Utilisation et certification de l'arithmétique d'intervalles dans un assistant de preuves

Francisco Cháves

N° d'ordre : 420 N° attribué par la bibliothèque : 07ENSL0420

ÉCOLE NORMALE SUPÉRIEURE DE LYON
Laboratoire de l'Informatique du Parallélisme

THÈSE

présentée et soutenue publiquement le 28 septembre 2007 par

Francisco José CHÁVES ALONSO

pour l'obtention du grade de

Docteur de l'École Normale Supérieure de Lyon
Spécialité : Informatique

au titre de l'École Doctorale de Mathématiques et d'Informatique Fondamentale de Lyon

---

Utilisation et certification de l'arithmétique d'intervalles dans un assistant de preuves

---

Directeurs de thèse : Marc DAUMAS et Nathalie REVOL

Après avis de : Fabienne JÉZÉQUEL
Jean-Michel MOREAU

Devant la commission d'examen formée de :

Marc DAUMAS	Membre
Fabienne JÉZÉQUEL	Membre/Rapporteuse
Jean-Michel MOREAU	Membre/Rapporteur
César MUÑOZ	Membre
Nathalie REVOL	Membre
Gilles VILLARD	Membre

Remerciements

Merci tout d'abord à mes chers directeurs de thèse Marc et Nathalie de m'avoir encadré, aidé, encouragé, soutenu, motivé, ... Ce travail n'aurait pu être possible sans eux. Leur direction a été très bonne pendant tout le déroulement de la thèse.

Merci à César pour toute sa collaboration pendant la thèse.

Merci aux rapporteurs Fabienne et Jean-Michel d'avoir relu le document et aussi de leurs commentaires et remarques.

Merci aux membres du jury d'être venus m'écouter et d'avoir partagé ce moment avec moi.

Merci à Nicolas pour m'avoir aidé et encouragé pendant toute la thèse, par ses mots en Espagnol de temps en temps et l'exemple de la fourmi.

Merci à Serge d'avoir été mon interprète au début de la thèse et aussi pour m'avoir aidé à résoudre les problèmes techniques liés à l'ordinateur.

Merci à toute l'équipe Arénaire, spécialement à Gilles et à Sylvie, merci encore à Ahmed, Arnaud, Claude-Pierre, Damien, Édouard, Florent, Hong Diep, Ilia, Jean-Luc, Jérémie, Nicolas J., Romain, Saurabh, Vincent.

Merci aux directeurs du LIP Jean-Michel et Frédéric et aussi merci à Corinne, Edwige, Isabelle, Danielle, Catherine, Audrey et Simone.

Merci à NASA Langley et au National Institute of Aerospace pour le cours de PVS.

Merci à MATHLOGAPS et à son représentant à l'ENSL Pascal, non seulement pour le financement de la thèse (c'est déjà beaucoup) mais aussi pour l'opportunité de discuter avec les autres participants au programme pendant les ateliers, le stage à Munich, etc.

Merci à Martin pour sa direction et son accueil pendant le stage à Ludwig Maximilians Universität München, et aux gens de son équipe. Merci aussi à Carl Achim et Regina pour les échanges pendant le stage.

Merci à Sylvain pour son aide avec les exemples de normes infinies.

Merci à mes cobureaux Sylvie, Guillaume M., Christoph, Nicolas, Guillaume R. et Ivan.

Merci à Eduardo pour tout son appui et ses services, à mes collègues colombiens Marcela, Tiberio et Rodrigo.

Merci à Claudia pour ses constants encouragements.

Merci au père Bonnet pour son amitié et ses conseils.

Merci au Dr. Bouchet et au Dr. Perrone.

Merci à ma famille, spécialement à ma mère.

Merci à mes amis Alexis, Ana, Arnaud, Camilo, Constanza, Diana, Dino, Dragisa, Edith, Gilberto, Jaime Uriel, José, Juan Pablo, María Elena, Margarita, Marianne, Marivel, Mercedes, Nancy, Patricia, Remi, Rafael, Ricardo, Santiago, Stéphane, etc.

Merci aux auteurs de PVS et aux gens qui contribuent aux logiciels libres que nous avons utilisés (LaTeX, emacs, kile, Debian, ...).

Enfin, merci à tous ceux qui ont contribué d'une manière ou d'une autre au succès de cette thèse.

Table des matières

Chapitre 1

Introduction

De plus en plus de calculs de surveillance, contrôle etc. sont effectués de façon logicielle. Pour certains de ces calculs, une erreur peut avoir des conséquences dramatiques : tel est le cas des logiciels utilisés dans les moyens de transport (avions, bus, automobiles...), par les équipes médicales (administration des radiations pour des maladies comme le cancer, dosage de médicaments, moniteurs de surveillance ...) ou dans l'armée (systèmes de défense, parties d'armes, alarmes...). Ceux-ci font partie de la liste, toujours plus longue, des applications critiques pour la vie ou la sûreté. Une erreur dans ce type de logiciel peut être très coûteuse, comme en témoigne la destruction de la fusée Ariane 5 lors de son premier vol en 1996 [1] ou peut coûter des vies humaines comme le prouvent la défaillance d'un antimissile Patriot en 1991 [2] et l'overdose de radiation à l'Instituto Oncologico Nacional à Panamá en 2000 [3].

Cependant, il est bien connu que les logiciels et les bibliothèques contiennent des erreurs. Les méthodes traditionnelles pour détecter ces erreurs ne sont pas suffisantes pour démontrer qu'un programme est correct, c'est-à-dire qu'il ne contient pas d'erreurs. C'est pour cette raison que le projet *Common Criteria* [4] travaille à développer des critères pour l'évaluation de la sécurité des technologies de l'information. Le Canada, la France, l'Allemagne, les Pays-Bas, le Royaume-Uni et les États-Unis participent à ce projet, pour développer un standard international pour l'évaluation de la sûreté du logiciel. Ils ont défini différents niveaux de sûreté du système. Le niveau le plus haut est l'*Evaluation Assurance Level 7* (EAL7) qui est décerné aux produits qui ont été vérifiés formellement, spécifiquement en utilisant un assistant de preuves.

Aujourd'hui les assistants de preuves tels que PVS peuvent être utilisés pour certifier qu'un programme est correct [5] mais ils ne sont pas fréquemment utilisés par les programmeurs. La raison est que, outre l'entraînement demandé pour les utiliser, les interfaces d'aide à la preuve ne sont pas conviviales pour tout le travail qui ne se fait pas automatiquement. Pour remédier à ce problème, les partisans des méthodes formelles travaillent à minimiser l'interaction avec les assistants de preuves jusqu'à ce que, idéalement, aucune interaction ne soit requise. L'expression *méthodes formelles invisibles* est utilisée pour désigner ce type de travaux [6, 7].

Dans cette direction, on peut citer les travaux de Jean-Christophe Filliâtre avec l'ou-

til *Why* pour démontrer la correction de programmes impératifs annotés avec assertions [8], de Guillaume Melquiond et al. avec l'outil Gappa pour vérifier et prouver des propriétés de programmes numériques qui utilisent les nombres à virgule flottante ou à virgule fixe [9] et ceux de César Muñoz *et al.* [10, 11] avec le développement d'une bibliothèque d'arithmétique d'intervalles en PVS. La bibliothèque de César Muñoz *et al.* est utile pour démontrer des inégalités et certifier des bornes d'expressions en utilisant l'arithmétique d'intervalles. Cependant, cette arithmétique peut retourner des intervalles trop larges, principalement à cause du phénomène de décorrélation des variables. Les modèles de Taylor sont utilisés pour réduire cette décorrélation.

Le but de cette thèse est de construire une bibliothèque de modèles de Taylor pour l'assistant de preuves PVS. La bibliothèque peut être utilisée pour dériver des bornes plus ou moins précises d'expressions arithmétiques, et aussi pour certifier des inégalités ou bornes d'expressions. Disposer d'une méthode pour vérifier des expressions avec un assistant de preuves permet de certifier certaines parties de logiciels de missions critiques.

1.1 Assistants de preuves

Les assistants de preuves sont une classe de programmes informatiques qui permet à des utilisateurs d'écrire des théories mathématiques composées de définitions, d'axiomes et de théorèmes. La majorité de ces programmes sont écrits en langage de programmation fonctionnelle comme Lisp et ML. L'utilisateur doit démontrer chaque théorème dans le système.

Le problème de l'utilisateur doit être formalisé en utilisant le langage de l'assistant de preuves. Différents choix sont disponibles pour le langage : principalement la logique du premier ordre (classique ou intuitionniste), ou la logique d'ordre supérieur. Les logiques d'ordre supérieur permettent, entre autres choses, d'utiliser des variables qui représentent des fonctions et aussi des variables qui représentent des prédicats dans les quantificateurs, et elles étendent le calcul des prédicats du premier ordre. Les logiques d'ordre supérieur sont typées, c'est-à-dire que chaque expression du système a une signature.

Chaque assistant de preuves a son langage propre. Le langage défini par l'assistant de preuves ne contient pas d'ambiguïté et peut donc être reconnu par un ordinateur. Cela permet au système de manipuler les expressions définies par l'utilisateur d'une manière purement syntaxique.

Les preuves de théorèmes peuvent être décrites de différentes manières : en donnant des étapes d'inférence, par l'application de tactiques ou stratégies, en démontrant des sous-preuves ou lemmes, etc. Le système vérifie que chaque étape donnée par l'utilisateur dans la preuve est correcte. La partie du système chargée de garantir qu'une preuve est correcte s'appelle le *vérificateur de preuves*. Dans certains systèmes, la preuve peut être écrite et vérifiée interactivement. Chaque assistant de preuves étant développé indépendamment, sa bibliothèque mathématique et les définitions utilisées pour définir

les différents concepts peuvent être différentes.

Il existe différents assistants de preuves, parmi les plus connus on trouve :

- ACL2 [12, 13, 14] ;
- Coq [15, 16, 17] ;
- HOL [18, 19] ;
- HOL Light [20, 21] ;
- Isabelle [22, 23] ;
- PVS [24, 25, 26, 27].

Pour une comparaison des différents assistants de preuves voir [28].

Les assistants de preuves ont été utilisés pour certifier des parties de logiciels et de matériels. Par exemple, l'assistant de preuves ACL2 a été utilisé pour vérifier que le microcode de la division flottante dans le processeur AMD [29] est correct ; le système PVS a été utilisé pour prouver la correction d'un algorithme pour la détection et la résolution des conflits dans l'espace aérien [30, 31]. Dans la vérification d'algorithmes et d'applications d'ingénierie, il peut apparaître des calculs qui doivent être justifiés par l'assistant de preuves. Disposer d'outils comme l'arithmétique d'intervalles peut faciliter beaucoup ce type de travail [10].

1.2 Arithmétique d'intervalles

Différentes représentations des nombres réels ont été utilisées pour calculer sur ordinateur. Ces représentations travaillent avec un nombre fini de chiffres. Par exemple, les nombres flottants (voir [32, 33] par exemple), qui utilisent une représentation de longueur fixe, permettent de calculer rapidement mais on ne peut pas représenter tous les nombres. On doit donc arrondir, ce qui implique que l'on commet une erreur d'arrondi, qui est l'erreur d'approximation entre le nombre réel et le nombre représentable. Il existe aussi d'autres représentations, comme le calcul exact avec des nombres entiers ou rationnels (voir [34] par exemple). Tous les calculs se réalisent dans cet ensemble, il n'y a pas d'erreurs de calcul. Avec cette technique la précision n'est pas limitée à une longueur fixe et la longueur des nombres peut devenir grande. Si c'est le cas, ces nombres occupent davantage de mémoire et les calculs deviennent coûteux.

Dans l'arithmétique par intervalles on remplace un nombre réel par un intervalle qui le contient. Par exemple, le nombre π peut être borné par l'intervalle $[3.1415, 3.1416]$. On calcule sur les intervalles plutôt que sur les valeurs et il est garanti que le résultat est contenu dans l'intervalle calculé. De plus on peut utiliser des représentations de longueur fixe et donc effectuer les opérations en temps constant. L'arithmétique par intervalles a été introduite dans les calculs sur ordinateurs par Ramon E. Moore en 1962 [35, 36]. De nombreuses applications ont été développées depuis son introduction [35, 36, 37, 38, 39], comme l'optimisation linéaire, la résolution d'équations différentielles ordinaires, le traitement d'incertitudes, etc.

1.2.1 Définition d'un intervalle

Un intervalle est un ensemble de nombres réels. Soient a et b deux nombres réels tels que $a \leq b$. On définit les intervalles de la façon suivante :

$$]a, b[= \{x | a < x < b\}, \tag{1.1}$$
$$[a, b[= \{x | a \leq x < b\}, \tag{1.2}$$
$$]a, b] = \{x | a < x \leq b\}, \tag{1.3}$$
$$[a, b] = \{x | a \leq x \leq b\}, \tag{1.4}$$
$$]a, +\infty[= \{x | a < x\}, \tag{1.5}$$
$$[a, +\infty[= \{x | a \leq x\}, \tag{1.6}$$
$$]-\infty, b[= \{x | x < b\}, \tag{1.7}$$
$$]-\infty, b] = \{x | x \leq b\}, \tag{1.8}$$
$$]-\infty, +\infty[= \mathbb{R}. \tag{1.9}$$

On appelle *intervalles fermés* les intervalles $[a, b]$, $[a, +\infty[$, $]-\infty, b]$, $]-\infty, +\infty[$, et l'intervalle $[a, b]$ *intervalle fermé borné*.

Les intervalles $]a, b[$, $]a, +\infty[$, $]-\infty, b[$, $]-\infty, +\infty[$ sont appelés intervalles ouverts.

On appelle *intervalles semi–ouverts* ou *intervalles semi–fermés* les intervalles $[a, b[$ et $]a, b]$.

On appelle le nombre a la borne inférieure et le nombre b la borne supérieure. On peut noter que les bornes peuvent appartenir ou non à l'intervalle.

Les intervalles sont des ensembles convexes, c'est-à-dire que si x et y appartiennent à l'intervalle, pour tout z réel tel que $x \leq z \leq y$ alors z appartient à l'intervalle.

On note en gras, comme $\boldsymbol{x}$, les variables qui dénotent des intervalles. Si $\boldsymbol{x}$ est un intervalle, on note $\underline{\boldsymbol{x}}$ la borne inférieure et $\overline{\boldsymbol{x}}$ la borne supérieure. L'intervalle qui correspond au réel x est l'intervalle fermé qui a ses deux bornes égales $[x, x]$.

On a besoin de formaliser l'arithmétique d'intervalles dans un assistant de preuves pour pouvoir justifier formellement des calculs qui apparaissent dans la vérification d'applications scientifiques [10]. En particulier Marc Daumas et Guillaume Melquiond, de l'équipe Arénaire, ont collaboré avec César Muñoz du NIA sur la résolution de conflits pour la navigation aérienne. Ils ont développé des formules plus précises que celles utilisées jusqu'alors et les ont prouvées formellement. Notre travail prolonge et généralise cette étude. Nous utilisons l'arithmétique d'intervalles qui a été développée en PVS par César Muñoz *et al.* [11, 10]. Elle utilise uniquement des intervalles fermés bornés dont les bornes sont des nombres rationnels. Nous avons choisi cette bibliothèque parce qu'elle s'adapte bien à notre besoin d'arithmétique d'intervalles et, à notre connaissance, c'est l'unique bibliothèque d'arithmétique d'intervalles en PVS. Une autre raison à ce choix est qu'il permet de définir des stratégies pour automatiser le processus de démonstration de formules, ce qui permet, dans l'esprit des méthodes formelles invisibles, de minimiser l'interaction entre l'utilisateur et l'assistant de preuves.

De plus, PVS a une vaste bibliothèque mathématique et des procédures de décision performantes, ce qui facilite le développement de nouvelles théories.

1.2.2 Opérations de base

Soient $\boldsymbol{x}$ et $\boldsymbol{y}$ des intervalles fermés bornés et c un scalaire, les opérations de base de l'arithmétique d'intervalles peuvent être calculés en utilisant leur monotonie, ce qui permet de les exprimer en utilisant les bornes des intervalles.
Pour l'addition on a

$$\boldsymbol{x} + \boldsymbol{y} = [\underline{\boldsymbol{x}} + \underline{\boldsymbol{y}}, \overline{\boldsymbol{x}} + \overline{\boldsymbol{y}}],$$

pour la soustraction

$$\boldsymbol{x} - \boldsymbol{y} = [\underline{\boldsymbol{x}} - \overline{\boldsymbol{y}}, \overline{\boldsymbol{x}} - \underline{\boldsymbol{y}}],$$

pour la multiplication par un scalaire c positif

$$c \cdot \boldsymbol{x} = [c\underline{\boldsymbol{x}}, c\overline{\boldsymbol{x}}],$$

pour la multiplication

$$\boldsymbol{x} \cdot \boldsymbol{y} = [\min(\underline{\boldsymbol{x}}\underline{\boldsymbol{y}}, \underline{\boldsymbol{x}}\overline{\boldsymbol{y}}, \overline{\boldsymbol{x}}\underline{\boldsymbol{y}}, \overline{\boldsymbol{x}\boldsymbol{y}}), \max(\underline{\boldsymbol{x}}\underline{\boldsymbol{y}}, \underline{\boldsymbol{x}}\overline{\boldsymbol{y}}, \overline{\boldsymbol{x}}\underline{\boldsymbol{y}}, \overline{\boldsymbol{x}\boldsymbol{y}})],$$

pour l'inverse
si $\boldsymbol{x}$ ne contient pas 0

$$1/\boldsymbol{x} = [1/\overline{\boldsymbol{x}}, 1/\underline{\boldsymbol{x}}],$$

pour la division
si $\boldsymbol{y}$ ne contient pas 0

$$\boldsymbol{x}/\boldsymbol{y} = \boldsymbol{x} \cdot 1/\boldsymbol{y}.$$

On peut également établir les formules pour des opérations ensemblistes :
pour l'union

$$\boldsymbol{x} \cup \boldsymbol{y} = [\min(\underline{\boldsymbol{x}}, \underline{\boldsymbol{y}}), \max(\overline{\boldsymbol{x}}, \overline{\boldsymbol{y}})],$$

pour l'intersection

$$\begin{aligned} \boldsymbol{x} \cap \boldsymbol{y} &= [\max(\underline{\boldsymbol{x}}, \underline{\boldsymbol{y}}), \min(\overline{\boldsymbol{x}}, \overline{\boldsymbol{y}})] && \text{si } \boldsymbol{x} \cap \boldsymbol{y} \neq \emptyset \\ &= \emptyset && \text{sinon.} \end{aligned}$$

La définition précédente des opérations arithmétiques et ensemblistes ne suffit pas quand il s'agit de les utiliser avec un assistant de preuves. On doit également fournir les propriétés qui caractérisent ces opérations. Les opérations sur les intervalles satisfont la propriété d'inclusion : si $x \in \boldsymbol{x}$ et $y \in \boldsymbol{y}$, alors

$$x + y \in \boldsymbol{x} + \boldsymbol{y},$$

$$x - y \in \boldsymbol{x} - \boldsymbol{y},$$

$$c \cdot x \in c \cdot \boldsymbol{x},$$
$$x \cdot y \in \boldsymbol{x} \cdot \boldsymbol{y},$$
$$x/y \in \boldsymbol{x}/\boldsymbol{y}, \qquad \text{si } 0 \notin \boldsymbol{y},$$
$$x \in \boldsymbol{x} \cup \boldsymbol{y} \quad \text{et} \quad y \in \boldsymbol{x} \cup \boldsymbol{y},$$
$$x \in \boldsymbol{x} \cap \boldsymbol{y} \quad \text{si } x \in \boldsymbol{x} \text{ et } x \in \boldsymbol{y}.$$

1.2.3 Décorrélation des variables

Quand une variable apparaît plusieurs fois dans une expression, l'arithmétique des intervalles « oublie » qu'il s'agit de la même variable et par conséquent le résultat calculé est trop large. Ce problème est connu sous le nom de *décorrélation des variables*. Par exemple, si $x \in [0, 1]$ et que l'on veut calculer

$$x - x,$$

il est clair que l'intervalle $[0, 0]$ borne cette expression ; mais l'arithmétique des intervalles évalue l'expression

$$[0, 1] - [0, 1] = [-1, 1].$$

Le résultat est correct au sens où il vérifie la propriété d'inclusion, mais il est plus grand que le résultat attendu [0,0]. Si on reprend la propriété d'inclusion pour la soustraction, on a pour tout $x \in \boldsymbol{x}$ et pour tout $y \in \boldsymbol{y}$,

$$x - y \in \boldsymbol{x} - \boldsymbol{y},$$

en remplaçant l'intervalle $\boldsymbol{y}$ par l'intervalle $\boldsymbol{x}$ on obtient

$$x - y \in \boldsymbol{x} - \boldsymbol{x}.$$

L'arithmétique des intervalles ne conserve pas la corrélation entre les deux occurrences de la même variable.

Plusieurs techniques ont été développées pour améliorer les résultats de l'arithmétique d'intervalles, nous en présentons deux dans ce qui suit.

1.2.4 Techniques pour remédier au problème de la décorrélation

Partitionnement d'intervalles

Moore dans sa thèse [35] présente une méthode qu'il appelle *raffinement* pour améliorer l'approximation d'une fonction rationnelle d'intervalle $\boldsymbol{f} : D \to \mathbb{I}$ où $\mathbb{I}$ est l'ensemble de tous les intervalles. En notant $w([a, b]) = b - a$ la longueur de l'intervalle $[a, b]$, le théorème 4.2 de [35] établit que pour un intervalle $\boldsymbol{x} \subset D$, il existe un nombre réel K tel que $w(\boldsymbol{f}(\boldsymbol{x})) \leq K w(\boldsymbol{x})$, c'est-à-dire que la largeur de l'intervalle de sortie peut diminuer si l'on diminue la largeur de l'intervalle d'entrée $\boldsymbol{x}$ [40].

De plus, on a l'inclusion suivante :

$$\boldsymbol{x} \subseteq \boldsymbol{y} \Longrightarrow \boldsymbol{f}(\boldsymbol{x}) \subseteq \boldsymbol{f}(\boldsymbol{y}).$$

Ces deux propriétés conduisent à l'idée de diviser en n parties l'intervalle d'entrée $[a, b]$, de la façon suivante :

$$[a, b] = \bigcup_{i=1}^{n} [a + (i-1)\frac{b-a}{n}, a + (i)\frac{b-a}{n}],$$

et on a

$$\{f(x)|x \in [a, b]\} \subseteq \bigcup_{i=1}^{n} \boldsymbol{f}([a + (i-1)\frac{b-a}{n}, a + (i)\frac{b-a}{n}]) \subseteq \boldsymbol{f}([a, b]).$$

Autrement dit, on peut améliorer la précision de l'intervalle de sortie en partitionnant l'intervalle d'entrée.

Modèles de Taylor

Une solution permettant de limiter la décorrélation consiste à remplacer l'expression ou la fonction à calculer, sur l'intervalle d'entrée $\boldsymbol{x}$ considéré, par un couple formé d'un polynôme P et d'un intervalle $\boldsymbol{r}$ [41]. Le polynôme P ressemble au début du développement de Taylor de la fonction, typiquement au centre de l'intervalle d'entrée $\boldsymbol{x}$, et c'est l'utilisation d'un tel polynôme qui permet de réduire le problème de décorrélation des variables. L'intervalle $\boldsymbol{r}$, également appelé reste intervalle, est un encadrement de l'erreur commise en approchant la fonction par le polynôme P sur l'intervalle d'entrée $\boldsymbol{x}$, il correspond typiquement au reste de Lagrange.

Plus précisément, pour obtenir le polynôme P et l'intervalle $\boldsymbol{r}$ d'une fonction $f : \boldsymbol{x} \subset \mathbb{R} \rightarrow \mathbb{R}$ de classe C^{n+1}, nous pouvons utiliser le théorème de Taylor avec reste de Lagrange sur l'intervalle $\boldsymbol{x}$: pour tout $x \in \boldsymbol{x}$ et tout $x_0 \in \boldsymbol{x}$, il existe $\eta \in \boldsymbol{x}$ tel que l'on ait

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \cdots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + \frac{f^{(n+1)}(\eta)}{(n+1)!}(x - x_0)^{n+1}.$$

On peut donc écrire

$$f(x) \in f(x_0) + f'(x_0)(\boldsymbol{x} - x_0) + \frac{f''(x_0)}{2!}(\boldsymbol{x} - x_0)^2 + \cdots + \frac{f^{(n)}(x_0)}{n!}(\boldsymbol{x} - x_0)^n + \frac{f^{(n+1)}(\boldsymbol{x})}{(n+1)!}(\boldsymbol{x} - x_0)^{n+1}$$

où $x \in \boldsymbol{x}$, $x_0 \in \boldsymbol{x}$.

Le problème avec l'utilisation directe du théorème de Taylor est que nous devons dériver explicitement la fonction étudiée pour obtenir son développement de Taylor. L'idée des modèles de Taylor est de définir une arithmétique : opérations arithmétiques et algébriques, fonctions élémentaires, etc. de manière à pouvoir construire le polynôme P et le reste intervalle $\boldsymbol{r}$ d'une expression à partir de ses composants en utilisant l'arithmétique ainsi définie [42, 43]. Nous donnerons plus d'informations sur les modèles de Taylor dans la première partie du chapitre 3.

1.3 Comment lire ce document

On pourra lire dans un premier temps l'introduction et la conclusion pour avoir une idée générale de la thèse. Le lecteur intéressé par un sujet précis mais pas par l'ensemble de la thèse lira avec profit l'introduction et la conclusion de la thèse, le résumé et la conclusion de chaque chapitre ainsi que les chapitres qui le concernent. Dans une dernière étape, nous suggérons de lire tout le documment.

Nous présentons maintenant l'organisation du document. Les deux premiers chapitres sont introductifs, le présent chapitre 1 étant une introduction aux concepts généraux sur les assistants de preuves et l'arithmétique par intervalles. Dans la section dédiée à l'arithmétique par intervalles, nous présentons les opérations de base, le problème de la décorrélation des variables et quelques techniques pour remédier à ce problème.

Le chapitre 2 constitue une introduction détaillée à PVS, l'assistant de preuves que nous utilisons. Nous présentons dans la section dédiée au langage de spécification les différents types de déclarations du langage llustrés par des exemples pour la majorité des commandes. La deuxième section constitue une introduction à l'assistant de preuves de PVS, à son utilisation et comporte une courte description des commandes utilisées dans ce document. Si le lecteur connaît déjà PVS, il peut omettre la lecture de ce chapitre. Le chapitre 3 présente les théories de base définies pour la construction des modèles de Taylor en PVS, ainsi que la construction des opérations de l'arithmétique des modèles de Taylor. Les opérations que nous avons construites sont l'addition, l'opposé, la soustraction, le produit par un scalaire et la multiplication. Nous présentons également dans ce chapitre notre construction des opérations inverse et racine carrée de l'arithmétique des modèles de Taylor.

Le chapitre 4 présente notre construction des modèles de Taylor pour l'exponentielle, l'arc-tangente et le sinus ainsi qu'une méthode générale de construction des modèles de Taylor. Dans la section portant sur le modèle de Taylor de l'arc-tangente, nous détaillons la théorie des séries alternées que nous avons dû développer en PVS pour sa construction.

Le chapitre 5 présente comment utiliser l'interpréteur de PVS pour évaluer un modèle de Taylor ainsi que les stratégies que nous avons développées pour simplifier et automatiser l'utilisation de nos travaux en PVS. Le chapitre 6 présente une application des modèles de Taylor implantés à la certification de la qualité d'un approximant polynomial. Les conclusions et perspectives de ce travail sont présentées au chapitre 7.

Chapitre 2

Introduction à PVS

Dans ce chapitre nous présentons PVS, l'assistant de preuves que nous avons utilisé. L'objectif est ici de donner une vue d'ensemble du système PVS : dans la première partie du chapitre nous présentons le langage, puis nous présentons ensuite le système de preuves. Ce chapitre est conçu comme un tutoriel au langage de spécification de PVS, en effet il n'en existe pas en français et nous voulions combler ce manque.

Dans le domaine de l'informatique, l'élaboration d'un logiciel passe par différentes étapes. L'une des plus importantes, avant de programmer, est de définir très précisément, ce *que* doit faire le logiciel. Cette partie du procédé est connue comme la *spécification* du logiciel. La spécification d'un logiciel est l'ensemble des fonctionnalités qu'un programme doit fournir. La spécification décrit *ce que* le programme doit faire, alors que le programme décrit *comment* cela se fait. Une spécification peut être implantée par différents logiciels.

Pour mettre au point une spécification, différents langages peuvent être utilisés [44, 45]. Le langage utilisé doit éviter les ambiguïtés et doit permettre de parler de choses qui se trouvent dans les langages de programmation, mais de manière plus abstraite, c'est-à-dire sans entrer dans les détails d'implantation. Pour cette raison sont préférés des langages qui sont basés sur la logique et qui ont une puissance d'expressivité suffisante. On préfère également que ces langages soient des langages formels, c'est-à-dire des langages qui sont décrits à l'aide de règles mathématiques précises.

Le système de vérification de prototypes PVS (*Prototype Verification System*) est un langage de spécification intégré avec des outils et un assistant de preuves[1]. Il consiste en un environnement pour le développement et l'analyse de spécifications formelles qui permet l'élaboration de théories et de preuves. Il est développé par SRI International. L'assistant de preuves de PVS est écrit dans le langage de programmation Lisp.

[1] `http://pvs.csl.sri.com/`

Une spécification PVS est un ensemble de théories. Une théorie est une liste de déclarations de types, constantes et formules. La déclaration des types et des constantes peut inclure des définitions. Une déclaration qui n'inclut pas une définition est dite non interprétée. Parmi les déclarations de formules, on trouve les axiomes et les théorèmes [26].

2.1 Le langage de spécification

Le langage PVS est un langage basé sur la logique classique d'ordre supérieur, c'est-à-dire, entre autres, qu'il autorise des variables à représenter des fonctions ou des prédicats. Un prédicat est une fonction qui a son image dans l'ensemble des booléens : `TRUE` et `FALSE`. PVS a comme base la logique classique du premier ordre et les termes appartiennent au λ-calcul. Le langage PVS est typé, chaque expression du système a une signature.

Afin de familiariser le lecteur avec PVS, nous allons maintenant donner un aperçu de sa syntaxe. Cette partie se veut être une introduction au langage de spécification PVS, introduction qui n'existe pas dans les documents accompagnant PVS. Il ne s'agit cependant pas d'un document présentant l'intégralité du langage PVS, un tel document serait trop long. Par la suite, nous insérerons des extraits en PVS pour illustrer notre propos. Les commentaires en PVS sont écrits avec le caractère `%`. Le reste de la ligne qui suit le `%` est considéré comme un commentaire.

2.1.1 Théories

Les composants principaux du langage sont les théories. Une théorie commence par le nom de la théorie (un identificateur), suivi de deux points puis du mot clé `THEORY`. Le mot clé `BEGIN` dénote le début des déclarations et il a son correspondant `END` suivi du nom de la théorie qui dénote sa fin.

```
theory_name : THEORY
  BEGIN
   ...
  END theory_name
```

Une théorie peut avoir des paramètres, dans un tel cas on dit que la théorie est paramétrée. Les paramètres sont inclus dans des crochets et séparés par une virgule. Par exemple, la théorie des modèles de Taylor a comme paramètres un nombre naturel positif `N` et un intervalle appelé `domInterval`. En PVS le type des entiers relatifs est `int`, celui des entiers naturels est `nat` et celui des entiers naturels strictement positifs est `posnat`.

```
taylor_model  [ N : posnat, domInterval : Interval ] : THEORY
```

```
  BEGIN
   ...
  END taylor_model
```

Les paramètres d'une théorie peuvent avoir des contraintes à satisfaire, qui sont appelées hypothèses. Les hypothèses sont définies entre les mots clés `ASSUMING` et `ENDASSUMING`.

Par exemple la théorie `monoid`, donnée ci-dessous, prend comme hypothèse que l'opération `o` est associative et possède un élément neutre `e`. Précisons que le mot clé `VAR` permet de déclarer des variables.

```
monoid  [ G : TYPE, e : G, o : [G,G->G] ] : THEORY
  BEGIN

    ASSUMING
      a, b, c : VAR G

      associativity: ASSUMPTION a o (b o c) = (a o b) o c

      unit: ASSUMPTION a o e = a AND e o a = a

    ENDASSUMING
  ...

  END monoid
```

Dans PVS, une théorie peut être construite à partir d'un ensemble de théories existantes. Par exemple, la théorie des modèles de Taylor est construite sur la théorie des polynômes. Une théorie est importée en utilisant le mot clé `IMPORTING`.

```
taylor_model  [ N : posnat, domInterval : Interval ] : THEORY
  BEGIN

  IMPORTING polynomials_ext
  ...

  END taylor_model
```

Les déclarations sont les composants principaux des spécifications. PVS permet une diversité de déclarations telles que des déclarations de types, des déclarations de variables, des déclarations de constantes, des déclarations récursives, des déclarations de formules, etc. Une déclaration peut utiliser d'autres déclarations définies préalablement ou importées d'autres théories. Chaque déclaration est identifiée dans la théorie par un identificateur.

2.1.2 Variables

Les variables sont déclarées comme étant locales à la théorie, c'est-à-dire qu'elles sont visibles uniquement dans la théorie qui les définit. Les variables de PVS sont logiques, elles ne sont pas des variables de programmation. Elles sont utilisées pour donner un nom et un type associé.

Le mot clé `VAR` permet de déclarer des variables. Voici quelques exemples de déclarations :

```
i, j : VAR int
n : VAR nat
x : VAR real
I : VAR Interval
```

Les variables sont utilisées pour simplifier les déclarations de types, prédicats ou de fonctions qui suivent leur définition. Pour les exemples suivants, supposons qu'on a les déclarations de variables ci-dessus et déclarons la fonction successeur :

```
succ(n) : nat = n + 1
```

Il n'est pas nécessaire de donner le type de `n`, il est déduit de sa déclaration.

On peut définir aussi une variable locale à la déclaration, comme dans :

```
succ(k:nat) : nat = k + 1
```

Les variables locales à la déclaration cachent la définition précédente, comme dans :

```
succ(i:nat) : nat = i + 1
```

Ici, la variable `i` de la fonction `succ` est de type `nat` et cache la déclaration de la variable `i` de type `int`. Les variables locales peuvent être utilisées dans les expressions de type, les quantificateurs, les λ-expressions et les expressions `LET` et `WHERE`.

2.1.3 Constantes

Dans PVS il y a deux classes de constantes : *interprétées* et *non interprétées*. Les constantes interprétées ont une valeur définie alors que les constantes non interprétées n'ont pas de valeur définie. Les propriétés des constantes non interprétées, telles que leur valeur, peuvent être données par des axiomes. Il est nécessaire que le type de la constante ne soit pas vide (voir §2.1.4).

Voici quelques exemples de définitions de constantes, la différence avec une déclaration de variable est l'absence du mot clé `VAR` et, si la constante est interprétée, la présence de sa valeur :

```
s : int = 7
N : nat
```

La constante `s` est interprétée, alors que la constante `N` est non interprétée.

2.1.4 Types

Les types jouent un rôle essentiel dans PVS. Un type est considéré comme un ensemble de valeurs. Pour définir un type on utilise le mot clé `TYPE`. Pour spécifier un type non vide, un type qui a au moins un élément, on utilise le mot clé `TYPE+` ou `NONEMPTY_TYPE`. Pour déterminer si deux types sont égaux, PVS utilise l'équivalence structurelle, c'est-à-dire qu'il examine la définition du type, et pas seulement le nom. Tout comme les constantes, les types peuvent être *non interprétés* ou *interprétés*.

Types non interprétés

Pour les types non interprétés il n'y a pas de caractéristiques particulières, seulement qu'ils sont disjoints des autres types. Ils sont importants pour définir des spécifications abstraites, sans les restreindre à un type particulier, comme par exemple pour spécifier une liste sans spécifier le type de ses éléments.

Un type non interprété est déclaré de la forme :

```
T : TYPE
S : TYPE+
U : NONEMPTY_TYPE
```

La première ligne déclare un type non interprété `T`, les deuxième et troisième lignes déclarent `S` et `U` comme des types non interprétés non vides.

Types prédéfinis

Dans la bibliothèque *prelude* qui est écrite entièrement en PVS et qui est chargée automatiquement par PVS, on trouve, entre autres, la déclaration des types de base. Ils peuvent être utilisés pour déclarer des variables ou des constantes et aussi pour créer de nouveaux types. On mentionne quelques-uns de ces types de base dans ce qui suit :

- les booléens, `bool`;
- les entiers, `int`;
- les naturels, `nat`;
- les rationnels, `rational`;
- les réels, `real`.

Sous-types définis par des prédicats

On peut définir en PVS de nouveaux types comme sous-ensembles des types qui sont déjà définis. Le type se construit à l'aide d'un prédicat que les éléments du sous-type doivent vérifier. Les propriétés du type originel sont héritées par le sous-type. Pour démontrer que le type est non vide, on peut ajouter la clause `CONTAINING` avec un élément qui satisfait le prédicat. Voici quelques exemples, au passage notons que « est différent » s'écrit `/=` en PVS :

```
posint  : TYPE = { n : int  | n >  0 }
nzreal  : TYPE = { x : real | x /= 0 }
odd_int : NONEMPTY_TYPE = { n : int  | odd?(n) } CONTAINING 1
```

Les types dépendants sont un cas particulier de sous-types définis par des prédicats. Ils dépendent de variables définies auparavant comme :

```
upfrom( i : int ) :
            NONEMPTY_TYPE =  {s : int | s >= i} CONTAINING i
subrange( i : int, j : int ) :
            TYPE = {k : int | i <= k AND k <= j}
ssubrange( i : int, j : { j : int | j > i }) :
            TYPE = {k : int | i <= k AND k <= j}
```

Le type `upfrom` déclare les valeurs entières supérieures ou égales à `i`. Le type `subrange` déclare les valeurs comprises entre `i` et `j`, ce type pouvant être vide. Le type `ssubrange` ressemble au type `subrange` mais ce type, dépendant de `i` et `j`, impose de plus la restriction que la valeur de `j` soit strictement supérieure à la valeur de `i`.

On peut abréger la notation `{ x : T | P(x) }` en `(P)`, par exemple en supposant que nous ayons le prédicat défini par

```
prime?( n : nat ) : bool = ...
```

on a que la déclaration de type

```
primes : TYPE = { n : nat | prime?(n) }
```

est équivalente à la déclaration

```
primes : TYPE = (prime?)
```

Une alternative est d'abréger le type des paramètres comme celui de `j` dans la définition du type `ssubrange` :

```
ssubrange( i : int, ( j : int | j > i )) : TYPE =
             {k : int | i <= k AND k <= j}
```

Avec les sous-types définis par des prédicats, la vérification de types (*typechecking*) est indécidable, c'est-à-dire que le système ne peut pas vérifier automatiquement tous les types du système. Le système de types de PVS vérifie la correction simple de types et il produit des obligations de preuves correspondant aux sous-types. Ces obligations de preuves sont nommées *Type-correctness conditions* (TCCs). Par exemple si on utilise le type `ssubrange` :

```
ssubrange( 1, 5)
```

l'obligation de preuve `5 > 1` est produite. Les obligations de preuves sont produites quand une vérification de types est effectuée, par exemple quand l'utilisateur demande la vérification des types de la théorie en utilisant la commande Emacs[2] `M-x typecheck` ou `M-x tc`. Pour voir les obligations de preuves produites, on utilise la commande Emacs `M-x show-tccs` ou `M-x tccs`. Beaucoup de ces obligations de preuves sont prouvées automatiquement par PVS en utilisant la commande Emacs `M-x typecheck-prove` ou `M-x tcp`. Quand PVS ne peut pas prouver une TCC, l'utilisateur doit fournir la preuve interactivement, le plus souvent en utilisant des stratégies simples. Pour fournir la preuve, il doit appeler l'assistant de preuves de la même façon que pour fournir la preuve d'un théorème, qui sera expliquée dans la §2.2.2.

Énumérations

Une énumération est créée en PVS en donnant la liste des éléments qui la composent. Ils deviennent les constantes du type. Chaque constante est différente des autres, et seules les constantes définies font partie du type. Les identificateurs de ces constantes peuvent être utilisés dans des expressions, comme par exemple

```
bool :  TYPE = { TRUE, FALSE }
color : TYPE = { red, green, blue }
```

Types produit

Les valeurs de type produit cartésien, appelées *tuples* ou n-uplets, sont formées par la juxtaposition de valeurs séparées par des virgules. Dans les valeurs d'un n-uplet, la position est importante, par exemple, la paire `(1, 2)` est différente de la paire `(2, 1)`. Les types des valeurs d'un n-uplet peuvent être différents pour chacun des éléments composant le n-uplet, comme l'illustre le n-uplet `(1, TRUE, red)`. Pour accéder aux éléments d'un n-uplet, on utilise les fonctions de projection `'1, '2...` par exemple,

```
(1, TRUE, red)'1 = 1
(1, TRUE, red)'2 = TRUE
(1, TRUE, red)'3 = red
```

Une autre façon d'effectuer des projections est d'utiliser les fonctions `proj_1`, `proj_2`...

Les types produits en PVS se forment en séparant les types qui les composent par des virgules. Par exemple le type `int_pair` de paires $\mathbb{Z} \times \mathbb{Z}$ en PVS est déclaré ainsi :

```
int_pair :  TYPE = [int, int]
```

et le type du n-uplet `(1, TRUE, red)` est `[int, bool, color]`.

[2] L'interface de PVS est *emacs* [46].

Types enregistrement

Les enregistrements sont similaires à des n-uplets, mais les éléments sont référencés par des étiquettes plutôt que par leur position. Les éléments d'un enregistrement sont appelés champs. Chaque champ peut contenir une valeur du type précisé dans la définition du type de l'enregistrement. Un champ est défini par son étiquette, suivi de ":" et du type correspondant. Les champs sont séparés par des virgules. La définition est délimitée par "[#" et "#]". À titre d'exemple voici quelques définitions d'enregistrements :

```
int_pair_record :  TYPE = [# left : int, right : int #]
ex_record : TYPE = [# value : int, flag : bool, col : color #]
```

Les valeurs des enregistrements sont conformes à la définition du type de chaque champ. La valeur de chaque champ est définie par l'étiquette, suivie de ":=" et la valeur. La valeur de l'enregistrement est délimitée par "(#" et "#)".

Des exemples de valeurs correspondant aux enregistrements ci-dessus sont présentés :

```
(# left := 1, right := 2 #)
(# value := 1, flag := TRUE, col := red #)
```

Pour accéder aux éléments d'un enregistrement, on utilise l'étiquette du champ, par exemple,

```
(# value := 1, flag := TRUE, col := red #)`value = 1
(# value := 1, flag := TRUE, col := red #)`flag = TRUE
(# value := 1, flag := TRUE, col := red #)`col = red
```

Les éléments sont référencés par les étiquettes et non par l'ordre dans lequel ils apparaissent ; par exemple, les deux expressions suivantes correspondent au même enregistrement :

```
(# left := 1, right := 2 #)
(# right := 2, left := 1 #)
```

Par contre, l'enregistrement

```
(# left := 1, right := 2 #)
```

est différent de l'enregistrement

```
(# left := 2, right := 1 #)
```

Types des fonctions

Les types des fonctions dans PVS sont écrits en utilisant "->" et en spécifiant le domaine (situé avant ->) et l'image (à droite de ->) à l'intérieur de "[" et "]". Par exemple le type suivant

```
operator :  TYPE = [[int, int] -> int]
```

définit le type `operator` comme le type des fonctions qui prennent un couple d'entiers (le type correspondant au couple est `[int, int]`) et qui retournent comme résultat un entier. Les parenthèses autour du type du couple ne sont pas nécessaires, on peut aussi écrire :

```
operator :  TYPE = [int, int -> int]
```

De façon alternative, le type d'une fonction peut être déclaré en utilisant le mot clé `FUNCTION` ou `ARRAY`. Par exemple, le type `[int, int -> int]` est équivalent au type `FUNCTION[int, int -> int]` ou au type `ARRAY[int, int -> int]`.

Les fonctions dans PVS sont totales, c'est-à-dire qu'elles sont définies pour toutes les valeurs de leur domaine. Il est possible de restreindre le domaine avec les sous-types utilisant des prédicats pour remplir cette condition de totalité.

Par exemple, on peut déclarer le type de la division comme :

```
div_type : TYPE = [real, { y : real | y /= 0 } -> real]
```

on peut le déclarer aussi comme :

```
nzreal  : TYPE = { x : real | x /= 0 }
div_type : TYPE = [real, nzreal -> real]
```

Un autre exemple est le type des tableaux "usuels" dans les langages de programmation, c'està-dire avec des indices allant de 1 à n :

```
T : TYPE
tableaux(n:posnat) : TYPE = [ subrange(1,n) -> T ]
```

Le type `subrange` a été défini dans cette section à la page 14.

Le type du domaine ou de l'image peuvent être des types de fonctions aussi. Par exemple :

```
ex_type : TYPE = [int -> [int -> int]]
```

C'est le type des fonctions qui prennent comme argument un entier et qui retournent une fonction des entiers sur les entiers.

Types abstraits de données

Un type abstrait de données est construit en donnant l'ensemble des constructeurs et des fonctions d'accès. Par exemple, le type des listes est :

```
list[ T : TYPE] : DATATYPE
BEGIN
  null : null?
  cons(car : T, cdr : list) : cons?
END list
```

Ici, `null` et `cons` sont appelés constructeurs. Ils permettent de construire toutes les valeurs de type `list`. Les fonctions `null?` et `cons?` sont automatiquement définies et sont les fonctions d'accès pour le type `list`. La fonction `null?` permet de tester si une liste est la liste `null`, par exemple :

```
null?(null)
```

est vrai, et

```
null?(cons(1, null))
```

est faux. De la même façon, la fonction `cons?` permet de tester si la liste est construite avec le constructeur `cons` ; par exemple :

```
cons?(null)
```

est faux, et

```
cons?(cons(1, null))
```

est vrai.

La déclaration d'un type abstrait entraîne la définition automatique d'une nouvelle théorie qui définit les axiomes, fonctions et principes d'induction utilisables par l'assistant de preuves.

2.1.5 Expressions

Les expressions sont utilisées dans les formules, les déclarations de constantes, les prédicats d'un sous-type, etc. En PVS, toutes les expressions sont typées, s'il y a des vérifications à faire, des conditions de correction de type sont émises.

Relations d'égalité

La relation d'égalité est définie pour tout type du système. Deux opérateurs sont définis, à savoir = et /=. Par exemple :

```
x = 7
x /= y
```

Les types des deux côtés de l'égalité ou de l'inégalité doivent être compatibles. Par exemple :

```
4 * x = 7
p AND q  /= r
```

sont des expressions valides si x est de type numérique et p, q et r sont de type booléen.

Des conditions de correction de type sont émises quand les expressions appartiennent à des sous-types. Le système de types rejette des expressions qui n'ont pas le même type, comme par exemple

```
TRUE /= 1
x * y = p AND q
```

Coercitions

PVS permet la surcharge de noms, c'est-à-dire que l'on peut définir le même nom avec des types différents. Dans certains cas, le vérificateur de types ne peut pas déterminer le type approprié. Par exemple

```
foo : bool
foo : [bool -> bool]   % ci-dessous, un théorème,
                       % introduit par le mot clé LEMMA
ex  : LEMMA foo = foo
```

dans le lemme ex le vérificateur de types ne peut pas déduire le type de l'expression foo. Dans de tels cas, les coercitions sont utilisées pour spécifier le type d'une expression explicitement. Une coercition est de la forme

```
expr :: T
```

elle signale au système de vérification de types que le type pour l'expression expr est T. Une application de la coercition est, dans notre exemple, de dire au système de vérification des types quel est le type utilisé :

```
ex  : LEMMA foo = foo::bool
```

Le vérificateur de types peut alors déduire que le type pour les deux occurrences de `foo` dans le lemme sont de type `bool`.

Une autre application de la coercition est d'indiquer au système de types quel est le type attendu pour le résultat de l'expression. Par exemple, si on a comme hypothèse que `b` divise `a`, on peut indiquer que le résultat de la division de `a` par `b` est entier en écrivant :

```
a/b :: int
```

Expressions arithmétiques

Les opérations usuelles sont définies en PVS. Les opérations de comparaison sont `<`, `<=`, `>` et `>=`. Les opérations arithmétiques binaires sont `+`, `-`, `*` et `/`. Les constantes numériques sont limitées aux entiers et aux rationnels. Le type de base pour l'arithmétique est le type `NUMBER`. Les entiers naturels, les entiers relatifs, les rationnels, les réels, etc. sont définis par des sous–types. Les coercitions sont réalisées automatiquement.

Expressions logiques

Les expressions logiques sont utilisées pour construire des formules du calcul propositionnel et des prédicats. Les constantes logiques sont les valeurs usuelles `TRUE` et `FALSE`. Les connecteurs propositionnels sont :

- pour la négation : `NOT`;
- pour la conjonction : `AND`, `&`;
- pour la disjonction : `OR`;
- pour l'implication : `IMPLIES`, `=>`;
- pour l'équivalence : `IFF`, `<=>`;
- pour le ou exclusif : `XOR`.

Le quantificateur universel est

```
FORALL x : P(x)
```

On peut également utiliser le mot clé `ALL` pour le quantificateur universel.

Le quantificateur existentiel est

```
EXISTS x : Q(x)
```

On peut aussi utiliser le mot clé `SOME` pour le quantificateur existentiel.

Les quantificateurs créent un environnement local pour les variables qui sont quantifiées. Ces variables peuvent inclure un type ou une restriction. Par exemple `x`, `y` et `z` doivent être des entiers naturels et `n` ne peut pas être nul dans les deux lignes ci-dessous :

```
FORALL (x, y, z : nat) : x*(y+z) = x*y + x*z
FORALL (n : real | n /= 0) : n/n = 1
```

Les types utilisés peuvent aussi être des types dépendant de variables définies auparavant comme :

```
FORALL (x : int), (y : {y : int | x < y}) : P(x, y)
```

On utilise la forme abrégée du type :

```
FORALL (x : int), (y : int | x < y) : P(x, y)
```

Expressions conditionnelles

En PVS il y a deux types d'expressions conditionnelles : la conditionnelle `IF-THEN-ELSE` et la conditionnelle `COND`.

L'expression `IF-THEN-ELSE` contient, au moins, trois expressions :

- une expression booléenne `b`, qu'on appelle la condition ;
- une expression `e1` qui est le résultat de l'expression `IF-THEN-ELSE` si l'expression `b` est vraie ;
- une expression `e2` qui est le résultat de l'expression `IF-THEN-ELSE` si l'expression `b` est fausse.

La forme de l'expression `IF-THEN-ELSE` est :

```
IF b THEN e1 ELSE e2 ENDIF
```

Pour que PVS puisse vérifier le type du résultat, il faut que les deux branches de la conditionnelle fournissent un résultat, et qui plus est du même type. Autrement dit, la partie `ELSE` n'est pas optionnelle et les types des expressions `e1` et `e2` doivent être identiques. Le type du résultat de l'expression `IF-THEN-ELSE` est le type commun à `e1` et `e2`.

Une expression `IF-THEN-ELSE` peut avoir des branchements multiples, en utilisant la partie `ELSIF` :

```
IF b_1 THEN e_1
ELSIF b_2 THEN e_2
...
ELSIF b_k THEN e_k
ELSE e
ENDIF
```

Elle est traduite en des expressions `IF-THEN-ELSE` imbriquées :

```
IF b_1 THEN e_1
ELSE (IF b_2 THEN e_2
      ELSE (...
              ELSE (IF b_k THEN e_k
                    ELSE e
                    ENDIF)
```

```
                ENDIF
             )
       ENDIF
      )
ENDIF
```

L'autre possibilité pour exprimer une conditionnelle est d'utiliser `COND`, qui est de la forme :

```
COND
     b_1 -> e_1
     b_2 -> e_2
     ...
     b_k -> e_k
ENDCOND
```

où les expressions `b_i`, `i : 1..k` sont des expressions booléennes, et les expressions `e_j`, `j : 1..k` sont des expressions de types compatibles.

Deux conditions régissent l'utilisation de la conditionnelle COND :

- la première, appelée condition de *disjonction*, requiert qu'au plus une des conditions `b_i` soit applicable ;
- la seconde, appelée condition de *couverture*, exige qu'au moins une des conditions `b_i` soit applicable.

Des obligations de preuves sont émises pour garantir ces conditions.

Si les obligations de preuves sont prouvées, l'expression `COND` est équivalente à l'expression `IF-THEN-ELSE` suivante :

```
IF b_1 THEN e_1
ELSIF b_2 THEN e_2
...
ELSIF b_(k-1) THEN e_(k-1)
ELSE e_k
ENDIF
```

La conditionnelle `COND` peut inclure une clause `ELSE` :

```
COND
     b_1 -> e_1
     b_2 -> e_2
     ...
     b_k -> e_k
     ELSE -> e
ENDCOND
```

Cette forme de conditionnelle ne demande pas la preuve de la condition de couverture.

En PVS, les expressions conditionnelles peuvent être construites sous forme de tables. Les tables sont équivalentes à des expressions COND. Par exemple :

```
TABLE %-------------------------%
      |[ n < 0 | n = 0 | n > 0 ]|
      %-------------------------%
      |   -1   |   0   |   1   ||
      %-------------------------%
ENDTABLE
```

Remarquez que la ligne de valeurs se termine par `||` et que toutes les autres lignes sont en commentaire, elles servent uniquement à améliorer la lisibilité.

Si la table est unidimensionnelle, comme dans le cas précédent, elle peut être écrite en vertical :

```
TABLE %----------------%
      |[ n < 0 |  -1  ]|
      %----------------%
      |  n = 0 |   0  ||
      %----------------%
      |  n > 0 |   1  ||
      %----------------%
ENDTABLE
```

Les tables peuvent avoir des formes plus complexes, comme la table bidimensionnelle suivante :

```
TABLE          %-------------------------%
               |[ x < 0 | x = 0 | x > 0 ]|
      %-----------------------------------%
      |  y < 0 |   -2   |  -1   |   2   ||
      %-----------------------------------%
      |  y = 0 |   -1   |   0   |   1   ||
      %-----------------------------------%
      |  y > 0 |   -3   |   1   |   3   ||
      %-----------------------------------%
ENDTABLE
```

Application

L'application d'une fonction `f` à un argument `x` se fait en écrivant `f(x)`. Pour les opérateurs binaires, on utilise une écriture infixe comme `x + y` ou préfixe comme `+(x, y)`. Les types de PVS forment un système d'ordre supérieur, c'est-à-dire que le résultat ou un paramètre d'une fonction peut aussi être une fonction. Par exemple, la fonction `g` avec le type `[nat->[nat->nat]]`, a comme paramètre un entier naturel et comme résultat une fonction qui a comme paramètre et comme résultat un entier naturel. La fonction `g` peut être évaluée, par exemple on peut écrire `g(0)` dont le résultat est une

fonction du type `[nat->nat]`. On peut évaluer `g` comme `g(0)(2)`, la valeur `2` est appliquée à la fonction résultat de `g(0)`.

Une fonction peut avoir un n-uplet comme paramètre. Par exemple la fonction `h` avec le type `[int, int -> int]` a comme paramètre une paire de type `[int, int]`. On peut appeler `h` avec la paire `(-2, 3)`, c'est-à-dire `h(-2, 3)`.

Dans les fonctions qui ont des types dépendant de variables définies auparavant, le type est substitué conformément aux paramètres donnés. Soit par exemple, la fonction `f` avec le type

```
[n : int, (m : int | n < m) -> { s : int | n < s /\ s <= m }]
```

l'application `f(-2, 3)` a pour type `{ s : int | -2 < s /\ s <= 3 }`. Il ne faut pas oublier aussi la TCC qui est ici `-2 < 3`.

λ-abstraction

Avec les λ-abstractions, on peut définir des fonctions anonymes. Par exemple, la fonction zéro qui pour tout réel a comme résultat `0` est

```
LAMBDA (x : real) : 0
```

La λ-expression pour la fonction successeur est

```
LAMBDA (n : nat) : n + 1
```

Le type d'une λ-abstraction

```
LAMBDA (x : T) : e
```

est dérivé à partir du type du paramètre `x` et du type de l'expression `e`. Si `e` est de type `T1` le type de

```
LAMBDA (x : T) : e
```

est

```
[T -> T1]
```

Les λ-expressions peuvent être utilisées là où une fonction de même type serait utilisée.

Expressions d'actualisation

En PVS il est possible de créer une fonction qui est « presque la même » qu'une autre fonction existante. Plus précisément, la nouvelle fonction retourne les mêmes résultats que la fonction originale, sauf pour certains arguments spécifiés. Par exemple la fonction

```
succ( n : nat ) : nat = n + 1
```

peut être utilisée pour créer une fonction presque identique, qui retourne `0` pour l'argument `0` et `1` pour l'argument `1` :

```
succ WITH [(0) := 0, (1) := 1]
```

La dernière expression est équivalente à l'expression :

```
(succ WITH [(0) := 0]) WITH [(1) := 1]
```

et aussi à l'expression

```
(LAMBDA n : IF n = 1 THEN 1
            ELSIF n = 0 THEN 0
            ELSE succ(n) ENDIF)
```

Il est également possible d'étendre le domaine de la fonction en utilisant `|->` au lieu de `:=`. Par exemple :

```
succ WITH [(-5) |-> 0]
```

La nouvelle fonction est de type

```
[{ i : int | i >=0 OR i = -5} -> nat]
```

et est équivalente à la fonction

```
(LAMBDA (i : { i : int | i >= 0 OR i = -5}) :
            IF i = -5 THEN 0
            ELSE succ(i) ENDIF)
```

Les actualisations peuvent aussi être appliquées aux n-uplets et aux enregistrements. Par exemple :

```
(1, true) WITH [‘1 := 0]
```

est la paire `(0, true)` et l'expression

```
(# a := 1, b := true #) WITH [‘b := false]
```

est l'enregistrement

```
(# a := 1, b := false #)
```

On peut aussi étendre le domaine des n-uplets et des enregistrements, en augmentant leur nombre d'éléments, de la façon suivante :

```
(1, true) WITH [‘3 |-> 2]
```

est le triplet `(1, true, 2)`, et

```
(# a := 1, b := true #) WITH [‘c |-> 2]
```

est l'enregistrement

```
(# a := 1, b := true, c := 2 #)
```

Dans les n-uplets on ne peut pas étendre le domaine en laissant des éléments non spécifiés. L'expression ci-dessous est illégale :

```
(1, true) WITH [‘4 |-> 2]
```

Définitions locales

Les définitions locales en PVS se construisent à l'aide des expressions `LET` et `WHERE`. Dans l'expression `LET` on définit des sous-expressions que l'on affecte à des variables locales qui sont utilisées dans l'expression suivant la clause `IN`. Par exemple :

```
LET a : int = -7,
    b : int = a + 3
 IN 5*a*b + 3*(a - b) - a*a - b*b
```

L'expression `LET` est traduite par la λ-expression :

```
(LAMBDA (a : int) : (LAMBDA (b : int) :
        5*a*b + 3*(a - b) - a*a - b*b)(a + 3))(-7)
```

L'expression `WHERE` est similaire à l'expression `LET` mais les variables locales sont écrites à la suite de l'expression principale. L'expression `WHERE` ci-dessous est équivalente à l'expression `LET` précédente :

```
5*a*b + 3*(a - b) - a*a - b*b
WHERE a : int = -7,
      b : int = a + 3
```

L'expression `LET` peut être utilisée pour obtenir les composants d'un n-uplet. Par exemple, si `t` est un triplet d'entiers, l'expression ci-dessous additionne ses composants :

```
LET (a, b, c) = t
 IN a + b + c
```

Ensembles

Un ensemble d'éléments de type `T` en PVS est représenté par un prédicat, c'est-à-dire par une fonction de `T` dans `bool` (`[T -> bool]`). Pour définir un ensemble, on peut utiliser une fonction anonyme de la forme

```
(LAMBDA (x : T) : p(x))
```

ou une expression d'ensemble

```
{ x : T | p(x) }
```

Les deux expressions sont équivalentes. La dernière expression peut être confondue avec une expression de type, mais le contexte permet de les distinguer.

La conditionnelle par cas

L'expression `CASES` est utilisée pour faire un filtrage selon les différentes valeurs possibles d'un type abstrait de données. Par exemple

```
CASES lst OF
    null : 0,
    cons(x, xs) : 1 + (length xs)
ENDCASES
```

Si la liste `lst` est la valeur `null` la conditionnelle retourne la valeur `0` et si `lst` est de la forme `cons(x, xs)` le résultat est la valeur `1 + (length xs)`, autrement dit on calcule la longueur de la liste `lst`.

La conditionnelle `CASES` peut inclure une clause `ELSE` :

```
CASES lst OF
    cons(x, xs) : 1 + (length xs)
    ELSE 0
ENDCASES
```

Si la clause `ELSE` est absente et si on n'utilise pas tous les constructeurs dans le filtrage, des obligations de preuve sont émises pour démontrer que les cas exclus ne sont pas possibles.

2.1.6 Déclaration de fonctions

Comme PVS est un langage fondé sur la logique d'ordre supérieur, les déclarations de fonctions sont considérées comme des déclarations de constantes. Autrement dit, en PVS une fonction est une constante qui a comme type un type de fonction. Les déclarations de fonctions peuvent être interprétées et non interprétées.

Voici quelques exemples de fonctions non interprétées :

```
sq(x : real) : real
gcd : [nat, nat -> nat]
```

Remarquez que les trois définitions ci-dessous sont équivalentes :

```
max : [int, int -> int]
max(m : int, n : int) : int
max(m, n : int) : int
```

On dit qu'une fonction est interprétée lorsque l'on précise de plus le « corps » de la fonction, comme dans les exemples ci-dessous :

```
sq(x : real) : real = x*x
max(m : int, n : int) : int = IF m >= n THEN m ELSE n ENDIF
abs(x : real) : posreal = IF x < 0 THEN -x ELSE x ENDIF
```

Fonctions par récurrence

En PVS on peut définir des fonctions récursives, mais pas des fonctions mutuellement récursives, c'est-à-dire que l'on ne peut pas définir une paire de fonctions `f` et `g` telles que `f` appelle `g` et `g` appelle `f`.

Pour définir une fonction récursive, on utilise le mot clé `RECURSIVE`. Les fonctions récursives doivent être totales. Pour satisfaire la condition de totalité, les fonctions récursives doivent fournir une fonction de mesure et optionnellement une relation d'ordre bien fondé pour garantir que la fonction récursive finit toujours.

Afin de prouver que la fonction récursive finit, on démontre que la mesure des arguments de chaque appel récursif décroît strictement, selon l'ordre donné.

Pour cela, il faut que les arguments de la fonction de mesure correspondent aux arguments de la fonction récursive, c'est-à-dire qu'ils aient le même nom et le même type. Il faut également que la relation d'ordre agisse sur les résultats de la fonction de mesure, autrement dit qu'elle soit définie sur un ensemble contenant l'image de la fonction de mesure. De plus, pour pouvoir montrer la terminaison, on demande à ce que la relation d'ordre soit bien fondée, c'est-à-dire que tout ensemble admette un plus petit élément (qui sera le cas final de la récursion). Une relation d'ordre bien fondé R sur un ensemble E non vide est telle que pour toute partie X de E, il existe un élément $x \in X$ pour lequel il n'y a aucun élément $y \in X$ vérifiant yRx. Si la relation d'ordre n'est pas donnée, l'ordre par défaut est "$<$" sur `nat`. Lors d'un appel à une fonction récursive, des obligations de preuve sont émises pour démontrer que la mesure de l'appel par récurrence est en relation avec la mesure des arguments de la fonction, c'est-à-dire que la mesure de chaque appel récursif décroît selon la relation d'ordre.

Par exemple, la déclaration de la fonction `factorial` est la suivante :

```
factorial(n : nat): RECURSIVE nat =
  IF n = 0 THEN 1 ELSE n * factorial(n - 1) ENDIF
MEASURE (LAMBDA (n : nat) : n)
```

Pour cette fonction, l'obligation de preuve ci-dessous est émise :

```
factorial_TCC2 : OBLIGATION
  OBLIGATION FORALL (n : nat) : NOT n = 0 IMPLIES n - 1 < n;
```

On peut abréger la fonction mesure en donnant l'expression qui définit la fonction, par exemple, on peut abréger la fonction `measure` de la fonction `factorial` par

```
MEASURE n
```

PVS introduit automatiquement la λ-abstraction correspondante.

2.1.7 Déclaration de formules

En PVS les formules sont utilisées pour introduire des formules logiques dans la théorie. Une formule commence par le nom de la formule (un identificateur), suivi par

deux points et le type de la formule, puis par une formule logique (une expression booléenne). Les types possibles pour les formules sont *axiome*, *hypothèse*, *obligation* et *théorème*. Les axiomes sont introduits par le mot clé `AXIOM` ou, de façon équivalente, par le mot clé `POSTULATE`. Les hypothèses sont introduites par le mot clé `ASSUMPTION` et doivent être déclarées dans la section des hypothèses (voir §2.1.1). Les obligations sont introduites par le mot clé `OBLIGATION` et elles sont émises automatiquement par le système pour les obligations de preuves ; les obligations ne peuvent pas être déclarées par l'utilisateur. Les théorèmes peuvent être introduits par l'un des mots clés suivants, tous étant équivalents : `CHALLENGE`, `CLAIM`, `CONJECTURE`, `COROLLARY`, `FACT`, `FORMULA`, `LAW`, `LEMMA`, `PROPOSITION`, `SUBLEMMA` ou `THEOREM`.

Les identificateurs associés à une déclaration de formule peuvent être utilisés dans les preuves. Quelques exemples de déclarations de formules sont donnés ci-dessous :

```
associative_mult : AXIOM x * (y * z) =  (x * y) * z
not_exists : LEMMA (EXISTS x : p(x)) = NOT (FORALL x : NOT p(x))
```

Pour les variables qui sont libres dans la formule (une variable est libre dans une formule si elle n'est pas déclarée dans un quantificateur ou une λ-abstraction), PVS assume la fermeture universelle, c'est-à-dire qu'il les quantifie avec le quantificateur universel (`FORALL`). C'est le cas des variables `x`, `y` et `z` dans l'axiome `associative_mult` donné en exemple.

Pour chaque formule déclarée en PVS, le système mémorise son état de preuve. L'état de preuve peut être l'un des états suivants :

untried qui signifie qu'aucune étape de preuve n'a été effectuée pour cette formule ;

proved qui signifie que la formule a été démontrée ;

unchecked qui signifie qu'il y a une preuve, mais la spécification a été modifiée après que la preuve a été faite ;

unfinished qui signifie qu'une preuve est en cours mais qu'elle n'est pas terminée.

Pour les formules qui ont un état de preuve *proved*, l'état peut de plus être *complet* ou *incomplet*. L'état d'une preuve est *complet* seulement dans le cas où toutes les formules (incluant les TCCs) dont la preuve dépend ont été complètement prouvées, c'est-à-dire ont l'état *complet*. On dit alors que la formule a été démontrée de manière satisfaisante pour le système. Dans les autres cas, la preuve est *incomplète* et il manque souvent la démonstration des TCCs.

2.1.8 Jugements

Le système de types peut émettre plusieurs obligations de preuves selon le type des expressions. Les jugements permettent de donner des assertions sur les types et sous–types, qui sont utilisées automatiquement lors de la vérification des types. Les jugements permettent de limiter le nombre d'obligations de preuves. Il existe deux sortes de jugements : les jugements sur les constantes et les jugements sur les sous-types.

Jugements sur les constantes

Un jugement sur une constante permet de lui donner un type plus strict que celui avec lequel elle a été déclarée, par exemple :

```
JUDGEMENT 5 HAS_TYPE (prime?)
```

établit que 5 est un nombre premier.

Les jugements sur les constantes sont plus intéressants quand on les applique aux fonctions. Ils permettent d'établir que, quand la fonction est appliquée aux arguments d'un type donné, le résultat a le type spécifié après le mot clé `HAS_TYPE`. Supposons que `e1` et `e2` sont des variables de type `even_int`, `o1` et `o2` sont des variables de type `odd_int`, les jugements

```
even_plus_even_is_even :  JUDGEMENT +(e1, e2) HAS_TYPE even_int
odd_plus_odd_is_even :    JUDGEMENT +(o1, o2) HAS_TYPE even_int
odd_plus_even_is_odd :    JUDGEMENT +(o1, e2) HAS_TYPE odd_int
even_plus_odd_is_odd :    JUDGEMENT +(e1, o2) HAS_TYPE odd_int
```

établissent les types appropriés pour l'addition des entiers pairs et impairs.

Jugements sur les sous-types

Les jugements sur les sous-types permettent d'établir des relations entre les types et les sous-types. Par exemple, si on a les déclarations

```
nonzero_real : NONEMPTY_TYPE = {x : real | x /= 0} CONTAINING 1
rational : NONEMPTY_TYPE FROM real
nonneg_rat : NONEMPTY_TYPE = {x : rational | x >= 0} CONTAINING 0
posrat :     NONEMPTY_TYPE = {x : nonneg_rat | x > 0}  CONTAINING 1
/ : [real, nonzero_real -> real]
```

Pour `p` de type `real` et `q` de type `posrat`, l'expression `p/q` émet l'obligation de preuve `q /= 0`. Le système de types ne sait pas qu'un élément de type `posrat` est un élément de type `nonzero_real`. Pour résoudre ce problème on déclare le jugement

```
JUDGEMENT posrat SUBTYPE_OF nonzero_real
```

L'intérêt d'utiliser des jugements est que le système de types émet moins d'obligations de preuves.

2.1.9 Résolution de noms

Une théorie est un ensemble de définitions et de théorèmes sur les définitions. Pour construire de nouvelles théories, on peut se baser sur des théories définies au préalable.

Quand on a importé plusieurs théories, il peut y avoir des ambiguïtés de noms parce que :

- le même nom est défini dans différentes théories importées ;
- le même nom est importé par différentes instances de la même théorie, c'est-à-dire par la même théorie avec des paramètres différents.

Il y a trois façons de faire référence à un nom en PVS :

- `nom`;
- `nom[paramètres]`;
- `théorie[paramètres].nom`.

Quand il n'y a pas d'ambiguïté, la première méthode marche bien. La deuxième méthode est utile pour lever certaines ambiguïtés, par exemple quand le même nom est importé par différentes instances de la même théorie, les paramètres permettant alors de distinguer l'instance à laquelle ils appartiennent. La troisième méthode est toujours non ambiguë. Comme nous l'avons dit auparavant, les coercitions peuvent aussi être utilisées pour résoudre des ambiguïtés provoquées par la surcharge de noms.

2.1.10 Conclusion

Dans cette section, nous avons présenté les différents éléments du langage de spécification PVS. Nous avons présenté les théories et nous avons décrit leurs principaux composants. Nous avons détaillé les différents types du langage, tels que les types définis par des prédicats, les types enregistrement, les types abstraits de données, etc. Nous avons expliqué les particularités des fonctions récursives. Nous avons également développé certaines particularités de PVS, telles que les expressions d'actualisation ou les différentes expressions conditionnelles. Enfin, nous avons présenté la déclaration des formules et les jugements.

2.2 L'assistant de preuves

Dans cette section, nous présentons l'assistant de preuves PVS. Il s'agit ici d'introduire quelques concepts et commandes que nous allons utiliser dans les chapitres suivants. Le lecteur intéressé trouvera des informations complémentaires sur l'assistant de preuves et ses commandes dans [47].

2.2.1 Preuves et séquents

Dans l'assistant de preuves de PVS, chaque preuve est représentée par un arbre. Chaque nœud de l'arbre est un *sous-but de la preuve*. Un sous-but est traité lorsqu'une

commande de preuve est invoquée et exécutée. Une commande de preuve ajoute des sous-arbres à l'arbre de preuve. La preuve est complète si chacune des feuilles de l'arbre est reconnue par PVS comme vraie. Chaque *sous-but de preuve* est un *séquent* qui est constitué d'une liste de formules dénommées *antécédents* et d'une liste de formules dénommées *conséquences*. En PVS un séquent est de la forme :

```
{-1}  A1
{-2}  A2
       ⋮
     |-----
{1}   C1
[2]   C2
       ⋮
```

où les A_i sont les *antécédents* et les C_j sont les *conséquences*. Il est équivalent d'avoir la formule $\neg F$ parmi les antécédents ou la formule F parmi les conséquences. Pour économiser de la place, on écrira un tel séquent dans le texte sous la forme $A_1, A_2, \ldots \vdash C_1, C_2, \ldots$. On l'interprète ainsi : la conjonction des antécédents implique la disjonction des conséquences, c'est-à-dire que $A_1 \wedge A_2 \wedge \cdots \implies C_1 \vee C_2 \vee \cdots$ et l'objectif est de démontrer que cette implication est vraie. Autrement dit, on considère les antécédents comme les hypothèses du séquent et les conséquences comme les conclusions que l'on désire établir.

Précisons maintenant la syntaxe de cette écriture. Le symbole `|-----` est utilisé pour séparer les deux listes. La liste des antécédents est numérotée par des nombres négatifs alors que la liste des conséquences est numérotée par des nombres positifs. Les numéros sont entourés par des crochets ou par des accolades. Quand le numéro est entouré par des crochets (comme `[2]`), cela signifie que la formule reste la même que dans le but parent, alors que si ce numéro est entouré par des accolades (comme `{1}`) cela signifie que la formule est nouvelle ou modifiée par rapport au but parent.

La racine de l'arbre de preuve est un but de la forme $\vdash C$ où C est le théorème à démontrer. Un séquent est reconnu comme vrai soit si un antécédent est faux (`false`), soit si une conséquence est vraie (`true`), soit encore si la même formule apparaît comme antécédent et comme conséquence. Quand un séquent est reconnu comme vrai, la branche correspondante de l'arbre de preuve est terminée. L'objectif est de construire un arbre de preuve dont toutes les branches se terminent par un séquent reconnu comme vrai.

2.2.2 L'assistant de preuves

L'interface de l'assistant de preuves de PVS est *emacs* [46]. Pour appeler l'assistant de preuves, on utilise la commande emacs `M-x prove` quand le curseur est sur le texte du théorème à démontrer. Les preuves en PVS sont construites de façon interactive. À chaque étape de la preuve un seul séquent est traité et il est appelé *séquent*

```
Rule? (help flatten)
(flatten/$ &rest fnums) :
    Disjunctively simplifies chosen formulas.  It simplifies
top-level antecedent conjunctions, equivalences, and negations, and
succedent disjunctions, implications, and negations from the sequent.
```

FIG. 2.1 – Exemple d'utilisation de la commande `help`.

courant ou *but courant*. Tant que la preuve n'est pas terminée, l'assistant de preuve demande une commande avec l'invite `Rule?`. L'utilisateur peut changer le séquent courant par un autre séquent qui n'est pas démontré (s'il en existe un) grâce à la commande `postpone`. Les commandes de preuve sont écrites entre parenthèses, comme par exemple `(postpone)`. La commande emacs `M-x siblings` montre les séquents dont la preuve n'est pas terminée dans un autre espace tampon d'emacs.

Il est possible de revenir en arrière et de défaire le résultat de l'application d'un ou plusieurs pas de preuve en utilisant la commande `undo`. La commande `undo` prend comme argument optionnel le nombre de pas à défaire, par exemple `(undo 3)` défait le résultat de l'application des trois dernières commandes sur la branche courante. Si elle est utilisée sans argument `(undo)`, elle annule l'effet de la dernière commande uniquement. La commande `(undo undo)` est utilisée pour exécuter à nouveau la dernière commande, qui a été annulée. Pour sortir de l'assistant de preuves sans terminer la preuve, on utilise la commande `(quit)`. Enfin, la commande `help` est utilisée pour demander de l'aide sur une commande. Sur la figure 2.2.2 on montre un exemple de son utilisation.

2.2.3 Commandes de preuve

Toutes les commandes PVS préservent la correction de la preuve. Une commande PVS peut être une règle ou une stratégie. Une règle est considérée comme une opération atomique qui crée zéro ou plusieurs séquents. L'application d'une stratégie consiste à effectuer plusieurs pas atomiques.

Les règles sont primitives ou définies. Une règle définie est un ensemble de règles primitives, mais elle est considérée comme un pas atomique, c'est-à-dire que PVS l'affiche comme un seul pas de la preuve et ne montre jamais le détail des étapes effectuées. Si on est intéressé par le détail de la preuve, ou si on veut pouvoir appliquer récursivement certains pas, on définit une stratégie : la différence entre une règle et une stratégie est floue en PVS mais elle semble résider d'une part dans la possibilité ou non d'obtenir une trace des pas effectués (boîte blanche dans le cas d'une stratégie ou boîte noire dans celui d'une règle) et d'autre part dans la complexité de la méthode. Beaucoup de commandes sont des règles définies, mais elles disposent aussi d'une version qui permet de les utiliser comme stratégie. Par exemple, la commande `prop`, qui simplifie un séquent

en utilisant la logique propositionnelle, est atomique ; sa version comme stratégie est `prop$`.

Pour illustrer la différence entre l'utilisation de la règle `prop` et de la stratégie `prop$`, nous utilisons la règle `prop` pour prouver le séquent :

```
simple_thm :

  |-------
{1}   NOT (p => q) OR (p => q)

Rule? (prop)
```

nous avons que la preuve est :

```
("" (prop))
```

alors que si nous utilisons la stratégie `prop$` pour le même séquent :

```
simple_thm :

  |-------
{1}   NOT (p => q) OR (p => q)

Rule? (prop$)
```

nous obtenons le détail des étapes de la preuve, à savoir :

```
("" (flatten) (split) (("1" (propax)) ("2" (propax))))
```

La stratégie a utilisé les règles `flatten`, `split` et `propax` pour prouver le séquent, ces règles sont détaillées ci-dessous. Pour voir le détail de la preuve d'un théorème, nous utilisons la commande emacs `M-x show-proof`.

Dans ce qui suit, nous allons expliquer quelques commandes de PVS que nous utilisons dans le document. Pour obtenir la liste complète des commandes de l'assistant de preuves PVS ou une explication plus détaillée de son utilisation, nous renvoyons le lecteur à [47].

Toutes les transformations détaillées ci-dessous peuvent facilement être vérifiées en utilisant le fait qu'un séquent $A_1, A_2, \ldots A_m \vdash C_1, C_2, \ldots C_n$ peut aussi s'écrire $\neg(A_1 \wedge A_2 \wedge \ldots \wedge A_m) \vee C_1 \vee C_2 \vee \ldots \vee C_n$, ou encore $(\neg A_1) \vee (\neg A_2) \vee \ldots \vee (\neg A_m) \vee C_1 \vee C_2 \vee \ldots \vee C_n$.

Flatten

La commande `flatten` est utilisée pour appliquer la simplification disjonctive au séquent courant. Une formule du séquent est disjonctive soit si elle est une formule des antécédents de la forme $\neg A$, $A \wedge B$ ou $A \iff B$, soit si elle est une formule des conséquences de la forme $\neg A$, $A \implies B$ ou $A \vee B$. La simplification disjonctive transforme un séquent dans lequel apparaissent des formules disjonctives en un séquent sans formules disjonctives. Les formules sont transformées en appliquant les transformations suivantes, jusqu'à ce qu'il n'y ait plus de formules disjonctives dans le séquent :

- un antécédent $\neg A$ est transformé en une conséquence A ;
- un antécédent $A \wedge B$ est transformé en deux antécédents : A et B ;
- un antécédent $A \iff B$ est transformé en deux antécédents : $A \implies B$ et $B \implies A$;
- une conséquence $\neg A$ est transformée en un antécédent A ;
- une conséquence $A \implies B$ est transformée en un antécédent A et en une conséquence B ;
- une conséquence $A \vee B$ est transformée en deux conséquences : A et B.

Par exemple, si nous appliquons la commande `flatten` au séquent de l'exemple précédent :

```
simple_thm :

  |-------
{1}   NOT (p => q) OR (p => q)

Rule? (flatten)
```

nous obtenons le séquent :

```
simple_thm :

{-1}  (p => q)
{-2}  p
  |-------
{1}   q

Rule?
```

Split

La commande `split` est utilisée pour diviser, ou simplifier, une formule conjonctive. Elle prend comme argument optionnel le numéro de la formule à laquelle elle s'applique. Si cet argument n'est pas spécifié, elle s'applique à la première formule conjonctive qu'elle trouve dans le séquent. Nous détaillons maintenant les cas où la formule est conjonctive et la façon dont la commande la divise en plusieurs séquents, chacun étant

le fils du séquent dont on est parti et chacun devant à son tour être prouvé pour que le séquent originel soit vrai :

- un antécédent $A_1 \vee A_2 \vee \cdots \vee A_n$ est divisé en n séquents ayant chacun l'un des A_i, $1 \leq i \leq n$ comme antécédent ;
- un antécédent $A \Longrightarrow B$ est divisé en deux séquents, l'un avec B comme antécédent et l'autre avec A comme conséquence ;
- une conséquence $A \wedge B$ est divisée en deux séquents, l'un avec A et l'autre avec B comme conséquence ;
- une conséquence $A \Longleftrightarrow B$ est divisée en deux séquents, l'un avec $A \Longrightarrow B$ et l'autre avec $B \Longrightarrow A$ comme conséquence ;
- un antécédent `IF` A `THEN` B `ELSE` C `ENDIF` est divisé en deux séquents, l'un avec $A \wedge B$ et l'autre avec $\neg A \wedge C$ comme conséquence ;
- une conséquence `IF` A `THEN` B `ELSE` C `ENDIF` est divisée en deux séquents, l'un avec $A \Longrightarrow B$ et l'autre avec $\neg A \Longrightarrow C$ comme conséquence.

Par exemple si nous appliquons la commande `split` au séquent de l'exemple précédent

```
simple_thm :

{-1}  (p => q)
{-2}  p
  |-------
{1}   q

Rule? (split)
```

nous obtenons deux séquents, l'un étant

```
simple_thm.1 :

{-1}  q
[-2]  p
  |-------
[1]   q
```

et l'autre

```
simple_thm.2 :

[-1]  p
  |-------
{1}   p
[2]   q
```

Propax

Pour appliquer les axiomes de la logique propositionnelle, nous utilisons la commande `propax` : un séquent est vrai s'il contient un antécédent `FALSE`, une conséquence `TRUE`, une conséquence `t=t`, ou s'il s'agit d'un séquent avec un antécédent `A` et une conséquence `B` où `A` est syntaxiquement égal à `B`, c'est-à-dire que A et B sont égaux, sauf pour ce qui concerne les noms de variables liées par un quantificateur ou une lambda-abstraction. La commande `propax` est appliquée automatiquement à chaque séquent de la preuve. Par exemple, cette commande est appliquée pour terminer la preuve des séquents résultant de l'application de la commande `split` dans l'exemple précédent, puisque dans chacun des séquents précédents, l'un des antécédents est égal à l'une des conséquences.

Bddsimp

La commande `bddsimp` simplifie le séquent en utilisant la logique propositionnelle. Elle utilise un paquet externe écrit en langage C qui se base sur les arbres binaires de décision (BDD ou *binary decision diagrams* en anglais).

Label

La commande `label` est utilisée pour marquer, ou nommer, des formules dans le séquent courant. Elle prend comme paramètres une chaîne de caractères et la liste des numéros des formules à marquer. Les marques sont héritées par les sous-formules qui apparaissent par l'application d'une règle qui modifie la formule, comme par exemple la commande `flatten` ou la commande `split`. Les marques peuvent être utilisés lorsqu'un numéro de formule est attendu.

Hide

La commande `hide` est utilisée pour cacher des formules dans le séquent courant, c'est-à-dire pour que les règles ne leur soient pas appliquées. Elle prend comme paramètre la liste des numéros de formules à cacher. Des exemples de son utilisation sont `(hide -1)` qui cache la formule avec le numéro `-1` et `(hide (-1 -3 2))` qui cache les formules `-1`, `-3` et `2`. Pour voir les formules cachées dans le séquent, on utilise la commande emacs `M-x show-hidden-formulas`. Pour rendre à nouveau visible une formule cachée, on utilise la commande `reveal` avec la liste des numéros de formules à révéler. Les numéros de formules qui sont utilisées par la commande `reveal` sont les numéros qui apparaissent par l'utilisation de la commande emacs `M-x show-hidden-formulas`.

Skip-msg

La commande `skip-msg` n'a aucun effet sur la preuve, mais elle prend en paramètre une chaîne de caractères qui est affichée à l'écran. Son utilisation principale est pour l'écriture de stratégies et plus particulièrement dans les cas d'échec ou d'erreur, où la seule chose à faire est d'informer l'utilisateur de cette erreur.

Skip

La commande `skip` ressemble à la commande `skip-msg`, elle n'a aucun effet dans la preuve et de plus elle est muette. Tout comme la commande `skip-msg`, elle est utilisée pour l'écriture de stratégies, dans les cas où aucune étape ne doit être effectuée à moins qu'une condition donnée ne soit vérifiée.

Expand

La commande `expand` prend comme paramètre une chaîne de caractères qui est le nom de la déclaration qu'elle doit remplacer par sa définition. Parmi ses paramètres optionnels, on trouve le numéro de la formule à laquelle doit s'appliquer la commande et l'occurrence à remplacer dans la formule.

Case

La commande `case` prend comme paramètres les formules à décomposer en sous-cas, écrites entre guillemets, par exemple `"x > 0"`. Dans ce cas, deux sous-buts sont créés, l'un ayant pour hypothèse (ou antécédent) supplémentaire que `x` est strictement positif et l'autre qui demande que l'on prouve que `x > 0`, autrement dit il a `x > 0` pour conséquence. Cette commande est très utile pour aider l'assistant de preuve à remplacer une expression `t` par une autre expression `s` telle que `t = s` et telle que l'utilisation de la forme `s` simplifie la preuve, quand l'assistant de preuve n'arrive pas à trouver cette égalité par d'autres moyens.

Si nous avons une formule $\Gamma \vdash \Delta$ où Γ est une liste d'antécédents et Δ est une liste de conséquences, la commande (`case` $A_1 \dots A_n$) crée les $n+1$ sous-buts :

$$\begin{array}{c} A_n \dots A_1, \Gamma \vdash \Delta \\ A_{n-1} \dots A_1, \Gamma \vdash A_n, \Delta \\ A_{n-2} \dots A_1, \Gamma \vdash A_{n-2}, A_n, \Delta \\ \vdots \\ A_1, \Gamma \vdash A_2, \Delta \\ \Gamma \vdash A_1, \Delta \end{array}$$

La commande `case` permet dans un premier temps de supposer les formules $A_1 \dots A_n$ pour prouver les conséquences Δ et par la suite de prouver chacune des formules A_i,

$1 \leq i \leq n$, autrement dit la commande `case` permet d'introduire des hypothèses qui sont vérifiées ultérieurement.

Cette commande permet de traiter les formules dont la preuve s'effectue par cas.

Lemma

La commande `lemma` est utilisée pour introduire une instance d'un lemme. Le nom du lemme est passé comme paramètre. La commande `lemma` prend comme paramètre optionnel la liste des substitutions de la forme $(x_1 t_1 \ldots x_n t_n)$ où x_i, $1 \leq i \leq n$ correspond à une variable du lemme et t_i à l'expression par laquelle la variable doit être substituée dans l'instance du lemme.

Rewrite

Pour utiliser un lemme tout en laissant au système le soin de déterminer quelles substitutions appliquer, nous utilisons la commande `rewrite`. Elle prend comme paramètre le nom du lemme à utiliser. Elle essaie de déterminer automatiquement quelles substitutions appliquer aux variables du lemme, en comparant les formules du séquent et la conclusion du lemme.

Inst

Pour instancier des variables apparaissant avec un quantificateur universel dans un antécédent ou avec un quantificateur existentiel dans une conséquence, nous utilisons la commande `inst`. Elle prend comme paramètres le numéro de la formule et les termes qui correspondent aux variables quantifiées. Pour laisser une variable non instanciée, nous utilisons `"_"` comme terme pour cette variable.

Inst ?

La commande `inst ?`, tout comme la commande `inst`, instancie des variables quantifiées universellement dans un antécédent ou quantifiées existentiellement dans une conséquence. La commande cherche dans les formules des instances appropriées pour ces variables. Les numéros des formules dans lesquelles la commande cherche des valeurs adéquates peuvent être donnés grâce au paramètre optionnel `where`.

Skolem

Pour éliminer un quantificateur universel dans une conséquence ou un quantificateur existentiel dans un antécédent, nous utilisons la commande `skolem`. La commande `skolem` est utilisée pour introduire des constantes qui remplacent des variables quantifiées. Les constantes doivent être de nouveaux identificateurs, c'est-à-dire qui ne sont jamais apparus plus tôt dans la démonstration. La commande prend comme paramètres

le numéro de la formule à laquelle s'applique la commande et la liste des constantes à introduire.

Skosimp

La commande `skosimp` ressemble à la commande `skolem`, à ceci près qu'elle introduit automatiquement les noms de constantes. Après l'introduction des constantes, elle applique la commande `flatten`. Le numéro de la formule à laquelle s'applique la commande est optionnel, par défaut elle s'applique à la première formule ayant des antécédents quantifiés existentiellement ou, s'il n'en existe pas, à la première formule ayant des conséquences quantifiées universellement.

2.2.4 Conclusion

Nous avons présenté dans cette section les concepts de base de l'assistant de preuves, tels que *l'arbre de preuve*, le *but*, le *séquent* et la *commande de preuve*. Nous avons établi qu'un *séquent* est composé d'une liste d'*antécédents* et d'une liste de *conséquences* et à quel moment un séquent est reconnu comme vrai. Nous avons vu que l'interface de l'assistant de preuves est *Emacs* et nous avons indiqué quelques commandes Emacs définies par PVS qui permettent de travailler avec l'assistant de preuves. Dans le reste de la section nous avons présenté quelques commandes de preuves que nous utilisons dans la suite du document.

Chapitre 3

Modèles de Taylor : opérations de base

Dans ce chapitre nous présentons les théories de base pour l'implantation des modèles de Taylor en PVS : suites à support fini et polynômes. Nous présentons la construction des opérations de base (l'addition, l'opposé, la soustraction, le produit par un scalaire et la multiplication) et les opérations inverse et racine carrée de l'arithmétique des modèles de Taylor.

Selon Neumaier [48], les modèles de Taylor ont été inventés par Lanford vers 1980. L'arithmétique de Taylor a été documentée dès 1984 par Eckmann, Koch et Wittwer [49, 50]. Indépendamment, une version un peu différente de l'arithmétique de Taylor avec reste a été popularisée en 1996 sous le nom de modèles de Taylor par Berz, Makino et leur groupe [51, 52, 53, 54].

L'implantation de l'arithmétique des modèles de Taylor que nous avons faite est un peu différente de celle de Makino [42], en particulier pour les opérations de division et racine carrée. De plus, nous avons effectué la démonstration de la propriété d'inclusion pour les modèles de Taylor implantés, travail qui, à notre connaissance, n'a jamais été réalisé par ailleurs.

Un modèle de Taylor est un couple $(p, \boldsymbol{I})$ composé d'un polynôme p de degré fixé N, et d'un intervalle $\boldsymbol{I}$. Ce couple représente l'ensemble des fonctions

$$\{f : \boldsymbol{x} \to \mathbb{R} \mid \forall x \in \boldsymbol{x},\ f(x) - p(x) \in \boldsymbol{I}\}.$$

On choisit usuellement $\boldsymbol{x} = [-1, 1]$.

Pour implanter les modèles de Taylor, on a besoin d'une implantation des polynômes en PVS. Pour l'implantation d'un polynôme nous considérons trois possibilités, à savoir :

1. une liste finie de monômes,
2. une suite finie de coefficients,

3. une série entière à support fini.

Le premier choix correspond à la construction d'un nouveau type inductif à la Coq [16]. Un élément d'un type inductif est construit en appliquant un nombre fini de fois les règles de construction. Le cas le plus simple correspond à deux règles. La règle de base construit les éléments simples (dans notre cas les polynômes constants) et la règle récursive construit les autres éléments (dans notre cas les polynômes de degré n à partir des polynômes de degré $n-1$). Cet ensemble de règles caractérise le type. Les règles qui définissent un type inductif sont appelées constructeurs. Ces types sont bien adaptés à Coq parce que le schéma de preuve par induction fait partie intégrante du calcul des constructions inductives de Coq, ce qui n'est pas le cas en PVS.

Le type défini par une fonction est bien adapté à un système de preuves comme PVS et ses automatismes. Dans le deuxième cas, à savoir l'implantation des polynômes comme suites finies de coefficients, nous avons besoin de définir une fonction qui permet d'évaluer le polynôme avec l'argument x. Il s'agit probablement de l'implantation la plus simple mais la troisième solution est plus ambitieuse et offre des possibilités de prolongement avec d'autres savoir-faire et dans d'autres domaines de recherche de membres du projet Arénaire. Nous avons donc choisi de définir les polynômes comme des séries entières à support fini. Cette définition a l'avantage d'être compatible avec les bibliothèques sur les séries du LaRC de la NASA[1] – en effet, la majorité des bibliothèques développées pour PVS proviennent de la NASA –, ce qui permet d'utiliser les théorèmes définis dans ces bibliothèques, et permet aussi de travailler avec les polynômes en tant que séries entières.

Pour développer les polynômes, on a construit une théorie des suites à support fini qui est présentée ci-dessous.

3.1 Suites à support fini

On dit qu'une suite $a : \mathbb{N} \to \mathbb{R}$ est à support fini N où $N \in \mathbb{N}$, ce que l'on écrit en PVS `finite_support(a, N)`, si pour tout n plus grand que N, ses éléments sont nuls :

$$\forall n > N : a_n = 0. \tag{3.1}$$

Une définition qui se trouve dans la bibliothèque des séries de la NASA est la convergence. On dit que la suite a converge vers l, et on l'écrit en PVS `convergence(a,l)`, si

$$\forall \epsilon : \exists n : \forall i : i \geq n \Rightarrow |a(i) - l| < \epsilon. \tag{3.2}$$

Nous avons prouvé formellement en PVS que si la suite a est à support fini N, la série $S_n = \sum_{k=0}^{n} a_k$ converge vers la somme des N premiers termes :

$$\texttt{finite\_support}(a, N) \Rightarrow \texttt{convergence}(S_n, \sum_{k=0}^{N} a_k). \tag{3.3}$$

[1] http ://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html.

3.1.1 Opérations de base

Nous avons montré en PVS que les opérations de négation, d'addition et de multiplication par un scalaire préservent le caractère de support fini. Soient la suite a à support fini N, la suite b à support fini M, $c \in \mathbb{R}$ alors

- l'opposé d'une suite à support fini est aussi à support fini :
  `finite_support(-a, N);`
- la somme de deux suites à support fini a et b est aussi une suite à support fini L où L est inférieur ou égal au maximum de N et M, $L \leq \max(M, N)$:
  `finite_support(a+b, L);`
- le produit par un scalaire c d'une suite à support fini est aussi une suite à support fini, de même support si c n'est pas nul :
  `finite_support(c * a, N).`

3.1.2 Multiplication

On montre que la multiplication de deux séries définies par des suites à support fini est donnée par le produit de Cauchy :

$$\texttt{finite\_support}(a, N) \wedge \texttt{finite\_support}(b, M) \tag{3.4}$$
$$\Rightarrow (\sum_{k=0}^{N} a_k) \cdot (\sum_{k=0}^{M} b_k) = \sum_{n=0}^{N+M} \sum_{k=0}^{n} a_k \cdot b_{n-k}.$$

Nous avons montré au préalable, par récurrence, que le résultat est aussi à support fini.

3.1.3 Puissance

On définit la puissance n, avec n un entier positif, d'une suite a comme une nouvelle suite. Cette suite est la fonction constante $\lambda k.0$ si a est la suite $\lambda k.0$ et n est strictement positif. Si la puissance n est égale à 0, le résultat est la suite qui a 1 pour première valeur et 0 pour toutes les autres valeurs. Dans tous les autres cas la fonction puissance est la multiplication récursive de la suite a avec la puissance $(n-1)$-ième de a.

```
pow(a : sequence[real], n : nat ) : RECURSIVE sequence[real] =
    IF a = (LAMBDA (k : nat) : 0) AND n > 0
        THEN  (LAMBDA (k : nat) : 0)
      ELSIF n = 0
        THEN (lambda k : IF k = 0 THEN 1 ELSE 0 ENDIF)
      ELSE cauchy(a,  pow(a, n - 1))
    ENDIF
    MEASURE n
```

La puissance d'une suite permettra de définir dans la section suivante la puissance d'un polynôme.

3.2 Polynômes

À l'époque où nous avions besoin de polynômes, il n'y en avait aucune implantation disponible en PVS. En même temps que nous implantions les polynômes, Lester (Manchester, Grande–Bretagne), en développait une version. Nous avons démontré que notre version est équivalente à la version de Lester. En addition à la version de Lester, nous avons la théorie des séries à support fini et également le produit, la puissance et la composition de polynômes.

Comme nous l'avons mentionné précédemment, nous avons défini les polynômes comme des séries entières à support fini, ce qui est, comme nous l'avons déjà expliqué, la définition la plus appropriée pour PVS :

$$\texttt{polynomial}(a, N)(x) = \sum_{k=0}^{N} a_k \cdot x^k. \tag{3.5}$$

```
polynomial(a, N)(x) : real = powerseries(a)(x)(N)
```

Pour les théorèmes sur les polynômes, on demande que les suites utilisées soient à support fini.

3.2.1 Opérations de base

L'addition de deux polynômes définis par les suites a et b est le polynôme de l'addition des suites $a + b$ et le nombre de termes non nuls, tout au plus, est le maximum du nombre de termes des polynômes a et b :

```
add_poly : LEMMA
                finite_support(a, N)
            AND finite_support(b, M)
      IMPLIES  polynomial(a + b, max(N, M))
               = polynomial(a, N) + polynomial(b, M)
```

Le produit par un scalaire c d'un polynôme défini par la suite a est le polynôme de la multiplication de c par la suite a :

```
scal_poly : LEMMA
                finite_support(a, N)
       IMPLIES c*polynomial(a, N) = polynomial(c*a, N)
```

La négation d'un polynôme défini par la suite a est le polynôme de la négation de ses éléments :

```
neg_poly : LEMMA
                finite_support(a, N)
        IMPLIES -polynomial(a, N) = polynomial(-a, N)
```

3.2.2 Multiplication

La multiplication de deux polynômes définis par les suites a et b est le polynôme du produit de Cauchy de a et b et son nombre de termes est la somme des nombres de termes de a et de b :

```
mul_polynomial : LEMMA
                     finite_support(a, N)
                 AND finite_support(b, M)
             IMPLIES polynomial(a, N)*polynomial(b, M)
                     = polynomial(cauchy(a, b), N+M)
```

3.2.3 Puissance

On montre en PVS que la fonction puissance d'un polynôme correspond au polynôme élevé à la n-ième puissance :

```
pow_polynomial : LEMMA
                     finite_support(a, N)
             IMPLIES polynomial(a, N)(x)^n
                     = polynomial(pow(a, n), n*N)(x)
```

3.2.4 Composition

La composition de polynômes est utile pour l'arithmétique sur les modèles de Taylor. Pour construire la suite des coefficients de la composition du polynôme

$$p(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_n \cdot x^n$$

avec le polynôme

$$q(x) = b_0 + b_1 \cdot x + b_2 \cdot x^2 + \cdots + b_m \cdot x^m$$

on étudie les coefficients de chaque terme de la composition

$$p(q(x)) = a_0 + a_1 \cdot q(x) + a_2 \cdot q(x)^2 + \cdots + a_n \cdot q(x)^n.$$

```
comp(a : sequence[real], b : sequence[real], d : nat ) :
    RECURSIVE sequence[real] =
        IF d = 0
           THEN (LAMBDA i :
                     IF i = 0
                         THEN a(0)
                         ELSE 0
                     ENDIF)
           ELSE
               LET c = (LAMBDA i :
                        IF i = d
                            THEN 0
                            ELSE a(i)
                        ENDIF)
               IN a(d) * pow(b, d ) + comp(c, b, d-1)
           ENDIF
    MEASURE d
```

FIG. 3.1 – Définition PVS de la fonction `comp` qui compose deux polynômes.

Soient a la suite des coefficients de p et b la suite des coefficients de q. Les coefficients du polynôme $a_i \cdot q(x)^i$, $i \neq 0$, sont donnés par la suite $a_i * b^i$. Pour $i = 0$, les coefficients sont donnés par la suite

$$(\lambda k : \texttt{if}\ \ k = 0\ \ \texttt{then}\ \ a_0\ \ \texttt{else}\ \ 0\ \ \texttt{endif}).$$

La suite des coefficients de la composition est donnée par la récurrence :

$$\texttt{comp}(a, b, d) = \begin{cases} (\lambda i : \texttt{if}\ \ i = 0\ \ \texttt{then}\ \ a_0\ \ \texttt{else}\ \ 0\ \ \texttt{endif}) & \texttt{si}\ \ d = 0 \\ a_d * (b^d) + \texttt{comp}(a, b, d-1) & \texttt{si}\ \ d \neq 0 \end{cases}$$

Remarquez que la suite des coefficients de $p(q(x))$ est $\texttt{comp}(a, b, n)$.

Pour faciliter les démonstrations par récurrence avec la définition PVS, on fait l'appel récursif avec la suite c au lieu de la suite a. La suite c est la suite a avec le terme a_d nul. La définition PVS pour la fonction `comp` est montrée dans la figure 3.1.

Pour montrer l'arrêt des appels récursifs, on utilise comme mesure le degré de la première suite, qui est un entier naturel dont la valeur diminue de 1 à chaque appel récursif.

On a montré par récurrence sur N que la composition est compatible avec la fonction `comp` :

```
comp_polynomial : LEMMA
```

```
            finite_support(a, N)
        AND finite_support(b, M)
    IMPLIES polynomial(a, N)(polynomial(b, M)(x))
            = polynomial(comp(a, b, N), N*M)(x);
```

3.3 Définition des modèles de Taylor

Un modèle de Taylor d'ordre N d'une fonction f sur un intervalle $\boldsymbol{x}$ est un couple qui est constitué d'un polynôme p de degré N et d'un reste qui est un intervalle $\boldsymbol{I}$. Le polynôme p est le développement de Taylor de la fonction f en l'un des points de $\boldsymbol{x}$, typiquement en 0 si $\boldsymbol{x} = [-1, 1]$. L'intervalle $\boldsymbol{I}$ encadre l'erreur d'approximation $f - p$ sur $\boldsymbol{x}$.

Nous présentons notre implantation des modèles de Taylor en PVS. Il s'agit d'une théorie paramétrée :

```
taylor_model  [ N : posnat, domInterval : Interval ] : THEORY
```

le paramètre `N` est le degré du polynôme du modèle de Taylor et le paramètre `domInterval` est l'intervalle pour la variable du polynôme.

Le type `domIntervalType` est défini pour déclarer des variables qui appartiennent à l'intervalle `domInterval` :

```
domIntervalType : TYPE = inInterval( domInterval )
```

Voici notre définition en PVS du type pour les modèles de Taylor :

```
tm : TYPE = [# P : fs_type, I : Interval #]
```

où `P` est la suite à support fini `N` des coefficients du polynôme du modèle de Taylor et `I` est le reste intervalle du modèle de Taylor. Avec cette définition, f est une fonction représentée par un modèle de Taylor (p, I) si on a pour tout $x \in J :=$ `domInterval`, $f(x) - p(x) \in I$. On note que I borne l'erreur entre $p(x)$ et $f(x)$ sur le domaine étudié J.

Pour évaluer le polynôme p sur l'intervalle $J :=$ `domInterval`, on utilise les deux fonctions ci-dessous qui calculent $\sum_{i=0}^{n} a_i \cdot J^i$ en utilisant cette expression sous forme de somme de monômes :

```
intervalSeqFromRealSeq( a : sequence[real] )
          : [nat -> Interval ] =
   LAMBDA ( n : nat ) :
        IF n = 0 THEN [|a(0)|] ELSE a(n) * domInterval^n ENDIF

intervalFromRealSeq( a : sequence[real], n : nat ) : Interval =
          Sigma(0, n, intervalSeqFromRealSeq(a) )
```

La fonction `trunc` est utilisée pour obtenir les coefficients du polynôme tronqué à l'ordre n

```
trunc(a : sequence[real], n : nat) : sequence[real] =
       restrict(a, 0, n)
```

c'est-à-dire que la suite des coefficients $b :=$ `trunc`(a, n) est la suite $b_i = a_i$ pour $0 \leq i \leq n$ et $b_i = 0$ pour $i > n$.

La fonction `trunc_remainder` est utilisée pour obtenir le reste des coefficients du polynôme tronqué à l'ordre n

```
trunc_remainder(a : sequence[real], n : nat) : sequence[real] =
      a - trunc(a, n)
```

c'est-à-dire que la suite des coefficients $b :=$ `trunc_remainder`(a, n) est la suite $b_i = 0$ pour $0 \leq i \leq n$ et $b_i = a_i$ pour $i > n$.

On a le théorème suivant pour découper un polynôme en une partie tronquée et le reste de la troncature

```
trunc_separation : LEMMA FORALL (bs : fs_type_n(m)) :
      m >= n IMPLIES
             polynomial(bs, m) = polynomial(trunc(bs, n), n)
                    + polynomial(trunc_remainder(bs, n), m)
```

Le type `fs_type_n(m)` indique que `bs` est à support fini `m`. La preuve se fait à partir de l'addition de polynômes et de la définition de `trunc` et `trunc_remainder`.

3.4 Opérations de base

Soient `t` et `u` des modèles de Taylor de polynômes de même degré N, l'addition de deux modèles de Taylor correspond à l'addition de ses polynômes et l'addition de ses intervalles :

```
+( t : tm, u : tm ) : tm = (# P := t`P + u`P, I := t`I + u`I #)
```

L'opposé d'un modèle de Taylor est l'opposé de son polynôme et l'opposé de son intervalle :

```
neg(t) : tm = (# P := - t`P, I := -t`I #)
```

La soustraction de modèles de Taylor `t - u` est l'addition du modèle de Taylor `t` avec la négation du modèle de Taylor `u` :

```
-( t : tm, u : tm ) : tm = t + neg(u)
```

Le produit d'un modèle de Taylor par un scalaire `c` est réalisé par la multiplication de son polynôme par le scalaire `c` et par la multiplication de l'intervalle par le scalaire `c` :

```
*( c, t ) : tm = (# P := c * t‘P, I := c * t‘I #)
```

Soient $t = (p_t, I_t)$ et $u = (p_u, I_u)$ deux modèles de Taylor, f une fonction du modèle de Taylor t et g une fonction du modèle de Taylor u, on a donc $f(x) = p_t(x) + r$ et $g(x) = p_u(x) + s$ pour $r \in I_t$ et $s \in I_u$. La multiplication des modèles de Taylor $t \times u$ représente le produit $f \times g$:

$$\begin{aligned}&(f \times g)(x)\\ =&f(x) \times g(x)\\ =&(p_t + r) \times (p_u + s)\\ =&p_t \times p_u + p_t \times s + p_u \times r + r \times s\end{aligned}$$

La partie intervalle du modèle de Taylor contient donc les termes faisant intervenir les parties intervalles des modèles de Taylor $t = (p_t, I_t)$ et $u = (p_u, I_u)$, à savoir

$$I_1 := p_t(J) \times I_u + p_u(J) \times I_t + I_t \times I_u.$$

Le polynôme $p_t \times p_u$ est de degré $2N$, alors que nos modèles de Taylor n'utilisent que des polynômes de degré au plus N ; il faut donc effectuer un traitement supplémentaire. Pour préciser ce traitement, notons $q = p_t \times p_u = \sum_{i=0}^{2N} q_i x^i$. Pour obtenir la partie polynomiale du modèle de Taylor $t \times u$, on tronque le polynôme q à l'ordre N. Les termes ignorés par cette troncature sont $\sum_{i=N+1}^{2N} q_i x^i$. Ils sont pris en compte dans le reste intervalle de la façon suivante : lorsque la variable x évolue dans l'intervalle J, le polynôme $\sum_{i=N+1}^{2N} q_i x^i$ varie dans l'intervalle $I_2 := \sum_{i=0}^{2N} q_i J^i$.

Pour que l'intervalle du modèle de Taylor $t \times u$ prenne bien en compte tous les termes négligés, on le calcule en additionnant les intervalles I_1 (qui fait intervenir les restes intervalles de t et u) et I_2 (qui prend en compte la troncature du produit des parties polynomiales de t et u).

En PVS, le modèle de Taylor pour la multiplication est :

```
*( t, u )  : tm = (# P := trunc(cauchy(t‘P, u‘P), N), I :=
 intervalFromRealSeq(trunc_remainder(cauchy(t‘P, u‘P), N), 2*N)
           + t‘I * intervalFromRealSeq( u‘P, N )
           + u‘I * intervalFromRealSeq( t‘P, N )
           + t‘I * u‘I
   #)
```

Le prédicat d'inclusion `containment` est défini pour garantir que le modèle de Taylor t est une représentation correcte de la fonction f :

```
tm_add_enclosure : LEMMA
               containment( f, t )
           AND containment( g, u )
       IMPLIES containment( f + g, t + u )

tm_scal_enclosure : LEMMA
               containment( f, t )
       IMPLIES containment( c * f, c * t )

tm_neg_enclosure : LEMMA
               containment( f, t )
       IMPLIES containment( -f, neg(t) )

tm_mult_enclosure : LEMMA
               containment( f, t )
           AND containment( g, u )
       IMPLIES containment( f * g,  t*u )
```

FIG. 3.2 – Propriétés d'inclusion pour les opérations de base.

```
containment( f : [domIntervalType -> real], t : tm ) : bool =
          FORALL xu : (f(xu) - polynomial(t'P, N)(xu))  ## t'I
```

Remarquez que la variable `xu` est de type `domIntervalType`, c'est-à-dire que `xu` $\in J$, $J :=$ `domInterval`.

Nous allons utiliser dans le reste du document indifféremment le terme propriété d'inclusion et propriété containment, même si le terme propriété d'inclusion n'est pas très adapté pour dénoter à la propriété containment des modèles de Taylor.

On a démontré directement à partir des définitions que les opérations de base satisfont la propriété d'inclusion (voir fig. 3.2).

3.5 Inverse

Soient f une fonction et $t = (p, I)$ un modèle de Taylor représentant f sur l'intervalle J, on a, pour tout $x \in J$, $f(x) = p(x) + r$ avec $r \in I$. Pour représenter l'inverse de f, on utilise l'équivalence ci-dessous (si ni f ni p ne s'annulent sur J), qui permet de se ramener à un cas où le développement en série de l'inverse est connu, c'est-à-dire $\frac{1}{1-t}$

avec t proche de 0 :

$$\begin{aligned}\frac{1}{f(x)} =& \frac{1}{p(x)+r} \\ =& \frac{1}{p(0)} \cdot \frac{1}{1-\left(1-\frac{p(x)}{p(0)}\right)} \cdot \frac{p(x)}{p(x)+r}.\end{aligned}$$

On définit $q(x) = 1 - \frac{p(x)}{p(0)}$ qui s'annule pour $x = 0$ et on obtient :

$$\frac{1}{f(x)} = \frac{1}{p(0)} \cdot \frac{1}{1-q(x)} \cdot \frac{p(x)}{p(x)+r}.$$

En utilisant le développement en série de $\frac{1}{1-q(x)}$, on a

$$\frac{1}{f(x)} = \frac{1}{p(0)} \cdot \left(\sum_{i=0}^{N} (q(x))^i + \frac{(q(x))^{N+1}}{1-q(x)}\right) \cdot \frac{p(x)}{p(x)+r}.$$

En utilisant $\frac{1}{1+x} = 1 - \frac{x}{1+x}$ avec $x = \frac{r}{p(x)}$, on le ré-écrit

$$\frac{1}{f(x)} = \frac{1}{p(0)} \cdot \left(\sum_{i=0}^{N} (q(x))^i + \frac{(q(x))^{N+1}}{1-q(x)}\right) \cdot \left(1 - \frac{r}{p(x)+r}\right).$$

On définit $s_1 = \frac{(q(x))^{N+1}}{1-q(x)}$ et $s_2 = -\frac{r}{p(x)+r}$ et on développe l'expression :

$$\begin{aligned}\frac{1}{f(x)} =& \frac{1}{p(0)} \cdot \left(\sum_{i=0}^{N} (q(x))^i + s_1\right) \cdot (1+s_2) \\ =& \frac{1}{p(0)} \sum_{i=0}^{N} (q(x))^i + \frac{1}{p(0)} \sum_{i=0}^{N} (q(x))^i \cdot s_2 + \frac{1}{p(0)} s_1 + \frac{1}{p(0)} s_1 s_2.\end{aligned}$$

Le polynôme $\frac{1}{p(0)} \sum_{i=0}^{N} q(x)^i$ tronqué à l'ordre N forme la partie polynomiale, les termes tronqués de la partie polynomiale ainsi que les termes de l'expression faisant intervenir s_1 et s_2 sont encadrés par un intervalle, dont le calcul est détaillé ci-dessous.

Si $q(x)$ appartient à l'intervalle Q, alors s_1 appartient à l'intervalle $\frac{Q^{N+1}}{1-Q}$.

Si $r(x) = 1 + x + \cdots + x^n$, le polynôme $\sum_{i=0}^{N} q(x)^i$ est borné en utilisant la fonction `intervalFromRealSeq`, qui évalue ce polynôme sur l'intervalle J.

Le terme s_2 souffre de décorrélation. Pour borner ce terme, on a défini un nouvel opérateur à partir des bornes de I/P :

$$-s_2 \in \left[\frac{1}{1+\frac{1}{\underline{I/P}}}, \frac{1}{1+\frac{1}{\overline{I/P}}}\right],$$

où P borne $p(x)$.

Cet opérateur ne peut pas être remplacé par l'une des expressions

$$\frac{1}{1+P/I} \qquad \text{ou} \qquad \frac{1}{1+\frac{1}{I/P}}$$

parce que I usuellement contient 0 et donc ne peut pas être utilisé comme diviseur.

Toutes les autres opérations sont effectuées en utilisant l'arithmétique d'intervalles.

Nous avons également démontré que l'inverse satisfait la propriété d'inclusion (voir fig. 3.3).

3.6 Racine carrée

Soient f une fonction et $t = (p, I)$ un modèle de Taylor représentant f sur l'intervalle J, on a, pour tout $x \in J$, $f(x) = p(x) + r$ avec $r \in I$. Pour calculer la racine carrée de f, on utilise l'identité ci-dessous (si $p(x)$ et $1 + \frac{r}{p(x)}$ sont positifs et si $p(x)$ ne s'annule pas) qui permet de se ramener à un cas où le développement en série de la racine carrée est connu, c'est-à-dire $\sqrt{1+t}$ où t est proche de 0 :

$$\begin{aligned}
\sqrt{f(x)} &= \sqrt{p(x)+r} \\
&= \sqrt{1+\frac{r}{p(x)}} \cdot \sqrt{p(x)} \\
&= \sqrt{1+\frac{r}{p(x)}} \cdot \sqrt{p(0)} \cdot \sqrt{1+\frac{p(x)}{p(0)}-1}.
\end{aligned}$$

On définit $u(x) = \sqrt{1+\frac{r}{p(x)}} - 1$, $q(x) = \frac{p(x)}{p(0)} - 1$ qui s'annule pour $x = 0$, donc

$$\sqrt{f(x)} = (1+u(x)) \cdot \sqrt{p(0)} \cdot \sqrt{1+q(x)}.$$

Soient $\mathrm{SqrtApp}(x, \mathrm{Ns})$ une approximation rationnelle d'ordre Ns de la racine carrée de x et $v = \sqrt{p(0)} - \mathrm{SqrtApp}(p(0), \mathrm{Ns})$, on peut écrire

$$\sqrt{f(x)} = (1+u(x)) \cdot (\mathrm{SqrtApp}(p(0), \mathrm{Ns}) + v) \cdot \sqrt{1+q(x)}.$$

Soient $\mathrm{S}(x, N)$ le développement de Taylor à l'ordre N de $\sqrt{1+x}$ en 0, et $w(x) = \sqrt{1+q(x)} - \mathrm{S}(q(x), N)$, on obtient, en développant l'expression :

$$\begin{aligned}
\sqrt{f(x)} &= (1+u(x)) \cdot (\mathrm{SqrtApp}(p(0), \mathrm{Ns}) + v) \cdot (\mathrm{S}(q(x), N) + w(x)) \\
&= \mathrm{SqrtApp}(p(0), \mathrm{Ns}) \cdot \mathrm{S}(q(x), N) + \mathrm{SqrtApp}(p(0), \mathrm{Ns}) \cdot w(x) \\
&\qquad + v \cdot (\mathrm{S}(q(x), N) + w(x)) \\
&\qquad + u(x) \cdot (\mathrm{SqrtApp}(p(0), \mathrm{Ns}) + v) \cdot (\mathrm{S}(q(x), N) + w(x))
\end{aligned}$$

```
tm_inv_enclosure :
  LEMMA FORALL (f : [domIntervalType -> nzreal],
        t : { t : tm | t‘P(0) /= 0
        AND (t‘I / intervalFromRealSeq(t‘P, N))‘lb /= 0
        AND (t‘I / intervalFromRealSeq(t‘P, N))‘ub /= 0
        AND (t‘I / intervalFromRealSeq(t‘P, N)) > -1
     }) :
        containment( f, t)
    AND
        (FORALL xu : polynomial(t‘P, N)(xu) /= 0
    AND
         (f(xu)-polynomial(t‘P, N)(xu))
     / polynomial(t‘P, N)(xu) /= 1
         % l’erreur relative est plus petite que 1.
         %   Autrement dit, on se trompe de moins de 100%.
    AND
         polynomial(LAMBDA (i : nat) :
            IF i = 0 THEN 0 ELSE -t‘P(i) / t‘P(0) ENDIF,
            N)(xu) /= 1)
         % le 1er terme du développement en série donne
         %    l’ordre de grandeur de f(xu), les autres
         %    monômes sont des termes correctifs.
    AND
         Zeroless?([| t‘P(0) |]) AND
         Zeroless?(([| 1 |] - intervalFromRealSeq(LAMBDA
                                  (i : nat) :
                                  IF i = 0
                                  THEN 0
                                  ELSE -t‘P(i) / t‘P(0)
                                  ENDIF,
                                  N))) AND
         Zeroless?(intervalFromRealSeq(t‘P, N))
    IMPLIES
        containment( 1/f, inv(t))
```

FIG. 3.3 – Propriété d’inclusion de l’inverse.

```
tm_sqrt_enclosure : LEMMA FORALL (
        (t : tm | t‘P(0) > 0 AND t‘I > -1
      AND
        intervalFromRealSeq(t‘P, N)/t‘P(0) > 0 ) ) :
      containment(f, t)
  AND
      (FORALL (xu : domIntervalType[N, domInterval] )
              : f(xu) > 0)
  AND
      intervalFromRealSeq(t‘P, N)/t‘P(0) > 0
  AND
      Zeroless?(intervalFromRealSeq(t‘P, N))
  AND
      t‘I/intervalFromRealSeq(t‘P, N) >= -1
  IMPLIES
      containment(
        (LAMBDA (xu : domIntervalType[N, domInterval] ) :
             sqrt(f(xu))), sqrt(t))
```

FIG. 3.4 – Propriété d'inclusion de la racine carrée.

Le polynôme $\mathrm{SqrtApp}(p(0), \mathrm{Ns}) \cdot \mathrm{S}(q(x), N)$ tronqué à l'ordre N forme la partie polynomiale du modèle de Taylor.

Soient $x \in X$, P qui vérifie $p(x) \in P$ et Q tel que $q(x) \in Q$. Soient aussi `Sqrt`$(X,$ `Ns`$)$ une fonction qui borne $\sqrt{x}$ sur X avec une approximation d'ordre `Ns` et enfin `nderiv_sqrt_offsetI`(X, n) une fonction qui borne la n-ième dérivée de $\sqrt{1+x}$. Pour borner le reste des termes, on note que pour tout $x \in X$:

$$u(x) \in \mathrm{Sqrt}(1 + I/P, \mathrm{Ns}) - 1, \tag{3.6}$$

$$v \in \mathrm{Sqrt}(p(0), \mathrm{Ns}), \tag{3.7}$$

$$w(x) \in \frac{\mathrm{nderiv\_sqrt\_offsetI}(0 \cup Q) \cdot Q^{N+1}}{(N+1)!}. \tag{3.8}$$

En utilisant ces définitions et l'arithmétique d'intervalles, on calcule la partie intervalle du modèle de Taylor. Ces formules de calcul nous ont permis de démontrer que la racine carrée satisfait la propriété d'inclusion (voir fig. 3.4).

3.7 Conclusion

Dans ce chapitre nous avons présenté notre implantation des modèles de Taylor, qui repose sur les théories de base des suites à support fini et des polynômes, que nous avons développées au préalable. Ces théories peuvent être utilisées indépendamment des modèles de Taylor. Nous avons défini le type des modèles de Taylor en PVS et les opérations de base : addition, opposé, soustraction, produit par un scalaire et multiplication, de l'arithmétique des modèles de Taylor. Nous avons également prouvé que les opérations satisfont la propriété d'inclusion. Enfin, nous avons construit l'inverse et la racine carrée des modèles de Taylor et démontré en PVS qu'ils satisfont aussi la propriété d'inclusion. Nous continuons ce document en présentant notre développement des modèles de Taylor pour d'autres fonctions telles que l'exponentielle et l'arctangente.

Chapitre 4

Modèles de Taylor : fonctions transcendantes

Dans ce chapitre, nous présentons la construction des modèles de Taylor pour l'exponentielle, l'arc-tangente et le sinus. Nous présentons de façon détaillée la construction d'un modèle de Taylor pour l'exponentielle, puis nous expliquons la méthode générale que nous avons développée pour construire des modèles de Taylor. Enfin nous détaillons la théorie des séries alternées que nous avons dû développer afin de pouvoir obtenir la convergence du reste du développement de Taylor de l'arc-tangente autour de 0, ainsi que son utilisation pour construire les modèles de Taylor de l'arc-tangente et du sinus.

Dès lors que l'on dispose des opérations arithmétiques de base sur les modèles de Taylor, on peut construire les modèles de Taylor pour des fonctions polynomiales. Le pas suivant est de construire les modèles de Taylor pour des fonctions transcendantes. Nous commençons par la construction du modèle de Taylor de l'exponentielle. À partir de cette construction nous présentons une méthode pour construire un modèle de Taylor d'une fonction g.

4.1 Exponentielle

La technique utilisée consiste à remplacer $f(x)$ par $p(x) + r$ avec $r \in I$, puis à isoler la partie intervalle r et enfin à se ramener en 0 pour pouvoir utiliser un développement de Taylor autour de 0. Pour se ramener en 0, il faut isoler et traiter séparément la partie constante du polynôme. Afin de traiter cette partie constante, il nous faut savoir évaluer $\exp(p(0))$, un nombre réel qui n'est pas exactement représentable sur ordinateur. Pour cette raison, on l'évalue en utilisant une approximation rationnelle $\widetilde{\exp}$ de l'exponentielle et l'erreur d'approximation est prise en compte dans la partie intervalle.

Soient f une fonction et $t = (p, I)$ un modèle de Taylor représentant f sur l'intervalle J, on a, pour tout $x \in J$, $f(x) = p(x) + r$ avec $r \in I$. Pour calculer l'exponentielle de f on utilise les identités ci-dessous :

$$\begin{aligned}\exp(f(x)) &= \exp(p(x) + r) \\ &= \exp(p(x)) \cdot \exp(r) \\ &= \exp(p(x)) + \exp(p(x)) \cdot (\exp(r) - 1).\end{aligned} \tag{4.1}$$

On a besoin de $\mathrm{Exp}(Y, \mathtt{NE})$ une fonction qui encadre $\exp(y)$ pour y dans un intervalle Y en utilisant une approximation d'ordre NE. Cette fonction est déjà disponible dans la bibliothèque d'intervalles de PVS [10]. Soit P un intervalle qui encadre $p(x)$ sur J, le terme $\exp(p(x)) \cdot (\exp(r) - 1)$ est encadré par l'intervalle

$$\mathrm{Exp}(P, \mathrm{NE}) \cdot (\mathrm{Exp}(I, \mathrm{NE}) - 1). \tag{4.2}$$

Pour se ramener en 0 et ensuite utiliser le développement en série de Taylor autour de 0 de l'exponentielle, il faut traiter à part la partie constante du polynôme $p(x)$. Pour cela on écrit

$$\exp(p(x)) = \exp(p(0)) \cdot \exp(p(x) - p(0)).$$

On définit $q(x) = p(x) - p(0)$ qui s'annule pour $x = 0$, on a

$$\exp(p(x)) = \exp(p(0)) \cdot \exp(q(x)). \tag{4.3}$$

On suppose que l'on dispose de $\widetilde{\exp}(x, n)$ une approximation rationnelle d'ordre n de l'exponentielle en un point x, on a

$$\exp(p(x)) = \widetilde{\exp}(p(0), \mathrm{NE}) \cdot \exp(q(x)) + (\exp(p(0)) - \widetilde{\exp}(p(0), \mathrm{NE})) \cdot \exp(q(x)). \tag{4.4}$$

Soit $Q(J)$ un intervalle qui contient $q(x)$ pour $x \in J$, le terme d'erreur $(\exp(p(0)) - \widetilde{\exp}(p(0), \mathrm{NE})) \cdot \exp(q(x))$ est encadré par l'intervalle

$$(\mathrm{Exp}(p(0), \mathrm{NE}) - \widetilde{\exp}(p(0), \mathrm{NE})) \cdot \mathrm{Exp}(Q(J), \mathrm{NE}). \tag{4.5}$$

En utilisant la série de Taylor de l'exponentielle, on sait qu'il existe une valeur c comprise entre 0 et $q(x)$ telle que l'on ait :

$$\exp(q(x)) = \sum_{i=0}^{N} \frac{1}{i!} q(x)^i + \exp(c) \cdot \frac{q(x)^{N+1}}{(N+1)!}. \tag{4.6}$$

Le terme $\exp(c) \cdot \frac{q(x)^{N+1}}{(N+1)!}$ est encadré par l'intervalle

$$\mathrm{Exp}(Q(J), \mathrm{NE}) \cdot \frac{(Q(J))^{N+1}}{(N+1)!}. \tag{4.7}$$

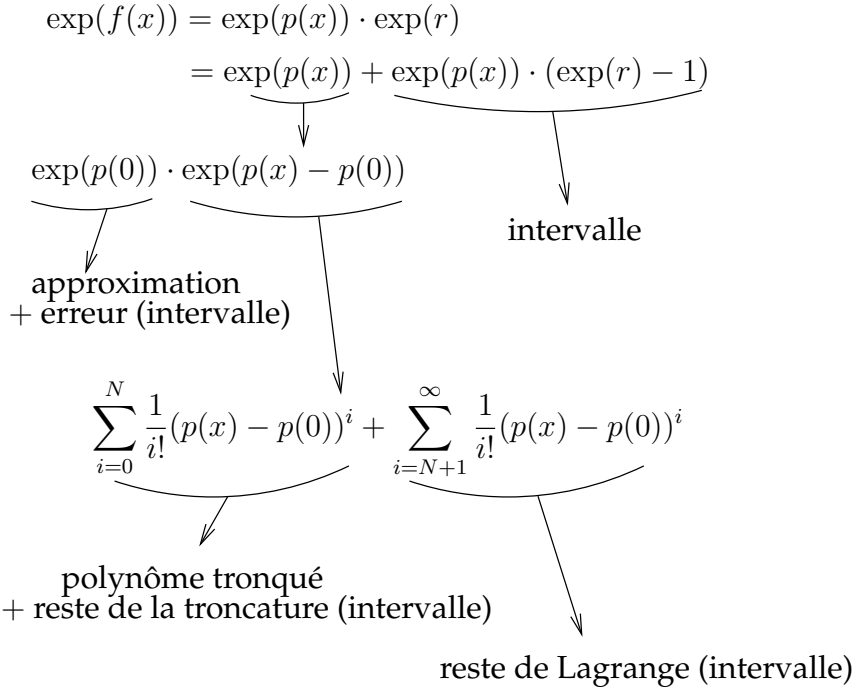

FIG. 4.1 – Construction du modèle de Taylor pour l'exponentielle.

À partir de (4.1), (4.4) et (4.6) on construit le modèle de Taylor (voir fig. 4.1). La partie polynomiale du modèle de Taylor est $\widetilde{\exp}(p(0)) \cdot \sum_{i=0}^{N} \frac{1}{i!} q(x)^i$ tronqué à l'ordre N. L'encadrement des termes tronqués de la partie polynomiale ainsi que du reste des termes est obtenu en utilisant (4.2), (4.5) et (4.7) et l'arithmétique par intervalles. Il forme la partie intervalle du modèle de Taylor. La construction du modèle de Taylor et la propriété d'inclusion de l'exponentielle sont présentés figure 4.2.

4.2 Méthode pour construire un modèle de Taylor d'une fonction

Nous présentons dans cette partie une méthode pour construire un modèle de Taylor qui représente une fonction $g : D \subset \mathbb{R} \to \mathbb{R}$ de classe C^{N+1}. Supposons que l'on ait déjà construit, pour une fonction f, $t = (p, I)$ un modèle de Taylor représentant f sur l'intervalle J : on a, pour tout $x \in J$, $f(x) = p(x) + r$ avec $r \in I$. On définit $P(J)$ comme l'intervalle qui encadre $p(x)$, obtenu en utilisant la fonction `intervalFromRealSeq` qui évalue le polynôme p en l'intervalle J. On montre maintenant comment construire un modèle de Taylor pour $g \circ f$.

Lemme 1. *Si G' est une fonction à valeurs intervalles qui encadre la dérivée g' de g sur J, alors*

$$\forall x \in J,\ g(f(x)) - g(p(x)) = g(p(x) + r) - g(p(x)) \in G'\left(P(J) + I\right) \cdot I$$

Démonstration. En appliquant le théorème de Taylor-Lagrange au premier ordre, on sait que $\exists \xi \in [a, y]$ tel que $g(y) = g(a) + g'(\xi)(y - a)$. On remplace y par $p(x) + r$ et a par $p(x)$ pour obtenir

$$g(p(x) + r) = g(p(x)) + g'(\xi) \cdot r$$

donc

$$g(p(x) + r) - g(p(x)) = g'(\xi) \cdot r,$$

$$\xi \in p(x) + r \subset p(x) + I \Longrightarrow \xi \in P(J) + I.$$

La dernière expression est encadrée par l'intervalle

$$G'\left(P(J) + I\right) \cdot I$$

ce qui démontre le résultat. □

Lemme 2. *Soient $q(x) = p(x) - p(0)$ qui s'annule pour $x = 0$ et $Q(J)$ une fonction à valeurs intervalles qui encadre $q(x)$. Si $G^{(n)}$ est une fonction qui encadre la dérivée n-ième de g, alors*

$$g(p(x)) - \sum_{i=0}^{N} \frac{g^{(i)}(p(0))}{i!} (q(x))^i \in \frac{G^{(N+1)}(P(J))}{(N+1)!} (Q(J))^{N+1}$$

```
exp( t : tm) : tm =
    (# P := lnexp_fnd@exp_estimate(t‘P(0), NE) *
            % Approximation rationelle de exp(P(0))
                  trunc(comp(sequence_fact_x,
                     (LAMBDA (i : nat) : IF i=0
                             THEN 0
                             ELSE t‘P(i) ENDIF), N), N),
            % N premiers termes du développement en
            %  série de Taylor
       I := lnexp_fnd@exp_estimate(t‘P(0), NE)
            * intervalFromRealSeq(
              trunc_remainder(comp(sequence_fact_x,
                (LAMBDA (i : nat) : IF i=0
                        THEN 0
                        ELSE t‘P(i) ENDIF), N),
                       N), N*N )
            % reste de la troncature de la partie polynomiale
            + lnexp_fnd@exp_estimate(t‘P(0), NE)*
                Exp(intervalFromRealSeq(
                (LAMBDA (i : nat) : IF i=0
                        THEN 0
                        ELSE t‘P(i) ENDIF), N), NE)
                 * intervalFromRealSeq(
                    (LAMBDA (i : nat) : IF i=0
                            THEN 0
                            ELSE t‘P(i) ENDIF), N)^(N+1)
                               / factorial(N+1)
             % Reste de Lagrange
            + (Exp([|t‘P(0)|], NE)
                - lnexp_fnd@exp_estimate(t‘P(0), NE))
               *Exp(intervalFromRealSeq(
                     (LAMBDA (i : nat) : IF i=0
                             THEN 0
                             ELSE t‘P(i) ENDIF), N), NE)
             % Erreur d’aproximation sur l’exponentielle de P(0)
            + Exp(intervalFromRealSeq(t‘P, N), NE)
                  *(Exp(t‘I, NE) - 1)
              % Terme qui correspond à la partie intervalle r
    #)

tm_exp_enclosure : LEMMA containment( f, t)
          IMPLIES containment( (LAMBDA xu : exp(f(xu))),
                       exp(t))
```

FIG. 4.2 – Modèle de Taylor et propriété d'inclusion de l'exponentielle.

Démonstration. En utilisant le théorème de Taylor-Lagrange à l'ordre N, qui s'énonce ainsi

$$\exists \xi \in [a, y] \,/\; g(y) = \sum_{i=0}^{N} \frac{g^{(i)}(a)}{i!}(y-a)^i + \frac{g^{(N+1)}(\xi)}{(N+1)!}(y-a)^{N+1}$$

avec $y = p(x)$ et $a = p(0)$ on a

$$g(p(x)) = \sum_{i=0}^{N} \frac{g^{(i)}(p(0))}{i!}(p(x)-p(0))^i + \frac{g^{(N+1)}(\xi)}{(N+1)!}(p(x)-p(0))^{N+1}.$$

En remplaçant $p(x) - p(0)$ par $q(x)$, on a donc

$$g(p(x)) - \sum_{i=0}^{N} \frac{g^{i}(p(0))}{i!}(q(x))^i = \frac{g^{(N+1)}(\xi)}{(N+1)!}(q(x))^{N+1}.$$

Cette dernière expression est bornée par l'intervalle

$$\frac{G^{(N+1)}(P(J))}{(N+1)!}(Q(J))^{N+1}$$

qui est bien l'encadrement annoncé. □

On suppose que l'on dispose de $\widetilde{g}(x, \mathrm{Ng})$ une approximation rationnelle de g d'ordre Ng et de Eg une fonction qui encadre l'erreur d'approximation de $\widetilde{g}$, c'est-à-dire que l'on a

$$g(x) - \widetilde{g}(x, \mathrm{Ng}) \in \mathrm{Eg}(x, \mathrm{Ng}). \tag{4.8}$$

Pour calculer le modèle de Taylor qui représente g, on utilise l'égalité ci-dessous

$$g(f(x)) = g(p(x) + r).$$

On sépare la partie r correspondant à la partie intervalle de t

$$g(f(x)) = g(p(x)) + g(p(x) + r) - g(p(x)). \tag{4.9}$$

La partie $g(p(x) + r) - g(p(x))$ est encadrée par un intervalle I_1 en utilisant le lemme 1.

En utilisant le développement en série de Taylor de $g(x)$, on obtient

$$g(p(x)) = \sum_{i=0}^{N} g^{(i)}(p(0)) \cdot q(x)^i + g(p(x)) - \sum_{i=0}^{N} g^{(i)}(p(0)) \cdot q(x)^i. \tag{4.10}$$

La partie $g(p(x)) - \sum_{i=0}^{N} g^{(i)}(p(0)) \cdot q(x)^i$ est encadrée par un intervalle I_2 en utilisant le lemme 2.

Il reste finalement à traiter le début du développement de Taylor, et pour cela il faut être capable d'évaluer g et ses dérivées successives en $p(0)$

$$\begin{aligned}\sum_{i=0}^{N} g^{(i)}(p(0)) \cdot q(x)^i =& g(p(0)) + \sum_{i=1}^{N} g^{(i)}(p(0)) \cdot q(x)^i \\ =& \widetilde{g}(p(0), \mathrm{Ng}) + \sum_{i=1}^{N} g^{(i)}(p(0)) \cdot q(x)^i + g(p(0)) - \widetilde{g}(p(0), \mathrm{Ng}). \quad (4.11)\end{aligned}$$

La partie $g(p(0)) - \widetilde{g}(p(0), \mathrm{Ng})$ est bornée par un intervalle I_3 donné par $\mathrm{Eg}(p(0), \mathrm{Ng})$. La partie $\widetilde{g}(p(0)) + \sum_{i=1}^{N} g^{(i)}(p(0)) \cdot q(x)^i$ tronquée à l'ordre N forme la partie polynomiale du modèle de Taylor, le reste des termes de la troncature de cette expression est borné par un intervalle I_4 en utilisant `intervalFromRealSeq`. On calcule la partie intervalle du modèle de Taylor en additionnant les intervalles I_1 (qui fait intervenir la partie intervalle I), I_2 (qui fait intervenir le reste du développement de Taylor), I_3 (qui fait intervenir l'approximation de $g(p(0))$ par un nombre rationnel) et I_4 (qui prend en compte la troncature de la partie polynomiale de $g(t)$).

4.3 Arc-tangente

Pour représenter l'arc-tangente on a commencé par appliquer la méthode proposée en §4.2. Cependant la partie intervalle pour ce modèle de Taylor de l'arc-tangente présente une forte décorrélation. La décorrélation provient principalement de l'évaluation par intervalles de la dérivée $(N+1)$-ième de l'arc-tangente utilisée pour borner la partie intervalle de l'expression (4.10). Par exemple, la dérivée 5-ième de $\mathrm{atan}(x)$ est

$$\frac{384x^4}{(1+x^2)^5} - \frac{282x^2}{(1+x^2)^4} + \frac{24}{(1+x^2)^3}$$

qui, lorsqu'on l'évalue sur l'intervalle $[-1, 1]$ et qu'on la divise par $5!$ donne l'intervalle $[\frac{-19}{8}, \frac{17}{5}] \backsimeq [-2.375, 3.4]$, ce qui est très large. Si on utilise Maple 10 pour déterminer les bornes de cette expression, on obtient l'intervalle $[-0.084375, 0.2]$. On peut penser que si on augmente l'ordre du modèle de Taylor la précision augmente, mais dans ce cas, la décorrélation de la dérivée $(N+1)$-ième augmente, par exemple la dérivée 7-ième de l'arc-tangente est

$$\frac{17280x^2}{(1+x^2)^5} - \frac{57600x^4}{(1+x^2)^6} + \frac{46080x^6}{(1+x^2)^7} - \frac{720}{(1+x^2)^5}.$$

Après évaluation sur l'intervalle $[-1, 1]$ et division par $7!$ on obtient l'intervalle $[\frac{-81}{7}, \frac{201}{16}] \backsimeq [-11.57143, 12.5625]$. En utilisant encore Maple pour déterminer les bornes de cette expression, on obtient un intervalle $X \subseteq [-0.14286, 0.07583]$.

Pour construire le modèle de Taylor pour l'arc-tangente en évitant le problème de décorrélation, nous utilisons un résultat sur les séries alternées qui n'est pas dans la

bibliothèque de PVS. Pour cette raison, nous avons développé une théorie des séries alternées.

4.3.1 Séries alternées

Un série alternée est une série réelle dont les termes en valeur absolu décroissent vers 0 et changent alternativement de signe.

Théorème 1. *Soit* (a_n) *une suite à termes positifs* $(\forall(n : \mathbb{N}).\ a_n > 0)$*, décroissante (*$a_{n+1} <= a_n$*), tendant vers 0 (*$\text{convergence}(a, 0)$*). Alors la série alternée* $\sum (-1)^n a_n$ *converge, et les restes partiels*

$$R_n = \sum_{k=n+1}^{+\infty} (-1)^k a_k \quad \textit{vérifient} \quad |R_n| \leq a_{n+1} \textit{ pour tout } n \in \mathbb{N}.$$

La démonstration se fait à partir des théorèmes ci-dessous. Pour faciliter la lecture des théorèmes qui suivent, on reproduit la définition des `series` de la bibliothèque de la Nasa[1], qui se trouve dans le fichier `series/series.pvs`

```
series(a) : sequence[real] =
                (LAMBDA (n : nat) : sigma(0, n, a))
```

On définit la fonction `alternate` d'une suite, qui crée une série alternée à partir d'une autre série, par :

```
alternate(a) : sequence[real] = LAMBDA (n : nat) : (-1)^n*a(n)
```

On note $S_n = \sum_{i=0}^{n} (-1)^k a_k$ la somme partielle d'ordre n, que l'on écrit en PVS comme

```
series(alternate(a))(n)
```

Si la suite a est décroissante, on a $S_{2n+2} - S_{2n} \leq 0$ et $S_{2n+3} - S_{2n+1} \geq 0$, ce qui s'écrit en PVS

```
alternate_aux0 : LEMMA FORALL n : decreasing(a) IMPLIES
        series(alternate(a))(2*n+2)
               -  series(alternate(a))(2*n) <= 0
           AND series(alternate(a))(2*n+3)
               -  series(alternate(a))(2*n + 1) >= 0
```

ceci est démontré à partir de la définition d'une série et du fait que la série a est décroissante.

Les deux lemmes `alternate_aux1` et `alternate_aux2` sont une conséquence du lemme `alternate_aux0`, ils établissent que la suite (S_{2n}) est décroissante et que la suite (S_{2n+1}) est croissante.

[1] `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`

```
alternate_aux1 : LEMMA decreasing(a) IMPLIES
    decreasing(LAMBDA n : series(alternate(a))(2*n))

alternate_aux2 : LEMMA decreasing(a) IMPLIES
    increasing(LAMBDA n : series(alternate(a))(2*n+1))
```

On établit à partir de la définition de S_n que $S_{2n+1} - S_{2n} = -a_{2n+1}$ et $S_{2n+2} - S_{2n+1} = a_{2n+2}$, ce qui se traduit en PVS par

```
alternate_aux3 : LEMMA
     series(alternate(a))(2*n+1)
         - series(alternate(a))(2*n) = -a(2*n+1)

alternate_aux4 : LEMMA
     series(alternate(a))(2*n+2)
         - series(alternate(a))(2*n+1) = a(2*n+2)
```

On démontre, par récurrence sur n et en utilisant le fait que $S_1 \leq S_{2n+2} \leq S_{2n} \cdots \leq S_2 \leq S_0$ et $S_1 \leq S_{2n+1} \leq S_{2n}$, que la suite (S_{2n}) est bornée inférieurement

```
alternate_aux5 : LEMMA decreasing(a) AND (FORALL n : a(n) > 0)
   IMPLIES
     bounded_below?(LAMBDA n : series(alternate(a))(2*n))
```

On démontre aussi, par récurrence sur n et en utilisant le fait que $S_1 \leq S_3 \leq \cdots \leq S_{2n+1} \leq S_{2n+3} \leq S_0$ et $S_{2n+3} \leq S_{2n+4} \leq S_0$, que la suite (S_{2n+1}) est bornée supérieurement

```
alternate_aux6 : LEMMA decreasing(a) AND (FORALL n : a(n) > 0)
   IMPLIES
     bounded_above?(LAMBDA n : series(alternate(a))(2*n+1))
```

Comme conséquence du théorème ci-dessus on a la convergence des deux séries (S_{2n}) et (S_{2n+1}) :

```
alternate_conv1 : LEMMA decreasing(a) AND (FORALL n : a(n) > 0)
   IMPLIES convergence(LAMBDA n : series(alternate(a))(2*n),
              inf(LAMBDA n : series(alternate(a))(2*n)))

alternate_conv2 : LEMMA decreasing(a) AND (FORALL n : a(n) > 0)
    IMPLIES convergence(LAMBDA n : series(alternate(a))(2*n+1),
              sup(LAMBDA n : series(alternate(a))(2*n+1)))
```

On montre que la différence de la série impaire avec la série paire converge vers 0 en utilisant l'identitité $S_{2n+1} - S_{2n} = -a_{2n+1}$ et l'hypothèse de convergence de la suite (a) vers 0

```
 alternate_conv0 : LEMMA decreasing(a) AND (FORALL n : a(n) > 0)
                       AND convergence(a, 0)
    IMPLIES
         convergence((LAMBDA n : series(alternate(a))(2*n+1))-
                     (LAMBDA n : series(alternate(a))(2*n)), 0)
```

Il en résulte donc que les suites (S_{2n+1}) et (S_{2n}) convergent vers une même limite S :

```
alternate_eqlim : LEMMA decreasing(a) AND (FORALL n : a(n) > 0)
                       AND convergence(a, 0)
    IMPLIES
          sup(LAMBDA n : series(alternate(a))(2*n+1)) =
          inf(LAMBDA n : series(alternate(a))(2*n))

S(a : { a | (FORALL n : a(n) > 0) AND decreasing(a)} ): real =
               inf(LAMBDA n : series(alternate(a))(2*n))
```

On en déduit que la suite (S_n) converge vers S

```
alternate_conv : LEMMA (FORALL n : a(n) > 0) AND
                        decreasing(a) AND convergence(a, 0)
                   IMPLIES convergence(series(alternate(a)),
                           S(a))

alternate_convergent : LEMMA (FORALL n : a(n) > 0) AND
                        decreasing(a) AND convergence(a, 0)
                   IMPLIES conv_series?(alternate(a))
```

On a que $S_{2n+1} \leq \mathrm{S} \leq S_{2n}$

```
 alternate_ineq : LEMMA decreasing(a) AND (FORALL n : a(n) > 0)
                       AND convergence(a, 0)
    IMPLIES
          series(alternate(a))(2*n+1) <= S(a) AND
          S(a) <= series(alternate(a))(2*n)
```

On définit la fonction R_n en PVS

```
Rn( a : { a | (FORALL n : a(n) > 0)
              AND decreasing(a)
              AND convergence(a, 0)}, n ) : real =
       inf_sum(n+1, alternate(a))
```

On établit que $R_n = \mathrm{S} - S_n$

```
alternate_diffrn : LEMMA (FORALL n : a(n) > 0) AND
                         decreasing(a) AND convergence(a, 0)
             IMPLIES
                    Rn(a, n) = S(a) - series(alternate(a))(n)
```

en utilisant le fait que la suite (S_n) converge vers S (lemme `alternate_conv`).

En utilisant les lemmes `alternate_diffrn`, `alternate_ineq`, `alternate_aux3` et `alternate_aux4` on a $|R_{2n}| \leq a_{2n+1}$ et $|R_{2n+1}| \leq a_{2n+2}$

```
alternate_rem0 : LEMMA (FORALL n : a(n) > 0) AND
                         decreasing(a) AND convergence(a, 0)
                    IMPLIES abs(Rn(a, 2*n)) <= a(2*n+1),

alternate_rem1 : LEMMA (FORALL n : a(n) > 0) AND
                         decreasing(a) AND convergence(a, 0)
                    IMPLIES abs(Rn(a, 2*n+1)) <= a(2*n+2)
```

ce qui montre que $|R_n| \leq a_{n+1}$.

```
alternate_rem : LEMMA (FORALL n : a(n) > 0) AND
                         decreasing(a) AND convergence(a, 0)
                    IMPLIES abs(Rn(a, n)) <= a(n+1)
```

ce qui termine la preuve en PVS du théorème 1.

4.3.2 Modèle de Taylor pour l'arc-tangente

Nous allons maintenant adapter la méthode présentée au §4.2, pour utiliser la propriété de l'arc-tangente d'avoir un développement de Taylor en 0 dont les coefficients forment une suite alternée. Soient f une fonction et $t = (p, I)$ un modèle de Taylor représentant f sur l'intervalle J, on a, pour tout $x \in J$, $f(x) = p(x) + r$ avec $r \in I$. Pour calculer l'arc-tangente on utilise les identités ci-dessous :

$$\begin{aligned} \mathrm{atan}(f(x)) =& \mathrm{atan}(p(x) + r) \\ =& \mathrm{atan}(p(x)) + \mathrm{atan}(p(x) + r) - \mathrm{atan}(p(x)). \end{aligned} \tag{4.12}$$

En utilisant le lemme 1 on a que $\mathrm{atan}(p(x) + r) - \mathrm{atan}(p(x))$ est encadré par l'intervalle

$$I_1 := I \cdot \frac{1}{1 + (P(J) + I)^2}.$$

De la même façon on a

$$\mathrm{atan}(p(x)) = \mathrm{atan}(p(x) - p(0)) + \mathrm{atan}(p(x)) - \mathrm{atan}(p(x) - p(0))$$

On définit $q(x) = p(x) - p(0)$ qui s'annule pour $x = 0$

$$\text{atan}(p(x)) = \text{atan}(q(x)) + \text{atan}(p(x)) - \text{atan}(q(x)). \tag{4.13}$$

Soit $Q(J)$ un intervalle qui encadre $q(x)$ pour $x \in J$. En utilisant encore une fois le lemme 1, on obtient que $\text{atan}(p(x)) - \text{atan}(q(x))$ est encadré par l'intervalle

$$I_2 := p(0) \cdot \frac{1}{1 + (Q(J) + p(0))^2}.$$

En utilisant le développement de Taylor de l'arc-tangente autour de zéro, on a

$$\begin{aligned} \text{atan}(q(x)) &= \sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1} \cdot (q(x))^{2i+1} \\ &= \sum_{i=0}^{N} \frac{(-1)^i}{2i+1} \cdot (q(x))^{2i+1} + \sum_{i=N+1}^{\infty} \frac{(-1)^i}{2i+1} \cdot (q(x))^{2i+1}. \end{aligned} \tag{4.14}$$

Si $q(x) \in Q(J) \subseteq [-1, 1]$, la partie $\sum_{i=N+1}^{\infty} \frac{(-1)^{2i+1}}{2i+1} \cdot (q(x))^{2i+1}$ est encadrée par l'intervalle $I_3 := \left[\frac{-1}{2(N+1)+1}, \frac{1}{2(N+1)+1}\right]$ en utilisant le théorème 1. La partie polynomiale du modèle de Taylor est $\sum_{i=0}^{N} \frac{(-1)^{2i+1}}{2i+1} \cdot (q(x))^{2i+1}$ tronqué à l'ordre N. Le reste des termes de la troncature est inclus dans un intervalle I_4 en utilisant `intervalFromRealSeq`. On calcule la partie intervalle du modèle de Taylor en additionnant les intervalles I_1 (qui fait intervenir la partie intervalle de t), I_2 (qui prend en compte l'erreur d'approximation pour la partie constante du polynôme p), I_3 (qui prend en compte le reste des termes du développement de Taylor) et I_4 (qui prend en compte le reste de la troncature de la partie polynomiale). Sur la figure 4.3 sont présentés le modèle de Taylor et la propriété d'inclusion de l'arc-tangente.

4.4 Sinus

En profitant du fait que la série de Taylor autour de zéro de sinus est une série alternée, on a développé le modèle de Taylor de sinus de façon similaire à celui de l'arc-tangente. Voici comment construire un modèle de Taylor pour $\sin \circ f$. Soient f une fonction et $t = (p, I)$ un modèle de Taylor représentant f sur l'intervalle J, on a, pour tout $x \in J$, $f(x) = p(x) + r$ avec $r \in I$. On suppose que l'on dispose de `Cos`(Y, n) une fonction qui encadre $cos(y)$, $y \in Y$ dans un intervalle en utilisant une approximation d'ordre n et `NS` un paramètre de la théorie qui représente l'ordre d'approximation du sinus.

Pour calculer le sinus on utilise les équivalences ci-dessous :

$$\begin{aligned} \sin(f(x)) &= \sin(p(x) + r) \\ &= \sin(p(x)) + \sin(p(x) + r) - \sin(p(x)). \end{aligned} \tag{4.15}$$

```
atan(t : tm) : tm =
    (# P := trunc(comp(seq_atan_0,
                (LAMBDA (i : nat) : IF i=0
                        THEN 0
                        ELSE t‘P(i) ENDIF), N), N),
       I := t‘I * 1/(1+Sq(
                     t‘I + (intervalFromRealSeq(t‘P, N))))
            + ([|t‘P(0)|]*1/(1+Sq(
                      [|t‘P(0)|] + (intervalFromRealSeq(
                         (LAMBDA (i : nat) : IF i=0
                                 THEN 0
                                 ELSE t‘P(i) ENDIF),
                               N)))))
            + (IF even?(N) THEN  [|-1/(1 + N), 1/(1 + N)|]
                           ELSE  [| -1/(2 + N), 1/(2 + N) |]
                           ENDIF
            + intervalFromRealSeq(
                    trunc_remainder(comp(seq_atan_0,
                    (LAMBDA (i : nat) : IF i=0
                            THEN 0
                            ELSE t‘P(i) ENDIF), N),
                       N), N*N )))
     #)

 tm_atan_enclosure : LEMMA containment( f, t )
              AND N > 2 AND (FORALL xu :
                polynomial(LAMBDA (i : nat) : IF i = 0
                             THEN 0
                             ELSE t‘P(i) ENDIF, N)(xu)
                     ## [| -1, 1 |])
              IMPLIES
                 containment( (LAMBDA xu : atan(f(xu))),
                             atan(t) )
```

FIG. 4.3 – Modèle de Taylor et propriété d’inclusion de l’arc-tangente.

En utilisant le lemme 1 on a que $\sin(p(x) + r) - \sin(p(x))$ est encadré par l'intervalle $I_1 := I \cdot \texttt{Cos}(P(X) + I, \texttt{NS})$.

De la même façon on a

$$\sin(p(x)) = \sin(p(x) - p(0)) + \sin(p(x)) - \sin(p(x) - p(0))$$

On définit $q(x) = p(x) - p(0)$ qui s'annule pour $x = 0$ et l'égalité précédente s'écrit

$$\sin(p(x)) = \sin(q(x)) + \sin(p(x)) - \sin(q(x)). \tag{4.16}$$

Soit $Q(J)$ un intervalle qui encadre $q(x)$ pour $x \in J$. En utilisant à nouveau le lemme 1, on a que $\sin(p(x)) - \sin(q(x))$ est encadré par l'intervalle $I_2 := p(0) \cdot \texttt{Cos}(Q(J) + p(0))$.

En utilisant le développement de Taylor de sinus autour de zéro, on a

$$\begin{aligned}\sin(q(x)) &= \sum_{i=0}^{\infty} \frac{(-1)^{2i+1}}{(2i+1)!} \cdot (q(x))^{2i+1} \\ &= \sum_{i=0}^{N} \frac{(-1)^{2i+1}}{(2i+1)!} \cdot (q(x))^{2i+1} + \sum_{i=N+1}^{\infty} \frac{(-1)^{2i+1}}{(2i+1)!} \cdot (q(x))^{2i+1}. \end{aligned} \tag{4.17}$$

Si $q(x) \in Q(J) \subseteq [-1, 1]$, la partie $\sum_{i=N+1}^{\infty} \frac{(-1)^{2i+1}}{(2i+1)!} \cdot (q(x))^{2i+1}$ est encadrée par l'intervalle $I_3 := \left[\frac{-1}{(2(N+1)+1)!}, \frac{1}{(2(N+1)+1)!}\right]$ en utilisant le théorème 1. La partie polynomiale du modèle de Taylor est $\sum_{i=0}^{N} \frac{(-1)^{2i+1}}{(2i+1)!} \cdot (q(x))^{2i+1}$ tronqué à l'ordre N. Le reste des termes de la troncature est encadré par un intervalle I_4 grâce à l'évaluation de ces termes sur l'intervalle J. On calcule la partie intervalle du modèle de Taylor en additionnant les intervalles I_1, I_2, I_3 et I_4. On rappelle que I_1 fait intervenir la partie intervalle du modèle de Taylor de f, que I_2 prend en compte la partie constante du polynôme p, que I_3 prend en compte le reste des termes du développement de Taylor et que I_4 prend en compte les reste de termes de la troncature de la partie polynomiale. Sur la figure 4.4 sont présentés le modèle de Taylor et la propriété d'inclusion du sinus.

4.5 Conclusion

Dans ce chapitre nous avons défini les modèles de Taylor pour l'exponentielle, l'arc-tangente et le sinus. Nous avons aussi défini une méthode générale pour construire le modèle de Taylor d'une fonction f. Cette méthode ne marche pas bien en pratique quand l'expression correspondant à la dérivée $(N+1)$-ième de la fonction a une décorrélation qui augmente avec N, comme c'est le cas de l'arc-tangente : dans ce cas augmenter le degré du modèle de Taylor n'améliore pas la qualité de l'encadrement. Pour calculer le modèle de Taylor de l'arc-tangente, nous avons développé une théorie des séries alternées en PVS. Cette théorie peut être utilisée pour démontrer la convergence

```
sin(t : tm) : tm =
    (# P := trunc(comp(seq_sin_0,
               (LAMBDA (i : nat) : IF i=0
                       THEN 0
                       ELSE t‘P(i) ENDIF), N), N),
       I := t‘I * Cos(t‘I + intervalFromRealSeq(t‘P, N), NS)
            + (([|t‘P(0)|]* Cos(
                        [|t‘P(0)|] + intervalFromRealSeq(
                  (LAMBDA (i : nat) : IF i=0
                          THEN 0
                          ELSE t‘P(i) ENDIF), N), NS)
            + (IF even?(N)
               THEN  [|-1/factorial(1 + N),
                       1/factorial(1 + N)|]
               ELSE  [| -1/factorial(2 + N),
                        1/factorial(2 + N) |]
               ENDIF
            + intervalFromRealSeq(
                trunc_remainder(comp(seq_sin_0,
                    (LAMBDA (i : nat) : IF i=0
                            THEN 0
                            ELSE t‘P(i) ENDIF), N),
                  N), N*N ))))
     #)

 tm_sin_enclosure : LEMMA containment( f, t )
      AND N > 2 AND
         (FORALL xu :
              polynomial(LAMBDA (i : nat) : IF i = 0
                                 THEN 0
                                 ELSE t‘P(i) ENDIF, N)(xu)
              ## [| -1, 1 |])
      IMPLIES
          containment( (LAMBDA xu : sin(f(xu))), sin(t) )
```

FIG. 4.4 – Modèle de Taylor et propriété d’inclusion du sinus.

d'une série alternée et aussi pour borner son reste. Nous avons également utilisé cette théorie pour construire le modèle de Taylor de la fonction sinus.

Nous avons construit les modèles de Taylor pour les fonctions différemment des implantations actuelles comme COSY (en Fortran) ou celle de R. Zumkeller [55] (en Coq) en particulier. Dans ces travaux, la construction du modèle de Taylor d'une fonction g, qui sera composée avec une fonction f, en COSY ou en Coq, suppose que l'on dispose déjà du modèle de Taylor de f. Tout d'abord, le développement en série de la fonction g est déterminé, puis un polynôme en $\bar{f}$ (la fonction f sans sa partie constante) est obtenu. Le modèle de Taylor de $g \circ f$ est alors construit en utilisant récursivement le modèle de Taylor de f et les modèles de Taylor de l'addition et de la multiplication. De plus, l'implantation de R. Zumkeller en Coq est basée sur une implémentation constructive des nombres réels qui est aussi utilisée pour représenter les valeurs lors de l'évaluation des modèles de Taylor.

Notre développement vise à réduire le nombre d'opérations, coûteuses en PVS. Pour cela, nous avons déterminé des formules, qui sont plus complexes à obtenir que celles de COSY par exemple, mais qui limitent la profondeur des appels récursifs. Au final, nous compensons le surcoût d'effort pour mettre au point les formules par le fait que PVS effectue moins de calculs pour construire et évaluer les modèles de Taylor.

Chapitre 5

Modèles de Taylor avancés en PVS

L'objectif de ce chapitre est de montrer comment nous avons simplifié et automatisé l'utilisation de nos travaux en PVS. Dans la première partie, nous montrons comment utiliser PVSio, l'interpréteur de PVS, pour évaluer un modèle de Taylor. Dans la deuxième partie, nous présentons les stratégies développées pour démontrer la propriété d'inclusion d'un modèle de Taylor construit à partir des opérations de base et des fonctions définies au chapitre 4. On présente ensuite une stratégie pour démontrer des inégalités entre expressions, en utilisant les modèles de Taylor.

Avec l'assistant de preuves PVS, il est possible non seulement de démontrer et de vérifier des théorèmes, mais aussi d'évaluer des fonctions et opérateurs définis dans les théories. Cette caractéristique de PVS est connue sous le nom de *ground evaluator* [56]. L'idée est de traduire et d'évaluer le code exécutable de la spécification en code Lisp. PVSio [57] est un paquetage qui étend l'interpréteur de PVS avec des caractéristiques de la programmation impérative et également avec une interface qui facilite son utilisation. Nous avons utilisé ce paquetage pour évaluer des modèles de Taylor et c'est ce que nous présentons ci-dessous.

5.1 Évaluation utilisant PVSio

Même si, pour l'utilisateur non averti des assistants de preuves, cela peut paraître une fonctionnalité qui va de soi, il n'en est rien et nous tenons à souligner que PVS dispose d'un interpréteur qui permet d'évaluer des expressions. Pour déterminer le modèle de Taylor d'une expression donnée, on peut donc utiliser PVSio, l'interpréteur de PVS. Pour l'utiliser il faut développer une théorie avec les fonctions à évaluer. Cet interpréteur calcule interactivement les coefficients du polynôme et l'intervalle.

```
examples : THEORY

 BEGIN

 IMPORTING tm_exp[5, 5, [|-1,1|]]

 cosh( t : tm ) : MACRO tm =
       (1/2)*(exp( t ) + exp( neg(t) ))

 sinh( t : tm ) : MACRO tm =
       (1/2)*(exp( t ) - exp( neg(t) ))

 tx : tm = cosh( 2 * 1/1000*tm_x) * sinh( 3 * 1/1000*tm_x )

 END examples
```

FIG. 5.1 – Exemple de théorie en PVS.

Nous illustrons ceci à l'aide de la théorie définie figure 5.1 que nous avons développée pour l'exemple. Cette théorie importe la théorie des modèles de Taylor de l'exponentielle avec le paramètre `N` (le degré du polynôme du modèle de Taylor) égal à 5, NE (l'ordre d'approximation de l'exponentielle) égal à 5 et `domInterval`, le domaine du polynôme du modèle de Taylor égal à $[-1, 1]$.

Nous avons défini dans la théorie `examples` les fonctions `sinh` et `cosh` qui évaluent les modèles de Taylor du sinus hyperbolique et cosinus hyperbolique appliqués à une fonction f, donnée par son modèle de Taylor t. Le modèle de Taylor `tm_x` est le modèle de Taylor de la fonction identité x, qui a comme polynôme $p := x$ et comme intervalle $I := [0, 0]$.

On définit `tx` comme le modèle de Taylor de l'expression

$$cosh\left(2 \cdot \frac{x}{1000}\right) \cdot sinh\left(3 \cdot \frac{x}{1000}\right).$$

La figure 5.2 montre les instructions permettant d'obtenir les parties polynomiale et intervalle du modèle de Taylor en utilisant PVSio. Les coefficients sont obtenus à partir des expressions `tx'P(0)`, `P(1)` jusqu'à `P(5)`. La partie intervalle est retournée par `tx'I`.

Pour récapituler, le modèle de Taylor pour notre expression est

$$3 \cdot \frac{x}{1000} + \frac{21}{2} \cdot \left(\frac{x}{1000}\right)^3 + \frac{521}{40} \cdot \left(\frac{x}{1000}\right)^5 + r$$

avec

$$r \in 5150892483 \cdot 10^{-28} \cdot [-1, 1].$$

```
<PVSio> tx‘P(0);
==>
0
<PVSio> tx‘P(1);
==>
3/1000
<PVSio> tx‘P(2);
==>
0
<PVSio> tx‘P(3);
==>
21/2000000000
<PVSio> tx‘P(4);
==>
0
<PVSio> tx‘P(5);
==>
521/40000000000000000
<PVSio> tx‘I;
==>
(# lb := -19966660037929209080778095595964694170499249884356754248912582792777246825769541627979310535210358464738763496047478702331322336437004695773022456032565137272401306723242233956386636433666858122000000000000000000000,
ub := 19966660037929209080778095595964694170499249884356754248912582792777246825769541627979310535210358464738763496047478702331322336437004695773022456032565137272401306723242233956386636433666858122000000000000000000000 #)
```

FIG. 5.2 – Trace de l'évaluation du modèle de Taylor de notre exemple.

5.2 Stratégie pour prouver l'inclusion dans les modèles de Taylor

Les stratégies permettent d'automatiser les preuves et de faciliter l'utilisation de l'assistant de preuves. Dans l'esprit des preuves invisibles, on a développé dans un premier temps une stratégie pour démontrer la propriété d'inclusion d'une expression dans un modèle de Taylor. Une stratégie PVS est écrite en langage Lisp en utilisant des instructions et structures définies en PVS, de façon à ne pas introduire d'incohérences dans l'assistant de preuves [58, 59].

Avant de détailler les stratégies que nous avons développées, nous allons expliquer les stratégies `branch`, `then` et `then@` définies au préalable dans PVS et qui sont utilisées ultérieurement. La stratégie `branch` prend deux paramètres, `step` (qui est une commande de preuve) et `steplist` (qui est une liste de commandes de preuve). Elle applique `step` une première fois et ensuite elle applique la i-ème commande de la liste `steplist` au i-ème sous-but produit par l'application de `step`. S'il y a plus de sous-buts que d'éléments dans la liste `steplist`, la dernière commande de la liste est appliquée à tous les sous-buts restants. La stratégie `then` a un seul paramètre, qui est une liste de commandes `steps`. Elle exécute la commande qui est le premier élément de cette liste `steps` et, pour chaque sous-but généré par cette exécution, elle applique récursivement la stratégie avec le reste des éléments de la liste comme paramètre. La stratégie `then@` a comme paramètre une liste de commandes `steps` et elle exécute séquentiellement chacune de ces commandes, afin d'avancer dans la démonstration principale.

On a créé une première stratégie `containment` pour démontrer l'inclusion d'une expression dans un modèle de Taylor, en utilisant les théorèmes d'inclusion. La stratégie examine récursivement l'expression et applique à chaque fois un théorème approprié. Pour appliquer la stratégie on a besoin de savoir construire le modèle de Taylor et de faire en sorte que la propriété d'inclusion pour les constantes, les variables, et la puissance (voir fig. 5.3) soit également connue du système.

Pour illustrer le procédé, nous allons effectuer pas à pas la démonstration de l'inclusion du sinus hyperbolique $\sinh x$ dans son modèle de Taylor :

```
foo1 : LEMMA containment(
       LAMBDA (x : domIntervalType) :
           (1/2)*(exp( x ) - exp( -(x) )),
           (1/2)*(exp( tm_x ) - exp( neg(tm_x) )) )
```

Pour le démontrer, on cherche l'opérateur principal de l'expression. Dans notre exemple, l'opérateur principal est la multiplication par un scalaire : le scalaire $c = 1/2$ multiplie le modèle de Taylor $t = (\exp(tm\_x) - \exp(-(tm\_x)))$. Une fois l'opérateur déterminé, on sélectionne le nom du théorème d'inclusion à appliquer, dans ce cas il s'agit de `tm_scal_lem`. On marque la formule à laquelle on applique le théorème avec le nom

```
tm_k(k : real) : tm =
    (# P := LAMBDA(n : nat) : IF n=0 THEN k ELSE 0 ENDIF,
       I := [|0|] #)

tm_k_lem : LEMMA
    containment(LAMBDA(x : domIntervalType) : c, tm_k(c))

tm_x : tm =
    (# P := LAMBDA(n : nat) : IF n=1 THEN 1 ELSE 0 ENDIF,
       I := [|0|] #)

tm_x_lem : LEMMA
   containment(LAMBDA(x : domIntervalType) : x, tm_x)

pow(t, n) : RECURSIVE tm =
  IF    n = 0 THEN tm_k(1)
  ELSIF n = 1 THEN t
  ELSE  t*pow(t, n-1)
  ENDIF
  MEASURE n

tm_pow_lem : LEMMA
   containment(f, t) IMPLIES
   containment(LAMBDA(x : domIntervalType) : f(x)^n, pow(t, n))
```

FIG. 5.3 – Modèles de Taylor et propriété d'inclusion pour une constante, une variable et la puissance.

« TM:C ». Ensuite, on instancie les variables du théorème tm_scal_lem et on simplifie en utilisant la logique propositionnelle. Le nouveau but est de démontrer l'inclusion de $exp(x) - exp(-(x))$ dans t :

```
containment(LAMBDA (x_1 : domIntervalType) :
                  (exp(x_1) - exp(-x_1)),
                (exp(tm_x) - exp(neg(tm_x))))
```

Pour démontrer le nouveau but, on utilise les mêmes pas qu'auparavant, mais l'opérateur principal de l'expression est désormais la soustraction et le théorème d'inclusion à appliquer est tm_sub_lem. Après simplification, il y a deux nouveaux sous-buts :

```
containment(LAMBDA (x : domIntervalType) : exp(x), exp(tm_x))
```

et

```
containment(LAMBDA (x : domIntervalType) : exp(-x),
            exp(neg(tm_x)))
```

En appliquant récursivement les mêmes pas, le premier but est démontré en utilisant la propriété d'inclusion de l'exponentielle (théorème tm_exp_lem) et de l'identité (théorème tm_x_lem). Le deuxième but est démontré en utilisant la propriété d'inclusion de l'exponentielle, de l'opposé (théorème tm_neg_lem) et de l'identité.

Ces différentes étapes sont automatisées dans une stratégie. Cette stratégie est composée de la fonction containment-id, d'une stratégie privée containment__ et de la stratégie containment. Nous allons tout d'abord commenter ligne par ligne la stratégie containment donnée figure 5.5 lignes 36 à 45. Elle a comme paramètres fnum le numéro de la formule à laquelle on applique la stratégie (par défaut 1), defs la liste des noms des fonctions à remplacer par leurs définitions (par défaut nul) et enfin nonrec? qui indique si l'application de la stratégie est récursive ou non (par défaut elle l'est). Ensuite on va décrire le reste de la stratégie. La commande skosimp (ligne 38) invoque la stratégie skolemize qui simplifie le séquent pour supprimer des quantifications universelles s'il y en a. Ensuite, la stratégie containment utilise les définitions defs (ligne 39) et elle applique la stratégie containment__ avec les paramètres fnum (ligne 40).

La stratégie containment__ (lignes 21 à 34) est chargée de la démonstration récursive de la propriété d'inclusion. Elle prend deux paramètres, le premier fnum est le numéro de la formule à laquelle est appliquée la stratégie (par défaut 1), et le deuxième paramètre nonrec? indique si elle s'applique récursivement ou non (par défaut elle est récursive). La première chose est d'assigner à la variable fexpr la formule à laquelle s'applique la stratégie (ligne 22). La variable ok contient le résultat de la vérification que la formule existe (fexpr est non nul) et qu'il s'agit d'une formule containment (lignes 23 et 24). Si cette vérification est concluante, la fonction containment-id (donnée fig. 5.4 et commentée ultérieurement) obtient le nom du théorème à appliquer et crée une variable lem associant ce théorème et fexpr, le modèle de Taylor auquel il faudra l'appliquer (ligne 26).

On marque la formule avec l'étiquette « `TM:C` » (ligne 28). Les lignes 29 à 31 correspondent à l'application du théorème (ligne 29), l'instanciation de ses variables (ligne 30) et la simplification en utilisant la logique propositionnelle (ligne 31). Enfin la formule est occultée (ligne 32). Si jamais la stratégie est récursive, l'appel récursif est effectué (ligne 33).

À partir d'une expression du modèle de Taylor, la fonction `containment-id` (fig. 5.4) retourne le nom du théorème d'inclusion à appliquer. Elle reçoit comme paramètre l'expression `expr`. L'expression peut être de deux types : un nom simple ou une application. Si l'expression est un nom simple (ligne 2) (tel que le modèle de Taylor d'une variable `tm_x`) alors le nom du théorème d'inclusion associé est formé par le symbole de l'expression suivi de `_lem` (ligne 3) ; par exemple pour le modèle de Taylor d'une variable, le nom est `tm_x_lem`. Si l'expression est une application et si l'opérateur est un nom simple (lignes 4 et 5) la variable `op` prend pour valeur le symbole de l'opérateur (ligne 6) et le nom du théorème d'inclusion associé correspondant à l'opérateur est construit. Le nom du théorème est de la forme `tm_~a_lem` où `~a` correspond à l'opérateur (lignes 8-20), par exemple si l'opérateur est `+`, `~a` est `add` ; si l'opérateur est `-`, `~a` est `sub` etc.

Pour illustrer l'utilisation de la stratégie `containment`, on rappelle que l'on a prouvé la propriété d'inclusion dans leurs modèles de Taylor pour les sinus et cosinus hyperboliques :

```
sinh( x : tm ) : tm = (1/2)*(exp( x ) - exp( neg(x) ))

cosh( x : tm ) : tm = (1/2)*(exp( x ) + exp( neg(x) ))

tm_sinh_enclosure : LEMMA containment( f, t)
   IMPLIES containment( (LAMBDA xu : sinh(f(xu))), sinh(t))

tm_cosh_enclosure : LEMMA containment( f, t)
   IMPLIES containment( (LAMBDA xu : cosh(f(xu))), cosh(t))
```

Avant de détailler la stratégie pour certifier des inégalités nous rappelons, en conclusion, que nous avons développé la stratégie `containment` qui démontre la propriété d'inclusion pour des fonctions construites à partir des opérations de base et des fonctions que nous avons développées. Nous avons utilisé cette stratégie pour démontrer la propriété d'inclusion pour les sinus et cosinus hyperboliques. La stratégie `containment` est utilisée par la stratégie pour certifier des inégalités que nous présentons dans la suite.

5.3 Stratégie pour certifier des inégalités

Afin de permettre de prouver facilement des inégalités ou des bornes sur des expressions, ce qui est d'un usage général, nous avons également mis au point une stratégie

```
(defun containment-id (expr)
 (or (and (name-expr? expr)
       (format nil "~a_lem" (id expr)))
     (and (application? expr)
       (name-expr? (operator expr))
        (let ((op (id (operator expr))))
          (format nil "tm_~a_lem"
            (cond ((equal op '|tm_k|) "k")
              ((equal op '|+|) "add")
              ((equal op '|-|) "sub")
              ((and (equal op '|*|)
               (equal (id (print-type (type (args1 expr))))
                   'tm)
               (equal (id (print-type (type (args2 expr))))
                   'tm))
                    "mult")
               ((equal op '|*|)
                "scal")
                ((equal op '|/|) "div")
                 (t op)))))))
```

FIG. 5.4 – Fonction `containment-id`.

```
21 (defstrat containment__ (&optional (fnum 1) nonrec?)
22   (let ((fexpr (extra-get-formula fnum))
23     (ok     (and fexpr
24                  (equal (id (operator fexpr)) '|containment|))))
25     (when ok
26      (let ((lem (containment-id (args2 fexpr))))
27       (then@
28        (relabel "TM:C" fnum)
29         (lemma lem)
30           (inst? -1)
31           (then (bddsimp)
32           (hide "TM:C")
33           (if nonrec? (skip) (containment__)))))))
34     "[TMs] Internal strategy.")
35
36 (defstep containment (&optional (fnum 1) defs nonrec?)
37   (then@
38    (skosimp)
39    (expand-defs defs : id? t)
40    (containment__ fnum nonrec?))
41   "[TMs] Proves containment(f, t) in FNUM.
42    Expands DEFS if specified.
43    Unless NONREC? is t, the strategy
44    is recursively applied."
45   "Proving containment(f, t)")
```

FIG. 5.5 – Stratégie d'inclusion `containment`.

qui utilise, de façon plus ou moins cachée pour l'utilisateur, des modèles de Taylor. Cette stratégie permet de certifier des inégalités ou des bornes sur des expressions. Quand on utilise un assistant de preuves pour démontrer une inégalité, il peut être nécessaire de guider l'assistant pas à pas dans la démonstration. Pour cette raison, les utilisateurs effectuent rarement la démonstration. Donc simplifier la façon de prouver les inégalités et les bornes d'expressions facilite l'utilisation de PVS.

La stratégie s'utilise de manière interactive et l'utilisateur peut guider la démonstration (avec des paramètres) en augmentant l'ordre du modèle de Taylor (plus l'ordre est élevé et plus l'erreur d'approximation est faible) ou en divisant le domaine de l'intervalle d'entrée (plus l'intervalle d'entrée est petit et plus le reste est précis). Si la stratégie permet à PVS de démontrer l'inégalité, la preuve est terminée, sinon elle reste dans l'état atteint avant d'être bloquée. L'utilisateur peut mettre à profit cette information pour relancer la preuve, en ajustant les paramètres de la stratégie. Nous n'en sommes pas arrivés à cacher complètement l'utilisation des modèles de Taylor et il faut encore que l'utilisateur précise les paramètres relatifs à leur utilisation (l'ordre du modèle de Taylor et le nombre de subdivisions de l'intervalle d'entrée). Toutefois, pour employer la stratégie, il n'est pas nécessaire que l'utilisateur ait connaissance des modèles de Taylor ou de leur construction en PVS, ce qui permet des applications très générales.

Pour construire cette stratégie de certification des inégalités, on a besoin de la fonction `expr2tm` (fig. 5.6) qui construit le modèle de Taylor correspondant à l'expression $e(x)$ passée en paramètre. La fonction prend comme paramètres l'expression `expr`, la variable du modèle de Taylor `var`, le domaine de la variable `dom` et l'ordre du modèle de Taylor `order`. La construction se réalise récursivement sur la structure de l'expression :

- si l'expression est la variable `var` (lignes 2 à 5), le modèle de Taylor associé est `tm_x[order, dom]` (ligne 5) ;
- si l'expression est un nombre rationnel (lignes 6 et 7), le modèle de Taylor est celui d'une constante `tm_k[order, dom](expr)` (ligne 7) ;
- si l'expression est une application (lignes 8 à 40), on assigne l'opérateur à la variable `op` et on procède au cas par cas :
  - si l'expression est $-e$ (lignes 10 à 13), le modèle de Taylor est `neg(E)` où `E` est le modèle de Taylor de e construit récursivement (lignes 12 et 13) ;
  - si l'expression est e^n (lignes 15 à 23) on distingue le cas où e est un nombre rationnel, on construit alors le modèle de Taylor d'une constante (ligne 17 et 18). Dans tous les autres cas, on construit le modèle de Taylor de la puissance `pow(E, n)` où E est le modèle de Taylor de e construit récursivement (ligne 20) ;
  - si l'expression est $f(e)$ où f est l'une des fonctions `sq`, `sqrt`, `exp`, `atan`, `sin` (lignes 24 à 29), on construit récursivement le modèle de Taylor E de e (ligne 29) pour obtenir le modèle de Taylor du résultat $f(E)$;
  - si l'expression est $e_1 \diamond e_2$, $\diamond \in \{+, -, *, / \}$ (lignes 30 à 39) on distingue le cas où e_1 et e_2 sont des constantes. Nous avons inclus ce cas particulier pour traiter comme constante un nombre rationnel e_1/e_2. Le résultat est alors le modèle de

Taylor d'une constante (lignes 31 à 35). Dans tous les autres cas, le résultat est le modèle de Taylor de l'opération $\diamond$ de E_1 et E_2 obtenus récursivement à partir de e_1 et e_2 ;

– dans tous les autres cas, l'expression est considérée comme une constante (lignes 40 et 41).

Pour prouver une expression de la forme $x \in J \vdash e(x) \diamond K$ où $\diamond \in \{<, \leq, >, \geq, \in\}$, en utilisant les modèles de Taylor d'ordre N (stratégie `taylormodels`) on effectue les étapes suivantes :

1. On construit le modèle de Taylor E d'ordre N pour $e(x)$ en utilisant la fonction `expr2tm` (fig. 5.6).
2. On prouve $\texttt{containment}(e, E)$ utilisant la stratégie `containment`, on a donc :
$$e(x) - E\text{`}P(x) \in E\text{`}I. \tag{5.1}$$
3. On prouve l'inclusion $E\text{`}P(x) \in \texttt{Taylor}(E\text{`}P, \texttt{point}(J))$ en utilisant le théorème `Taylor_sharp`. On utilise la fonction $\texttt{Taylor}(p, a)$, qui retourne le développement de Taylor avec coefficients intervalles du polynôme p autour de a, afin d'introduire le point de référence du polynôme de Taylor.
La fonction $\texttt{point}(J)$, passée en paramètre, retourne le point de référence a appartenant au domaine J autour duquel s'effectue le développement de Taylor, par défaut on utilise la fonction `midpoint` qui retourne le milieu de l'intervalle J.
Un bonne sélection du point de référence peut minimiser la largeur de l'intervalle de sortie [40, 55]. On a donc
$$e(x) \in E\text{`}I + \texttt{Taylor}(E\text{`}P, \texttt{point}(J)). \tag{5.2}$$
4. En utilisant l'arithmétique par intervalles on prouve :
$$E\text{`}I + \texttt{Taylor}(E\text{`}P, \texttt{point}(J)) \diamond K \tag{5.3}$$
où $\diamond$ est l'opérateur de comparaison.
5. En utilisant un théorème pour cet opérateur $\diamond$ (défini dans la bibliothèque de l'arithmétique par intervalles), on obtient la conclusion souhaitée
$$e(x) \diamond K.$$

Dans ce qui suit, nous allons commenter ligne par ligne la stratégie `taylormodels` donnée figures 5.7 et 5.8. Elle a comme paramètres `var` la variable de l'expression, `dom` le domaine de l'expression (si ce domaine est `nul` la stratégie recherche dans le séquent l'antécédent $\texttt{var} \in J$ pour obtenir le domaine J), `fnum` le numéro de la formule à laquelle on applique la stratégie (par défaut 1), `order` l'ordre du modèle de Taylor à utiliser (par défaut 2), `point` la fonction `point` expliquée pour l'étape 3 qui précède (par défaut `midpoint`), `step` la stratégie à utiliser pour évaluer l'équation 5.3 (par défaut `eval-formula`).

```
(defun expr2tm (expr var dom order)
  (cond ((and (name-expr? expr)
              (string= (format nil "~a" (id expr))
                       var))
              (format nil "tm_x[~a, ~a]" order dom))
        ((rational-expr? expr (list (list var)) nil nil)
         (format nil "tm_k[~a, ~a](~a)" order dom expr))
        ((application? expr)
         (let ((op (id (operator expr))))
           (cond ((and (equal op '|-|)
                       (is-prefix-operator expr '-))
                  (format nil "neg(~a)"
                          (expr2tm (args1 expr) var dom order)))
                 ((equal op '|^|)
                  (cond ((rational-expr? (args1 expr)
                                         (list (list var)) nil nil)
                         (format nil "tm_k[~a, ~a](~a)"
                                 order dom expr))
                         (t (format nil "pow(~a, ~a)"
                                    (expr2tm (args1 expr) var dom order)
                                    (args2 expr)
                         ))
                   ))
                 ((member op '(sq sqrt exp atan sin))
                  (format nil "~a[~a, ~a](~a)"
                              op
                              order
                              dom
                              (expr2tm (args1 expr) var dom order)))
                 ((member op '(+ - * /))
                  (cond ((and (rational-expr? (args1 expr)
                                              (list (list var)) nil nil)
                              (rational-expr? (args2 expr)
                                              (list (list var)) nil nil))
                         (format nil "tm_k(~a)" expr))
                        (t (format nil "(~a ~a ~a)"
                                   (expr2tm (args1 expr) var dom order)
                                   op
                                   (expr2tm (args2 expr) var dom order))))
                 (t (format nil "tm_k(~a)" expr)))))
        (t (format nil "tm_k(~a)" expr))))
```

FIG. 5.6 – Fonction qui à partir d'une expression construit son modèle de Taylor.

Tout d'abord, on assigne à la variable `fexpr` la formule à laquelle s'applique la stratégie (ligne 7). La variable `mysharp` contient la formule $x \in J$, si elle se trouve dans les antécédents, sinon elle contient `nul` (lignes 8 à 10). La variable `error` contient le résultat du test qui détermine s'il s'agit d'un domaine vide (`nul`) ou d'une formule quantifiée universellement (lignes 11 à 13). Dans ce cas, la stratégie ne peut pas être appliquée et la stratégie se termine en renvoyant un message d'erreur à l'utilisateur (lignes 14 à 16). Dans les cas restants, la stratégie se poursuit.

La variable `mydom` contient le domaine `dom` s'il est donné, sinon il contient le domaine J stocké dans la variable `mysharp` (ligne 17). La variable `lem` contient le théorème à utiliser pour prouver la conclusion $e(x) \diamond k$ en utilisant les équations 5.2 et 5.3 (ligne 18). La variable `myf` contient la fonction $\lambda x : J.e(x)$ nécessaire pour le premier argument du prédicat `containment` (lignes 19 et 20). La variable `myop` contient l'opérateur correspondant à l'opérateur de comparaison $\diamond$ de l'équation 5.3 (lignes 21 à 23). La variable `mytm` contient le modèle de Taylor E construit à partir de $e(x)$ en utilisant la fonction `expr2tm` (fig. 5.6) détaillée auparavant. La variable `contstr` contient le prédicat `containment`$(\lambda x : J.e(x), E)$ (lignes 25 et 26).

Reprenons, dans la stratégie `taylormodels`, la partie qui effectue la preuve (lignes 27 à 56). On utilise la commande `case` pour introduire comme antécédent le prédicat `containment` (ligne 27) que l'on prouve ultérieurement, en utilisant la stratégie `containment` (voir §5.2), à la ligne 56. Lignes 28 à 37 sont définies les variables `mytay` qui vaut `Taylor`$(E\text{`}P, \texttt{point}(J))$ (lignes 28 et 29), `polystr` qui vaut $E\text{`}P(x) \in$ `Taylor`$(E\text{`}P,$ `point`$(J))$ (lignes 30 à 32) et `tmstr` qui vaut $E\text{`}I +$ `Taylor`$(E\text{`}P,$ `point`$(J)) \diamond k$ (lignes 33 à 37). Lignes 39 et 40, on utilise la définition du prédicat `containment` pour obtenir $e(x) - E\text{`}P \in E\text{`}I$ ; ligne 41, on introduit $E\text{`}P(x) \in$ `Taylor`$(E\text{`}P, \texttt{point}(J))$ qui est démontré plus tard, lignes 54 et 55, en utilisant le théorème `Taylor_sharp`. On utilise le théorème `sub2add` de l'arithmétique par intervalles pour obtenir l'équation 5.2, à savoir $e(x) \in E\text{`}I +$ `Taylor`$(E\text{`}P, \texttt{point}(J))$ (lignes 43 à 46). Ensuite on introduit comme antécédent l'équation 5.3 : $E\text{`}I +$ `Taylor`$(E\text{`}P, \texttt{point}(J)) \diamond k$ (ligne 47) qui se démontre par évaluation en utilisant l'arithmétique par intervalles (ligne 53). On finit la preuve en utilisant le théorème défini dans la variable `lem` qui devient, après instanciation de ses variables (lignes 50 et 51) :

$$e(x) \in E\text{`}I + \texttt{Taylor}(E\text{`}P, \texttt{point}(J)) \ \wedge \ E\text{`}I + \texttt{Taylor}(E\text{`}P, \texttt{point}(J)) \diamond k \Longrightarrow e(x) \diamond k$$

et qui utilise les équations 5.2 et 5.3 pour obtenir la conclusion (lignes 49 à 52).

Comme nous l'avons remarqué au §1.2.4, on peut améliorer la précision de l'intervalle de sortie en divisant le domaine de la variable du modèle de Taylor. La stratégie `taylors`, donnée figure 5.9, combine la bissection de l'intervalle d'entrée avec la stratégie `taylormodels`. La stratégie a les mêmes paramètres que la stratégie `taylormodels`, plus le paramètre `split` (ligne 3) qui est le nombre d'intervalles `ni` en lesquels la stratégie doit partitionner le domaine J (par défaut J n'est pas découpé). Les premières étapes de la stratégie (lignes 8 à 18) sont quasiment les mêmes que celles de la stratégie `taylormodels` (fig. 5.7, lignes 7 à 17), il s'agit de la vérification des

```
(defstep taylormodels (var &optional
                           dom
                           (fnum 1)
                           (order 2)
                           (point "midpoint")
                           (step (eval-formula)))
  (let ((fexpr    (extra-get-formula fnum))
        (mysharp  (cdar (find-sharp
                  '_
                  #'(lambda (expr) (id-sharp? expr var)))))
        (error    (or (and (null dom)
                           (null mysharp))
                      (forall-expr? fexpr))))
    (if error
        (skip-msg "Strategy taylormodels cannot
                  be applied to this sequent")
      (let ((mydom   (or dom (args2 mysharp)))
            (lem     (interval-lemma (id (operator fexpr))))
            (myf     (format nil "LAMBDA(~a : inInterval(~a)) : ~a"
                             var mydom (args1 fexpr)))
            (myop    (if (equal (id (operator fexpr)) "##")
                         "<<"
                       (id (operator fexpr))))
            (mytm    (expr2tm (args1 fexpr) var mydom order))
            (contstr (format nil "containment(~a, ~a)"
                             myf mytm)))
        (branch (case contstr)
                ((let ((mytay (format nil "Taylor(~a'P, (~a)(~a))"
                                      mytm point mydom))
                       (polystr
                        (format nil "polynomial(~a'P, ~a)(~a) ## ~a"
                                mytm order var mytay))
                       (tmstr
                        (format
                         nil "~a'I + ~a ~a ~a"
                         mytm mytay myop (args2 fexpr)))
                       )
                   (then@
                    (expand "containment" :assert? none)
                    (inst -1 var)
```

FIG. 5.7 – Stratégie `taylormodels`.

```
41                          (branch (case polystr)
42                                  ((then@
43                                    (lemma "sub2add")
44                                    (inst?)
45                                    (inst?)
46                                    (bddsimp)
47                                    (branch (case tmstr)
48                                            ((then@
49                                              (lemma lem)
50                                              (inst? : where +)
51                                              (inst?)
52                                              (bddsimp))
53                                             step)))
54                                   (then (rewrite "Taylor_sharp")
55                                         step)))))
56                        (containment mydom order))))))
57     "[TMs] Proves exp ## EXP via Taylor's Models"
58     "Proving exp ## EXP via Taylor's Models")
```

FIG. 5.8 – Stratégie `taylormodels` (suite et fin).

paramètres en entrée. S'il ne faut pas découper le domaine (ligne 19), on applique directement la stratégie `taylormodels` avec les paramètres donnés (ligne 20). Dans le cas contraire, on utilise la stratégie `splitint` (lignes 21 et 22), qui est définie dans la bibliothèque d'intervalles de PVS et qui démontre que si $x \in J$, alors $x \in \bigcup_{i=1}^{ni} J_i$ et qui, en outre, crée `ni` sous-buts, chacun de ces sous-buts ayant pour antécédent $x \in J_i$ où x est la variable `var` et J_i est le i-ème intervalle de la partition. Pour chacun des ces sous-buts, on utilise la stratégie `taylormodels` sans donner l'intervalle du domaine pour qu'elle le cherche dans les antécédents.

Figure 5.10, nous présentons deux exemples de l'utilisation de la stratégie `taylors`. Dans le premier exemple on démontre, en utilisant un modèle de Taylor d'ordre 2, que si $x \in [0, 1]$ alors $x \cdot (1 - x) \in [0, 1/4]$. Ce résultat est l'intervalle de largeur minimale. Si on utilise seulement l'arithmétique par intervalles, on obtient $x \cdot (1-x) \in [0, 1]$, ce qui est beaucoup plus large. Dans le deuxième exemple on démontre, en utilisant un modèle de Taylor d'ordre 5 et en effectuant 2 bissections du domaine de x, que si $x \in [0, 1]$, alors $\exp(x) - 1 - x - x^2 \in [-36/100, 12/105]$. Si on ne partitionne pas l'intervalle, la stratégie n'arrive pas à le démontrer.

5.4 Conclusion

Dans ce chapitre nous avons montré comment utiliser l'interpréteur de PVS pour évaluer le modèle de Taylor d'une expression. Nous avons présenté la stratégie que

```
(defstep taylors (var &optional
                      dom
                      (split 1)
                      (fnum 1)
                      (order 2)
                      (point "midpoint")
                      (step (eval-formula)))
  (let ((fexpr    (extra-get-formula fnum))
        (mysharp  (cdar (find-sharp
                         '_
                         #'(lambda (expr) (id-sharp? expr var)))))
        (error    (or (and (null dom)
                           (null mysharp))
                      (forall-expr? fexpr))))
    (if error
        (skip-msg "Strategy taylors cannot
                   be applied to this sequent")
      (let ((mydom   (or dom (args2 mysharp))))
        (if (equal split 1)
            (taylormodels var mydom fnum order point step)
          (let ((ssplit '(splitint ((var, mydom, split)))))
            (then ssplit
                  (taylormodels var
                                :fnum fnum :order order :point point
                                :step step)))))))
  "[TMs] Proves exp ## EXP via splitting and Taylor's Models"
  "Proving exp ## EXP via splitting and Taylor's Models")
```

FIG. 5.9 – Stratégie `taylors`.

```
test : THEORY

 BEGIN

 IMPORTING taylor_model, tm_exp

 x : VAR real

 test0 : LEMMA
   x   ## [|0, 1|] IMPLIES
   x * (1 - x) ## [| 0, 1/4 |]
 %|- test0 : PROOF (then (skosimp) (taylors "x")) QED

 test1 : LEMMA
    x ## [|0, 1|] IMPLIES
    exp(x)  - 1 - x - sq(x) ## [|-36/100, 12/105|]
 %|- test1 : PROOF (then (skosimp)
 %                 (taylors "x" :split 2 :order 5)) QED

 END test
```

FIG. 5.10 – Exemples d'utilisation de la stratégie `taylors`.

nous avons développée pour démontrer la propriété d'inclusion et nous l'avons utilisée pour prouver cette propriété `containment` pour les sinus et cosinus hyperboliques. Nous avons également développé une stratégie pour certifier des inégalités ou des bornes sur des expressions. Pour prouver une inégalité en utilisant les modèles de Taylor, il suffit d'utiliser la stratégie `taylors`. Cette stratégie permet interactivement d'augmenter l'ordre du modèle de Taylor (plus l'ordre est élevé et plus l'erreur d'approximation est faible) ou de diviser le domaine de l'intervalle d'entrée (plus l'intervalle d'entrée est petit et plus le reste est précis). Si la stratégie permet à PVS de démontrer l'inégalité, la preuve est terminée, sinon elle reste dans le séquent qui n'est pas démontré. L'utilisateur peut utiliser cette information pour relancer la preuve, en ajustant les paramètres de la stratégie. Cette dernière stratégie ne masque pas totalement à l'utilisateur la technique employée pour démontrer l'inégalité demandée et il reste encore du travail à faire pour rendre totalement invisible l'emploi des modèles de Taylor.

Chapitre 6

Applications

Dans ce chapitre nous présentons deux applications utilisant les modèles de Taylor que nous avons implantés. Il s'agit dans les deux cas de la certification de l'erreur d'approximation par un polynôme, respectivement pour les fonctions arc-tangente et exponentielle.

Les deux applications présentées dans ce chapitre sont issues de travaux dans lesquels l'équipe Arénaire est impliquée, ou encore auxquels Daumas est associé. Dans les deux cas, même si les contextes sont très différents, le problème mathématique est le même : une fonction est approchée par un polynôme, sur un domaine donné, et il s'agit de garantir que l'erreur d'approximation est inférieure à un seuil donné.

La première application est directement liée aux travaux d'Arénaire : pour implanter en arithmétique flottante une fonction élémentaire f, on cherche une approximation p de cette fonction sur un petit domaine et on évalue ce polynôme en arithmétique flottante, ce qui implique une erreur d'arrondi. L'erreur totale est majorée par la somme de l'erreur d'approximation et de l'erreur d'arrondi (on ne mentionne pas ici les erreurs liées à la réduction d'argument, qui permet de se ramener au petit domaine, ni à la transformation inverse). On sait de mieux en mieux automatiser le calcul d'un majorant pour l'erreur d'arrondi [9], mais pas encore majorer l'erreur d'approximation. Or savoir majorer l'erreur totale est indispensable si l'on veut certifier que l'on calcule bien l'arrondi correct de l'évaluation de ladite fonction élémentaire.

La seconde application provient d'un algorithme de détection et de résolution des conflits aériens [60, 61]. Cet algorithme utilise une transformation des coordonnées géodésiques en coordonnées cartésiennes, localement, qui fait intervenir la fonction arctangente. Pour implanter à moindre coût, sur un système embarqué, cette fonction trigonométrique, on l'approche par un polynôme « simple ». Pour pouvoir prouver en PVS la correction de cet algorithme de détection et de résolution des conflits aériens, il est donc nécessaire de savoir majorer l'erreur d'approximation.

```
ln(x: posreal): real =
      Integral(1,x, (LAMBDA (t: posreal): 1/t))

exp(x): { py | x = ln(py)}
exp_def        : LEMMA exp = inverse(ln)
```

FIG. 6.1 – Définition en PVS de $\ln$ et $\exp$.

6.1 Approximation de la fonction exponentielle, pour savoir l'évaluer avec arrondi correct

L'un des axes de recherche de l'équipe Arénaire est l'évaluation avec arrondi correct des fonctions élémentaires ($\sin$, $\tan$, $\exp$, $\ln$, $\arccos$, $\sinh$, $\operatorname{arccoth}$ ...). Il s'agit de déterminer comment implanter des algorithmes permettant d'évaluer ces fonctions sur ordinateur, pour les précisions usuelles (simple et double précision IEEE-754 [62, 63]) et au moindre coût. L'exigence d'arrondi correct signifie que pour chaque fonction f et pour chaque argument x, on retourne la valeur correspondant à $f(x)$ calculée avec une précision infinie et arrondie selon le mode d'arrondi désiré. L'approche actuelle consiste à ramener l'argument x dans un petit domaine, à évaluer la fonction dans ce petit domaine puis à revenir à la valeur initiale de l'argument ; nous ne détaillerons pas ces phases de réduction d'argument et de reconstruction. Dans le petit domaine considéré, on détermine une approximation polynomiale p de la fonction à évaluer, puis on évalue le polynôme p au point donné. En effet, pour évaluer une fonction telle que l'exponentielle sur ordinateur, on n'utilise pas sa définition mathématique parce que cette définition, qui est abstraite, ne permet pas de l'évaluer directement. Par exemple en PVS la définition de l'exponentielle, qui est donnée figure 6.1, ne peut pas être utilisée pour l'évaluation.

C'est pour cette raison que l'on utilise une approximation polynomiale de la fonction. L'objectif étant d'obtenir une valeur assez précise pour que l'on puisse garantir que la valeur retournée est bien l'arrondi correct de la valeur exacte, on doit garantir que l'approximation utilisée est satisfaisante. Pour obtenir une telle garantie, il faut d'une part obtenir un polynôme avec une erreur d'approximation très faible et garantie, et d'autre part mettre au point un schéma d'évaluation en arithmétique flottante offrant une erreur d'arrondi également très faible et garantie. L'erreur totale est alors majorée par la somme des bornes sur ces deux erreurs.

La détermination d'un « bon » polynôme d'approximation et la partie concernant l'arithmétique flottante (schéma d'évaluation, précision de calcul, établissement et preuve des bornes sur les erreurs d'arrondi) font l'objet des thèses de doctorat de Sylvain Chevillard, Christoph Lauter et Guillaume Melquiond.

Jusqu'à présent, pour obtenir une garantie sur l'erreur d'approximation, nous en étions réduits à faire confiance à un logiciel de calcul numérique puis à mener les calculs

à la main pour vérifier ces résultats numériques. Nous pouvons utiliser les modèles de Taylor pour certifier que l'erreur commise entre la fonction f et le polynôme p est bornée par une valeur v : pour $x \in \boldsymbol{x}, |f(x) - p(x)| \leq v$, de manière équivalente on montre que $f(x) - p(x) \in [-v, v]$. Plus la valeur de v est petite, meilleure est l'approximation de la fonction par le polynôme.

Nous montrons ci-dessous l'exemple du calcul de l'exponentielle en simple précision. Une approximation polynomiale de l'exponentielle, sur l'intervalle $[-1/64, 1/64]$, est

$$p(x) = 1 + x + \frac{524297}{1048576} \cdot x^2 + \frac{349529}{2097152} \cdot x^3$$

pour les besoins du calcul en simple précision, nous voulons certifier que l'erreur d'approximation est inférieure à $5 \cdot 10^{-10} = \frac{1}{2000000000}$, c'est-à-dire

$$|e^x - p(x)| \leq \frac{1}{2000000000}$$

ou de façon équivalente

$$e^x - p(x) \in \left[-\frac{1}{2000000000}, \frac{1}{2000000000} \right]. \tag{6.1}$$

Nous avons obtenu les résultats récapitulés dans le tableau suivant. La première colonne indique l'ordre du modèle de Taylor utilisé, la seconde indique en combien de sous-intervalles a été découpé l'intervalle étudié, la troisième colonne indique si la preuve a pu être terminée ou non, la quatrième colonne donne le nombre de sous-intervalles sur lesquels la borne a été vérifiée (dans le cas où la preuve a été terminée, ce nombre est égal au nombre de sous-intervalles) et la dernière colonne indique le temps de calcul, en secondes, sur une machine Intel Pentium 4 de 3GHz et 1GB de mémoire :

Ordre	Partitions	Prouvée	Intervalles Prouvées	Temps (sec)
3	8	non	2	89.99
3	9	non	3	162.98
3	10	non	4	132.93
3	20	non	10	433.51
4	8	non	6	95.92
4	9	oui	9	172.56
4	10	oui	10	132.57
5	8	non	6	103.64
5	9	oui	9	180.51
5	10	oui	10	139.47
6	8	non	6	324.70
6	9	oui	9	428.22
6	10	oui	10	419.51

Nous ne pouvons pas démontrer la borne donnée en (6.1) en utilisant des modèles de Taylor d'ordre 3 et un partitionnement en au plus 20 sous-intervalles. Toutefois, avec les modèles de Taylor d'ordre 4, nous arrivons à prouver (6.1) avec un partitionnement du domaine d'entrée en 9 sous-intervalles. Les intervalles sur lesquels PVS ne parvient pas à établir la borne, en utilisant des modèles de Taylor d'ordre 4, 5 et 6 et un partitionnement en 8 sous-intervalles sont : $[-1/64, -3/256]$ et $[3/256, 1/64,]$. En utilisant cette information, nous avons essayé de prouver la borne (6.1) avec des modèles d'ordre 4 en découpant l'intervalle de départ en 3 sous-intervalles seulement, qui sont $[-1/64, -3/256]$, $[-3/256, 3/256]$ et $[3/256, 1/64,]$, pour voir s'il est possible d'effectuer la preuve avec un nombre inférieur de sous-intervalles. Nous avons alors eu besoin de 10 sous-intervalles : deux pour démontrer la borne sur $[-1/64, -3/256]$ qui était le premier intervalle considéré, six pour démontrer la borne sur $[-3/256, 3/256]$ qui était le deuxième, et enfin deux pour démontrer la borne sur $[3/256, 1/64,]$ qui était le troisième. Nous avons également essayé de démontrer la borne sur le deuxième intervalle avec des modèles de Taylor d'ordres 5 et 6, mais le nombre de sous-intervalles nécessaires reste le même, à savoir six.

On pourrait penser qu'en augmentant l'ordre du modèle de Taylor, on peut démontrer une inégalité en découpant en un nombre de sous-intervalles plus faible, mais cet exemple, avec des modèles de Taylor d'ordres 4, 5 ou 6 montre que ce n'est pas toujours vrai.

Nous avons également essayé d'utiliser le graphique de l'erreur d'approximation entre l'exponentielle et le polynôme donné (figure 6.2) pour découper l'intervalle d'entrée en 5 sous-intervalles qui sont $[-1/64, -13/1000]$, $[-13/1000, -5/1000]$, $[-5/1000, 5/1000]$, $[5/1000, 13/1000]$ et $[13/1000, 1/64]$, pour que chaque intervalle ne contienne qu'un seul extrémum local. Avec ce découpage, nous n'avons pas pu réduire le nombre d'intervalles à moins de 9 pour démontrer (6.1), en effet nous avons prouvé la borne sur les sous-intervalles $[-13/1000, -5/1000]$ et $[5/1000, 13/1000]$ en les re-découpant chacun en 3. Par contre le temps employé par PVS a été de 74.46 secondes (avec des modèles de Taylor d'ordre 4), soit moins de la moitié du temps nécessaire pour démontrer la borne avec le même nombre de sous-intervalles mais avec un découpage régulier. Ceci peut être aussi la raison pour laquelle le temps employé pour une démonstration avec un partitionnement en 10 sous-intervalles est inférieur au temps avec un partitionnement en 9 sous-intervalles.

6.2 Approximation de la fonction arc-tangente, pour un algorithme de résolution de conflits aériens

Cette application constitue une partie de la preuve d'un algorithme de détection et résolution de conflits aériens. Le contexte est celui du vol libre : le trafic aérien devenant de plus en plus intense, il deviendra bientôt impossible aux contrôleurs aériens de gérer complètement les trajectoires de tous les avions. Afin de les décharger de certaines

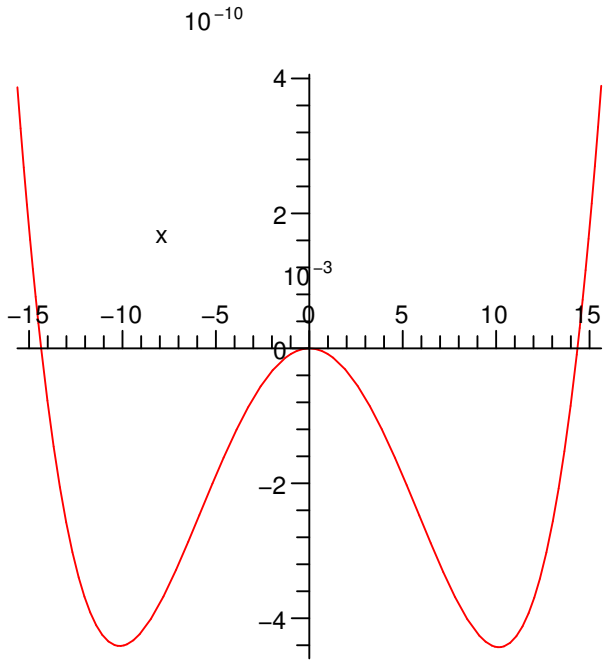

FIG. 6.2 – Graphique de $e^x - p(x)$, $x \in \left[-\frac{1}{64}, \frac{1}{64}\right]$

tâches, il est envisagé de laisser aux équipages des avions eux-mêmes le soin de définir leur trajectoire. Le problème considéré ici est celui de la résolution des conflits : on dit que deux avions sont en conflit s'ils s'approchent trop près l'un de l'autre. Il faut dans ce cas que l'un des avions, ou les deux, effectue une manœuvre d'évitement, dite de résolution du conflit. Un algorithme a été proposé et prouvé en PVS [60, 61]. Cet algorithme utilise la représentation des positions et des vitesses des avions dans un espace euclidien. Or la géométrie terrestre est celle d'un sphéroïde oblate et, étant données les vitesses des avions (de 500 à 1000 km/h) et les fenêtres de temps considérées (de l'ordre de 5mn), la prise en compte ou non de cette géométrie conduit à des différences de positions pouvant aller jusqu'à 1500m [64]. Une solution simple serait d'augmenter les distances de sécurité d'autant, cependant cela risque d'entraîner un nombre important de fausses alertes et par conséquent de manœuvres d'évitement inutiles. Une autre solution consiste à passer de cet espace euclidien à un espace plus conforme à la géométrie de la Terre et vice-versa. Une telle transformation couramment utilisée est le standard WGS84. Les formules permettant de calculer la distance géodésique entre deux avions, ou distance sur la surface de la Terre, font intervenir la fonction arc-tangente, que l'on approche sur l'intervalle $[-1/30, 1/30]$ par le polynôme :

$$q(x) = x - \frac{11184811}{33554432} \cdot x^3 + \frac{13421773}{67108864} \cdot x_5.$$

Ce polynôme [65] est inspiré de l'approximation disponible dans [66, p. 221], il présente l'avantage de ne faire intervenir que des coefficients dont le dénominateur est une puissance de 2, autrement dit ces coefficients sont exactement représentables par des nombres flottants.

Nous voulons démontrer que

$$\forall x \in \left[\frac{-1}{30},\ \frac{1}{30}\right],\ \mathsf{atan}(x) - q(x) \in \left[-\frac{1}{2^i}, \frac{1}{2^i}\right],$$

pour i entre 10 et 20. En utilisant la stratégie `taylors` (voir §5.3) avec des modèles de Taylor d'ordres 3, 4, 5 et 6 et sans partitionner l'intervalle d'entrée, nous obtenons les temps de démonstration suivants :

	temps (seconds)			
i	Ordre 3	Ordre 4	Ordre 5	Ordre 6
10	5.79	5.57	6.24	34.72
11	5.69	5.80	6.29	34.69
12	5.81	5.73	6.29	34.56
13	5.78	5.55	6.33	34.49
14	5.96	5.54	6.36	34.81
15	6.17	5.51	6.30	34.82
16	5.93	5.53	6.29	34.62
17	5.85	5.57	6.24	34.84
18	5.85	5.54	6.21	35.28
19	5.83	5.59	6.21	34.63
20	5.80	5.58	6.22	34.55

Le temps principal pour les démonstrations est celui de l'évaluation des modèles de Taylor, l'évaluation de l'expression 2^{-i} n'a apparemment pas une incidence significative.

6.3 Conclusion

Dans ce chapitre nous avons utilisé les modèles de Taylor pour garantir que l'erreur d'approximation d'une fonction par un polynôme est inférieure à un seuil donné. Spécifiquement, nous avons utilisé une approximation polynomiale pour l'exponentielle et pour l'arc-tangente.

Une limitation que nous avons rencontrée est que nous ne pouvons pas calculer avec des modèles de Taylor d'ordre supérieur à 6. Pour un ordre supérieur à 6, l'évaluation des modèles de Taylor prend trop de temps et PVS finit par se bloquer avant de terminer ses calculs. Nous ne comprenons pas encore la raison de ce comportement de PVS avec notre implantation des modèles de Taylor, nous soupçonnons essentiellement une limite de puissance (vitesse et peut-être mémoire).

Avec des modèles de Taylor d'ordre 6, nous pouvons démontrer que l'erreur d'approximation du polynôme donné pour l'exponentielle est de l'ordre de $O(10^{-10})$, ce qui est suffisant pour l'évaluation de l'exponentielle avec arrondi correct en simple précision IEEE-754. En revanche, si nous voulons certifier des erreurs d'approximations pour la double précision IEEE-754, il faudrait pouvoir prouver des bornes sur l'erreur d'approximation de l'ordre de $O(2^{-60})$, soit $O(10^{-18})$, ce qui implique que nous devrons pouvoir utiliser des modèles de Taylor d'ordres plus grands.

Quand nous utilisons le graphique de l'erreur d'approximation pour guider le partitionnement de l'intervalle, nous utilisons un découpage non régulier. Une amélioration que nous pourrions apporter à la stratégie est de prendre comme argument des sous-intervalles donnés en paramètres pour les utiliser dans la démonstration.

Chapitre 7

Conclusion et perspectives

7.1 Résumé des travaux effectués

Pendant cette thèse nous avons écrit une bibliothèque pour l'arithmétique sur les modèles de Taylor pour l'assistant de preuves PVS. La bibliothèque contient les opérations de base de l'arithmétique des modèles de Taylor, les opérations inverse et racine carrée et les fonctions exponentielle, arc-tangente, sinus, sinus et cosinus hyperboliques. Nous avons montré en PVS que les opérations et les fonctions ainsi définies satisfont la propriété d'inclusion, travail qui, à notre connaissance, n'a jamais été réalisé par ailleurs.

Nous avons développé deux stratégies PVS pour simplifier et automatiser l'utilisation des modèles de Taylor. La première, nommée `containment`, permet de montrer la propriété d'inclusion d'une expression dans un modèle de Taylor en utilisant les théorèmes d'inclusion que nous avons démontrés pour les fonctions et les opérations de base de l'arithmétique des modèles de Taylor. En effet, il est fastidieux d'effectuer ce type de preuves en PVS sans utiliser la stratégie que nous avons développée. La deuxième stratégie, nommée `taylors`, permet de certifier des inégalités ou des bornes sur des expressions en utilisant les modèles de Taylor. Cette stratégie permet interactivement d'augmenter l'ordre du modèle de Taylor (plus l'ordre est élevé et plus l'erreur d'approximation est faible) ou de diviser le domaine de l'intervalle d'entrée (plus l'intervalle d'entrée est petit et plus le reste est précis).

Notre bibliothèque est composée de 10 fichiers PVS (1552 lignes), 9 fichiers de preuves (31275 lignes) et un fichier de stratégies (175 lignes). Les dépendances entre les théories que nous avons développées sont montrées sur la figure 7.1. Cette figure se lit ainsi : une flèche de a vers b signifie que la théorie b dépend de la théorie a. À chaque théorie correspond un fichier portant le même nom que la théorie, suivi du suffixe `.pvs`.

Ci-dessous, nous présentons, pour chaque théorie, son contenu :

`finite_support` : cette théorie contient les définitions et théorèmes pour les suites à support fini, voir §3.1 ;

`polynomials_ext` : cette théorie contient le théorèmes et définitions pour les polynômes, voir §3.2 ;

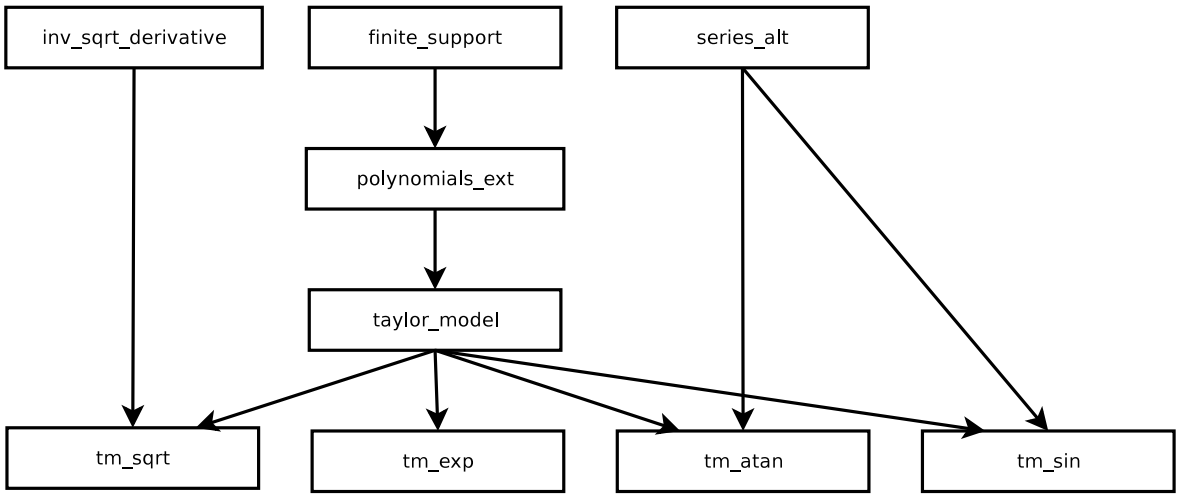

FIG. 7.1 – Dépendances entre les théories que nous avons développées.

`series_ext` : cette théorie contient les théorèmes et définitions pour les séries alternées, voir §4.3.1 ;

`taylor_model` : cette théorie contient la définition des modèles de Taylor et des opérations, ainsi que les théorèmes d'inclusion pour les opérations définies, voir le chapitre 3 ;

`inv_sqrt_derivative` : cette théorie contient quelques théorèmes sur l'inverse de la racine carrée, nécessaires pour établir le modèle de Taylor de la racine carrée ;

`tm_sqrt` : cette théorie contient la définition du modèle de Taylor de la racine carrée, son théorème d'inclusion et quelques théorèmes nécessaires pour son développement, voir §3.6 ;

`tm_exp` : tout comme la théorie `tm_sqrt` mais pour le modèle de Taylor de la fonction exponentielle, voir §4.1 ;

`tm_atan` : tout comme la théorie `tm_sqrt` mais pour le modèle de Taylor de l'arctangente, voir §4.3 ;

`tm_sin` : tout comme la théorie `tm_sqrt` mais pour le modèle de Taylor du sinus, voir §4.4.

Nous tenons à préciser qu'aucun de nos développements ne contient d'axiome, il n'y a donc aucun risque d'introduire une contradiction en PVS par l'utilisation de nos théories.

Notre bibliothèque peut être utilisée pour construire des modèles de Taylor pour des expressions données, pour dériver des bornes plus ou moins précises pour des expressions arithmétiques, et également pour certifier des inégalités ou des bornes sur des expressions.

Pour développer les modèles de Taylor, nous avons eu besoin d'implanter une théorie des suites à support fini, une théorie des polynômes et une théorie des séries alternées. Dans la théorie des suites à support fini nous avons montré, entre autres, que la multiplication de deux séries définies par des suites à support fini est donnée par le produit de Cauchy. Indépendamment et simultanément, Lester (Manchester, Grande–Bretagne) a développé une théorie des polynômes. Nous avons démontré que notre version est équivalente à la version de Lester. En comparaison avec la version de Lester, nous avons plus de développements : la théorie des séries à support fini et également le produit, la puissance et la composition de polynômes. Par ailleurs, notre théorie des séries alternées peut être utilisée pour démontrer la convergence d'une série alternée et aussi pour borner son reste. Ces théories peuvent être utilisées indépendamment des théories sur les modèles de Taylor.

7.2 Difficultés rencontrées

Une des difficultés a été de se familiariser avec l'outil PVS. Mon apprentissage initial de PVS a été rapide grâce au cours donné par les laboratoires NASA Langley et NIA, que j'ai suivi en mai 2005, et aussi grâce à l'aide constante de Muñoz et à la liste de diffusion d'aide de PVS.

Quelques-uns des points forts de PVS sont son niveau d'automatisation, la définition de façon native des nombres rationnels et sa large bibliothèque mathématique. En revanche, la manipulation des expressions, et tout particulièrement la manipulation des définitions des fonctions des modèles de Taylor pour montrer leur propriété d'inclusion, n'a pas été facile du tout. Avec la bibliothèque `Manip` de Di Vito [67], les manipulations d'expressions de type réel sont allégées, mais en revanche manipuler des expressions de types différents du type réel est une tâche qui reste complexe en PVS.

Bien que disposer d'une large bibliothèque mathématique en PVS soit un avantage, une difficulté est qu'il n'existe pas d'outil pour rechercher quels théorèmes on peut appliquer à un but de preuve PVS (qui pourrait être semblable à l'interface *proof general* de l'assistant de preuves Isabelle). Par conséquent, si on ne connaît pas le nom du théorème à appliquer, on doit le rechercher directement dans le fichier de bibliothèques de PVS.

Une autre difficulté a été de rester à jour avec les différentes actualisations des bibliothèques de PVS du laboratoire NASA Langley. Parfois des noms de théorèmes ont été modifiés, ou des théories ont changé de bibliothèques d'une version à la suivante. De ce fait, il est probable que les preuves que nous avons déjà effectuées deviennent invalides si jamais il y a un changement de version de ces bibliothèques. Au cours de la thèse, ces changements ont impliqué une vérification et une actualisation périodique des preuves, or refaire des preuves qui ont déjà été faites est un travail qui peut paraître inutile.

7.3 Perspectives

Une première amélioration que nous pouvons réaliser est de remplacer le schéma d'évaluation des polynômes par le schéma de Horner par exemple. Cela est possible, en utilisant la propriété de sous-distributivité de l'arithmétique par intervalles : $\boldsymbol{x}(\boldsymbol{y}+\boldsymbol{z}) \subseteq \boldsymbol{xy} + \boldsymbol{xz}$. Cette modification peut non seulement améliorer la précision des intervalles mais également le temps d'évaluation des modèles de Taylor.

Une autre modification à faire à court terme est que la bibliothèque actuelle est développée avec PVS 3.2, il faudrait l'actualiser pour la nouvelle version PVS 4.0. Il faudrait aussi compléter les fonctions disponibles pour les modèles de Taylor avec d'autres fonctions telles que le cosinus, la tangente et le logarithme.

Une piste de travail à plus long terme est l'automatisation encore plus poussée de notre stratégie `taylors` : il s'agit d'étudier comment déterminer s'il est préférable de diviser le domaine d'entrée ou d'augmenter l'ordre du modèle de Taylor, en fonction de l'expression à borner ou de l'inégalité à prouver. L'idée est de rendre invisible l'emploi des modèles de Taylor à l'utilisateur. De cette façon, on pourrait développer une interface externe à PVS pour certifier des inégalités ou des bornes sur des expressions.

Un autre piste est d'étudier et d'implanter l'extension des modèles de Taylor au cas multivariable. Ici les questions d'efficacité se poseront de manière cruciale.

Enfin, il n'est pas obligatoire d'utiliser comme base les polynômes de Taylor. Une piste est d'explorer d'autres bases pour les polynômes, comme par exemple les polynômes de Bernstein, les polynômes trigonométriques ou les séries de Poisson [41].

Table des figures

Bibliographie

[1] Jacques-Louis Lions et al. Ariane 5 flight 501 failure report by the inquiry board. Technical report, European Space Agency, Paris, France, 1996.

[2] Information Management and Technology Division. Patriot missile defense : software problem led to system failure at Dhahran, Saudi Arabia. Report B-247094, United States General Accounting Office, 1992. http://www.fas.org/spp/starwars/gao/im92026.htm.

[3] Debbie Gage and John McCormick. We did nothing wrong. *Baseline*, 1(28) :32–58, 2004. http://www.baselinemag.com/article2/0,1397,1544403,00.asp.

[4] Common Criteria. http://www.commoncriteriaportal.org/.

[5] Philip E. Ross. The exterminators. *IEEE Spectrum*, 42(9) :36–41, 2005. http://www.spectrum.ieee.org/sep05/1454.

[6] John Rushby. Disappearing formal methods. In *High-Assurance Systems Engineering Symposium*, pages 95–96, Albuquerque, NM, November 2000. Association for Computing Machinery. http://www.csl.sri.com/~rushby/hase00.html.

[7] Ashish Tiwari, Natarajan Shankar, and John Rushby. Invisible formal methods for embedded control systems. *Proceedings of the IEEE*, 91(1) :29–39, 2003. http://www.csl.sri.com/~rushby/papers/procieee03.pdf.

[8] Jean-Christophe Filliâtre. Why : a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003. http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz.

[9] Guillaume Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2006. http://perso.ens-lyon.fr/guillaume.melquiond/doc/06-these.pdf.

[10] César Muñoz and David Lester. Real number calculations and theorem proving. In *18th International Conference on Theorem Proving in Higher Order Logics*, pages 239–254, Oxford, England, 2005. http://research.nianet.org/~munoz/Papers/tphols05.pdf.

[11] Marc Daumas, Guillaume Melquiond, and César Muñoz. Guaranteed proofs using interval arithmetic. In Paolo Montuschi and Eric Schwarz, editors, *Proceedings of the*

17th Symposium on Computer Arithmetic, pages 188–195, Cape Cod, Massachusetts, 2005. http://research.nianet.org/~munoz/Papers/arith-17.pdf.

[12] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning : An Approach*. Kluwer Academic Publishers, 2000.

[13] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning : ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[14] Matt Kaufmann and J. Strother Moore. ACL2. http://www.cs.utexas.edu/users/moore/acl2/.

[15] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. *The Coq proof assistant : a tutorial : version 8.0*, 2004. http://coq.inria.fr.

[16] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

[17] L'outil d'aide à la preuve Coq. http://coq.inria.fr/coq-fra.html.

[18] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL : A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[19] HOL automated proof system for higher order logic. http://hol.sourceforge.net/.

[20] John Harrison. HOL light : a tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 265–269, 1996. http://www.cl.cam.ac.uk/users/jrh/papers/demo.ps.gz.

[21] The HOL Light theorem prover. http://www.cl.cam.ac.uk/~jrh13/hol-light/.

[22] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer Verlag, 2002.

[23] Isabelle. http://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[24] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS : a prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, pages 748–752, Saratoga, New-York, 1992. Springer-Verlag. http://pvs.csl.sri.com/papers/cade92-pvs/cade92-pvs.ps.

[25] Sam Owre, Natarajan Shankar, John M. Rushby, and David W. J. Stringer-Calvert. *PVS System Guide*. SRI International, 2001. Version 2.4. http://pvs.csl.sri.com/doc/pvs-system-guide.pdf.

[26] Sam Owre, Natarajan Shankar, John M. Rushby, and David W. J. Stringer-Calvert. *PVS Language Reference*. SRI International, 2001. Version 2.4. http://pvs.csl.sri.com/doc/pvs-language-reference.pdf.

[27] PVS Specification and Verification System. http://pvs.csl.sri.com/.

[28] Freek Wiedijk. Comparing mathematical provers. In Andrea Asperti, Bruno Buchberger, and James Davenport, editors, *Proceedings of Mathematical Knowledge Management (MKM)*, pages 188–202, 2003. `http://www.cs.ru.nl/~freek/comparison/diffs.pdf`.

[29] David Russinoff, Matt Kauffmann, Eric Smith, and Robert Sumners. Formal verification of floating-point RTL at AMD using the ACL2 theorem prover. In *Proceedings of the 17th IMACS World Congress on Computational and Applied Mathematics*, Paris, France, 2005. `http://sab.sscc.ru/imacs2005/papers/T2-I-94-1021.pdf`.

[30] Ricky Butler, Jeff Maddalon, Alfons Geser, and César Muñoz. Formal verification of a conflict resolution and recovery algorithm. Technical Paper NASA/TP-2004-213015, NASA Langley Research Center, Hampton, VA, 2004. `http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20040065775_2004063175.pdf`.

[31] César Muñoz, Víctor Carreño, Gilles Dowek, and Ricky Butler. Formal verification of conflict detection algorithms. *International Journal on Software Tools for Technology Transfer*, 4(3) :371–380, 2003.

[32] Jean-Michel Muller. *Arithmétique des Ordinateurs*. Masson, 1989.

[33] David Goldberg. *Computer architecture : a quantitative approach*, chapter Floating point arithmetic. Morgan Kaufmann, 1990.

[34] Valérie Ménissier-Morain and Pierre Weis. An exact arithmetic package for ML. *Science for Computer Programming*, 1995. `http://www-calfor.lip6.fr/~vmm/documents/scp95.ps.gz`.

[35] Ramon Edgar Moore. *Interval arithmetic and automatic error analysis in digital computing*. PhD thesis, Stanford University, 1962. `http://interval.louisiana.edu/Moores_early_papers/disert.pdf`.

[36] Ramon E. Moore. *Interval analysis*. Prentice Hall, 1966.

[37] Eldon Hansen. *Global optimization using interval analysis*. Marcel Dekker, 1992.

[38] R. Baker Kearfott. *Rigorous global search : continuous problems*. Kluwer Academic Publishers, 1996.

[39] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Eric Walter. *Applied interval analysis*. Springer, 2001.

[40] Arnold Neumaier. *Interval methods for systems of equations*. Cambridge University Press, 1990.

[41] Nedialko S. Nedialkov, Vladik Kreinovich, and Scott A. Starks. Interval Arithmetic, Affine Arithmetic, Taylor Series Methods : Why, What Next ? *Numerical Algorithms*, 37 :325–336, 2004. `http://citeseer.ist.psu.edu/nedialkov03interval.html`.

[42] Kyoko Makino. *Rigourous Analysis of Nonlinear Motion in Particle Accelerators*. PhD thesis, Michigan State University, East Lansing, Michigan, USA, 1998. http://www.bt.pa.msu.edu/pub/papers/makinophd/makinophd.pdf.

[43] Martin Berz and Georg Hoffstätter. Computation and application of Taylor polynomials with interval remainder bounds. *Reliable Computing*, 4(1) :83–97, 1998. citeseer.ist.psu.edu/berz98computation.html.

[44] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed ? *ACMTOPLAS : ACM Transactions on Programming Languages and Systems*, 21, 1999. citeseer.ist.psu.edu/article/lamport98should.html.

[45] J. Rushby. Formal specification and verification for critical systems : Tools, achievements, and prospects. In *EPRI Workshop on Methodologies for Cost-Effective, Reliable Software Verification and Validation*, pages 9–1 to 9–14, Chicago, 1992. Electric Power Research Institute (EPRI). citeseer.ist.psu.edu/article/rushby91formal.html.

[46] Emacs. http://www.gnu.org/software/emacs/.

[47] Sam Owre, Natarajan Shankar, John M. Rushby, and David W. J. Stringer-Calvert. *PVS Prover Guide*. SRI International, 2001. Version 2.4. http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf.

[48] Arnold Neumaier. Taylor forms – use and limits. *Reliable Computing*, 9(1) :43–79, 2003. http://citeseer.ist.psu.edu/neumaier02taylor.html.

[49] Jean-Pierre Eckmann, Hans Koch, and Peter Wittwer. A computer-assisted proof of universality in area-preserving maps. *Memoirs of the AMS*, 47(289), 1984.

[50] Jean-Pierre Eckmann, Andreas Malaspinas, and Sylvie Oliffson-Kamphorst. A software tools for analysis in function spaces. In *Computer Aided Proofs in Analysis*, pages 147–166, New York, 1991. Springer.

[51] Kyoko Makino and Martin Berz. Verified Integration of ODEs and Flows using Differential Algebraic Methods on High-Order Taylor Models. *Reliable Computing*, 4 :361–369, 1998. http://www.bt.pa.msu.edu/pub/papers/rdaint/rdaint.pdf.

[52] Jens Hoefkens. *Rigourous Numerical Analysis with High-Order Taylor Models*. PhD thesis, Michigan State University, East Lansing, Michigan, USA, 2001. http://www.bt.pa.msu.edu/pub/papers/hoefkensphd/hoefkensphd.pdf.

[53] Kyoko Makino and Martin Berz. 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.

[54] Nathalie Revol, Kyoko Makino, and Martin Berz. Taylor models and floating-point arithmetic : proof that arithmetic operations are validated in COSY. *Journal of Logic and Algebraic Programming*, 64(1) :135–154, 2005. http://dx.doi.org/10.1016/j.jlap.2004.07.008.

[55] Roland Zumkeller. Formal Global Optimisation with Taylor Models. In U. Furbach and N. Shankar, editors, *Automated Reasoning*, volume 4130/2006 of *Lecture Notes in Computer Science*, pages 408–422. Springer Verlag, 2006. `http://www.lix.polytechnique.fr/~zumkeller/Publications_files/rz_formal_taylor-1.pdf`.

[56] PVS ground Evaluator. `http://pvs.csl.sri.com/experimental/eval.shtml`.

[57] PVSio. `http://research.nianet.org/~munoz/PVSio/`.

[58] Myla Archer, Ben Di Vito, and César Muñoz. Developing user strategies in PVS : A tutorial. In *Proceedings of Design and Application of Strategies/Tactics in Higher Order Logics STRATA'03*, NASA/CP-2003-212448, pages 16–42, NASA LaRC,Hampton VA 23681-2199, USA, September 2003. `http://research.nianet.org/fm-at-nia/STRATA2003/`.

[59] Sam Owre and Natarajan Shankar. Writing PVS proof strategies. In *Proceedings of Design and Application of Strategies/Tactics in Higher Order Logics STRATA'03*, NASA/CP-2003-212448, pages 1–15, NASA LaRC,Hampton VA 23681-2199, USA, September 2003. `http://research.nianet.org/fm-at-nia/STRATA2003/`.

[60] Gilles Dowek, Alfons Geser, and César Muñoz. Tactical conflict detection and resolution in a 3-D airspace. In *Proceedings of the 4th USA/Europe Air Traffic Management R&DSeminar, ATM 2001*, Santa Fe, New Mexico, 2001. `http://research.nianet.org/~munoz/Papers/atm2001.pdf`.

[61] César Muñoz, Victor Carreño, Gilles Dowek, and Ricky Butler. Formal verification of conflict detection algorithms. *International Journal on Software Tools for Technology Transfer*, 4(3) :371–380, 2003.

[62] American National Standards Institute and Institute of Electrical and Electronic Engineers. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985,* New York, 1985.

[63] William Kahan. Lecture notes on the status of the IEEE standard 754 for binary floating point arithmetic. Published on the net, 1996.

[64] Guillaume Melquiond. Robustesse d'algorithmes pour l'évitement des collisions aériennes. Master's thesis, École Normale Supérieure de Lyon, Lyon, France, 2003.

[65] Marc Daumas, David Lester, and César Muñoz. Verified Real Number Calculations : A Library for Interval Arithmetic. Research Report 00168402, HAL `http://hal.archives-ouvertes.fr/hal-00168402/fr/`, 2007.

[66] Peter Markstein. *IA-64 and elementary functions : speed and precision*. Prentice Hall, 2000.

[67] Manip. `http://shemesh.larc.nasa.gov/people/bld/manip.html`.