des obligations de preuve pour divers systèmes de preuve formelle (Coq, PVS, Isabelle/HOL, HOL 4, HOL Light, Mizar,...). Ces outils de vérification de programmes ne traitaient pas les nombres flottants. J'ai participé aux phases de décision concernant l'extension de ces outils aux nombres flottants sous forme d'un langage de spécification pour les propriétés relatives aux nombres à virgule flottante. Ces ajouts [BF07] sont distribués avec le système.
2. L'autre méthode consiste à effectuer la formalisation et la preuve du problème correspondant au programme ou à l'algorithme étudié (preuve de l'erreur de méthode), puis à utiliser, éventuellement, le mécanisme d'extraction du système Coq [CDT] afin d'obtenir automatiquement un programme prouvé correct. Ce principe d'extraction [Let03] ne traite pas non plus la famille des nombres réels. J'ai commencé par décider d'extraire vers des réels exacts et non vers des nombres flottants et je recherche actuellement un modèle adéquat de réels exacts (réaliste d'un point de vue de l'implantation, efficace, ...) pour étendre l'extraction de Coq. D'autre part, cette spécification et preuve de l'erreur de méthode, réalisée en collaboration avec 4 autres membres de ces projets, est également nécessaire pour la preuve de l'erreur de calcul issue de la phase précédente.

Des avancées notables ont eu lieu depuis le début du projet : un support

1. http://www-lipn.univ-paris13.fr/CerPAN/
2. http://fost.saclay.inria.fr/
3. http://caduceus.lri.fr/
4. http://frama-c.com/

de l'arithmétique à virgule flottante a été intégré aux outils de preuve de programmes C Caduceus et Frama-C. Il a ensuite été enrichi pour interpréter ces nouvelles spécifications et les transformer en obligations de preuves Coq ; la preuve Coq d'une majoration quadratique de l'expression analytique de l'erreur de calcul est achevée [Bol09] ; une interface entre Gappa (automatisation de preuves d'arithmétique d'intervalle) et Coq a été développée [BFM09] ; la preuve Coq de l'erreur de méthode est terminée [BCF$^+$10], ce qui clôt l'étude de cas sur l'équation des ondes en une dimension (1D). L'ensemble du travail faisant le lien entre la preuve de l'erreur de calcul [Bol09] et la preuve de l'erreur de méthode [BCF$^+$10] a été publié dans *Journal of Automated Reasoning* [BCF$^+$12].

1.2 Choix des problèmes

Un des objectifs de cette collaboration entre chercheurs numériciens (analyse numérique, domaine des mathématiques appliquées) et chercheurs des méthodes formelles (en particulier spécification, preuves formelles et preuves de programmes) est d'apporter de nouvelles techniques de validation des programmes numériques critiques. L'étape préliminaire consiste à étudier un cas à la fois intéressant et abordable. Une importante difficulté réside dans le mot "abordable" et cette section est dédiée à ce problème.

Quelle étude de cas ? Dans une interaction entre deux domaines scientifiquement éloignés tels que l'analyse numérique et la logique, l'exemple de travail doit être intéressant et constructeur pour les deux domaines. Il ne faut donc pas qu'il soit trop simple pour l'un ni trop compliqué pour l'autre. Dans cette démarche, notre première idée était d'étudier nos techniques de preuves sur un problème de gradient faisant appel à la résolution d'équations aux dérivées partielles (EDP).

1.2.1 Le gradient analytique

Rappels sur le problème inverse. Un *modèle direct* est une fonction F telle que :

$$F : x \in \mathbb{R}^n_x \mapsto y \in \mathbb{R}^n_y$$

où x est un paramètre et y correspond aux mesures calculées.

Le *problème inverse* consiste alors à calculer F^{-1} qui peut se faire grâce à la formulation aux moindres carrés :

$$F^{-1}(\overline{y}) = argmin_x J(x)$$

où $\overline{y}$ correspond aux mesures données et $J(x) = \frac{1}{2}\|F(x) - \overline{y}\|^2$. ∎

Lorsque les paramètres d'une équation aux dérivées partielles ne sont pas bien connus, ils peuvent être estimés à partir de certaines mesures de la solution. Le problème d'estimation de paramètres est formulé habituellement comme un problème de minimisation d'une fonction moindres carrés. La technique de l'état adjoint est un outil efficace pour calculer le gradient analytique de cette

fonction qui peut ensuite servir de données à un code d'optimisation. La décomposition en valeurs singulières est un outil puissant pour l'analyse de sensitivité déterministe. Elle permet de quantifier le nombre de paramètres qui peuvent être estimés à partir des mesures de la solution. Ceci peut permettre de choisir une paramétrisation pour les coefficients recherchés, et même de concevoir les expériences de mesure. Ces techniques sont, par exemple, utilisées en sismique afin de parvenir à sonder les profondeurs de la terre par l'envoi d'ondes [FG01, FM04]. Dans ce cas précis, il s'agit de minimiser la fonction $J(P) = \frac{1}{2}\|d - F(P)\|^2$ où d représente la mesure expérimentale et $F(P)$ la fonction théorique. Faire cette minimisation revient à trouver les valeurs pour lesquelles $\vec{\bigtriangledown}(J(P)) = 0$, où $\vec{\bigtriangledown}$ est le gradient.

Le programme qui calcule le gradient issu de ce problème soulève plusieurs difficultés. La première est l'utilisation inévitable des nombres flottants. La deuxième provient du fait que ce programme est un programme impératif. La troisième est due à la nécessité de préserver une certaine efficacité des calculs. Ces trois points ne posent pas de problème en soi pour ce qui est de l'algorithme ni de la programmation, mais les difficultés surviennent lorsqu'il est question de spécifier et/ou prouver formellement le programme. En effet, les outils de preuve formelle, même s'ils ont progressé très significativement dans ce domaine ces dernières années, sont encore en développement pour ce qui est des nombres réels et flottants. Par ailleurs, la plupart de ces outils sont mieux adaptés à traiter des programmes fonctionnels et ceux qui sont dédiés à la preuve de programmes impératifs sont également relativement naissants (si on fait abstraction de la méthode B [Abr96], qui tend à être inutilisée tant du point de vue industriel que académique). Concernant l'efficacité, les techniques qui consistent, à partir d'une preuve, à extraire un programme produisent généralement des programmes plutôt inefficaces (il est, en effet, difficile de contrôler l'efficacité au cours de la preuve). Les réels exacts, une alternative séduisante aux nombres flottants, ne sont absolument pas compétitifs en terme d'efficacité face aux nombres flottants. Ainsi, les trois difficultés citées ci-dessus sont récurrentes dans tous les problèmes numériques que l'on souhaite traiter par ordinateur.

Cette étude de cas semblait d'une taille et d'une difficulté raisonnable compte tenue des avancées actuelles des outils d'aide à la preuve pour traiter les problèmes d'analyse réelle.

Mais, cet exemple, déjà considéré comme trivial par les numériciens, s'est révélé trop ambitieux en tant que première étude de cas, principalement en raison de la difficulté posée par la notion de dérivée, spécifiée dans le système Coq comme la limite du taux d'accroissement. Deux solutions sont possibles dans un tel cas : soit compléter la bibliothèque de l'outil d'aide à la preuve par les notions mathématiques manquantes, soit envisager une adaptation de la difficulté des exemples abordés. La première solution présente l'avantage de compléter l'outil, ce qui pourra servir aux autres utilisateurs, mais a l'inconvénient de demander un temps de développement important sans grand intérêt scientifique. Nous avons donc fait le second choix, afin de continuer dans un domaine purement d'analyse numérique avec des exemples concrets issus de ce domaine.

1.2.2 L'équation des ondes

Rappels sur l'équation des ondes en une dimension et sur un des schémas numérique associé possible (voir [BCF+10] ou [BCF+12] pour des rappels plus détaillés).

- *Équation des ondes :* soit $u : \mathbb{R}^2 \to \mathbb{R}$ suffisamment régulière telle que :

$$\frac{\partial^2 u(x,t)}{\partial t^2} - c^2 \frac{\partial^2 u(x,t)}{\partial x^2} = s(x,t)$$

  où $s(x,t)$ est le terme source tel que $s(x,0) = 0$, $u_0(x)$ et $u_1(x)$ sont les conditions initiales représentant respectivement la position initiale et la vitesse initiale.
- *Schéma numérique à trois points :* (voir Figure 1.1) le but est d'approcher $u_j^k \approx u(j\Delta x, k\Delta t)$.

$$\frac{u_j^k - 2u_j^{k-1} + u_j^{k-2}}{\Delta t^2} - c^2 \frac{u_{j+1}^{k-1} - 2u_j^{k-1} + u_{j-1}^{k-1}}{\Delta x^2} = s_j^{k-1}$$

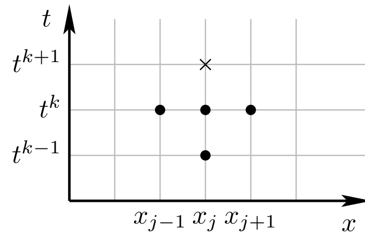


FIGURE 1.1 – Le schéma à 3 points : u_j^{k+1} ($\times$) dépend de u_{j-1}^k, u_j^k, u_{j+1}^k et u_j^{k-1} ($\bullet$).

∎

Dans le cadre de l'équation des ondes en une dimension, nous avons pu prouver l'erreur due aux flottants (travaux de Sylvie Boldo principalement) ainsi que ce qui s'appelle l'erreur de méthode, c'est-à-dire la stabilité du schéma tel qu'il est codé (en Fortran) et son ordre d'approximation. En d'autres termes, cela correspond également à prouver l'erreur faite par l'approximation de la solution exacte issue du schéma numérique utilisé en montrant l'erreur de convergence. Mathématiquement parlant :

Nous regardons si u et u_j^k sont proches lorsque $(\Delta x, \Delta t) \to 0$. Soit $e_j^k \stackrel{\text{def}}{=} \bar{u}_j^k - u_j^k$, l'erreur de convergence où $\bar{u}_j^k$ représente la valeur de u au point (j,k) de la grille. Nous cherchons alors à borner $\left\| e_h^{k_{\Delta t}(t)} \right\|_{\Delta x}$, la moyenne de l'erreur de convergence sur tous les points de la grille à un temps donné $k_{\Delta t}(t) = \left\lfloor \frac{t}{\Delta t} \right\rfloor \Delta t$.

Nous voulons prouver que :

$$\left\| e_h^{k_{\Delta t}(t)} \right\|_{\Delta x} = O_{[0,t_{\max}]}(\Delta x^2 + \Delta t^2)$$

Plusieurs méthodes existent pour cela, nous en discuterons dans la section1.2.3 suivante. Mais nous pouvons voir d'ores et déjà que les notions

de "grand O", dérivées partielles, normes, sommation, borne inférieure, limite, majoration, etc. vont intervenir. Ces notions tellement élémentaires pour les mathématiciens, au point qu'aucune d'entre elle n'est définie dans aucun article, peuvent soulever de sérieux problèmes lorsqu'il s'agit de les formaliser dans un outil d'aide à la preuve tel que Coq. Nous verrons en section 1.4 le cas du "grand O".

L'équation des ondes est l'équation générale qui décrit la propagation d'une onde, on la schématise par une impulsion qui se propage à travers une corde d'un bout à l'autre. La cas de la dimension 1 a été étudié et est parfaitement connu des numériciens. Il ne présente pas grand intérêt dans la pratique ce qui rend cet exemple moins attrayant que celui du gradient. Néanmoins, il s'est avéré être une très bonne étude de cas, qui a révélé un grand nombre de problèmes au niveau formel, souvent ignorés dans les preuves "papier-crayon" mais dont la spécification est obligatoire dans le système formel.

1.2.3 La méthode énergétique versus la méthode de Fourier

La preuve usuelle (faite par les mathématiciens) pour prouver la convergence du schéma numérique utilise les transformées de Fourier. Une autre méthode utilise une technique dite énergétique (décrite dans [BCF+12]). Passer par les transformées de Fourier est un cheminement naturel et simple, mathématiquement parlant. Par contre, pour un système de preuves formelles, cela requiert l'existence d'une formalisation des transformées de Fourier et donc bien sûr de toutes les notions de base nécessaires pour cela (intégrale de Lebesgue par exemple). Dans la plupart des systèmes de preuves, ces formalisations et preuves n'existent pas (les nombres réels n'ont guère qu'une dizaine d'années en Coq). Pour la même raison qui nous avait conduit à simplifier notre première étude de cas, nous avons ici choisi la seconde méthode (énergétique).

Les développements de base sont indispensables à une "véritable" utilisation des systèmes de preuves formelles dans le cadre d'études "réalistes". Ces bibliothèques augmentent grâce aux travaux comme celui-ci, mais la progression est lente car les chercheurs se concentrent souvent sur une problème difficile et n'ont pas la possibilité de consacrer un temps important à la formalisation des mathématiques de base. Cette situation est problématique et représente à mon sens un réel handicap pour l'utilisation de ces outils. Il me semble néanmoins important de préciser que ces "mathématiques de base" soulèvent assez souvent des problèmes théoriques logiques liés à la théorie sous-jacente aux prouveurs eux-mêmes. Dans le chapitre 5 nous décrirons un des exemples les plus frappant qui étaye cette remarque : celui de la formalisation des nombres réels.

1.3 Choix des approches

Après le choix d'une étude de cas appropriée, comme expliqué en section 1.2, viennent les choix concernant les techniques pour mener à bien les validations souhaitées. Rien que dans le domaine des preuves formelles stricto sensu, les techniques possibles sont nombreuses et les orientations prises peuvent être di-

verses et conduire à des objectifs relativement différents.

La méthodologie consiste à commencer par se demander quelle genre de preuve est nécessaige : une preuve de correction par rapport à la spécification, une preuve du programme lui-même, une preuve des erreurs occasionnées par la méthode employée, une preuve sur des erreurs de calcul, etc. ? Suivant les réponses à ces questions, les trois méthodes principales sont :

1. spécifier puis formaliser puis prouver
2. spécifier puis formaliser puis prouver en vue d'en extraire un programme
3. prouver le programme

Il est également possible d'utiliser ces méthodes de manière complémentaire pour certaines parties du problème, ce que nous avons fait dans les projets CerPAN et FOST.

Dans la suite de ce chapitre, nous nous concentrerons sur trois problématiques issues des questions précédentes.

Utiliser l'extraction ?

Rappels informels sur l'extraction de Coq : l'extraction de Coq [Let03] est un mécanisme permettant, à partir d'un λ-terme de preuve intuitionniste d'obtenir un programme (en Ocaml [5] par exemple), ceci grâce à l'isomorphisme dit de Curry-Howard. Le programme ainsi obtenu, issu de la preuve Coq, est considéré comme sûr (indépendamment du principe d'extraction). ▮

Cette méthodologie demande au préalable la spécification, la formalisation et les preuves intuitionnistes en Coq des propriétés à montrer à partir desquelles le programme va être obtenu. Il est à noter, et c'est un point important dans le domaine de l'analyse numérique, que le programme obtenu n'est pas spécialement efficace (il peut l'être comme ne pas l'être) dans le sens où aucune optimisation n'est faite. Modifier le programme obtenu afin d'y faire quelques changements, même s'ils peuvent sembler mineurs, va à l'encontre de la méthode de programmation par extraction, les modifications manuelles ôtant l'aspect "sûr" provenant du λ-terme de preuve.

Un autre aspect important, qui va dicter nos choix, réside dans le mot "intuitionniste" (ou constructif). Le fait que ce qui est extractible actuellement soit le contenu calculatoire des preuves (pas les propositions logiques) est particulièrement important dans notre contexte de preuves sur l'analyse numérique utilisant des nombres réels. La partie suivante va permettre d'expliquer pourquoi nous avons jusqu'à présent laissé de côté une éventuelle extraction.

Quels réels ?

Cette question se pose dès que l'on formalise des propriétés d'analyse. L'analyse classique, celle qui est utilisée en analyse numérique par les numériciens, est basée sur des nombres réels classiques, à savoir, par exemple, les nombres construits comme limites de suites de Cauchy de nombres rationnels. Mais nous avons vu dans la paragraphe précédent que les théories classiques posent de sérieux problèmes pour l'extraction actuelle de Coq. Des travaux sur la réalisabilité

5. http://caml.inria.fr/ocaml/index.en.html

et l'extraction classique sont en cours [Miq07, Her12]. Nous aborderons les différentes formalisations possibles, les différences entre les nombres réels classiques et intuitionnistes dans le chapitre 5. Néanmoins, il nous semble également important de noter ici que les nombres réels de la bibliothèque standard [May01, CDT] actuellement distribuée avec le système Coq, sont une axiomatisation classique. En plus d'être classique, cela signifie donc qu'au niveau de l'extraction, il faut commencer par réaliser les axiomes. Dans l'optique d'étendre le principe d'extraction à la famille des nombres réels, nous avons réfléchi à la possibilité de réaliser les axiomes par une implantation de nombres réels exacts [O'C08, KS11]. La solution qui consiste à formaliser nos problèmes numériques (équation des ondes acoustique) avec des réels intuitionnistes (bibliothèque CoRN [GN00] dont nous parlerons également au chapitre 5) n'a pas été retenue pour deux raisons :

- les preuves issues du domaine de l'analyse numérique sont classiques, il serait possible de les faire constructivement en s'appuyant sur l'analyse constructive de E.Bishop [Bis67] par exemple ; cela serait un autre projet, un projet à très long terme, tels le projet sur le *Fundamental Theorem of Algebra* [GWZ00], la conjecture de Kepler [6], la formalisation et la preuve du théorème de Feit-Thompson [7] ou les *Univalent Foundations of Mathematics* [8].
- une des caractéristiques de l'axiomatisation classique de la bibliothèque standard des réels de Coq est l'utilisation de l'égalité de Leibniz. L'égalité de Leibniz n'est pas utilisable pour des réels constructifs, c'est l'*apartness* (voir chapitre 5, section 5.1) qui la remplace. L'égalité de Leibniz est celle utilisée par les numériciens. Lorsque nous avons commencé les premiers travaux liés à l'analyse numérique (en 1998 lors de ma thèse de doctorat), il n'y avait pas non plus d'outil sur les setoïdes (un type muni juste d'une relation d'équivalence), qui commencent à devenir utilisables pour de la réécriture plus générique (nous en parlerons également dans le chapitre 5).

Les avancées théoriques sur la théorie des types et les nouvelles implantations de ses dix dernières années ouvrent des perspectives importantes concernant les nombres réels et rendent parfois inconfortables ou dépassés les choix effectués il y a 5 ou 6 ans. Néanmoins, pour nos travaux focalisés sur l'interaction avec l'analyse numérique, nous faisons le choix de continuer à la fois en nous accommodant des anciens choix et à la fois en essayant de les faire évoluer lorsque cela est possible.

Gestion des erreurs de flottants et de l'erreur de méthode

Nous avons vu que deux principaux problèmes peuvent se poser lors de l'implantation d'un algorithme codant un schéma numérique. Le schéma peut être faux et les erreurs dues aux nombres flottants des machines peuvent modifier de manière importantes les résultats et les faire diverger. Il faut donc montrer à la fois l'erreur de méthode et l'erreur de flottant. Il est aisé de voir que les problèmes sont de nature différente. De ce fait, les manières de les prouver peuvent être différentes puis reliées (ce que nous appellerons erreur totale). En ce qui concerne l'erreur de flottant, il s'agit d'un genre d'erreur lié à l'arithmétique des ordinateurs, ce qui oriente vers une méthodologie de preuve de programme

6. http://code.google.com/p/flyspeck/
7. http://ssr2.msr-inria.inria.fr/~jenkins/current/progress.html
8. http://www.math.ias.edu/sp/univalent

plutôt que de faire une formalisation puis une preuve directe. En ce qui concerne l'erreur de méthode, elle peut être prouvée également sur le programme directement ou seulement par spécification directement dans Coq. Étant donné que des annotations sur le programme sont nécessaires pour l'erreur de flottants, il est judicieux d'introduire des annotations pour l'erreur de méthode, ce qui donnera une preuve du programme sur les deux points critiques. Le seul problème réside alors dans le traitement des obligations de preuve. Dans l'exemple de l'équation des ondes, il a été nécessaire de formaliser et prouver entièrement en Coq les propriétés de consistance, de stabilité et de convergence afin de pouvoir utiliser ces propriétés pour montrer les obligations de preuve issues de l'erreur de méthode.

1.4 Spécifications et preuves

Nous avons choisi de détailler ici un exemple illustrant le genre de problèmes (mathématiques ou théoriques) que les preuves formelles peuvent mettre en exergue. Il s'agit de la définition utilisée pour le "grand O".

Lorsque l'on utilise l'égalité $a = O(b)$, on suppose que a et b sont deux expressions définies sur le même domaine et que leur interprétation logique à l'aide des quantificateurs est naturelle. Dans le cas qui nous occupe, la situation est un peu plus complexe. Soit

$$f(\mathbf{x}, \boldsymbol{\Delta}\mathbf{x}) = O(g(\boldsymbol{\Delta}\mathbf{x}))$$

lorsque $\|\boldsymbol{\Delta}\mathbf{x}\|$ tend vers 0. Si on suppose que l'égalité est vraie pour chaque $\mathbf{x} \in \mathbb{R}^2$, son interprétation est

$$\forall \mathbf{x}, \exists \alpha > 0, \exists C > 0, \forall \boldsymbol{\Delta}\mathbf{x}, \quad \|\boldsymbol{\Delta}\mathbf{x}\| \leq \alpha \Rightarrow |f(\mathbf{x}, \boldsymbol{\Delta}\mathbf{x})| \leq C \cdot |g(\boldsymbol{\Delta}\mathbf{x})|,$$

ce qui signifie que les constantes α et C sont en fait des fonctions de $\mathbf{x}$. Une telle interprétation est inutile puisque la borne inférieure de α pourrait très bien être 0 et la borne supérieure de C, $+\infty$.

Une interprétation appropriée demande l'introduction d'un "grand O uniforme" par rapport à la variable $\mathbf{x}$:

$$\exists \alpha > 0,\ \exists C > 0,\ \forall \mathbf{x} \in \Omega_{\mathbf{x}},\ \forall \boldsymbol{\Delta}\mathbf{x} \in \Omega_{\boldsymbol{\Delta}\mathbf{x}},$$
$$\|\boldsymbol{\Delta}\mathbf{x}\| \leq \alpha \Rightarrow |f(\mathbf{x}, \boldsymbol{\Delta}\mathbf{x})| \leq C \cdot |g(\boldsymbol{\Delta}\mathbf{x})|. \quad (1.1)$$

Il est alors possible de faire apparaître la dépendant en $\Omega_{\mathbf{x}}$ et en $\Omega_{\boldsymbol{\Delta}\mathbf{x}}$ en écrivant les égalités de la manière suivante :

$$f(\mathbf{x}, \boldsymbol{\Delta}\mathbf{x}) = O_{\Omega_{\mathbf{x}}, \Omega_{\boldsymbol{\Delta}\mathbf{x}}}(g(\boldsymbol{\Delta}\mathbf{x})).$$

Cette définition nous permet de définir précisément la notion de fonction "suffisamment régulière". En effet, la convergence du schéma numérique requiert que la solution de l'équation soit "suffisamment régulière". Nous introduisons deux opérateurs, qui, pour une fonction réelle f du plan, renvoie les valeurs $\frac{\partial f}{\partial x}$ et $\frac{\partial f}{\partial t}$ en ces points. Ces opérateurs permettent de définir les polynômes de Taylor usuels de f d'ordre n :

$$\mathrm{TP}_n(f, \mathbf{x}) \stackrel{\text{def}}{=} (\Delta x, \Delta t) \mapsto \sum_{p=0}^{n} \frac{1}{p!} \left(\sum_{m=0}^{p} \binom{p}{m} \cdot \frac{\partial^p f}{\partial x^m \partial t^{p-m}}(\mathbf{x}) \cdot \Delta x^m \cdot \Delta t^{p-m} \right).$$

Soit $\Omega_{\mathbf{x}} \subset \mathbb{R}^2$. Nous avons alors que le polynôme de Taylor précédent est une approximation uniforme de f d'ordre n sur $\Omega_{\mathbf{x}}$ avec le "grand O uniforme" :

$$f(\mathbf{x} + \mathbf{\Delta x}) - \mathrm{TP}_n(f, \mathbf{x})(\mathbf{\Delta x}) = O_{\Omega_{\mathbf{x}}, \mathbb{R}^2}\left(\|\mathbf{\Delta x}\|^{n+1}\right).$$

La fonction f est alors dite *suffisamment régulière d'ordre n uniformément sur $\Omega_{\mathbf{x}}$* lorsque tous les polynômes de Taylor d'ordre inférieur à n sont des approximations uniformes de f sur $\Omega_{\mathbf{x}}$.

Cette notion de "grand O uniforme" a représenté un verrou assez important pour notre preuve formelle. Les supports issus du domaine de l'analyse numérique ne précisent pas la notion exacte de "grand O" utilisée et, de plus, font une utilisation assez peu rigoureuse des quantificateurs logiques. Bien évidemment, les outils de preuve formelle obligent à clarifier ces points.

1.5 Perspectives

Nous pensons utiliser les expérimentations et formalisations des précédents travaux de ces 7 dernières années afin de poursuivre sur la preuve formelle de programmes d'éléments finis. L'idée est d'augmenter la sûreté de logiciels critiques par des méthodes difficiles et coûteuses en temps de développement, mais qui apportent une forte garantie. Nous continuerons d'utiliser des méthodes formelles qui consistent à faire vérifier nos preuves par l'ordinateur : il vérifie chaque étape élémentaire de chaque preuve pour en assurer la correction.

Il ne s'agit plus d'un exemple jouet servant à rôder nos techniques, mais de vrais programmes sur les éléments finis qui sont eux-mêmes à la pointe de la recherche en analyse numérique. Cela constitue un gros investissement car la méthode des éléments finis, même si elle est bien maîtrisée mathématiquement, implique un gros travail de formalisation et de preuves formelles pour être facilement utilisable sur des exemples. Après l'étude d'un cas simple (Dirichlet homogène pour l'équation de Laplace-Poisson posée sur le carré unité avec maillage régulier), nous nous attaquerons à un cas réaliste avec application à la modélisation d'écoulements dans un milieu poreux fracturé. Il est en effet important de montrer la possibilité d'utiliser ces méthodes sur un exemple de niveau recherche en analyse numérique. Enfin, les logiciens ne souhaiteront probablement pas prouver les programmes de calcul scientifique dans les prochaines décennies et il faut donc donner aux numériciens des outils pour les prouver eux-mêmes. Nous visons donc la création d'une boîte à outils utilisable par des non-spécialistes et qui donne des garanties fortes de correction des programmes.

Cette suite reste dans l'interdisciplinarité (logique/analyse numérique) et est ambitieuse car le but est de prouver formellement un vrai algorithme à la pointe de l'analyse numérique en tenant compte à la fois de sa preuve mathématique complexe et des erreurs d'arrondis.

Chapitre 2

Réseaux de Petri

[Ce chapitre se réfère aux articles [CMP08, CMP09, CMP10], dont le [CMP10] se trouve en annexe.]

Les travaux décrits dans ce chapitre résultent d'une interaction entre les domaines de la vérification et des preuves formelles. Ces deux domaines ont souvent un but affiché commun, celui de rendre sûr certaines applications, algorithmes, etc. Si nous y ajoutons une complémentarité certaine, il semble naturel de tenter de les faire interagir. Dans les cas qui nous occupent, la complémentarité se situe principalement au niveau de l'automatisation du traitement des problèmes et de leur finitude. Si la plupart des méthodes de vérification traitent les problèmes finis avec succès, les preuves formelles, bien que plus fastidieuses à utiliser, savent traiter les problèmes infinis. Nous avons choisi de nous intéresser à l'étude de raffinements de réseaux de Petri.

La section 2.1 présente le contexte et un résumé des résultats. En section 2.2, nous discutons des différentes approches possibles pour aborder la preuve formelle de tels problèmes. La section 2.3 détaille quelques problèmes rencontrés, puis en section 2.4 nous abordons des perspectives possibles concernant cette thématique.

2.1 Contexte et résumé des résultats

Ce travail a été réalisé en collaboration avec des membres de l'axe "Spécification et vérification" de l'équipe LCR du LIPN.

De nos jours, les systèmes à modéliser et à analyser sont de taille de plus en plus grande. Pour s'affranchir des problèmes inhérents à l'explosion du nombre d'états, une spécification est souvent développée pas à pas (ce qui correspond à une démarche classique de développement de génie logiciel) : tout d'abord, un modèle abstrait est défini. Une fois que ses propriétés sont vérifiées, une étape de raffinement, qui introduit un niveau de détail supplémentaire, peut être effectuée. Un tel ajout peut préciser la description de l'actuel fonctionnement d'une partie du système, ou introduire une partie supplémentaire. Le modèle ainsi raffiné est ensuite vérifié. Une nouvelle étape de raffinement peut alors

être appliquée, et ce jusqu'à atteindre un niveau suffisant de description.

Une approche incrémentale par raffinements à l'aide de réseaux de Petri est souvent utilisée pour la spécification de systèmes complexes. Un réseau de Petri [Pet66] (pouvant être représenté comme un graphe biparti orienté) est composé de places, de transitions, d'arcs et de jetons. C'est un moyen de représenter divers systèmes travaillant sur des variables discrètes. Trois types de raffinement ont été introduits par Lakos et Lewis [LL01], pour les réseaux de Petri de haut niveau (i.e. coloré et hiérarchique) : le raffinement de sous-réseau, le raffinement de nœud et le raffinement de type. Outre les avantages du raffinement pour la spécification de systèmes, ils permettent de faciliter la vérification de propriétés. En effet, d'une part certaines propriétés sont préservées par le raffinement, et d'autre part, l'analyse du modèle abstrait permet de guider celle du modèle concret, économisant ainsi en temps de calcul et espace mémoire lors de la construction de graphes d'états.

Le processus de raffinement de réseaux est complètement manuel et la vérification des contraintes peut être assez complexe. Par conséquent, nous avons prouvé en Coq qu'un réseau est effectivement un raffinement d'un autre réseau (abstrait). Nous proposons donc une formalisation en Coq des définitions de raffinements ainsi que des réseaux auxquels ils s'appliquent. Ceci a tout d'abord été développé pour les raffinements de sous-réseaux et de nœuds des réseaux places/transitions, et appliqué à quelques exemples. Ce travail a été étendu, dans le cadre d'un stage de Master 2, aux réseaux colorés.

Les premières expériences [CMP08] nous ont permis de déterminer la spécification Coq la mieux adaptée (suite à 3 formalisations différentes) [CMP09, CMP10] ainsi que de déterminer l'outil d'édition de réseaux de Petri adapté pour une interface avec Coq.

2.2 Choix de formalisations

Rappels sur les réseaux de Petri places/transitions.

Les réseaux de Petri considérés ici sont définis dans [Lew02]. Nous nous restreindrons ici aux réseaux de Petri places/transitions.

Un *réseau de Petri* est un quadruplet $\mathcal{N} = \langle P, T, W, M_0 \rangle$ où :

1. P est l'ensemble des *places* ;
2. T est l'ensemble des *transitions*, tel que $P \cap T = \emptyset$;
3. $W : (P \times T) \cup (T \times P) \longrightarrow \mathbb{N}$ est la fonction de valuation des *arcs* ;
4. $M_0 : P \longrightarrow \mathbb{N}$ est le *marquage initial*.

Un réseau de Petri peut être représenté comme un graphe biparti orienté. Remarquons qu'il y a au plus un arc dans chaque direction pour un couple (place, transition) donné, et que l'action d'un arc est exprimée par son sens et la valeur de W.

Un marquage représente un état du système modélisé. Le marquage initial en est un cas particulier exprimant l'état initial du système. Les marquages sont représentés par des jetons dans les places.

Un *marquage* d'un réseau de Petri $\mathcal{N} = \langle P, T, W, M_0 \rangle$ est une fonction $M : P \longrightarrow \mathbb{N}$.

Lorsque le système est dans un état représenté par un marquage M, certaines transitions peuvent être franchies, conduisant à un nouvel état M'.

Le comportement d'un réseau de Petri peut être ainsi défini : soient un réseau de Petri $\mathcal{N} = \langle P, T, W, M_0 \rangle$ et un marquage M de ce $\mathcal{N}$. Une transition $t \in T$ est *franchissable* dans le marquage M (noté $M[t\rangle$) ssi :

$$\forall p \in P : W(p, t) \leq M(p)$$

Dans ce cas, le *franchissement* de la transition t dans le marquage M conduit à un nouveau marquage M' (on note $M[t\rangle M'$ tel que :

$$\forall p \in P : M'(p) = M(p) - W(p, t) + W(t, p)$$

∎

La principale difficulté de ce travail réside, comme souvent d'ailleurs, dans le choix des structures de données de formalisation. Dans ce cas précis, les types choisis pour formaliser un réseau de Petri vont conditionner les preuves traitant de ces réseaux.

Trois formalisations différentes ont été effectuées. Ces trois formalisations s'inscrivent à la fois dans une démarche d'expérimentation liée aux réseaux de Petri mais aussi liée aux évolutions de `Coq`. Notre première formalisation a utilisé une bibliothèque d'ensembles (`Sets`). Pour la seconde nous avons utilisé `FSets`, une bibliothèque modulaire d'ensembles qui venait juste d'être développée. La première bibliothèque n'était pas adaptée à notre problématique et, surtout, ne contenait pas assez de propriétés nécessaires à nos preuves de raffinement sans devoir compléter de manière significative la bibliothèque elle-même. Le développement en cours de `FSets`, ayant pour vocation de remplacer `Sets`, nous a poussé à ne pas poursuivre dans cette première voie. Le deuxième développement, modulaire, avec `FSets` était relativement satisfaisant pour la formalisation et les preuves des réseaux de Petri places/transitions. Néanmoins, la modularité ne nous a pas aidé lorsqu'il a fallu ajouter des couleurs aux réseaux afin de formaliser les réseaux de Petri colorés (CPN).

Rappels sur les CPN. un réseau de Petri coloré $\mathcal{N}$ est un 8-uplet $\mathcal{N} = \langle P, T, A, C, E, \mathbb{M}, \mathbb{Y}, M_0 \rangle$ tel que :

1. P est un ensemble de places
2. T est un ensemble de transitions, tel que $P \cap T = \emptyset$
3. A est un ensemble d'arcs, tel que $A \subseteq (P \times T) \cup (T \times P)$
4. $C : P \cup T \rightarrow \Sigma$ où Σ est un univers d'ensembles colorés non-vides (ou types), détermine les couleurs des places et les modes de transition.
5. $E : A \rightarrow \Phi\Sigma$ dénote les étiquettes des arcs, tel que $E(p, t), E(t, p) : C(t) \rightarrow \mu C(p)$
6. $\mathbb{M} = \mu\{(p, c) | p \in P, c \in C(p)\}$ est un ensemble de marquages qui associe un multiensemble de valeurs c à chaque place p de P.
7. $\mathbb{Y} = \mu\{(t, c) | t \in T, c \in C(t)\}$ est un ensemble d'étapes (multiensemble de transitions avec leur mode de déclenchement).

8. M_0 est le marquage initial, $M_0 \in \mathbb{M}$.

où $\Phi\Sigma$ est une fonction sur Σ définie par $\Phi\Sigma = \{X \to Y \mid X, Y \in \Sigma\}$ et $\mu X = \{X \to \mathbb{N}\}$ sont des multiensembles d'un ensemble X, et $\mathbb{N}$ est l'ensemble des entiers naturels.

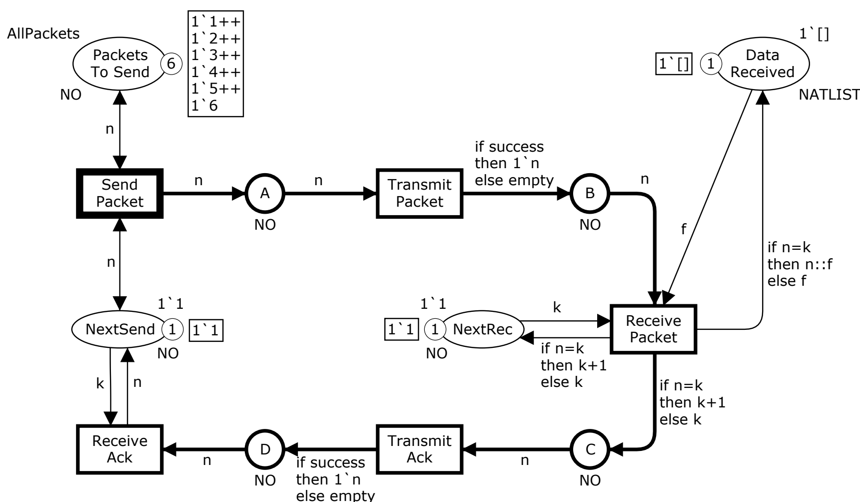


FIGURE 2.1 – Exemple d'un protocole simple

Dans l'exemple de la Figure 2.1, le marquage de le place `PacketsToSend` est le multiensemble `1‘1++1‘2++1‘3++1‘4++1‘5++1‘6`, où `1‘6` désigne une occurrence de valeur 6 et `++` désigne l'opérateur d'addition du multiensemble. ∎

Les couleurs peuvent être vues comme des types "à la Curry-Howard" (propriétés, preuves, programmes). De ce fait, il s'est avéré plus commode, aussi bien pour les preuves que dans le but d'étendre la formalisation des réseaux places/transitions, d'avoir une formalisation très simple à base de listes. Il s'agit de la troisième formalisation, présentée dans [CMP09, CMP10]. La souplesse de cette structure en listes pour les places et les transitions et en types pour les couleurs nous a également permis d'automatiser les preuves. Une gestion des non-preuves (lorsque la tactique automatique ne parvient pas à prouver qu'un tel réseau est bien un raffinement de tel autre) a également été faite afin de savoir pourquoi la preuve du raffinement échoue.

2.3 Difficultés

Rappels sur la vivacité et le blocage d'un réseau de Petri :

Un réseau de Petri est *vivant* si toute ses transitions sont vivantes. Une transition t est vivante si pour tout marquage M accessible depuis le marquage initial, t est franchie au moins une fois. La vivacité exprime le fait qu'en tout instant de l'évolution du système, le franchissement à terme d'une transition n'est jamais impossible.

Un réseau est dit *sans blocage* si tout marquage depuis le marquage initial n'est pas un marquage "puits". Un marquage M d'un réseau de Petri est appelé marquage "puits" si aucune transition n'est franchissable depuis M. ∎

Dans le cadre du stage de Master 2 de Naïm Aber [1], le protocole wtop-and-wait [BGP05] a été formalisé. Plus précisément, la partie statique (le réseau) du protocole a été formalisée. Les formalisations des aspects dynamiques tels que les propriétés de vivacité ou de blocage, n'ont pas été achevées.

Il y a très peu de formalisations qui prennent en compte ces aspects temps-réel dynamiques. Une telle expérimentation a été faite dans le prouveur HOL [HT09]

Un avantage à utiliser les outils de preuves formels en plus des outils de *model checking* traditionnels est la possibilité de prouver des propriétés sur des réseaux potentiellement infinis. Il y a donc une complémentarité entre les méthodes et cette complémentarité peut être mise à profit par le développement d'outils d'interfaçage. Même en restant dans un nombre fini de places et de transitions, ces interfaces seront nécessaires, ne serait-ce que pour définir les réseaux. Par exemple, faire une preuve sur un réseau ayant n places et transitions requiert l'écriture dans Coq d'autant de lignes pour remplir les listes. Une génération automatique s'impose donc, d'autant que n peut facilement valoir 1 million.

2.4 Perspectives possibles

Ces perspectives ne sont pour le moment que des perspectives possibles. En effet, les collaborations entre les auteurs provenant à la fois du domaine des réseaux de Petri et des preuves formelles sont actuellement suspendues car ces objectifs ne sont pas prioritaires et le temps nécessaire à ces recherches n'est pas disponible.

Néanmoins, il y aurait un certain nombre de perspectives possibles à court et moyen terme. Étudier comment prouver en Coq les propriétés telles que la vivacité nous semble particulièrement intéressant et pertinent.

De la même façon, une interface entre un outil d'édition de réseaux de Petri (tels que [AMI] ou [CPN]) et Coq est indispensable.

1. `http://lipn.univ-paris13.fr/~aber/RapportDeStage.pdf`

Chapitre 3

Approximation polynomiale rigoureuse

[Ce chapitre se réfère à l'article [BJMD+12] qui se trouve en annexe. Un autre article sur la partie preuve est en préparation.]

Un autre domaine critique où l'on souhaite pouvoir valider formellement les résultats est celui de l'arithmétique des ordinateurs. Nous nous intéressons ici aux calculs réalisés par les processeurs utilisant l'arithmétique flottante, en d'autres termes, aux résultats donnés par des calculs effectués à l'aide de nombres flottants. Les nombres réels, utilisés pour les raisonnement mathématiques, ne sont pas tous représentables en machine, et pour cause, certains sont transcendants (et donc non dénombrables). Une manière de représenter les nombres à virgules est la représentation à virgule flottante, à l'aide de l'arrondi lorsque le nombre réel voulu n'est pas représentable. C'est le calcul de ces arrondis qui sont critiques et auxquels les preuves formelles peuvent apporter un gage de sûreté. Ces spécifications représentent également un défi pour les outils de preuves qui n'ont pas l'habitude de traiter avec des structures continues.

La section 3.1 présente le contexte et un résumé des résultats depuis 2010. La section 3.2 tentera d'expliquer une des difficultés qu'il est nécessaire de surmonter, celle du calcul dans un outil d'aide à la preuve et dans Coq en particulier. En section 3.3, nous discuterons des choix techniques effectués pour être le plus générique possible. Nous détaillerons ensuite en section 3.4 les changements de spécification dues aux preuves, puis en section 3.5 nous aborderons les perspectives concernant cette thématique.

3.1 Contexte et résumé des résultats

Ce travail se situe dans le cadre du projet ANR TaMaDi[1] (*Table Maker's Dilemma*), porté par J.-M.Muller, ANR 2010.

1. `http://tamadiwiki.ens-lyon.fr/`

Le résultat d'une opération flottante entre deux nombres flottants n'est, en général, pas exactement représentable par un nombre flottant. Le résultat doit alors être arrondi. La norme IEEE-754-2008, standard pour la représentation des nombres à virgule flottante, requiert l'arrondi correct pour les quatre opérations de base, la racine carrée et le FMA (*Fused Multiply and Add* défini par $x * y + z$) et la préconise pour les autres fonctions. On dit que l'arrondi est correct si le résultat calculé est toujours égal à l'arrondi de la valeur exacte. Exiger l'arrondi correct présente l'avantage d'assurer la portabilité des calculs numériques. Mais garantir l'arrondi correct d'une fonction f en n'importe quel point x à virgule flottante impose la connaissance préalable de la plus petite distance non nulle entre $f(x)$ et la frontière de deux arrondis, les valeurs de x réalisant cette distance étant appelé pires cas. Ce problème est appelé le dilemme du fabricant de tables. Une méthode naïve est applicable pour la simple précision. Pour les précisions supérieures, deux algorithmes ont été conçus : Lefèvre [LM01] et SLZ [SLZ03]. Ces algorithmes découpent le domaine d'étude en petits sous-intervalles puis remplacent la fonction f par un polynôme d'approximation P. Ces deux algorithmes reposent sur des calculs très longs et complexes. L'idée est donc d'apporter des garanties fortes sur le résultat de ces calculs coûteux en contrôlant formellement l'erreur commise lorsque l'on remplace la fonction f par le polynôme P. Ce remplacement peut être effectué par une approximation polynomiale rigoureuse (RPA) [Jol11].

À cette fin, nous avons développé en Coq une bibliothèque d'approximation polynomiale rigoureuse utilisant les modèles de Taylor. La difficulté est de calculer, en un temps raisonnable, ce polynôme ainsi qu'un intervalle bornant l'erreur d'approximation au sein même de l'assistant de preuves. La spécification a été publiée [BJMD+12] dans les actes de la conférence NFM 2012.

3.2 La problématique du calcul

Coq n'est pas un langage de programmation ordinaire dans le sens où il se concentre sur la notion de preuve. Coq calcule de mieux en mieux, Coq fait du calcul formel de temps en temps, mais ces deux aspects ne sont pas ses atouts principaux. Le "vrai" calcul commence néanmoins, depuis une dizaine d'années, à manquer pour un certain nombres de formalisations et de preuves dans des domaines spécifiques. Ce travail, sur la preuve formelle de propriétés liées à l'arithmétique des ordinateurs, en est une parfaite illustration.

3.2.1 Les nombres en Coq

Rappels sur la norme IEEE-754(2008).

La norme IEEE-754(2008) est un standard pour la représentation des nombres à virgule flottante. Un nombre flottant est formé de trois éléments : la mantisse, l'exposant et le signe. Le bit de poids fort est le bit de signe. Les e bits suivants représentent l'exposant décalé, et les m bits suivants (m bits de poids faible) représentent la mantisse. Un nombre flottant est donc ainsi représenté :

$valeur = signe \times 1, mantisse \times 2^{exposant-d}$ avec $signe = \pm 1$ et $d = 2^{e-1} - 1$.

Par exemple, sur 32 bits, il y aura 1 bit de signe, 8 bits d'exposant, 23 bits

de mantisse, la valeur sera égale à $signe \times 1, mantisse \times 2^{exposant-127}$. Il est simple de vérifier que $-15,25 = C1740000_{16}$.

Tous les nombres n'étant pas exactement représentables ($\frac{1}{10} = 00111101110011001100110011001101_2 \simeq 0.100000001490116119 37_{10}$ par exemple), la norme spécifie également des modes d'arrondi :

- au plus près : retourne le nombre machine le plus proche du résultat exact ;
- vers zéro : retourne l'arrondi à $+\infty$ pour les nombres négatifs et l'arrondi à $-\infty$ pour les nombres positifs ;
- vers $+\infty$: retourne le plus petit nombre machine supérieur ou égal au résultat exact ;
- vers $-\infty$: retourne le plus grand nombre machine inférieur ou égal au résultat exact ;
- correct : le résultat calculé doit toujours être égal à l'arrondi de la valeur exacte.

La norme IEEE-754-2008, standard pour la représentation des nombres à virgule flottante, requiert l'arrondi correct pour les quatre opérations de base, la racine carrée et le FMA (*Fused Multiply and Add* défini par $x * y + z$) et la préconise pour les autres fonctions. Pour plus de détails sur les problèmes liés aux arrondis et aux flottants, le lecteur pourra se référer à [MD12]. ∎

Rappels sur l'arithmétique d'intervalle.

En arithmétique d'intervalles, un nombre réel x est représenté par une paire de nombres flottants $(x_{\text{inf}}, x_{\text{sup}})$, et $x \in [x_{\text{inf}}, x_{\text{sup}}]$.

Les opérations de base, pour deux intervalles $[a, b]$ et $[c, d]$, sont ainsi définies :

- $[a, b]+[c, d] = [min(a + c, a + d, b + c, b + d), max(a + c, a + d, b + c, b + d)] = [a + c, b + d]$
- $[a, b]-[c, d] = [min(a - c, a - d, b - c, b - d), max(a - c, a - d, b - c, b - d)] = [a - d, b - c]$
- $[a, b]\times[c, d] = [min(a \times c, a \times d, b \times c, b \times d), max(a \times c, a \times d, b \times c, b \times d)]$
- $[a, b]\div[c, d] = [min(a \div c, a \div d, b \div c, b \div d), max(a \div c, a \div d, b \div c, b \div d)]$ si 0 n'est pas dans $[c, d]$

La norme IEEE pour l'arithmétique d'intervalle est en cours de développement [2]. ∎

Les nombres deviennent rapidement nécessaires pour formaliser une grande majorité de théorèmes mathématiques, d'algorithmes ou de propriétés générales. Les entiers naturels ont été implantés, quasiment dès leurs premières versions, dans les prouveurs.

Les *entiers naturels* sont des nombres qui ne posent pas de difficulté majeures aux ordinateurs. Pour ce qui est des systèmes de preuves formelles, il faut néanmoins veiller à avoir des entiers dont les propriétés mathématiques sont simples à utiliser. L'ensemble des entiers naturels est un semi-anneaux abélien ordonné. Dans un langage de programmation "ordinaire" les entiers sont les entiers binaires du processeur qui sont adaptés aux calculs. Certains outils d'aide à la preuve utilisent également ces entiers en rajoutant une couche axiomatique qui permet d'utiliser les propriétés logiques. C'est le cas de PVS par exemple qui utilise les entiers de LISP.

En Coq, le système logique est celui du calcul de constructions inductives. De ce fait, le système repose sur la méthode de preuves par induction, c'est à

2. http ://grouper.ieee.org/groups/1788/

dire par application d'un schéma inductif. Un raisonnement par induction ne peut se faire que s'il est possible de l'appliquer à un type inductif. Or nous savons que les nombres entiers peuvent être construits par induction. Il s'agit des entiers de Peano, dont les deux constructeurs sont 0 et le successeur S. On retrouve alors le raisonnement mathématique par récurrence. Les entiers de Peano sont donc parfaitement adaptés aux preuves mais ils le sont beaucoup moins pour le calcul puisque, par exemple, pour représenter 10 il faut écrire $(S\ (S\ (S\ (S\ (S\ (S\ (S\ (S\ (S\ (S\ 0))))))))))$. En Coq il y a également une bibliothèque d'entiers binaires qui permet de faire à la fois du calcul plus efficace et des raisonnement inductifs (mais beaucoup moins intuitif que celui sur les entiers de Peano).

Les *entiers relatifs* sont aussi des nombres qui peuvent être définis inductivement. En Coq ce sont des entiers binaires, basés sur les entiers naturels binaires avec le 0, les positifs et les négatifs.

Les *nombres réels* peuvent être formalisés d'un grand nombre de façons. Cela peut être une axiomatisation ou une construction par les méthodes des suites de Cauchy, de Cantor, des coupures de Dedekind, ils peuvent être classiques ou intuitionnistes, ils peuvent être du premier ou du second ordre, etc. Nous décrirons de manière plus détaillée quelques unes de ces possibilités dans le chapitre 5. La bibliothèque standard *Reals* [May01, CDT] de Coq est une axiomatisation classique. En ce qui concerne le calcul, sujet qui nous intéresse principalement dans cette section, les nombres réels (les vrais ; voir chapitre 5) ne sont pas adaptés. Les nombres réels exacts [O'C08, KS11] permettent de calculer, mais ils ne sont pas suffisamment efficaces pour nos besoins.

Les *nombres flottants*, c'est-à-dire les nombres à virgule flottante, sont une représentation machine approchée des nombres réels. Plusieurs bibliothèques flottantes ont été développées en Coq. La plus récente est la bibliothèque Flocq [BM11]. Ces bibliothèques implémentent la norme IEEE-754 et s'appuient sur la bibliothèque standard des nombres réels, ce qui permet de garantir les propriétés mathématiques voulues.

L'*arithmétique d'intervalle* est une autre manière de représenter en machine les nombres réels. La bibliothèque CoqInterval [3] est une bibliothèque d'arithmétique d'intervalles où les bornes des intervalles sont soit des nombres flottants (Flocq), soit des valeurs abstraites (les réels de *Reals* étendus).

3.2.2 Réduire et Calculer

Dans la section 3.2.1 nous avons évoqué la problématique du calcul dans les prouveurs avec les nombres. Coq est basé sur une logique qui permet de calculer dans le sens où il est possible de définir une fonction Coq qui peut y être exécutée. La principale forme de calcul est alors le principe de conversion des λ-termes, qui forment la base du CCI. Plusieurs formes de conversions existent dans Coq, telles que l'alpha-conversion, la beta-reduction, la zeta-reduction, la delta-conversion, la iota-conversion, etc (voir [CDT]). Ces conversions permettent de réduire les λ-termes de Coq, ce qui est une forme de calcul. Par exemple, pour montrer que $(S\ (S\ 0)) + (S\ (S\ 0)) = (S\ (S\ (S\ (S\ 0))))$, il suffit d'effectuer la réduction de $(S\ (S\ 0)) + (S\ (S\ 0))$ en $(S\ (S\ (S\ (S\ 0))))$ puis d'utiliser la réflexivité de l'égalité.

3. http://www.lri.fr/~melquion/soft/coq-interval/

Outre l'automatisation, il existe des tactiques qui permettent d'effectuer ces conversions. La plus ancienne est la tactique `compute`. Cette tactique est équivalente à `cbv beta delta iota zeta` et effectue les normalisations suivant les réductions demandées. La tactique `vm_compute` [GL02] effectue la réduction mais en utilisant une machine virtuelle byte-code. Cet algorithme est bien plus efficace que celui de `compute`, mais il est moins sûr puisque basé sur une machine virtuelle non entièrement prouvée. La tactique `native_compute` [BDG11] effectue la réduction mais en utilisant la machine Ocaml native. De la même manière, cette méthodologie est encore plus efficace, mais encore moins sûr puisqu'il faudrait prouver formellement le compilateur natif d'Ocaml.

Ces deux dernières méthodes de réduction ont été développées récemment et sont encore en cours de développement (`native_compute` ne fait encore pas partie de la version distribuée de `Coq`). `Coq` est de plus en plus utilisé pour des calculs, le travail exposé ici en est une illustration. Ces calculs doivent être efficaces et prouvés. Ces nouvelles machines de conversion sont là pour apporter la partie efficacité à `Coq`.

3.3 Choix techniques

Le but de ce travail est l'implantation de modèles de Taylor efficaces et formellement prouvés d'approximation polynomiales. Une des particularité de cette implantation est sa généricité par rapport à deux composantes : la base de polynômes et les types de données.

3.3.1 Modularité par rapport à la base de polynômes

Un modèle de Taylor n'est en fait qu'une instance d'une notion plus générale qui est *l'approximation polynomiale rigoureuse* (RPA) [Jol11]. Soit une fonction f, une RPA est une paire $(T, \mathbf{\Delta})$ où T est un polynôme et $\mathbf{\Delta}$ un intervalle contenant l'erreur d'approximation entre f et T. Nous pouvons alors choisir des polynômes de Taylor pour T et les modèles de Taylor pour les approximations, mais il existe également les modèles de Chebyshev, basés sur les polynômes de Chebyshev. Pouvoir utiliser les deux modèles simplement représente un avantage certain. Les modèles de Taylor sont par exemple bien adaptés lorsque l'erreur d'approximation n'est pas trop large. Dans le cas contraire, les modèles de Chebyshev sont bien plus fins.

`Coq` fournit trois mécanismes permettant de faire des implantations génériques : les “type classes”, les “records” et les “modules”. Les modules sont moins génériques que les deux autres techniques qui sont de première classe (i.e. peuvent être reçus en argument et/ou renvoyés comme résultat d'une fonction), mais les applications de modules étant faites statiquement et notre implantation ne requérant qu'une généricité simple, nous avons choisi les modules qui nous ont semblé avoir un comportement calculatoire (calculs plus rapides) plus adapté à nos besoins.

Il y a donc un module abstrait pour les RPA, instanciées par des modules pour les modèles de Taylor.

La figure 3.1 résume cette généricité.

3.3.2 Modularité par rapport aux types de données

Nous manipulons des polynômes. La question habituelle réside donc dans quelle représentation utiliser ces polynômes. Et, de manière usuelle, ont tendance à s'opposer la commodité pour faire les preuves et l'efficacité du calcul. Nous avons donc choisi de généraliser également les types de données des polynômes ainsi que leurs coefficients, puis d'en faire ensuite des instanciations. La généralisation est faite grâce aux modules et une instanciation est faite avec des listes, les coefficients des polynômes pouvant être soit des réels (Reals), soit des flottants (Flocq), soit des intervalles de réels ou des intervalles de flottants (CoqInterval). En particulier, choisir une implantation fonctionnelle de polynômes (i.e. avec des listes) nous donne des modèles de Taylor directement exécutables dans Coq. Une autre instanciation par des tableaux est prévue. elle devrait être plus efficace.

La figure 3.1 résume également cette généricité.

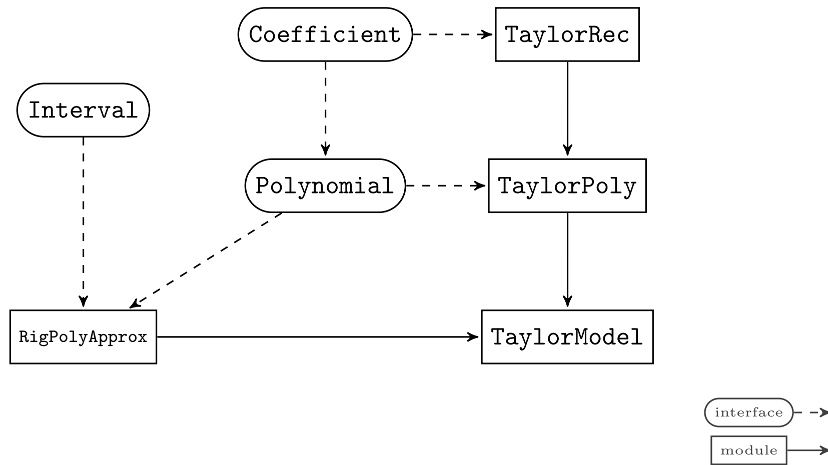


FIGURE 3.1 – Hiérarchie des modules pour les modèles de Taylor

Un des avantages majeurs de ces différentes généricités est que les preuves pourront être faites au niveau abstrait. Il n'y a donc pas besoin de "descendre" au niveau des structures de données telles que les listes ou les tableaux lors des preuves.

3.4 Les changements dus aux preuves

Malgré ce que nous venons de dire précédemment sur les avantages de la généricité pour les preuves, nous avons tout de même dû effectuer quelques modifications par rapport à la figure 3.1. Des modules de "lien" ont en effet dû être ajoutés. Ces modules de lien permettent de rester générique suivant que certaines propriétés des coefficients des polynômes sont présentent ou pas. Par exemple, la propriété de l'associativité de l'addition est vérifiée pour les réels, mais pas pour les flottants ni pour les intervalles. De ce fait, cette propriété ne peut être considérée à un haut niveau de généricité car elle ne serait alors pas instanciable pour certains types. De même, l'évaluation de l'addition de

deux polynômes n'est pas toujours égale à l'addition de l'évaluation des deux polynômes.

La figure 3.2 montre les ajouts qui ont été fait pour les preuves. Les flèches incurvées représentent les liens entre les modules qui ont dûs être ajoutés.

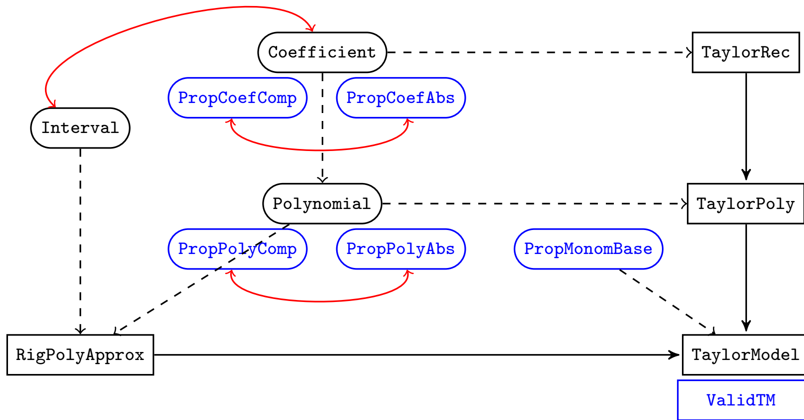


FIGURE 3.2 – Hiérarchie adaptée pour les preuves

3.5 Conclusion et Perspectives

Une publication concernant la partie sur les preuves proprement dites est en cours.

Ce travail ouvre de véritables perspectives quant à la possibilité de prouver formellement des pires cas pour les arrondis flottants. Ces pires cas sont extrêmement longs à calculer, plusieurs mois parfois et utilisant des algorithmes optimisés. Il est donc primordial de pouvoir être sûr de ces pires cas. Notre développement en Coq montre que les calculs sont raisonnables : les tests présentés dans [BJMD+12] montrent que nous ne sommes, en moyenne, que dix fois plus lent en Coq que des outils comme Sollya [CJL10], dédiés, entre autres, à ces calculs d'approximation polynomiale. Cette prouesse est due en partie à l'utilisation de `vm_compute`. Ces calculs en Coq pourraient également être encore plus rapides en utilisant, par exemple, les algorithmes de Karatsuba ou Toom-Kook à la place de l'algorithme naïf.

Une autre étape naturelle est de coupler ces preuves avec des tests de positivité des polynômes, par exemple en utilisant la technique des sommes de carrés (*sums-of-squares*), ce qui nous donnera une technique automatique de vérification formelle d'approximation polynomiale ainsi qu'une nouvelle évaluation de l'erreur d'approximation. Si cette erreur s'avère encore trop large, les modèles de Chebyshev pourront remplacer avantageusement les modèles de Taylor. Ce remplacement devrait pouvoir se faire à moindre coût grâce à nos choix initiaux pour une interface générique par rapport à la base de polynômes, comme le montre la figure 3.3.

Une suite logique à plus long terme consiste en l'étude de la correspondance entre les fonctions de base et le modèle d'approximation utilisé. Cela peut être

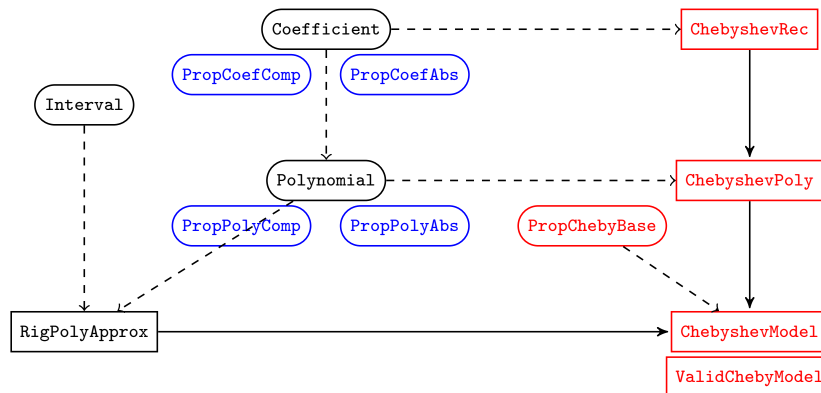


FIGURE 3.3 – Instanciation avec les modèles de Chebyshev

fait de manière manuelle, à l'aide d'une relation de récurrence à partir de la définition formelle. Mais il existe une approche plus générique à explorer, celle du DDMF [4] (*Dynamic Dictionary of Mathematical Functions*), qui nous générerait ces récurrences. Il s'agirait alors d'introduire des certificats par une approche sceptique (voir chapitre 4).

4. `http://ddmf.msr-inria.inria.fr/`

Chapitre 4

Calcul formel

[Ce chapitre se réfère aux articles [BHMT10, DM05, DM06], dont le [BHMT10] se trouve en annexe.]

Dans ce chapitre nous présenterons deux exemples d'interactions entre le domaine du calcul formel et celui des preuves formelles. Je considère depuis de nombreuses années ces deux domaines comme étant assez proches. Pouvoir utiliser les points forts de chacun d'entre eux m'a toujours semblé prometteur. Le point fort du calcul formel est le calcul et les algorithmes. Le point fort des preuves formelles est la confiance vis à vis de la correction des résultats.

La section 4.1 sera dédiée à la preuve formelle d'une fonction du logiciel SCHUR. La section 4.2 présentera l'élaboration d'une tactique en utilisant une interface entre Coq et Maple. En section 4.3 nous aborderons les perspectives dans cette thématique.

4.1 Preuve formelle de correction du logiciel SCHUR

Il s'agit de montrer formellement la correction, en utilisant l'outil de preuve de programme Frama-C [CCPg, FMM$^+$, MM], de certaines fonctions critiques (dans le sens primordiales) du logiciel de calcul de fonctions symétriques (SCHUR [1]) repris et développé par Franck Butelle, Ronald King et Frédéric Toumazet. Dans le cadre du projet PEPS CerBISS pluridisciplinaire CNRS/INST2I porté par F. Butelle, nous avons prouvé la correction d'une fonction clé, la fonction conjuguée [BHMT10].

Rappels de combinatoire algébrique.

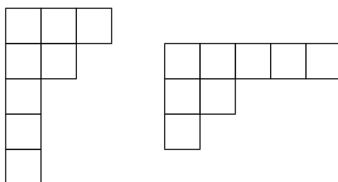
Une **partition** d'un entier positif n est une façon d'écrire n comme une somme d'une séquence d'entiers positifs non croissants. Par exemple $\lambda = (4, 2, 2, 1)$ et $\mu = (2, 1)$ sont des partitions de $n = 9$ et $n' = 3$ respectivement. Nous écrirons $|\lambda| = n$ et $|\mu| = n'$ [And84].

1. `http://schur.sourceforge.net/`

Le **diagramme de Ferrers** F^λ associé à une partition $\lambda = (\lambda_1, \lambda_2, ..., \lambda_p)$ est constitué de $|\lambda| = n$ « boîtes », rangées en $l(\lambda) = p$ lignes justifiées à gauche de longueur $\lambda_1,\ \lambda_2,\ ...,\ \lambda_p$. Les colonnes de F^λ sont orientées vers le bas. F^λ est appelée la forme de la partition λ.

Le conjugué de la partition d'un entier est géométriquement le symétrique, par rapport à la diagonale, de la représentation sous forme de diagramme de Ferrer de la partition.

Par exemple, pour $\lambda = (3, 2, 1, 1, 1)$, le diagramme de Ferrer F^λ et le diagramme de Ferrer de la partition conjuguée sont représentés ainsi :



Donc le conjugué de la partition $(3, 2, 1, 1, 1)$ est $(5, 2, 1)$.

Un **tableau de Young semi-standard** de forme λ est une numérotation des boîtes de F^λ avec des éléments de l'ensemble $\{1, 2, ..., n\}$, dans l'ordre croissant $(\geq)$ dans une ligne et strictement croissant en descendant une colonne.

Un tableau de Young est dit **standard** si et seulement si chaque entrée apparaît seulement une fois. Voici un exemple de tableau de forme $(4, 2, 2, 1)$

1	2	2	5
2	4		
3	6		
5			

Une **fonction symétrique** d'un ensemble de variables $\{x_1, x_2, \ldots\}$ est une fonction $f(x_1, x_2, \ldots)$ de ces variables qui est invariante sous toute permutation de ces variables (par exemple $f(x_1, x_2, \ldots) = f(x_2, x_1, \ldots)$). Cette définition est usuellement restreinte aux fonctions polynomiales. La base linéaire la plus importante de l'algèbre des fonctions symétriques est appelée les **fonctions de Schur** qui sont combinatoirement définies ainsi : pour un tableau de Young semi-standard T donné de forme λ, écrire $\mathbf{x}^T$ le produit des x_i pour tous les i apparaissant dans le tableau. Alors

$$s_\lambda(\mathbf{x}) = \sum_{T \in \mathrm{Tab}(\lambda)} \mathbf{x}^T. \tag{4.1}$$

où $\mathrm{Tab}(\lambda)$ est l'ensemble de tous les tableaux de forme λ. Nous noterons $s_\lambda(\mathbf{x})$, s_λ. Par exemple, considérons les tableaux de forme $(2, 1)$ utilisant uniquement trois variables x_1, x_2, x_3 :

1	1
2	

1	1
3	

2	2
3	

1	2
3	

1	3
2	

1	2
2	

1	3
3	

2	3
3	

La fonction de Schur associée est alors :

$$s_{(21)}(x_1, x_2, x_3) = x_1^2 x_2 + x_1^2 x_3 + x_2^2 x_3 + 2x_1 x_2 x_3 + x_1 x_2^2 + x_1 x_3^2 + x_2 x_3^2 \tag{4.2}$$

et

$$s_{(21)} = s_{(21)}(x_1, x_2, x_3) + s_{(21)}(x_1, x_2, x_3, x_4) + \dots$$

Notons que, avec cette définition combinatoire, la symétrie de $s_{(21)}(x_1, x_2, x_3)$ n'est pas évidente *a priori*. ∎

4.1.1 Description de la problématique

SCHUR est un logiciel interactif permettant de faire des calculs sur les groupes de Lie et les fonctions symétriques. Il est le résultat de 20 années de recherches en combinatoire algébrique, en physique, en chimie. Cet outil est également utilisé pour mettre au point des conjectures ainsi que pour l'enseignement.

SCHUR était à l'origine écrit en Pascal par le Professeur B. Wybourne. SCHUR a été traduit automatiquement du Pascal vers le C (40000 lignes de C). Les commentaires du code furent alors perdus. Notons également qu'un code traduit automatiquement n'est *a fortiori* pas écrit dans le but d'être formellement prouvé.

Dans ces conditions, prouver la correction d'un tel logiciel représente à la fois un gage de sûreté évident et un défi. L'objectif de montrer 40 000 lignes de code n'étant pas raisonnable à court ni moyen terme avec les techniques actuelles, nous avons souhaité nous intéresser à une fonction “significative” de SCHUR. “Significative” signifie 4 choses :

1. critique
2. apparaissant de manière récurrente dans le code
3. pas trop simple
4. pas trop compliquée

Ce travail est alors vu comme une première expérimentation de faisabilité.

4.1.2 Méthodologie

Le premier point réside dans le choix de la partie du code à prouver.

Dans un logiciel de combinatoire algébrique, pour toute structure combinatoire de base telle que les permutations ou les partitions, il y a typiquement 50 à 200 fonctions différentes. La conjuguée d'une partition est un très bon exemple de ce qu'une de ces fonctions calcule, c'est typiquement de la chirurgie sur des listes d'entiers ou sur des listes de listes d'entiers ou sur des structures récursives plus avancées telles que des arbres... Dans un calcul basique, la plupart du temps est passé à adapter ou itérer ces fonctions sur certains ensembles d'objets. Mais, à cause de l'explosion combinatoire, ces ensembles peuvent être très grands et donc ces fonctions doivent être optimisées finement.

Nous devons rappeler quelques résultats bien connus en théorie des fonctions symétriques : bien que les fonctions de Schur ont été historiquement définies par Jacobi [Jac41], elles ont été nommées ainsi en l'honneur de Schur qui a découvert leur rôle crucial dans la théorie des représentations du groupe symétrique et du groupe général linéaire. Plus précisément, après la découverte par Frobenius que la représentation irréductible des groupes symétriques est indexée par des partitions d'entier, Schur a montré que ces fonctions peuvent être interprétées comme les caractères de ces représentations irréductibles, et par la dualité de Schur-Weyl, des caractères des groupes de Lie et des algèbres de Lie.

Notamment nous obtenons la représentation du groupe général linéaire (GL_n) et des groupes unitaires (U_n) [Lit50] des représentations du groupe symétrique. Dans ce cadre, le conjugué d'une partition essentiellement encode le produit tensoriel d'une représentation par la représentation "signée".

D'autres travaux par Schur-Littlewood impliquent la somme infinie de fonctions de Schur associée aux partitions [LP88], dont les conjuguées ont des formes particulières.

Nous avons vu au chapitre 1 qu'il existait deux principales solutions pour prouver formellement un problème (sous forme d'algorithme ou de programme) par rapport à sa spécification. Soit on formalise directement l'algorithme dans le système de preuve formelle, puis on prouve les propriétés qui nous intéressent, soit on prouve le programme. Dans un cas comme celui-ci, prouver le programme qui existe et qui calcule est l'objectif. Cette méthode permet, en plus de algorithmes implantés dans le programme, de prouver les problèmes intrinsèques à toute implantation tels les débordements de tableaux etc.

Prouver formellement un programme à l'aide d'outils tels que Frama-C signifie qu'il faut annoter le programme. Les détails sont donnés dans [BHMT10]. Une fois les annotations faites, des obligations de preuve sont générées par l'outil. Ces obligations sont alors données à des prouveurs.

Les prouveurs utilisés pour montrer ces obligations de preuves appartiennent à deux catégories différentes : les prouveurs automatiques SMT (Satisfiability Modulo Theories) tels que Simplify [DNS05], Alt-Ergo [CCBL], Z3 [Mic], CVC3 [BT07] et les prouveurs interactifs tels que Coq [BC04], PVS [SORgC], Isabelle/HOL [NPW02].

Le problème de Satisfiabilité Modulo Théories (SMT) est un problème de décision pour des formules logiques classiques du premier ordre avec égalité. La logique du premier ordre avec égalité étant indécidable, les solveurs SMT cherchent à valider les problèmes par des techniques heuristiques. Pour des situation plus complexes, nous utiliserons les prouveurs interactifs pour prouver les obligations de preuves.

Deux difficultés surgissent alors. Celle de faire les "bonnes" annotations [BCF$^+$] et la confiance que l'on peut avoir dans les prouveurs :

1. De bonnes annotations : dans un premier temps, ce sont les prouveurs SMT qui vont tenter de montrer les obligations générées. C'est en cas d'échec que les méthodologies employées interviennent. Si un aucun prouveur SMT n'arrive à montrer une obligation, les raisons peuvent en être multiples : soit les annotations sont mauvaises (insuffisamment précises), soit les types logiques définis pour les annotations sont trop complexes, soit la propriété correspondant à l'annotation est trop complexe.

   Dans tous les cas, nous avons commencé par utiliser une méthode simple qui consiste à commencer la preuve de l'obligation "problématique" (non prouvée par les prouveurs automatiques) en utilisant un prouveur interactif d'ordre supérieur tel que Coq. Cette méthode nous a permis de trouver des failles et/ou des manques et/ou des erreurs dans les annotations. Elle nous a également permis de redéfinir des types mieux exploités par les prouveurs automatiques. Par exemple, nous avions commencé par axio-

matiser la notion de compteur de lignes d'une partition d'entiers durant la boucle. Nous nous sommes rendu compte que définir un prédicat pour cette notion aidait grandement les prouveurs automatiques.

Dans notre cas, Coq a été exclusivement utilisé comme débogueur et comme conseiller pour nos annotations puisque, au final, toutes les obligations (49 au total) ont pu être montrées par au moins deux des prouveurs SMT utilisés. Notons également que cette méthode nous a permis de nous rendre compte que le tableau qui représente le conjugué (le tableau B dans le code) est initialisé à zéro dans une autre partie (éloignée de la fonction conjuguée) des 40 000 lignes de code du logiciel SCHUR. Une annotation de précondition a donc pu être rajoutée, sans laquelle la postcondition disant que B était bien le conjugué de A ne pouvait être montrée ni dans Coq ni pas les prouveurs automatiques.

2. Quelle confiance accorder aux prouveurs ? Nous venons de dire qu'au moins deux prouveurs SMT ont montré chacune des 49 obligations de preuves. La contrainte que nous nous sommes imposées sur les "au moins deux" prouveurs automatiques est une contrainte concernant la confiance que l'on peut avoir dans les prouveurs SMT. Une expérience malheureuse où une des versions que nous utilisions prouvait toutes les obligations (même les fausses) nous a alertée. Les SMT prouveurs sont encore assez opaques dans leur algorithmes de décision dans le sens où ils fournissent difficilement des traces à l'utilisateur. Une thèse est en cours sur ce sujet [AFG$^+$11]. Notons que ce problème de confiance n'est pas du même ordre dans un système de preuves comme Coq puisque les preuves fournissent un λ-terme de preuve qui peut être vérifié par un outil extérieur.

Cette première expérimentation sur ce genre de programme complexe, comportant plusieurs dizaines de milliers de lignes de code sans commentaires et dont les algorithmes utilisés sont très astucieux est positive. Il a finalement été possible d'isoler une fonction pertinente et de la prouver.

L'étape à venir que nous nous proposons de traiter concerne la preuve d'une fonction purement énumérative, par exemple le calcul des coefficients de Littlewood–Richardson.

4.2 Procédure de décision pour les corps algébriquement clos

Avec D. Delahaye, nous avons implanté une procédure de décision [DM06] basée sur la méthode d'élimination des quantificateurs, qui utilise l'interface [DM05] que nous avons également développée, entre le prouveur Coq et l'outil de calcul formel Maple [2].

4.2.1 Description de la problématique

Comme nous l'avons soulevé à plusieurs reprises, une des difficultés dans les outils d'aide à la preuve est l'automatisation. En particulier, l'utilisateur ne devrait pas être ralenti par des preuves simples et sans intérêt d'un point

2. http://www.maplesoft.com/products/Maple/

de vue purement mathématique. La plupart de ces preuves simples se traitent d'ailleurs par le calcul (par exemple le calcul formel ou le calcul sur des entiers). Un exemple très simple est celui de la factorisation d'un polynôme à racines dans $\mathbb{Q}$. Tous les outils de calcul formel savent faire cela rapidement. Les outils de preuve généralement pas. L'idée de ces travaux est de tirer partie de la puissance d'autres outils mieux adaptés au calcul que les prouveurs tout en obtenant un résultat prouvé.

4.2.2 Méthodologie

Utiliser des calculs dans un prouveur peut se faire selon 4 approches.

L'approche autarcique

Les calculs sont effectués au sein même du prouveur. Les algorithmes correspondants doivent dont y être implantés et prouvés. Cette approche est la plus sure mais elle est également la plus difficile. Certains algorithmes ne sont même pas implantables dans le système logique de Coq qui doit respecter les règles du CCI.

L'approche croyante

Les calculs sont fait par un outil extérieur, ils sont utilisés dans le prouveur mais sans avoir été prouvés. Cette approche est la plus simple mais la moins sûre.

L'approche sceptique

Les calculs sont fait pas un outil extérieur, ils sont utilisés dans le prouveur mais après avoir été prouvés corrects par celui-ci. Avec cette approche, il est possible d'utiliser la puissance de calcul d'un outil extérieur tout en prouvant formellement les résultats obtenus. Notons ici que ce ne sont pas les algorithmes ni les outils extérieurs utilisés qui sont prouvés mais chaque résultat, au coup par coup. C'est cette particularité qui rend cette approche à la fois intéressante et relativement simple. Cette approche part du constat qu'il est plus aisé de prouver qu'un résultat donné est vrai que de le calculer en le prouvant.

L'approche par certificat

Cette approche est à relier à l'approche sceptique. L'approche par certificat consiste à avoir un certificat (fonction booléenne) formellement prouvé qui saura décider si un résultat est vrai ou pas. Si l'on se place du point de vue de la preuve formelle, cette approche est semblable à l'approche sceptique. Du point de vue d'un utilisateur extérieur (de calcul formel par exemple), cette approche présente l'avantage de fournir le certificat permettant de vérifier la validité des calculs.

Cette approche est plus récente que les trois autres. Elle a été utilisée récemment dans [GPT08, MD12].

L'approche sceptique nous a permis de développer une procédure de décision dans Coq, inspirée d'un algorithme de [BPR06]. Maple est appelé pour calculer

les pgcd et les polynômes de Bézout. Les résultats donnés par Maple sont prouvés dans Coq puis utilisés dans la tactique [DM06].

4.3 Perspectives

Les travaux à l'interaction entre les domaines du calcul formel et des preuves formelles confirment leurs complémentarités. Nous souhaitons continuer dans cette voie, les retours étant très bénéfiques pour les deux branches.

En ce qui concerne la suite de la preuve du logiciel SCHUR, l'étape à venir que nous nous proposons de traiter concerne la preuve d'une fonction purement énumérative, le calcul de coefficients de Littlewood–Richardson. Cet objectif représente une étape bien plus ardue. Pour prouver le programme qui calcule ces coefficients, il faut arriver à formaliser les propriétés qui les caractérisent. Or, de par la composante énumérative de la fonction, cette spécification formelle peut s'avérer assez délicate. Cet objectif représente donc un défi tant au niveau preuve de programmes qu'au niveau des spécifications mathématiques.

Chapitre 5

Les nombres réels dans Coq

[Ce chapitre relate une réflexion en cours. Il n'y a pas encore de publications sur ces réflexions.]

Les mots "*nombres réels*" sont souvent employés comme terme générique afin de représenter différentes notions selon le milieu scientifique qui les utilise. En effet, loin des formalisations mathématiques, qui sont déjà nombreuses, nous trouvons les nouvelles formalisations introduites par la science de l'informatique. Dans toutes ces formalisations concernant les nombres réels, nous pouvons citer, entre autres, les définitions mathématiques théoriques [Bou66], les travaux dans la domaine de l'informatique sur l'arithmétique exacte [Vui90, MM94, EPE96, O'C08, KS11], les nombres flottants dans les langages de programmation, les formalisations dans les systèmes formels [Dut96, Got00, May01, GWZ00, Har98, How86] et dans le domaine du calcul formel [Rio91].

Problématique

Dans les langages de programmation, qui servent à faire des calculs, on trouve les nombres à virgule flottante (ou nombres flottants). Ces nombres ne sont pas des nombres réels, mais en sont une approximation plus ou moins satisfaisante. En effet, on dénombre, en plus des résultats erronés, certaines propriétés élémentaires du corps des réels non respectées. Les deux exemples qui vont suivre illustrent ces deux problèmes. Pour plus de détails nous pourrons consulter [MM94, Vui90, Mul89, MBdD$^+$10]. La suite $(a_n)_{n\in\mathbb{N}}$ suivante, due à Jean-Michel Muller, est un exemple, parmi d'autres, des problèmes liés aux erreurs d'arrondi :

$$a_0 = \frac{11}{2},\ a_1 = \frac{61}{11},\ a_{n+1} = 111 - \frac{1130 - 3000/a_{n-1}}{a_n}$$

Bien que la notion de calcul dans un langage de programmation n'est en aucun cas une démonstration, on peut tout de même espérer retrouver un résultat “assez proche” de celui donné par une preuve mathématique (informelle ou formelle). Par récurrence on montre que

$$a_n = \frac{6^{n+1} + 5^{n+1}}{6^n + 5^n}$$

dont la limite à l'infini vaut 6.
En effectuant les calculs, en définissant une fonction récursive, en double précision (dans Ocaml), on obtient les valeurs suivantes :

n	2	14	15	16	17	18	19	21	25	26
a_n	5.59	15.41	67.47	97.14	99.82	99.98	99.99	99.99	100.	100.

D'après ces résultats on aurait tendance à croire que cette suite converge vers 100 et non vers 6.

L'exemple qui suit va nous permettre de mieux comprendre pourquoi nous ne pouvons utiliser les nombres flottants tels qu'ils sont définis dans ces langages de programmation. En particulier, la propriété d'associativité de l'addition n'est pas respectée pour les valeurs ci-dessous. Les calculs sont, ici encore, effectués en Ocaml.

```
# (10000003.+(-10000000.))+7.501;;
- : float = 10.501
# 10000003.+(-10000000.+7.501);;
- : float = 10.5010000002
```

Ce qui signifie que $(10000003 + -10000000) + 7.501 \neq 10000003 + (-10000000 + 7.501)$.

Les problèmes liés aux résultats erronés ont mis en avant la nécessité d'une arithmétique exacte. L'idée est d'obtenir des résultats exacts ou "aussi proches que l'on veut". Pour cela, l'utilisateur indique une borne supérieure sur l'erreur d'arrondi autorisée sur le résultat final et cette borne est respectée tout au long de calcul. Plusieurs travaux ont été réalisés sur ce sujet [Vui90, MM94, O'C08, KS11].

Cette dernière décennie a également vue progresser les bibliothèques basées sur les nombres à virgule flottante. Comme nous l'avons abordé au chapitre 3, les erreurs d'arrondi sont étudiées (et même prouvées) avec une extrême attention, ce qui garantit certaines propriétés importantes. Par exemple, nous pouvons citer la bibliothèque MPFR [FHL$^+$07] qui est une bibliothèque C de flottants multi-précision avec arrondi correct.

Nous pensons que ces approches pourraient servir à une extraction dans `Coq`, en spécifiant comme type `ML` une certaine implantation, puisque nous pourrions alors faire en sorte que les propriétés de corps soient respectées.

Dans les systèmes de calcul formel, nous trouvons plusieurs formes de définition pour les nombres réels. En plus des flottants en précision arbitraire (où il est possible de choisir une précision fixe, 1000 chiffres décimaux, par exemple, dans `Maple` [Rio91]), il existe une couche supplémentaire formelle. La donnée $sin(1)$ peut être vue comme la valeur formelle ou comme sa valeur flottante 0.8414709848. Néanmoins, même si ces systèmes sont plus valides

que les langages de programmation pour corroborer certaines propriétés, ils ne permettent pas de les vérifier de façon sûre. En effet, le contre-exemple cité précédemment concernant la suite a_n reste valable, c'est-à-dire que la suite continue également de converger vers 100 dans les systèmes de calcul formel.

Pour cette raison, il est impératif d'avoir de "vrais" nombres réels si l'on veut pouvoir faire des *preuves* qui doivent utiliser les propriétés de corps des nombres réels.

Histoire des nombres réels dans les prouveurs

Les débuts des nombres réels dans les prouveurs sont détaillés dans [May01] (1985 : construction intuitionniste par D.J. Howe dans Nuprl ; 1991 : axiomatisation classique par C. Jones dans Lego ; 1995 : construction classique par J. Harrison dans HOL ; 1996,2000 : axiomatisation classique par B. Dutertre et A. Gottliebsen dans PVS ; 1997 : axiomatisation classique par M. Mayero dans Coq ; 1999 : réels non standard par R. Gamboa dans ACL2 ; 2000 : axiomatisation intuitionniste par M. Niqui dans Coq ; 2000 : réels non standard par J. Fleuriot dans Isabelle). Dans Coq, les nombres réels furent motivés par la preuve du théorème des trois intervalles, un théorème dont l'énoncé ne fait intervenir que des entiers mais dont la preuve requiert des réels [May99, May00, May01]. Ces développements ont été accompagnés de tactiques d'automatisation, aussi bien dans Coq que dans PVS [DM01, MnM01].

Depuis 2002, la bibliothèque standard [May01] de Coq (V6.3,1999) a été étendue par O. Desmettre à l'analyse réelle ; la trigonométrie et l'intégration. A. Ciaffaglione, P. Di Gianantonio [CG06], B. Spitters, R. Krebbers [KS11], R. O'Connor et C. Kaliszyk [O'C08], ont réalisé une implantation en Coq de réels exacts. C. Cohen [Coh12] s'est intéressé aux réels algébriques dans Coq. Des bibliothèques de nombres à virgules flottantes ont été implantées au-dessus des nombres réels (standard) [BM11].

Ce chapitre est dédié aux réflexions menées dans le but d'une bibliothèque unique. Unique dans le sens où aussi bien les utilisateurs de l'analyse classique que intuitionnistes pourraient utiliser la même base. Nous commençons en section 5.1 par donner une description des deux principales librairies existantes dans Coq, puis, en section 5.2, nous exposerons les débuts de recherche réalisés dans une perspective d'unification. En section 5.3, nous verrons quelles sont les perspectives actuelles.

5.1 Formalisations des nombres réels en Coq

Afin de pouvoir les comparer, nous allons présenter de manière assez détaillée les deux principales bibliothèques des nombres réels qui sont utilisées par la majorité de développements actuels.

5.1.1 La bibliothèque standard : Reals

Comme nous l'avons dit en début de chapitre, la bibliothèque standard de Coq a été développée entre 1997 et 2003 et modifiée à la marge par la suite.

Cette bibliothèque est une axiomatisation basée sur le fait que $\mathbb{R}$ est un corps ordonné archimédien et complet. Comme pour toute axiomatisation, des constantes sont nécessaires :

$$R \quad \text{(set R)}$$
$$0 : R$$
$$1 : R$$
$$+ : R \to R \to R$$
$$\times : R \to R \to R$$
$$- : R \to R \quad \text{(symbole : opposé)}$$
$$/ : R \to R \quad \text{(symbole : inverse)}$$
$$<: R \to R \to \texttt{Prop}$$
$$=: R \to R \to \texttt{Prop}$$

Les autres constantes ($>$, $\leq$, $\geq$, $-$, $/$) sont définies à partir des précédentes :

$$r_1 > r_2 : r_2 < r_1$$
$$r_1 \leq r_2 : r_1 < r_2 \text{ ou } r_1 = r_2$$
$$r_1 \geq r_2 : r_1 > r_2 \text{ ou } r_1 = r_2$$
$$r_1 - r_2 : r_1 + -r_2 \quad \text{(moins et opposé)}$$
$$r_1/r_2 : r_1 * /r_2 \quad \text{(division et inverse)}$$

Les axiomes sont les suivants :

$$\text{commutativité de l'addition} : \forall r_1, r_2 : R, r_1 + r_2 = r_2 + r_1 \tag{5.1}$$
$$\text{associativité de l'addition} : \forall r_1, r_2, r_3 : R, (r_1 + r_2) + r_3 = r_1 + (r_2 + r_3) \tag{5.2}$$
$$\text{opposé} : \forall r : R, r + -r = 0 \tag{5.3}$$
$$\text{élément neutre de l'addition} : \forall r : R, 0 + r = r \tag{5.4}$$
$$\text{commutativité de la multiplication} : \forall r_1, r_2 : R, r_1 \times r_2 = r_2 \times r_1 \tag{5.5}$$
$$\text{associativité de la multiplication} : \forall r_1, r_2, r_3 : R, (r_1 \times r_2) \times r_3 = r_1 \times (r_2 \times r_3) \tag{5.6}$$
$$\text{inverse} : \forall r : R, \text{si } r \neq 0 \text{ alors } \frac{r}{r} = 1 \tag{5.7}$$
$$\text{élément neutre de la multiplication} : \forall r : R, 1 \times r = r \tag{5.8}$$
$$\text{anneau} : 1 \neq 0 \tag{5.9}$$
$$\text{distributivité} : \forall r_1, r_2, r_3 : R, r_1 \times (r_2 + r_3) = r1 \times r_2 + r_1 \times r_3 \tag{5.10}$$
$$\tag{5.11}$$

ordre total : $\forall r_1, r_2 : R, r_1 < r_2$ ou $r_1 = r_2$ ou $r_1 > r_2$ (5.12)

asymétrie de lt : $\forall r_1, r_2 : R,$ si $r_1 < r_2$ alors $r_2 \not< r_1$ (5.13)

transitivité de lt : $\forall r_1, r_2, r_3 : R,$ si $r_1 < r_2$ et $r_2 < r_3$ alors $r_1 < r_3$ (5.14)

compatibilité pour l'addition : $\forall r, r_1, r_2 : R,$ si $r_1 < r_2$ alors $r + r_1 < r + r_2$ (5.15)

compatibilité pour mult : $\forall r, r_1, r_2 : R,$ si $0 < r$ et $r_1 < r_2$ alors $r \times r_1 < r \times r_2$ (5.16)

Archimédien : $\forall r : R, \exists n : N, r < n$ et $n - r \leq 1$ (5.17)

complet : tout ensemble E borné et non vide de R admet une borne supérieure (5.18)

La plupart de ces propriétés sont usuelles d'un point de vue strictement mathématique, mais du point de vue de la théorie des types et de Coq en particulier, nous allons en détailler quelques unes.

Une sorte pour R

Rappels sur le Calcul des Constructions Inductives (CCI), les "sorte" de Coq et l'imprédicativité. Le système de types de Coq, issu du CCI, se fonde sur les deux principales propriétés qui sont que chaque terme bien formé a au moins un type et que chaque type bien formé est un terme du calcul. De ce fait, un type a également un type, qui a lui même un type, etc. le type d'un type est ce que l'on appelle une "sorte". Il existe 3 sortes en `Coq` : `Prop`, `Set` et `Type`. Une manière rapide de présenter ces 3 sortes consiste à dire que `Prop` est la sorte des propositions logiques, `Set` et `Type` celles des types calculables, `Type` contenant en plus des univers ($\texttt{Type}_i : \texttt{Type}_{i+1}$) afin d'éviter l'incohérence `Type` :`Type` [GLT89].
Dans les versions de `Coq` antérieures à la version 8, la sorte `Set` était *imprédicative*. Par exemple, le terme `forall A:Set, A -> A` était de type `Set`. L'imprédicativité de `Set` permettait de définir des types inductifs dans le Calcul des Constructions (CC) de Thierry Coquand en 1984 [Coq85] et avant l'extension au CCI par Christine Paulin en 1996 [PM96]. Par exemple, une méthode équivalente pour définir

```
Inductive prod (A B : Set) :=  pair : A -> B -> prod AB
```

en utilisant l'imprédicativité de `Set` est

```
Definition prod (A B : Set) := forall C: Set, (A->B->C) -> C.
```

Pour plus de détails sur l'imprédicativité, le lecteur pourra se référer à [Wer94, PM96, Miq01]. ∎

Pour axiomatiser une théorie, la première chose à faire est de définir le type de donnée principal. Dans notre cas il s'agit du type de R représentant l'ensemble $\mathbb{R}$ des nombres réels. Nous pouvons décider de donner le type `Prop`, `Set` ou `Type` à R. En ce qui concerne la sorte `Prop`, nous pouvons l'exclure immédiatement, les réels ne sont pas des propositions. Le choix se situe entre `Set` et `Type`. R a vécu dans `Type`, puis, à partir de la version 8, dans `Set`. Jusqu'à assez récemment, les modèles possibles de `Set` imprédicatif n'étaient pas bien connus, laissant principalement place à l'intuition quant à la cohérence des problèmes faisant intervenir l'existence de fonctions non calculables. C'était pour cette raison que nous avions mis R dans `Type`. Pour plus de détails sur ce choix, se référer à [May01].

Au début des années 2000, des travaux ont été menés afin de comprendre plus précisément les relations entre décidabilité, réalisabilité et imprédicativité. Benjamin Werner a montré [1] en 2003 que, sous certaines conditions (décidabilité de l'égalité de R, l'axiome de description utilisé pour la complétude, ...) il était possible, en utilisant un argument à la Diaconescu, de montrer `{~A}+{~~A}`, ce qui rentre en contradiction avec l'imprédicativité de `Set`.

Depuis l'introduction des types inductifs, l'imprédicativité de `Set` n'était que très rarement utilisée. Afin d'éviter une incohérence dans la théorie même de Coq, l'équipe de développement décida pour le passage à la version 8, de déclarer la sorte `Set` comme prédicative par défaut et de mettre R dans `Set`.

Le symbole et l'axiome pour l'inverse (5.7)

Nous avons fait le choix de définir la fonction inverse comme une *fonction totale*, i.e. de type `R->R`. Nous aurions pu choisir de la définir en tant que fonction partielle, i.e. `forall x:R, x<>0 -> R`. Dans ce dernier cas, à chaque écriture du symbole inverse, une preuve du fait que l'inverse doit être différent de zéro est alors nécessaire. Par contre, avec une fonction inverse totale, il est possible d'écrire $\frac{1}{0}$, ce qui n'est pas un problème. En effet, Il est possible de donner une valeur arbitraire à l'interprétation de $\frac{1}{0}$ tout en préservant la cohérence grâce à l'axiome 5.7, qui, lui, requiert l'hypothèse que l'inverse doit être différent de zéro pour autoriser la simplification. L'équivalence entre les deux méthodes est montrée dans [May01].

L'axiome d'ordre total (5.12)

En Coq il existe plusieurs méthodes pour énoncer la totalité de l'ordre. Nous allons nous intéresser aux deux méthodes, l'une utilisant le *ou* définit à l'aide de `sumbool` et l'autre de `or` :

```
Inductive sumbool (A B : Prop) : Set :=
    left : A -> {A} + {B} | right : B -> {A} + {B}

Inductive or (A B : Prop) : Prop :=
    or_introl : A -> A \/ B | or_intror : B -> A \/ B
```

La différence se situe au niveau de leur type `sumbool: Prop -> Prop -> Set` et `or: Prop -> Prop -> Prop`.

La première définition permet de construire des valeurs (grâce au principe d'élimination forte), tandis que la seconde est juste propositionnelle. Prenons l'exemple de la fonction *min*. Avec la première définition, nous pouvons "décider" du min en choisissant grâce à l'axiome si nous sommes dans le cas "de gauche" ou "de droite". Mais étant donné que sur les nombres réels, nous sommes justement incapables de décider de l'égalité de deux nombres, cet axiome s'apparente à un axiome de la logique classique. Notons que le même axiome dans les entiers (dont l'égalité est décidable) n'apporte pas une composante classique. Rappelons que grâce à la prédicativité de `Set`, définir un ordre total pour les nombres réels en utilisant `sumbool` n'introduit pas d'incohérence. C'est ce choix que nous avons fait.

1. `http://coq.inria.fr/V8.1/stdlib/Coq.Logic.Diaconescu.html`

L'axiome de complétude (5.18)

Pour définir la complétude correspondant à la définition 5.18, nous avons besoin des notions de "borné", "non vide" et de "borne supérieure". Ces notions sont définies en Coq de la façon suivante :

```
Definition is_upper_bound (E:R -> Prop) (m:R) :=
     forall x:R, E x -> x <= m.
Definition bound (E:R -> Prop) :=
     exists m : R, is_upper_bound E m.
Definition is_lub (E:R -> Prop) (m:R) :=
     is_upper_bound E m /\
     (forall b:R, is_upper_bound E b -> m <= b).

Axiom completeness :
     forall E:R -> Prop,
        bound E -> (exists x : R, E x) -> { m:R | is_lub E m }.
```

Notons que nous formalisons "borné" par il existe un majorant et E "non vide" par il existe au moins un élément dans E.

Similairement au "ou", il existe plusieurs manières de spécifier un "il existe". Pour les mêmes raisons que celles exposées à la section précédente (prédicativité de `Set`,...), nous avons choisi un "il existe" à valeur dans `Set`.

L'axiome d'Archimède (5.17)

Dans ce cas également il existe plusieurs manières de formaliser la propriété 5.17. La première consiste à utiliser un "il existe" tel que cela est fait dans la plupart des définitions mathématiques et dans 5.17. La seconde méthode, que nous avons utilisée, consiste à *skolémiser* la propriété : il s'agit de définir un nouveau symbole (appelé symbole de Skolem) par quantificateur existentiel qui interviendra dans l'énoncé de l'axiome afin d'éliminer les occurrences des quantificateurs existentiel :

```
Parameter up : R -> Z.
Axiom archimed : forall r:R, IZR (up r) > r /\ IZR (up r) - r <= 1.
```

où `IZR` est la fonction d'injection de $\mathbb{Z}$ dans $\mathbb{R}$.

Notons que la seconde partie de l'axiome est plus forte que l'axiome d'Archimède usuel. Ce choix est historique, il permettait juste de définir les parties entière et fractionnaire directement [May00]. Cet énoncé est équivalent à l'énoncé usuel `forall r:R, IZR (up r) > r` et la preuve [May01] est triviale en utilisant le fait qu'un ensemble ordonné et minoré admet un plus petit élément.

5.1.2 La bibliothèque intuitionniste : CoRN

Rappels sur les sétoïdes et la relation d'"apartness". Un sétoïde est un ensemble muni d'une relation d'équivalence. En logique intuitionniste, la relation d'équivalence est remplacée par la relation d'*apartness*, noté $\#$, et l'on parle de sétoïde constructif. Une relation d'*apartness* est une relation binaire irréflexive, symétrique et co-transitive :

1. $\neg(x\#x)$
2. $x\#y \to y\#x$
3. $x\#y \to (x\#z \vee y\#z)$

∎

La bibliothèque C-CoRN (Constructive Coq Repository at Nijmegen), développée par Milad Niqui dans le cadre du projet FTA (Fondamental Theorem of Algebra) [GN00, GWZ00], est une axiomatisation de nombres réels intuitionnistes. En particulier, cela signifie qu'il est possible, en réalisant les axiomes intuitionnistes, d'obtenir une implantation d'algorithmes réels. Cette bibliothèque suit la structure hiérarchique de la théorie algébrique qui commence aux sétoïdes, suivie par les groupes, les anneaux puis les corps.

Soit R un ensemble. Les constantes et les relations nécessaires sont les suivantes :

$$\begin{array}{cl} \mathbf{0} & : R \\ \mathbf{1} & : R \\ = & : \text{prédicat binaire sur } R \\ \# & : \text{prédicat binaire sur } R \end{array}$$

Un nouveau type R^* (R sauf 0) est introduit :

$$(x, \phi_x) : R^* \Leftrightarrow x : R \wedge \phi_x : x\#\mathbf{0} \ .$$

Remarquons que ce type est en fait une paire contenant un élément de R et une preuve ϕ_x de $x\#0$.

Les opérations sur R sont ainsi définies :

$$\begin{array}{rl} < & :: \text{un prédicat binaire sur } R \\ + & : R \longrightarrow R \longrightarrow R \\ - & : R \longrightarrow R \\ * & : R \longrightarrow R \longrightarrow R \\ (_,_)^{-\mathbf{1}} & : R^* \longrightarrow R \end{array}$$

Et l'on appelle $\langle R, =, \#, <, +, \mathbf{0}, -, *, \mathbf{1}, (_,_)^{-\mathbf{1}}, \rangle$ un *corps constructif ordonné* s'il satisfait les propriétés suivantes :

$$\forall x : R.\neg(x\#x) \tag{5.19}$$
$$\forall x, y : R.(x\#y) \to (y\#x) \tag{5.20}$$
$$\forall x, y : R.(x\#y) \to \forall z : R.(x\#z) \vee (z\#y) \tag{5.21}$$
$$\forall x, y : R.\neg(x\#y) \leftrightarrow (x = y) \tag{5.22}$$
$$\forall x_1, x_2, y_1, y_2 : R.x_1 + y_1\#x_2 + y_2 \to x_1\#x_2 \vee y_1\#y_2 \tag{5.23}$$
$$\tag{5.24}$$

$$\forall x, y, z : R.x + (y + z) = (x + y) + z \tag{5.25}$$
$$\forall x : R.x + \mathbf{0} = x \tag{5.26}$$
$$\forall x, y : Rx + y = y + x \tag{5.27}$$
$$\forall x, y : R. - x\# - y \rightarrow x\#y \tag{5.28}$$
$$\forall x : R.x + (-x) = \mathbf{0} \tag{5.29}$$
$$\forall x_1, x_2, y_1, y_2 : R.x_1 * y_1\#x_2 * y_2 \rightarrow x_1\#x_2 \vee y_1\#y_2 \tag{5.30}$$
$$\forall x, y, z : R.x * (y * z) = (x * y) * z \tag{5.31}$$
$$\forall x : R.x * \mathbf{1} = x \tag{5.32}$$
$$\forall x, y : R.x * y = y * x \tag{5.33}$$
$$\forall x, y, z : R.x * (y + z) = (x * y) + (x * z) \tag{5.34}$$
$$\mathbf{1\#0} \tag{5.35}$$
$$\forall x : R\forall H_x : x\#\mathbf{0}.(\mathbf{x}, \mathbf{H_x})^{\mathbf{-1}}\#\mathbf{0} \tag{5.36}$$
$$\forall x : R\forall H_x : x\#\mathbf{0}.x * (\mathbf{x}, \mathbf{H_x})^{\mathbf{-1}} = \mathbf{1} \tag{5.37}$$
$$\forall x, y, z : R.x < y \rightarrow y = z \rightarrow x < z \tag{5.38}$$
$$\forall x, y, z : R.x < y \rightarrow x = z \rightarrow z < y \tag{5.39}$$
$$\forall x_1, x_2, y_1, y_2 : R.x_1 < y_1 \rightarrow x_2 < y_2 \vee x_1\#x_2 \vee y_1\#y_2 \tag{5.40}$$
$$\forall x, y, z : R.x < y \rightarrow y < z \rightarrow x < z \tag{5.41}$$
$$\forall x : R.\neg x < x \tag{5.42}$$
$$\forall x, y : R.x < y \rightarrow \forall z : R.x + z < y + z \tag{5.43}$$
$$\forall x, y : R.\mathbf{0} < x \rightarrow \mathbf{0} < y \rightarrow \mathbf{0} < x * y \tag{5.44}$$
$$\forall x, y : R.x\#y \leftrightarrow x < y \vee y < x \tag{5.45}$$

Tout comme dans la section sur la bibliothèque standard des réels de Coq, nous allons détailler ci-après quelques particularités de cette bibliothèque intuitionniste.

La structure hiérarchique

L'ensemble R avec les deux relations binaires satisfaisant 5.19–5.22 est un sétoïde constructif.

Un sétoïde constructif avec une opération binaire et une constante satisfaisants 5.23–5.27 est un monoïde constructif.

Un monoïde constructif avec une fonction unaire satisfaisant 5.28 et 5.29 est un groupe constructif.

Un groupe constructif avec une opération binaire et une constante satisfaisants 5.30–5.35 est un anneau constructif.

Un anneau constructif avec une fonction partielle unaire satisfaisant 5.36 –5.37 est un corps constructif.

Dans la suite de cette section nous omettrons le mot “constructif” lorsque nous citerons ces structures.

L'axiome d'Archimède

Il est défini de la manière suivante :

$$\forall x : R \exists n : \mathbb{N}, x \leq \mathsf{nring}(n). \tag{5.46}$$

où $x \leq y := \neg(y < x)$ et nring représente l'injection de $\mathbb{N}$ dans un anneau.

L'axiome de complétude

La complétude ici est définie comme l'existence d'une limite d'une suite de Cauchy de R.

$$\text{CauchySeq}_R :=$$

$$\{g : \mathbb{N} \longrightarrow R | \forall \epsilon > \mathbf{0} \exists M : \mathbb{N} \forall m : \mathbb{N}, M \leq_{\mathbb{N}} m \rightarrow g(M) - \epsilon < g(m) < g(M) + \epsilon\}$$

Notons que cette définition ne fait pas intervenir de valeurs absolues sur R puisque c'est une opération qui ne peut être définie qu'*a posteriori* à partir de l'axiome de complétude.

R est alors un corps ordonné archimédien et complet si :

$$\forall g : \text{CauchySeq}_R, \forall \epsilon > \mathbf{0}\ \exists M \forall m \geq_{\mathbb{N}} M, \lim g - \epsilon < g(m) < \lim g + \epsilon \tag{5.47}$$

avec une opération limite telle que $\lim : \text{CauchySeq}_R \longrightarrow R$.

Le choix des sortes

Tout comme pour la bibliothèque standard, la question de la sorte de R se pose. Actuellement [2], c'est la sorte `Type` qui est utilisée pour R, la valeur de l'*apartness* et celle de $\leq$. Ce n'était pas le cas dans les premières versions qui ont servi à la preuve du FTA. Il est naturel que les types de # et de $\leq$ soient à valeur dans `Prop`. Néanmoins, cette bibliothèque étant constructive, un souhait légitime consiste à vouloir donner la possibilité d'en extraire (voir le rappel sur l'extraction de la section 1.3 du chapitre 1) un programme. Le fait que ces opérateurs n'avaient pas de contenu calculatoire gênait l'extraction et le changement vers la sorte `Type` a alors été introduit [CFS03, CFL06]. Cela a également induit des changement dans certains axiomes tel que l'axiome d'Archimède par exemple, qui est passé de la formulation

```
∀x: R, ∃n : nat,  x < nring n.
```

(où le $\exists$ est à valeur dans `Prop`)

à la formulation :

```
∀x: R, {n : nat | x ≤ nring n}.
```

(où les accolades représentent une existentielle à valeur dans `Set`.)

2. En 2010. Cela est en train de changer à nouveau (Bas Spitters).

5.2 Réflexions pour une bibliothèque unique

Les deux bibliothèques présentées ci-dessus (*Reals* et *CoRN*) sont intensivement utilisées par les utilisateurs de Coq. Une des difficultés pour un utilisateur novice des nombres réels est de comprendre pourquoi il en existe plusieurs et laquelle il faut choisir. La réponse à la première question est simple, comme nous l'avons exposé en section 5.1, l'une est intuitionniste et l'autre "plutôt" classique. Nous allons détailler ci-après les différences majeures et expliciter le "plutôt classique". Par contre, la réponse à la seconde question peut s'avérer délicate. Savoir à l'avance ce qu'une formalisation va nécessiter n'est pas toujours si simple. L'idée de ce travail est donc, afin de minimiser la complexité des preuves formelles lorsqu'il y a des nombres réels, d'étudier la possibilité d'une bibliothèque unique réunissant les principales caractéristiques des deux précédentes.

5.2.1 Récapitulatif des points problématiques

Rappels sur les principes d'omniscience et de Markov. Le principe d'omniscience énonce que si A est un ensemble et P est une propriété dépendant d'un paramètre x élément de A, alors tous les éléments de A vérifient P ou bien l'un des éléments de A ne vérifie pas P. Ce principe est un principe issu de la logique classique (du tiers exclu) et n'est pas admis en logique intuitionniste.

La principe de Markov énonce que pour tout réel x, $\neg(x = 0) \to x \neq 0$. Ce principe est plus faible que le principe d'omniscience. En effet, dans la plupart des systèmes logiques y compris constructifs, une preuve de $\neg(\forall n\ a_n = 0)$ consiste à prouver $\exists n\ a_n \neq 0$. ∎

En collaboration avec M. Niqui, nous avons isolé les problèmes suivants :

1. L'axiome d'ordre total. Cet axiome établit que soit un réel x est égal à 0 soit qu'il est différent. En analyse constructive, cela signifie intrinsèquement que l'on est capable de décider d'une telle propriété. Or, nous savons que l'égalité dans les nombres réels est indécidable. Néanmoins, la totalité de l'ordre est couramment utilisée dans les preuves mathématiques. Si nous considérons le principe d'omniscience, alors l'ordre total en est une instance et nous pouvons alors dire que cet axiome est classique. De ce fait, il n'existe pas et ne peut pas exister dans la bibliothèque intuitionniste C-CoRN ni dans l'éventuelle partie commune provenant des deux bibliothèques.
2. L'*apartness* vs l'égalité de Leibniz. Comme nous l'avons vu dans le point 1, l'égalité de Leibniz ne peut pas être utilisée constructivement pour les nombres réels qui ne sont pas décidables. Pour cette raison, la relation d'*apartness* remplace l'égalité. Il s'agit ici d'une incompatibilité importante entre les deux bibliothèques. L'égalité de Leibniz est très appréciée par les utilisateurs qui s'intéressent aux preuves classiques de problèmes numériques. Il serait bien sûr possible d'utiliser une égalité de sétoïdes pour ces preuves. Une question qui est soulevée est alors celle de la simplicité et de la compréhension de la notion d'*apartness* par les utilisateurs.
3. L'axiome (5.7) correspondant à la fonction *inverse* de la bibliothèque standard n'est pas prouvable en C-CoRN. Comme expliqué précédemment, la

précondition $x \neq 0$ n'est pas prouvable en C-CoRN. On ne peut donc déduire l'axiome (5.7) à partir de C-CoRN tandis que les propriétés sur la fonction partielle de C-CoRN sont prouvables à partir des axiomes de *Reals* [KO09] en utilisant une forme de principe de Markov.

4. L'axiome de complétude. La notion de borne supérieure est également une notion délicate en logique intuitionniste. En effet, tout ensemble de réels majoré admet une borne supérieure. Cependant, cette borne est généralement impossible à obtenir ce qui la rend juste formelle. L'existence de cette borne n'est donc pas constructive mais classique. Une manière de résoudre ce problème est d'utiliser la version constructive de la complétude par les suites de Cauchy telle qu'elle est donnée dans C-CoRN.
5. $<$ et $=$ à valeurs dans `Type` ou `Prop`. Cette différence n'est pas un vrai problème. Bien que cela semble peu naturel au premier abord, il est parfaitement possible de définir une inégalité à valeurs dans `Type` également pour une bibliothèque classique.
6. La hiérarchie de C-CoRN. Cette différence ne représente pas de difficulté, une hiérarchisation pouvant être mise en place pour l'ensemble simplement et étant même souhaitable.

5.2.2 Quelques options possibles

Suite aux difficultés isolées ci-dessus, nous avons dégagé trois principales méthodes possibles, que nous exposons ci-après, en vue d'une bibliothèques commune issue des bibliothèques précédentes.

Dans tous les cas, quelques améliorations mineures, telles que remplacer les fonctions d'injection par des prédicats comme c'est le cas dans l'extension SSreflect [GMT08], pourront être faites. La structure hiérarchique quant à elle est indispensable.

Un anneau commun

Cette solution consiste à intégrer dans une partie commune les propriétés de sétoïde, de groupe et d'anneau que nous noterons $< A, =, +, \times, -, 0, 1 >$. Puis, au niveau des propriétés de corps, effectuer l'embranchement :

- un corps avec une inégalité et une fonction inverse totale formera la branche classique de la bibliothèque
- un corps avec l'*apartness* et une fonction inverse partielle formera la branche intuitionniste

Cette solution est la plus simple de toutes. Elle présente néanmoins l'inconvénient de réduire à son minimum la partie commune de la bibliothèque, ce qui en fait une solution peu intéressante au final.

Un corps ordonné commun

Cette solution sépare la notion d'ordre et introduit la notion de raisonnement par cas (dichotomie). Nous commençons par définir un sétoïde, un semi-groupe puis un anneau. Ensuite, nous introduisons la notion d'ordre avec le symbole $<$, ce qui nous donne un sétoïde ordonné, un semi-groupe ordonné, un anneau ordonné, un corps ordonné puis un corps ordonné archimédien complet. Nous pouvons alors ajouter la notion de dichotomie (plus faible que la décidabilité)

qui revient à faire un raisonnement par cas sur $x < y$ ou $\neg(x < y)$. Nous obtenons alors une bibliothèque similaire à la bibliothèque standard classique. La plupart des tactiques de `Coq` pourraient s'appliquer directement ou après une légère adaptation. La figure 5.1 résume cette solution ("O" est utilisé pour *Ordered*, "D" pour *Dichotomy*, "SL" pour *Standard Library*).

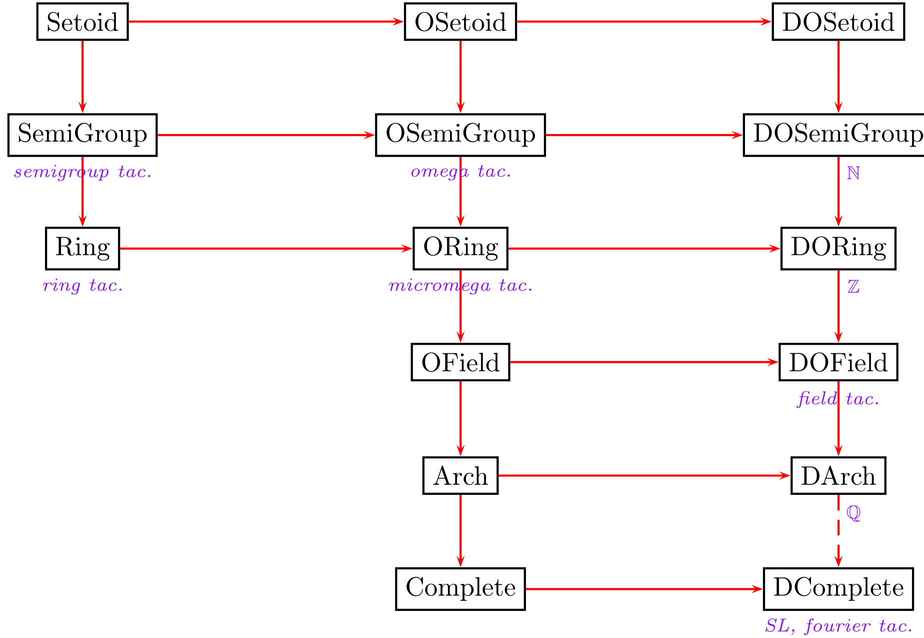


FIGURE 5.1 – Seconde proposition vers une unification des bibliothèques

Cette proposition présente le même inconvénient que la précédente, à savoir pas de véritable partie commune. Seuls quelques aménagements faits à la marge (introduction de <, affaiblissement de l'ordre total) permettant de rapprocher les deux bibliothèques.

Suivre C-CoRN

Cette solution suit la hiérarchie algébrique de C-CoRN en remplaçant l'*apartness* par < à valeur dans `Prop`, l'ordre total par la dichotomie, l'axiome de complétude énoncé grâce à la borne supérieure par celui énoncé grâce aux suites de Cauchy et l'inverse étant une fonction totale mais dont la condition $x \neq 0$ est remplacée par ($x < 0$ ou $0 < x$).

Nous avons réalisé une implantation partielle de cette solution en utilisant les Types Classes de Mathieux Sozeau [SO08].

Ces travaux datent de l'année 2010. D'autres travaux récents concernent les deux bibliothèques C-CoRN et *Reals*. Nous en ferons référence à la section suivante.

5.3 Perspectives

Les nombres réels dans les systèmes de preuves formels représentent depuis une vingtaine d'années un domaine de recherche en expansion. Cette tentative d'unification d'une partie de ces deux bibliothèques est maintenant à mettre en relation avec les travaux de thèse en cours de Catherine Lelay, les travaux de Bas Spitters, Robbert Krebbers, Russel O'Connor, Cesary Kaliszyk et avec le projet *Univalent Foundation of Mathematics*.

Dans le cadre du projet Coquelicot [3] (dont je suis partie prenante), nous avons proposé un sujet de thèse sur la refonte de la bibliothèque standard de Coq. Catherine Lelay travaille sur ce sujet depuis octobre 2011. Elle s'est intéressée pour le moment à une refonte de l'analyse réelle classique [BLM12] sans toutefois toucher aux fondements de la bibliothèque.

Les travaux de Bas Spitters, Robbert Krebbers, Russel O'Connor et Cesary Kaliszyk [KS11, KO09] se focalisent sur la bibliothèque intuitionniste C-CoRN en étudiant ses fondements. Certains verrous ont été levés, par exemple les relations $<$ et $\#$ sont maintenant à valeur dans `Prop` sans que cela pose de problème du point de vue de l'extraction (utilisation de ConstructiveEpsilon [4] de Coq).

Les travaux de Cyril Cohen sur les nombres réels algébriques (constructifs) [Coh12] sont également une étude très intéressante. Il semble, en effet, que pour un grand nombre de preuves, ces réels soient suffisants.

Dans le cadre du projet *Univalent Foundation of Mathematics* [5] porté par Steve Awodey, Thierry Coquand et Vladimir Voevodsky, l'étude des problématiques spécifiques aux nombres réels est également prévue.

Les récents travaux sur les formalisations des nombres réels représentent une avancée très intéressante. Néanmoins, bien que les différents acteurs soient en contact, la possibilité d'avoir une bibliothèque unique, qui conserverait les avantages des deux précédentes n'est pas encore très claire et les directions explorées restent minces. Les utilisateurs de la bibliothèque classique sont très attachés à l'égalité de Leibniz qui n'est pas adaptée au raisonnement avec des nombres réels intuitionnistes. L'égalité de Leibniz est également mal venue si l'on souhaite appliquer les récents travaux sur la réalisabilité et l'extraction classique [Miq09].

La principale direction que je souhaiterais explorer est donc celle du remplacement de l'égalité de Leibniz par une notion qui satisferait les utilisateurs (plus naturelle que la notion d'*apartness*). Les notions d'ordre total et de totalité de la fonction inverse dépendent du choix qui pourra être fait pour l'égalité. Les autres points tels que la complétude, la hiérarchisation algébrique, les sortes ne représentent pas une véritable difficulté *a priori*.

3. http ://coquelicot.saclay.inria.fr/
4. `http://coq.inria.fr/stdlib/Coq.Logic.ConstructiveEpsilon.html`
5. `http://www.math.ias.edu/sp/univalent`

Bilan et perspectives : récapitulatif

C'est grâce à mon double cursus de mathématiques appliquées et d'informatique théorique que l'idée m'est venue, au début de mon DEA (SPP) en 1996-1997, de faire interagir ces deux domaines. De manière générale, les numériciens qui étudient des problèmes tels que les écoulements de fluide, la propagation des ondes, la prospection, etc., ont, en plus des résolutions mathématiques de ces problèmes (équations dont les solutions exactes ne sont pas connues), également à implanter des algorithmes qui tentent de résoudre ces équations. Ces programmes peuvent s'avérer critiques, par exemple lorsqu'il s'agit de la résistance d'un pont ou de l'écoulement de l'air autour d'une fusée... Ces programmes présentent deux difficultés majeures : d'une part, une programmation correcte (et efficace) est nécessaire et d'autre part, ces programmes sont des programmes numériques, ils utilisent intensivement les nombres à virgule flottante, ce qui peut engendrer des erreurs de calcul et d'approximation conséquentes. Depuis environ 15 ans, je soutiens donc l'idée que les preuves formelles peuvent/doivent apporter une aide afin de garantir la sûreté de ces solutions (tant au niveau mathématiques que informatique).

Les preuves formelles sont relativement récentes au regard des mathématiques mais également au regard de l'informatique et de la logique. Le premier outil de preuve formelle (Automath [6], 1967) fut développé par de Bruijn et existe toujours. C'est un langage formel, basé sur le λ-calcul (Church, 1936) typé, permettant de spécifier les mathématiques et incluant un vérifieur de preuves (proof checker) afin de vérifier sa correction. Il fut le précurseur des systèmes tels que Nuprl [7] (1979) et Coq [8] (1984).

Les travaux traitant directement des applications des méthodes formelles aux programmes d'analyse numérique sont rares. La principale raison est l'utilisation intensive des nombres à virgule flottante, en guise de nombres réels, dans les programmes numériques, alors que les méthodes formelles manipulent plutôt des nombres entiers, ou plus généralement des structures discrètes. Le développement de librairies spécifiant les nombres réels (pas encore flottants) est encore plus récente : Howe dans Nuprl en 1985, Jones dans Lego en 1991, Harrison dans HOL en 1995, Mayero dans Coq en 1997, Dutertre dans PVS en 1997,... Les premiers flottants formalisés, qui doivent s'appuyer sur des réels, datent des années 2000 : Harrison dans HOL en 1999, Boldo dans Coq en 2004,

6. `http://www.win.tue.nl/automath/`
7. `http://www.nuprl.org/`
8. `http://coq.inria.fr/`

il n'y a pas encore de nombres flottants en PVS.

De la même manière, les recherches concernant les interactions entre les preuves formelles et des domaines numériques sont assez rares, effectuées par un petit noyau de chercheurs, une quinzaine au niveau international, dont cinq ou six en France, en augmentation depuis les années 2000.

Les expériences d'interaction que j'ai avec les numériciens sont particulièrement encourageantes et j'ai donc, un peu sur ce même modèle, commencé des interactions avec d'autres domaines tels que l'arithmétique à virgule flottante, la combinatoire algébrique,...

Mon projet se situe dans la continuation de cette optique d'interactions, thème précurseur et novateur, qui apporte des avancés significatives aux domaines concernés. Il s'agit souvent de travaux à moyen, voire à long terme. En effet, aussi bien les outils de preuve formelle (et même parfois la théorie) ne sont pas encore adaptés. À titre d'exemple, formaliser et prouver un problème issu de l'analyse numérique alors qu'il n'y a pas de nombres réels est impossible. Il faut donc commencer par les prérequis, ce qui peut prendre quelques années et soulever des problèmes logiques, comme cela a été (et est encore) justement le cas en Coq pour les nombres réels. Un autre exemple typique est la spécification et la preuve en Coq de l'erreur de méthode et de flottant du schéma numérique à trois points de l'équation des ondes 1D, qui nous (6 membres) a pris 5 ans (avant la première publication).

Preuves formelles de programmes d'éléments finis Je souhaite en particulier poursuivre l'interaction avec les numériciens de l'INRIA Rocquencourt (François Clément, Jean Roberts, ...). Nous pensons maintenant (suite à nos travaux initiés en 2005) être en mesure de nous attaquer à des problèmes qui ne sont plus des jouets mais de vrais programmes sur les éléments finis qui sont eux-mêmes à la pointe de la recherche en analyse numérique. Cela constitue un gros investissement car la méthode des éléments finis, même si elle est bien maîtrisée mathématiquement, implique un gros travail de formalisation et de preuve formelle pour être utilisable sur des exemples. Il est en effet important de montrer l'utilisabilité de ces méthodes sur un exemple de niveau recherche en analyse numérique. Enfin, les logiciens ne souhaiteront probablement pas prouver les programmes de calcul scientifique dans les prochaines décennies et il faut donc donner aux numériciens des outils pour les prouver eux-mêmes. Nous visons donc la création d'une boîte à outils utilisable par des non-spécialistes et qui donne des garanties fortes de correction des programmes.

Comme énonce ci-avant, ces preuves requièrent une utilisation intensive des nombres flottants et des nombres réels sur lesquels les flottants s'appuient. la spécification et la formalisation des nombres réels dans un assistant de preuves tel que Coq soulèvent des problèmes théoriques majeurs. Je pense poursuivre dans la voie décrite au chapitre 5 "Formalisations des nombres réels en Coq".

Approximation Polynomiale Rigoureuse En collaboration avec les membres de l'EPI INRIA Aric de l'ENS de Lyon et de l'EPI Marelle de l'INRIA Sophia-Antipolis, je souhaite également poursuivre les travaux concernant les RPAs. Une suite logique consiste en l'étude de la correspondance entre les fonction de bases et le modèle d'approximation utilisé. Cela peut être fait de

manière manuelle, à l'aide d'une relation de récurrence à partir de la définition formelle. Mais il existe une approche plus générique à explorer, celle du DDMF (Dynamic Dictionary of Mathematical Functions).

Les modèles de Taylor représentent un premier exemple de preuve formelle de pires cas. L'étape suivante naturelle est de le coupler avec des tests de positivité des polynômes, par exemple en utilisant la technique des sommes de carrés (sums-of-squares), ce qui nous donnera une technique automatique de vérification formelle d'approximation polynomiale ainsi qu'une nouvelle évaluation de l'erreur d'approximation. Si cette erreur s'avère encore trop large, les modèles de Chebyshev pourront remplacer avantageusement les modèles de Taylor. ce remplacement devrait pouvoir se faire à moindre coût grâce à nos choix initiaux pour une interface générique par rapport à la base de polynômes.

Autres interactions En ce qui concerne la suite de la preuve du logiciel SCHUR, l'étape à venir que nous nous proposons de traiter concerne la preuve d'une fonction purement énumérative, le calcul de coefficient de Littlewood–Richardson. Cet objectif représente une étape bien plus ardue. Pour prouver le programme qui calcule ces coefficients, il faut arriver à formaliser les propriétés qui les caractérisent. Or, de par la composante énumérative de la fonction, cette spécification formelle peut s'avérer assez délicate. Cet objectif représente donc un défi tant au niveau preuve de programmes qu'au niveau des spécifications mathématiques.

Il y a des interactions qui me semblent également très peu explorées et dont les résultats pourraient se révéler intéressants. Il s'agit de la formalisation, au sein d'un système formel, de certains systèmes logiques. Cela a par exemple été fait pour Coq (Coq-in-Coq de Bruno Barras). Les systèmes de formalisation de preuves sont naturellement liés au λ-calcul et à la théorie de la démonstration. Toutefois très peu de travaux à ce jour ont été fait pour relier les théories logiques des assistants de preuves (comme Coq) à la logique linéaire. C'est une direction nouvelle que j'aimerais bien explorer.

Bibliographie

[Abr96] J.-R. Abrial. *The B-book : Assigning programms to meanings.* 1996.

[AFG+11] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of sat/smt solvers to coq through proof witnesses. In *CPP*, pages 135–150, 2011.

[AMI] CPN-AMI *: Home Page.* `http://www-src.lip6.fr/logiciels/mars/CPNAMI/`.

[And84] George E. Andrews. *The theory of partitions.* Cambridge University Press, 1984.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development : Coq'Art : The Calculus of Inductive Constructions.* EATCS, Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BCF+] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL : ANSI/ISO C Specification Language.* `http://frama-c.cea.fr/download/acsl-implementation-Beryllium-20090902.pdf`.

[BCF+10] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Formal Proof of a Wave Equation Resolution Scheme : the Method Error. In *Proceedings of ITP 2010 (Interactive Theorem Proving)*, pages 147–162. Springer-Verlag LNCS, 2010.

[BCF+12] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave equation numerical resolution : a comprehensive mechanized proof of a c program. *J. Autom. Reasoning*, 2012. To appear.

[BDG11] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In *CPP*, pages 362–377, 2011.

[BF07] Sylvie Boldo and Jean-Christophe Filliâtre. Formal Verification of Floating-Point Programs. In *18th IEEE International Symposium on Computer Arithmetic*, pages 187–194, Montpellier, France, June 2007.

[BFM09] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining Coq and Gappa for certifying floating-point programs. In Jacques Carette, Lucas Dixon, Claudio Sarcedoti Coen, and

Stephen M. Watt, editors, *16th Calculemus Symposium*, volume 5625 of *Lecture Notes in Artificial Intelligence*, pages 59–74, Grand Bend, ON, Canada, 2009.

[BGP05] Jonathan Billington, Guy Edward Gallasch, and Laure Petrucci. Verification of the class of stop-and-wait protocols modelled by coloured petri nets. *Nord. J. Comput.*, 12(3) :251–274, 2005.

[BHMT10] Franck Butelle, Florent Hivert, Micaela Mayero, and Frédéric Toumazet. Formal Proof of SCHUR Conjugate Function. In *Proceedings of Calculemus 2010*, pages 158–171. Springer-Verlag LNAI, 2010.

[Bis67] Errett Bishop. Foundations of constructive analysis. In *New York : Academic Press*, 1967.

[BJMD+12] Nicolas Brisebarre, Mioara Joldes, Erik Martin-Dorel, Micaela Mayero, Jean-Michel Muller, Ioana Pasca, Laurence Rideau, and Laurent Théry. Rigorous Polynomial Approximations using Taylor Models in Coq. In *Proceedings of NFM 2012 (Nasa Formal Methods)*, volume 7226, pages 85–99. Springer-Verlag LNCS, 2012.

[BLM12] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Improving real analysis in coq : a user-friendly approach to integrals and derivatives. In *CPP*, 2012. to appear.

[BM11] Sylvie Boldo and Guillaume Melquiond. Flocq : A unified library for proving floating-point algorithms in coq. In *IEEE Symposium on Computer Arithmetic*, pages 243–252, 2011.

[Bol09] Sylvie Boldo. Floats & Ropes : a case study for formal numerical program verification. In *36th International Colloquium on Automata, Languages and Programming*, volume 5556 of *LNCS - ARCoSS*, pages 91–102, Rhodos, Greece, July 2009. Springer.

[Bou66] N. Bourbaki. *Éléments de mathématique. Théorie des ensembles.* Hermann, 1966.

[BPR06] S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in real algebraic geometry.* 2nd edition, Springer, 2006.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19th *International Conference on Comp uter Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. `http://cs.nyu.edu/acsys/cvc3` Release 20091011.

[CCBL] Sylvain Conchon, Evelyne Contejean, François Bobot, and Stéphane Lescuyer. *Alt-Ergo is an automatic theorem prover dedicated to progra m verification.* `http://ergo.lri.fr`. Release 0.9.

[CCPg] Loïc Correnson, Pascal Cuoq, Armand Puccetti, and Julien Si gnoles. *Frama-C User Manual, Beryllium release.* `http://frama-c.cea.fr`.

[CDT] The Coq Development Team. *The Coq Proof Assistant Reference Manual.* INRIA-Rocquencourt.

[CFL06] Luís Cruz-Filipe and Pierre Letouzey. A large-scale experiment in executing extracted programs. *Electr. Notes Theor. Comput. Sci.*, 151(1) :75–91, 2006.

[CFS03] Luís Cruz-Filipe and Bas Spitters. Program extraction from large proof developments. In *TPHOLs*, pages 205–220, 2003.

[CG06] Alberto Ciaffaglione and Pietro Di Gianantonio. A certified, corecursive implementation of exact real numbers. *Theor. Comput. Sci.*, 351(1) :39–51, 2006.

[CJL10] Sylvain Chevillard, Mioara Joldes, and Christoph Quirin Lauter. Sollya : An environment for the development of numerical codes. In *ICMS*, pages 28–31, 2010.

[CMP08] Christine Choppy, Micaela Mayero, and Laure Petrucci. Experimenting formal proofs of petri nets refinements. In *Proceedings of REFINE 2008, Electr. Notes Theor. Comput. Sci.*, volume 214, pages 231–254, 2008.

[CMP09] Christine Choppy, Micaela Mayero, and Laure Petrucci. Coloured Petri net refinement specification and correctness proof with Coq. In *Proceedings of NFM09*, 2009.

[CMP10] Christine Choppy, Micaela Mayero, and Laure Petrucci. Coloured Petri net refinement specification and correctness proof with Coq. volume 6, pages 195–202, 2010.

[Coh12] Cyril Cohen. Construction of real algebraic numbers in coq. In *ITP*, pages 67–82, 2012.

[Coq85] Thierry Coquand. *Une théorie des constructions.* Thèse de Doctorat, Université Paris 7, 1985.

[CPN] *cpntools.* `http://wiki.daimi.au.dk/cpntools/cpntools.wiki`.

[DM01] David Delahaye and Micaela Mayero. `Field` : une procédure de décision pour les nombres réels en Coq. In *Journées Francophones des Langages Applicatifs, Pontarlier.* INRIA, Janvier 2001.

[DM05] David Delahaye and Micaela Mayero. Dealing with Algebraic Expressions over a Field in Coq using Maple. In *Journal of Symbolic Computation : special issue on the integration of automated reasoning and computer algebra systems*, volume 39, pages 569–592, 2005.

[DM06] David Delahaye and Micaela Mayero. Quantifier Elimination over Algebraically Closed Fields in a Proof Assistant using a Computer Algebra System. In *Proceedings of Calculemus 2005*, volume 151(1) of *ENTCS*, pages 57–73, 2006.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify : a theorem prover for program checking. *J. ACM*, 52(3) :365–473, 2005. Release 1.5.4.

[Dut96] Bruno Dutertre. Elements of mathematical analysis in PVS. In *Proc. TPHOL 96*, volume 1125. Springer LNCS, 1996.

[EPE96] A. Edalat, P. Potts, and M. Escardo. Semantics of Exact Real Arithmetic. In *Proc. LICS 97.* Springer LNCS, 1996.

[FG01] G. Chavent F.Clément and S. Gómez. Migration-based traveltime waveform inversion of 2-d simple structures : A synthetic example. *Geophysics*, 66 :845–860, 2001.

[FHL+07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR : A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2), June 2007.

[FM04] A. Cartalade F.Clément, N. Khvoenkova and P. Montarnal. Analyse de sensibilité et estimation de paramètres de transport pour une équation de diffusion, approche par état adjoint. Technical report, INRIA, 2004.

[FMM+] Jean-Christophe Filliâtre, Claude Marché, Yannick Moy, Thierry Hubert, and Nicolas Rousset. *Why is a software verification platform.* `http://why.lri.fr`. Release 2.21.

[GL02] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *ICFP*, pages 235–246, 2002.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[GMT08] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2008.

[GN00] Herman Geuvers and Milad Niqui. Constructive reals in coq : Axioms and categoricity. In *TYPES*, pages 79–95, 2000.

[Got00] Hanne Gottliebsen. Transcendental Functions and Continuity Checking in PVS. In *Proc. TPHOL 2000.* Springer LNCS, Septembre 2000.

[GPT08] Benjamin Grégoire, Loïc Pottier, and Laurent Théry. Proof certificates for algebra and their application to automatic geometry theorem proving. In *Automated Deduction in Geometry*, pages 42–59, 2008.

[GWZ00] Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. A constructive proof of the fundamental theorem of algebra without using the rationals. In *TYPES*, pages 96–111, 2000.

[Har98] John Harrison. *Theorem Proving with the Real Numbers.* Springer-Verlag, 1998.

[Her12] Hugo Herbelin. A constructive proof of dependent choice, compatible with classical logic. In *LICS*, pages 365–374, 2012.

[How86] Douglas J. Howe. *Implementing Analysis.* PhD thesis, Cornell University, 1986.

[HT09] Osman Hasan and Sofiène Tahar. Performance analysis and functional verification of the stop-and-wait protocol in hol. *J. Autom. Reasoning*, 42(1) :1–33, 2009.

[Jac41] Carl Gustav Jacob Jacobi. De functionibus alternantibus earumque divisione per productum e differentiis elementorum conflatum. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 22 :360–371, 1841. Reprinted in Gesammelten Werke III, G. Reimer, Berlin, 1884.

[Jol11] Mioara Joldes. *Rigourous Polynomial Approximations and Applications.* PhD thesis, Ecole Normale Supérieure de Lyon, septembre 2011.

[KO09] Cezary Kaliszyk and Russell O'Connor. Computing with classical real numbers. *Journal of Formalized Reasoning*, 2(1) :27–39, 2009.

[KS11] Robbert Krebbers and Bas Spitters. Computer certified efficient exact reals in coq. In *Calculemus/MKM*, pages 90–106, 2011.

[Let03] Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *2nd International Workshop on Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *LNCS*, Berg en Dal, Netherlands, 2003. Springer.

[Lew02] Glenn Lewis. *Incremental specification and analysis in the context of coloured Petri nets.* PhD thesis, University of Hobart, Tasmania, 2002.

[Lit50] Dudley Ernest Littlewood. *The Theory of Group Characters.* Oxford University Press, 1950. second edition.

[LL01] C. Lakos and G. Lewis. Incremental state space construction of coloured Petri nets. volume 2075, pages 263–282, 2001.

[LM01] Vincent Lefèvre and Jean-Michel Muller. Worst cases for correct rounding of the elementary functions in double precision. In *IEEE Symposium on Computer Arithmetic*, pages 111–118, 2001.

[LP88] A. Lascoux and P. Pragacz. S-function series. *J. Phys. A : Math. Gen.*, 21 :4105–4118, 1988.

[May99] Micaela Mayero. The Three Gap Theorem : Specification and Proof in Coq. Technical Report 3848, INRIA, 1999.

[May00] Micaela Mayero. The Three Gap Theorem (steinhauss conjecture). In *Proceedings of TYPES'99*, volume 1956, pages 162–173. Springer-Verlag LNCS, 2000.

[May01] Micaela Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique.* PhD thesis, Université Paris VI, décembre 2001.

[MBdD+10] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic.* Birkhäuser, 2010.

[MD12] Erik Martin-Dorel. *Contributions à la Vérification Formelle d'Algorithmes Arithmétiques.* PhD thesis, Ecole Normale Supérieure de Lyon, septembre 2012.

[Mic] Microsoft Research. *Z3 An Efficient SMT Solver.* `http://research.microsoft.com/en-us/um/redmond/projects/z3`. Release 2.4.

[Miq01] A. Miquel. *Le calcul des constructions implicite : syntaxe et sémantique.* Thèse de Doctorat, Université Paris 7, Décembre 2001.

[Miq07] Alexandre Miquel. Classical program extraction in the calculus of constructions. In *CSL*, pages 313–327, 2007.

[Miq09] A. Miquel. *De la formalisation des preuves à l'extraction de programmes.* Habilitation à diriger des recherches, Université Paris 7, Décembre 2009.

[MM] Claude Marché and Yannick Moy. *Jessie Tutorial.* `http://frama-c.cea.fr/jessie/jessie-tutorial.pdf`. Release 2.21.

[MM94] Valérie Ménissier-Morain. *Arithmétique exacte. Conception, algorithme et performances d'une implémentation informatique en précision arbitraire.* Thèse de Doctorat, Université Paris VII, Décembre 1994.

[MnM01] César Muñoz and Micaela Mayero. Real Automation in the Field. Technical Report NASA/CR-2001-211271 Report 39, ICASE Nasa-Langley Research Center, December 2001.

[Mul89] J.-M. Muller. *Arithmétique des ordinateurs.* Masson, 1989.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[O'C08] Russell O'Connor. Certified exact transcendental real number computation in coq. In *TPHOLs*, pages 246–261, 2008.

[Pet66] Carl Adam Petri. Communication with automata. Technical report, Griffiss Air Force Base, 1966. Traduction en Anglais de sa thèse : *Kommunikation mit Automaten, Rheinisch-Westfäliches Institut für Intrumentelle Mathematik an der Universität Bonn, Schrift Nr 2, 1962.*

[PM96] C. Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur.* Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[Rio91] Renaud Rioboo. *Quelques aspects du calcul exact avec les nombres réels.* Thèse de Doctorat, Université Paris 6, Février 1991.

[SLZ03] Damien Stehlé, Vincent Lefèvre, and Paul Zimmermann. Worst cases and lattice reduction. In *IEEE Symposium on Computer Arithmetic*, pages 142–147, 2003.

[SO08] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *TPHOLs*, pages 278–293, 2008.

[SORgC] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Strin ger Calvert. *PVS prover guide.* `http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf`.

[Vui90] J. Vuillemin. Exact Real Computer Arithmetic with Continued Fractions. In *IEEE Transactions on computers 39*, volume 8, pages 1087–1105, 1990.

[Wer94] B. Werner. *Une théorie des constructions inductives.* Thèse de Doctorat, Université Paris VII, Mai 1994.

Annexes

Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Wave Equation Numerical Resolution: a Comprehensive Mechanized Proof of a C Program

Sylvie Boldo · François Clément · Jean-Christophe Filliâtre · Micaela Mayero · Guillaume Melquiond · Pierre Weis

Received: date / Accepted: date

Abstract We formally prove correct a C program that implements a numerical scheme for the resolution of the one-dimensional acoustic wave equation. Such an implementation introduces errors at several levels: the numerical scheme introduces method errors, and floating-point computations lead to round-off errors. We annotate this C program to specify both method error and round-off error. We use Frama-C to generate theorems that guarantee the soundness of the code. We discharge these theorems using SMT solvers, Gappa, and Coq. This involves a large Coq development to prove the adequacy of the C program to the numerical scheme and to bound errors. To our knowledge, this is the first time such a numerical analysis program is fully machine-checked.

Keywords Formal proof of numerical program · Convergence of numerical scheme · Proof of C program · Coq formal proof · Acoustic wave equation · Partial differential equation · Rounding error analysis

This research was supported by the ANR projects CerPAN (ANR-05-BLAN-0281-04) and F∮st (ANR-08-BLAN-0246-01).

S. Boldo · G. Melquiond · J.-C. Filliâtre
INRIA Saclay – Île-de-France, ProVal, Orsay cedex, F-91893

J.-C. Filliâtre · S. Boldo · G. Melquiond
LRI, UMR 8623, Université Paris-Sud, CNRS, Orsay cedex, F-91405

F. Clément · P. Weis
INRIA Paris – Rocquencourt, Estime, Le Chesnay cedex, F-78153

M. Mayero
LIPN, UMR 7030, Université de Paris-Nord, CNRS, Villetaneuse, F-93430
LIP, Arénaire (INRIA Grenoble - Rhône-Alpes, CNRS UMR 5668, UCBL, ENS Lyon), Lyon, F-69364

1 Introduction

Ordinary differential equations (ODE) and partial differential equations (PDE) are ubiquitous in engineering and scientific computing. They show up in nuclear simulation, weather forecast, and more generally in numerical simulation, including block diagram modelization. Since solutions to nontrivial problems are non-analytic, they must be approximated by numerical schemes over discrete grids.

Numerical analysis is a part of applied mathematics that is mainly interested in proving the *convergence* of these schemes [22], that is, proving that approximation quality increases as the size of discretization steps decreases. The approximation quality represents the distance between the exact continuous solution and the approximated discrete solution; this distance must tend toward zero in order for the numerical scheme to be useful.

A numerical scheme is typically proved to be convergent with pen and paper. This is a difficult, time-consuming, and error-prone task. Then the scheme is implemented as a C/C++ or Fortran program. This introduces new issues. First, we must ensure that the program correctly implements the scheme and is immune from runtime errors such as out-of-bounds accesses or overflows. Second, the program introduces round-off errors due to floating-point computations and we must prove that those errors could not lead to irrelevant results. Typical pen-and-paper proofs do not address floating-point nor runtime errors. Indeed the huge number of proof obligations, and their complexity, make the whole process almost intractable. However, with the help of mechanized program verification, such a proof becomes feasible. In the first place, because automated theorem provers can alleviate the proof burden. More importantly, because the proof is guaranteed to cover all aspects of the verification.

Our case study. We consider the acoustic wave equation in an one-dimensional space domain. The equation describes the propagation of pressure variations (or sound waves) in a fluid medium; it also models the behavior of a vibrating string. Among the wide variety of numerical schemes to approximate the 1D acoustic wave equation, we choose the simplest one: the second order centered finite difference scheme, also known as *three-point scheme*. To keep it simple, we assume an homogeneous medium (the propagation velocity is constant) and we consider discretization over regular grids with constant discretization steps for time and space. Our goal is to prove the correctness of a C program implementing this scheme.

Method and tools. We use the Jessie plug-in of Frama-C [43,32] to perform the deductive verification of this C program. The source code is augmented with *ACSL annotations* [6] to describe its formal specification. When submitted to Frama-C, proof obligations are generated. Once these theorems are proved, the program is guaranteed to satisfy its specification and to be free from runtime errors. Part of the proof obligations are discharged by automated provers, *e.g.* Alt-Ergo [10], CVC3 [5], Gappa [25], and Z3 [47]. The more complicated ones, such as the one related to the convergence of the numerical scheme, cannot be proved automatically. These obligations were manually proved with the Coq [8,20] interactive proof assistant. In the end, we have formally verified all the properties of the C program. To our knowledge, this is the first time this kind of verification is machine-checked.

The annotated C program and the Coq sources of the formal development are available from

http://fost.saclay.inria.fr/wave_total_error.html

State of the art. There is an abundant literature about the convergence of numerical schemes, *e.g.* see [56,58]. In particular, the convergence of the three-point scheme for the wave equation is well-known and taught relatively early [7]. Unfortunately, no article goes into all the details needed for a formal proof. These mathematical "details" may have been skipped for readability, but some mandatory details may have also been omitted due to oversights.

In the fields of automatic provers and proof assistants, few works have been done for the formalization and mechanical proofs of mathematical analysis, and even fewer works for numerical analysis. The first developments on real numbers and real analysis are from the late 90's, in systems such as ACL2 [33], Coq [44], HOL Light [35], Isabelle [31], Mizar [54], and PVS [29]. An extensive work has been done by Harrison regarding $\mathbb{R}^n$ and the dot product [36]. Constructive real analysis [34,24,38] and further developments in numerical analysis [49,50] have been carried out at Nijmegen. We can also mention the formal proof of an automatic differentiation algorithm [45].

As explained by Rosinger in 1985, old methods to bound round-off errors were based on "unrealistic linearizing assumptions" [51]. Further work was done under more realistic assumptions about round-off errors [51,52], but none of these assumptions were proved correct with respect to the numerical schemes. As Roy and Oberkampf, we believe that round-off errors should not be treated as random variables and that traditional statistical methods should not be used [53]. They propose the use of interval arithmetic or increased precision to control accuracy. Note that their example of hypersonic nozzle flow is converging so fast that round-off errors can be neglected [53]. Interval arithmetic can also take method error into account [55]. The final interval is then claimed to contain the exact solution. This is not formally proved, though. Additionally, the width of the final interval can be quite large.

There are other tools to bound round-off errors not dedicated to numerical schemes. Some successful approaches are based on abstract interpretation [23,27]. In our case, they are of little help, since there is a complex phenomenon of error compensation during the computations. Ignoring this compensation would lead to bounds on round-off errors growing as fast as $O(2^k)$ (k being the number of time steps). That is why we had to thoroughly study the propagation of round-off errors in this numerical scheme to get tighter bounds. It also means that most of the proofs had to be done by hand to achieve this part of the formal verification.

Outline. Section 2 presents the PDE, the numerical scheme, and their mathematical properties. Section 3 is devoted to the proofs of the convergence of the numerical scheme and the upper bound for the round-off error. Finally, Section 4 describes the formalization, *i.e.* the tools used, the annotated C program, and the mechanized proofs.

2 Numerical Scheme for the Wave Equation

A partial differential equation (PDE) modeling an evolution problem is an equation involving partial derivatives of an unknown function of several independent space and time variables. The uniqueness of the solution is obtained by imposing initial conditions, i.e. values of the function and some of its derivatives at initial time. The problem of the vibrating string tied down at both ends, among many other physical problems, is formulated as an *initial-boundary value problem* where the boundary conditions are additional constraints set on the boundary of the supposedly bounded domain [56].

This section, as well as the steps taken at Section 3.1 to conduct the convergence proof of the numerical scheme, is inspired by [7].

2.1 The Continuous Equation

The chosen PDE models the propagation of waves along an ideal vibrating elastic string that is tied down at both ends, see [1,18], see also Figure 1. The PDE is obtained from Newton's laws of motion [48].

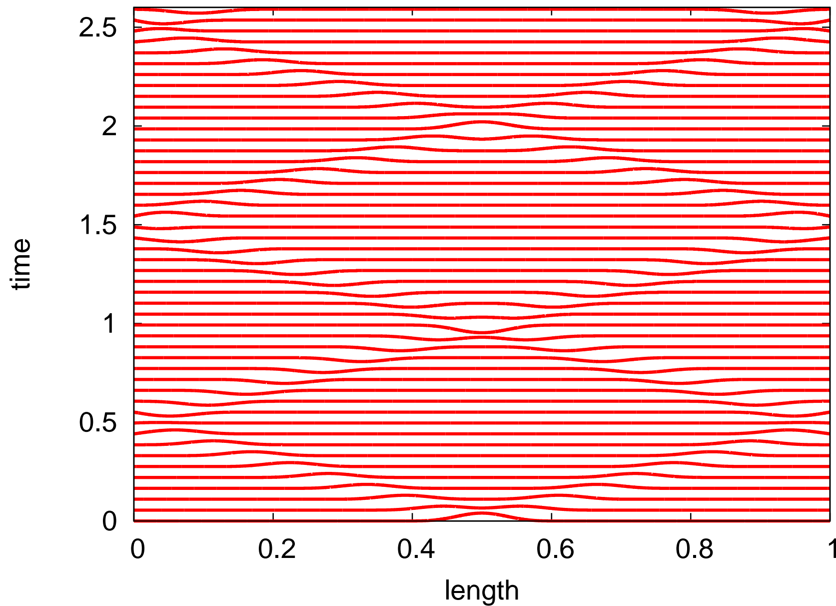


Fig. 1 Space-time representation of the signal propagating along a vibrating string. Each curve represents the string at a different time step.

The gravity is neglected, so the string is supposed rectilinear when at rest. Let $x_{\min}$ and $x_{\max}$ be the abscissas of the extremities of the string. Let $\Omega =$

$[x_{\min}, x_{\max}]$ be the bounded space domain. Let $p(x,t)$ be the transverse displacement of the point of the string of abscissa x at time t from its equilibrium position; it is a (signed) scalar. Let c be the constant propagation velocity; it is a positive number that depends on the section and density of the string. Let $s(x,t)$ be the external action on the point of abscissa x at time t; it is a source term, such that $t = 0 \Rightarrow s(x,t) = 0$. Finally, let $p_0(x)$ and $p_1(x)$ be the initial position and velocity of the point of abscissa x. We consider the initial-boundary value problem

$$\forall t \geq 0,\ \forall x \in \Omega,\ \ (L(c)\, p)(x,t) \stackrel{\text{def}}{=} \frac{\partial^2 p}{\partial t^2}(x,t) + A(c)\, p(x,t) = s(x,t), \tag{1}$$

$$\forall x \in \Omega,\ \ (L_1\, p)(x,0) \stackrel{\text{def}}{=} \frac{\partial p}{\partial t}(x,0) = p_1(x), \tag{2}$$

$$\forall x \in \Omega,\ \ (L_0\, p)(x,0) \stackrel{\text{def}}{=} p(x,0) = p_0(x), \tag{3}$$

$$\forall t \geq 0,\ \ p(x_{\min},t) = p(x_{\max},t) = 0 \tag{4}$$

where the differential operator $A(c)$ is defined by

$$A(c) \stackrel{\text{def}}{=} -\, c^2 \frac{\partial^2}{\partial x^2}. \tag{5}$$

This simple partial derivative equation happens to possess an analytical solution given by the so-called d'Alembert's formula [39], obtained from the method of characteristics and the image theory [37], $\forall t \geq 0$, $\forall x \in \Omega$,

$$p(x,t) = \frac{1}{2}(\tilde{p}_0(x-ct) + \tilde{p}_0(x+ct)) + \frac{1}{2c}\int_{x-ct}^{x+ct} \tilde{p}_1(y)dy + \frac{1}{2c}\int_0^t \left(\int_{x-c(t-\sigma)}^{x+c(t-\sigma)} \tilde{s}(y,\sigma)dy \right) d\sigma \tag{6}$$

where the quantities $\tilde{p}_0$, $\tilde{p}_1$, and $\tilde{s}$ are respectively the successive antisymmetric extensions in space of p_0, p_1, and s defined on Ω to the entire real axis $\mathbb{R}$.

We have formally verified d'Alembert's formula as a separate work [41]. But for the purpose of the current work, we just admit that under reasonable conditions on the Cauchy data p_0 and p_1 and on the source term s, there exists a unique solution p to the initial-boundary value problem (1)–(4) for each $c > 0$. Simply supposing the existence of a solution instead of exhibiting it, opens the way to scale our approach to more general cases for which there is no known analytic expression of a solution, *e.g.* in the case of a nonuniform propagation velocity c.

For such a solution p, it is natural to associate at each time t the positive definite quadratic quantity

$$E(c)(p)(t) \stackrel{\text{def}}{=} \frac{1}{2}\left\| \left(x \mapsto \frac{\partial p}{\partial t}(x,t) \right) \right\|^2 + \frac{1}{2}\left\| (x \mapsto p(x,t)) \right\|^2_{A(c)} \tag{7}$$

where $\langle q, r \rangle \stackrel{\text{def}}{=} \int_\Omega q(x) r(x) dx$, $\|q\|^2 \stackrel{\text{def}}{=} \langle q, q \rangle$ and $\|q\|^2_{A(c)} \stackrel{\text{def}}{=} \langle A(c)\, q, q \rangle$. The first term is interpreted as the kinetic energy, and the second term as the potential energy, making E the mechanical energy of the vibrating string.

2.2 The Discrete Equations

Let $i_{\max}$ be the positive number of intervals of the space discretization. Let the space discretization step Δx and the discretization function $i_{\Delta x}$ be defined as

$$\Delta x \stackrel{\text{def}}{=} \frac{x_{\max} - x_{\min}}{i_{\max}} \quad \text{and} \quad i_{\Delta x}(x) \stackrel{\text{def}}{=} \left\lfloor \frac{x - x_{\min}}{\Delta x} \right\rfloor .$$

Let us consider the time interval $[0, t_{\max}]$. Let $\Delta t \in]0, t_{\max}[$ be the time discretization step. We define

$$k_{\Delta t}(t) \stackrel{\text{def}}{=} \left\lfloor \frac{t}{\Delta t} \right\rfloor \quad \text{and} \quad k_{\max} \stackrel{\text{def}}{=} k_{\Delta t}(t_{\max}).$$

Now, the compact domain $\Omega \times [0, t_{\max}]$ is approximated by the regular discrete grid defined by

$$\forall k \in [0..k_{\max}],\ \forall i \in [0..i_{\max}], \quad \mathbf{x}_i^k \stackrel{\text{def}}{=} (x_i, t^k) \stackrel{\text{def}}{=} (x_{\min} + i\Delta x, k\Delta t). \qquad (8)$$

For a function q defined over $\Omega \times [0, t_{\max}]$ (resp. Ω), the notation q_{h} denotes any discrete approximation of q at the points of the grid, *i.e.* a discrete function over $[0..i_{\max}] \times [0..k_{\max}]$ (resp. $[0..i_{\max}]$). By extension, the notation q_{h} is also a shortcut to denote the matrix $(q_i^k)_{0 \le i \le i_{\max}, 0 \le k \le k_{\max}}$ (resp. the vector $(q_i)_{0 \le i \le i_{\max}}$). The notation $\bar{q}_{\text{h}}$ is reserved to the approximation defined on $[0..i_{\max}] \times [0..k_{\max}]$ by

$$\bar{q}_i^k \stackrel{\text{def}}{=} q(\mathbf{x}_i^k) \quad (\text{resp. } \bar{q}_i \stackrel{\text{def}}{=} q(x_i)).$$

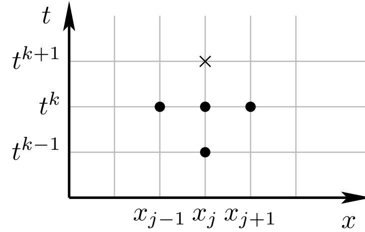


Fig. 2 Three-point scheme: p_i^{k+1} (at $\times$) depends on p_{i-1}^k, p_i^k, p_{i+1}^k, and p_i^{k-1} (at $\bullet$).

Let $p_{0\text{h}}$ and $p_{1\text{h}}$ be two discrete functions over $[0..i_{\max}]$. Let s_{h} be a discrete function over $[0..i_{\max}] \times [0..k_{\max}]$. Then, the discrete function p_{h} over $[0..i_{\max}] \times [0..k_{\max}]$ is said to be the solution of the three-point[1] finite difference scheme, as illustrated in Figure 2, when the following set of equations holds:

$$\forall k \in [2..k_{\max}],\ \forall i \in [1..i_{\max} - 1],$$
$$(L_{\text{h}}(c)\, p_{\text{h}})_i^k \stackrel{\text{def}}{=} \frac{p_i^k - 2p_i^{k-1} + p_i^{k-2}}{\Delta t^2} + (A_{\text{h}}(c)\,(i' \mapsto p_{i'}^{k-1}))_i = s_i^{k-1}, \qquad (9)$$

[1] In the sense "three spatial points", for the definition of matrix $A_{\text{h}}(c)$.

$$\forall i \in [1..i_{\max}-1],\ \ (L_{1\mathrm{h}}(c)\,p_{\mathrm{h}})_i \stackrel{\mathrm{def}}{=} \frac{p_i^1 - p_i^0}{\Delta t} + \frac{\Delta t}{2}(A_{\mathrm{h}}(c)\,(i' \mapsto p_{i'}^0))_i = p_{1,i}, \quad (10)$$

$$\forall i \in [1..i_{\max}-1],\ \ (L_{0\mathrm{h}}\,p_{\mathrm{h}})_i \stackrel{\mathrm{def}}{=} p_i^0 = p_{0,i}, \quad (11)$$

$$\forall k \in [0..k_{\max}],\ \ p_0^k = p_{i_{\max}}^k = 0 \quad (12)$$

where the matrix $A_{\mathrm{h}}(c)$, a discrete analog of $A(c)$, is defined for any vector q_{h}, by

$$\forall i \in [1..i_{\max}-1], \quad (A_{\mathrm{h}}(c)\,q_{\mathrm{h}})_i \stackrel{\mathrm{def}}{=} -c^2 \frac{q_{i+1} - 2q_i + q_{i-1}}{\Delta x^2}. \quad (13)$$

A discrete analog of the energy is also defined by[2]

$$E_{\mathrm{h}}(c)(p_{\mathrm{h}})^{k+\frac{1}{2}} \stackrel{\mathrm{def}}{=} \frac{1}{2}\left\|\left(i \mapsto \frac{p_i^{k+1} - p_i^k}{\Delta t}\right)\right\|_{\Delta x}^2 + \frac{1}{2}\left\langle (i \mapsto p_i^k), (i \mapsto p_i^{k+1}) \right\rangle_{A_h(c)} \quad (14)$$

where, for any vectors q_{h} and r_{h},

$$\langle q_{\mathrm{h}}, r_{\mathrm{h}} \rangle_{\Delta x} \stackrel{\mathrm{def}}{=} \sum_{i=0}^{i_{\max}} q_i r_i \Delta x, \qquad \|q_{\mathrm{h}}\|_{\Delta x}^2 \stackrel{\mathrm{def}}{=} \langle q_{\mathrm{h}}, q_{\mathrm{h}} \rangle_{\Delta x},$$
$$\langle q_{\mathrm{h}}, r_{\mathrm{h}} \rangle_{A_h(c)} \stackrel{\mathrm{def}}{=} \langle A_{\mathrm{h}}(c)\,q_{\mathrm{h}}, r_{\mathrm{h}} \rangle_{\Delta x}, \qquad \|q_{\mathrm{h}}\|_{A_h(c)}^2 \stackrel{\mathrm{def}}{=} \langle q_{\mathrm{h}}, q_{\mathrm{h}} \rangle_{A_h(c)}.$$

Note that the three-point scheme is parameterized by the discrete Cauchy data $p_{0\mathrm{h}}$ and $p_{1\mathrm{h}}$, and by the discrete source term s_{h}. Of course, when these discrete inputs are respectively approximations of the continuous functions p_0, p_1, and s (*e.g.* when $p_{0\mathrm{h}} = \bar{p}_{0\mathrm{h}}$, $p_{1\mathrm{h}} = \bar{p}_{1\mathrm{h}}$, and $s_{\mathrm{h}} = \bar{s}_{\mathrm{h}}$), then the discrete solution p_{h} is an approximation of the continuous solution p.

2.3 Convergence

Let ξ be in $]0,1[$. The CFL(ξ) condition (for Courant-Friedrichs-Lewy, see [22]) states that the discretization steps satisfy the relation

$$\frac{c\Delta t}{\Delta x} \le 1 - \xi. \quad (15)$$

The convergence error e_{h} measures the distance between the continuous and discrete solutions. It is defined by

$$\forall k \in [0..k_{\max}], \forall i \in [0..i_{\max}], \quad e_i^k \stackrel{\mathrm{def}}{=} \bar{p}_i^k - p_i^k. \quad (16)$$

Note that when $p_{0\mathrm{h}} = \bar{p}_{0\mathrm{h}}$, then for all i, $e_i^0 = 0$. The truncation error ε_{h} measures at which precision the continuous solution satisfies the numerical scheme. It is defined for $k \in [2..k_{\max}]$ and $i \in [1..i_{\max}-1]$ by

$$\varepsilon_i^k \stackrel{\mathrm{def}}{=} (L_{\mathrm{h}}(c)\,\bar{p}_{\mathrm{h}})_i^k - \bar{s}_i^{k-1}, \quad (17)$$
$$\varepsilon_i^1 \stackrel{\mathrm{def}}{=} (L_{1\mathrm{h}}(c)\,\bar{p}_{\mathrm{h}})_i - \bar{p}_{1,i}, \quad (18)$$
$$\varepsilon_i^0 \stackrel{\mathrm{def}}{=} (L_{0\mathrm{h}}\bar{p}_{\mathrm{h}})_i - \bar{p}_{0,i}. \quad (19)$$

[2] By convention, the energy is defined between steps k and $k+1$, hence the notation $k + \frac{1}{2}$.

Again, note that when $p_{0\mathrm{h}} = \bar{p}_{0\mathrm{h}}$ and $p_{1\mathrm{h}} = \bar{p}_{1\mathrm{h}}$, then for all i, $\varepsilon_i^0 = 0$ and $\varepsilon_i^1 = e_i^1/\Delta t$. Furthermore, when there is also $s_\mathrm{h} = \bar{s}_\mathrm{h}$, then the convergence error e_h is itself solution of the same numerical scheme with inputs defined by, for all i, k,

$$p_{0,i} = \varepsilon_i^0 = 0, \quad p_{1,i} = \varepsilon_i^1 = \frac{e_i^1}{\Delta t}, \quad \text{and } s_i^k = \varepsilon_i^{k+1}.$$

The numerical scheme is said to be convergent of order 2 if the convergence error tends toward zero at least as fast as $\Delta x^2 + \Delta t^2$ when both discretization steps tend toward zero.[3] More precisely, the numerical scheme is said to be convergent of order (m,n) uniformly on the interval $[0, t_{\max}]$ if the convergence error satisfies[4]

$$\left\| \left(i \mapsto e_i^{k_{\Delta t}(t)} \right) \right\|_{\Delta x} = O_{[0,t_{\max}]}(\Delta x^m + \Delta t^n). \tag{20}$$

The numerical scheme is said to be consistent with the continuous problem at order 2 if the truncation error tends toward zero at least as fast as $\Delta x^2 + \Delta t^2$ when the discretization steps tend toward 0. More precisely, the numerical scheme is said to be consistent with the continuous problem at order (m, n) uniformly on interval $[0, t_{\max}]$ if the truncation error satisfies

$$\left\| \left(i \mapsto \varepsilon_i^{k_{\Delta t}(t)} \right) \right\|_{\Delta x} = O_{[0,t_{\max}]}(\Delta x^m + \Delta t^n). \tag{21}$$

The numerical scheme is said to be stable if the discrete solution of the associated homogeneous problem (*i.e.* without any source term, $s(x,t) = 0$) is bounded independently of the discretization steps. More precisely, the numerical scheme is said to be stable uniformly on interval $[0, t_{\max}]$ if the discrete solution of the problem without any source term satisfies

$$\begin{aligned} &\exists \alpha, C_1, C_2 > 0,\ \forall t \in [0, t_{\max}],\ \forall \Delta x, \Delta t > 0, \quad \sqrt{\Delta x^2 + \Delta t^2} < \alpha \Rightarrow \\ &\quad \left\| \left(i \mapsto p_i^{k_{\Delta t}(t)} \right) \right\|_{\Delta x} \le (C_1 + C_2 t)(\|p_{0\mathrm{h}}\|_{\Delta x} + \|p_{0\mathrm{h}}\|_{A_h(c)} + \|p_{1\mathrm{h}}\|_{\Delta x}). \end{aligned} \tag{22}$$

The result to be formally proved at Section 3.1 states that if the continuous solution p is regular enough on $\Omega \times [0, t_{\max}]$ and if the discretization steps satisfy the CFL(ξ) condition, then the three-point scheme is convergent of order (2, 2) uniformly on interval $[0, t_{\max}]$.

We do not admit (nor prove) the Lax equivalence theorem which stipulates that for a wide variety of problems and numerical schemes, consistency implies the equivalence between stability and convergence. Instead, we establish that consistency and stability implies convergence in the particular case of the one-dimensional acoustic wave equation.

2.4 Program

The main part of the C program is listed in Listing 1.

The grid steps Δx and Δt are respectively represented by the variables `dx` and `dt`, the grid sizes $i_{\max}$ and $k_{\max}$ by the variables `ni` and `nk`, and the propagation

[3] Note that Δx tending toward 0 actually means that $i_{\max}$ goes to infinity.

[4] See Section 3.1.1 for the precise definition of the big O notation.

Listing 1 The main part of the C code, without annotations.

```
/* Compute the constant coefficient of the stiffness matrix. */
a1 = dt/dx*v;
a  = a1*a1;

/* First initial condition and boundary conditions. */
/* Left boundary. */
p[0][0] = 0.;
/* Time iteration -1 = space loop. */
for (i=1; i<ni; i++) {
  p[i][0] = p0(i*dx);
}
/* Right boundary. */
p[ni][0] = 0.;

/* Second initial condition (with p1=0) and boundary conditions. */
/* Left boundary. */
p[0][1] = 0.;
/* Time iteration 0 = space loop. */
for (i=1; i<ni; i++) {
  dp = p[i+1][0] - 2.*p[i][0] + p[i-1][0];
  p[i][1] = p[i][0] + 0.5*a*dp;
}
/* Right boundary. */
p[ni][1] = 0.;

/* Evolution problem and boundary conditions. */
/* Propagation = time loop. */
for (k=1; k<nk; k++) {
  /* Left boundary. */
  p[0][k+1] = 0.;
  /* Time iteration k = space loop. */
  for (i=1; i<ni; i++) {
    dp = p[i+1][k] - 2.*p[i][k] + p[i-1][k];
    p[i][k+1] = 2.*p[i][k] - p[i][k-1] + a*dp;
  }
  /* Right boundary. */
  p[ni][k+1] = 0.;
}
```

velocity c by the variable v. The initial position $p_{0\text{h}}$ is represented by the function p0. The initial velocity $p_{1\text{h}}$ and the source term s_h are supposed to be zero and are not represented. The discrete solution p_h is represented by the two-dimensional array p of size $(i_{\max}+1)\times(k_{\max}+1)$. (This is a simple naive implementation, a more efficient implementation would store only two time steps.)

To assemble the stiffness matrix $A_\text{h}(c)$, we only have to compute the square of the CFL coefficient $\frac{c\Delta t}{\Delta x}$ (lines 1–2). Then, we recognize the space loops for the initial conditions: Equation (11) on lines 6–8, and Equation (10) with $p_{1\text{h}} = 0$ on lines 14–17. The space-time loop on lines 23–28 for the evolution problem comes from Equation (9). And finally, the boundary conditions of Equation (12) are spread out on lines 9–10, 18–19, and 29–30.

3 Bounding Errors

3.1 Method Error

We first present the notions necessary to formalize and prove the method error. Then, we detail how the proof is conducted: we establish the consistency, the stability and prove that these two properties imply convergence in the case of the one-dimensional acoustic wave equation.

3.1.1 Big O, Differentiability, and Regularity

When considering a big O equality $a = O(b)$, one usually assumes that a and b are two expressions defined over the same domain and its interpretation as a quantified formula comes naturally. Here the situation is a bit more complicated. Consider

$$f(\mathbf{x}, \mathbf{\Delta x}) = O(g(\mathbf{\Delta x}))$$

when $\|\mathbf{\Delta x}\|$ goes to 0. If one were to assume that the equality holds for any $\mathbf{x} \in \mathbb{R}^2$, one would interpret it as

$$\forall \mathbf{x}, \exists \alpha > 0, \exists C > 0, \forall \mathbf{\Delta x}, \quad \|\mathbf{\Delta x}\| \le \alpha \Rightarrow |f(\mathbf{x}, \mathbf{\Delta x})| \le C \cdot |g(\mathbf{\Delta x})|,$$

which means that constants α and C are in fact functions of $\mathbf{x}$. Such an interpretation happens to be useless, since the infimum of α may well be zero while the supremum of C may be $+\infty$.

A proper interpretation requires the introduction of a uniform big O relation with respect to the additional variable $\mathbf{x}$:

$$\exists \alpha > 0,\ \exists C > 0,\ \forall \mathbf{x} \in \Omega_{\mathbf{x}},\ \forall \mathbf{\Delta x} \in \Omega_{\mathbf{\Delta x}},\\ \|\mathbf{\Delta x}\| \le \alpha \Rightarrow |f(\mathbf{x}, \mathbf{\Delta x})| \le C \cdot |g(\mathbf{\Delta x})|. \quad (23)$$

To emphasize the dependency on the two subsets $\Omega_{\mathbf{x}}$ and $\Omega_{\mathbf{\Delta x}}$, uniform big O equalities are now written

$$f(\mathbf{x}, \mathbf{\Delta x}) = O_{\Omega_{\mathbf{x}}, \Omega_{\mathbf{\Delta x}}}(g(\mathbf{\Delta x})).$$

We now precisely define the notion of "sufficiently regular" functions in terms of the full-fledged notation for the big O. The further result on the convergence of the numerical scheme requires that the solution of the continuous equation is actually sufficiently regular. We introduce two operators that, given a real-valued function f defined on the 2D plane and a point in the plane, return the values $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial t}$ at this point. Given these two operators, we can define the usual 2D Taylor polynomial of order n of a function f:

$$\mathrm{TP}_n(f, \mathbf{x}) \stackrel{\text{def}}{=} (\Delta x, \Delta t) \mapsto \sum_{p=0}^{n} \frac{1}{p!} \left(\sum_{m=0}^{p} \binom{p}{m} \cdot \frac{\partial^p f}{\partial x^m \partial t^{p-m}}(\mathbf{x}) \cdot \Delta x^m \cdot \Delta t^{p-m} \right).$$

Let $\Omega_{\mathbf{x}} \subset \mathbb{R}^2$. We say that the previous Taylor polynomial is a uniform approximation of order n of f on $\Omega_{\mathbf{x}}$ when the following uniform big O equality holds:

$$f(\mathbf{x} + \mathbf{\Delta x}) - \mathrm{TP}_n(f, \mathbf{x})(\mathbf{\Delta x}) = O_{\Omega_{\mathbf{x}}, \mathbb{R}^2}\left(\|\mathbf{\Delta x}\|^{n+1} \right).$$

A function f is then said to be *sufficiently regular of order n uniformly on $\Omega_{\mathbf{x}}$* when all its Taylor polynomials of order smaller than n are uniform approximations of f on $\Omega_{\mathbf{x}}$.

3.1.2 Consistency

The consistency of a numerical scheme expresses that, for $\mathbf{\Delta x}$ small enough, the continuous solution taken at the points of the grid almost solves the numerical scheme. More precisely, we formally prove that when the continuous solution of the wave equation (1)–(4) is sufficiently regular of order 4 uniformly on $[x_{\min}, x_{\max}] \times [0, t_{\max}]$, the numerical scheme (9)–(12) is consistent with the continuous problem at order (2, 2) uniformly on interval $[0, t_{\max}]$ (see definition (21) in Section 2.3). This is obtained using the properties of Taylor approximations; the proof is straightforward while involving long and complex expressions.

The key idea is to always manipulate uniform Taylor approximations that will be valid for all points of all grids when the discretization steps goes down to zero.

For instance, to take into account the initialization phase corresponding to Equation (10), we have to derive a uniform Taylor approximation of order 1 for the following continuous function (for any v sufficiently regular of order 3)

$$((x,t),(\Delta x, \Delta t)) \mapsto \frac{v(x,t+\Delta t)-v(x,t)}{\Delta t} - \frac{\Delta t}{2} c^2 \frac{v(x+\Delta x,t)-2v(x,t)+v(x-\Delta x,t)}{\Delta x^2}.$$

Note that the expression of this function involves both $x + \Delta x$ and $x - \Delta x$, meaning that we need a Taylor approximation which is valid for both positive and negative growths. The proof would have been impossible if we had required $0 < \Delta x$ (as a space grid step) in the definition of the Taylor approximation.

In contrast with the case of an infinite string [13], we do not need here a lower bound for $c\frac{\Delta t}{\Delta x}$.

3.1.3 Stability

The stability of a numerical scheme expresses that the growth of the discrete solution is somehow bounded in terms of the input data (here, the Cauchy data $u_{0\mathrm{h}}$ and $u_{1\mathrm{h}}$, and the source term s_{h}). For the proof of the round-off error (see Section 3.2), we need a statement of the same form as definition (22) of Section 2.3. Therefore, we formally prove that, under the CFL(ξ) condition (15), the numerical scheme (9)–(12) is stable uniformly on interval $[0, t_{\max}]$.

But, as we choose to prove the convergence of the numerical scheme by using an energetic technique[5], it is more convenient to formulate the stability in terms of the discrete energy. More precisely, we also formally prove that under the CFL(ξ) condition (15), the discrete energy (14) satisfies the following overestimation,

$$\sqrt{E_{\mathrm{h}}(c)(p_{\mathrm{h}})^{k+\frac{1}{2}}} \leq \sqrt{E_{\mathrm{h}}(c)(p_{\mathrm{h}})^{\frac{1}{2}}} + \frac{\sqrt{2}}{2\sqrt{2\xi-\xi^2}} \cdot \Delta t \cdot \sum_{k'=1}^{k} \left\| \left(i \mapsto s_i^{k'}\right) \right\|_{\Delta x}$$

[5] The popular alternative, using the Fourier transform, would have required huge additional Coq developments.

for all $t \in [0, t_{\max}]$ and with $k = \lfloor \frac{t}{\Delta t} \rfloor - 1$.

The evolution of the discrete energy between two consecutive time steps is shown to be proportional to the source term. In particular, the energy is constant when the source is inactive. Then, we obtain the following underestimation of the discrete energy,

$$\forall k, \quad \frac{1}{2}\left(1 - \left(c\frac{\Delta t}{\Delta x}\right)^2\right) \left\| \left(i \mapsto \frac{p_i^{k+1} - p_i^k}{\Delta t}\right) \right\|_{\Delta x} \leq E_{\mathrm{h}}(c)(p_{\mathrm{h}})^{k+\frac{1}{2}}.$$

Therefore, the non-negativity of the discrete energy is directly related to the CFL(ξ) condition.

Note that this stability result is valid for any input data $p_{0\mathrm{h}}$, $p_{1\mathrm{h}}$, and s_{h}.

3.1.4 Convergence

The convergence of a numerical scheme expresses the fact that the discrete solution gets closer to the continuous solution as the discretization steps go down to zero. More precisely, we formally prove that when the continuous solution of the wave equation (1)–(4) is sufficiently regular of order 4 uniformly on $[x_{\min}, x_{\max}] \times [0, t_{\max}]$, and under the CFL($\xi$) condition (15), the numerical scheme (9)–(12) is convergent of order (2, 2) uniformly on interval $[0, t_{\max}]$ (see definition (20) in Section 2.3).

Firstly, we prove that the convergence error e_{h} is itself the discrete solution of a numerical scheme of the same form but with different input data[6]. In particular, the source term (on the right-hand side) is here the truncation error ε_{h} associated with the initial numerical scheme for p_{h}. Then, the previous stability result holds, and we have an overestimation of the square root of the discrete energy associated with the convergence error $E_{\mathrm{h}}(c)(e_{\mathrm{h}})$ that involves a sum of the corresponding source terms, *i.e.* the truncation error. Finally, the consistency result also makes this sum a big O of $\Delta x^2 + \Delta t^2$, and a few more technical steps conclude the proof.

3.2 Round-off Error

As each operation is done with IEEE-754 floating-point numbers [46], round-off errors will occur and may endanger the accuracy of the final results. On this program, naive forward error analysis gives an error bound that is proportional to $2^k 2^{-53}$ for the computation of a p_i^k. If this bound was sensible, it would cause the numerical scheme to compute only noise after a few steps. Fortunately, round-off error actually compensate themselves. To take into account the compensations and hence prove a usable error bound, we need a precise statement of the round-off error [12] to exhibit the cancellations made by the numerical scheme.

3.2.1 Local Round-off Errors

Let δ_i^k be the (signed) floating-point error made in the two lines computing p_i^k (lines 26–27 in Listing 1). Floating-point values as computed by the program

[6] Of course, there is no associated continuous problem.

will be underlined: $\underline{a}$, $\underline{p}_i^k$ to distinguish them from the discrete values of previous sections. They match the expressions a and p[i][k] in the annotations, while a and p_i^k can be represented in the annotations by \exact(a) and \exact(p[i][k]), as described in Section 4.1.4.

The δ_i^k are defined as follow:

$$\delta_i^{k+1} = \underline{p}_i^{k+1} - (2\underline{p}_i^k - \underline{p}_i^{k-1} + a \times (\underline{p}_{i+1}^k - 2\underline{p}_i^k + \underline{p}_{i-1}^k)).$$

Note that the program explained in Section 2.4 gives us that

$$\underline{p}_i^{k+1} = \mathrm{fl}\left(2\underline{p}_i^k - \underline{p}_i^{k-1} + \underline{a} \times (\underline{p}_{i+1}^k - 2\underline{p}_i^k + \underline{p}_{i-1}^k)\right)$$

where $\mathrm{fl}(\cdot)$ means that all the arithmetic operations that appear between the parentheses are actually performed by floating-point arithmetic, hence a bit off.

In order to get a bound on δ_i^k, we need to have the range of $\underline{p}_i^k$. For this bound to be usable in our correctness proof, we need the range to be $[-2, 2]$. We have proved this fact by using the bounds on the method error and the round-off error of all the $\underline{p}^k$ and $\underline{p}^{k-1}$.

To prove the bound on δ_i^k, we perform forward error analysis and then use interval arithmetic to bound each intermediate error. We prove that, for all i and k, we have $|\delta_i^k| \leq 78 \times 2^{-52}$ for a reasonable error bound for a, that is to say $|\underline{a} - a| \leq 2^{-49}$.

3.2.2 Convolution of Round-off Errors

Note that the global floating-point error $\Delta_i^k = \underline{p}_i^k - p_i^k$ depends not only on δ_i^k, but also on all the δ_{i+j}^{k-l} for $0 < l \leq k$ and $-l \leq j \leq l$. Indeed round-off errors propagate along floating-point computations. Their contributions to Δ_i^k, which are independent and linear (due to the structure of the numerical scheme), can be computed by performing a convolution with a function $\lambda : (\mathbb{Z} \times \mathbb{Z}) \to \mathbb{R}$. This function λ represents the results of the numerical scheme when fed with a single unit value:

$$\lambda_0^0 = 1 \qquad \forall i \neq 0,\ \lambda_i^0 = 0$$
$$\lambda_{-1}^1 = \lambda_1^1 = a \qquad \lambda_0^1 = 2(1-a) \qquad \forall i \notin \{-1, 0, 1\},\ \lambda_i^1 = 0$$
$$\lambda_i^k = a \times (\lambda_{i-1}^{k-1} + \lambda_{i+1}^{k-1}) + 2(1-a) \times \lambda_i^{k-1} - \lambda_i^{k-2}$$

Theorem 1

$$\Delta_i^k = \underline{p}_i^k - p_i^k = \sum_{l=0}^{k} \sum_{j=-l}^{l} \lambda_j^l \ \delta_{i+j}^{k-l}.$$

Details of the proof can be found in [12], but this point of view using convolution is new. The proof mainly amounts to performing numerous tedious transformations of summations until both sides are proved equal.

The previous proof assumes that the double summation is correct for all (i', k') such that $k' < k$. This would be correct if there was an unbounded set of i where p_i^k is computed. This is no longer the case for a finite string. Indeed, at the ends of the range ($i = 0$ or $i_{\max}$), p_i^k and $\underline{p}_i^k$ are equal to 0, so Δ_i^k has to be 0 too.

The solution is to define the successive antisymmetric extension in space (as is done for d'Alembert's formula in Section 2.1) and to use it instead of δ. This ensures that both Δ_0^k and $\Delta_{i_{\max}}^k$ are equal to 0. It does not have any consequence on the values of Δ_i^k for $0 < i < i_{\max}$.

3.2.3 Bound on the Global Round-off Error

The analytic expression of Δ_i^k can be used to obtain a bound on the round-off error. We will need two lemmas for this purpose.

Lemma 1 $\displaystyle\sum_{i=-\infty}^{+\infty} \lambda_i^k = k + 1.$

Proof We have

$$\sum_{i=-\infty}^{+\infty} \lambda_i^{k+1} = 2\breve{a} \sum_{i=-\infty}^{+\infty} \lambda_i^k + 2(1-\breve{a}) \sum_{i=-\infty}^{+\infty} \lambda_i^k - \sum_{i=-\infty}^{+\infty} \lambda_i^{k-1} = 2 \sum_{i=-\infty}^{+\infty} \lambda_i^k - \sum_{i=-\infty}^{+\infty} \lambda_i^{k-1}.$$

The sum by line verifies a simple linear recurrence. As $\sum \lambda_i^0 = 1$ and $\sum \lambda_i^1 = 2$, we have $\sum \lambda_i^k = k + 1$. □

Lemma 2 $\lambda_i^k \geq 0.$

Proof The demonstration was found out by M. Kauers and V. Pillwein.

If we denote by P the Jacobi polynomial, we have

$$\lambda_n^j = \sum_{k=j}^{n} \binom{2k}{j+k} \binom{n+k+1}{2k+1} (-1)^{j+k} a^k = a^j \sum_{k=0}^{n-j} P_k^{(2j,0)}(1-2a)$$

Now the conjecture follows directly from the inequality of Askey and Gasper [3], which asserts that $\sum_{k=0}^{n} P_k^{(r,0)}(x) > 0$ for $r > -1$ and $-1 < x \leq 1$ (see Theorem 7.4.2 in The Red Book [2]). □

Theorem 2

$$\left|\Delta_i^k\right| = \left|\underline{p}_i^k - p_i^k\right| \leq 78 \times 2^{-53} \times (k+1) \times (k+2).$$

Proof According to Theorem 1, Δ_i^k is equal to $\sum_{l=0}^{k} \sum_{j=-l}^{l} \lambda_j^l \ \delta_{i+j}^{k-l}$. We know that for all j and l, $|\delta_j^l| \leq 78 \times 2^{-52}$ and that $\sum \lambda_i^l = l + 1$. Since the λ_i^k are nonnegative, the error is easily bounded by $78 \times 2^{-52} \times \sum_{l=0}^{k} (l+1)$. □

3.3 Total Error

Let $\mathcal{E}_{\mathrm{h}}$ be the total error. It is the sum of the method error (or convergence error) e_{h} of Sections 2.3 and 3.1.4, and of the round-off error Δ_h of Section 3.2.

From Theorem 2, we can estimate[7] the following upper bound for the spatial norm of the round-off error when $\Delta x \le 1$ and $\Delta t \le t_{\max}/2$: for all $t \in [0, t_{\max}]$,

$$\left\| \left(i \mapsto \Delta_i^{k_{\Delta t}(t)} \right) \right\|_{\Delta x} = \sqrt{\sum_{i=0}^{i_{\max}} \left(\Delta_i^{k_{\Delta t}(t)} \right)^2 \Delta x}$$
$$\le \sqrt{(i_{\max}+1)\Delta x} \times 78 \times 2^{-53} \times \left(\frac{t_{\max}}{\Delta t} + 1 \right) \times \left(\frac{t_{\max}}{\Delta t} + 2 \right)$$
$$\le \sqrt{x_{\max} - x_{\min} + 1} \times 78 \times 2^{-53} \times 3 \times \frac{t_{\max}^2}{\Delta t^2}.$$

Thus, from the triangular inequality for the spatial norm, we obtain the following estimation of the total error:

$$\forall t \in [0, t_{\max}],\ \forall \mathbf{\Delta x},\quad \|\mathbf{\Delta x}\| \le \min(\alpha_e, \alpha_\Delta) \Rightarrow$$
$$\left\| \left(i \mapsto \mathcal{E}_i^{k_{\Delta t}(t)} \right) \right\|_{\Delta x} \le C_e(\Delta x^2 + \Delta t^2) + \frac{C_\Delta}{\Delta t^2}$$

where the convergence constants α_e and C_e were extracted from the Coq proof (see Section 3.1.4) and are given in terms of the constants for the Taylor approximation of the exact solution at degree 3 (α_3 and C_3), and at degree 4 (α_4 and C_4) by

$$\alpha_e = \min(1, t_{\max}, \alpha_3, \alpha_4),$$
$$C_e = 2\mu t_{\max}\sqrt{x_{\max} - x_{\min}} \left(\frac{C'}{\sqrt{2}} + \mu(t_{\max}+1)C'' \right)$$

with $\mu = \frac{\sqrt{2}}{\sqrt{2\xi - \xi^2}}$, $C' = \max(1, C_3 + c^2 C_4 + 1)$, and $C'' = \max(C', 2(1+c^2)C_4)$, and where the round-off constants α_Δ and C_Δ, as explained above, are given by

$$\alpha_\Delta = \min(1, t_{\max}/2),$$
$$C_\Delta = 234 \times 2^{-53} \times t_{\max}^2 \sqrt{x_{\max} - x_{\min} + 1}.$$

To give an idea of the relative importance of both errors, we consider the academic case where the space domain is the interval $[0, 1]$, the velocity of waves is $c = 1$, and there is no initial velocity ($u_1(x) = 0$) nor source term ($s(x, t) = 0$). We suppose that the initial position is given by $u_0(x) = \chi(2(x - x_0)/l)$ where $x_0 = 0.5$, $l = 0.25$, and χ is the C^4 function defined on $[-1, 1]$ by $\chi(z) = (\cos(\frac{\pi}{2} z))^5$, and with null continuation on the real axis. For this function, we may take $\alpha_3 = \alpha_4 = \sqrt{2}/2$, $C_3 = 5120\sqrt{2}$, and $C_4 = 409600/3$. The corresponding solution presents two hump-shaped signals that propagate in each direction along the string, see Figure 1.

The upper bound on the total error is represented in Figure 3. Note that everything is in logarithmic scale. Of course, decreasing the size of the grid step decreases the method error, but in the same time, it increases the round-off error.

---

[7] When $\frac{t_{\max}}{\Delta t} \ge 2$, we have $\left(\frac{t_{\max}}{\Delta t} + 1\right)\left(\frac{t_{\max}}{\Delta t} + 2\right) \le 3\frac{t_{\max}^2}{\Delta t^2}$.

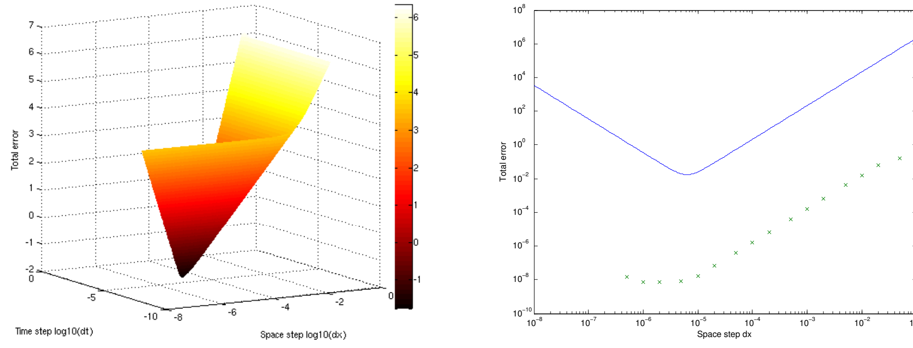


Fig. 3 Upper bound for the total error in log-scale. Left: for Δx and Δt satisfying the CFL condition. The lighter area (in yellow) represents the higher values above 10^4, whereas the darker area represents the lower values below 10^{-1}. Right: for an optimal CFL condition with $\Delta t = \frac{1-\xi}{c}\Delta x$. The green crosses represent the effective total error computed by the C program for a few values of the space step.

Hence, the existence of a minimum for the upper bound on the total error (about 0.02 in our test case), corresponding to optimal grid step sizes. Fortunately, the effective total error usually happens to be much smaller than this upper bound (by about a factor of 10^6 in our example).

Even if the effective total error on this example is off by several orders of magnitude with respect to the theoretical bound, this experiment is still reassuring. First, the left side of Figure 3 shows that the optimal choice (the darker part) for choosing Δx and Δt is reached near the limit of the CFL condition. This property matches common knowledge from numerical analysis. Second, the right side shows that both the effective error and the theoretical error have the same asymptotic behavior. So the properties we have verified in this work are not intrinsically easier than the best theorems one could state. It is just that the constants of the formulas extracted from the proofs (which we did not tune for this specific purpose) are not optimal for this example.

4 Mechanization of Proofs

In Sections 3.1 and 3.2, we have mostly described the method and round-off errors introduced when solving the wave equation problem with the given numerical scheme. We do not yet know whether this formalization actually matches the program described in Section 2.4 and fully given in Appendix A. In addition, the program might contain programming errors like out-of-bound accesses, which would possibly be left unattended while comparing the program and its formalization.

To fully verify the program, our process is as follows. First, we annotated the C program with comments specifying its behavioral properties, that is, what the program is supposed to compute. Second, we let Frama-C/Why generate proof obligations that state that the program matches its specification and that its execution is safe. Third, we used automated provers and Coq to prove all of these obligations.

Section 4.1 presents all the tools we have used for verifying the C program. Then Section 4.2 explains how the program was annotated. Finally, Section 4.3 shows how we proved all the obligations, either automatically or with a proof assistant.

4.1 Tools

Several software packages are used in this work. The formal proof of the method error has been made in Coq. The formal proof of the round-off error has been made in Coq, and using the Gappa tactic. The certification of the C program has used Frama-C (with the Jessie plug-in), and to prove the produced goals, we used Gappa, SMT provers, and the preceding Coq proofs. This section is devoted to present these tools and necessary libraries.

4.1.1 Coq

Coq[8] is a formal system that provides an expressive language to write mathematical definitions, executable algorithms, and theorems, together with an interactive environment for proving them [8]. Coq's formal language is based on the Calculus of Inductive Constructions [21] that combines both a higher-order logic and a richly-typed functional programming language. Coq allows to define functions or predicates, that can be evaluated efficiently, to state mathematical theorems and software specifications, and to interactively develop formal proofs of these theorems. These proofs are machine-checked by a relatively small *kernel*, and certified programs can be extracted from them to external programming languages like Objective Caml, Haskell, or Scheme [42].

As a proof development system, Coq provides interactive proof methods, decision and semi-decision algorithms, and a tactic language for letting the user define its own proof methods. Connection with external computer algebra system or theorem provers is also available.

The Coq library is structured into two parts: the initial library, which contains elementary logical notions and data-types, and the standard library, a general-purpose library containing various developments and axiomatizations about sets, lists, sorting, arithmetic, real numbers, etc.

In this work, we mainly use the Reals standard library [44], that is a classical axiomatization of an Archimedean ordered complete field. We chose Reals to make our numerical proofs because we do not need an intuitionistic formalization.

For floating-point numbers, we use a large Coq library[9] initially developed in [26] and extended with various results afterwards [11]. It is a high-level formalization of IEEE-754 with gradual underflow. This is expressed by a formalization where floating-point numbers are pairs (n, e) associated with real values $n \times \beta^e$. The requirements for a number to be in the format $(e_{\min}, \beta^p)$ are

$$|n| < \beta^p \quad \text{and} \quad e_{\min} \le e.$$

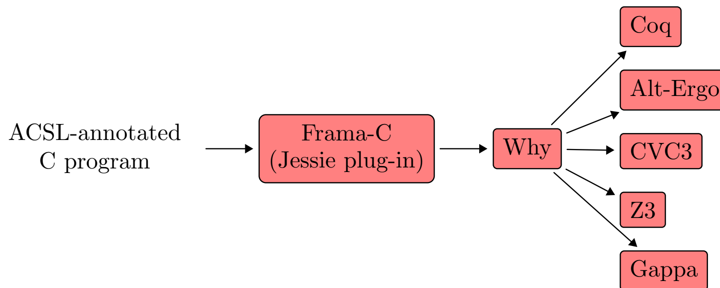
[8] `http://coq.inria.fr/`

[9] `http://lipforge.ens-lyon.fr/www/pff/`

This formalization is convenient for human interactive proofs as testified by the numerous proofs using it. The huge number of lemmas available in the library (about 1400) makes it suitable for a large range of applications. This library has since then been superseded by the Flocq library [16], but it was not yet available at the time we proved the floating-point results of this work.

4.1.2 Frama-C, Jessie, Why, and the SMT Solvers

We use the Frama-C platform[10] to perform formal verification of C programs at the source-code level. Frama-C is an extensible framework that combines static analyzers for C programs, written as plug-ins, within a single tool. In this work, we use the Jessie plug-in for deductive verification. C programs are annotated with behavioral contracts written using the *ANSI C Specification Language* (ACSL for short) [6]. The Jessie plug-in translates them to the Jessie language [43], which is part of the Why verification platform [30]. This part of the process is responsible for translating the semantics of C into a set of Why logical definitions (to model C types, memory heap, etc.) and Why programs (to model C programs). Finally, the Why platform computes verification conditions from these programs, using traditional techniques of weakest preconditions, and emits them to a wide set of existing theorem provers, ranging from interactive proof assistants to automated theorem provers. In this work, we use the Coq proof assistant (Section 4.1.1), SMT solvers Alt-Ergo [19], CVC3 [5] and Z3 [47], and the automated theorem prover Gappa (Section 4.1.3). Details about automated and interactive proofs can be found in Section 4.3. The dataflow from C source code to theorem provers can be depicted as follows:



More precisely, to run the tools on a C program, we use a graphical interface called gWhy. A screenshot is in Appendix B. In this interface, we may call one prover on one or on many goals. We then get a graphical view of how many goals are proved and by which prover.

In ACSL, annotations are using first-order logic. Following the *programming by contract* approach, the specifications involve preconditions, postconditions, and loop invariants. Contrary to other approaches focusing on run-time assertion checking, ACSL specifications do not refer to C values and functions, even if pure, but refer instead to purely logical symbols. In the following contract for a function computing the square of an integer x

```
//@ ensures \result == x * x;
int square(int x);
```

[10] http://www.frama-c.cea.fr/

the postcondition, introduced with ensures, refers to the return value \result and argument x. Both are denoting mathematical integer values, for the corresponding C values of type int. In particular, x * x cannot overflow. Of course, one could give function square a more involved specification that handles overflows, *e.g.* with a precondition requiring x to be small enough. Simply speaking, we can say that C integers are reflected within specifications as mathematical integers, in an obvious way. The translation of floating-point numbers is more subtle and explained in Section 4.1.4.

4.1.3 Gappa

The Gappa tool[11] adapts the interval-arithmetic paradigm to the proof of properties that occur when verifying numerical applications [25]. The inputs are logical formulas quantified over real numbers whose atoms are usually enclosures of arithmetic expressions inside numeric intervals. Gappa answers whether it succeeded in verifying it. In order to support program verification, one can use *rounding* functions inside expressions. These unary operators take a real number and return the closest real number in a given direction that is representable in a given binary floating-point format. For instance, assuming that operator $\circ$ rounds to the nearest binary64 floating-point number, the following formula states that the relative error of the floating-point addition is bounded:

$$\forall x, y \in \mathbb{R},\ \exists \varepsilon \in \mathbb{R},\ |\varepsilon| \le 2^{-53} \wedge \circ(\circ(x) + \circ(y)) = (\circ(x) + \circ(y)) \times (1 + \varepsilon).$$

Converting straight-line numerical programs to Gappa logical formulas is easy and the user can provide additional hints if the tool were to fail to verify a property. The tool is specially designed to handle codes that are performing convoluted manipulations. For instance, it has been successfully used to verify a state-of-the-art library of correctly-rounded elementary functions [28]. In the current work, Gappa has been used to check much simpler properties. (In particular, no user hint was needed to discharge a proof automatically.) But the length of their proofs would discourage even the most dedicated users if they were to be manually handled. One of the properties is the round-off error of a local evaluation of the numerical scheme (Section 3.2.1). Other properties mainly deal with proving that no exceptional behavior occurs while executing the program: due to the initial values, all the computed values are sufficiently small to never cause overflow.

The verification of some formulas requires reasonings that are so long and intricate [28], that it might cast some doubts on whether an automatic tool actually succeeded in proving them. This is especially true when the tool ends up proving a property stronger than what the user expected. That is why Gappa also generates a formal certificate that can be mechanically checked by a proof assistant. This feature has served as the basis for a Coq tactic for automatically solving goals related to floating-point and real arithmetic [15]. The tactic reads the current Coq goal, generates a Gappa goal, executes Gappa on it, recovers the certificate, and converts it to a complete proof term that Coq matches against the current goal. At this point, whether Gappa is correct or not no longer matters: the original Coq goal is formally proved by a complete Coq proof.

[11] http://gappa.gforge.inria.fr/

This tactic has been used whenever a verification condition would have been directly proved by Gappa, if not for some confusing notations or encodings of matrix elements. We just had to apply a few basic Coq tactics to put the goal into the proper form and then call the Gappa tactic to discharge it automatically.

4.1.4 Floating-Point Formalizations

A natural question is the link between the various representations of floating-point numbers. We assume that the execution environment (mostly the processor) complies with the IEEE-754 standard [46], which defines formats, rounding modes, and operations. The C program we consider is compiled in an assembly code that will directly use these formats and operations. We also assume that the compiler optimizations preserve the visible semantics of floating-operations from the original code, *e.g.* no use of the extended registers. Such optimizations could have been taken into account though, but at a cost [17].

When verifying the C program, the floating-point operations are translated by Frama-C/Jessie/Why following some previous work by two of the authors [14]. A floating-point number f is modeled in the logic as a triple of real numbers (r, e, m). Value r simply stands for the real number that is immediately represented by f; value e stands for the *exact* value of f, as obtained if no rounding errors had occurred; finally, value m stands for the *model* of f, which is a placeholder for the value intended to be computed and filled by the user. The two latter values have no existence in the program, but are useful for the specification and the verification. In particular, they help state assertions about the rounding or the model error of a program. In ACSL, the three components of the model of a floating-point number f can be referred to using f, \exact(f), and \model(f), respectively. \round_error(f) is a macro for the rounding error, that is, \abs(f - \exact(f)).

For instance, the following excerpt from our C program specifies the error on the content of the dx variable, which represents the grid step Δx (see Section 2).

```
dx = 1./ni;
/*@ assert
  @   dx > 0. && dx <= 0.5 &&
  @   \abs(\exact(dx) - dx) / dx <= 0x1.p-53;
  @ */
```

Note that 0x1.p-53 is a valid ACSL (and C too) literal meaning 2^{-53}.

Proof obligations are extracted from the annotated C program by computing weakest preconditions and then translated to automated and interactive provers. For SMT provers, the three fields r, e, and m, of floating-point numbers are expressed as real numbers and operations on floating-point numbers are uninterpreted relations axiomatized with basic properties such as bounds on the rounding error or monotonicity. For Gappa too, the fields are seen as real numbers. The tool, however, knows about floating-point arithmetic and its relation to real arithmetic. So floating-point operations are translated to the corresponding symbols from Gappa.

For Coq, we use the formalization described in Section 4.1.1 with a limited precision and gradual underflow (so that subnormal numbers are correctly translated). It is based on the real numbers of the standard library, which are also used for the translation of the exact and the model parts of the floating-point number.

While the IEEE-754 standard defines infinities and NaNs (Not-a-Number) as floating-point values, our translation does not take them into account. This does not compromise the correctness of the translation though, as each operation has a precondition that raises a proof obligation to guarantee that no exceptional events occur, such as overflow or division by zero, and therefore no infinities nor NaNs are produced by the program.

To summarize, there is one assumption about the actual arithmetic being executed (IEEE-754 compliant and no overly aggressive optimizations from the compiler) and three formalizations of floating-point arithmetic used to verify the program: one used by Jessie/Why and then sent to the SMT solvers, one used by Gappa, and one used by Coq. The combination of these three different formalizations does not introduce any inconsistency. Indeed, we have formally proved in Coq that Gappa's and Coq's formalizations are equivalent for floating-point formats with limited precision and gradual underflow, that is, IEEE-754 formats. We have also formally proved that the Jessie/Why specifications and the properties for SMT provers are compatible with these formalizations, including the absence of special values (infinity or NaN) and the possibility to disregard the upper bound on reals representing floating-point numbers.

In fact, there is a fourth formalization of floating-point arithmetic involved, which is the one used internally by the interval computations of Gappa for proving results about real-valued expressions. It is not equivalent to the previous ones, since it is a multi-precision arithmetic, but it has no influence whatsoever on the formalization that Gappa uses for modeling floating-point properties.

4.2 Program Annotations

The full annotations are given in Appendix A. We give here hints about how to specify this program.

There are two axiomatics. The first one corresponds to the mathematics: the exact solution of the wave equation and its properties. It defines the needed values (the exact solution p, and its initialization p_0). We here assume that s and p_1 are zero functions. It also defines the derivatives of p ($psol_1$, first derivative for the first variable of p, and $psol_{11}$, second derivative for the first variable, and $psol_2$ and $psol_{22}$ for the second variable) as functions such that their value is the limit of $\frac{p(x+\Delta x)-p(x)}{\Delta x}$ when $\Delta x \to 0$. As the ACSL annotations are only first order, these definitions are quite cumbersome: each derivative needs 5 lines to be defined.

We also put as axioms the fact that the solution has the expected properties (1–4). The last property needed on the exact solution is its regularity. We require it to be near its Taylor approximations of degrees 3 and 4 on the whole interval $[x_{\min}, x_{\max}]$. For instance, the following annotation states the property for degree 3.

```
/*@ axiom psol_suff_regular_3:
  @   0 < alpha_3 && 0 < C_3 &&
  @   \forall real x; \forall real t; \forall real dx; \forall real dt;
  @   0 <= x <= 1 ==> \sqrt(dx * dx + dt * dt) <= alpha_3 ==>
  @   \abs(psol(x + dx, t + dt) - psol_Taylor_3(x, t, dx, dt)) <=
  @     C_3 * \abs(\pow(\sqrt(dx * dx + dt * dt), 3));
  @*/
```

The second axiomatic corresponds to the properties and loop invariant needed by the program. For example, we require the matrix to be separated: it means that a line of the matrix should not mix with another line (or a modification could alter another point of the matrix). We also state the existence of the loop invariant analytic_error that is needed for applying the results of Section 3.2.

The initializations functions are specified, but not stated. This corresponds firstly to the function array2d_alloc that initializes the matrix and p_zero that produces an approximation of the p_0 function. Our program verification is modular: our proofs are generic with respect to p_0 and its implementation.

The preconditions of the main functions are the following ones:

- $i_{\max}$ and $k_{\max}$ must be greater than one, but small enough so that $i_{\max} + 1$ and $k_{\max} + 1$ do not overflow;
- the grid sizes $\mathbf{\Delta x}$ must fulfill some mathematical conditions that are required for the convergence of the scheme;
- the floating-point values computed for the grid sizes must be near their mathematical values;
- to prevent exceptional behavior in the computation of a, the time discretization step must be greater than 2^{-1000} and $\frac{c\Delta t}{\Delta x}$ must be greater than 2^{-500}.

There are two postconditions, corresponding to the method and round-off errors. See Sections 3.1 and 3.2 for more details.

4.3 Automation and Manual Proofs

This section is devoted to formal specifications and proofs corresponding to the bounds proved in Section 3. We give some key points of the automated proofs.

Big O. In section 3.1.1, we present two interpretations of the big O notation. Usual mathematical pen-and-paper proofs switch from one interpretation to the other depending on which one is the most adapted, without noticing that they may not be equivalent. The formal development was helpful in bringing into light the erroneous reasoning hidden by the usage of big O notations. We introduced the notion of uniform big O in [13] in the context of an infinite string. In the present paper, we consider the case of the finite string, hence for compactness reasons, both notions are in fact equivalent. However, we still use the more general uniform big O notion to share most of the proof developments between the finite and the infinite cases. Regarding automation, a decision procedure has been developed in [4]; unfortunately, those results were not applicable since we needed a more powerful big O.

Differential operators. As long as we were studying only the method error, we did not have to define the differential operators nor assume anything about them [13]. Their only properties appeared through their usage: function p is a solution of the partial differential equation and it is sufficiently regular. This is no longer possible for the annotated C program. Indeed, due to the underlying logic, the annotations have to define p as a solution of the PDE by using first-order formulas stating differentiability, instead of second-order formulas involving differential operators. Since the formalization of Taylor approximations has been left unchanged, the

natural way to relate the C annotations with the Coq development is therefore to define the operators as actual differential operators. Note that this has forced us to introduce a small axiom. Indeed, our definition of Taylor approximation depends on differential operators that are total functions, while Coq's standard library defines only partial operators. So we have assumed the existence of some total operators that are equal to the partial ones whenever applied to differentiable functions. The axiom states absolutely nothing about the result of these operators for nondifferentiable functions, so no inconsistencies are introduced this way. This is just a specific instance of Hilbert ε operator [57], which does not make the logic inconsistent [40].

Method error. The Coq proof of the method error is about 5000-line long. About half of it is dedicated to the wave equation and the other half is re-usable (definition and properties of the dot product, the big O, Taylor expansions...). We formally proved without any axiom that the numerical scheme is convergent of order 2, which is the known mathematical result. An interesting aspect of the formal proof in Coq is that we were able to extract the constants α and C appearing in the big O for the convergence result in order to obtain their precise values. The recursive extraction was fully automatic after making explicit some inlining. The mathematical expressions are given in Section 3.3.

Round-off errors. Except for Lemma 2, all the proofs described in section 3.2 have been done and machined-checked using Coq. In particular, the proof of the bound on δ_i^k was done automatically by calling Gappa from Coq. Lemma 2 is a technical detail compared to the rest of our work, that is not worth the immense Coq development it would require: keen results on integrals but also definitions and results about the Legendre, Laguerre, Chebychev, and Jacobi polynomials.

The program proof. Given the program code, the Why tool generates 149 verification conditions that have to be proved. While possible, proving all of them in Coq would be rather tedious. Moreover, it would lead to a rather fragile construct: any later modification to the code, however small it is, would cause different proof obligations to be generated, which would then require additional human interaction to adapt the Coq proofs. We prefer to have automated provers (SMT solvers and Gappa) discharge as many of them as possible, so that only the most intricate ones are left to be proven in Coq. The following table shows how many goals are discharged automatically and how many are left to the user.[12]

Prover	Proved Behavior VC	Proved Safety VC	Total
Alt-Ergo	18	80	98
CVC3	18	89	107
Gappa	2	20	22
Z3	21	63	84
Automatically proved	**23**	**94**	**117**
Coq	**21**	**11**	**32**
Total	44	105	149

[12] Note that verification conditions might be discharged by one or several automated provers.

On safety goals (matrix access, loop variant decrease, overflow), automatic provers are helpful: they prove about 90 % of the goals. On behavior goals (loop invariant, assertion, postcondition), automatic provers succeed for half of the goals. As our loop invariant involves an uninterpreted predicate, the automatic provers cannot prove all the behavior goals (they would have been too complicated anyway). That is why we resort to an interactive higher-order theorem prover, namely Coq.

Coq proofs are split into two sets: first, the mathematical proof of convergence and second, the proofs of bounded round-off errors and absence of runtime errors. Appendix C displays the layout of the Coq formalization.

The following tabular gives the compilation times of the Coq files on a 3-GHz dual core machine.

Type of proofs	Nb spec lines	Nb lines	Compilation time
Convergence	991	5 275	42 s
Round-off + runtime errors	7 737	13 175	32 min

Note that most proof statements regarding round-off and runtime errors are automatically generated (7 321 lines out of 7 737) by the Frama-C/Jessie/Why framework.

The compilation time may seem prohibitive; it is mainly due to the size of the theorems and to calls to the `omega` decision procedure for Presburger arithmetic. The difficulty does not lie in the arithmetic statement itself, but rather in a large number of useless hypotheses. In order to reduce the compilation time, we could manually massage the hypotheses to speed up the procedure, but this would defeat the point of using an automatic tactic.

5 Conclusion

In the end, having *formally* verified the C program means that all of the proof obligations generated by Frama-C/Jessie/Why have been proved, either by automated tools or by Coq formal proofs. These formal proofs depend on some axioms specific to this work: the fact about Jacobi polynomials, the existence of a solution to the EDP, and the existence of differential operators. The last two have been tackled by subsequent works, which means that the only remaining Coq axiom is the one about Jacobi polynomials.

We succeeded in verifying a C program that implements a numerical scheme for the resolution of the one-dimensional acoustic wave equation. This is comprised of three sets of proofs. First we formalized the wave equation and proved the convergence of a scheme for its numerical resolution. Second we proved that the C program behaves safely: no out-of-bound array accesses and no overflow during floating-point computations. Third we proved that the round-off errors are not causing the numerical results to go astray. This is the first verification of this kind of program that covers all its aspects, both mathematics and implementation.

This work shows a tight synergy between researchers from applied mathematics and logic. Three domains are intertwined here: applied mathematics for an initial proof that was enriched and detailed upon request, computer arithmetic for smart bounds on round-off errors, and formal methods for machine-checking

them. This may be the reason why such proofs never appeared before, as this kind of collaboration is uncommon.

Each proof came with its own hurdles. For ensuring the correct behavior of the program, the most tedious point was to prove that setting a result value did not cause other values to change, that is, that all the lines of the matrix are properly separated. In particular, verifying the loop invariant requires checking that, except for the new value, the properties of the memory are preserved. An unexpectedly tedious part was to check that the program actually complies with our mathematical model for the numerical scheme.

Another difficulty lies in the mathematical proof itself. We based our work on proofs found in books, courses, and articles. It appears that pen-and-paper proofs are sometimes sketchy: they may be fuzzy about the needed hypotheses, especially when switching quantifiers. We have also learned that filling the gaps may cause us to go back to the drawing board and to change the basic blocks of our formalization to make them more generic (*e.g.* devising a big O that needs to be uniform and also generic with respect to a property P).

An unexpected side effect of having performed this formal verification in Coq is our ability to automatically extract the constants hidden inside the proofs. That way, we are able to explicitly bound the total error rather than just having the usual $O(\Delta x^2 + \Delta t^2)$ bound. In particular, we can compare the magnitudes of the method error and round-off error and then decide how to scale the discretization grid.

Coq could have offered us more: it would have been possible to describe and prove the algorithm directly in Coq. The same formalism would have been used all the way long, but we were more interested in proving a real-life program in a real-life language. This has shown us the difficulties lying in the memory handling for matrices. In the end, we have a C code with readable annotations instead of a Coq theorem and that seems more convincing to applied mathematicians.

For this exploratory work, we considered the simple three-point scheme for the one-dimensional wave equation. Further works involve scaling to higher-dimension. The one-dimensional case showed us that summations and finite support functions play a much more important role in the development than we first expected. We are therefore moving to the SSReflect interface and libraries for Coq [9], so as to simplify the manipulations of these objects in the higher-dimensional case.

This example also exhibits a major cancellation of rounding errors and it would be interesting to see under which conditions numerical schemes behave so well.

Another perspective is to generalize our approach to other higher-order numerical schemes for the same equation, and to other PDEs. However, the proofs of Section 3.1 are entangled with particulars of the presented problem, and would therefore have to be redone for other problems. So a more fruitful approach would be to prove once and for all the Lax equivalence theorem that states that consistency implies the equivalence between convergence and stability. This would considerably reduce the amount of work needed for tackling other schemes and equations.

References

1. Achenbach, J.D.: Wave Propagation in Elastic Solids. North Holland, Amsterdam (1973)
2. Andrews, G.E., Askey, R., Roy, R.: Special functions. Cambridge University Press, Cambridge (1999)
3. Askey, R., Gasper, G.: Certain rational functions whose power series have positive coefficients. The American Mathematical Monthly **79**, 327–341 (1972)
4. Avigad, J., Donnelly, K.: A Decision Procedure for Linear “Big O” Equations. J. Autom. Reason. **38**(4), 353–373 (2007)
5. Barrett, C., Tinelli, C.: CVC3. In: 19th International Conference on Computer Aided Verification (CAV ’07), *LNCS*, vol. 4590, pp. 298–302. Springer-Verlag (2007). Berlin, Germany
6. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, version 1.5 (2009). URL `http://frama-c.cea.fr/acsl.html`
7. Bécache, E.: Étude de schémas numériques pour la résolution de l’équation des ondes. Master Modélisation et simulation, Cours ENSTA (2009). URL `http://www-rocq.inria.fr/~becache/COURS-ONDES/Poly-num-0209.pdf`
8. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq’Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer (2004)
9. Bertot, Y., Gonthier, G., Ould Biha, S., Pasca, I.: Canonical Big Operators. In: 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs’08), *LNCS*, vol. 5170, pp. 86–101. Springer, Montreal, Canada (2008)
10. Bobot, F., Conchon, S., Contejean, E., Iguernelala, M., Lescuyer, S., Mebsout, A.: The Alt-Ergo automated theorem prover (2008). URL `http://alt-ergo.lri.fr/`
11. Boldo, S.: Preuves formelles en arithmétiques à virgule flottante. Ph.D. thesis, École Normale Supérieure de Lyon (2004)
12. Boldo, S.: Floats & Ropes: a case study for formal numerical program verification. In: 36th International Colloquium on Automata, Languages and Programming, *LNCS - ARCoSS*, vol. 5556, pp. 91–102. Springer, Rhodos, Greece (2009)
13. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Formal proof of a wave equation resolution scheme: the method error. In: M. Kaufmann, L.C. Paulson (eds.) 1st Interactive Theorem Proving Conference (ITP), *LNCS*, vol. 6172, pp. 147–162. Springer, Edinburgh, Scotland (2010)
14. Boldo, S., Filliâtre, J.C.: Formal Verification of Floating-Point Programs. In: 18th IEEE International Symposium on Computer Arithmetic, pp. 187–194. Montpellier, France (2007)
15. Boldo, S., Filliâtre, J.C., Melquiond, G.: Combining Coq and Gappa for certifying floating-point programs. In: J. Carette, L. Dixon, C.S. Coen, S.M. Watt (eds.) 16th Calculemus Symposium, *Lecture Notes in Artificial Intelligence*, vol. 5625, pp. 59–74. Grand Bend, ON, Canada (2009)
16. Boldo, S., Melquiond, G.: Flocq: A unified library for proving floating-point algorithms in Coq. In: E. Antelo, D. Hough, P. Ienne (eds.) 20th IEEE Symposium on Computer Arithmetic, pp. 243–252. Tübingen, Germany (2011)
17. Boldo, S., Nguyen, T.M.T.: Proofs of numerical programs when the compiler optimizes. Innovations in Systems and Software Engineering **7**, 1–10 (2011)
18. Brekhovskikh, L.M., Goncharov, V.: Mechanics of Continua and Wave Dynamics. Springer (1994)
19. Conchon, S., Contejean, E., Kanig, J., Lescuyer, S.: CC(X): Semantical combination of congruence closure with solvable theories. In: Post-proceedings of the 5th International Workshop on Satisfiability Modulo Theories (SMT 2007), *Electronic Notes in Computer Science*, vol. 198-2, pp. 51–69. Elsevier Science Publishers (2008)
20. The Coq reference manual. URL `http://coq.inria.fr/refman/`
21. Coquand, T., Paulin-Mohring, C.: Inductively defined types. In: P. Martin-Löf, G.Mints (eds.) Colog’88, *LNCS*, vol. 417. Springer-Verlag (1990)
22. Courant, R., Friedrichs, K., Lewy, H.: On the partial difference equations of mathematical physics. IBM Journal of Research and Development **11**(2), 215–234 (1967)
23. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: ESOP, no. 3444 in LNCS, pp. 21–30 (2005)
24. Cruz-Filipe, L.: A Constructive Formalization of the Fundamental Theorem of Calculus. In: H. Geuvers, F. Wiedijk (eds.) 2nd International Workshop on Types for Proofs and Programs (TYPES 2002), *LNCS*, vol. 2646. Springer, Berg en Dal, Netherlands (2002)

25. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. Transactions on Mathematical Software **37**(1), 1–20 (2010)
26. Daumas, M., Rideau, L., Théry, L.: A generic library for floating-point numbers and its application to exact computing. In: TPHOLs, pp. 169–184 (2001)
27. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of FLUCTUAT on safety-critical avionics software. In: FMICS, *LNCS*, vol. 5825, pp. 53–69. Springer (2009)
28. de Dinechin, F., Lauter, C., Melquiond, G.: Certifying the floating-point implementation of an elementary function using Gappa. Transactions on Computers **60**(2), 242–253 (2011)
29. Dutertre, B.: Elements of mathematical analysis in PVS. In: J. von Wright, J. Grundy, J. Harrison (eds.) 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96), *LNCS*, vol. 1125, pp. 141–156. Springer, Turku, Finland (1996)
30. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: 19th International Conference on Computer Aided Verification, *LNCS*, vol. 4590, pp. 173–177. Springer, Berlin, Germany (2007)
31. Fleuriot, J.D.: On the mechanization of real analysis in Isabelle/HOL. In: M. Aagaard, J. Harrison (eds.) 13th International Conference on Theorem Proving and Higher-Order Logic (TPHOLs'00), *LNCS*, vol. 1869, pp. 145–161. Springer (2000)
32. The Frama-C platform for static analysis of C programs (2008). URL `http://www.frama-c.cea.fr/`
33. Gamboa, R., Kaufmann, M.: Nonstandard analysis in ACL2. Journal of Automated Reasoning **27**(4), 323–351 (2001)
34. Geuvers, H., Niqui, M.: Constructive reals in Coq: Axioms and categoricity. In: P. Callaghan, Z. Luo, J. McKinna, R. Pollack (eds.) 1st International Workshop on Types for Proofs and Programs (TYPES 2000), *LNCS*, vol. 2277, pp. 79–95. Springer, Durham, United Kingdom (2002)
35. Harrison, J.: Theorem Proving with the Real Numbers. Springer (1998)
36. Harrison, J.: A HOL theory of euclidean space. In: J. Hurd, T.F. Melham (eds.) 18th International Conference on Theorem Proving and Higher-Order Logic (TPHOLs'05), *LNCS*, vol. 3603, pp. 114–129. Springer (2005)
37. John, F.: Partial Differential Equations. Springer (1986)
38. Krebbers, R., Spitters, B.: Type classes for efficient exact real arithmetic in Coq. arXiv:1106.3448v1 (2011). URL `http://arXiv.org/abs/1106.3448`
39. le Rond D'Alembert, J.: Recherches sur la courbe que forme une corde tendue mise en vibrations. In: Histoire de l'Académie Royale des Sciences et Belles Lettres (Année 1747), vol. 3, pp. 214–249. Haude et Spener, Berlin (1749)
40. Lee, G., Werner, B.: Proof-irrelevant model of CC with predicative induction and judgmental equality. Logical Methods in Computer Science **7**(4:5) (2011)
41. Lelay, C., Melquiond, G.: Différentiabilité et intégrabilité en Coq. Application à la formule de d'Alembert. In: 23èmes Journées Francophones des Langages Applicatifs, pp. 119–133. Carnac, France (2012)
42. Letouzey, P.: A new extraction for Coq. In: H. Geuvers, F. Wiedijk (eds.) 2nd International Workshop on Types for Proofs and Programs (TYPES 2002), *LNCS*, vol. 2646. Springer, Berg en Dal, Netherlands (2003)
43. Marché, C.: Jessie: an intermediate language for Java and C verification. In: Programming Languages meets Program Verification (PLPV), pp. 1–2. ACM, Freiburg, Germany (2007)
44. Mayero, M.: Formalisation et automatisation de preuves en analyses réelle et numérique. Ph.D. thesis, Université Paris VI (2001)
45. Mayero, M.: Using theorem proving for numerical analysis (correctness proof of an automatic differentiation algorithm). In: V. Carreño, C. Muñoz, S. Tahar (eds.) 15th International Conference on Theorem Proving and Higher-Order Logic, *LNCS*, vol. 2410, pp. 246–262. Springer, Hampton, VA, USA (2002)
46. Microprocessor Standards Committee: IEEE Standard for Floating-Point Arithmetic. IEEE Std. 754-2008 pp. 1–58 (2008). DOI 10.1109/IEEESTD.2008.4610935
47. de Moura, L., Bjørner, N.: Z3, an efficient SMT solver. In: TACAS, *Lecture Notes in Computer Science*, vol. 4963, pp. 337–340. Springer (2008)
48. Newton, I.: Axiomata, sive Leges Motus. In: Philosophiae Naturalis Principia Mathematica, vol. 1. London (1687)
49. O'Connor, R.: Certified exact transcendental real number computation in Coq. In: 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08), *LNCS*, vol. 5170, pp. 246–261. Springer (2008)

50. O'Connor, R., Spitters, B.: A computer-verified monadic functional implementation of the integral. Theoretical Computer Science **411**(37), 3386–3402 (2010)
51. Rosinger, E.E.: Propagation of round-off errors and the role of stability in numerical methods for linear and nonlinear PDEs. Applied Mathematical Modelling **9**(5), 331–336 (1985)
52. Rosinger, E.E.: L-convergence paradox in numerical methods for PDEs. Applied Mathematical Modelling **15**(3), 158–163 (1991)
53. Roy, C.J., Oberkampf, W.L.: A comprehensive framework for verification, validation, and uncertainty quantification in scientific computing. Computer Methods in Applied Mechanics and Engineering **200**(25-28), 2131–2144 (2011)
54. Rudnicki, P.: An overview of the MIZAR project. In: Types for Proofs and Programs, pp. 311–332 (1992)
55. Szyszka, B.: An interval method for solving the one-dimensional wave equation. In: 7th EUROMECH Solid Mechanics Conference (ESMC2009). Lisbon, Portugal (2009)
56. Thomas, J.W.: Numerical Partial Differential Equations: Finite Difference Methods. No. 22 in Texts in Applied Mathematics. Springer (1995)
57. Zach, R.: Hilbert's "Verunglueckter Beweis," the first epsilon theorem, and consistency proofs. URL `http://front.math.ucdavis.edu/math.LO/0204255`
58. Zwillinger, D.: Handbook of Differential Equations. Academic Press (1998)

A Source Code

```
/*@ axiomatic dirichlet_maths {
  @
  @ logic real c;
  @ logic real p0(real x);
  @ logic real psol(real x, real t);

  @ axiom c_pos: 0 < c;

  @ logic real psol_1(real x, real t);
  @ axiom psol_1_def:
  @   \forall real x; \forall real t;
  @   \forall real eps; \exists real C; 0 < C && \forall real dx;
  @   \abs(dx) < C ==>
  @   \abs((psol(x + dx, t) - psol(x, t)) / dx - psol_1(x, t)) < eps;

  @ logic real psol_11(real x, real t);
  @ axiom psol_11_def:
  @   \forall real x; \forall real t;
  @   \forall real eps; \exists real C; 0 < C && \forall real dx;
  @   \abs(dx) < C ==>
  @   \abs((psol_1(x + dx, t) - psol_1(x, t)) / dx - psol_11(x, t)) < eps;

  @ logic real psol_2(real x, real t);
  @ axiom psol_2_def:
  @   \forall real x; \forall real t;
  @   \forall real eps; \exists real C; 0 < C && \forall real dt;
  @   \abs(dt) < C ==>
  @   \abs((psol(x, t + dt) - psol(x, t)) / dt - psol_2(x, t)) < eps;

  @ logic real psol_22(real x, real t);
  @ axiom psol_22_def:
  @   \forall real x; \forall real t;
  @   \forall real eps; \exists real C; 0 < C && \forall real dt;
  @   \abs(dt) < C ==>
  @   \abs((psol_2(x, t + dt) - psol_2(x, t)) / dt - psol_22(x, t)) < eps;

  @ axiom wave_eq_0: \forall real x; 0 <= x <= 1 ==> psol(x, 0) == p0(x);
  @ axiom wave_eq_1: \forall real x; 0 <= x <= 1 ==> psol_2(x, 0) == 0;
  @ axiom wave_eq_2:
  @   \forall real x; \forall real t;
  @   0 <= x <= 1 ==> psol_22(x, t) - c * c * psol_11(x, t) == 0;
  @ axiom wave_eq_dirichlet_1: \forall real t; psol(0, t) == 0;
  @ axiom wave_eq_dirichlet_2: \forall real t; psol(1, t) == 0;

  @ logic real psol_Taylor_3(real x, real t, real dx, real dt);
  @ logic real psol_Taylor_4(real x, real t, real dx, real dt);

  @ logic real alpha_3; logic real C_3;
  @ logic real alpha_4; logic real C_4;

  @ axiom psol_suff_regular_3:
  @   0 < alpha_3 && 0 < C_3 &&
  @   \forall real x; \forall real t; \forall real dx; \forall real dt;
  @   0 <= x <= 1 ==> \sqrt(dx * dx + dt * dt) <= alpha_3 ==>
  @   \abs(psol(x + dx, t + dt) - psol_Taylor_3(x, t, dx, dt)) <=
  @     C_3 * \abs(\pow(\sqrt(dx * dx + dt * dt), 3));
```

```
   @ axiom psol_suff_regular_4:
   @   0 < alpha_4 && 0 < C_4 &&
60 @   \forall real x; \forall real t; \forall real dx; \forall real dt;
   @   0 <= x <= 1 ==> \sqrt(dx * dx + dt * dt) <= alpha_4 ==>
   @   \abs(psol(x + dx, t + dt) - psol_Taylor_4(x, t, dx, dt)) <=
   @     C_4 * \abs(\pow(\sqrt(dx * dx + dt * dt), 4));

65 @ axiom psol_le:
   @   \forall real x; \forall real t;
   @   0 <= x <= 1 ==> 0 <= t ==> \abs(psol(x, t)) <= 1;

   @ logic real T_max;
70 @ axiom T_max_pos: 0 < T_max;

   @ logic real C_conv; logic real alpha_conv;
   @ lemma alpha_conv_pos: 0 < alpha_conv;
   @
75 @ } */


 /*@ axiomatic dirichlet_prog {
   @
80 @  predicate analytic_error{L}
   @    (double **p, integer ni, integer i, integer k, double a, double dt)
   @    reads p[..][..];
   @
   @  lemma analytic_error_le{L}:
85 @    \forall double **p; \forall integer ni; \forall integer i;
   @    \forall integer nk; \forall integer k;
   @    \forall double a; \forall double dt;
   @    0 < ni ==> 0 <= i <= ni ==> 0 <= k ==>
   @    0 < \exact(dt) ==>
90 @    analytic_error(p, ni, i, k, a, dt) ==>
   @    \sqrt(1. / (ni * ni) + \exact(dt) * \exact(dt)) < alpha_conv ==>
   @    k <= nk ==> nk <= 7598581 ==> nk * \exact(dt) <= T_max ==>
   @    \exact(dt) * ni * c <= 1 - 0x1.p-50 ==>
   @    \forall integer i1; \forall integer k1;
95 @    0 <= i1 <= ni ==> 0 <= k1 < k ==>
   @    \abs(p[i1][k1]) <= 2;
   @
   @  predicate separated_matrix{L}(double **p, integer leni) =
   @    \forall integer i; \forall integer j;
100 @   0 <= i < leni ==> 0 <= j < leni ==> i != j ==>
   @    \base_addr(p[i]) != \base_addr(p[j]);
   @
   @ logic real sqr_norm_dx_conv_err{L}
   @   (double **p, real dx, real dt, integer ni, integer i, integer k)
105 @   reads p[..][..];
   @ logic real sqr(real x) = x * x;
   @ lemma sqr_norm_dx_conv_err_0{L}:
   @   \forall double **p; \forall real dx; \forall real dt;
   @   \forall integer ni; \forall integer k;
110 @   sqr_norm_dx_conv_err(p, dx, dt, ni, 0, k) == 0;
   @ lemma sqr_norm_dx_conv_err_succ{L}:
   @   \forall double **p; \forall real dx; \forall real dt;
   @   \forall integer ni; \forall integer i; \forall integer k;
   @   0 <= i ==>
115 @   sqr_norm_dx_conv_err(p, dx, dt, ni, i + 1, k) ==
   @     sqr_norm_dx_conv_err(p, dx, dt, ni, i, k) +
   @     dx * sqr(psol(1. * i / ni, k * dt) - \exact(p[i][k]));
```

```
  @ logic real norm_dx_conv_err{L}
  @   (double **p, real dt, integer ni, integer k) =
  @   \sqrt(sqr_norm_dx_conv_err(p, 1. / ni, dt, ni, ni, k));
  @
  @ } */


/*@ requires leni >= 1 && lenj >= 1;
  @ ensures
  @   \valid_range(\result, 0, leni - 1) &&
  @   (\forall integer i; 0 <= i < leni ==>
  @     \valid_range(\result[i], 0, lenj - 1)) &&
  @   separated_matrix(\result, leni);
  @ */
double **array2d_alloc(int leni, int lenj);


/*@ requires (l != 0);
  @ ensures
  @   \round_error(\result) <= 14 * 0x1.p-52 &&
  @   \exact(\result) == p0(\exact(x));
  @ */
double p_zero(double xs, double l, double x);


/*@ requires
  @   ni >= 2 && nk >= 2 && l != 0 &&
  @   dt > 0. && \exact(dt) > 0. &&
  @   \exact(v) == c && \exact(v) == v &&
  @   0x1.p-1000 <= \exact(dt) &&
  @   ni <= 2147483646 && nk <= 7598581 &&
  @   nk * \exact(dt) <= T_max &&
  @   \abs(\exact(dt) - dt) / dt <= 0x1.p-51 &&
  @   0x1.p-500 <= \exact(dt) * ni * c <= 1 - 0x1.p-50 &&
  @   \sqrt(1. / (ni * ni) + \exact(dt) * \exact(dt)) < alpha_conv;
  @
  @ ensures
  @   \forall integer i; \forall integer k;
  @   0 <= i <= ni ==> 0 <= k <= nk ==>
  @   \round_error(\result[i][k]) <= 78. / 2 * 0x1.p-52 * (k + 1) * (k + 2);
  @
  @ ensures
  @   \forall integer k; 0 <= k <= nk ==>
  @   norm_dx_conv_err(\result, \exact(dt), ni, k) <=
  @     C_conv * (1. / (ni * ni) + \exact(dt) * \exact(dt));
  @ */
double **forward_prop(int ni, int nk, double dt, double v,
                      double xs, double l) {

  /* Output variable. */
  double **p;

  /* Local variables. */
  int i, k;
  double a1, a, dp, dx;

  dx = 1./ni;
  /*@ assert
    @   dx > 0. && dx <= 0.5 &&
    @   \abs(\exact(dx) - dx) / dx <= 0x1.p-53;
```

```
    @ */
180
  /* Compute the constant coefficient of the stiffness matrix. */
  a1 = dt/dx*v;
  a = a1*a1;
  /*@ assert
    @   0 <= a <= 1 &&
185 @   0 < \exact(a) <= 1 &&
    @   \round_error(a) <= 0x1.p-49;
    @ */

  /* Allocate space-time variable for the discrete solution. */
190 p = array2d_alloc(ni+1, nk+1);

  /* First initial condition and boundary conditions. */
  /* Left boundary. */
  p[0][0] = 0.;
195 /* Time iteration -1 = space loop. */
  /*@ loop invariant
    @   1 <= i <= ni &&
    @   analytic_error(p, ni, i - 1, 0, a, dt);
    @ loop variant ni - i; */
200 for (i=1; i<ni; i++) {
    p[i][0] = p_zero(xs, l, i*dx);
  }
  /* Right boundary. */
  p[ni][0] = 0.;
205 /*@ assert analytic_error(p, ni, ni, 0, a, dt); */

  /* Second initial condition (with p_one=0) and boundary conditions. */
  /* Left boundary. */
  p[0][1] = 0.;
210 /* Time iteration 0 = space loop. */
  /*@ loop invariant
    @   1 <= i <= ni &&
    @   analytic_error(p, ni, i - 1, 1, a, dt);
    @ loop variant ni - i; */
215 for (i=1; i<ni; i++) {
    /*@ assert \abs(p[i-1][0]) <= 2; */
    /*@ assert \abs(p[i][0]) <= 2; */
    /*@ assert \abs(p[i+1][0]) <= 2; */
    dp = p[i+1][0] - 2.*p[i][0] + p[i-1][0];
220   p[i][1] = p[i][0] + 0.5*a*dp;
  }
  /* Right boundary. */
  p[ni][1] = 0.;
  /*@ assert analytic_error(p, ni, ni, 1, a, dt); */
225
  /* Evolution problem and boundary conditions. */
  /* Propagation = time loop. */
  /*@ loop invariant
    @   1 <= k <= nk &&
230 @   analytic_error(p, ni, ni, k, a, dt);
    @ loop variant nk - k; */
  for (k=1; k<nk; k++) {
    /* Left boundary. */
    p[0][k+1] = 0.;
235 /* Time iteration k = space loop. */
    /*@ loop invariant
      @   1 <= i <= ni &&
```

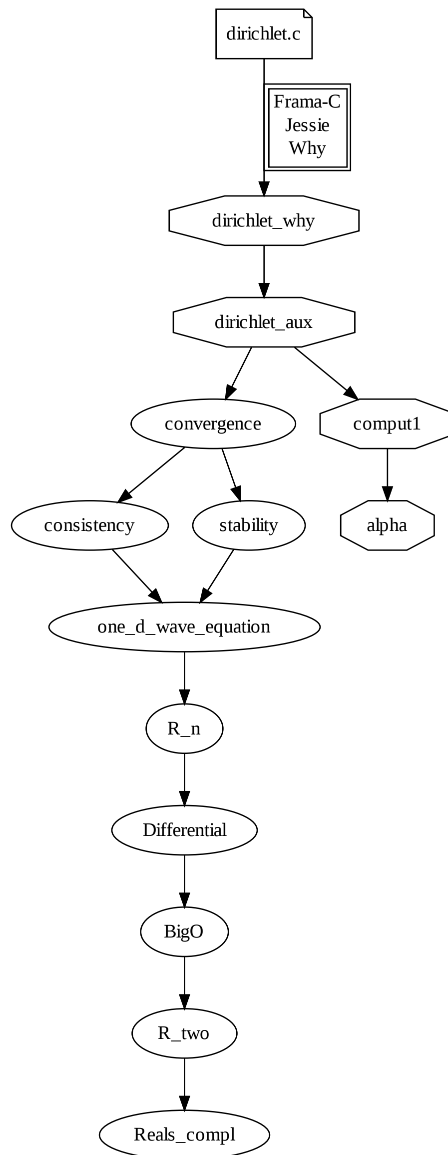
```
        @    analytic_error(p, ni, i - 1, k + 1, a, dt);
        @ loop variant ni - i; */
240     for (i=1; i<ni; i++) {
          /*@ assert \abs(p[i-1][k]) <= 2; */
          /*@ assert \abs(p[i][k]) <= 2; */
          /*@ assert \abs(p[i+1][k]) <= 2; */
          /*@ assert \abs(p[i][k-1]) <= 2; */
245       dp = p[i+1][k] - 2.*p[i][k] + p[i-1][k];
          p[i][k+1] = 2.*p[i][k] - p[i][k-1] + a*dp;
        }
        /* Right boundary. */
        p[ni][k+1] = 0.;
250     /*@ assert analytic_error(p, ni, ni, k + 1, a, dt); */
      }

      return p;

255 }
```

B Screenshot

This is a screenshot of gWhy: we have the list of all the verification conditions and if they are proved by the various automatic tools.

C Dependency Graph

In the following graph, the ellipse nodes are Coq files formalizing the wave equation and the convergence of its numerical scheme. The octagon nodes are Coq files that deal with proof obligations generated from the `dirichlet.c` program file, that is, propagation of round-off errors and error-free execution. Arrows represent dependencies between the Coq files.



Noname manuscript No.
(will be inserted by the editor)

Coloured Petri net refinement specification and correctness proof with Coq

Christine Choppy · Micaela Mayero ·
Laure Petrucci

Received: date / Accepted: date

Abstract In this work, we address the issue of the formal proof (using the proof assistant Coq) of refinement correctness for symmetric nets, a subclass of coloured Petri nets. We provide a formalisation of the net models, and of their type refinement in Coq. Then the Coq proof assistant is used to prove the refinement correctness lemma. An example adapted from a protocol example illustrates our work.

Keywords Refinement · coloured Petri nets · theorem proving

1 Introduction

Modelling and analysing large and complex systems requires elaborate techniques and support. To harness the problems inherent to designing and model-checking a large system (such as the state space explosion problem), a specification is often developed step-by-step. First, an abstract model is designed, and its properties are verified. Once it is proven correct, a refinement step takes place, introducing further detail. Such an addition can either be enhancing the description of the actual functioning of part of the system, or introducing an additional part. This new refined model is then verified, and another refinement step can take place. This process is applied until an adequate level of description is obtained.

There are several advantages to using refinement and hence to start with a more abstract model. It gives a better and more structured view of the system under study. The components within the system are clearly identified and the modelling process is eased. The modeller does not have to bother with spurious details. The validation process also becomes easier: the system properties are checked at each step. Thus, abstract models are validated before new details are added. Moreover, when model-checking is used, the analysis of a full concrete model may not be amenable, due to its very large state space. Refinement helps in coping with this problem since it may preserve some properties or analysis results obtained at an earlier step for a more abstract model may be used for the analysis of the refined model. For example, Lakos and Lewis [9] use the state space of the abstract model to compute the state space of the refined one:

LIPN, UMR CNRS 7030, Université Paris 13, FRANCE
E-mail: firstname.lastname@lipn.univ-paris13.fr

some tests for enabledness of transitions are avoided, as well as the construction of partial markings that have already been computed. Moreover, the state space can be structured. Hence, this approach saves both space and time in the analysis, countering the state space explosion problem.

In this paper, we consider specifications written as symmetric nets, a subclass of coloured Petri nets [7]. Lakos and Lewis [8,9] consider three kinds of Petri nets refinements for coloured Petri nets: node (place or transition) refinement, subnet refinement, and type refinement. Our work provides a formalisation in Coq [5] of both the abstract Petri net and the refined net, as well as refinement correctness lemmas, together with the refinement correctness proof as shown in Figure 1.

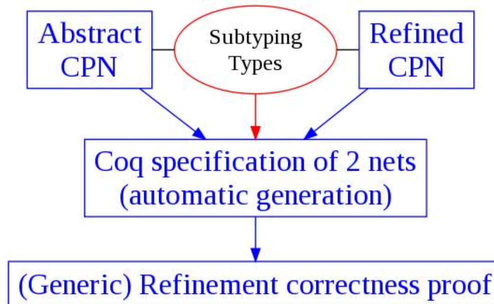


Fig. 1 Refinement model and proof

In a previous work [4], we considered mainly place/transition Petri nets (sketching how coloured nets might be taken into account), and the subnet refinement (the node refinement processing being similar).

In this paper, we address coloured nets, which are more complex in nature, and the formalisation in Coq had to be significantly changed to take the typing issues into account. The *type refinement* formalisation and correctness lemma are also addressed, thus pursuing the initial work of [4]. A protocol example adapted from [7] illustrates our work.

This type refinement is interesting since it allows for the specification of concrete and useful properties in practice. It requires a more complete formalisation, since colour sets (seen as types) are necessary. When compared to place/transition nets, the use of colours decreases the size of nets, leading to more amenable models.

The paper is structured as follows. Section 2 recalls the definitions of coloured Petri nets. Section 3 recalls the different Petri net refinements and provides a correctness lemma for the type refinement. Then, section 4 describes the case study (a protocol example) and formalisations for proving the type refinement of the net in this example. When the proof fails, it can still be used to obtain insight in the model, as explained in section 5. Conclusions and future work are finally discussed in section 6.

2 Coloured Petri nets definition

The definition of coloured Petri nets [8,9] used in this paper is the following:

Definition 1 (Coloured Petri net) A Coloured Petri net $\mathcal{N}$ is an 8-tuple $\mathcal{N} = \langle P, T, A, C, E, \mathbb{M}, \mathbb{Y}, M_0 \rangle$ such that:

1. P is a set of places
2. T is a set of transitions, such that $P \cap T = \emptyset$
3. A is a set of arcs, such that $A \subseteq (P \times T) \cup (T \times P)$
4. C: $P \cup T \rightarrow \Sigma$ where Σ is a universe of non-empty colour sets (or types), determines the colours of places and the transition modes.
5. E: $A \rightarrow \Phi\Sigma$ yields the arc inscriptions, such that $E(p,t), E(t,p)$: $C(t) \rightarrow \mu C(p)$
6. $\mathbb{M} = \mu\{(p,c)|p \in P, c \in C(p)\}$ is a set of markings that associate a multi-set of values c with each place p of P.
7. $\mathbb{Y} = \mu\{(t,c)|t \in T, c \in C(t)\}$ is a set of steps (multisets of transitions with their firing mode).
8. M_0 is the initial marking, $M_0 \in \mathbb{M}$.

where $\Phi\Sigma$ is a function over Σ defined by $\Phi\Sigma = \{X \rightarrow Y \mid X, Y \in \Sigma\}$ and $\mu X = \{X \rightarrow \mathbb{N}\}$ are multisets over a set X, and $\mathbb{N}$ is the set of natural numbers.

In the example in Figure 2, the marking of place `PacketsToSend` is the multiset `1‘1++1‘2++1‘3++1‘4++1‘5++1‘6`, where `1‘6` denotes one occurrence of value 6, and `++` denotes the multiset addition operator.

Definition 2 [8] The incremental effects $E^+, E^- : \mathbb{Y} \rightarrow \mathbb{M}$ of the occurrence of a step Y are given by:

1. $E^-(Y) = \sum_{(t,m) \in Y} \sum_{(p,t) \in A} \{p\} \times E(p,t)(m)$
2. $E^+(Y) = \sum_{(t,m) \in Y} \sum_{(t,p) \in A} \{p\} \times E(t,p)(m)$

E^- defines the input arc inscriptions while E^+ defines the output arc inscriptions.

Type refinement modifies the information carried by the tokens (a colour is a value of a token) while the net structure is unchanged. Type refinement brings additional information, which may be done e.g. by adding components in a tuple, or by representing an abstract data type by a more concrete one. The properties of the refined type should be preserved, that is if type A is refined by type B, then type B should satisfy the properties of A after an adequate syntactic translation. As for nets, it should always be possible to associate a behaviour of the abstract net with a behaviour of the refined one. The type refinement issue is associated with the issue of abstraction and implementation in the context of formal specifications (e.g. algebraic specifications [12]), and with studies on subtyping in the context of object-oriented programming languages [11, 3]. In this work (as in the work of Lakos [8,9]), the type refinement considered adds components in a tuple.

Since coloured Petri nets can use very general types and functions over these types which are thus not amenable, we here restrict ourselves to the *symmetric Petri nets* subclass [**?**]. Symmetric nets are defined as coloured Petri nets that allow only the use of particular types and functions: enumerated types, booleans, integer intervals, tuples and combinations of these, as well as the associated functions. We actually also handle lists of such types that can easily be manipulated by the Coq theorem prover.

3 Definitions of refinements

As mentioned previously, Lakos and Lewis [8, 9] consider three kinds of Petri nets refinements, node (place or transition) refinement, subnet refinement, and type refinement. *Node refinement* consists in replacing a place (transition) by a place- (transition-) bordered subnet. *Subnet refinement* consists in adding net components (places, transitions and arcs or even additional tokens). In this section the definition of type refinement is recalled, and we give the corresponding correctness lemma. The lemmas for subnet and node refinements can be found in [4]. Up to now, very little work has been achieved concerning type refinement.

In the following definition of *type refinement* [8], $\mathcal{N}_a = \langle P, T, A, C, E, \mathbb{M}, \mathbb{Y}, M_0 \rangle$ is the abstract net and $\mathcal{N}_r$ is the refined net.

Definition 3 (Type refinement) A morphism $\phi : \mathcal{N}_a \to \mathcal{N}_r$ is a *type refinement* if:

1. ϕ is the identity function on P, T, A, i.e. $\forall p \in P$: $\phi(p) = p$, etc.
2. $\forall x \in P \cup T$: $C(x) <: \phi(C)(x)$, i.e. $C(x)$ is a subtype of $\phi(C)(x)$
3. $\forall x \in P \cup T$: $\forall c \in C(x)$: $\phi(1`(x, c)) = 1`(x, \Pi_{\phi(C)(x)}(c))$, where Π is a projection function
4. $\forall (p, t) \in A$: $\forall (t, c) \in \mathbb{Y}$:
   $\phi(E^-(1`(t, c)))(p) = \Pi_{\phi(C)(p)}(E(p, t)(c)) = \phi(E)(p, t)(\Pi_{\phi(C)(t)}(c))$
   $\forall (t, p) \in A$: $\forall (t, c) \in \mathbb{Y}$:
   $\phi(E^+(1`(t, c)))(p) = \Pi_{\phi(C)(p)}(E(t, p)(c)) = \phi(E)(t, p)(\Pi_{\phi(C)(t)}(c))$

The following interpretation will be used in section 4.2 to formalise the type refinement in Coq:

Lemma 1

1. *The network structure (places, transitions and arcs) is kept unchanged, i.e.* $P = P'$, $T = T'$, $uA = uA'$ *where* P', T' *and* uA' *are resp. the sets of places, transitions and arcs (without their associated type) of the refined net while* P, T *and* uA *are those of the abstract net.*
2. *For any token* $1`(x', c')$*, of place* $x'$ *for colour* $c'$ *in the initial marking of the refined net, there exists a corresponding token* $1`(x, c)$ *in the initial marking of the abstract net. They must be such that both the subtyping and projection relations (resp. denoted* $<:$ *and* $\prod$*) are satisfied:* $c <: c'$ *and* $c = \prod_c(c')$*.*
3. *The arc expressions are refined according to the token refinement:* $\prod_{C_r(p)}(C_a(arc)) = C_r(arc)(C_r(t))$*.*

According to the formal definition of type refinement in [10], the net structure is unchanged. Since type refinement consists in incorporating additional information in token values, a token type in the refined net is a subtype of the one in the abstract net.

4 Case study: the simple protocol

4.1 Description

In this section, the correctness lemma is illustrated by a simple protocol example adapted from [7].

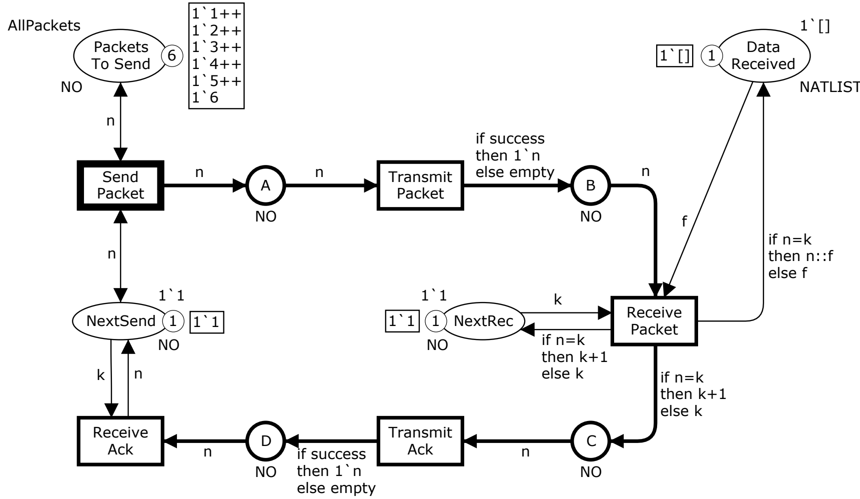


Fig. 2 Example of a simple protocol

```
colset BOOL = bool;
colset NO = int;
colset NATLIST = list NO;
var n,k : NO;
var f : NATLIST;
var success : BOOL;
val AllPackets = 1‘1++1‘2++1‘3++1‘4++1‘5++1‘6;
```

Fig. 3 Colour set declarations for the simple protocol

Figure 2 describes this simple protocol with the associated type (also called colour sets) declarations in figure 3. The left-hand side part models the sender, the right-hand side the receiver, while the middle part represents the network. The sender state is modelled by two places: *PacketsToSend* and *NextSend*. The receiver state is modelled by the *DataReceived* place. Places A, B, C and D constitute the network.

Note that place *PacketsToSend* is initially marked by six tokens with integer values. The textual inscription under a place is called “the colour set” of this place, which represents the available set of token colours. For example, the tokens in place *NextSend* always have an integer value. Here, the colour set `NO` is used to model sequence numbers. The inscription at the top right of place *NextSend* specifies that the initial marking of this place contains a single token with colour (value) 1, while the rectangle on the right exhibits the current marking and the circle contains the number of tokens. Informally, 1‘1 means that the data packet number 1 is to be sent. Finally, we will eventually obtain in place *DataReceived* a list of natural numbers: $[6, 5, 4, 3, 2, 1]$. Let us note that arc expressions yield token values together with their multiplicity. However, when the multiplicity is 1, it is omitted, thus n denotes 1‘n.

This example is refined by associating additional information with tokens (while the net structure in terms of places and transitions is unchanged). The refined net is presented in Figure 4 and the associated colour sets in Figure 5.

The colour sets of places *PacketsToSend*, A, B and *DataReceived* are extended from `NO` to `NOxDATA` which is defined as the cartesian product of the sets describing

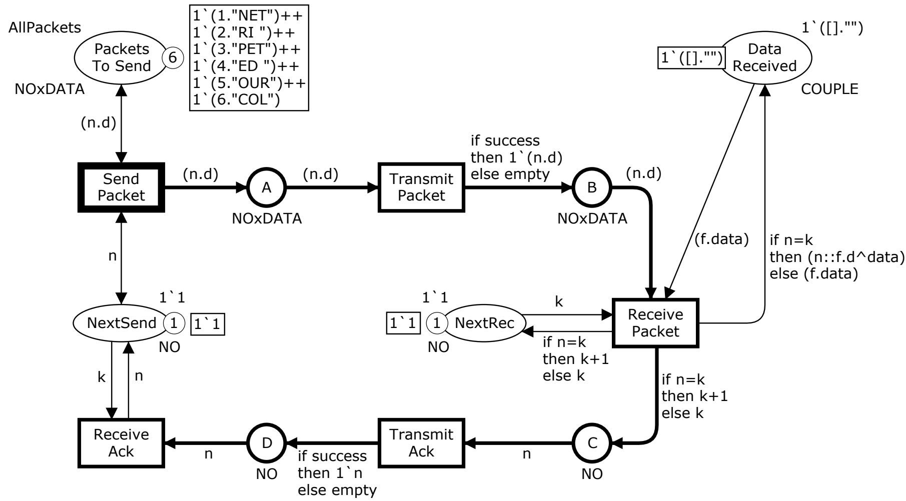


Fig. 4 Refined protocol example

```
colset BOOL = bool;
colset NO = int;
colset NATLIST = list NO;
colset DATA = string;
colset NOxDATA = product NO * DATA;
colset COUPLE = product NATLIST * DATA;
var n,k : NO;
var d,data : DATA;
var f : NATLIST;
var success : BOOL;
val AllPackets = 1'(1,"NET")++1'(2,"RI ")++1'(3,"PET")
               ++1'(4,"ED ")++1'(5,"OUR")++1'(6,"COL");
```

Fig. 5 Colour set declarations for the refined protocol

types `NO` and `DATA`. Note that here some places are not refined, e.g. *NextSend* is still of type `NO`.

The type refinement achieved in Figure 4 not only changes the type of tokens and places but also modifies the arcs expressions accordingly. Note that there exists a subtyping relation between e.g. n and (n, d) (the colour sets are given in Figures 3 and 5) on the arc between *PacketsToSend* and *SendPacket*.

4.2 Formalisation and proof in Coq

The simple protocol example is now formalised. Tokens carry complex information, and several functions are required to certify the system. In this example, the arc expressions are of four possible types (all with an integer multiplicity).

nat ∗	(nat)
	(nat ∗ string)
	(list nat)
	(list nat ∗ string)

In Coq, these kinds of arcs are defined by an inductive type:

```
Definition nat_Tuple := list nat.
Definition string_Tuple := list string.

Inductive arc_type : Type :=
  |bi_types: nat*nat -> arc_type
  |tri_types: nat*(nat*string) -> arc_type
  |bi_n_tuples: nat*nat_Tuple -> arc_type
  |tri_tuples: nat*(nat_Tuple*string_Tuples) -> arc_type.
```

Then, places and transitions are indexed by natural numbers:

```
Record Place : Type := mkPlace
        { Pr : nat }.

Record Transition : Type := mkTransition
        { Tr : nat }.
```

We now present an excerpt of the definitions of places, transitions and arcs for our example. Sets of places, transitions or arcs (both the untyped arc, i.e. the edge in the graph, and the arc expression) are represented by lists:

```
Definition P1_PacketsToSend := mkPlace 1.
...
Definition list_P:= P1_PacketsToSend::P2_A::P3_B::P4_NextRec::
                    P5_DataReceived::P6_C::P7_D::P8_NextSend::nil.

Definition T1_SendPacket := mkTransition 1.
...
Definition list_T:= T1_SendPacket::T2_TransmitPacket::T3_ReceivePacket::
                    T4_TransmitAck::T5_ReceiveAck::nil.

Definition uAP1T1 := (P1_PacketsToSend,T1_SendPacket).
...
Definition AP1T1 := (P1_PacketsToSend,T1_SendPacket,bi_types (1,n)).
...
Definition list_APT := AP1T1::AP2T2::AP3T3::AP4T3::AP5T3::
                       AP6T4::AP7T5::AP8T5::AP8T1::nil.
Definition list_ATP := AT1P1::AT1P2::AT1P8::AT2P3::AT3P4::
                       AT3P5::AT3P6::AT4P7::AT5P8::nil.
Definition list_uATP := uAT1P1...
...
Definition list_APT' := A'P1T1::A'P2T2::A'P3T3::A'P4T3::A'P5T3::
                        A'P6T4::A'P7T5::A'P8T5::A'P8T1::nil.
Definition list_ATP' := A'T1P1::A'T1P2::A'T1P8::A'T2P3::A'T3P4::
                        A'T3P5::A'T3P6::A'T4P7::A'T5P8::nil.
```

The most interesting aspect of type refinement is due to arc expressions. Type refinement can be seen as a relation between types, which is subtyping. For example, the following table presents subtyping relations involved in our refinement of the simple protocol (note that the first line is unchanged by the refinement, and this still needs to be checked).

ARC EXPRESSIONS	EXAMPLE OF ARC VALUES	VALUE TYPE	COQ arc_type
if n=k then k+1 else k	1‘6	nat*nat	bi_types
n (n, d)	1‘6 1‘(6, ”COL”)	nat*nat nat*(nat*string)	bi_types tri_types
if n=k then n::f else f if n=k then (n::f, d∧data) else (f, data)	1‘[6] 1‘([6], ”COL”)	nat*list nat nat*(list nat*list string)	bi_n_tuples tri_tuples
f (f, data)	1‘[6]) 1‘(6, ”COL”)	nat*list nat nat*(list nat*list string)	bi_n_tuples tri_tuples

The subtyping relation must be formalised for this example. We begin by defining a function `is_sub` which gives a relation between types of `arc_type`. This relation is then extended to tuples (`is_sub_tupl_apt`) and lists of tuples (`is_sub_l_apt`) for describing arcs from places to transitions. Similar extensions are defined for arcs from transitions to places.

```
Definition is_sub (subtyp:arc_type)(typ:arc_type) : Prop :=
  match subtyp, typ with
  | (bi_types _), (bi_types _) => True
  | (tri_types _), (bi_types _) => True
  | (tri_tuples _), (bi_n_tuples _) => True
  | _, _ => False
  end.

Definition is_sub_tupl_apt (subtupl: Place * Transition * arc_type)
           (tupl: Place * Transition * arc_type) : Prop :=
           (is_sub (snd subtupl) (snd tupl)).

Fixpoint is_sub_l_apt (subl: list (Place * Transition * arc_type))
         (l: list (Place * Transition * arc_type)) {struct subl} : Prop :=
    match subl, l with
     | nil, nil => True
     | (cons a tla), (cons b tlb) =>
            (is_sub_tupl_apt a b) /\ (is_sub_l_apt tla tlb)
     | _, _ => False
    end.
```

In this code, several operators are used: `snd` returns the second component of a pair, `nil` denotes the empty list and `cons` is the list constructor, adding an element at the head of a list.

The type refinement correctness lemma can now be written, with the help of lemma 1 (where, for the sake of readability, we indicate in parenthesis to which item the COQ code relates):

```
Lemma type_colour_refined:
      eqlist Place list_P list_P' /\                      (1.)
      eqlist Transition list_T list_T' /\                 (1.)
      eqlist (Place*Transition) list_uAPT list_uAPT' /\    (1.)
      eqlist (Transition*Place) list_uATP list_uATP' /\    (1.)
      eqlist (list (nat*nat)) list_MP (hd_list list_MP')/\ (2.)
```

```
      is_sub_l_apt list_APT' list_APT /\                    (3.)
      is_sub_l_atp list_ATP' list_ATP.                      (3.)
```

where `eqlist` is an equality between lists and $l = l'$ is equivalent to $l \subseteq l'$ and $l' \subseteq l$, `list_MP` and `list_MP'` define the initial markings, and function `hd_list` is defined as follows:

```
Fixpoint hd_list_couple (l:list (nat*(nat*string))):=
     match l with
       | nil=>nil
       |(a,(b,c))::tl=>(a,b)::(hd_list_couple tl)
     end.

Fixpoint hd_list (l:list (list (nat*(nat*string)))):=
      match l with
        |nil=>nil
        |a::tl=>(hd_list_couple a)::(hd_list tl)
      end.
```

Thanks to our simple and general formalisation, the formal correctness proof is almost automatic.

```
Proof.
repeat split;unfold incl;tauto.
Qed.
```

Note that this simple formalisation was obtained after carefully studying different possibilities for encoding Petri net elements in Coq which are detailed in [4]. Moreover, the proof could be simplified using powerful constructs such as the `split` tactic, which is particularly well-suited for our purposes. This tactic applies to inductive types with a single constructor, which is the case for the `/\` operator in the lemma `type_colour_refined`. While in [4] we used more traditional and complex proofs, we since got a better insight that lead us to produce a much simpler and more powerful Coq proof that is independent of the example considered.

The full development is available at `http://www-lipn.univ-paris13.fr/~mayero/CPNCoq/Jensen_protocol_JNASA.v`.

5 When the proof fails

When the proof fails, it still gives valuable information as regards the refinement to be proven: either the lists representing the net graph elements (places, transitions, or arcs) do not match, and the refinement relation does not hold ; or the error occurs when examining arc expressions. It may then be the case that the refinement property does not hold, but also that the type refinement between the supposedly refined and abstract arc expression cannot be automatically proven.

In order to try to know in which case we are, we have to render the Coq proof less automatic. For example, suppose that the subtyping function `is_sub` is erroneous. In

this function, let us now assume that the following line is missing:
`| (tri_tuples _), (bi_n_tuples _) => True`

This means that: `| (tri_tuples _), (bi_n_tuples _) => False`

In this case, the proof script "repeat split;unfold incl;tauto." provides the following error message:

```
"File "./Jensen_protocol_NFM_detect_error.v", line 266, characters 25-30
User error: Tauto failed"
```

And the goal remains unchanged:

```
type_colour_refined < Show.
1 subgoal

  ============================
   eqlist Place list_P list_P' /\
   eqlist Transition list_T list_T' /\
   eqlist (Place * Transition) list_uAPT list_uAPT' /\
   eqlist (Transition * Place) list_uATP list_uATP' /\
   eqlist (list (nat * nat)) list_MP (hd_list list_MP') /\
   is_sub_l_apt list_APT' list_APT /\ is_sub_l_atp list_ATP' list_ATP
```

From this error message, we know that the problem is detected by the tactic `tauto`. Due to the high level of proof automation, we have to begin by finding which goal is concerned. To do so, it is possible in Coq to catch errors using `try` :

```
repeat split;unfold incl;try tauto.
```

If the proof does not terminate then Coq leaves subgoals to be proved (instead of failing). The user can interactively see where the problems are. In our example, it is the last two subgoals :

```
type_colour_refined < repeat split;unfold incl;try tauto.
2 subgoals

  ============================
   is_sub_tupl_apt A'P5T3 AP5T3

subgoal 2 is:
 is_sub_tupl_atp A'T3P5 AT3P5
```

We can further detail the proof:

```
unfold is_sub_tupl_apt;unfold is_sub;simpl.
```

Thanks to the interactive interface of Coq (called toplevel), we can see that we have to prove False (instead of True)! This is due to the fact that the function `is_sub` is not correct.

```
type_colour_refined < unfold is_sub_tupl_apt;unfold is_sub;simpl.
2 subgoals

  ============================
   False
```

This means that a subtyping property is missing.

6 Conclusion

When modelling and validating critical systems, one often proceeds in a step-by-step fashion: a first abstract model is designed and validated ; it is then refined so as to take into account additional details ; and this process is repeated as many times as necessary. In order to guarantee that the behaviour of the system is preserved by refinement, it should obey some rules. Three kinds of refinements of coloured Petri nets were formally defined in [9]. Our aim here was to show that the proof of refinement — i.e. that a refined net actually is a refinement of an abstract net — can be automated using theorem-proving techniques, thus avoiding error-prone and lengthy manual proofs.

Previous work focussed on two kinds of refinements: node refinement and subnet refinement, while the third one was scarcely mentioned. This paper has shown that when restricting coloured Petri nets to an appropriate subclass, type refinement can also be handled.

This work confirms that our choices of formalisation (with a first version in [4]) are suitable. The prerequisite to the refinements is the formalisation of a given Petri net. This formalisation is probably the most tedious part of our work and requires a significant automation. Since the refinement issues we tackle are meant to be integrated within a step-by-step modelling process, the refined net should be designed by the user starting from the abstract net. Therefore, places and transitions that are in both nets should remain exactly the same and can be identified by their name. Hence, we do not address the problem of proving that a net is a refinement of another one, starting from arbitrary nets.

The possibility to easily integrate automation is a key issue in the work presented in this paper. For example, as seen in section 4, to define all the places, all the transitions and all the arcs manually is certainly not efficient, especially if the net has more than 50 places/transitions. To solve this problem, an interface to PNML (Petri Net Markup Language, [2]) is under development, using PNML Framework [**?**]. PNML is part of the ISO/IEC 15909-2 standard (since November 2009). It aims at becoming the common language for Petri nets tools, e.g. CPN-AMI [1], CPN-Tools [6] or other tools supporting Petri nets. Such files can be directly translated into Coq to generate the places, transitions and arcs.

We think that our method scales up rather well. Indeed, the proof is generic and does not change with the nets considered. The only modifications are subtyping re-

lations and type definitions. Moreover, when proceeding step-by-step, refinements are applied one at a time. Therefore, the nets to be considered are only slightly different.

To complete this work, we should consider refinement as part of a modular design process. In such a framework, a type refinement can affect several modules which could be checked separately for refinement, and one must ensure that type refinement has been applied consistently in all modules.

Acknowledgements The implementation of this work in COQ was achieved with the help of Yibei Yu, a trainee supervised by the authors. We would like to thank Charles Lakos for fruitful discussions.

References

1. CPN-AMI*: Home Page*. http://www-src.lip6.fr/logiciels/mars/CPNAMI/.
2. J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri Net Markup Language: Concepts, technology and tools. In *Proc. 24th Int. Conf. Application and Theory of Petri Nets (ICATPN'2003), Eindhoven, The Netherlands, June 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 483–505. Springer, 2003.
3. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985.
4. Christine Choppy, Micaela Mayero, and Laure Petrucci. Experimenting Formal Proofs of Petri Nets Refinements. *Electr. Notes Theor. Comput. Sci.*, 214:231–254, 2008.
5. *The Coq proof assistant*. http://coq.inria.fr.
6. *cpntools*. http://wiki.daimi.au.dk/cpntools/cpntools.wiki.
7. Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets, Modelling and Validation of Concurrent Systems*. Monograph to be published by Springer Verlag, 2008.
8. Charles Lakos. Composing abstractions of coloured Petri nets. In Nielsen, M. and Simpson, D., editors, *Lecture Notes in Computer Science: 21st International Conference on Application and Theory of Petri Nets (ICATPN 2000), Aarhus, Denmark, June 2000*, volume 1825, pages 323–345. Springer-Verlag, 2000.
9. Charles Lakos and Glen Lewis. Incremental state space construction of coloured Petri nets. In *Proc. 22nd Int. Conf. Application and Theory of Petri Nets (ICATPN'01), Newcastle, UK, June 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 263–282. Springer, 2001.
10. Glen Lewis. *Incremental specification and analysis in the context of coloured Petri nets*. PhD thesis, University of Hobart, Tasmania, 2002.
11. Barbara Liskov and Jeannette M. Wing. A new definition of the subtype relation. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 118–141, London, UK, 1993. Springer-Verlag.
12. Fernando Orejas, Marisa Navarro, and Ana Sanchez. Algebraic implementation of abstract data types: a survey of concepts and new compositionality results. *Mathematical Structures in Computer Science*, pages 33–67, 1996.

Rigorous Polynomial Approximation Using Taylor Models in Coq*

Nicolas Brisebarre[1], Mioara Joldeş[4], Érik Martin-Dorel[1],
Micaela Mayero[1,2], Jean-Michel Muller[1], Ioana Paşca[1],
Laurence Rideau[3], and Laurent Théry[3]

[1] LIP, CNRS UMR 5668, ENS de Lyon, INRIA Grenoble - Rhône-Alpes, UCBL, Arénaire, Lyon, F-69364
[2] LIPN, UMR 7030, Université Paris 13, LCR, Villetaneuse, F-93430
[3] Marelle, INRIA Sophia Antipolis - Méditerranée, Sophia Antipolis, F-06902
[4] CAPA, Dpt. of Mathematics, Uppsala Univ., Box S-524, 75120, Uppsala, Sweden

Abstract. One of the most common and practical ways of representing a real function on machines is by using a polynomial approximation. It is then important to properly handle the error introduced by such an approximation. The purpose of this work is to offer guaranteed error bounds for a specific kind of rigorous polynomial approximation called Taylor model. We carry out this work in the Coq proof assistant, with a special focus on genericity and efficiency for our implementation. We give an abstract interface for rigorous polynomial approximations, parameterized by the type of coefficients and the implementation of polynomials, and we instantiate this interface to the case of Taylor models with interval coefficients, while providing all the machinery for computing them. We compare the performances of our implementation in Coq with those of the Sollya tool, which contains an implementation of Taylor models written in C. This is a milestone in our long-term goal of providing fully formally proved and efficient Taylor models.

Keywords: certified error bounds, Taylor models, Coq proof assistant, rigorous polynomial approximation.

1 Rigorous Approximation of Functions by Polynomials

It is frequently useful to replace a given function of a real variable by a simpler function, such as a polynomial, chosen to have values close to those of the given function, since such an approximation may be more compact to represent and store but also more efficient to evaluate and manipulate. As long as evaluation is concerned, polynomial approximations are especially important. In general the basic functions that are implemented in hardware on a processor are limited to addition, subtraction, multiplication, and sometimes division. Moreover, division is significantly slower than multiplication. The only functions of one variable

* This research was supported by the TaMaDi project of the French ANR (ref. ANR-2010-BLAN-0203-01).

A. Goodloe and S. Person (Eds.): NFM 2012, LNCS 7226, pp. 85–99, 2012.
© Springer-Verlag Berlin Heidelberg 2012

that one may evaluate using a bounded number of additions/subtractions, multiplications and comparisons are piecewise polynomials: hence, on such systems, polynomial approximations are not only a good choice for implementing more complex functions, they are frequently the only one that makes sense.

Polynomial approximations for widely used functions used to be tabulated in handbooks [1]. Nowadays, most computer algebra systems provide routines for obtaining polynomial approximations of commonly used functions. However, when bounds for the approximation errors are available, they are not guaranteed to be accurate and are sometimes unreliable.

Our goal is to provide efficient and quickly computable *rigorous polynomial approximations*, i.e., polynomial approximations for which (i) the provided error bound is tight and not underestimated, (ii) the framework is suitable for formal proof (indeed, the computations are done in a formal proof checker), while requiring computation times similar to those of a conventional C implementation.

1.1 Motivations

Most numerical systems depend on standard functions like exp, sin, etc., which are implemented in libraries called `libms`. These `libms` must offer guarantees regarding the provided accuracy: they are heavily tested before being published, but for precisions higher that single precision, an exhaustive test is impossible [16]. Hence a proof of the behavior of the program that implements a standard function should come with it, whenever possible. One of the key elements of such a proof would be the guarantee that the used polynomial approximation is within some threshold from the function. This requirement is even more important when *correct rounding* is at stake. Most `libms` do not provide correctly rounded functions, although the IEEE 754-2008 Standard for Floating-Point (FP) Arithmetic [22] recommends it for a set of basic functions. Implementing a correctly rounded function requires rigorous polynomial approximations at two steps: when actually implementing the function in a given precision, and—before that—when trying to solve the *table maker's dilemma* for that precision.

The 1985 version of the IEEE Standard for FP Arithmetic requires that the basic operations ($+$, $-$, $\times$, $\div$, and $\sqrt{\cdot}$) should produce *correctly rounded results*, as if the operations were first carried out in infinite precision and these intermediate results were then rounded. This contributed to a certain level of portability and provability of FP algorithms. Until 2008, there was no such analogous requirement for standard functions. The main impediment for this was the *table maker's dilemma*, which can be stated as follows: consider a function f and a FP number x. In most cases, $y = f(x)$ cannot be represented exactly. The correctly rounded result is the FP number that is closest to y. Using a finite precision environment, only an approximation $\hat{y}$ to y can be computed. If that approximation is not accurate enough, one cannot decide the correct rounding of y from $\hat{y}$. Ziv [41] suggested to improve the accuracy of the approximation until the correctly rounded value can be decided. A first improvement over that approach derives from the availability of tight bounds on the worst-case accuracy required to compute some functions [25], which made it possible to write a `libm` with

correctly rounded functions, where correct rounding is obtained at modest additional costs [37]. The TaMaDi project [32] aims at computing the worst-case accuracy for the most common functions and formats. Doing this requires very accurate polynomial approximations that are formally verified.

Beside the Table Maker's Dilemma, the implementation of correctly rounded elementary functions is a complex process, which includes finding polynomial approximations for the considered function that are accurate enough. Obtaining good polynomial approximations is detailed in [10,9,12]. In the same time, the approximation error between the function and the polynomial is very important since one must make sure that the approximation is good enough. The description of a fast, automatic and verifiable process was given in [23].

In the context of implementing a standard function, we are interested in finding polynomial approximations for which, given a degree n, the maximum error between the function and the polynomial is minimum: this "minimax approximation" has been broadly developed in the literature and its application to function implementation is discussed in detail in [12,33]. Usually this approximation is computed numerically [38], so an a posteriori error bound is needed. Obtaining a tight bound for the approximation error reduces to computing a tight bound for the supremum norm of the error function over the considered interval. Absolute error as well as relative errors can be considered. For the sake of simplicity, in this paper, we consider absolute errors only (relative errors would be handled similarly). Our problem can be seen as a univariate rigorous global optimization problem, however, obtaining a tight and formally verified interval bound for the supremum norm of the error function presents issues unsuspected at a first sight [14], so that techniques like interval arithmetic and Taylor models are needed. An introduction to these concepts is given below.

Interval arithmetic and Taylor models. The usual arithmetic operations and functions are straightforwardly extended to handle intervals. One use of interval arithmetic is bounding the image of a function over an interval. Interval calculations frequently overestimate the image of a function. This phenomenon is in general proportional to the width of the input interval. We are therefore interested in using thin input intervals in order to get a tight bound on the image of the function. While subdivision methods are successfully used in general, when trying to solve this problem, one is faced with what is known as a "dependency phenomenon": since function f and its approximating polynomial p are highly correlated, branch and bound methods based on using intervals of smaller width to obtain less overestimation, end up with an unreasonably high number of small intervals. To reduce the dependency, *Taylor models* are used. They are a basic tool for replacing functions with a polynomial and an interval remainder bound, on which basic arithmetic operations or bounding methods are easier.

1.2 Related Work

Taylor models [27,35,28] are used for solving rigorous global optimization problems [27,5,14,6] and obtaining validated solutions of ODEs [34] with applications

to critical systems like particle accelerators [6] or robust space mission design [26]. Freely available implementations are scarce. One such implementation is available in Sollya [13]. It handles univariate functions only, but provides multiple-precision support for the coefficients. It was used for proving the correctness of supremum norms of approximation errors in [14], and so far it is the only freely available tool that provides such routines. However, this remains a C implementation that does not provide formally proved Taylor models, although this would be necessary for having a completely formally verified algorithm.

There have been several attempts to formalize Taylor models (TMs) in proof assistants. An implementation of multivariate TMs is presented in [42]. They are implemented on top of a library of exact real arithmetic, which is more costly than FP arithmetic. The purpose of that work is different than ours. It is appropriate for multivariate polynomials with small degrees, while we want univariate polynomials and high degrees. There are no formal proofs for that implementation. An implementation of univariate TMs in PVS is presented in [11]. Though formally proved, it contains ad-hoc models for a few functions only, and it is not efficient enough for our needs, as it is unable to produce Taylor models of degree higher than 6. Another formalization of Taylor models in Coq is presented in [15]. It uses polynomials with FP coefficients. However, the coefficients are axiomatized, so we cannot compute the actual Taylor model in that implementation. We can only talk about the properties of the involved algorithms.

Our purpose is to provide a modular implementation of univariate Taylor models in Coq, which is efficient enough to produce very accurate approximations of elementary real functions. We start by presenting in Section 2 the mathematical definitions of Taylor models as well as efficient algorithms used in their implementation. We then present in Section 3 the Coq implementation. Finally we evaluate in Section 4 the quality of our implementation, both from the point of view of efficient computation and of numerical accuracy of the results.

2 Presentation of the Taylor Models

2.1 Definition, Arithmetic

A Taylor model (TM) of order n for a function f which is supposed to be $n+1$ times differentiable over an interval $[a,b]$, is a pair $(T, \boldsymbol{\Delta})$ formed by a polynomial T of degree n, and an interval part $\boldsymbol{\Delta}$, such that $f(x) - T(x) \in \boldsymbol{\Delta}, \forall x \in [a,b]$. The polynomial can be seen as a Taylor expansion of the function at a given point. The *interval remainder* $\boldsymbol{\Delta}$ provides an enclosure of the approximation errors encountered (truncation, roundings).

For usual functions, the polynomial coefficients and the error bounds are computed using the Taylor-Lagrange formula and recurrence relations satisfied by successive derivatives of the functions. When using the same approach for composite functions, the error we get for the remainder is too pessimistic [14]. Hence, an arithmetic for TMs was introduced: simple algebraic rules like addition, multiplication and composition with TMs are applied recursively on the structure of function f, so that the final model obtained is a TM for f over

$[a,b]$. Usually, the use of these operations with TMs offers a much tighter error bound than the one directly computed for the whole function [14]. For example, addition is defined as follows: let two TMs of order n for f_1 and f_2, over $[a,b]$: $(P_1, \boldsymbol{\Delta}_1)$ and $(P_2, \boldsymbol{\Delta}_2)$. Their sum is an order n TM for $f_1 + f_2$ over $[a,b]$ and is obtained by adding the two polynomials and the remainder bounds: $(P_1, \boldsymbol{\Delta}_1) + (P_2, \boldsymbol{\Delta}_2) = (P_1 + P_2, \boldsymbol{\Delta}_1 + \boldsymbol{\Delta}_2)$. For multiplication and composition, similar rules are defined.

We follow the definitions in [23,14], and represent the polynomial T with *tight interval coefficients*. This choice is motivated by the ease of programming (rounding errors are directly handled by the interval arithmetic) and also by the fact that we want to ensure that the true coefficients of the Taylor polynomial lie inside the corresponding intervals. This is essential for applications that need to handle removable discontinuities [14]. For our formalization purpose, we recall and explain briefly in what follows the definition of valid Taylor models [23, Def. 2.1.3], and refer to [23, Chap. 2] for detailed algorithms regarding operations with Taylor models for univariate functions.

2.2 Valid Taylor Models

A Taylor model for a function f is a pair $(T, \boldsymbol{\Delta})$. The relation between f and $(T, \boldsymbol{\Delta})$ can be rigorously formalized as follows.

Definition 1. *Let $f : \boldsymbol{I} \to \mathbb{R}$ be a function, $\boldsymbol{x_0}$ be a small interval around an expansion point x_0. Let T be a polynomial with interval coefficients $\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}$ and $\boldsymbol{\Delta}$ an interval. We say that $(T, \boldsymbol{\Delta})$ is a Taylor model of f at $\boldsymbol{x_0}$ on $\boldsymbol{I}$ when*

$$\begin{cases} \boldsymbol{x_0} \subseteq \boldsymbol{I} \text{ and } 0 \in \boldsymbol{\Delta}, \\ \forall \xi_0 \in \boldsymbol{x_0}, \exists \alpha_0 \in \boldsymbol{a_0}, \ldots, \alpha_n \in \boldsymbol{a_n}, \forall x \in \boldsymbol{I}, \exists \delta \in \boldsymbol{\Delta}, f(x) - \sum_{i=0}^{n} \alpha_i \left(x - \xi_0\right)^i = \delta. \end{cases}$$

Informally, this definition says that there is always a way to pick some values α_i in the intervals a_i so that the difference between the resulting polynomial and f around $\boldsymbol{x_0}$ is contained in $\boldsymbol{\Delta}$. This validity is the invariant that is preserved when performing operations on Taylor models. Obviously, once a Taylor model $(T, \boldsymbol{\Delta})$ is computed, if needed, one can get rid of the interval coefficients $\boldsymbol{a_i}$ in T by picking arbitrary α_i and accumulating in $\boldsymbol{\Delta}$ the resulting errors.

2.3 Computing the Coefficients and the Remainder

We are now interested in an automatic way of providing the terms $\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}$ and $\boldsymbol{\Delta}$ of Definition 1 for basic functions. It is classical to use the following.

Lemma 1 (Taylor-Lagrange Formula). *If f is $n+1$ times differentiable on a domain $\boldsymbol{I}$, then we can expand f in its Taylor series around any point $x_0 \in \boldsymbol{I}$ and we have: $\forall x \in \boldsymbol{I}$, $\exists \xi$ between x_0 and x such that*

$$f(x) = \underbrace{\left(\sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!} \left(x - x_0\right)^i\right)}_{T(x)} + \underbrace{\frac{f^{(n+1)}(\xi)}{(n+1)!} \left(x - x_0\right)^{n+1}}_{\Delta(x,\xi)}.$$

Computing interval enclosures $\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}$, for the coefficients of T, reduces to finding enclosures of the first n derivatives of f at x_0 in an efficient way. The same applies for computing $\boldsymbol{\Delta}$ based on an interval enclosure of the $n+1$ derivative of f over $\boldsymbol{I}$. However, the expressions for successive derivatives of practical functions typically become very involved with increasing n. Fortunately, it is not necessary to generate these expressions for obtaining values of $\{f^{(i)}(x_0), i = 0, \ldots, n\}$. For basic functions, formulas are available since Moore [31] (see also [21]). There one finds either recurrence relations between successive derivatives of f, or a simple closed formula for them. And yet, this is a case-by-case approach, and we would like to use a more generic process, which would allow us to deal with a broader class of functions in a more uniform way suitable to formalization.

Recurrence Relations for D-finite Functions. An algorithmic approach exists for finding recurrence relations between the Taylor coefficients for a class of functions that are solutions of linear ordinary differential equations (LODE) with polynomial coefficients, called *D-finite functions*. The Taylor coefficients of these functions satisfy a linear recurrence with polynomial coefficients [40]. Most common functions are *D-finite*, while a simple counter-example is tan. For any D-finite function one can generate the recurrence relation directly from the differential equation that defines the function, see, e.g., the Gfun module in Maple [39]. From the recurrence relation, the computation of the first n coefficients is done in linear time. Let us take a simple example and consider $f = \exp$. It satisfies the LODE $f' = f$, $f(0) = 1$, which gives the following recurrence for the Taylor coefficients $(c_n)_{n\in\mathbb{N}}$: $(n+1)c_{n+1} - c_n = 0$, $c_0 = 1$, whose solution is $c_n = 1/n!$.

This property lets us include in the class of *basic functions* all the D-finite functions. We will see in Section 3.2 that this allows us to provide a uniform and efficient approach for computing Taylor coefficients, suitable for formalization. We note that our data structure for that is *recurrence relation + initial conditions* and that the formalization of the isomorphic transformation from the *LODE + initial conditions*, used as input in Gfun is subject of future research.

3 Formalization of Taylor Models in Coq

We provide an implementation[1] of TMs that is efficient enough to produce very accurate approximating polynomials in a reasonable amount of time. The work is carried out in the Coq proof assistant, which provides a formal setting where we will be able to formally verify our implementation. We wish to be as generic as possible. A TM is just an instance of a more general object called *rigorous polynomial approximation* (RPA). For a function f, a RPA is a pair $(T, \boldsymbol{\Delta})$ where T is a polynomial and $\boldsymbol{\Delta}$ an interval containing the approximation error between f and T. We can choose Taylor polynomials for T and get TMs but other types of approximation are also available like Chebyshev models, based on Chebyshev polynomials. This generic RPA structure will look like:

[1] It is available at `http://tamadi.gforge.inria.fr/CoqApprox/`

```
Structure rpa := { approx: polynomial; error: interval }
```

In this structure, we want genericity not only for `polynomial` with respect to the type of its coefficients and to its physical implementation but also for the type for intervals. Users can then experiment with different combinations of datatypes. Also, this genericity lets us factorize our implementation and will hopefully facilitate the proofs of correctness. We implement Taylor models as an instance of a generic RPA following what is presented in Section 2. Before describing our modular implementation, we present the Coq proof assistant, the libraries we have been using and how computation is handled.

3.1 The Coq Proof Assistant

Coq [4] is an interactive theorem prover that combines a higher-order logic and a richly-typed functional programming language. Thus, it provides an expressive language for defining not only mathematical objects but also datatypes and algorithms and for stating and proving their properties. The user builds proofs in Coq in an interactive manner. In our development, we use the SSReflect [19] extension that provides its own tactic language and libraries.

There are two main formalizations of real numbers in Coq: an axiomatic one [29] and a constructive one [18]. For effective computations, several implementations of computable real numbers exist. A library for multiple-precision FP arithmetic is described in [8]. Based on this library, an interval arithmetic library is defined in [30]. It implements intervals with FP bounds. Also, the libraries [36] and [24] provide an arbitrary precision real arithmetic. All these libraries are proved correct by deriving a formal link between the computational reals and one of the formalizations of real numbers. We follow the same idea: implement a computable TM for a given function and formally prove its correctness with respect to the abstract formalization of that function in Coq. This is done by using Definition 1 and the functions defined in the axiomatic formalization.

The logic of Coq is computational: it is possible to write programs in Coq that can be directly executed within the logic. This is why the result of a computation with a correct algorithm can always be trusted. Thanks to recent progress in the evaluation mechanism [7], a program in Coq runs as fast as an equivalent version written directly and compiled in OCaml. There are some restrictions to the programs that can be executed in Coq: they must always terminate and be purely functional, i.e., no side-effects are allowed. This is the case for the above mentioned computable real libraries. Moreover, they are defined within Coq on top of the multiple-precision arithmetic library based on binary tree described in [20]. So only the machine modular arithmetic (32 or 64 bits depending on the machine) is used in the computations in Coq.

For our development of Taylor models we use polynomials with coefficients being some kind of computable reals. Following the description in Section 2, we use intervals with FP bounds given by [30] as coefficients. Since the interval and FP libraries are proved correct, so is the arithmetic on our coefficients. By choosing a functional implementation for polynomials (e.g., lists), we then obtain

TMs that are directly executable within Coq. Now, we describe in detail this modular implementation.

3.2 A Modular Implementation of Taylor Models

Coq provides three mechanisms for modularization: *type classes*, *structures*, and *modules*. Modules are less generic than the other two (that are first-class citizens) but they have a better computational behavior: module applications are performed statically, so the code that is executed is often more compact. Since our generic implementation only requires simple parametricity, we have been using modules.

First, abstract interfaces called `Module Type`s are defined. Then concrete "instances" of these abstract interfaces are created by providing an implementation for all the fields of the `Module Type`. The definition of `Module`s can be parameterized by other `Module`s. These parameterized modules are crucial to factorize code in our structures.

Abstract Polynomials, Coefficients and Intervals. We describe abstract interfaces for *polynomials* and for their *coefficients* using Coq's `Module Type`. The interface for coefficients contains the common base of all the computable real numbers we may want to use. Usually coefficients of a polynomial are taken in a ring. We cannot do this here. For example, addition of two intervals is not associative. Therefore, the abstract interface for coefficients contains the required operations (addition, multiplication, etc.) only, where some basic properties (associativity, distributivity, etc.) are ruled out. The case of abstract polynomials is similar. They are also a `Module Type` but this time parameterized by the coefficients. The interface contains only the operations on polynomials (addition, evaluation, iterator, etc.) with the properties that are satisfied by all common instantiations of polynomials. For intervals, we directly use the abstract interface provided by the Coq.Interval library [30].

Rigorous Polynomial Approximations. We are now able to give the definition of our rigorous polynomial approximation.

```
Module RigPolyApprox (C : BaseOps)(P : PolyOps C)(I : IntervalOps).
Structure rpa : Type := RPA { approx : P.T;  error : I.type }.
```

The module is parameterized by `C` (the coefficients), by `P` (the polynomials with coefficients in `C`), and by `I` (the intervals).

Generic Taylor Polynomials. Before implementing our Taylor models, we use the abstract coefficients and polynomials to implement generic Taylor polynomials. These polynomials are computed using an algorithm based on recurrence relations as described in Section 2.3. This algorithm can be implemented in a generic way. It takes as argument the relation between successive coefficients, the initial conditions and outputs the Taylor polynomial.

We detail the example of the exponential, which was also presented in Section 2.3. The Taylor coefficients $(c_n)_{n\in\mathbb{N}}$ satisfy $(n+1)c_{n+1} - c_n = 0$. The corresponding Coq code is

```
Definition exp_rec (n : nat) u := tdiv u (tnat n).
```

where `tdiv` is the division on our coefficients and `tnat` is an injection of integers to our type of coefficients. We then implement the generic Taylor polynomial for the exponential around a point x_0 with the following definition.

```
Definition T_exp n x0 := trec1 exp_rec (texp x0) n.
```

In this definition, `trec1` is the function in the polynomial interface that is in charge of producing a polynomial of size `n` from a recurrence relation of order 1 (here, `exp_rec`) and an initial condition (here, `texp x0`, the value of the exponential at `x0`). The interface also contains `trec2` and `trecN` for producing polynomials from recurrences of order 2 and order N with the appropriate number of initial conditions. Having specific functions for recurrences of order 1 and 2 makes it possible to have optimized implementations for these frequent recurrences. All the functions we currently dispose of in our library are in fact defined with `trec1` and `trec2`. We provide generic Taylor polynomials for constant functions, identity, $x \mapsto \frac{1}{x}$, $\sqrt{\cdot}$, $\frac{1}{\sqrt{\cdot}}$, exp, ln, sin, cos, arcsin, arccos, arctan.

Taylor Models. We implement TMs on top of the RPA structure by using polynomials with coefficients that are intervals with FP bounds, according to Section 2. Yet we are still generic with respect to the effective implementation of polynomials. For the remainder, we also use *intervals with FP bounds.* This datatype is provided by the Coq.Interval library [30], whose design is also based on modules, in such a way that it is possible to plug all the machinery on the desired kind of Coq integers (i.e., `Z` or `BigZ`).

In a TM for a basic function (e.g., exp), polynomials are instances of the generic Taylor polynomials implemented with the help of the recurrence relations. The remainder is computed with the help of the Taylor-Lagrange formula in Lemma 1. For this computation, thanks to the parameterized module, we reuse the generic recurrence relations. The order-n Taylor model for the exponential on interval `X` expanded at the small interval `X0` is as follows:

```
Definition TM_exp (n : nat) X X0 :=
  RPA (T_exp n X0) (Trem T_exp n X X0).
```

We implement Taylor models for the addition, multiplication, and composition of two functions by arithmetic manipulations on the Taylor models of the two functions, as described in Section 2. Here is the example of addition:

```
Definition TM_add (Mf Mg : rpa) :=
  RPA (P.tadd (approx Mf) (approx Mg))
      (I.add (error Mf) (error Mg)).
```

The polynomial approximation is just the sum of the two approximations and the interval error is the sum of the two errors. Multiplication is almost as intuitive. We consider the truncated multiplication of the two polynomials and we make sure that the error interval takes into account the remaining parts of the truncated multiplication. Composition is more complex. It uses addition and multiplication of Taylor polynomials. Division of Taylor models is implemented in term of multiplication and composition with the inverse function $x \mapsto 1/x$. The corresponding algorithms are fully described in [23].

Discussion on the Formal Verification of Taylor Models. The Taylor model `Module` also contains a version of Taylor polynomials defined with axiomatic real number coefficients. These polynomials are meant to be used only in the formal verification when linking the computable Taylor models to the corresponding functions on axiomatic real numbers. This link is given by Definition 1. The definition can be easily formalized in the form of a predicate `validTM`.

```
Definition validTM X X0 M f :=
  I.subset X0 X /\
  contains (error M) 0 /\
  let N := tsize (approx M) in
  forall x0, contains X0 x0 -> exists P, tsize P = N /\
    ( forall k, (k < N) ->
      contains (tnth (approx M) k) (tnth P k) ) /\
    forall x, contains X x ->
      contains (error M) (f x - teval P (x - x0)).
```

The theorem of correctness for the Taylor model of the exponential `TM_exp` then establishes the link between the model and the exponential function `Rexp` that is defined in the real library.

```
Lemma TM_exp_correct :
  forall X X0 n, validTM X X0 (TM_exp n X X0) Rexp.
```

Our goal is to formally prove the correctness of our implementation of Taylor models. We want proofs that are generic, so a new instantiation of the polynomials would not require changing the proofs. In a previous version of our Coq development we had managed to prove correct Taylor models for some elementary functions and addition. No proofs are available yet for the version presented here but adapting the proofs to this new setting should be possible.

4 Benchmarks

We want to evaluate the performances of our Coq implementation of Taylor models. For this we compare them to those of Sollya [13], a tool specially designed to handle such numerical approximation problems.

The Coq Taylor models we use for our tests are implemented with polynomials represented as simple lists with a linear access to their coefficients. The coefficients of the approximating polynomial in our instantiation of Taylor models as well as the interval errors are implemented by intervals with multiple-precision FP bounds as available in the Coq.Interval library described in [30]. Since we need to evaluate the initial conditions for recurrences, only the basic functions already implemented in Coq.Interval can have their corresponding Taylor models.

In Sollya, polynomials have interval coefficients and are represented by a (coefficient) array of intervals with multiple-precision FP bounds. Sollya's `autodiff()` function computes interval enclosures of the successive derivatives of a function at a point or over an interval, relying on interval arithmetic computations and recurrence relations similar to the ones we use in our Coq development. Thus, we use it to compute the Taylor models we are interested in.

Timings, Accuracy and Comparisons

We compare the Coq and the Sollya implementations presented above on a selection of several benchmarks. Table 1 gives the timings as well as the tightness obtained for the remainders. These benchmarks have been computed on a 4-core computer, Intel(R) Xeon(R) CPU X5482 @ 3.20GHz.

Each cell of the first column of Table 1 contains a target function, the precision in bits used for the computations, the order of the TM, and the interval under consideration. When "split" is mentioned, the interval has been subdivided into a specified amount of intervals of equal length (1024 subintervals for instance in line 3) and a TM has been computed over each subinterval. Each TM is expanded at the middle of the interval. The symbols $\mathrm{RD}_t(\ln 4)$, resp. $\mathrm{RU}_t(\ln 2)$, denote $\ln(4)$ rounded toward $-\infty$, resp. $\ln(2)$ rounded toward $+\infty$, using precision t.

Columns 2 and 3 give the total duration of the computations (for instance, the total time for computing the 1024 TMs of the third line) in Coq and Sollya respectively. Columns 4 and 5 present an approximation error obtained using Coq and Sollya, while the last column gives, as a reference, the true approximation error, computed by ad-hoc means (symbolically for instance), of the function by its Taylor polynomial. Note that when "split" is mentioned, the error presented corresponds to the one computed over the last subinterval (for instance, $[2 - 1/256, 2]$ for the arctan example). For simplicity, the errors are given using three significant digits.

In terms of accuracy, the Coq and Sollya results are close. We have done other similar checks and obtained the same encouraging results (the error bounds returned by Coq and Sollya have the same orders of magnitude). This does not prove anything but is nevertheless very reassuring. Proving the correctness of an implementation that produces too large bounds would be meaningless.

Coq is 6 to 10 times slower than Sollya, which is reasonable. This factor gets larger when composition is used. One possible explanation is that composition implies lots of polynomial manipulations and the implementation of polynomials as simple lists in Coq maybe too naive. An interesting alternative could be to use persistent arrays [2] to have more efficient polynomials. Another

Table 1. Benchmarks and timings for our implementation in Coq

	Execution time		Approximation error		
	Coq	Sollya	Coq	Sollya	Mathematical
exp prec=120, deg=20 I=$[1, \mathrm{RD}_{53}(\ln 4)]$ no split	7.40s	0.01s	7.90×10^{-35}	7.90×10^{-35}	6.57×10^{-35}
exp prec=120, deg=8 I=$[1, \mathrm{RD}_{53}(\ln 4)]$ split in 1024	20.41s	3.77s	3.34×10^{-39}	3.34×10^{-39}	3.34×10^{-39}
exp prec=600, deg=40 I=$[\mathrm{RU}_{113}(\ln 2), 1]$ split in 256	38.10s	16.39s	6.23×10^{-182}	6.22×10^{-182}	6.22×10^{-182}
arctan prec=120, deg=8 I=$[1, 2]$ split in 256	11.45s	1.03s	7.43×10^{-29}	2.93×10^{-29}	2.85×10^{-29}
exp × sin prec=200, deg=10 I=$[1/2, 1]$ split in 2048	1m22s	12.05s	6.92×10^{-50}	6.10×10^{-50}	5.89×10^{-50}
exp/sin prec=200, deg=10 I=$[1/2, 1]$ split in 2048	3m41s	13.29s	4.01×10^{-43}	9.33×10^{-44}	8.97×10^{-44}
exp ∘ sin prec=200, deg=10 I=$[1/2, 1]$ split in 2048	3m24s	12.19s	4.90×10^{-47}	4.92×10^{-47}	4.90×10^{-47}

possible improvement is at algorithmic level: while faster algorithms for polynomial multiplication exist [17], currently in all TMs related works $O(n^2)$ naive multiplication is used. We could improve that by using a Karatsuba-based approach, for instance.

5 Conclusion and Future Works

We have described an implementation of Taylor models in the Coq proof assistant. Two main issues have been addressed. The first one is genericity. We wanted our implementation to be applicable to a large class of problems. This motivates our use of modules in order to get this flexibility. The second issue is efficiency. Working in a formal setting has some impact in terms of efficiency. Before starting to prove anything, it was then crucial to evaluate if the computational power provided by Coq was sufficient for our needs. The results given in Section 4 clearly indicate that what we have is worth proving formally.

We are in the process of proving the correctness of our implementation. Our main goal is to prove the validity theorem given in Section 2 formally. This is tedious work but we believe it should be completed in a couple of months. As we aim at a complete formalization, a more subtle issue concerns the Taylor models for the basic functions and in particular how the model and its corresponding function can be formally related. This can be done in an ad-hoc way, deriving the recurrence relation from the formal definition. An interesting future work would be to investigate a more generic approach, trying to mimic what is provided by the Dynamic Dictionary of Mathematical Functions [3] in a formal setting.

Having Taylor models is an initial step in our overall goal of getting formally proved worst-case accuracy for common functions and formats. A natural next step is to couple our models with some positivity test for polynomials, for example some sums-of-squares technique. This would give us an automatic way of verifying polynomial approximations formally. It would also provide another way of evaluating the quality of our Taylor approximations. If they turned out to be not accurate enough for our needs, we could always switch to better kinds of approximations such as Chebyshev truncated series, thanks to our generic setting.

References

1. Abramowitz, M., Stegun, I.A.: Handbook of mathematical functions with formulas, graphs, and mathematical tables. National Bureau of Standards Applied Mathematics Series, vol. 55. For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C (1964)
2. Armand, M., Grégoire, B., Spiwack, A., Théry, L.: Extending Coq with Imperative Features and Its Application to SAT Verification. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 83–98. Springer, Heidelberg (2010)
3. Benoit, A., Chyzak, F., Darrasse, A., Gerhold, S., Mezzarobba, M., Salvy, B.: The Dynamic Dictionary of Mathematical Functions (DDMF). In: Fukuda, K., van der Hoeven, J., Joswig, M., Takayama, N. (eds.) ICMS 2010. LNCS, vol. 6327, pp. 35–41. Springer, Heidelberg (2010)
4. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer, Heidelberg (2004)
5. Berz, M., Makino, K.: Rigorous global search using Taylor models. In: SNC 2009: Proceedings of the 2009 Conference on Symbolic Numeric Computation, pp. 11–20. ACM, New York (2009)
6. Berz, M., Makino, K., Kim, Y.K.: Long-term stability of the tevatron by verified global optimization. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 558(1), 1–10 (2006); Proceedings of the 8th International Computational Accelerator Physics Conference - ICAP 2004
7. Boespflug, M., Dénès, M., Grégoire, B.: Full Reduction at Full Throttle. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 362–377. Springer, Heidelberg (2011)
8. Boldo, S., Melquiond, G.: Flocq: A Unified Library for Proving Floating-point Algorithms in Coq. In: Proceedings of the 20th IEEE Symposium on Computer Arithmetic, Tübingen, Germany, pp. 243–252 (2011)

9. Brisebarre, N., Chevillard, S.: Efficient polynomial L^∞-approximations. In: Kornerup, P., Muller, J.M. (eds.) 18th IEEE Symposium on Computer Arithmetic, pp. 169–176. IEEE Computer Society, Los Alamitos (2007)
10. Brisebarre, N., Muller, J.M., Tisserand, A.: Computing Machine-efficient Polynomial Approximations. ACM Trans. Math. Software 32(2), 236–256 (2006)
11. Cháves, F.: Utilisation et certification de l'arithmétique d'intervalles dans un assistant de preuves. Thèse, École normale supérieure de Lyon - ENS LYON (September 2007), http://tel.archives-ouvertes.fr/tel-00177109/en/
12. Chevillard, S.: Évaluation efficace de fonctions numériques. Outils et exemples. Ph.D. thesis, École Normale Supérieure de Lyon, Lyon, France (2009), http://tel.archives-ouvertes.fr/tel-00460776/fr/
13. Chevillard, S., Joldeş, M., Lauter, C.: Sollya: An Environment for the Development of Numerical Codes. In: Fukuda, K., van der Hoeven, J., Joswig, M., Takayama, N. (eds.) ICMS 2010. LNCS, vol. 6327, pp. 28–31. Springer, Heidelberg (2010)
14. Chevillard, S., Harrison, J., Joldeş, M., Lauter, C.: Efficient and accurate computation of upper bounds of approximation errors. Theoretical Computer Science 16(412), 1523–1543 (2011)
15. Collins, P., Niqui, M., Revol, N.: A Taylor Function Calculus for Hybrid System Analysis: Validation in Coq. In: NSV-3: Third International Workshop on Numerical Software Verification (2010)
16. de Dinechin, F., Lauter, C., Melquiond, G.: Assisted verification of elementary functions using Gappa. In: Proceedings of the 2006 ACM Symposium on Applied Computing, Dijon, France, pp. 1318–1322 (2006), http://www.lri.fr/~melquion/doc/06-mcms-article.pdf
17. von zur Gathen, J., Gerhard, J.: Modern computer algebra, 2nd edn. Cambridge University Press, New York (2003)
18. Geuvers, H., Niqui, M.: Constructive Reals in Coq: Axioms and Categoricity. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) TYPES 2000. LNCS, vol. 2277, pp. 79–95. Springer, Heidelberg (2002)
19. Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA (2008)
20. Grégoire, B., Théry, L.: A Purely Functional Library for Modular Arithmetic and Its Application to Certifying Large Prime Numbers. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 423–437. Springer, Heidelberg (2006)
21. Griewank, A.: Evaluating Derivatives - Principles and Techniques of Algorithmic Differentiation. SIAM (2000)
22. IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754™-2008 (August 2008)
23. Joldeş, M.: Rigourous Polynomial Approximations and Applications. Ph.D. dissertation, École Normale Supérieure de Lyon, Lyon, France (2011), http://perso.ens-lyon.fr/mioara.joldes/these/theseJoldes.pdf
24. Krebbers, R., Spitters, B.: Computer Certified Efficient Exact Reals in Coq. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) Calculemus/MKM 2011. LNCS, vol. 6824, pp. 90–106. Springer, Heidelberg (2011)

25. Lefèvre, V., Muller, J.M.: Worst cases for correct rounding of the elementary functions in double precision. In: Burgess, N., Ciminiera, L. (eds.) Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16), Vail, CO (June 2001)
26. Lizia, P.D.: Robust Space Trajectory and Space System Design using Differential Algebra. Ph.D. thesis, Politecnico di Milano, Milano, Italy (2008)
27. Makino, K.: Rigorous Analysis of Nonlinear Motion in Particle Accelerators. Ph.D. thesis, Michigan State University, East Lansing, Michigan, USA (1998)
28. Makino, K., Berz, M.: Taylor models and other validated functional inclusion methods. International Journal of Pure and Applied Mathematics 4(4), 379–456 (2003), http://bt.pa.msu.edu/pub/papers/TMIJPAM03/TMIJPAM03.pdf
29. Mayero, M.: Formalisation et automatisation de preuves en analyses réelle et numérique. Ph.D. thesis, Université Paris VI (2001)
30. Melquiond, G.: Proving Bounds on Real-Valued Functions with Computations. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 2–17. Springer, Heidelberg (2008)
31. Moore, R.E.: Methods and Applications of Interval Analysis. Society for Industrial and Applied Mathematics (1979)
32. Muller, J.M.: Projet ANR TaMaDi – Dilemme du fabricant de tables – Table Maker's Dilemma (ref. ANR 2010 BLAN 0203 01), http://tamadiwiki.ens-lyon.fr/tamadiwiki/
33. Muller, J.M.: Elementary Functions, Algorithms and Implementation, 2nd edn. Birkhäuser, Boston (2006)
34. Neher, M., Jackson, K.R., Nedialkov, N.S.: On Taylor model based integration of ODEs. SIAM J. Numer. Anal. 45, 236–262 (2007)
35. Neumaier, A.: Taylor forms – use and limits. Reliable Computing 9(1), 43–79 (2003)
36. O'Connor, R.: Certified Exact Transcendental Real Number Computation in Coq. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 246–261. Springer, Heidelberg (2008)
37. The Arénaire Project: CRlibm, Correctly Rounded mathematical library (July 2006), http://lipforge.ens-lyon.fr/www/crlibm/
38. Remez, E.: Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. C.R. Académie des Sciences 198, 2063–2065 (1934) (in French)
39. Salvy, B., Zimmermann, P.: Gfun: a Maple package for the manipulation of generating and holonomic functions in one variable. ACM Trans. Math. Software 20(2), 163–177 (1994)
40. Stanley, R.P.: Differentiably finite power series. European Journal of Combinatorics 1(2), 175–188 (1980)
41. Ziv, A.: Fast evaluation of elementary mathematical functions with correctly rounded last bit. ACM Trans. Math. Software 17(3), 410–423 (1991)
42. Zumkeller, R.: Formal Global Optimisation with Taylor Models. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 408–422. Springer, Heidelberg (2006)

Formal Proof of SCHUR Conjugate Function* **

Franck Butelle[1], Florent Hivert[2], Micaela Mayero[1,3], and Frédéric Toumazet[4]

[1] LIPN UMR 7030, Université Paris 13, Villetaneuse, F-93430
[2] LITIS EA 4108, Université de Rouen, Saint-Etienne-du-Rouvray, F-76801
[3] LIP, INRIA Grenoble – Rhône-Alpes, UMR 5668, UCBL, ENS Lyon, Lyon, F-69364
[4] LIGM UMR 8049, Université de Marne-la-Vallée, Champs sur Marne, F-77454

Abstract. The main goal of our work is to formally prove the correctness of the key commands of the SCHUR software, an interactive program for calculating with characters of Lie groups and symmetric functions. The core of the computations relies on enumeration and manipulation of combinatorial structures. As a first "proof of concept", we present a formal proof of the conjugate function, written in C. This function computes the conjugate of an integer partition. To formally prove this program, we use the Frama-C software. It allows us to annotate C functions and to generate proof obligations, which are proved using several automated theorem provers. In this paper, we also draw on methodology, discussing on how to formally prove this kind of program.

1 Introduction

SCHUR [1] is an interactive software for calculating properties of Lie groups and symmetric functions [2]. It is used in research by combinatorists, physicists, theoretical chemists [3] as well as for educational purpose as a learning tool for students in algebraic combinatorics. One of its main uses is to state conjectures on combinatorial objects. For such use, it is important to have some confidence in the results produced by SCHUR.

Until now, the method used to get some confidence in the results has mostly been based on just one example for each command.

The computation of other examples is complex due to the well known combinatorial explosion, especially when using algorithms associated to the symmetric group, see section 2.1. Unfortunately, the combinatorial explosion as well as computing time forbid test generation or verification techniques (model checking). Therefore, in this paper, we focus on formal proof of the existing program.

With the aim of verifying the whole software, we start with proving the correctness of its fundamentals bricks. The main combinatorial object used in SCHUR is integer partition. The first non-trivial operation on integer partitions is the conjugate. Moreover, conjugate function is necessary for more than half of the 240 interactive commands of SCHUR. From this point of view, we can say that conjugate is a critical function of SCHUR.

* This research is supported by the PEPS-CNRS project CerBISS.
** The final publication of this paper is available at www.springerlink.com

The very first work consists in isolating (see 4.3) the code of this function from the program. Next, we chose to use the most popular tool of program proof community's Frama-C [4] successor of Caduceus. Frama-C is a plug-in system. In order to prove programs, we used Jessie [5], the deductive verification plug-in of C programs annotated with ACSL [6]. The generated verification conditions can be submitted to external automatic provers, and for more complex situations, to interactive theorem provers as well (see section 2.2).

After a short presentation of software tools and theoretical concepts, we will present the formal proof of a program. Finally, after discussing difficulties and mistakes encountered along the way, we will propose a methodology to prove such a software, and finally discuss future work.

2 Presentation of the Software Used

2.1 The SCHUR Software

SCHUR is an interactive software for calculating properties of Lie groups and symmetric functions. A Symmetric Function is a function which is symmetric, or invariant, under any permutation of its variables. For example $f(x_1, x_2, x_3) = x_1 + x_2 + x_3$ is a symmetric function.

SCHUR has originally written by Prof. Brian G. Wybourne in Pascal language. Then it was translated into C by an automatic program making it quite difficult to read. There are almost no comments in the code, the code is more than 50,000 lines long with many global variables. Local variables have names such as $W52$ and so on.

After the death of Prof. Wybourne in November 2003, some people felt that his program should be maintained, and if possible enhanced, with a view to making it freely available to the mathematics and physics research community.

Nowadays, it is open source under the GPL license and includes more than 240 commands. The code still includes very few comments. Some mistakes have been corrected but some interactive commands are so intricate that it is difficult to have more than a few examples to check them against and most people do not even know if the result is correct or not.

This is why we started to work on this code. Firstly some of the commands in SCHUR are very well implemented (for example, plethysm is computed faster by SCHUR than by many other combinatorial toolboxes). Formally proving some key functions inside would also be a major advance for its research community.

2.2 The Frama-C Software

Frama-C [4] is an open source extensible platform dedicated to source code analysis of C software. It is co-developed by two French public institutions: CEA–LIST (Software Reliability Laboratory) and INRIA-Saclay (ProVal project).

Frama-C is a plug-in system. In order to prove programs, we use Jessie [5], the deductive verification plug-in of C programs annotated with ACSL [6]. It

uses internally the languages and tools of the Why platform [7]. The Jessie plug-in uses Hoare-style [8] weakest precondition computations to formally prove ACSL properties. The generated verification conditions (VC) can be submitted to external automatic provers such as Simplify [9], Alt-Ergo [10], Z3 [11], CVC3 [12].

These automatic provers belong to SMT (Satisfiability Modulo Theories) solvers. The SMT problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. First-order logic is undecidable. Due to this high computational difficulty, it is not possible to build a procedure that can solve arbitrary SMT problems. Therefore, most procedures focus on the more realistic goal of efficiently solving problems that occur in practice.

For more complex situations, interactive theorem provers can be used to establish the validity of VCs, like Coq [13], PVS [14], Isabelle/HOL [15], etc. For our purpose, we used Coq (see section 4.2) since it is the one best known to the authors.

3 The Conjugate Function

In this section, the basics of algebraic combinatorics are given so that the reader can understand what is actually proved. Interestingly in this field, though the interpretation of what is actually computed can be of a very abstract algebraic level, the computation itself boils down most of the time to possibly intricate but rather elementary manipulations.

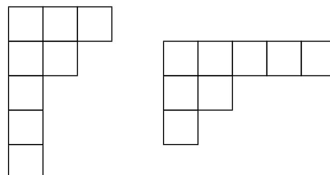
3.1 Combinatorial and Algebraic Background: Integer Partitions

A **partition** of a positive integer n is a way of writing n as a sum of a non-increasing sequence of integers. For example $\lambda = (4, 2, 2, 1)$ and $\mu = (2, 1)$ are partitions of $n = 9$ and $n' = 3$ respectively. We write $|\lambda| = n$ and $|\mu| = n'$ [16].

The **Ferrers diagram** F^λ associated to a partition $\lambda = (\lambda_1, \lambda_2, ..., \lambda_p)$ consists of $|\lambda| = n$ boxes, arranged in $l(\lambda) = p$ left-justified rows of lengths λ_1, λ_2, ..., λ_p. Rows in F^λ are oriented downwards (or upwards for some authors). F^λ is called the shape of λ.

Definition 1. *The conjugate of an integer partition is the partition associated to the diagonal symmetric of its shape.*

For example, for $\lambda = (3, 2, 1, 1, 1)$, here is the Ferrers diagram F^λ and the Ferrers diagram of the conjugate partition:



So the conjugate partition of $(3, 2, 1, 1, 1)$ is $(5, 2, 1)$.

A **semi-standard Young tableau** of shape λ is a numbering of the boxes of F^λ with entries from $\{1, 2, ..., n\}$, weakly increasing across rows and strictly increasing down columns. A tableau is **standard** if and only if each entry appears only once. Here is an example of shape $(4, 2, 2, 1)$ tableau:

1	2	2	5
2	4		
3	6		
5			

A **symmetric function** of a set of variables $\{x_1, x_2, \ldots\}$ is a function $f(x_1, x_2, \ldots)$ of those variables which is invariant under any permutation of those variables (that is for example $f(x_1, x_2, \ldots) = f(x_2, x_1, \ldots)$). This definition is usually restricted to polynomial functions. The most important linear basis of symmetric function's algebra is called the **Schur functions** and they are combinatorially defined as follows: for a given semi-standard Young tableau T of shape λ, write $\mathbf{x}^T$ the product of the x_i for all i appearing in the tableau. Then

$$s_\lambda(\mathbf{x}) = \sum_{T \in \mathrm{Tab}(\lambda)} \mathbf{x}^T. \tag{1}$$

where $\mathrm{Tab}(\lambda)$ is the set of all tableaux of shape λ. We will note $s_\lambda(\mathbf{x})$, s_λ. For example, consider the tableaux of shape $(2, 1)$ using just three variables x_1, x_2, x_3:

$$\begin{array}{|c|c|}\hline 1&1\\\hline 2\\\cline{1-1}\end{array}\quad \begin{array}{|c|c|}\hline 1&1\\\hline 3\\\cline{1-1}\end{array}\quad \begin{array}{|c|c|}\hline 2&2\\\hline 3\\\cline{1-1}\end{array}\quad \begin{array}{|c|c|}\hline 1&2\\\hline 3\\\cline{1-1}\end{array}\quad \begin{array}{|c|c|}\hline 1&3\\\hline 2\\\cline{1-1}\end{array}\quad \begin{array}{|c|c|}\hline 1&2\\\hline 2\\\cline{1-1}\end{array}\quad \begin{array}{|c|c|}\hline 1&3\\\hline 3\\\cline{1-1}\end{array}\quad \begin{array}{|c|c|}\hline 2&3\\\hline 3\\\cline{1-1}\end{array}$$

The associated Schur function is therefore:

$$s_{(21)}(x_1, x_2, x_3) = x_1^2 x_2 + x_1^2 x_3 + x_2^2 x_3 + 2x_1 x_2 x_3 + x_1 x_2^2 + x_1 x_3^2 + x_2 x_3^2 \tag{2}$$

thus:

$$s_{(21)} = s_{(21)}(x_1, x_2, x_3) + s_{(21)}(x_1, x_2, x_3, x_4) + \ldots$$

Note that, with this combinatorial definition, the symmetry of $s_{(21)}(x_1, x_2, x_3)$ is not exactly obvious.

We need to recall some well-known results in symmetric function theory: though Schur functions have historically been defined by Jacobi [17], they were named in the honor of Schur who discovered their crucial role in the representation theory of the symmetric group and the general linear group. Namely, after the discovery by Frobenius that the irreducible representation of the symmetric groups are indexed by integer partitions, Schur showed that those functions can be interpreted as characters of those irreducible representation, and by Schur-Weyl duality characters of Lie groups and Lie algebras. Notably we obtain the representation of the general linear groups (GL_n) and unitary groups (U_n) [18] from the symmetric group representations. In this setting, the conjugate of the

partition essentially encodes the tensor product of a representation by the sign representation.

Further work by Schur-Littlewood involve infinite sum of Schur functions associated to partitions [19], whose conjugates have a particular form. In particular, these series are used to obtain symplectic (Sp_{2n}) and orthogonal character groups (O_n) (symmetric and orthogonal Schur functions) from standard Schur functions [20].

One particularly important and difficult computational problem here is plethysm (see SCHUR reference manual [1] and [2]). It is the analogue in symmetric functions of the substitution of polynomial inside another polynomial $f(x) \mapsto f(g(x))$. It is called plethysm because by some combinatorial explosion, it involves very quickly a lot (a plethora) of terms, making it something very difficult to compute efficiently. For example, $s_{(21)}(s_{(21)})$, the first example with non trivial partitions in the input is already very hard to compute by hand. First we can regard $s_{(21)}$ as a function in as many monomial as in (2):

$$s_{(21)}(s_{(21)})(x_1,x_2,x_3) = s_{(21)}(x_1^2x_2, x_1^2x_3, x_2^2x_3, x_1x_2x_3, x_1x_2x_3, x_1x_2^2, x_1x_3^2, x_2x_3^2)$$

it can be shown that the following holds:

$$\begin{aligned} s_{(21)}(s_{(21)}) = {} & s_{(22221)} + s_{(321111)} + 2s_{(32211)} + s_{(3222)} + s_{(33111)} + \\ & 3s_{(3321)} + 2s_{(42111)} + 3s_{(4221)} + 3s_{(4311)} + 3s_{(432)} + \\ & s_{(441)} + s_{(51111)} + 2s_{(5211)} + s_{(522)} + 2s_{(531)} + s_{(54)} + s_{(621)} \end{aligned}$$

3.2 Computation in Algebraic Combinatorics

Basically, the architecture of a software for computing in algebraic combinatorics is composed of two parts:

- a computer algebra kernel dealing with the bookkeeping of expressions and linear combinations (parsing, printing, collecting, Gaussian and Groebner elimination algorithm...);
- a very large bunch of small combinatorial functions which enumerate and manipulate the combinatorial data structures.

In algebraic combinatorics software, for each basic combinatorial structure such as permutations or partitions, there are typically 50-200 different functions. Conjugating a partition is a very good example of what those many functions do, that is surgery on lists of integers or lists of lists of integers or more advanced recursive structures like trees... In a basic computation, most of the time is spent mapping or iterating those functions on some sets of objects. But due to combinatorial explosion those sets can be very large so these functions must be very well optimized.

3.3 Properties

The definition of conjugate (diagonal symmetric of its partition shape) is easy to understand but may conduct to naive implementations that may be inefficient.

Let us suppose that we represent an integer partition by an integer array starting from 1. For example $\lambda = (3, 2, 1, 1, 1)$ gives $t[1] = 3$, $t[2] = 2$,... $t[l(\lambda)] = 1$. Recall that $t[i]$ is non-increasing, that is $t[i+1] \leq t[i]$.

One way to compute the conjugate is to count boxes: in our previous example the first column of λ had 4 boxes, the second had 3 etc. Therefore, to compute the number of boxes in a column j we need to know how many lines are longer than j. As a consequence, if t_c is the array representing the conjugate, the following formula gives the value of the entries of the conjugate:

$$t_c[j] = |\{i \mid 1 \leq i \leq l(\lambda) \wedge t[i] \geq j\}| \ .$$

Note that $t_c[j] = 0$ if $j > t[1]$, so the previous expression must be computed only from $j = 1$ to $j = t[1]$. This last property will be one of our predicates used to check the correctness of loop invariants.

3.4 SCHUR Implementation

Here follows the code of the conjugate function extracted from the SCHUR software. We expanded type definitions (C "structs" and "typedefs") from the original code just to simplify the work of Frama-C and to make this part of code independent from the rest of the SCHUR software (getting rid of global variables and so on).

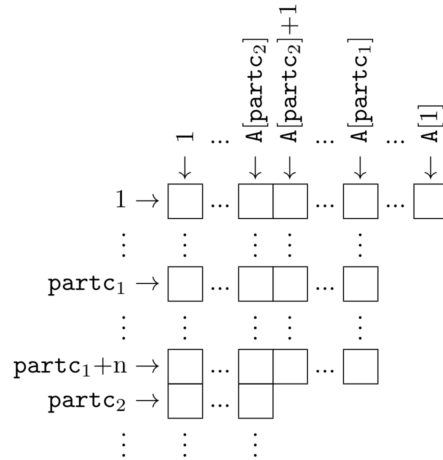
```
#define MAX 100

void conjgte (int A[MAX], int B[MAX]) {
  int i, partc = 1, edge = 0;

  while (A[partc] != 0) {
     edge = A[partc];
     do
          partc = partc + 1;
     while (A[partc] == edge);
     for (i = A[partc] + 1; i <= edge; i++)
          B[i] = partc - 1;
  }
}
```

Note that this implementation is not naive (and not so easy to understand) but its time complexity is optimal (linear in the length of the partition).

The algorithm is based on looking for the set of descents of the partition[5]. The do–while loop follows a "flat" portion of the partition ($t[i] = t[i-1]$) until a descent is found. Next the for–loop assigns the values of the B array according to the flat portion. The following figure clarifies this: we have denoted $\texttt{partc}_1$ the value of `partc` at the entrance of while loop. $\texttt{partc}_2$ is the value of `partc` after

[5] A descent is such that $t[i] < t[i-1]$

the do–while loop. For clarity's sake we supposed A[$\texttt{partc}_2$]+1 to be different from A[$\texttt{partc}_1$]. If we count boxes column by column to construct array B, it is clear that B[i]=$\texttt{partc}_2$-1 for all A[$\texttt{partc}_2$]+1 $\leq i \leq$ A[$\texttt{partc}_1$]=edge.



4 The Formal Proof of the Conjugate Function

4.1 Annotations

In the following paragraphs we present the annotations added to the code. Note that this is the only additions made to it. First we have to specify the model of integers we want to deal with:

```
#pragma JessieIntegerModel(strict)
```

This means that int types are modeled by integers with appropriate bounds, and for each arithmetic operation, it is mandatory to show that no overflow occurs.

Next, we have to express in first-order logic what an integer partition (stored in an array) is:

```
#define MAX 100
/*@ predicate is_partition{L}(int t[]) =
    (\forall integer i; 1 <= i < MAX ==> 0 <= t[i] < (MAX-1)) &&
    (\forall integer i,j; 1 <= i <=j < MAX ==> t[j] <= t[i]) &&
    t[MAX-1]==0;
  */
```

Note that annotations are coded in the C comments, starting with a @. The {L} term is the context (pre, post, etc.), we won't detail it here, see [5, 6] for details.

The data structure (array of integers) comes from the way the SCHUR software represents integer partitions. 0 is used as a mark of end of array, just like character strings in C. The MAX value comes from the original source code as well. The first line of the predicate `is_partition` expresses that we are able to compute the conjugate (if at least one element is greater than or equal to MAX-1, the conjugate will no be able to be stored in an array of size MAX-1 with the last element fixed to 0). From the source code it is expressed by an external simple test on `t[1]`, but expressing it like that simplifies automatic provers job. The second line of the predicate defines the non-increasing order.

The following predicate is needed to express how we count blocs to compute the conjugate. It may be read as z equals the number of elements of partition t, whose indexes are included in $\{1, .., j-1\}$ and whose values are greater than or equal to k. It is theoretically possible to express it as an axiomatic theory, a kind of function, but automatic provers we use make a better use of predicates. Note that we need to explicit the $z = 0$ case, in order to be able to prove the global post-condition `is_conjugate(A,B)`.

```
/*@ predicate countIfSup{L}(int t[],integer j,integer k,integer z)=
    is_partition{L}(t) &&
    1<= j <= MAX &&
    1<= k < MAX &&
    ((1<=z<j && \forall integer i ;  1<=i<=z ==> t[i]>= k)
     || (z==0 && \forall integer i ; 1<=i<j ==> t[i]<k)) ;
 */
```

Here is what we want to obtain at the end of the computation, t2 is a conjugate of t1 if the following holds:

```
/*@ predicate is_conjugate{L}(int t1[], int t2[]) =
    \forall integer k ; 1<=k<MAX ==> countIfSup(t1,MAX,k,t2[k]);
 */
```

Finally, here is the function. First we have to give precise requirements on the inputs. For example, (`\valid(A+ (1..(MAX-1)))` means that memory has been allocated so array indexes from 1 to MAX-1 are allowed). From the original code, the B array is supposed to be initialized with zeros before calling the function. This is translated into a `requires` directive. Next, we specify which memory elements are modified by the function (`assigns`). This is used for safety proofs. In the end, the output is correct if the post-condition (`ensures`) is met.

```
/*@ requires \valid(A+ (1..(MAX-1)));
    requires \valid(B+ (1..(MAX-1)));
    requires is_partition(A);
    requires \forall integer k; 1<=k<MAX ==> B[k]== 0;
    assigns B[1..A[1]];
    ensures is_conjugate(A,B);
    */
```

```
void conjgte (int A[MAX], int B[MAX])
{
  int i, partc=1, edge = 0 ;
```

Now we have to define the loop variant and invariant for each loop (to prove properties). The "loop variant" must decrease, while remaining non negative, to be able to prove termination. We also use a "`ghost` variable" to store the state of a variable before any modification.

```
  /*@ loop variant MAX-partc;
      loop invariant 1<=partc<MAX;
      loop assigns B[1..A[1]];
      loop invariant \forall integer k;
         A[partc]+1 <=k <= A[1] ==> countIfSup(A,MAX,k,B[k]);
    */
  while (A[partc] != 0) {
     edge = A[partc];

     /*@ ghost int old_partc = partc; */

    /*@ loop variant MAX-partc;
        loop invariant old_partc<=partc ;
        loop invariant \forall integer k;
            old_partc<= k <= partc ==> A[k]==edge;
        loop invariant partc<MAX-1;
     */
     do
         partc = partc + 1;
     while (A[partc] == edge);
```

We also use the `assert` directive to have a verification point of a property that may help automatic provers for the next properties or global ones.

```
     /*@ assert countIfSup(A,partc,edge,partc-1);*/

     /*@ loop variant edge-i;
         loop invariant i >= A[partc]+1 && edge+1>=i ;
         loop invariant \forall integer k;
             A[partc]+1 <=k <i ==> countIfSup(A,MAX,k,B[k]);
         loop assigns B[ (A[partc]+1)..edge];
       */
     for (i = A[partc] + 1; i <= edge; i++)
         B[i] = partc - 1;
  }
}
```

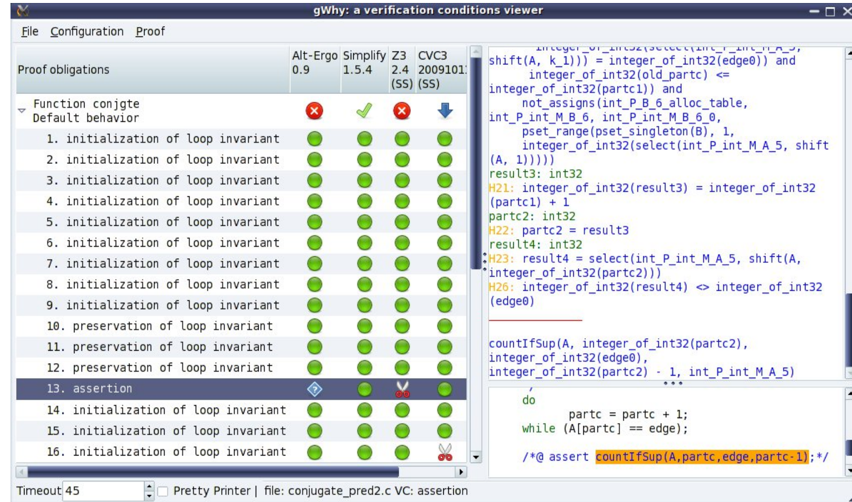


Fig. 1. Graphical Interface: default behavior

4.2 Proofs

The figures 1 to 3 are snapshots of gWhy (Frama-c graphical interface when using plugin Jessie). We applied this tool on the previous annotated code.

The Verification Conditions (VC, also called proof obligations) that have to be proved one by one (line by line) appear to the left of each of the following snapshots. In the upper right part of the window, we can check at a glance what hypotheses are known and what is to be proved at the bottom of it (under the line). No circularity paradox is possible here, since the proof of a VC can only rely on other VC higher in the control-flow graph of the function.

In the lower right part of the window, the corresponding part of the annotation is highlighted in the source code with some lines before and after it.

We will now focus on the VC part, to the left. We can see (green) dots meaning that this property has been proved by this prover. There is also (blue) rhombus with a question mark inside (see assertion 13), indicating that this prover will not be able to to prove this property. Actually, this does not mean that this VC is wrong, remember that these provers use heuristics. Sometimes, you may see scissors meaning that the maximum execution time has been reached without proving the VC. Again, this does not mean that the corresponding VC is wrong. Finally, at the top of a column a (green) check or (red, with a white cross inside) point is shown. The first one means that all properties have been proved by that prover. In fig.1, The (blue) arrow at the top of the CVC3 column means that it is still computing some unshown VC (greater than number 16).

The last figure is the final part. The provers have worked on the safety of the code, that is to say, integer bounds (overflow problems), pointer referencing and termination.

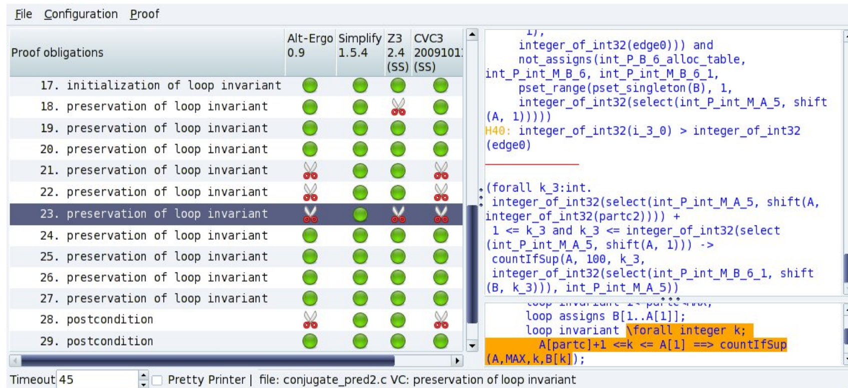


Fig. 2. Graphical Interface continued

As seen in section 4.1, the `B` array has to be initialized with zeros before calling the function. This requirement has been enlightened thanks to the annotations and tools, in particular because without the line `requires \forall integer k;1<=k<MAX ==> B[k]==0`, the postcondition which states that `B` is a conjugate of `A` cannot be proved.

We have also used Coq proof assistant. However, it not being essential to our present point, we chose to live aside the detail of this procedure (see section 4.3).

4.3 Problems, Mistakes

As usual when using formal proof tools, there are several ways to formalize or to annotate programs. Choices made during at this stage are very important for future proofs. For example, declaring a function as an axiomatic theory or as a predicate will suppose corresponding proofs to be different. We can make a similar remark with data-types used in programs.

For these reasons, using "good" annotations which allows automatic provers to prove verification conditions (VC) successfully is a clever way to go about it.

When we deal with 40,000 lines of undocumented code, another critical part of the work consists in "correctly" isolating the piece of code that we want to prove. The code can use global variables, initializations made by other functions, or use intricate data-types and so on.

In the following paragraphs, these problems and associated mistakes are discussed.

Isolating a Part of Program. Generally speaking, the analyzed function must be free of external calls. More precisely if a function is called from it, it has to be incorporated in the code (like macro expansion) or, at least, independently proved.

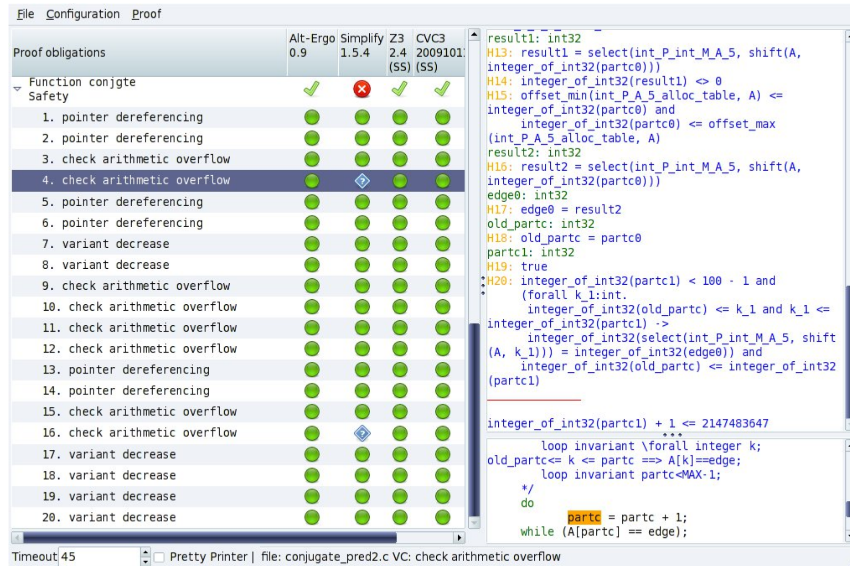


Fig. 3. Graphical Interface: Safety

Next, data types must be simplified. Even if Frama-C can cope with simple structures, it is better to have a first pass on them (unions suppression, typedef expansion and so on).

How to Make Good Annotations? As previously explained, ACLS is a language which is used to annotate C programs. Annotating an existing program consists in choosing properties (comportment, results,...) that the user wants to be "confirmed", such as preconditions, loop invariants, post conditions. In our case, for example, one of the most relevant properties we proved is that *the result B is the conjugate of the partition A*. This property is stated as a postcondition.

As usual, there are several ways to formalize annotations. Particularly when using external provers, a good method is to know how provers work. Here, we have to remember that the automatic provers are SMT solvers (see section 2.2).

As an example, we can give the definition of `countIfSup`. In a first formalization we wrote it as an "axiomatization". But due to another problem that we will describe in the next paragraph, we needed to make some proofs in Coq which used `countIfSup`. Then, to make it easier for Coq, we decided to try to define it inductively. Thanks to this other definition, some conditions were automatically proved by SMT solvers. This example shows how important formalization choices can be.

In the next paragraph, we will explain and illustrate how Coq allowed us to correct some errors in our annotations.

Why Coq? Once annotations are completed, the method consists in using automatic provers (using gWhy for example). As previously explained, if all proof obligations are proved by at least one prover, the work can be considered as finished. But, if one or more proof obligations is/are still unproved, several approaches are possible: the first one consists in verifying that annotations are "sufficient", that is to say a precondition or a loop invariant is not missing. Another approach, when the user suppose that his annotations are correct, is to use an external non automatic prover to try to prove proof obligations that have not been verified previously.

In our case, we used the interactive theorem prover Coq twice. The first time was because a postcondition had not been proved by SMT provers. When we began Coq proof, we discovered that the definition of `countIfSup` was incomplete: the second part of the "`||`" (logical or) was missing.

The second time we used Coq was to prove a loop invariant. Similarly, we detected another incompleteness in `countIfSup` definition ($j < MAX$ instead of $j \leq MAX$). Proof assistants are well adapted to detect this kind of problems. Indeed, building formal proofs manually, a user can easily see which hypotheses are necessary.

After having corrected and replaced the "axiomatization" of `countIfSup` by a predicate, all proof obligations have been proved by at least one automatic prover.

Note that the new definition allowed us to remove from the annotations one additional lemma which, at first, appeared necessary.

Other Vicissitudes. Among the main encountered difficulties, we can mention the confidence in the provers we used. In our case, one of the versions of CVC3 was faulty and proved all VC correct, even when they were false. For this reason we decided to consider that a proof obligation was proved when at least two automatic provers succeed on proving it. It is the case for all our obligations except one (VC # 23 is only proved by Simplify). The proof of VC # 23 is in progress using Coq.

5 Conclusion and Future Work

We have isolated and formally proved one of the key commands of the SCHUR software. This work reinforced us in the idea of formally proving chosen parts of software of the same kind, composed of 40,000 lines of undocumented code.

Thanks to this approach, we have focused on critical points (such as particular initializations of arrays and appropriate bounds) from the original code and by extension, we have understood the progression axis of the methodology. In particular, it is better to know how SMT automatic provers work to try to make a "good" annotation so that obligation proofs will be more easily proved by them. In the methodology, non automatic external provers like Coq may be used to refine annotations, and to prove obligations when no automatic provers succeed.

The conjugate function is a basic brick of combinatorics. This give us perspective to prove other functions. Therefore, as a future work, the second step is to prove algorithms relying on exhaustive enumeration algorithm, such as computation of Littlewood-Richardson coefficients, Koskas numbers, Koskas matrices, representation multiplicity in tensor product decompositions, etc.

The final objective will be to build proved libraries usable for scientific community.

References

1. Butelle, F., King, R., Toumazet, F.: SCHUR, an interactive program for calculating properties of Lie groups and symmetric functions, http://schur.sourceforge.net. Release 6.06.
2. MacDonald, I.G.: Symmetric Functions and Hall Polynomials. Clarendon Press, Oxford University Press (New York) (1979) 2nd edition in 1998.
3. King, R.C., Bylicki, M., Karwowski, J., eds.: Symmetry, Spectroscopy and SCHUR. Nicolaus Copernicus University Press (2006)
4. Correnson, L., Cuoq, P., Puccetti, A., Signoles, J.: Frama-C User Manual, Beryllium release, http://frama-c.cea.fr
5. Marché, C., Moy, Y.: Jessie Tutorial, http://frama-c.cea.fr/jessie/jessie-tutorial.pdf. Release 2.21.
6. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.cea.fr/download/acsl-implementation-Beryllium-20090902.pdf
7. Filliâtre, J.C., Marché, C., Moy, Y., Hubert, T., Rousset, N.: Why is a software verification platform, http://why.lri.fr. Release 2.21.
8. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10) (1969) 576–580 and 583
9. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3) (2005) 365–473 Release 1.5.4.
10. Conchon, S., Contejean, E., Bobot, F., Lescuyer, S.: Alt-Ergo is an automatic theorem prover dedicated to program verification, http://ergo.lri.fr. Release 0.9.
11. Microsoft Research: Z3 An Efficient SMT Solver, http://research.microsoft.com/en-us/um/redmond/projects/z3. Release 2.4.
12. Barrett, C., Tinelli, C.: CVC3. In Damm, W., Hermanns, H., eds.: Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07). Volume 4590 of Lecture Notes in Computer Science., Springer-Verlag (July 2007) 298–302 http://cs.nyu.edu/acsys/cvc3 Release 20091011.
13. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. EATCS, Texts in Theoretical Computer Science. Springer Verlag (2004)
14. Shankar, N., Owre, S., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS prover guide, http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
16. Andrews, G.E.: The theory of partitions. Cambridge University Press (1984)
17. Jacobi, C.G.J.: De functionibus alternantibus earumque divisione per productum e differentiis elementorum conflatum. Journal für die reine und angewandte Mathematik (Crelles Journal) **22** (1841) 360–371 Reprinted in Gesammelten Werke III, G. Reimer, Berlin, 1884.

18. Littlewood, D.E.: The Theory of Group Characters. Oxford University Press (1950) second edition.
19. Lascoux, A., Pragacz, P.: S-function series. J. Phys. A: Math. Gen. **21** (1988) 4105–4118
20. Newell, M.J.: On the representations of the orthogonal and symplectic groups. In: Proc. Roy. Irish Acad., Section A: Mathematical and Physical Sciences. Volume 54. (1951) 143–152

Résumé : Les travaux présentés dans ce document se situent, pour une grande majorité, à l'inter-domaine entre les preuves formelles et d'autres domaines de l'informatique (arithmétique à virgule flottante, vérification, combinatoire algébrique) ou des mathématiques (analyse numérique, théorie des nombres, géométrie). Deux motivations principales sont à l'origine de ces travaux : premièrement les preuves formelles sont relativement récentes au regard des mathématiques mais également au regard de l'informatique et de la logique, deuxièmement on veut prouver formellement la résolution de problèmes critiques, que cela soit au sens de prouver directement des programmes ou de prouver des algorithmes ou des propriétés de correction par rapport à des spécifications. L'utilisation de ces méthodes est bénéfique pour les communautés concernés par ces interactions. Pour celle des preuves formelles, cela permet d'accroître leur développement, aussi bien du point de vue des concepts théoriques nécessaires que pour la facilité d'utilisation. Pour celle du domaine considéré, de se prévaloir d'une preuve formelle de son problème critique et d'envisager éventuellement de poursuivre l'utilisation de telles méthodes, sûres.

Les quatre premiers chapitres sont dédiés à la présentation d'exemples d'interaction entre les preuves formelles et un domaine critique d'application. Nous présentons respectivement, les travaux concernant l'analyse numérique, les réseaux de Petri, l'arithmétique flottante, le calcul formel. Ces chapitres présentent les contextes scientifiques et historiques liés aux articles écrits suite à ces travaux, sans rentrer dans les détails techniques se trouvant dans les publications. L'avant-dernier chapitre est tout spécifiquement dédié à la problématique des nombres réels dans Coq. Le dernier chapitre présente un bilan et des perspectives.

Mots clés : preuves formelles, spécification, certification, Coq, nombres réels, preuves de programmes numériques, calcul formel, formalisation des mathématiques, logique, théorie des types, vérification, analyse numérique, arithmétique des ordinateurs, raffinement de réseaux de Petri, automatisation de preuves