**THÈSE de DOCTORAT de l'UNIVERSITÉ PARIS 6**

Spécialité :

**Informatique Fondamentale**

présentée par

**Micaela MAYERO**

pour obtenir le grade de **DOCTEUR de l'UNIVERSITÉ PARIS 6**

Sujet de la thèse :

# Formalisation et automatisation de preuves en analyses réelle et numérique

soutenue le 20 décembre 2001

devant le jury composé de :

| | | | |
|---|---|---|---|
| **Mme** | **Thérèse** | **HARDIN** | Présidente |
| **MM.** | **Gilles** | **DOWEK** | Directeur |
| | **Herman** | **GEUVERS** | Rapporteurs |
| | **Laurent** | **THÉRY** | |
| | **Yves** | **BERTOT** | |
| | **Laurent** | **HASCOËT** | Examinateurs |
| | **Jean-Michel** | **MULLER** | |
| | **César** | **MUÑOZ** | |

*A mis padres, a Quica*
*À David*
*À ma Sylvie*
*À Gérard, à Jeannine, à Ben, à Pierrot, à Roméo*

# Remerciements

Cela fait un certain nombre d'années déjà que je m'intéresse à l'interaction entre les mathématiques et l'informatique. Le sujet de stage de DEA proposé par Gérard Huet m'a tout de suite enthousiasmé de par le défi qu'il représentait. Je remercie donc Gérard Huet de m'avoir encadrée pour ce stage, qui a représenté un point de départ pour mon futur travail de thèse. C'est à ce moment là que j'ai fait plus ample connaissance avec Gilles Dowek, qui m'a bien souvent été d'une aide précieuse tout au long de ce stage. Bien que le thème sur lequel je souhaitais travailler ne fasse pas partie de ses axes principaux de recherche, Gilles Dowek a tout de même accepté de diriger mon travail de thèse. Cette tâche n'était pas particulièrement simple car le sujet était assez flou et, de fait, beaucoup de directions possibles. Je remercie beaucoup Gilles pour toute son aide et ses conseils judicieux, sur tous les aspects de la vie d'un chercheur, au cours de ces quatre années.

Je tiens également à remercier Thérèse Hardin d'avoir accepté de présider mon jury de thèse. Thérèse Hardin fut la première personne à me parler de $\lambda$-calcul, ce qui m'a incitée à poursuivre dans cette voie. Pour cette raison je l'en remercie doublement.

Je remercie Herman Geuvers, Laurent Théry et Yves Bertot d'avoir accepté de rapporter sur ma thèse, ainsi que pour leurs remarques constructives, qui ont beaucoup amélioré ce document. Je les remercie aussi pour toutes les discussions que nous avons pu avoir à chacune de nos rencontres lors de ces quatre années.

Je remercie l'ensemble des membres du jury d'avoir accepté d'en faire partie et pour l'intérêt qu'ils ont porté à mon travail. Je suis très reconnaissante à Laurent Hascoët d'être venu gentiment à mon secours en m'aidant à mieux appréhender le système O∂yssée. Mon dernier chapitre ne serait pas ce qu'il est sans son soutien. J'ai connu César Muñoz lorsqu'il était encore dans le projet Coq, et j'ai été particulièrement heureuse de le revoir et de travailler avec lui. Je le remercie pour son accueil chaleureux à Hampton. Jean-Michel Muller s'est souvent intéressé à mon travail, bien que les preuves formelles ne fassent pas directement partie de ses thèmes de recherche, et je l'en remercie.

Je remercie les numériciens des projets Ondes et Estimes de l'INRIA, qui se sont intéressés au projet et qui étaient présents lors de nos rencontres. En particulier, François Clément et Patrick Joly qui sont à l'origine de l'idée de travailler sur O∂yssée.

J'ai également eu le plaisir de passer une semaine chez Intel à Portland où j'ai fait la connaissance de John Harrison. Travailler avec lui, entre autres durant cette semaine, a été très agréable. Je le remercie infiniment pour m'avoir fait profiter de sa grande expérience et pour s'être intéressé à mon travail.

Au cours de ces dernières années, j'ai eu l'occasion de faire beaucoup de nouvelles connaissances, qui ont toutes apporté leur contribution, sous quelque forme que ce soit, à ce travail. En premier lieu, je remercie tous les membres passés et présents du projet Coq-LogiCal de Rocquencourt, d'Orsay, de Lyon, en particulier Gilles Dowek, Chritine Paulin-Mohring, Benjamin Werner, Jean-Christophe Filliâtre, Bruno Barras, Judicaël Courant, Eduardo Ginénez, Cristina Cornes, César Muñoz, Jean Duprat, Jean Goubault-Larrecq, Patrick Loiseleur, Pierre-Louis Curien (qui ont toujours été de bon conseils), David Delahaye (qui connaît presque mon travail aussi bien que moi, pour m'avoir souvent écoutée en parler et pour m'avoir considérablement simplifié la vie avec $\mathcal{L}_{tac}$), Alexandre Miquel (pour avoir supporté mes très nombreux harcèlements sur la question épineuse des sortes de Coq). Je remercie également Loïc Pottier, Renaud Rioboo et Claude Marché pour leurs conseils scientifiques préliminaires au développement de la tactique Field, ainsi que David Delahaye avec qui c'est un plaisir de collaborer.
Je suis particulièrement heureuse d'avoir fait la connaissance des membres du projet Lemme de l'INRIA-Sophia, et je remercie de tout cœur Laurence Rideau, Loïc Pottier, Laurent Théry, Yves Bertot pour avoir porté un intérêt particulier à mon travail (et pour avoir été les premiers à l'utiliser et à le faire évoluer) ainsi que pour nos nombreuses discussions amicales. Merci également à André Hirschowitz pour ses remarques.
Je remercie également les membres des thèmes SPI et CALFOR du LIP6, en particulier Thérèse Hardin, Valérie Ménissier-Morain, Renaud Rioboo, Sylvain Boulmé, avec qui les contacts ont toujours été très chaleureux.
Je remercie les membres du projet PPS, principalement Pascal Manoury, Emmanuel Chailloux, Guy Cousineau pour avoir été les piliers de mes connaissances en $\lambda$-calcul.

L'environnement au sein duquel s'effectue un travail de recherche a une très grande influence. À l'INRIA-Rocquencourt, cet environnement de travail est spécialement propice. Je remercie tous les membres des projets Cristal et Moscova pour la formation unique, et au quotidien, en matière d'ingénierie système. Je pense à Pierre Weis, Xavier Leroy, Didier Rémy, Michel Mauny, Damien Doligez.
Je souhaite mentionner tout spécialement Daniel de Rauglaudre, qui, en plus de m'avoir appris énormément de choses sur des domaines divers de l'informatique, a su apporter, à chacune de ses incursions dans le bureau, un rayon (même plusieurs) de gentillesse et de bonne humeur. Certaines discussions philosophiques et métaphysiques restent mémorables, tout comme une certaine "roue de bicyclette", partie de mon travail qui a souvent intrigué Daniel, ce qui a parfois eu pour effet de me débloquer lors d'un problème. Pour toute cette bonne ambiance, je l'en remercie mille et une fois.
Je remercie les assistantes et assistant de projets, sans qui toute démarche deviendrait très vite insurmontable, en particulier, Nelly Maloisel, Dany Moreau, Sylvie Loubressac et David Massot.

Je remercie toute ma famille pour son soutien : *papá y mamá, que, a pesar del alejamiento, estuvieron siempre presentes, siguiéndome paso a paso y apoyándome sin dudar, mismo cuando todo no era tan claro para mí*; Sylvie qui a toujours été là pour moi, comme une vraie grand-mère (plus que *pseudo*) ; mes beaux-parents et Ben, pour avoir grandement contribué à un équilibre nécessaire entre réflexion et détente ; David, évidemment, pour avoir supporté mes moments cafardeux et/ou sautes d'humeurs lorsque quelque chose me contrariait et qui a vraiment tout partagé en toute circonstance, ce qui pourrait faire des bons moments *ses* bons moments, des mauvais *ses* mauvais moments (c'est déjà bien moins amusant) et, en quelque sorte, de ce travail, sa seconde thèse.

# Table des matières

# Introduction

L'objectif de cette thèse est l'étude d'interactions entre les méthodes formelles et le domaine de l'analyse numérique. Beaucoup de programmes critiques sont, en effet, issus de cette science, mais les travaux traitant des applications des méthodes formelles aux programmes d'analyse numérique sont rares. Diverses raisons sont à l'origine de cet état de fait. Une première raison est l'utilisation importante des nombres réels dans les programmes numériques, alors que les méthodes formelles manipulent plutôt des nombres entiers, ou plus généralement des structures discrètes. Le domaine des nombres réels dans les méthodes formelles reçoit de plus en plus un intérêt légitime. Nous pouvons citer, par exemple, les interactions récentes entre le calcul formel et la preuve de programme [8, 2], les développements dans les systèmes formels de librairies flottantes, qui sont bien souvent effectués à partir des nombres réels [22, 59, 48, 75, 74, 52], ou encore les preuves formelles en aéronautique qui utilisent de la géométrie réelle [65]. Une deuxième raison provient du fait que les méthodes formelles s'appliquent généralement aux programmes fonctionnels et elles sont, de ce fait, moins bien adaptées aux programmes d'analyse numérique, qui sont, le plus souvent, impératifs. Cette difficulté est aujourd'hui en train de recevoir une attention accrue dans le domaine des méthodes formelles (voir, par exemple, [33]).

Cette thèse s'inscrit dans ce mouvement. Nous défendons, en effet, le point de vue que les outils de méthodes formelles, en particulier les systèmes de preuves formelles tels que Coq, sont aujourd'hui de plus en plus adaptés pour que l'on s'intéresse aux nombres réels, ce qui ouvre une brèche pour l'application de ces systèmes aux programmes d'analyse numériques réels. Nous proposons d'illustrer ces affirmations par l'étude de deux exemples issus, l'un de la géométrie réelle, et l'autre de l'analyse numérique. Le premier exemple est la preuve formelle du théorème des trois intervalles (conjecture de Steinhaus). Il s'agit d'un théorème purement mathématique faisant intervenir des propriétés utilisant à la fois des fonctions réelles et relevant de l'intuition géométrique, ce qui en fait un excellent candidat pour notre étude. Le second exemple concerne la preuve de corrections, en Coq, de l'algorithme de différentiation automatique de programmes FORTRAN utilisé dans le mode direct du système O∂yssée. Ce problème nécessite la formalisation d'une sémantique d'un noyau du langage FORTRAN, ainsi que l'utilisation de notions de l'analyse réelle, telle que la dérivée. Cet exemple permettra de mettre en avant une des interactions possibles entre systèmes de preuves formelles et programmes numériques réels.

Afin de pouvoir traiter ces deux exemples, nous avons eu besoin de démontrer un nombre important de propriétés sur les nombres réels. Pour cela, nous avons dû commencer par étudier les différentes façons de définir les nombres réels dans un système tel que Coq. L'ensemble de ces développements (incluant les bases de l'analyse réelle telles que les notions de limites, dérivées,...) constituent une librairie qui est aujourd'hui distribuée avec le système Coq. Lors de ce développement et des deux démonstrations, nous avons souvent ressenti le besoin d'outils d'automatisation afin de traiter des démonstrations intermédiaires simples mais parfois fastidieuses. Nous avons pour cela développé plusieurs tactiques. Nous détaillerons particulièrement la tactique réflexive *Field*, qui résout des égalités sur les corps abéliens, développée à l'aide du nouveau langage de tactiques du système Coq.

# Chapitre 1

# Formalisation des nombres réels.

Les mots "*nombres réels*" sont souvent employés comme terme générique afin de représenter différentes notions selon le milieu scientifique qui les utilise. En effet, loin des formalisations mathématiques, qui sont déjà nombreuses, nous trouvons les nouvelles formalisations introduites par la science de l'informatique. Dans toutes ces formalisations concernant les nombres réels, nous pouvons citer, entre autres, les définitions mathématiques théoriques [9, 11, 27, 15, 57], les travaux dans la domaine de l'informatique sur l'arithmétique exacte [90, 64, 31, 30, 66], les nombres flottants dans les langages de programmation, les formalisations dans les systèmes formels [29, 43, 62, 39, 47, 50] et dans le domaine du calcul formel [72].
Nous ferons une brève présentation, afin de mieux identifier notre problématique future, des nombres réels dans l'informatique en général. Puis, nous détaillerons particulièrement les formalisations possibles dans les méthodes formelles, avant d'exposer la formalisation concernant les nombres réels dans Coq[1]. Pour cela, nous discuterons du choix de formalisation. Ce choix sera guidé par certaines spécificités du système. Enfin, nous présenterons des fonctions élémentaires sur les nombres réels et démontrerons certaines de leurs propriétés.

## 1.1 Les nombres réels en général

Dans les langages de programmation, qui servent à faire des calculs, on trouve les nombres à virgule flottante (ou nombres flottants). Ces nombres ne sont pas des nombres réels, mais en sont une approximation plus ou moins satisfaisante. En effet, on dénombre, en plus des résultats erronés, certaines propriétés élémentaires du corps des réels non respectées. Les deux exemples qui vont suivre illustrent ces deux problèmes. Pour plus de détails nous pourrons consulter [64, 90, 66]. La suite $(a_n)_{n\in\mathbb{N}}$ suivante, due à Jean-Michel Muller, est un exemple, parmi d'autres, des problèmes liés aux erreurs d'arrondi :

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

Bien que la notion de calcul dans un langage de programmation n'est en aucun cas une démonstration, on peut tout de même espérer retrouver un résultat "assez proche" de celui

[1]Le lecteur non familiarisé avec ce système pourra se référer régulièrement à l'annexe D pour une présentation générale des notations ainsi que des spécificités que nous utiliserons.

donné par une preuve mathématique (informelle ou formelle). Par récurrence on montre que

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

dont la limite à l'infini vaut 6.

En effectuant les calculs, en définissant une fonction récursive, en double précision (dans Ocaml), on obtient les valeurs suivantes :

| $n$ | 2 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 25 | 26 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $a_n$ | 5.59 | 15.41 | 67.47 | 97.14 | 99.82 | 99.98 | 99.99 | 99.99 | 99.99 | 100. | 100. |

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

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

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

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

Les problèmes liés aux résultats erronés ont mis en avant la nécessité d'une arithmétique exacte. L'idée est d'obtenir des résultats exacts ou "aussi proches que l'on veut". Pour cela, l'utilisateur indique une borne supérieure sur l'erreur d'arrondi autorisée sur le résultat final et cette borne est respectée tout au long de calcul. Plusieurs travaux ont été réalisés sur ce sujet [90, 64].
Nous pensons que cette approche pourrait servir à une extraction dans Coq, en spécifiant comme type `ML` une implantation de réels exacts, puisque nous pourrions alors faire en sorte que les propriétés de corps soient respectées.

Dans les systèmes de calcul formel, nous trouvons plusieurs formes de définition pour les nombres réels. En plus des flottants en précision arbitraire (où il est possible de choisir une précision fixe, 1000 chiffres décimaux, par exemple, dans `Maple` [72]), il existe une couche supplémentaire formelle. La donnée $sin(1)$ peut être vue comme la valeur formelle ou comme sa valeur flottante 0.8414709848. Néanmoins, même si ces systèmes sont plus valides que les langages de programmations pour corroborer certaines propriétés, ils ne permettent pas de les vérifier de source sûre. En effet, le contre-exemple cité précédemment concernant la suite $a_n$ reste valable [64], c'est-à-dire que la suite continue également de converger vers 100 dans les systèmes de calcul formel.

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

### 1.1.1 Les choix de formalisation dans les systèmes formels

Dans cette section, nous présenterons les principales formalisations possibles pour définir le corps des nombres réels. Le développement que nous allons présenter dans la suite de ce chapitre sera fait dans le système Coq, qui repose sur la logique intuitionniste. De ce fait, il nous semble important de distinguer également les formalisations classiques et intuitionnistes. En effet, dans notre cas et comme nous le verrons par la suite, un des premiers choix que nous devrons faire concerne cet aspect[2]

#### Construire les nombres réels

Les deux formalisations les plus connues sont la *méthode de Cantor* et les *coupures de Dedekind*. Nous en ferons une brève présentation *classique*. Ces méthodes peuvent être adaptées, presque directement, à une logique constructive [6, 60, 87, 82].

**Méthode de Cantor** Cette méthode identifie un nombre réel à une suite de nombres rationnels convergente (vers ce nombre) en utilisant la propriété suivante.
Une suite $(s_n)$ converge vers $s$ si :

$$\forall \varepsilon > 0, \exists N \; \forall n \geq N, |s_n - s| < \varepsilon$$

Cette définition introduit naturellement la notion de suite de Cauchy. Pour plus de détails concernant cette méthode, on pourra se référer à [9, 11, 47].

**Coupures de Dedekind** Cette méthode identifie un nombre réel à un ensemble de nombres rationnels plus petit que lui. Si nous nommons $S$ cet ensemble, il s'agit d'une coupure s'il vérifie les propriétés suivantes :

1. $\exists x, x \in S$
2. $\exists x, x \notin S$
3. $\forall x \in S, \forall y < x, y \in S$
4. $\forall x \in S, \exists y > x, y \in S$

Pour tous les détails concernant cette méthode, on pourra se référer à [54, 47].

#### Axiomatiser les nombres réels

Axiomatiser une théorie revient à poser un ensemble minimal d'axiomes qui permettent de la spécifier. Nous détaillerons une axiomatisation classique des nombres réels dans la suite de ce chapitre.

Des axiomatisations constructives sont possibles. Dans ce cas, le problème principal vient de l'égalité. De manière générale, nous ne pouvons décider si deux nombres réels sont égaux ou pas. L'égalité usuelle (de Leibniz) ne peut être utilisée car, en logique intuitionniste, cela signifierait justement qu'il est possible de construire une preuve d'égalité entre deux nombres réels[3]. Il s'agit donc de trouver un opérateur jouant le rôle de l'égalité. Une telle axiomatisation a été faite au sein du projet FTA [39], en utilisant un opérateur dit d'*apartness* et d'équivalence axiomatisés comme ci-dessous et que nous noterons respectivement @ et $\equiv$.

[2] Le lecteur n'étant pas familiarisé avec ces notions de logique liée aux systèmes formels pourra tout d'abord se référer à l'annexe D section D.2.

[3] Nous verrons pourquoi ce problème ne se pose pas en logique classique.

L'idée est que si nous ne savons pas si deux réels sont égaux, il est généralement possible de savoir s'il ne le sont pas. Si nous voyons le symbole d'*apartness* comme un $\neq$, nous pouvons alors poser l'axiome de négation de la non réflexivité (premier axiome). De la même manière, si $x \neq y$ alors $y \neq x$ (deuxième axiome). Le troisième axiome est une forme d'inégalité triangulaire et enfin le dernier permet de définir l'équivalence $\equiv$ par rapport à l'*apartness* :
- $\forall x, \neg(@\ x\ x)$
- $\forall x, y, (@\ x\ y) \Rightarrow (@\ y\ x)$
- $\forall x, y, (@\ x\ y) \Rightarrow \forall z, (@\ x\ z) \vee (@\ z\ y)$
- $\forall x, y, \neg(@\ x\ y) \Leftrightarrow (\equiv\ x\ y)$

#### Une solution alternative

Une autre solution, utilisée dans PVS, est de profiter de la puissance du calcul sur les nombres entiers, que ce soit pour les types entier, relatif, rationnel ou réel. Cette méthode permet, en particulier, de s'abstenir de montrer des propriétés totalement calculatoires et triviales telles que $2 + 3 = 5$ ou que $2 \neq 0$ qui ne demandent alors plus de stratégie particulière (comme c'est le cas dans beaucoup de systèmes). Par contre, pour ce qui est des propriétés générales d'anneau, de corps ou les principes d'induction, ils doivent être également définis, tout comme dans les autres systèmes (axiomatisées pour les réels dans ce cas). Cette méthode est possible dans un système tel que PVS car celui-ci ne construit pas de preuves mais vérifie que la propriété est correcte. De ce fait, calculer $2 + 3$ de manière primitive et vérifier que c'est bien 5 suffit[4].

#### L'analyse non standard

Il existe une autre manière de définir les nombres réels, issue de l'analyse non standard, dont le précurseur fut A. Robinson [73]. E. Nelson [68] en donna une présentation axiomatique appelée *Internal Set Theory*. L'ensemble des nombres réels *standard* est complété par les nombres *hyperréels* (infinitésimaux ou infiniment grands). On obtient une structure de corps commutatif contenant les nombres réels mais dans lequel l'axiome d'Archimède n'est plus valide. Un nombre *hyperréels* est définit de la façon suivante :
- $x$ est infinitésimal si $|x|$ est strictement inférieur à tout standard positif
- $x$ est fini si $1/x$ est non infinitésimal
- $x$ est infiniment grand si $1/x$ est infinitesimal
- $x$ et $y$ sont infiniment proches si $x - y$ est infinitésimal

Cette approche est particulièrement adaptée, par exemple, au traitement des problèmes issus de la physique ou des théories de la mesure et des probabilités.

### 1.1.2 Quelques exemples de formalisations

Dans cette section nous allons citer quelques exemples de formalisations des nombres réels dans différents systèmes formels.

Dans le système HOL, il s'agit d'une construction classique utilisant la méthode de Cantor (un peu modifiée afin de se ramener le plus souvent possible aux entiers ou aux relatifs) faite par John Harrison [47] en 1995. Une grande partie de l'analyse standard y est maintenant formalisée.

[4] Nous verrons comment construire la preuve d'un exemple similaire en Coq.

Dans PVS, il s'agit d'une axiomatisation classique, où les constantes entières, relatives et rationnelles sont le codage primitif de la machine. Bruno Dutertre a développé en 1996 une partie de l'analyse réelle [29] . Hanne Gottliebsen y a codé notamment les fonctions transcendantes en 2000 [43].

Dans Lego, Claire Jones [51] a développé une axiomatisation classique en 1991.

Dans Nuprl, il s'agit d'une construction intuitionniste basée sur la formalisation de Bishop [6] faite par Douglas J.Howe en 1985 [50, 16].

Dans Mizar les coupures de Dedekind et une notion de sous-ensemble ont été utilisées afin de faire une construction classique [88].

Dans Coq, plusieurs formalisations existent maintenant ou sont en cours. La formalisation distribuée avec le système est celle que nous avons faite et que nous présenterons ci-après et au chapitre 2. Il s'agit d'une axiomatisation classique. Il y a également l'axiomatisation intuitionniste que nous avons précédemment décrite très brièvement, faite par Milad Niqui au sein du projet `FTA` [39]. Une construction intuitionniste est en cours par Alberto Ciaffaglione [26], basée sur l'idée qu'un nombre réel est une séquence potentiellement infinie (streams) de digits ternaires $(-1, 0, 1)$, ce qui se représente simplement grâce aux types co-inductifs de Coq.

Dans Isabelle, Jacques Fleuriot [34] a développé une bibliothèque d'analyse non standard.

Dans ACL2, Ruben Gamboa [38] a développé une bibliothèque d'analyse non standard.

## 1.2 Quels choix pour Coq ?

Nous avons vu précédemment les différentes possibilités de formaliser les nombres réels dans différents systèmes, que ce soit des langages de programmation ou des systèmes de calcul formel ou de preuves formelles. Commençons par exposer les avantages et inconvénients de chacune des formalisations. Nous avons deux sources d'opposition :

- construction / axiomatisation
- formalisation intuitionniste[5] / classique

Une `construction` présente un avantage certain de sûreté dans le sens où tout le développement, fait dans un système de preuves, ne peut qu'"être conforme" à ce système puisque tout y aura été prouvé. Par contre, comme nous avons pu nous en rendre compte précédemment lors des exemples de constructions à partir des méthodes de Cantor ou de Dedekind, et plus en détail dans [47], une telle construction demande un certain effort de formalisation et de développement.
Dans notre cas présent, nous avons choisi de diminuer le coût de développement en temps. En effet, il s'agit ici de formaliser les nombres réels. Cette théorie est bien connue et, de ce fait, le risque d'une formalisation, où un ensemble minimal de définitions ne serait pas prouvé, est faible.

Une `axiomatisation` permet d'arriver rapidement aux aspects plus intéressants et recherchés d'un développement, à condition, bien entendu, que les axiomes posés ne suscitent pas le moindre doute. Ce dernier point peut être vu comme le point négatif de cette méthode, puisque qu'un axiome faux permettrait de montrer $\perp$ (c'est-à-dire rendrait toute proposition prouvable), ce qui n'est pas souhaitable.

Une `formalisation constructive` ne peut être faite que dans un système de preuves basé sur la logique intuitionniste. Ce n'est pas un problème dans notre cas puisque c'est

[5] appelée également *formalisation constructive*

justement le cas de Coq. L'avantage d'une telle formalisation est qu'il est alors possible d'utiliser l'information calculatoire provenant des preuves effectuées. En effet, une fonction, représentée par un prédicat $P$ dont la proposition $\forall x\ \exists y\ (P\ x\ y)$ est démontrable de manière intuitionniste, sera calculable . En particulier, cet aspect prend toute son importance lorsque l'on veut utiliser le principe d'extraction de Coq. Notons que s'il s'agit d'une axiomatisation, il faudra commencer par réaliser les axiomes.
Dans certains cas, une formalisation constructive peut s'avérer trop restrictive. Notamment, abandonner le tiers exclus revient à abandonner des résultats importants en analyse[6], par exemple le théorème de Rolle [57] selon lequel une fonction continue et dérivable sur un intervalle, et prenant la même valeur aux bornes de cet intervalle, admet un point pour lequel la dérivée de la fonction en ce point est nulle.

Une `formalisation classique` est indispensable lorsqu'il est nécessaire de montrer l'existence de fonctions non calculables . De ce fait, s'il s'agit, par exemple, de montrer des théorèmes d'analyse mathématique, une telle formalisation semble mieux adaptée.
L'inconvénient majeur est, bien entendu, de ne pas pouvoir profiter de l'avantage offert par la formalisation intuitionniste, à savoir l'extraction de programmes.

D'un point de vue pragmatique, une différence importante apparaît, à la fois lors de la formalisation et lors de l'utilisation. C'est la propriété d'*ordre total* qui fait la différence entre une formalisation classique et une intuitionniste. Dans notre cas, l'utilisation d'une extension de l'égalité de Leibniz fait du développement classique un développement simple d'emploi. En particulier, le fait d'avoir cette égalité usuelle permet l'utilisation de tactiques de réécriture, qui sont très bien comprises et implantées dans les systèmes de preuves, par opposition où rien n'est prévu de manière générale pour gérer, par exemple, un opérateur tel que l'*apartness*, que nous avons cité précédemment.

Pour un résumé rapide de tous ces points, on pourra se référer au tableau 1.1. Nous notons par + les avantages et par - les aspects négatifs. Les considérations de temps ("long, rapide") se rapportent à la rapidité d'implantation des propriétés de base, tandis que les considérations de difficultés ("compliqué, simple") concernent aussi bien l'implantation que l'utilisation.

<table>
<tr><th></th><th></th><th>intuitionniste</th><th>classique</th></tr>
<tr><td rowspan="2">construction</td><td>+</td><td>prouvé<br>extractible</td><td>prouvé</td></tr>
<tr><td>-</td><td>long<br>compliqué</td><td>long</td></tr>
<tr><td rowspan="2">axiomatisation</td><td>+</td><td>rapide<br>± extractible</td><td>rapide<br>simple</td></tr>
<tr><td>-</td><td>non prouvé<br>compliqué</td><td>non prouvé<br>non extractible</td></tr>
</table>

TAB. 1.1 – Différentes méthodes de formalisation- Avantages et inconvénients

Les propriétés de base des nombres réels n'étant, dans notre cas, qu'un outil permettant de formaliser d'autres concepts, nous avons choisi de ne pas nous attarder sur les propriétés

[6]pour plus de détails à ce sujet nous pourrons nous référer à [87]

élémentaires et bien connues qui sont à l'origine de la théorie du corps des réels. Un de nos objectifs étant la formalisation d'une partie de `l'analyse standard`, cela nous incite donc fortement à nous placer dans un cadre classique. Pour ces deux principales raisons, nous avons choisi *d'axiomatiser classiquement* les nombres réels en tant que `corps commutatif, ordonné, archimédien et complet`.

## 1.3 Rappels préliminaires sur la théorie des corps

Dans cette section, nous allons tenter de définir pas à pas ce qu'est un corps commutatif ordonné archimédien et complet. Nous ne rappellerons pas les définitions de `commutativité, associativité, distributivité et élément neutre`.

Nous commencerons par donner les définitions relatives aux corps commutatifs.

**Définition 1.3.1 (monoïde)** $(A, \#)$ *est un monoïde si* $A$ *possède un élément neutre et si* $\#$ *est associative.*

**Définition 1.3.2 (symétrisable)** *Soit* $(A, \#)$ *un monoïde d'élément neutre* $e$.
*Un élément* $b$ *de* $A$ *est symétrisable pour* $\#$ *si le système* $b\#x = x\#b = e$ *admet une solution.*

**Définition 1.3.3 (monoïde abélien)** $(A, \#)$ *est un monoïde abélien si* $(A, \#)$ *est un monoïde et* $\#$ *est commutative.*

**Définition 1.3.4 (groupe (resp. abélien))** $(A, \#)$ *est un groupe (resp. abélien) si* $(A, \#)$ *est un monoïde (resp. abélien) et tout élément de* $A$ *est symétrisable.*

**Définition 1.3.5 (anneau)** $(A, \#, *)$ *est un anneau si* $(A, \#)$ *est un groupe abélien,* $*$ *est associative dans* $A$ *et* $*$ *est distributive par rapport à* $\#$.

**Définition 1.3.6 (anneau commutatif)** $(A, \#, *)$ *est un anneau commutatif si* $(A, \#, *)$ *est un anneau et* $*$ *est commutative dans* $A$.

**Définition 1.3.7 (corps (resp. commutatif))** *Notons* $nz(A)$ *l'ensemble* $A$ *privé de l'élément neutre de* $A$ *pour* $\#$[7].
$(A, \#, *)$ *est un corps commutatif si* $(A, \#, *)$ *est un anneau (resp. commutatif) et* $(nz(A), *)$ *est un groupe.*

**Remarque 1.3.1** *L'élément neutre pour* $*$ *n'apparaît qu'avec la notion de corps. Notons néanmoins qu'il est également possible de l'obtenir "plus tôt" en utilisant un anneau unitaire plutôt qu'un anneau.*

Nous pouvons maintenant définir un corps ordonné, puis archimédien et enfin complet.

[7] $nz(A)$ se note également $A^*$ lorsque $\#$ représente le $+$

**Définition 1.3.8 (corps ordonné)** *Un corps $(A, \#, *)$ est dit totalement ordonné s'il est muni d'une relation d'ordre totale (noté $<$) telle que $<$ :*

1. *est antisymétrique*
2. *est transitive*
3. *est compatible pour $\#$, c'est-à-dire*
   $\forall r, x, y \in A,\ x < y \Rightarrow r\#x < r\#y$
4. *est compatible pour $*$, c'est-à-dire*
   $\forall r, x, y \in A,\ e < r$ *et* $x < y \Rightarrow r * x < r * y$ *où $e$ est l'élément neutre de $\#$*

**Définition 1.3.9 (corps ordonné archimédien)** *Un corps ordonné $A$ est dit archimédien si, pour tout $r \in A$ il existe un entier[8] $n$ vérifiant $r < n$.*

**Définition 1.3.10 (corps ordonné complet)** *Soit $E$ une partie non vide et majorée de $A$. Un corps ordonné $A$ est dit complet si l'ensemble des majorants de $E$ admet un plus petit élément (appelé borne supérieure de $E$).*

Nous allons maintenant appliquer ces définitions au corps des réels ($\mathbb{R}$) et ainsi les formaliser dans Coq. Les symboles $\#$ et $*$ seront respectivement instanciés par $+$ et $\times$, leur élément neutre associé sont 0 et 1. Nous noterons que tout corps archimédien complet étant isomorphe à $\mathbb{R}$ (une preuve en est donnée dans [57]), une formalisation de la structure abstraite donnée par les théorèmes ci-dessus équivaut à une formalisation du corps des réels.

## 1.4 Choix d'une sorte pour $\mathbb{R}$

Avant toute chose, lorsque l'on décide d'axiomatiser une théorie, il faut commencer par définir le type de données principal, en l'occurrence, dans notre cas, le type `R`, représentant $\mathbb{R}$. Dans `Coq`, nous avons le choix entre trois sortes, c'est-à-dire que nous pouvons poser un des trois axiomes suivants :

```
R:Prop.
R:Set.
R:Type.
```

Commençons par énoncer les différences entre ces trois sortes.

### 1.4.1 Les sortes de Coq

#### La sorte Prop

`Prop` est l'univers des propositions logiques. Un objet de type `Prop` est appelé une *proposition*. D'autre part, `Prop` est une sorte *imprédicative*, c'est-à-dire que le terme `(P :Prop)P→ P` est également de type `Prop`. Dans le cas particulier des propositions, le fait que `Prop` soit imprédicative ne pose pas de problème lorsqu'il s'agit d'en donner une interprétation : une proposition $P$ est interprétée par sa valeur de vérité (le modèle d'oubli des preuves, communément appelé *proof irrelevance*) .

[8] Cette définition demande un plongement de l'entier $n$ dans le corps $A$ afin de traiter l'inégalité $r < n$. Nous verrons, en particulier, lors de la formalisation faite en `Coq` (page 21) comment plonger les entiers dans le corps des réels.

Un autre point de comparaison, qui nous intéresse particulièrement pour la suite, est la présence ou non de schémas d'élimination. Il n'y a pas d'élimination forte dans `Prop`, c'est-à-dire que nous ne sommes pas capables de construire d'objets `Set` ou `Type` à partir d'une proposition. Dans la pratique, cela signifie qu'il est impossible, par exemple, de récupérer le témoin d'un théorème d'existence ou de destructurer une disjonction lorsqu'il s'agit de montrer une propriété autre qu'une proposition (cf. exemple 1.4.1 dans `Coq`). On pourra se référer à l'annexe D (section D.4.1) ou à [91], [69] pour plus de détails.

```
1 subgoal

  A : Prop
  H : A\/~A
  ========================
    {A}+{~A}

elim_ou < Elim H.
Error: or_rec not declared
```

**Exemple 1.4.1 :**
**pas d'élimination forte dans `Prop`.**

Nous apercevons déjà ici le fait qu'une propriété contenant une information calculatoire ne peut être montrée à partir d'une proposition.

#### La sorte `Set`

La sorte `Set` a été définie pour être le type des spécifications. Cela inclut les programmes et les ensembles usuels tels que les booléens, les entiers, les listes etc. `Set` est également une sorte imprédicative , c'est-à-dire que `(A :Set)A→ A` est de type `Set`. Contrairement au cas de `Prop`, trouver un modèle satisfaisant pour `Set` est plus difficile. En effet, le type `nat` étant non vide et de type `Set`, dans un modèle vérifiant la "proof irrelevance", il est interprété par un singleton. De ce fait, la proposition $0 = 1$ est valide dans un tel modèle. Il est donc impossible de garder ce modèle si l'on veut poser l'axiome $0 \neq 1$, ou si l'on veut poser un principe d'élimination forte qui permette de démontrer cette proposition. Une autre approche consiste à ne pas prendre un modèle vérifiant la "proof irrelevance", mais à construire un modèle en utilisant des idées issues de la théorie de la calculabilité et de la réalisabilité. L'inconvénient est alors que l'on ne peut plus poser l'existence de fonctions non calculables. Signalons enfin un travail récent [63], qui permet de construire des modèles en utilisant une méthode issue de la théorie des espaces cohérents [40].
La sorte `Set` a une élimination forte, ce qui signifie qu'il est possible, par exemple, de récupérer le témoin d'un théorème d'existence définie à valeur dans `Set`, contrairement au cas de `Prop` vu précédemment.

#### La sorte `Type`

Les sortes elles-mêmes sont des termes ordinaires. De ce fait, les sortes doivent aussi avoir un type. Puisque poser qu'un type `A` est lui-même de type `A` est incohérent [41], il existe dans `Coq` une hiérarchie infinie d'univers notée `Type`. Par exemple, nous aurons pour tout entier `i` la propriété suivante : `Type`(i) est de type `Type`(i+1)[9]. Cette méthode permet

[9]Notons que, dans `Coq`, les indices d'univers ne sont pas visibles par l'utilisateur.

de pallier au problème du paradoxe de Girard [40]. De ce fait, on voit aisément que la sorte `Type` est *prédicative*, c'est-à-dire, en faisant apparaître les indices : `(S :Type(i)) S`$\rightarrow$ `S` est de type `Type`(i+1). Dans cette sorte, qui ne diffère de `Set` "que" par sa prédicativité, les modèles permettant d'en donner un interprétation sont les *modèles ensemblistes*.
La sorte `Type` a également un principe d'élimination forte.

### 1.4.2 Cas particulier : $\mathbb{R}$

En ce qui concerne la sorte `Prop`, la question ne se pose pas particulièrement. En effet, comme nous l'avons vu précédemment, `Prop` est la sorte permettant de représenter les propositions logiques. D'un point de vue théorique, il serait possible de poser `R` de type `Prop`, mais d'un point de vue pratique et intuitif, on ne le souhaite pas pour, au moins, deux raisons. La première est que l'ensemble des nombres réels n'est pas une proposition mais un ensemble de nombres. La seconde est que l'on souhaite profiter de l'élimination forte de `Coq`, ce qui n'est pas possible avec la sorte `Prop`.
Le problème se pose donc entre la sorte `Set` et la sorte `Type`. Au premier abord, `Set` semble tout-à-fait appropriée étant donné que c'est une "sorte calculatoire", permettant l'extraction et destinée à être le type des spécifications. Cela permettrait, qui plus est, de garder une cohérence avec les autres types définissant des nombres, tels que les entiers naturels `nat` et les entiers relatifs `Z`. Cependant, il se pose le problème de la cohérence d'une axiomatisation des réels avec `R :Set`. Si on pose l'existence des fonctions discontinues, ni les modèles d'oubli des preuves ni les modèles de réalisabilité ne permettent de montrer cette cohérence. Ce n'est que récemment que les modèles issus de la théorie des espaces cohérents[10] que nous avons cités ci-dessus ont permis de montrer un résultat de cohérence pour cette théorie. Ce résultat clarifie, en particulier, la problématique : c'est l'imprédicativité qui pose problème. La sorte `Set` imprédicative a été introduite dans le système `Coq` pour coder les types inductifs lorsque ceux-ci n'étaient pas encore primitifs[11]. Par exemple, une manière équivalente de définir le produit `Inductive prod [A:Set;B:Set]:= pair:(A,B:Set)A->B->A*B` en utilisant la propriété d'imprédicativité est `Definition prod [A:Set;B:Set]:= (C:Set)(A->B->C)->C`. Elle semble[12] avoir néanmoins une influence pas toujours souhaitée sur l'ensemble des autres sortes en général (de part le principe de cumulativité énoncé dans l'annexe D). En ce qui concerne le type des réels `R`, certaines skolémisations seraient alors interdites, par exemple, définir la borne supérieure d'une partie en écrivant `bs:(R->Prop)->R` avec `R:Set` casserait la *paramétricité* [71] de l'imprédicativité. Cela signifie qu'il serait possible de définir `R :Set` mais cela induirait certaines restrictions de la part de l'utilisateur, ce qui est à l'heure actuelle dangereux car mal compris. Cette question reste, à notre connaissance, non résolue. Le statut ambigu de l'imprédicativité peut se résumer par la figure 1.1, qui nous a été en grande partie suggéré par Alexandre Miquel.

Pour toutes ces raisons, nous avons préféré placer le type `R` des nombres réels dans `Type`. Il n'y a, de ce fait, aucun risque d'introduire une incohérence.

## 1.5 Les axiomes

À partir des définitions précédentes, axiomatiser le fait que $(\mathbb{R}, +, \times, <)$ est un corps commutatif ordonné archimédien et complet revient à se donner des constructeurs de base,

[10] voir thèse en cours d'Alexandre Miquel
[11] Actuellement, la propriété d'imprédicativité n'est utilisée que dans une seule contribution.
[12] Les effets de l'imprédicativité de `Set` au sein d'un système tel que `Coq` ne sont pas encore entièrement bien compris.

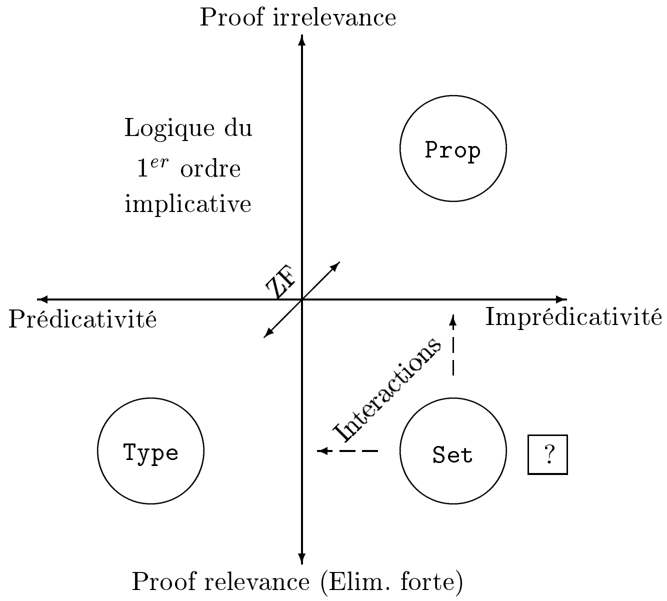



FIG. 1.1 – Cadran situant les Sortes de Coq suivant les 4 points cardinaux essentiels

puis les axiomes nécessaires et suffisants à ces constructeurs pour exprimer les propriétés voulues[13]. Nous avons posé pour cela les 17 axiomes ci-après. Les variables $x, y, z, r$ sont de type réel, la variable $n$ est de type entier. Les deux prédicats "majoré" et "est une borne sup de" sont énoncés séparément en tant que *définition* :

**Remarque 1.5.1 (sur les axiomes 16 et 17)**
*- Dans l'axiome 16 (Archimède fort), nous pouvons remarquer que* $n$ *est un entier tandis que* $r$ *est un reél. De ce fait, afin de pouvoir les comparer, nous devrons faire un plongement (ou injection) des entiers dans les reéls, ce qui nous permettra de ne manipuler que des réels.*
*- Dans l'axiome 17 (complétude), la partie* $E$ *doit être non vide. Ceci peut être codé en exprimant le fait qu'il existe un réel* $x$ *qui appartient à* $E$.

## 1.5.1 Les opérateurs

Commençons par le type de $\mathbb{R}$ :

```
Parameter R:Type.
```

Définissons maintenant les paramètres qui nous seront utiles pour formaliser le fait que $\mathbb{R}$ est corps commutatif (définition 1.3.7) :

```
Parameter R0:R.           (* le 0 des réels *)
Parameter R1:R.           (* le 1 des réels *)
Parameter Rplus:R->R->R.  (* le plus *)
Parameter Rmult:R->R->R.  (* le multiplié *)
Parameter Ropp:R->R.      (* l'opposé *)
Parameter Rinv:R->R.      (* l'inverse *)
```

[13]Nous pourrons nous référer, par exemple, à [27, 62]

| | |
|---|---|
| 1 | $x+y=y+x$ |
| 2 | $(x+y)+z=x+(y+z)$ |
| 3 | $x+(-x)=0$ |
| 4 | $0+x=x$ |
| 5 | $x\times y=y\times x$ |
| 6 | $(x\times y)\times z=x\times(y\times z)$ |
| 7 | $x\neq 0\Rightarrow x^{-1}\times x=1$ |
| 8 | $1\times x=x$ |
| 9 | $1\neq 0$ |
| 10 | $x\times(y+z)=(x\times y)+(x\times z)$ |
| 11 | $x<y\vee x=y\vee x>y$ |
| 12 | $x<y\Rightarrow\neg y<x$ |
| 13 | $x<y\Rightarrow y<z\Rightarrow x<z$ |
| 14 | $x<y\Rightarrow r+x<r+y$ |
| 15 | $0<r\Rightarrow x<y\Rightarrow r\times x<r\times y$ |
| 16 | $\exists n, n>r\wedge n-r\leq 1$ |
| 17 | $E$ majoré $\Rightarrow\exists x, x\in E\Rightarrow\exists r, r$ est une borne sup de $E$ |

Nous avons fait le choix de prendre une fonction *totale* pour représenter la fonction *inverse*. Nous aurions pu la définir comme une fonction partielle en posant

```
Parameter Rinv2:(x:R)``x<>0``->R.
```

c'est-à-dire une fonction où il serait nécessaire d'apporter, à chaque utilisation, une preuve que l'élément est non nul. En employant la première méthode, il nous est, bien sûr, possible d'écrire 1/0. Cela ne pose pas de problème. D'un point de vue sémantique, nous pouvons donner n'importe quelle valeur réelle comme interprétation de 1/0. Par contre, la cohérence va bien être préservée grâce à l'axiome de simplification suivant :

```
Axiom Rinv_l:(r:R)``r<>0``->``(1/r)*r==1``.
```

Cet axiome exige la preuve que $r$ soit non nul pour pouvoir simplifier $(1/r)*r$.
Pour ce qui est de la seconde méthode, il faudrait poser l'axiome suivant [14] :

```
Axiom Rinv2_l:(r:R)(p:``r<>0``)``(Rinv2 r p)*r==1``.
```

Nous voyons ici que l'hypothèse de non nullité de $r$ est également nécessaire, mais, cette fois, pour pouvoir ne serait-ce qu'écrire l'axiome correspondant. Cette dernière méthode, bien que se rapprochant plus de la définition mathématique usuelle, est extrêmement lourde à manier dans un système de preuves formelles. Pour nous en convaincre, récapitulons les deux méthodes de formalisation (voir notations 1.5.1, en utilisant une syntaxe mathématique usuelle) afin de nous attarder un instant sur l'exemple suivant :

| Formalisation 1 (Coq) | Formalisation 2 |
|---|---|
| $Rinv : R\rightarrow R$ | $Rinv2:\forall x:R.(x\neq 0)\rightarrow R$ |
| $Rinv\_l:\forall r:R.r\neq 0\rightarrow(1/r)*r=1$ | $Rinv2\_l:$ |
| | $\forall r:R.\forall p:r\neq 0.(Rinv2\ r\ p)*r=1$ |

**notations 1.5.1**

[14]Nous nous passerons de syntaxe pour Rinv2 dans ce cas, puisque nous sommes dans un cadre hypothétique de définition. Les doubles quotes font partie de la syntaxe des réels en Coq (voir annexe D) et, en particulier, "2" représente (Rplus R1 R1).

**Exemple 1.5.1 (Utilisation de `Rinv`)** *Regardons ce qui se passe lors de l'énoncé et de la démonstration, sans utiliser de tactique d'automatisation, de la propriété*
$\forall x \ 1/x = 2 \times 1/2x$.

- *Avec la formalisation 1 :*
  `Goal (x:R)``x<>0``->``1/x==2*1/(2*x)``.`
  *Pour montrer ce but, nous avons besoin du lemme et des hypothèses suivantes :*
  - *Rinv_Rmult* : $\forall r_1, r_2.$ *si* $r_1 \neq 0$ *et* $r_2 \neq 0$ *alors* $1/(r_1.r_2) = (1/r_1).(1/r_2)$, *lemme existant dans la librairie des réels que nous avons développée*
  - $x \neq 0$, *hypothèse du but*
  - $2 \neq 0$, *que nous pouvons montrer sans difficulté*
- *Avec la formalisation 2 :*
  *Nous avons plusieurs méthodes pour poser ce lemme. La première que nous énoncerons ici consiste à mettre les preuves nécessaires en hypothèse. Dans ce cas précis nous avons besoin que $x$ soit différent de 0 ainsi que $2x$. Cette méthode n'est pas satisfaisante dans la mesure où, à chaque application ultérieure de ce lemme, il faudra également donner la preuve que $2x \neq 0$ alors que seule celle de $x \neq 0$ suffit.*

  ```
  Goal (x:R; p:``x <> 0``; q:``2*x <> 0``)
           ``(Rinv2 x p) == 2*(Rinv2 (2*x) q)``.
  ```

  *La seconde consiste à donner le terme preuve que $2x \neq 0$ puisqu'il se déduit de l'hypothèse $x \neq 0$. Pour ce, il semble plus commode de poser et de prouver le lemme intermédiaire noté* `2x_neq0` *afin de l'utiliser lors de la définition de notre lemme.*

  ```
  Lemma 2x_neq0:(x:R)``x<>0``->``2*x<>0``.
  Goal (x:R; p:``x <> 0``)
         ``(Rinv2 x p) == 2*(Rinv2 (2*x) (2x_neq0 x p))``.
  ```

  *Une autre méthode, totalement illisible, à notre avis, consiste à donner directement la preuve que $2x \neq 0$ sans passer par un lemme intermédiaire. Mais quelle que soit la preuve que nous donnons, ce lemme est plus restrictif et il ne dit pas que le résultat est vrai pour une preuve autre que celle donnée.*
  *Pour montrer ces buts, nous avons besoin des mêmes outils que dans la formalisation 1.*

Comme nous avons pu le constater, c'est surtout au niveau de l'écriture et de la lisibilité même de la propriété à démontrer que se situe la différence.

Pour le lecteur étonné par la formalisation 1, on pourra remarquer que, la différence ne se situant pas au niveau des lemmes, les seuls lemmes pouvant êtres démontrés en utilisant `Rinv` et qui sont impossibles à écrire avec `Rinv2` sont les lemmes qui peuvent se réécrire de façon à obtenir une équation réflexive, par exemple, $\frac{1}{0} + x = x + \frac{1}{0}$ ou $\frac{1}{0} \times 0 = 0$, sans avoir fait aucune simplification entre numérateur et dénominateur. Le fait de pouvoir montrer que $1/0 = 1/0$ ne pose pas de problème particulier, de par la définition même de l'égalité de Leibniz. Notons également, qu'un formalisme similaire est adopté en théorie des ensembles. Nous nous référons à l'opérateur de choix (noté $\tau$) qui, quand on l'applique à un ensemble non vide, donne un élément de cet ensemble. Néanmoins, certains auteurs [9] autorisent son application à l'ensemble vide, à condition que l'axiome $P(\tau \ P)$ soit restreint à $P$ non vide.

Les deux formalisations précédentes ne sont pas les seules. En HOL, par exemple, un autre choix a été fait : la fonction inverse est également définie comme une fonction totale, mais la valeur 0 est attribuée à l'inverse de 0. Dans ce cas, en plus du lemme `Rinv_l` (nommé `REAL_MUL_RINV` en HOL), il est possible de montrer, entre autres, mais cette fois sans l'hypothèse $x \neq 0$, que $\forall x.1/(1/x) = x$. Nous pourrons nous référer à [47] pour plus de détails concernant cette méthode de formalisation.

Il nous reste à poser le constructeur pour les propriétés d'ordre. Un seul est nécessaire, l'ordre inférieur strict par exemple :

```
Parameter Rlt:R->R->Prop.
```

Les autres ($<$, $\leq$ et $\geq$ resp.) sont définis à partir de celui là :

```
Definition Rgt:R->R->Prop:=[r1,r2:R](Rlt r2 r1).
Definition Rle:R->R->Prop:=[r1,r2:R]((Rlt r1 r2)\/(r1==r2)).
Definition Rge:R->R->Prop:=[r1,r2:R]((Rgt r1 r2)\/(r1==r2)).
```

Ces huit constructeurs sont suffisants pour axiomatiser tout corps commutatif ordonné. Par commodité, nous avons choisi de skolémiser l'axiome d'Achimède au lieu de poser directement l'axiome sous sa forme existentielle. Nous définissons donc une fonction `up` qui représente l'entier juste supérieur à un réel donné en paramètre. Nous verrons par la suite son utilisation lors de la spécification de l'axiome lui-même.

```
Parameter up:R->Z.
```

### 1.5.2 Les axiomes correspondants

Pour la liste complète des axiomes, nous pourrons nous référer à l'annexe A. Dans cette section, nous présenterons surtout les axiomes présentant un intérêt particulier compte tenu de la formalisation choisie. Nous donnerons également un "échantillon" d'axiome pour chaque catégorie de propriété.

L'axiome suivant traduit la commutativité de l'addition. Nous avons le même pour la multiplication.

```
Axiom Rplus_sym:(r1,r2:R)``r1+r2==r2+r1``.
```

Nous ne nous attarderons par sur l'associativité ni sur la transitivité de l'addition par rapport à la multiplication.

L'axiome suivant donne sa sémantique à l'opposé.

```
Axiom Rplus_Ropp_r:(r:R)``r+(-r)==0``.
```

Celui correspondant à l'inverse a été donné précédemment.

En ce qui concerne les éléments neutres, voici l'axiome qui nous permet de définir 1 comme élément neutre de la multiplication. L'addition a son équivalent.

```
Axiom Rmult_1l:(r:R)``1*r==r``.
```

Avant d'aborder les axiomes de la relation d'ordre, il nous reste à traduire la propriété que $(\mathbb{R}^*, \times)$ est aussi un groupe. Pour cela il faut distinguer les éléments neutres de l'addition et de la multiplication :

```
Axiom R1_neq_R0:``1<>0``.
```

Pour que `Rlt` représente réellement l'opérateur strictement inférieur, il lui faut être accompagné, entre autres, d'axiomes tels que l'antisymétrie, la transitivité, la compatibilité avec l'addition et la multiplication. Mais c'est l'axiome d'*ordre total* qui va, tout particulièrement, nous intéresser. Pour cela, nous préférons utiliser le supérieur strict `Rgt`, afin de faciliter la lecture de l'axiome de totalité.

Une transcription en langage mathématique de cet axiome pourrait être :

$$\forall r_1, r_2.\ r1 < r2 \text{ ou } r1 = r2 \text{ ou } r1 > r2$$

En Coq nous avons alors le choix entre utiliser des disjonctions à valeur dans `Prop` ou dans `Type`, sachant que celles dans `Type` impliquent celles dans `Prop`. Nous avons choisi la solution dans `Type` suivante :

```
Axiom total_order_T:(r1,r2:R)(sumorT (sumboolT ``r1<r2`` r1==r2)
                                        ``r1>r2``).
```

où `sumorT` et `sumboolT` sont respectivement de type `Type->Prop->Type` et `Prop->Prop->Type` et représentant des disjonctions, au même titre que `sumor` et `sumbool` déjà définies dans Coq.

Les disjonctions sont à valeur dans la sorte `Type` afin de profiter, comme nous l'avons dit précédemment dans ce chapitre, du principe d'élimination forte tout en évitant tout risque de contradiction. En particulier, nous pourrons utiliser cet axiome pour construire des fonctions telles que le *min* ou le *max* de deux nombres, en exploitant ce principe à l'aide d'un `Cases` qui destructurera les *ou*. Cet axiome présente également des particularités. C'est une instance du *tiers exclu* portant sur l'égalité dans les nombres réels. Considérons plutôt le lemme suivant, prouvé équivalent (dans Coq) à notre axiome de totalité :

```
Lemma Req_EMT:(r1,r2:R)(sumboolT r1==r2 ``r1<>r2``).
```

Nous pouvons dire que cette propriété est une instanciation du tiers exclu dans la mesure où, de part l'impossibilité de montrer l'une des deux formes, notre seule issue est de l'interpréter classiquement. Ce n'est par exemple pas le cas pour les nombres entiers, où un lemme similaire est défini, puisque, grâce au principe de récurrence, il est possible de décider si pour tout $n$ et $m$ entiers, $n = m$ ou non [15].

L'axiome d'Archimède que nous avons posé est légèrement plus fort que la plupart des énoncés que l'on trouve dans les livres de mathématique c'est-à-dire juste l'existence d'un nombre entier strictement supérieur à tout réel. L'axiome suivant établit que, pour tout réel, il existe un entier qui lui est immédiatement et strictement supérieur[16]. Nous verrons comment cette définition nous permet de définir, en particulier, la fonction *partie entière* très simplement.

Nous remarquons que, pour énoncer cet axiome, nous utilisons à la fois les nombres entiers et les nombres réels. De ce fait, il est nécessaire de définir une fonction d'injection des entiers dans les réels. Pour cela nous définissons une fonction qui plonge les entiers naturels ($\mathbb{N}$) dans $\mathbb{R}$, puis les entiers relatifs ($\mathbb{Z}$) dans $\mathbb{R}$. Dans les deux cas nous utilisons les structures inductives de `nat` et `Z`. Par exemple, si nous prenons le cas des entiers naturels, nous aurons à formaliser en Coq la fonction d'injection $\iota : \mathbb{N} \to \mathbb{R}$ (notée `INR` en Coq) suivante :

$$\iota(n) = \begin{cases} 0_{\mathbb{R}} & \text{si } n = 0_{\mathbb{N}} \\ \iota(m) +_{\mathbb{R}} 1_{\mathbb{R}} & \text{si } n = Succ(m) \end{cases}$$

[15] il s'agit de `eq_nat_dec dans Coq`

[16] ce qui est, en général, un théorème et non un axiome.

Nous pourrons nous référer à l'annexe A, ou directement à la librairie diffusée dans le version courante V7.0 de Coq, pour la formalisation en Coq de ces deux fonctions d'injection.

L'axiome d'Archimède que nous avons posé est donc le suivant :

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

où IZR est une fonction d'injection définie précédemment, up le constructeur de Skolem représentant la fonction qui nous donne l'entier.

Sachant qu'un ensemble ordonné et minoré admet un plus petit élément, nous pouvons montrer simplement que ces deux formalisations sont équivalentes :

1. si $\forall r : \mathbb{R}, \exists z : \mathbb{Z}$ t.q. $z > r$ et $z - r \leq 1$ alors $\forall r : \mathbb{R}, \exists z : \mathbb{Z}$ t.q. $z > r$
   Trivial, il suffit de prendre la première propriété de l'hypothèse.
2. si $\forall r : \mathbb{R}, \exists z : \mathbb{Z}$ t.q. $z > r$ alors $\forall r : \mathbb{R}, \exists z : \mathbb{Z}$ t.q. $z > r$ et $z - r \leq 1$
   Soit l'ensemble des entiers $I(r)$ suivant : $\{z : \mathbb{Z}$ t.q. $z > r\}$. $I(r)$ est bien un ensemble ordonné et minoré. De ce fait, il admet un plus petit élément. Notons le $n$. $n$ vérifie bien la propriété.

Pour le développement en Coq, nous avons préféré poser directement l'axiome un peu plus fort uniquement par souci de commodité, car cela nous a permis d'éviter de formaliser les notions d'ensembles minorés admettant un plus petit élément.

Le dernier axiome, celui qui permet d'affirmer que nous sommes bien en train de formaliser, en particulier le corps des réels et non celui des rationnels par exemple, est l'axiome de complétude. Nous avons pu voir à la remarque 1.5.1 une manière ensembliste de coder le fait que $E$ doit être une partie non vide de $\mathbb{R}$. En Coq, la partie $E$ est vue comme le prédicat `E :R->Prop` et de ce fait, dire que $\exists x,\ x \in E$ s'écrit "il existe un x tel que `(E x)`" ou encore en Coq `(ExT [x :R] (E x))`. Nous pouvons alors énoncer l'axiome de complétude :

```
Axiom complet:(E:R->Prop)(bound E)->
              (ExT [x:R] (E x))->
              (ExT [m:R](is_lub E m)).
```

où bound, le prédicat *majoré*, et is_lub, le prédicat *est une borne supérieure*, étant ainsi défini :

```
Definition is_upper_bound:=[E:R->Prop][m:R]
    (x:R)(E x)->``x <= m``.

Definition bound:=[E:R->Prop]
    (ExT [m:R](is_upper_bound E m)).

Definition is_lub:=[E:R->Prop][m:R]
    (is_upper_bound E m)/\(b:R)(is_upper_bound E b)
     ->``m <= b``.
```

## 1.6 Les Propriétés et fonctions de base

Par souci de lisibilité, il est plus clair de définir les opérateurs pour le moins binaire et la division [17] :

[17] Dans la pratique, ces définitions, ainsi que celles concernant les trois opérateurs d'ordre autre que Rlt, sont faites dans le fichier contenant les déclarations des opérateurs. Ainsi, nous sommes capables d'élaborer

```
Definition Rminus:R->R->R:=[r1,r2:R](Rplus r1 (Ropp r2)).
Definition Rdiv:R->R->R:=[r1,r2:R](Rmult r1 (Rinv r2)).
```

### 1.6.1 Les propriétés de base

Nous ne présenterons pas une grande quantité de propriétés de base, et nous pourrons nous référer directement à la librairie distribuée de Coq [86] ou à l'annexe B pour plus de détails.

A partir de l'axiomatisation précédente, nous verrons qu'il est possible, mais également nécessaire, de montrer un certain nombres de propriétés "triviales". Par "trivial", nous nous référons à des propriétés élémentaires des corps commutatifs ordonnés. Par exemple, la propriété suivante[18], qui n'est pas un axiome, est bien évidemment très utile.

**Exemple 1.6.1** *Montrons, à partir de nos 17 axiomes, que* $\forall r.\ r \times 0 = 0.$ [19]
*Avant de montrer ce lemme, il faut commencer par montrer le lemme de compatibilité*
`r_Rplus_plus` *:* $\forall r, r_1, r_2.\ r + r_1 = r + r_2 \rightarrow r_1 = r_2$
*On veut montrer l'égalité* $r_1 = r_2$ *sous l'hypothèse* $H : r + r_1 = r + r_2$.
*On procède par étapes de réécriture successives à partir de l'égalité réflexive :*
$-r + (r + r1) = -r + (r + r1)$
$-r + (r + r1) = -r + (r + r2)$ *(par H)*
$(-r + r) + r1 = (-r + r) + r2$ *(axiome* `Rplus_assoc`*)*
$(r + -r) + r1 = (r + -r) + r2$ *(axiome* `Rplus_sym`*)*
$0 + r1 = 0 + r2$ *(axiome* `Rplus_Ropp_r`*)*
$r1 = r2$ *(axiome* `Rplus_Ol`*)*

*Nous pouvons maintenant prouver notre lemme initial. Il suffit d'appliquer* `r_Rplus_plus` *afin d'ajouter r des deux cotés de l'égalité, puis de suivre le cheminement suivant :*
$r + r \times 0 = r + 0$
$r \times 1 + r \times 0 = r + 0$ *(*`Rmult_1l`*)*
$r(1 + 0) = r + 0$ *(*`Rmult_Rplus_distr`*)*
$r \times 1 = r$ *(*`Rplus_Ol` *deux fois)*
$r = r$ *(*`Rmult_1l`*)*

Comme nous avons pu le constater dans cet exemple, les axiomes seuls ne suffisent pas à faire un développement utilisable. Actuellement, nous avons plus ou moins 250 lemmes triviaux qui viennent enrichir cette librairie (cf. annexe B). Ces lemmes ont, dans un premier temps, été prouvés manuellement (c'est-à-dire comme dans l'exemple ci-dessus), puis, grâce à l'élaboration de nouvelles tactiques dans Coq telles `Ring` ou `Field`, dont nous parlerons un peu plus tard, beaucoup d'entre eux sont, maintenant, prouvés automatiquement.

Dans l'exemple ci-après, nous verrons le cheminement suivi pour prouver une grande majorité des lemmes faisant intervenir les relations d'ordre. Ces preuves diffèrent des preuves faites sur les autres types de données arithmétiques tels les nombres entiers ou relatifs. Prouvons que $0 \neq 2$. Dans le cas où 0 et 2 sont de type `nat` ou `Z`, la preuve est immédiate en utilisant la tactique `Discriminate` [86] puisque ces deux types sont définis inductivement en utilisant respectivement les constructeurs O, S et ZERO, POS, NEG. Par contre, le

une grammaire pour tous les opérateurs, ce qui permet une écriture plus mathématique de l'axiomatisation et de toutes les propriétés qui s'en suivent.

[18] dans la librairie de Coq, la plupart de ces preuves sont automatisées grâce à la tactique `Ring`.

[19] Pour le nom des axiomes utilisés, on pourra voir l'annexe A.

type `R` n'est pas défini inductivement et les constantes telles que 2 ou $-2$, par exemple, ne représentent qu'un sucre syntaxique pour `(Rplus R1 R1)` et `(Ropp (Rplus R1 R1))`. Nous devons donc procéder de la façon suivante :

**Exemple 1.6.2 (Preuve de $0 \neq 2$ dans $\mathbb{R}$)** *L'idée est de montrer que $0 < 2$, ce qui implique bien notre but (`Rlt_not_eq`). En appliquant le lemme de transitivité du $<$ (`Rlt_trans`), montrer $0 < 2$ revient à montrer $0 < 1$ et $1 < 2$. Nous ne ferons pas la preuve ici que $0 < 1$ car il s'agit du lemme `Rlt_R0_R1` prouvé séparément. Montrer que $1 < 2$ se fait par compatibilité de l'addition à partir de $0 < 1$ :*
$1 + 0 < 1 + 1$ *(`Rlt_compatibility`)*
$1 < 2$ *(`Rplus_Or`)*

*Le script* Coq *est le suivant :*

```
Goal ``0<>2``.

Apply Rlt_not_eq.
Apply (Rlt_trans ``0`` ``1`` ``2``).
Apply Rlt_R0_R1.
Pattern 1 ``1``;Rewrite <-(Rplus_Or ``1``).
Apply Rlt_compatibility.
Apply Rlt_R0_R1.
```

Cet exemple montre, à nouveau, la nécessité de développer des tactiques automatiques. Ce besoin est particulièrement visible dans le cas d'une axiomatisation, puisque nos seuls outils sont alors les tactiques de réécriture. Les règles de conversion habituelles de Coq ($\beta$, $\delta$, $\iota$ décrites à l'annexe D) ne sont d'aucune aide dans une axiomatisation. Nous verrons pourquoi au chapitre 4, lorsque nous détaillerons l'automatisation pour les nombres réels.

## 1.6.2 Les fonctions de base

Dans cette section nous énoncerons quelques définitions et propriétés de fonctions mathématiques usuelles définies en Coq.

#### Le min et le max

Ces deux fonctions étant définies de façon symétrique, nous nous intéresserons, par exemple, à la fonction calculant le minimum. Sa définition est la suivante :

$$min(x,y) = \begin{cases} x & \text{si } x \leq y \\ y & \text{sinon} \end{cases}$$

En Coq, une manière de définir des fonctions définies par cas est d'utiliser un `Cases`. Nous noterons que c'est lors de ces définitions que nous utilisons les lemmes d'ordre total définis à valeur dans la sorte `Type`. En effet, comme nous avons pu le voir au paragraphe 1.4.1, nous utilisons ici le principe d'élimination forte[20] sur le lemme `total_order_Rle` pour destructurer le "ou", suivant que $x \leq y$ ou que $\neg(x \leq y)$ :

```
Lemma total_order_Rle:
  (r1,r2:R)(sumboolT ``r1<=r2`` ~(``r1<=r2``)).
```

[20] Nous pourrons nous référer à la règle de typage du `Cases` (annexe D)

```
Definition Rmin :R->R->R:=[x,y:R]
  Cases (total_order_Rle x y) of
    (leftT _)  => x
  | (rightT _) => y
  end.
```

où le lemme `total_order_Rle` se montre aisément en utilisant l'axiome `total_order_T`.

La fonction `max` est définie de manière symétrique.

Une des propriétés qui a été prouvée et qui sera souvent utilisée lors des preuves concernant des propriétés sur les limites et les dérivées (voir chapitre 2) est la suivante :

```
Lemma Rmin_Rgt:(r1,r2,r:R)(Rgt (Rmin r1 r2) r)<->
    ((Rgt r1 r)/\(Rgt r2 r)).
```

Cette preuve se fait par cas, en utilisant la tactique `Case` qui génère les deux cas suivant que $r1 \leq r2$ ou non.

### La valeur absolue

La fonction valeur absolue se définit de manière similaire aux fonctions `min` et `max`. En effet, nous avons :

$$|x| = \begin{cases} -x & \text{si } x < 0 \\ x & \text{sinon} \end{cases}$$

Nous avons donc deux cas suivant que $x < 0$ ou que $x \geq 0$, ce que nous avons défini en Coq en montrant le lemme :

```
Lemma case_Rabsolu:(r:R)(sumboolT (Rlt r R0) (Rge r R0)).
```

Nous avons alors la définition suivante pour la valeur absolue :

```
Definition Rabsolu:R->R:=
      [r:R](Cases (case_Rabsolu r) of
              (leftT  _) => ``-r``
             |(rightT _) => r
            end).
```

Cette fonction est utilisée dans une grande majorité de définitions et de preuves concernant la limite et la dérivée. À cet effet nous avons prouvé un certain nombre de propriétés. Les preuves se font pour la plupart par cas. Parmi ces propriétés nous pouvons citer entre autres :

$$|0| = 0$$

$$x < 0 \Rightarrow |x| = -x$$

$$x \geq 0 \Rightarrow |x| = x$$

$$0 \leq |x|$$

$$\big|\ |x|\ \big| = |x|$$

$$|x - y| = |y - x|$$

$$|x \times y| = |x| \times |y|$$

$$x \neq 0 \Rightarrow |x^{-1}| = |x|^{-1}$$
$$|x| - |y| \leq |x - y|$$
$$|x + y| \leq |x| + |y|$$

Nous allons donner une preuve de cette dernière inégalité (inégalité triangulaire), qui fait une cinquantaine de lignes en Coq : **Preuve** de $|x+y| \leq |x| + |y|$ :
Par cas :

1. $x + y < 0$, $x < 0$, $y < 0$. Nous devons alors montrer que $-(x+y) \leq -x + -y$.
   Par réécriture nous avons que $-(x+y) = -x + -y$, ce qui nous ramène à montrer que $-x + -y \leq -x + -y$, trivial d'après la définition de $\leq$ .
2. $x + y < 0$, $x < 0$, $y \geq 0$. Nous devons alors montrer que $-(x+y) \leq -x + y$.
   Par réécriture nous avons que $-(x+y) = -x + -y$. En utilisant l'axiome de compatibilité (ce qui revient ici à simplifier l'inégalité par $-x$), nous devons alors montrer que $-y \leq y$. Or $y \geq 0$ implique que $-y \leq 0$. L'inégalité est prouvée par transitivité.
3. $x + y < 0$, $x \geq 0$, $y < 0$. Nous devons alors montrer que $-(x+y) \leq x + -y$.
   Symétrique au cas précédent.
4. $x + y < 0$, $x \geq 0$, $y \geq 0$. Nous devons alors montrer que $-(x+y) \leq x + y$.
   Par contradiction. $y \geq 0$ $x \geq 0$ implique que $x + y \geq 0$ qui elle-même implique que $x + y \not< 0$ qui contredit la première hypothèse.
5. $x + y \geq 0$, $x < 0$, $y < 0$. Nous devons alors montrer que $x + y \leq -x + -y$.
   Par contradiction, symétrique au cas précédent.
6. $x + y \geq 0$, $x < 0$, $y \geq 0$. Nous devons alors montrer que $x + y \leq -x + y$.
   En utilisant l'axiome de compatibilité (ce qui revient ici à simplifier l'inégalité par $y$), nous devons alors montrer que $x \leq -x$. Or $x < 0$ implique que $-x > 0$. L'inégalité est prouvée par transitivité et par le fait que $a < b \Rightarrow a \leq b$.
7. $x + y \geq 0$, $x \geq 0$, $y < 0$. Nous devons alors montrer que $x + y \leq x + -y$.
   Symétrique au cas précédent.
8. $x + y \geq 0$, $x \geq 0$, $y \geq 0$. Nous devons alors montrer que $x + y \leq x + y$.
   Trivial d'après la définition de $\leq$ .

■

### La fonction puissance

La fonction puissance, où l'exposant $n$ est un entier naturel, est définie de manière récursive sur le type des naturels `nat` comme suit :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \times x^m & \text{si } n = Succ(m) \end{cases}$$

Ce qui s'écrit en Coq :

```
Fixpoint pow [r:R;n:nat]:R:=
  Cases n of
    O      => R1
   |(S n) => (Rmult r (pow r n))
  end.
```

La plupart des preuves concernant cette fonction se font par récurrence sur la puissance entière. Prouvons, par exemple, que $x \times x^n = x^{(S\ n)}$ :

1. $n = 0$ : $x \times 1 = x \times 1$ trivial
2. supposons la propriété vraie pour $n$ et prouvons-la pour $(S\ n)$ : nous avons donc l'hypothèse de récurrence $x \times x^n = x^{(S\ n)}$, montrons que $x \times x^{(S\ n)} = x^{(S\ (S\ n))}$ ce qui se réduit, d'après la définition de la puissance, en $x \times x \times x^n = x \times x^{(S\ n)}$ puis en $x \times x \times x^n = x \times x \times x^n$.

Ce qui donne la preuve suivante en Coq, en utilisant la tactique `Simpl` qui applique les règles de réduction ($\beta\iota$ et $\delta$) que nous avons données dans l'annexe D.

```
Lemma tech_pow_Rmult:(x:R)(n:nat)
  ``x*(pow x n)==(pow x (S n))``.
Induction n; Simpl; Trivial.
Save.
```

## 1.7 Partie entière et partie fractionnaire

Dans cette section nous donnerons une formalisation des fonctions *partie entière* et *partie fractionnaire*. Ces fonctions seront particulièrement utilisées lors de la formalisation des propriétés géométriques nécessaires à la preuve du théorème que nous présenterons au chapitre 3.

### 1.7.1 Définitions

Dans toute la suite, nous noterons $\{r\}$ la partie fractionnaire de $r$ et $E(r)$ sa partie entière.

#### Partie entière

La définition mathématique de la fonction partie entière est la suivante :

**Définition 1.7.1 (partie entière)** *La partie entière d'un nombre réel $x$ est l'unique entier relatif $e$ vérifiant $e \leq x < e + 1$.*

En Coq nous pourrons définir cette fonction à partir de l'axiome d'Archimède. Pour ce faire, nous utiliserons le symbole de Skolem `up` que nous avons défini lors de l'axiomatisation. Rappelons, en effet, que `up(r)` représente l'entier immédiatement supérieur au réel `r`. De ce fait, la partie entière d'un nombre réel $r$ est exactement $up(r) - 1$[21] :

```
Definition Int_part:R->Z:=[r:R]`(up r)-1`.
```

Nous pouvons remarquer que le choix effectué lors de l'énoncé de l'axiome d'Archimède nous a permis de définir facilement la partie entière. En particulier, le fait d'avoir fait le plongement sur les entiers relatifs nous permet également de gérer la partie entière d'un nombre négatif

[21] On pourra remarquer que $up(r) - 1$ est un nombre relatif et que cela est visible syntaxiquement dans Coq par les simples quotes qui caractérisent ces nombres, tout comme les doubles quotes pour les nombres réels.

**Partie fractionnaire**

La partie fractionnaire se définit mathématiquement de la manière suivante :

$$\{r\} = r - E(r)$$

Contrairement à la partie entière qui renvoie un résultat de type entier, la partie fractionnaire rend un nombre réel. De ce fait, il est nécessaire de transformer $E(r)$ en un nombre réel en faisant une injection. Pour cela, nous utilisons la même fonction que celle utilisée pour définir l'axiome d'Achimède : `IZR`. La partie fractionnaire se définie donc en `Coq` ainsi :

```
Definition frac_part:R->R:=[r:R]``r-(IZR (Int_part r))``.
```

### 1.7.2 Propriétés

Dans cette section nous nous intéresserons particulièrement à une dizaine de propriétés concernant à la fois les parties entière et fractionnaire, qui sont indissociables des définitions. Tout comme une grande quantité de lemmes de base sur le corps des nombres réels est indispensable à cette bibliothèque (par exemple $x \times 0 = 0$), les lemmes présentés ci-après sont indispensables à la formalisation de ces deux fonctions, dans le sens où pour utiliser confortablement une nouvelle notion il est souvent nécessaire de montrer des propriétés élémentaires s'y rapportant.

**Preuve** de $0 \leq \{r\} < 1$ :
- Commençons par montrer que $0 \leq \{r\}$ :
  En remplaçant la partie fractionnaire par sa définition c'est-à-dire par $r - E(r)$ puis la partie entière par $up(r) - 1$, nous nous ramenons à montrer que $0 \leq r - up(r) + 1$. Cette inéquation peut se réécrire en $up(r) - r \leq 1$, ce qui est exactement la seconde propriété énoncée dans notre axiome d'Archimède.
- Montrons maintenant que $\{r\} < 1$ :
  Nous procédons de la même manière que précédemment. Nous nous ramenons donc à montrer que $r - up(r) + 1 < 1$, qui se réécrit en $r < up(r)$ qui est exactement la propriété principale énoncée dans notre axiome d'Archimède.

■

Les deux propriétés ci-après sont, ce que nous appellerons, des lemmes techniques, issues du théorème d'Archimède. Dans tous les lemmes qui vont suivre, $z, z_i \in \mathbb{Z}$ et $r, r_i \in \mathbb{R}$ avec $i = 0, 1, 2, ...$

**Preuve** de $r < z \leq r + 1 \Rightarrow z = up(r)$ :
D'après notre axiome d'Archimède, nous avons que $r < up(r) \leq r + 1$. D'autre part, nous avons la propriété suivante[22], qui traduit le fait qu'il n'y a qu'un seul entier entre un réel $r_0$ et $r_0 + 1$ : si $r_0 < z \leq r_0 + 1$ et $r_0 < z_0 \leq r_0 + 1$ alors $z = z_0$. En appliquant cette propriété avec l'axiome d'Archimède et l'hypothèse du lemme que nous voulons montrer, on obtient bien le résultat voulu. ■

**Preuve** de $z \leq r < z + 1 \Rightarrow up(r) = z + 1$ :
En ajoutant 1 des deux cotés de l'inégalité $z \leq r$ on obtient $1 + z \leq 1 + r$ puis, en utilisant l'hypothèse, $r < z + 1 \leq r + 1$ ce qui nous permet d'appliquer le lemme précédent pour montrer le résultat. ■

[22]prouvée et faisant partie des lemmes de base de la bibliothèque

Les huit propriétés suivantes permettent de compléter cette formalisation concernant les fonctions partie entière et partie fractionnaire.

**Preuve** de $\{r_1\} \geq \{r_2\} \Rightarrow E(r_1 - r_2) = E(r_1) - E(r_2)$ :
Nous avons montré précédemment que $\forall r,\ 0 \leq \{r\} < 1$. Nous avons donc que $0 \leq \{r_1\} < 1$ et que $0 \leq \{r_2\} < 1$, ce qui implique que $-1 < \{r_1\} - \{r_2\} < 1$. En utilisant l'hypothèse $\{r_1\} \geq \{r_2\}$ on en déduit que $0 \leq \{r_1\} - \{r_2\} < 1$. En remplaçant $\{r_i\}$, $i = 1, 2$ par leur définition c'est-à-dire par $r_i - E(r_i)$ on obtient après quelques opérations élémentaires que $E(r_1) - E(r_2) \leq r_1 - r_2 < E(r_1) - E(r_2) + 1$. Nous pouvons alors appliquer le second lemme technique montré précédemment, ce qui nous donne que $up(r_1 - r_2) = E(r_1) - E(r_2) + 1$. D'après la définition de la partie entière on a $up(r_1 - r_2) = E(r_1 - r_2) + 1$ d'où le résultat. ■

Ce lemme permet de montrer de manière quasi immédiate que
$\{r_1\} \geq \{r_2\} \Rightarrow \{r_1 - r_2\} = \{r_1\} - \{r_2\}$ :

**Preuve**
Il suffit de remplacer les $\{r_i\}$, $i = 1, 2$ par leur définition c'est-à-dire par $r_i - E(r_i)$. On obtient $r_1 - r_2 - E(r_1 - r_2) = r_1 - r_2 - (E(r_1) - E(r_2))$. En simplifiant par $r_1 - r_2$ puis en multipliant l'égalité par $-1$ (lemmes de compatibilité) on se ramène à montrer que $E(r_1 - r_2) = E(r_1) - E(r_2)$ (cf. lemme précédent). ■

**Preuve** de $\{r_1\} < \{r_2\} \Rightarrow E(r_1 - r_2) = E(r_1) - E(r_2) - 1$ :
De manière similaire au cas précédent, on montre que
$E(r_1) - E(r_2) - 1 < r_1 - r_2 < E(r_1) - E(r_2)$.
Nous pouvons en déduire que $E(r_1) - E(r_2) - 1 \leq r_1 - r_2 < E(r_1) - E(r_2) - 1 + 1$ et donc, en utilisant le même lemme technique que précédemment on a $up(r_1 - r_2) = E(r_1) - E(r_2)$.
D'après la définition de la partie entière on a $up(r_1 - r_2) = E(r_1 - r_2) + 1$
d'où $E(r_1 - r_2) + 1 = E(r_1) - E(r_2)$. ■

Ce lemme permet de montrer simplement que
$\{r_1\} < \{r_2\} \Rightarrow \{r_1 - r_2\} = \{r_1\} - \{r_2\} + 1$.

**Preuve** de $\{r_1\} + \{r_2\} \geq 1 \Rightarrow E(r_1 + r_2) = E(r_1) + E(r_2) + 1$ :
Par hypothèse et en utilisant le propriété que $\forall r, 0 \leq \{r\} < 1$, on a $1 \leq \{r_1\} + \{r_2\} < 2$. En remplaçant les parties fractionnaires par leur définition, on obtient que
$E(r_1) + E(r_2) + 1 \leq r_1 + r_2 < E(r_1) + E(r_2) + 1 + 1$. Par application de notre lemme technique, on a $up(r_1 + r_2) = E(r_1) + E(r_2) + 2$ ce qui nous donne le résultat en remplaçant $up(r_1 + r_2)$ par sa valeur en fonction de la partie entière. ■

Ce lemme permet de montrer, comme précédemment, que
$\{r_1\} + \{r_2\} \geq 1 \Rightarrow \{r_1 + r_2\} = \{r_1\} + \{r_2\} - 1$

**Preuve** de $\{r_1\} + \{r_2\} < 1 \Rightarrow E(r_1 + r_2) = E(r_1) + E(r_2)$ :
De manière analogue au lemme concernant le cas symétrique, nous avons que
$0 \leq \{r_1\} + \{r_2\} < 1$. En remplaçant les parties fractionnaires par leur définition, on obtient que $E(r_1) + E(r_2) \leq r_1 + r_2 < E(r_1) + E(r_2) + 1$. Par application de notre lemme technique, on a $up(r_1 + r_2) = E(r_1) + E(r_2) + 1$ ce qui nous donne le résultat en remplaçant $up(r_1 + r_2)$ par sa valeur en fonction de la partie entière. ■

Ce qui nous permet de montrer de manière quasi immédiate que
$\{r_1\} + \{r_2\} < 1 \Rightarrow \{r_1 + r_2\} = \{r_1\} + \{r_2\}$.

## 1.8 Discussion

À partir du développement de cette axiomatisation et de sa librairie incluant environ 300 lemmes et fonctions élémentaires, au sens mathématique, nous allons formuler quelques remarques se rapportant aux nombres réels ainsi qu' aux spécificités introduites par le système Coq. Outre le choix de la formalisation pour les nombres réels (construction ou axiomatisation), une des premières questions à laquelle nous avons dû répondre a été le choix de la sorte de Coq à utiliser. Nous avons vu que ce problème est, en grande partie, encore non résolu car, malgré les nombreuses discussions informelles qui ont eu lieu à ce sujet, nous ne pouvons, à l'heure actuelle, qu'en présenter l'intuition. Coq étant un système basé sur la logique intuitionniste, se poser la question d'un développement classique ou pas est pertinent, et, de fait, nous pouvons nous demander dans quelle mesure la logique classique doit intervenir. Dans notre développement nous pouvons parfaitement identifier ce qui le rend classique. Nous pensons que ce point peut être intéressant dans l'optique d'un éventuel développement intuitionniste fait à partir de celui-là. Il y deux aspects qui en font une axiomatisation classique : l'axiome de totalité et l'adoption de l'égalité de Leibniz. Notons que la question de l'utilisation de l'opérateur de choix de Hilbert (comme dans HOL) ne se pose pas car, dans Coq, il introduit une inconsistance à cause de l'imprédicativité de `Set`. Le fait que notre formalisation ne soit pas intuitionniste nous a aussi permis de définir la fonction inverse comme une fonction totale. Nous avons pu voir que cette manière de procéder, bien qu'étant contraire à toute intuition mathématique, est bien plus commode à utiliser dans un système formel et n'introduit pas d'inconsistance.

Même si le développement complet de cette bibliothèque (incluant les développements présentés au chapitre suivant) a pris un temps significatif, nous pensons qu'axiomatiser les propriétés de corps est un compromis acceptable entre puissance d'utilisation (dans le sens où toutes les propriétés de l'analyse réelle classique sont démontrables) et effort de formalisation (par rapport à une construction, par exemple). Notons également que beaucoup de lemmes ainsi qu'une automatisation (nous en reparlerons au chapitre 4) sont nécessaires à un bon confort d'utilisation. Cette remarque est vraie de manière générale, mais plus encore lors d'une axiomatisation ou lorsque l'on ne peut bénéficier de commodités apportées par le type de donné (comme pour les types définis inductivement), ce qui est donc également le cas dans la construction des nombres réels en HOL. C'est en grande partie pour cette dernière raison qu'un développement axiomatique n'est pas forcément désavantageux (à condition de prendre soin de minimiser le nombre d'axiomes, ce qui diminue, de fait, le risque d'inconsistance de la théorie) par rapport à une construction, lorsqu'il ne s'agit pas du but principal mais seulement d'un outil nécessaire.

D'autres fonctions telles que la fonction factorielle, la somme ou la somme infinie ont été formalisées. Ces fonctions seront définies dans le chapitre suivant, lorsque nous étudierons la formalisation des fonction transcendantes.

Cette bibliothèque est également en évolution, de par l'intégration de contributions d'utilisateurs. Nous pouvons citer, par exemple, le développement concernant la fonction puissance de type $\mathbb{R} \to \mathbb{Z} \to \mathbb{R}$ fait par Laurent Théry.

A partir de cette bibliothèque, nous verrons qu'il est également possible de montrer des théorèmes de géométrie utilisant des nombres réels. Au chapitre 3, nous démontrerons, en particulier, un théorème non trivial ne se prêtant pas naturellement à une formalisation dans un système formel.

# Chapitre 2

# Formalisation de l'Analyse

Afin de nous rapprocher de l'analyse numérique et, en particulier, des notions nécessaires pour montrer la correction, en Coq, d'un algorithme de différentiation automatique, nous devons formaliser les principales définitions et propriétés de l'analyse. Nous nous intéresserons, entre autres, aux définitions concernant les limites, les dérivées, les fonctions sommes[1], les séries ainsi que les fonctions transcendantes.

## 2.1 La Limite

### 2.1.1 Définition

Plusieurs solutions s'offrent à nous pour formaliser la notion de limite. Nous nous intéresserons à la limite d'une fonction. Une première étape est soit de définir directement la limite dans le corps des réels pour une fonction à un argument, soit de commencer par une définition plus générale que nous pourrons instancier. Une seconde étape consiste à choisir la définition la plus adéquate en vue des notions que nous souhaitons définir à partir de cette définition.

Afin de formaliser la limite d'une manière assez générale, nous commencerons par définir la notion d'espace métrique.

**Définition 2.1.1 (Espace métrique)** *Un espace métrique est constitué d'un ensemble $E$ et d'une application $(x, y) \to d(x, y)$ de $E \times E \to \mathbb{R}_+$ ayant les propriétés suivantes : pour tout $x, y, z \in E$*

1. *symétrie : $d(x, y) = d(y, x)$*
2. *équivalence : $d(x, y) = 0 \Leftrightarrow x = y$*
3. *inégalité triangulaire : $d(x, y) \leq d(x, z) + d(z, y)$*

Cette définition fait intervenir l'espace $\mathbb{R}_+$. Néanmoins, définir un sous-espace de $\mathbb{R}$ en Coq nous amènerait à utiliser des outils supplémentaires tels que des définitions d'injection ou des coercions. Dans le souci d'éviter l'utilisation d'un sous-ensemble supplémentaire[2], nous avons modifié cette définition afin d'éviter de définir l'espace $\mathbb{R}_+$. Pour cela, nous

[1]certaines propriétés concernant ces fonctions sont le fruit d'un travail commun avec John Harrison.
[2]il y a déjà $\mathbb{N}$ et $\mathbb{Z}$ ayant leur fonction d'injection respectives

avons éliminé la restriction à $\mathbb{R}_+$ en rajoutant une propriété pour la fonction $d$ (positivité). Nous avons donc en Coq, en utilisant la structure de record :

```
Record Metric_Space:Type:= {
   Base:Type;
   dist:Base->Base->R;
   dist_pos:(x,y:Base)``(dist x y) >= 0``;
   dist_sym:(x,y:Base)(dist x y)==(dist y x);
   dist_refl:(x,y:Base)(dist x y)==``0`` <-> x==y;
   dist_tri:(x,y,z:Base)``(dist x y) <= (dist x z)+(dist z y)`` }.
```

Nous allons maintenant présenter les principales manières de définir la notion de limite, afin de choisir celle qui nous semble le mieux adaptée pour notre développement. Bien que toutes ces définitions concernent la même notion, certains critères sont plus ou moins commodes lorsqu'il s'agit d'une définition dans un système formel.
Une première méthode consiste à utiliser la notion de *voisinage* et de *point adhérent*. Les définitions qui suivent proviennent de [57].

**Définition 2.1.2 (Voisinage)** *On appelle voisinage d'un élément $x$ toute partie $V$ de $\mathbb{R}$ contenant un intervalle ouvert contenant $x$.*

**Définition 2.1.3 (Point adhérent)** *On dit qu'un élément $a$ est adhérent à un ensemble $A$ si chaque voisinage de $a$ contient au moins un point de $A$.*

**Définition 2.1.4 (Limite (avec voisinage))** *Soit un point $a$ adhérent à $A$. On dit que $f(x)$ tend vers $b$ lorsque $x$ tend vers $a$ si, à chaque voisinage $V$ de $b$ on peut associer un voisinage $U$ de $a$ tel que l'on ait $f(x) \in V$ pour tout $x \in U \cap A$.*

Comme nous pouvons le constater, cette formalisation demande de montrer des lemmes concernant des propriétés des notions de voisinage et d'adhérence, ce qui n'est pas nécessaire en utilisant plutôt le critère usuel suivant. Pour une explication complète sur l'équivalence des deux définitions, nous pourrons nous référer à [57].

**Définition 2.1.5 (Limite)** *Pour que $f(x)$ tende vers $b$ lorsque $x$ tend vers $a$ sur $X$, il faut et il suffit qu'à tout nombre $\varepsilon > 0$ on puisse associer un nombre $\alpha > 0$ tel que les relations $|x-a| < \alpha$ et $x \in X$ entraînent $|f(x) - b| < \varepsilon$.*

Cette définition comprend plusieurs variantes. En particulier les inégalités $|x - a| < \alpha$ et $|f(x) - b| < \varepsilon$ peuvent être des inégalités larges. Cette variante n'ayant pas d'incidence sur l'ensemble des propriétés qu'il est possible de démontrer, car équivalente, nous ne nous attarderons par dessus.
Une autre variante concerne également l'inégalité $|x - a| < \alpha$. Il s'agit de lui ajouter la condition $0 < |x - a|$, comme le font certains auteurs [11]. A ce sujet, citons un extrait de la thèse de John Harrison [47] :

> "Crudely speaking, the problem is that limits are not compositional : if $f(x) \to y_0$ as $x \to x_0$ and $g(y) \to z_0$ as $y \to y_0$, it may not be the case that $(g \circ f) \to z_0$ as $x \to x_0$. The reason is that the definition of limit :
>
> $$\forall\varepsilon > 0.\exists\delta > 0.\forall y.0 < |y - y_0| < \delta \Rightarrow |g(y) - z_0| < \varepsilon$$
>
> includes the extra property that $0 < |y - y_0|$, i.e. $y \neq y_o$. This is necessary since in many situations (e.g. the derivative) the function whose limit is being considered might be undefined or nonsensical at $y = y_o$."

Cette définition de la limite, également nommée *épointée*, est effectivement plus pratique pour définir la notion de dérivée que nous verrons à la section suivante. Néanmoins, nous avons préféré utiliser la définition 2.1.5 (à la Bourbaki). Elle permet de démontrer des propriétés supplémentaires sans toutefois compliquer de façon trop rédhibitoire la formalisation de la dérivée. Nous démontrerons, en particulier, la propriété de composition des limites, citée par John Harrison précédemment, sans introduire d'hypothèse de continuité. Un contre-exemple traitant de ce problème est également donné dans [11].

En Coq, dans l'espace métrique que nous avons défini, nous avons alors la définition de la limite suivante :

```
Definition limit_in:=
   [X:Metric_Space; X':Metric_Space; f:(Base X)->(Base X');
    D:(Base X)->Prop; x0:(Base X); l:(Base X')]
   (eps:R)(Rgt eps R0)->
   (EXT alp:R | (Rgt alp R0)/\(x:(Base X))(D x)/\
                (Rlt (dist X x x0) alp)->
                (Rlt (dist X' (f x) l) eps)).
```

La notion de limite est formalisée comme un *prédicat*. En effet, il ne s'agit pas de calculer une limite (en particulier, dans la mesure où nous ne savons pas si celle-ci existe), mais d'énoncer le fait que $l$ est bien la limite de la fonction $f$ en $x_0$ sur un domaine $D$. Ce dernier est également codé comme un prédicat unaire sur un ensemble (de type `R->Prop`). En particulier, l'expression mathématique $\forall x$ réel $\in D$, $P$ s'écrira en Coq `(x:R)(D x)->P`.

A partir de cette définition, nous pouvons définir, par exemple, la limite dans $\mathbb{R}$ pour des fonctions à un argument. Pour cela, il nous faut commencer par montrer que $\mathbb{R}$ est un espace métrique. Comme distance sur $\mathbb{R}$, nous prendrons la distance usuelle c'est-à-dire :

$$d(x, y) = |x - y|$$

définie en Coq de la manière suivante :

```
Definition R_dist:R->R->R:=[x,y:R](Rabsolu ``x-y``).
```

Montrons que $\mathbb{R}$ muni de la distance `R_dist` est un espace métrique tel que nous l'avons défini.

**Lemme 2.1.1 (positivité (`R_dist_pos`))** $|x - y| \geq 0$

**Preuve** Cette preuve est immédiate en appliquant, à $x - y$, une des propriété concernant la valeur absolue et montrée lors du développement les propriétés de base sur les nombres réels : $\forall x,\ 0 \leq |x|$ ■

**Lemme 2.1.2 (symétrie (`R_dist_sym`))** $|x - y| = |y - x|$

**Preuve** Par cas :
- $x - y < 0$ et $y - x < 0$ : montrons que $-(x - y) = -(y - x)$. Par contradiction sur les hypothèses.
- $x - y < 0$ et $y - x \geq 0$ : montrons que $-(x - y) = y - x$. Trivial.
- $x - y \geq 0$ et $y - x < 0$ : montrons que $x - y = -(y - x)$. Trivial.
- $x - y \geq 0$ et $y - x \geq 0$ : montrons que $x - y = y - x$. Par contradiction sur les hypothèses.

■

**Lemme 2.1.3 (équivalence (R_dist_refl))** $|x-y|=0 \Leftrightarrow x=y$

**Preuve**

1. Montrons que $|x-y|=0 \rightarrow x=y$. Par cas :
   - $x-y \geq 0$ : montrons que $x-y=0 \rightarrow x=y$. Ceci est une application immédiate d'une propriété du moins binaire Rminus.
   - $x-y<0$ : montrons que $-(x-y)=0 \rightarrow x=y$. Nous commençons par appliquer une propriété sur l'opposé Ropp ($r=0 \rightarrow \ -r=0$) afin d'obtenir, à partir de $-(x-y)=0$, la nouvelle hypothèse $x-y=0$, ce qui nous ramène au cas précédent.
2. Montrons que $x=y \rightarrow |x-y|=0$. Il suffit de remplacer $x$ par $y$ dans $|x-y|=0$ et d'utiliser la propriété montrée précédemment sur la valeur absolue $|0|=0$.

■

**Lemme 2.1.4 (inégalité triangulaire (R_dist_tri))** $|x-y| \leq |x-z|+|z-y|$

**Preuve** Il suffit d'appliquer l'inégalité triangulaire (montrée page 26) avec $x-z$ et $z-y$.
■

Il nous est maintenant possible de définir l'espace métrique pour $\mathbb{R}$ (R_met) ainsi que la limite d'une fonction à un argument (limit1_in).

```
Definition R_met:Metric_Space:=(Build_Metric_Space R R_dist
  R_dist_pos R_dist_sym R_dist_refl R_dist_tri).
```

où Build_Metric_Space est le constructeur du type record pour notre définition des espaces métriques.

```
Definition limit1_in:(R->R)->(R->Prop)->R->R->Prop:=
  [f:R->R; D:R->Prop; l:R; x0:R](limit_in R_met R_met f D x0 l).
```

### 2.1.2 Propriétés

Nous énoncerons quelques propriétés que nous utiliserons dans la section concernant la dérivée. Ce développement étant également disponible dans la version courante de Coq (V7.0), nous pourrons nous référer à la bibliothèque des réels pour l'intégralité des propriétés démontrées.

**Lemme 2.1.5 (tech_limit)** *Pour toute fonction $f$ à un argument et à valeur dans $\mathbb{R}$, et tout domaine de définition $D$, si $x_0 \in D$ et $\lim\limits_{x \to x_0} f(x)=l$ alors $l=f(x_0)$.*

**Preuve** Sous les hypothèses suivantes, montrons que $l=f(x_0)$ :
- H : $x_0 \in D$
- $\mathrm{H}_0$ : $\forall \varepsilon>0 \ \exists \alpha>0, \forall x \in D, |x-x_0|<\alpha \rightarrow |f(x)-l|<\varepsilon$

Nous avons que $|f(x_0)-l| \geq 0$ (propriété de la valeur absolue), ce qui signifie que soit $|f(x_0)-l|>0$ soit $|f(x_0)-l|=0$. Considérons les deux cas :
- Cas $|f(x_0)-l|>0$ : montrons que $|f(x_0)-l|<|f(x_0)-l|$, ce qui assure que ce cas est impossible (car $<$ est antireflexive). En appliquant l'hypothèse $\mathrm{H}_0$ avec $\varepsilon=|f(x_0)-l|$, ceci revient à prouver, en prenant $x=x_0$ et en utilisant H, que $|x_0-x_0|<\alpha_1$ pour un $\alpha_1$ arbitraire strictenebt positif, ce qui est direct car $|x_0-x_0|=0$ (par R_dist_refl).

– Cas $|f(x_0) - l| = 0$ : Trivial (par `R_dist_refl`).

■

Les preuves que nous donnons ici sont volontairement très proches des preuves faites en Coq, ce qui nous permet d'aborder les mathématiques sous un angle "méthodes formelles". Les deux approches sont différentes du fait que, dans notre cas, aucun détail n'est négligé, aussi trivial puisse-t-il paraître d'un point de vue mathématique. Nous souhaitons ainsi présenter les preuves d'une manière la plus "formelle" possible. A titre d'exemple, nous donnerons la preuve en Coq :

```
Lemma tech_limit:(f:R->R)(D:R->Prop)(l:R)(x0:R)(D x0)->
   (limit1_in f D l x0)->l==(f x0).
Proof.
(* introduction des hypotheses *)
Intros f D l x0 H H0.
(* separation de |f(x0)-l|>=0 *)
Case (Rabsolu_pos (Rminus (f x0) l)); Intros H1.
(* cas |f(x0)-l|>0 *)
(* mise en evidence de la contradiction *)
Absurd (Rlt (dist R_met (f x0) l) (dist R_met (f x0) l)).
Apply Rlt_antirefl.
(* utilisation de H0 *)
Case (H0 (dist R_met (f x0) l)); Auto.
Intros alpha1 (H2, H3); Apply H3; Split; Auto.
Case (dist_refl R_met x0 x0); Intros Hr1 Hr2; Rewrite Hr2; Auto.
(* cas |f(x0)-l|=0 *)
Case (dist_refl R_met (f x0) l); Intros Hr1 Hr2; Apply sym_eqT; Auto.
Save.
```

Nous montrerons également certaines propriétés, parmi d'autres formalisées en Coq, que nous considérons les plus utiles et/ou les plus intéressantes. Nous détaillerons, en particulier, les propriétés concernant l'addition, la multiplication, la composition et l'unicité de la limite.

**Lemme 2.1.6 (Addition des limites)** *Soit $D$ le domaine de définition des fonctions $f$ et $g$. Si $\lim_{x \to x_0} f(x) = l$ et $\lim_{x \to x_0} g(x) = l'$ alors $\lim_{x \to x_0} (f(x) + g(x)) = l + l'$*

**Preuve** Nommons $H_1$ la première hypothèse concernant la limite de $f$ et $H_2$ celle pour $g$. En utilisant $H_1$ et $H_2$ avec $\frac{\varepsilon}{2}$ nous obtenons l'existence d'un $\alpha_1$ et d'un $\alpha_2$ qui nous donne les deux nouvelles hypothèses $H_1$' et $H_2$' suivantes :
- $H_1$' : $\forall x \in D, |x - x_0| < \alpha_1 \to |f(x) - l| < \frac{\varepsilon}{2}$
- $H_2$' : $\forall y \in D, |y - x_0| < \alpha_2 \to |g(y) - l'| < \frac{\varepsilon}{2}$

Il nous faut maintenant trouver un $\alpha$ afin de montrer que
$\exists \alpha, \forall x_2 \in D, |x_2 - x_0| < \alpha \to |f(x_2) + g(x_2) - (l + l')| < \varepsilon$

Prenons $\alpha = min(\alpha_1, \alpha_2)$. Utilisons $H_1$' et $H_2$' avec $x_2$ ainsi que la propriété[3] si $a < min(b, c)$ alors $a < b$ et $a < c$, avec $a = |x_2 - x_0|$, $b = \alpha_1$ et $c = \alpha_2$.

Il nous reste à montrer que $|f(x_2) + g(x_2) - (l + l')| < \varepsilon$ sous les hypothèses suivantes :
- $H_1$" : $|f(x_2) - l| < \frac{\varepsilon}{2}$

[3] montrée lors du précédent développement des fonctions de base des réels

- H$_2$" : $|g(x_2) - l'| < \frac{\varepsilon}{2}$

En additionnant H$_1$" et H$_2$", on obtient $|f(x_2) - l| + |g(x_2) - l'| < \varepsilon$.

Soit la propriété $|(a + c) - (b + d)| \leq |a - b| + |c - d|$ montré lorsque nous avons défini la distance `R_dist`. Nous avons donc que $|(f(x_2) + g(x_2)) - (l + l')| \leq |f(x_2) - l| + |g(x_2) - l'|$, ce qui, par transitivité, achève la preuve. ∎

**Lemme 2.1.7 (Multiplication des limites)** *Soit $D$ le domaine de définition des fonctions $f$ et $g$. Si $\lim\limits_{x \to x_0} f(x) = l$ et $\lim\limits_{x \to x_0} g(x) = l'$ alors $\lim\limits_{x \to x_0} (f(x) \times g(x)) = l \times l'$*

**Preuve** Nommons H$_1$ la première hypothèse concernant la limite de $f$ et H$_2$ celle pour $g$. En utilisant H$_1$ avec $min(1, \frac{\varepsilon}{1+|l|+|l'|})$ et H$_2$ avec $\frac{\varepsilon}{1+|l|+|l'|}$ nous obtenons les deux nouvelles hypothèses H$_1$' et H$_2$' suivantes :
- H$_1$' : $\forall x \in D, |x - x_0| < \alpha_1 \to |f(x) - l| < min(1, \frac{\varepsilon}{1+|l|+|l'|})$
- H$_2$' : $\forall y \in D, |y - x_0| < \alpha_2 \to |g(y) - l'| < \frac{\varepsilon}{1+|l|+|l'|}$

Il nous faut maintenant montrer que
$\exists \alpha, \forall x_2 \in D, |x_2 - x_0| < \alpha \to |f(x_2) \times g(x_2) - l \times l'| < \varepsilon$

Prenons $\alpha = min(\alpha_1, \alpha_2)$. En utilisant le même raisonnement que pour la preuve précédente, il nous reste à montrer que $|f(x_2) \times g(x_2) - l \times l'| < \varepsilon$ sous les hypothèses suivantes :
- H$_1$" : $|f(x_2) - l| < min(1, \frac{\varepsilon}{1+|l|+|l'|})$
- H$_2$" : $|g(x_2) - l'| < \frac{\varepsilon}{1+|l|+|l'|}$

$|f(x_2) \times g(x_2) - l \times l'|$ peut s'écrire également sous la forme $|f(x_2) \times (g(x_2) - l') + l' \times (f(x_2) - l)|$. Montrons alors que $|f(x_2) \times (g(x_2) - l')| + |l' \times (f(x_2) - l)| < \varepsilon$, ce qui achève la preuve par transitivité et inégalité triangulaire.

Utilisons la propriété sur la valeur absolue $|a.b| = |a|.|b|$ puis montrons les deux inégalités suivantes. D'une part, prouvons que $|l'|\ |(f(x_2) - l)| \leq |l'| \frac{\varepsilon}{1+|l|+|l'|}$. Trivial en utilisant la propriété sur le $min$ avec H$_1$" puis en multipliant chaque membre de la nouvelle inégalité $|f(x_2) - l| < \frac{\varepsilon}{1+|l|+|l'|}$ par |l'| (qui est $\geq 0$).

D'autre part, prouvons que $|f(x_2)|\ |(g(x_2) - l')| < (1 + |l|) \frac{\varepsilon}{1+|l|+|l'|}$. Pour cela utilisons la propriété suivante *si* $r_3 \geq 0$ , $r_2 > 0$ , $r_1 < r_2$ *et* $r_3 < r_4$ *alors* $r_1 \times r_3 < r_2 \times r_4$ avec $r_3 = |g(x_2) - l'|$, $r_2 = 1 + |l|$, $r_1 = |f(x_2)|$ et $r_4 = \frac{\varepsilon}{1+|l|+|l'|}$. Or nous avons que :
- $|g(x_2) - l'| \geq 0$
- $1 + |l| > 0$ car $0 \leq |l| \to 0 < |l| + 1$
- $|f(x_2)| < 1 + |l|$ car $|f(x_2)| - |l| \leq |f(x_2) - l|$ et $|f(x_2) - l| < 1$ par H$_1$" d'où $|f(x_2)| - |l| < 1$
- $|g(x_2) - l'| < \frac{\varepsilon}{1+|l|+|l'|}$ par H$_2$"

En additionnant les deux inégalités montrées précédemment, on obtient bien
$|f(x_2) \times (g(x_2) - l')| + |l' \times (f(x_2) - l)| < |l'| \frac{\varepsilon}{1+|l|+|l'|} + (1 + |l|) \frac{\varepsilon}{1+|l|+|l'|}$ ($= \varepsilon$ car $1 + |l| + |l'| \neq 0$).
∎

**Lemme 2.1.8 (Composition des limites)** *Soit $D_f$ le domaine de définition de la fonction $f$ et $D_g$ celui de la fonction $g$. Si $\lim\limits_{x \to x_0} f(x) = l$ sur $D_f$ et $\lim\limits_{x \to l} g(x) = l'$ sur $D_g$ alors $\lim\limits_{x \to x_0} g(f(x)) = l'$ sur $D_{g \circ f}$*

Avant de donner la preuve, commençons par donner la définition de $D_{g \circ f}$. Ce domaine est représenté par un prédicat qui dépend des domaines $D_f$ et $D_g$ et d'une fonction $f$. Ainsi, $x \in D_{g \circ f}$, qui signifie $x \in D_f$ et $f(x) \in D_g$, s'écrit en Coq :

```
Definition Dgf:=[Df,Dg:R->Prop][f:R->R][x:R](Df x)/\(Dg (f x)).
```

Nous utiliserons cette définition lors de la preuve.

**Preuve** Nommons $H_1$ la première hypothèse concernant la limite de $f$ et $H_2$ celle pour $g$. En utilisant $H_2$ avec $\varepsilon$ puis $H_1$ avec le témoin existentiel $x$ issu de $H_2$, nous obtenons les deux nouvelle hypothèses $H_1$' et $H_2$' suivantes :
- $H_1$' : $\forall y \in D_g,\ |y - l| < x \to |g(y) - l'| < \varepsilon$
- $H_2$' : $\forall z \in D_f,\ |z - x_0| < x_1 \to |f(z) - l| < x$

Il nous faut maintenant montrer que
$\exists \alpha, \forall x_2 \in D_{g \circ f},\ |x_2 - x_0| < \alpha \to |g(f(x_2)) - l'| < \varepsilon$

Prenons $\alpha = x_1$. Montrons maintenant que $x_2 \in D_f$. Nous savons que $x_2 \in D_{g \circ f}$ donc, par définition que $x_2 \in D_f$ et que $f(x_2) \in D_g$. Nous pouvons alors utiliser $H_2$' avec $x_2$ ce qui montre que $|f(x_2) - l| < x$.

Il nous reste donc à montrer que $|g(f(x_2)) - l'| < \varepsilon$ sous les hypothèses $H_1$', $f(x_2) \in D_g$ et $|f(x_2) - l| < x$. Le résultat nous est donné directement en utilisant $H_1$' avec $f(x_2)$. ■

Les trois lemmes dont nous venons de donner les preuves seront utilisées lors de développement concernant la dérivée. Parmi les propriétés faisant partie de cette bibliothèque, nous nous attarderons sur celle concernant l'unicité de la limite. Elle fait intervenir la notion d'adhérence, que nous n'avons pas encore utilisée mais dont nous ne pouvons nous abstenir pour ce lemme d'unicité.

Nous noterons $\bar{D}$ l'adhérence de $D$, que nous définirons en Coq en énonçant que $a$ est adhérent à $D$. Nous n'utilisons pas la définition 2.1.3, mais plutôt la propriété que $a$ est un point adhérent à $D$ si $\forall \alpha > 0, \exists x \in D, |x - a| < \alpha$ :

```
Definition adhDa:(R->Prop)->R->Prop:=[D:R->Prop][a:R]
  (alp:R)(Rgt alp R0)->(EXT x:R | (D x)/\(Rlt (R_dist x a) alp)).
```

**Lemme 2.1.9 (Unicité de la limite)** *Si $x_0 \in \bar{D}$, il existe un unique élément $l$ de $\mathbb{R}$ tel que $\lim\limits_{x \to x_0} f(x) = l$.*

Ce qui s'énonce en Coq de la manière suivante :

```
Lemma single_limit:(f:R->R)(D:R->Prop)(l:R)(l':R)(x0:R)
  (adhDa D x0)->(limit1_in f D l x0)->(limit1_in f D l' x0)->l==l'.
```

**Preuve** Commençons par montrer que $|l - l'| < 2\varepsilon$ sous les hypothèses du lemme (utilisées avec $\varepsilon$ pour les deux limites) :
- H : $x_0 \in \bar{D}$ c'est-à-dire $\forall \alpha > 0,\ \exists t \in D\ |t - x_0| < \alpha$ (adhérence)
- $H_0$ : $\forall y \in D,\ |y - x_0| < \alpha_1 \to |f(y) - l'| < \varepsilon$ ($l$ est limite de $f$)
- $H_1$ : $\forall z \in D,\ |z - x_0| < \alpha_2 \to |f(z) - l| < \varepsilon$ ($l'$ est limite de $f$)

Utilisons l'hypothèse H avec $min(\alpha_1, \alpha_2)$, ce qui nous donne (après un travail similaire à celui fait lors des preuves précédentes) les deux nouvelles hypothèses :
- $H_0$' :$|f(x_2) - l'| < \varepsilon$
- $H_1$' :$|f(x_2) - l| < \varepsilon$

En additionnant $H_0$' et $H_1$' et le fait que $|x-y| = |y-x|$, on a $|l-f(x_2)|+|f(x_2)-l'| < 2\varepsilon$, ce qui nous donne $|l - l'| < 2\varepsilon$ par transitivité et inégalité triangulaire.

Utilisons cette nouvelle hypothèse afin de montrer que $l = l'$. Pour cela, nous utiliserons la proposition suivante, montrée précédemment (en utilisant l'ordre total suivant que $r > 0$ ou $r \leq 0$) dans le développement : $\forall r(\forall \varepsilon > 0 \rightarrow r < \varepsilon) \rightarrow r \leq 0$.

Après avoir également montré que $(\forall \varepsilon > 0\ |l - l'| < 2\varepsilon) \rightarrow (\forall \varepsilon > 0\ |l - l'| < \varepsilon)$, nous pouvons en déduire que $|l - l'| \leq 0$, ce qui nous permet d'en déduire le résultat. ■

Les preuves concernant, en particulier, le moins unaire et binaire, ainsi que les constantes ont été faites de façon similaire.

## 2.2 Dérivée et Continuité

A partir de la définition de la limite que nous avons utilisée, nous verrons comment formaliser la notion de dérivée. Nous comparerons, en particulier, deux définitions possibles.

Commençons par donner une définition mathématique de la dérivée. Cette définition fait suite à nos définitions 2.1.4 et 2.1.5 de la limite [57].

**Définition 2.2.1 (Dérivée)** *Soit $I$ un voisinage du point $t_0$ dans $\mathbb{R}$, et $f$ une application de $I$ dans un espace métrique $E$. On dit que $f$ est dérivable au point $t_0$ si l'application*

$$I \backslash \{t_0\} \rightarrow E, \qquad t \mapsto \frac{f(t) - f(t_0)}{t - t_0}$$

*a une limite au point $t_0$. Cette limite est appelée dérivée de $f$ au point $t_0$.*

Nous pensons que cette définition est la plus appropriée dans notre cas et nous allons tenter d'expliquer pourquoi. Pour cela, nous commencerons par donner une autre définition [11] de la dérivée équivalente à la précédente.

**Définition 2.2.2 (Dérivée)** *On dit que $f$ est dérivable au point $x$ si l'application*

$$h \mapsto \frac{f(x+h) - f(x)}{h}$$

*a une limite en 0. Cette limite est appelée dérivée de $f$ au point $x$.*

Cette dernière définition est celle utilisée, par exemple, dans la formalisation de l'analyse dans HOL ([42]) par John Harrison [47] et dans PVS ([79], [77], [78]) par Bruno Dutertre [29]. Néanmoins, dans le cas de Coq cette définition est plus difficile à utiliser que la définition 2.2.1. En effet, de par notre définition de la limite, qui fait intervenir la notion de domaine $D$, la définition 2.2.2 fait intervenir des propriétés de translation puisque, en particulier, cette définition comprend un terme de la forme $f(x + h)$. C'est pour cette raison que nous avons préféré la première définition, que nous énonçons en Coq comme suit. Rappelons, en utilisant une syntaxe mathématique, le définition que nous souhaitons utiliser :

$$d(x_0) = \lim_{x \to x_0, x \in D \backslash x_0} \frac{f(x) - f(x_0)}{x - x_0}$$

où $D \backslash x_0$ est le domaine de définition de l'application $x \mapsto \frac{f(x)-f(x_0)}{x-x_0}$ et $d(x_0)$ la dérivée en $x_0$.

Pour écrire cette définition en Coq nous commencerons par définir un domaine $D \backslash x$ :

```
Definition D_x:(R->Prop)->R->R->Prop:=[D:R->Prop][y:R][x:R](D x)/\``y<>x``.
```

D_x est un prédicat qui dépend de $D$ et de $x$. La définition Coq précédente traduit le fait que $y \in D \backslash x$ signifie que $y \in D$ et $y \neq x$.

Puis, similairement à la limite, nous énonçons que $d$ est la dérivée de $f$ en $x_0$ sur $D$, en utilisant la définition de la limite donnée précédemment. En Coq nous avons donc :

```
Definition D_in:(R->R)->(R->R)->(R->Prop)->R->Prop:=
  [f:R->R; d:R->R; D:R->Prop; x0:R](limit1_in
    [x:R] ``(f x)-(f x0)/(x-x0)``
    (D_x D x0) (d x0) x0).
```

Afin de montrer certaines propriétés sur la dérivée, nous allons introduire la notion de continuité.

**Définition 2.2.3 (Continuité)** *On dit que $f$ est continue en un point $a$ sur un domaine $D$ si $f(x)$ tend vers $f(a)$ lorsque $x$ tend vers $a$.*

ou encore en Coq :

```
Definition continue_in:(R->R)->(R->Prop)->R->Prop:=
       [f:R->R; D:R->Prop; a:R](limit1_in f (D_x D a) (f a) a).
```

Nous pouvons maintenant montrer une propriété que nous utiliserons à plusieurs reprises. Comme nous l'avons fait pour les preuves précédentes, nous nous efforcerons d'en donner une preuve qui sera la plus fidèle possible à celle effectuée formellement dans Coq, ce qui nous permet également de bien isoler l'utilisation de la propriété d'ordre total, seul aspect classique de notre formalisation.

**Lemme 2.2.1 (Dérivabilité-Continuité)** *Si $f$ est une fonction dérivable en $x_0$ alors $f$ est continue en $x_0$.*

**Preuve** Commençons par traduire l'hypothèse que $f$ admet une dérivée $d(x_0)$ en $x_0$ et nommons cette hypothèse $\mathrm{H}_1$ :
$\forall \varepsilon > 0, \exists \alpha > 0$ t.q. $\forall x \in D \backslash x_0, |x - x_0| < \alpha \to |\frac{f(x)-f(x_0)}{x-x_0} - d(x_0)| < \varepsilon$

Il nous faut montrer que :
$\forall \varepsilon > 0, \exists \alpha > 0$ t.q. $\forall x \in D \backslash x_0, |x - x_0| < \alpha \to |f(x) - f(x_0)| < \varepsilon$

Nous distinguons deux cas selon que $d(x_0) = 0$ ou pas.

– Cas $d(x_0) = 0$ :
En utilisant l'hypothèse $\mathrm{H}_1$ avec $\varepsilon$, en remplaçant $d(x_0)$ par 0 et en nommant $\alpha_1$ le témoin existentiel issu de $\mathrm{H}_1$, cela revient à montrer que
$\exists \alpha > 0$ t.q. $\forall x \in D \backslash x_0, |x - x_0| < \alpha \rightarrow |f(x) - f(x_0)| < \varepsilon$ sous l'hypothèse
$\forall x \in D \backslash x_0, |x - x_0| < \alpha_1 \rightarrow |\frac{f(x)-f(x_0)}{x-x_0}| < \varepsilon$

Choisissons $\alpha = min(1, \alpha_1)$. Nous pouvons donc en déduire (propriété du $min$) que $|x - x_0| < 1$ et $|x - x_0| < \alpha_1$. Nous nous ramenons donc à montrer, en instanciant notre nouvelle hypothèse par $x$ et $|x - x_0| < \alpha_1$, que $|\frac{f(x)-f(x_0)}{x-x_0}| < \varepsilon$ implique que $|f(x) - f(x_0)| < \varepsilon$.

En utilisant quelques propriétés élémentaires concernant la valeur absolue, nous pouvons en déduire que $|f(x) - f(x_0)| < |x - x_0|\varepsilon$ car $|x - x_0| \neq 0$ puisque $x \in D \backslash x_0$ (voir la définition de `D_x` donnée précédemment).

Or, nous savons que $|x - x_0| < 1$, donc que $|x - x_0|\varepsilon < \varepsilon$.

– Cas $d(x_0) \neq 0$ :
Procédons de le même façon. Choisissons $\alpha = min(\frac{1}{2}, \alpha_1, \frac{\varepsilon}{|2d(x_0)|})$. Nous pouvons donc en déduire que $|x - x_0| < \frac{1}{2}$ (nommons-la $\mathrm{I}_1$), $|x - x_0| < \alpha_1$ et $|x - x_0| < \frac{\varepsilon}{|2d(x_0)|}$ (nommons-la $\mathrm{I}_2$). Nous savons également, par hypothèse, que $d(x_0) \neq 0$ (nommons-la $\mathrm{E}_1$) et que $|x - x_0| \neq 0$ (nommons-la $\mathrm{E}_2$).

Il nous reste à monter que $|\frac{f(x)-f(x_0)}{x-x_0} - d(x_0)| < \varepsilon$ implique que $|f(x) - f(x_0)| < \varepsilon$.

Notre nouvelle hypothèse peut se transformer de la sorte :
$|f(x) - f(x_0) - (x - x_0)d(x_0)| < \varepsilon|x - x_0|$ puis en utilisant une propriété d'inégalité triangulaire $|f(x) - f(x_0)| - |x - x_0|\ |d(x_0)| < \varepsilon|x - x_0|$.

Il avons donc que $|f(x) - f(x_0)| < |x - x_0|\ |d(x_0)| + \varepsilon|x - x_0|$. Il nous reste à montrer que $|x - x_0|\ |d(x_0)| + \varepsilon|x - x_0| < \varepsilon$, ce qui est quasi immédiat par $\mathrm{I}_1$, $\mathrm{I}_2$, $\mathrm{E}_1$ et $\mathrm{E}_2$.

■

A l'aide des définitions précédentes, il nous est possible de montrer les principales propriétés attendues. Nous omettrons d'indiquer le domaine de définition lorsque la propriété en est indépendante. Nous pouvons citer, par exemple et entre autres :

1. 0 est la dérivée de la fonction constante en $x_0$.
2. 1 est la dérivée de la fonction $x \mapsto x$ en $x_0$.
3. si $f$ admet une dérivée $df(x_0)$ et si $g$ admet une dérivée $dg(x_0)$ alors $df(x_0) + dg(x_0)$ est la dérivée de $f + g$ en $x_0$.
4. si $f$ admet une dérivée $df(x_0)$ et si $g$ admet une dérivée $dg(x_0)$ alors $df(x_0)g(x_0) + dg(x_0)f(x_0)$ est la dérivée de $f \times g$ en $x_0$.
5. $n.x_0^{n-1}$ est la dérivée de la fonction $x \mapsto x^n$ en $x_0$.
6. soient $D_f$ et $D_g$ les domaines de définition respectifs des fonctions $f$ et $g$. Si $f$ admet une dérivée $df(x_0)$ et si $g$ admet une dérivée $dg(f(x_0))$ alors $df(x_0)dg(f(x_0))$ de $g \circ f$ est la dérivée en $x_0$ sur $D_{g \circ f}$.
7. si $f$ admet une dérivée $df(x_0)$ alors $n.f^{n-1}(x_0).df(x_0)$ est la dérivée de la fonction $f^n$.

La dérivée étant, en fait, une limite, la preuve de ces propriétés se fait en utilisant les propriétés similaires que nous avons montré sur la limite. Néanmoins, nous donnerons tout de même ici la preuve de la propriété de multiplication (4) et de composition (6) car elles ont une spécificité : elles utilisent la propriété de continuité.

**Preuve** [Propriété 4] Rammenons-nous à un énoncé faisant apparaître les limites. Nous nous plaçons sur un domaine $D$ de définition. Nous voulons alors montrer que

$$\frac{f(x)g(x) - f(x_0)g(x_0)}{x - x_0} \xrightarrow[x \to x_0]{} df(x_0)g(x_0) + dg(x_0)f(x_0)$$

sous les hypothèses suivantes : $\frac{f(x)-f(x_0)}{x-x_0} \xrightarrow[x \to x_0]{} df(x_0)$ et $\frac{g(x)-g(x_0)}{x-x_0} \xrightarrow[x \to x_0]{} dg(x_0)$

D'autre part, par le lemme de dérivabilité-continuité 2.2.1 nous avons que $f(x) \xrightarrow[x \to x_0]{} f(x_0)$.

En utilisant le lemme de multiplication des limites 2.1.7, nous pouvons en déduire que : $\frac{g(x)-g(x_0)}{x-x_0} f(x) \xrightarrow[x \to x_0]{} dg(x_0)f(x_0)$ ainsi que
$\frac{f(x)-f(x_0)}{x-x_0} g(x_0) \xrightarrow[x \to x_0]{} df(x_0)g(x_0)$ puisque $g(x_0) \xrightarrow[x \to x_0]{} g(x_0)$.

Puis, de par le lemme d'addition des limites 2.1.6 nous obtenons que :
$\frac{f(x)-f(x_0)}{x-x_0} g(x_0) + \frac{g(x)-g(x_0)}{x-x_0} f(x) \xrightarrow[x \to x_0]{} df(x_0)g(x_0) + dg(x_0)f(x_0)$

Or, nous avons effectivement que $\frac{f(x)-f(x_0)}{x-x_0} g(x_0) + \frac{g(x)-g(x_0)}{x-x_0} f(x) = \frac{f(x)g(x)-f(x_0)g(x_0)}{x-x_0}$
■

Nous avons également choisi de présenter la preuve suivante car elle illustre très clairement l'importance des domaines de définitions pour ces propriétés.

**Preuve** [Propriété 6] Rammenons-nous à un énoncé faisant apparaître les limites. Nous voulons montrer que

$$\frac{g(f(x)) - g(f(x_0))}{x - x_0} \xrightarrow[x \to x_0, x \in D_{g \circ f} \backslash x_0]{} df(x_0)dg(f(x_0))$$

sous les hypothèse suivantes :

H$_1$ : $\frac{f(x)-f(x_0)}{x-x_0} \xrightarrow[x \to x_0, x \in D_f \backslash x_0]{} df(x_0)$ et H$_2$ : $\frac{g(x)-g(f(x_0))}{x-f(x_0)} \xrightarrow[x \to f(x_0), x \in D_g \backslash f(x_0)]{} dg(f(x_0))$

En utilisant le lemme de dérivabilité-continuité 2.2.1 nous avons que
H$_3$ : $f(x) \xrightarrow[x \to x_0, x \in D_f \backslash x_0]{} f(x_0)$

De par le lemme de composition des limites 2.1.8 appliqué avec H$_3$ et H$_2$, nous pouvons en déduire que
H$_4$ : $\frac{g(f(x))-g(f(x_0))}{f(x)-f(x_0)} \xrightarrow[x \to x_0, x \in D_f \backslash x_0 \cap D_g \backslash f(x_0)]{} dg(f(x_0))$

Nous allons maintenant montrer que
H$_5$ : $\frac{f(x)-f(x_0)}{x-x_0} \xrightarrow[x \to x_0, x \in D_f \backslash x_0 \cap D_g \backslash f(x_0)]{} df(x_0)$

Pour cela il suffit de montrer que si $x \in D_f \backslash x_0 \cap D_g \backslash f(x_0)$ alors $x \in D_f \backslash x_0$ ce qui est immédiat, puis d'utiliser $H_1$.

En utilisant le lemme de multiplication des limites 2.1.7 avec $H_4$ et $H_5$, on a également que
$H_6$ : $\frac{g(f(x))-g(f(x_0))}{f(x)-f(x_0)}.\frac{f(x)-f(x_0)}{x-x_0} \xrightarrow[x \to x_0, x \in D_f \backslash x_0 \cap D_g \backslash f(x_0)]{} dg(f(x_0))df(x_0)$

À partir de l'hypothèse $H_6$, nous montrerons notre but initial en deux étapes :

1. montrons que si $x \in D_f \backslash x_0 \cap D_g \backslash f(x_0)$ alors $x \in D_{g \circ f} \backslash x_0$ : trivial en utilisant les définition de $D_{g \circ f}$ et $D \backslash x$, ce qui nous donne :
   $x \in D_{g \circ f} \backslash x_0 \equiv x \in D_f, f(x) \in D_g, x \neq x_0$.
2. montrons maintenant que $\frac{g(f(x))-g(f(x_0))}{f(x)-f(x_0)}.\frac{f(x)-f(x_0)}{x-x_0} \xrightarrow[x \to x_0, x \in D_{g \circ f} \backslash x_0]{} dg(f(x_0))df(x_0)$ implique $\frac{g(f(x))-g(f(x_0))}{x-x_0} \xrightarrow[x \to x_0, x \in D_{g \circ f} \backslash x_0]{} df(x_0)dg(f(x_0))$

   Pour cela nous devons distinguer deux cas suivant que $f(x) = f(x_0)$ ou pas :
   - Cas $f(x) = f(x_0)$ : De part notre définition totale de la fonction inverse, la preuve est immédiate en réécrivant $f(x) - f(x_0)$ par 0.
   - Cas $f(x) \neq f(x_0)$ : nous pouvons, dans ce cas, simplifier pas $f(x) - f(x_0)$ dans notre nouvelle hypothèse, ce qui achève la preuve.

■

## 2.3 Les fonctions somme et somme infinie

Nous présentons ici les formalisations nécessaires au développement concernant les suites et séries.

### 2.3.1 Fonctions somme

Nous donnons ici les définitions des fonctions sommes suivantes : $\sum_{i=0}^{n} f(i)$ et $\sum_{i=s}^{n} f(i)$, où $f$ est une fonction de type $\mathbb{N} \to \mathbb{R}$. Comme pour la fonction puissance, nous utiliserons un `fixpoint` :

```
Fixpoint sum_f_R0 [f:nat->R;n:nat]:R:=
  Cases n of
     O     => (f O)
    |(S i) => ``(sum_f_R0 f i)+(f (S i))``
  end.
```

Pour définir $\sum_{i=s}^{n} f(i)$ nous utiliserons le fait que cette somme est égale à $\sum_{i=0}^{n-s} f(i+s)$ :

```
Definition sum_f [s,n:nat;f:nat->R]:R:=
  (sum_f_R0 [x:nat](f (plus x s)) (minus n s)).
```

Comme pour la fonction puissance, les preuves se font par récurrence sur $n$. Citons, par exemple, les deux propriétés suivantes :

1. $(x-1)\sum_{i=0}^{n} x^i = x^{n+1} - 1$

2. $|\sum_{i=0}^{n} f(i)| \leq \sum_{i=0}^{n} |f(i)|$

La première propriété ayant son équivalent sur les séries entières (somme infinie) nous en donnons une preuve, ce qui nous permettra de comparer les deux :

**Preuve** Par récurrence sur n :

1. $n = 0$ : d'une part nous avons $(x-1)\sum_{i=0}^{0} x^i = (x-1)x^0 = x-1$ et d'autre part $x^{0+1} - 1 = x - 1$.
2. Supposons la propriété vraie pour $n$ et montrons-la pour $n+1$. Nous devons donc montrer que $(x-1)\sum_{i=0}^{n+1} x^i = x^{n+2} - 1$ sachant que $(x-1)\sum_{i=0}^{n} x^i = x^{n+1} - 1$ (hypothèse de récurrence). Or $\sum_{i=0}^{n+1} x^i = x^{n+1} + \sum_{i=0}^{n} x^i$. Ce qui nous donne, en utilisant l'hypothèse de récurrence, $(x-1)\sum_{i=0}^{n+1} x^i = (x-1)(x^{n+1} + \sum_{i=0}^{n} x^i) = (x-1)x^{n+1} + (x^{n+1} - 1)$, qui est bien égal à $x^{n+2} - 1$.

■

### 2.3.2 Fonction somme infinie

Nous allons définir la notion de somme infinie : $\sum_{i=0}^{+\infty} f(i)$. Pour ce, nous utiliserons une définition similaire à celle que nous avons utilisée pour la limite car $\sum_{i=0}^{+\infty} f(i) = \lim_{n \to +\infty} \sum_{i=0}^{n} f(i)$.

**Définition 2.3.1 (somme infinie)** *Soit une fonction $f$ à valeur des entiers dans les réels. Pour que $\sum_{i=0}^{n} f(i)$ tende vers $l$ lorsque $n$ tend vers $+\infty$, il faut et il suffit qu'à tout nombre $\varepsilon > 0$ on puisse associer un entier $N$ tel que $\forall n \geq N$, $|\sum_{i=0}^{n} f(i) - l| < \varepsilon$. La somme infinie $\sum_{i=0}^{+\infty} f(i)$ est représentée par $l$.*

En Coq nous avons :

```
Definition infinit_sum:(nat->R)->R->Prop:=[f:nat->R;l:R]
  (eps:R)``eps >= 0``->
  (Ex[N:nat](n:nat)(ge n N)->``(R_dist (sum_f_R0 f n) l) < eps``).
```

## 2.4 Les Suites et Séries

Ce développement a surtout été fait en vue de définir les fonctions transcendantes de la section suivante. Il comprend une formalisation des suites de Cauchy et des séries entières [15].

### 2.4.1 Les suites de Cauchy

En ce qui concerne les suites et séries, ce sont les critères de convergence qui nous intéressent particulièrement, car ils vont nous permettre de montrer l'existence de limites qui seront utilisées pour définir les fonctions transcendantes. Nous allons définir, par exemple, la notion de suite convergente, suivi du critère de Cauchy et d'autres propriétés s'y rapportant.

**Définition 2.4.1 ($U_n$ converge)** *$l$ étant un nombre réel, on dit que la suite ($U_n$) a pour limite $l$ si, pour tout nombre $\varepsilon > 0$, il existe un entier $N \geq 0$ tel que, pour tout $n \geq N$, on ait $|U_n - l| < \varepsilon$. On dit alors que la suite ($U_n$) est convergente.*

La définition en Coq suit exactement cette définition. La suite $U_n$ a été déclarée en paramètre, en utilisant le principe de sections (voir annexe D) de Coq. Une suite est vue comme une fonction de type `nat->R`. $U_0$ s'écrit alors `(U O)`.

```
Definition Un_cv:R->Prop:=[l:R] (eps:R)``eps > 0``->
  (Ex[N:nat](n:nat)(ge n N)->``(R_dist (Un n) l) < eps``).
```

Nous introduisons la notion de suite de Cauchy, qui nous aidera, par la suite, à montrer des propriétés de convergence.

**Définition 2.4.2 (Critère de Cauchy)** *On dit que la suite ($U_n$) a une limite dans $\mathbb{R}$ si, pour tout nombre $\varepsilon > 0$, il existe un entier $N \geq 0$ tel que, pour tout $n, m \geq N$, on ait $|U_n - U_m| < \varepsilon$. On dit alors que la suite ($U_n$) est de Cauchy.*

Avec son équivalence en Coq :

```
Definition Cauchy_crit:Prop:=(eps:R)``eps > 0``->
 (Ex[N:nat] (n,m:nat)(ge n N)->(ge m N)->``(R_dist (Un n) (Un m)) < eps``).
```

Nous pouvons maintenant montrer, par exemple, le lemme suivant :

**Lemme 2.4.1** *Toute suite croissante et majorée converge.*

Afin d'énoncer ce lemme en Coq, nous commencerons par formaliser les notions de *suite croissante* et d'*ensemble des* `Un`. Une suite $U_n$ est croissante si pout tout $n$, $U_n \leq U_{n+1}$, ce qui s'écrit en Coq :

```
Definition Un_growing:Prop:=(n:nat)(Rle (Un n) (Un (S n))).
```

L'ensemble `EUn` des `Un` sera représenté à l'aide du prédicat suivant :

```
Definition EUn:R->Prop:=[r:R](Ex [i:nat] (r==(Un i))).
```

Ce qui nous permet d'écrire le lemme précédent en Coq :

```
Lemma Un_cv_crit:Un_growing->(bound EUn)->(ExT [l:R] (Un_cv l)).
```

Nous verrons lors de la preuve qu'une telle suite a pour limite la borne supérieure de l'ensemble des $U_n$ :

**Preuve** Puisque $EU_n$ est majoré (par hypothèse) et non vide[4], nous pouvons utiliser la propriété de complétude. Nous devons alors montrer que $U_n$ converge, sous les hypothèses suivantes :

- H : $U_n$ croissante : $\forall n,\ U_n \leq U_{n+1}$
- $H_0$ : $EU_n$ majoré : $\exists m,\ \forall x \in EU_n\ x \leq m$.

[4] cette propriété ce montre trivialement car l'ensemble contient au moins un $U_n$

- $H_1$ : complétude : en appliquant l'axiome de complétude `complet` donné au chapitre 1 à l'ensemble $EU_n$, nous avons que $EU_n$ admet une borne supérieure, que nous noterons $x$. Montrer que $U_n$ converge équivaut à montrer que : $\exists l,\ \forall \varepsilon > 0\ \exists N, \forall n \geq N\ |U_n - l| < \varepsilon$
Choisissons $l = x$. Nous devons alors montrer que $\exists N, \forall n \geq N\ |U_n - x| < \varepsilon$ avec $\varepsilon > 0$.
Montrons avant tout deux propriétés que nous utiliserons dans la suite de cette preuve :
- $H_2$ : $\forall n\ U_n \leq x$. En utilisant la propriété de complétude cette propriété est immédiate, puisque $\forall n\ U_n \in EU_n$.
- $H_3$ : $\exists N,\ x - \varepsilon < U_N$. Utilisons ici le lemme classique suivant :
$\neg(\forall N,\ x - \varepsilon \geq U_N) \to \exists N,\ x - \varepsilon < U_N$. Nous devons alors montrer que $\neg(\forall N,\ x - \varepsilon \geq U_N)$, ce qui revient à montrer que si $\forall N,\ x - \varepsilon \geq U_N$ alors $\perp$. En utilisant l'axiome de complétude, nous obtenons que $x \leq x - \varepsilon$ où $\varepsilon > 0$, ce qui implique bien $\perp$.
Revenons à notre preuve. Utilisons $H_3$ et nommons $x_1$ le témoin. Nous pouvons alors choisir $N = x_1$ pour $\exists N, \forall n \geq N\ |U_n - x| < \varepsilon$. Il nous faut alors montrer que $|U_n - x| < \varepsilon$ sachant que $n \geq x_1$. En supprimant la valeur absolue, nous obtenons les deux cas suivant à prouver :

1. $U_n - x < \varepsilon$ : quasi immédiat en utilisant $H_2$ et par transitivité puisque $0 < \varepsilon$.
2. $-\varepsilon < U_n - x$ : puisque $n \geq x_1$ et que $U_n$ est croissante, on a que $U_n \geq U_{x_1}$. D'autre part, par $H_3$ nous avons que $x - \varepsilon < U_{x_1}$, ce qui nous donne le résultat par transitivité.

■

### 2.4.2 Les séries entières

Rappelons la définition d'une série entière : $\displaystyle\sum_{n=0}^{+\infty} A_n x^n$

Pour définir cette notion en `Coq`, nous procédons de la même manière que pour les suites de Cauchy. Les coefficients $A_n$ sont déclarés dans une section comme suit : `An :nat->R`. Puis nous utilisons la fonction `infinit_sum` définie précédemment dans ce chapitre :

```
Definition Pser:R->R->Prop:=[x,l:R]
  (infinit_sum [n:nat]``(An n)*(pow x n)`` l).
```

Il nous est maintenant possible de montrer la propriété similaire à celle que nous avons montrée pour les sommes finies : $|x| < 1 \to \displaystyle\sum_{n=0}^{+\infty} x^n = \frac{1}{1-x}$

**Preuve** En utilisant la définition mathématique `Pser`, cela revient à montrer que :
$|x| < 1 \to \forall \varepsilon > 0\ \exists N, \forall n \geq N\ |\displaystyle\sum_{i=0}^{n} x^i - \frac{1}{1-x}| < \varepsilon$. Dans la suite de la preuve, nous nommerons H l'hypothèse $|x| < 1$. Nous étudierons deux cas (en utilisant l'axiome classique sur l'égalité) :

– $x = 0$ : montrons que $\exists N, \forall n \geq N\ |\displaystyle\sum_{i=0}^{n} 0^i - 1| < \varepsilon$ où $\varepsilon > 0$.

Choisissons $N = 0$. Il nous reste à prouver que $|\displaystyle\sum_{i=0}^{n} 0^i - 1| < \varepsilon,\ \forall n \geq 0$.

Or $\displaystyle\sum_{i=0}^{n} 0^i = 1$ car : par récurrence sur $n$ on a :

– $n = 0$ : $\displaystyle\sum_{i=0}^{0} 0^i = 0^0 = 1$

– on suppose que $\sum_{i=0}^{n} 0^i = 1$ et on montre que $\sum_{i=0}^{n+1} 0^i = 1$.
Or $\sum_{i=0}^{n+1} 0^i = \sum_{i=0}^{n} 0^i + 0^{n+1} = 1 + 0 = 1$ en utilisant l'hypothèse de récurrence.

De ce fait, nous devons donc montrer que $|1 - 1| < \varepsilon$, ce qui équivaut à l'hypothèse $\varepsilon > 0$.

– $x \neq 0$ : montrons que $\exists N, \forall n \geq N \ |\sum_{i=0}^{n} x^i - \frac{1}{1-x}| < \varepsilon$ où $\varepsilon > 0$.
Pour ce, nous utiliserons le lemme suivant, que nous avons montré précédemment dans le développement en Coq, mais dont nous ne donnerons pas de preuve ici :
`pow_lt_1_zero` : $\forall x, |x| < 1 \rightarrow \forall e > 0, \exists N \ \forall n \geq N \ |x^n| < e$.

En utilisant `pow_lt_1_zero` avec $x$, $|x| < 1$ (hypothèse H) et $\varepsilon\frac{|1-x|}{|x|}$ nous avons la nouvelle hypothèse suivante : $\exists N \ \forall n \geq N \ |x^n| < \varepsilon\frac{|1-x|}{|x|}$ nommons-la $\mathrm{H}_1$.
Remarquons que pour instancier $e$ par $\varepsilon\frac{|1-x|}{|x|}$ il faut montrer que $\varepsilon\frac{|1-x|}{|x|} > 0$, ce qui est relativement simple car $x \neq 1$ par H. Nommons $N_1$ le témoin issue de $\mathrm{H}_1$. Dans la propriété d'origine, nous pouvons alors choisir $N = N_1$.
Il nous faut alors montrer que $|\sum_{i=0}^{n} x^i - \frac{1}{1-x}| < \varepsilon, \forall n \geq N_1$. Nous avons déjà montré que $|1 - x| > 0$. En multipliant les deux membres de l'inégalité, nous avons
$|1-x|.|\sum_{i=0}^{n} x^i - \frac{1}{1-x}| < |1-x|.\varepsilon$, qui équivaut, après simplifications, à
$|(1-x)\sum_{i=0}^{n} x^i - 1| < |1-x|.\varepsilon$.
Or, en utilisant la première propriété sur les sommes finies (page 44), cela revient à montrer que : $|-x^{n+1}| < |1-x|\varepsilon$ ou encore $|x|.|x^n| < |1-x|\varepsilon$. Puisque nous sommes dans le cas $x \neq 0$, nous pouvons transformer cette inégalité en $|x|.|x^n| < |1-x|\varepsilon\frac{|x|}{|x|}$, puis simplifier les deux membres par $|x|$ afin de pouvoir appliquer l'hypothèse $\mathrm{H}_1$ quasi directement.

■

## 2.5 Les fonctions transcendantes

Nous rappelons que les fonctions transcendantes sont des fonctions non algébriques, sachant qu'une fonction algébrique $f(x)$ est une fonction qui peut être exprimée au seul moyen de $x$ et de ses puissances entières ou fractionnaires (racine n-èmes). Les principales fonctions transcendantes qui nous intéressent sont les fonctions *exponentielle, sinus et cosinus*. Nous en donnerons une définition en utilisant les séries entières puis nous verrons comment les définir en Coq. Pour cette formalisation, nous pourrons nous inspirer de celle effectuée en HOL [47] ainsi que de celle effectuée récemment dans PVS [43]. Néanmoins, pour diverses raisons que nous exposerons à la fin de ce chapitre, nous ne pourrons utiliser les preuves faites dans HOL ou PVS. D'autre part, certaines preuves en Coq n'étant pas totalement achevées, nous ne les présenterons pas de manière détaillée ici.

Commençons par donner une définition possible (celle que nous utiliserons) des fonctions *exponentielle, sinus et cosinus* :

$$exp(x) = \sum_{i=0}^{+\infty} \frac{x^i}{i!} \qquad sin(x) = \sum_{i=0}^{+\infty} \frac{(-1)^i x^{2i+1}}{(2i+1)!} \qquad cos(x) = \sum_{i=0}^{+\infty} \frac{(-1)^i x^{2i}}{(2i)!}$$

Nous définissons tout d'abord la fonction factorielle en Coq de manière récursive :

```
Fixpoint fact [n:nat]:nat:=
  Cases n of
    O     => (S O)
   |(S n) => (mult (S n) (fact n))
  end.
```

Pour définir les fonctions transcendantes en Coq, en vue d'obtenir des fonctions se rapprochant des mathématiques, c'est-à-dire des fonctions de type $\mathbb{R} \rightarrow \mathbb{R}$, une manière de procéder est la suivante : montrer que les séries infinies définies ci-dessus convergent vers une limite $l$, puis énoncer que ce $l$ est justement la valeur recherchée. Prenons l'exemple de la fonction exponentielle. Dans la pratique en Coq, cela signifie :

1. Montrer qu'il existe un $l$ vers lequel $\displaystyle\sum_{i=0}^{+\infty} \frac{x^i}{i!}$ converge.
2. Extraire le témoin $l$, ce qui définira exactement $exp(x)$.

Il s'agit donc principalement de montrer la convergence, la seconde étape étant tout à fait triviale en Coq, par exemple, de la façon suivante :

```
Lemma exist_exp:(x:R)(sigTT R [l:R]
  (infinit_sum [i:nat]``/(INR (fact i))*(pow x i)`` l)).

Definition exp:R->R:=[x:R](Cases (exist_exp x) of
                               (existTT a b)=>a end).
```

Pour montrer le lemme de convergence, une solution est d'utiliser la *règle de d'Alembert* ([27] ou [15]).

**Lemme 2.5.1 (Règle de d'Alembert)** *Si* $|\frac{A_{n+1}}{A_n}|$ *admet une limite* $L$ *lorsque* $n \rightarrow +\infty$ *et si* $L < 1$ *alors la série* $\displaystyle\sum_{n=0}^{+\infty} A_n x^n$ *est convergente.*

Ce dernier lemme prouvé[5], il suffit, par exemple, de l'appliquer à $\frac{1}{n!}$ pour définir complètement la fonction exponentielle[6] c'est-à-dire qu'il suffit de monter que $|\frac{\frac{1}{(n+1)!}}{\frac{1}{n!}}| \rightarrow 0$ lorsque $n \rightarrow +\infty$.

## 2.6 Discussion

Exception faite de certains lemmes concernant les fonctions transcendantes et dont les preuves ne sont pas encore achevées en Coq, les développements présentés jusqu'ici sont distribués avec la version de Coq. Nous avons pu confirmer, de par les divers développements faits par les utilisateurs, que les théorèmes de l'analyse peuvent être montrés à partir de

[5]preuve en cours en Coq
[6]cette preuve a été faite en Coq

cette librairie. Nous pouvons en citer un certain nombre, par exemple, la formalisation et la preuve du théorème des accroissements finis[7], faite au sein du projet Lemme à l'INRIA-Sophia-Antipolis.

Comme nous l'avons dit précédemment, nous allons tenter ici d'expliquer pourquoi l'utilisation de développements effectués dans d'autres systèmes, en vue d'un développement en Coq, n'est pas satisfaisante.

Lorsqu'il s'agit de commencer un nouveau développement, indépendamment du l'outil ou du langage de programmation choisi, il y a principalement deux choix possibles lorsque qu'un développement similaire existe dans un autre système : nous pouvons reprendre les développements existant ou pas. Dans ce cas particulier pour Coq, nous avons pris le parti de ne pas réutiliser les développements sur les nombres réels existants (ou en cours) dans d'autres systèmes d'aide à la preuve. Notre choix a été guidé par plusieurs critères, que nous tenterons d'expliquer ici. Nous avons principalement pris en compte le temps de développement d'une telle librairie ainsi que la faisabilité. Lorsque nous avons commencé ce développement, nous nous sommes intéressée à celui existant en HOL [47]. Plus récemment, une bibliothèque similaire a été complétée en PVS [29, 43], dont nous pourrions nous inspirer pour achever notre preuve manquante pour définir complètement les fonctions transcendantes.

Contrairement à Coq, ces deux systèmes sont basés sur une logique classique. De ce fait, même si notre développement est classique de par l'utilisation de l'axiome d'ordre total, les preuves, elles, sont intuitionnistes, excepté lorsque cet axiome est utilisé. Nous sommes donc en mesure d'isoler les parties classiques de notre développement. Ce n'est pas le cas dans les systèmes purement classiques puisque l'automatisation utilise également cette propriété. Certaines définitions ou énoncés de lemmes peuvent également être totalement différents, en particulier de part l'utilisation de l'opérateur de choix de Hilbert souvent utilisé dans ces systèmes classiques tel que HOL mais mal venu dans Coq.

La spécificité des systèmes est la première raison qui, dans notre cas, complique cette démarche de réutilisation de preuves. Mais ce n'est pas la seule. L'automatisation ainsi que le langage des preuves posent également des difficultés. En ce qui concerne l'automatisation, elle est bien sûr différente suivant les systèmes, ce qui rend une traduction quasi inutile. Nous pouvons citer à ce sujet l'exemple simple consistant à montrer que $2 \neq 0$, qui ne demande aucune preuve en PVS de par le codage des nombres réels dans ce système. Le langage de preuve est également un facteur rédhibitoire. Par langage de preuve nous faisons référence à la syntaxe des scripts de preuve du système. Aussi bien HOL que PVS ou que Coq utilisent un langage procédural pour représenter les preuves. Dans la pratique, cela signifie qu'à partir d'un script de preuve, il est impossible d'en déduire la preuve mathématique correspondante, à moins de "rejouer" les preuves dans le système, puis de les transcrire sur papier avant de les retranscrire en Coq. Cette démarche est extrêmement coûteuse en temps de développement. Une parade à ce problème serait l'élaboration d'un langage de preuve qui permettrait de comprendre, en lisant la preuve, le démarche suivie. Des travaux sont fait dans cette optique [49, 21, 24].

Nous pourrions tout à fait comparer cette volonté de réutilisation de preuves à celle de réutiliser des bibliothèques de programmes, telles que, par exemple, les librairies FORTRAN d'analyse numérique. Dans le cas des systèmes de preuves formelles, la tâche est tout aussi ardue, sinon plus.

[7]travail de stage de maîtrise effectué par Marc Chiaverini et Michel Hirschowitz sous la direction de Loïc Pottier

# Chapitre 3

# Le théorème des trois intervalles

Dans ce chapitre nous nous intéresserons à la formalisation et à la preuve d'un théorème mathématique utilisant à la fois les nombres réels et des propriétés relevant de l'intuition géométrique. De ce fait, ce genre de théorèmes est un véritable défi pour un système d'aide à la preuve. En particulier, nous verrons, d'une part, comment formaliser ces intuitions géométriques, et d'autre part, nous discuterons de la pertinence de la formalisation des réels présentée précédemment pour montrer de tels théorèmes.

## 3.1 Petit historique

Le théorème des trois intervalles est à l'origine une conjecture de H.Steinhaus datant des année 50. Ultérieurement, cette conjecture a fait l'objet de plusieurs travaux [81, 84, 83, 45, 80] et une preuve a été donnée par Tony van Ravenstein [89] en 1986. La démonstration que nous proposons ici est une présentation de la preuve entièrement formelle [61] que nous avons construite dans Coq [62] en s'appuyant sur celle de Tony van Ravenstein.

Notons également que ce théorème est actuellement utilisé dans le domaine de l'arithmétique des ordinateurs (étude des nombres flottants), par exemple pour un calcul de minoration de distances lié au *dilemme du fabriquant de tables* [56, 55] lui-même utilisé afin de trouver un meilleur arrondi possible lors de calculs sur les fonctions transcendantes.

## 3.2 Notations et définitions

Commençons par définir les notations et définitions utilisées pour énoncer et prouver ce théorème. Nous traitons de la distribution de $N$ points placés consécutivement autour d'un cercle modulo un angle $\alpha$. Nous pourrons nous référer à la figure 3.1.

### 3.2.1 Notations

- $\mathbb{N}$ désigne l'ensemble des entiers naturels.
- $\mathbb{R}$ désigne l'ensemble des nombres réels.
- La partie entière d'un nombre réel $r$ est notée $E(r)$.
- La partie fractionnaire d'un nombre réel $r$, c'est-à-dire $r - E(r)$, est notée $\{r\}$.
- Le premier point placé sur le cercle est le point 0.

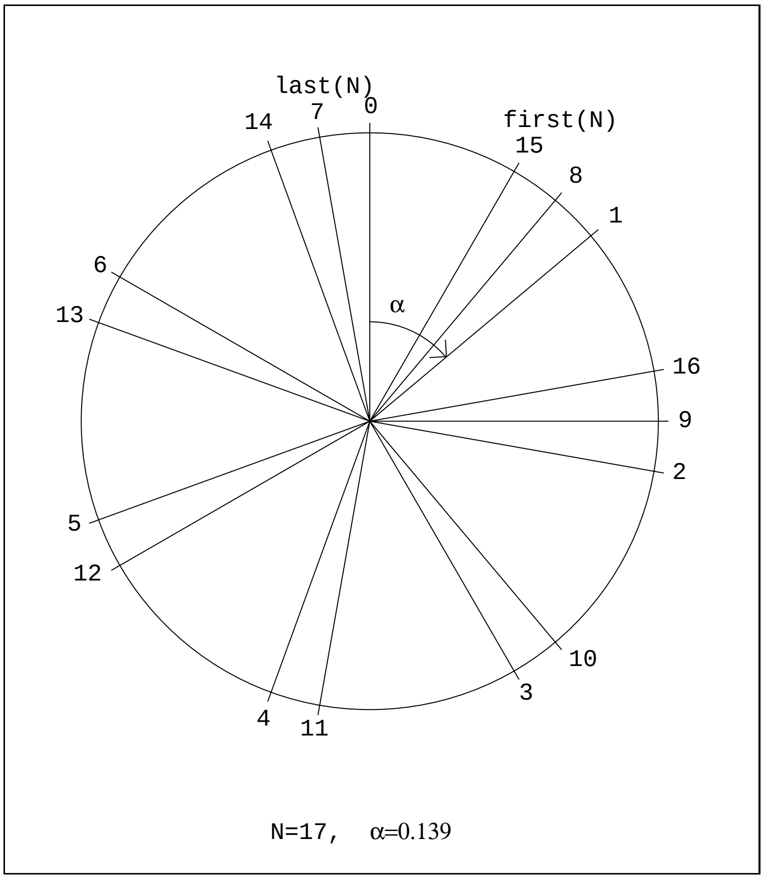



FIG. 3.1 – Le théorème des trois intervalles

- Sauf mention explicite, on considérera $N$ points répartis sur le cercle. On les numérote de 0 à $N-1$.
- On prendra un cercle de **circonférence unité** et orienté dans le sens des aiguilles d'une montre.
- L'unité de $\alpha$ est le tour (et non le radian). Nous avons donc $0 \leq \alpha < 1$.
- Le premier point ($\neq 0$ si $N > 1$) à droite de 0 est noté $first(N)$. $first(N)$ est une fonction des entiers dans les entiers.
- Le dernier point ($\neq 0$ si $N > 1$) avant 0 est noté $last(N)$. $last(N)$ est une fonction des entiers dans les entiers.
- Le successeur d'un point $n$ sur le cercle est noté $after(N, n)$ et est une fonction de couple d'entiers dans les entiers.
- $n \in Cercle$ équivaut à $0 \leq n < N$.

**Remarque 3.2.1** *La distance, sur le cercle, du point* 0 *au point* $n$ *est* $\{n.\alpha\}$.

### 3.2.2 Définitions

L'énoncé, sous forme mathématique, ainsi que la preuve de ce théorème repose sur les définitions des fonctions $first$, $last$ et $after$.

**La fonction $first$**

En Coq, la définition de cette fonction se fait en deux temps. Nous commençons par montrer qu'il existe bien un tel point, puis nous définirons la fonction $first$ elle-même en récupérant[1] le témoin.

**Lemme 3.2.1 (Existence de first)** *Si $N \geq 2$ alors il existe un entier $first(N) \in \mathbb{N}$ t.q. $0 < first(N) < N$ et $\forall m \in \mathbb{N}$ si $0 < m < N$ alors $\{first(N).\alpha\} \leq \{m.\alpha\}$.*

**Preuve** Par induction sur $N$.
. $N = 2$ : dans ce cas $first(N) = 1$.
. Supposons le lemme vrai pour $N$ : nous avons alors qu'il existe $first(N) \in \mathbb{N}$ t.q. $0 < first(N) < N$ et $\forall m \in \mathbb{N}$ si $0 < m < N$ alors $\{first(N).\alpha\} \leq \{m.\alpha\}$ ; montrons qu'il existe $first(N+1) \in \mathbb{N}$ t.q. $0 < first(N+1) < N+1$ et $\forall m \in \mathbb{N}$ si $0 < m < N+1$ alors $\{first(N+1).\alpha\} \leq \{m.\alpha\}$. Nous procédons par cas :
- si $\{first(N).\alpha\} \leq \{N.\alpha\}$ alors $first(N+1) = first(N)$.
- si $\{first(N).\alpha\} > \{N.\alpha\}$ alors $first(N+1) = N$.

Il reste alors à vérifier que dans chaque cas, ces valeurs vérifient les propriétés voulues, ce qui se fait aisément. ∎

Cette preuve d'existence nous permet donc de connaître la valeur,pour un $N$ donné, de $first(N)$. Le lemme d'existence s'exprime en Coq de la façon suivante :

```
Lemma exist_first:(N:nat)(ge N (2))->
   (sigT nat [n:nat]((lt (0) n)/\(lt n N)/\
    (m:nat)(lt (0) m)/\(lt m N)->(ordre_total n m))).
```

où `ordre_total` est le prédicat représentant $\lambda n, m : \mathbb{N}.0 < \alpha < 1- > \{n.\alpha\} \leq \{m.\alpha\}$ défini de la manière suivante :

```
Definition frac_part_n_alpha:nat->R:=
  [n:nat](frac_part ``(INR n)*alpha``).

Definition ordre_total:=[n,m:nat]
 ``0 < alpha < 1``->
  ``(frac_part_n_alpha n) <= (frac_part_n_alpha m)``.
```

et où `sigT` est de type `(A :Set)(A->Prop)->Type` et représentant un quantificateur existentiel, au même titre que `sig` de type `(A :Set)(A->Prop)->Set` déjà défini dans Coq. Définir `sigT` nous est nécessaire pour pouvoir récupérer l'entier naturel qui représente $first(N)$ :

```
Definition first:=[N:nat]
 (Cases (N_classic N) of
    (inright p)=>(<nat>
       Cases (exist_first N p) of
             (existT a b)=>a
       end)
   |   _        => O
  end).
```

[1] Il est fait ici référence au principe d'élimination forte présenté à l'annexe D.

Détaillons cette définition. Comme nous l'avons vu, cette définition n'est valable que pour $N > 2$. Pour $N = 0$ c'est-à-dire lorsqu'il n'y a aucun point sur le cercle nous posons $first(0) = 0$, et pour $N = 1$ c'est-à-dire lorsqu'il n'y a qu'un seul point sur le cercle on a $first(1) = 0$. Le premier `Cases` sur `N_classic` permet de formaliser ces trois cas. Le second `Cases`, quant à lui, permet de récupérer le témoin de l'existentielle[2], `existT` étant le constructeur de `sigT`.

Cette définition pourrait s'énoncer mathématiquement de la façon suivante :

$$first(N) = \begin{cases} 0 & \text{si } N = 0 \text{ ou } N = 1 \\ \text{propriété} & \text{si } N \geq 2 \end{cases}$$

où propriété=l'entier $n$ tel que $0 < n < N$ et $\forall m, 0 < m < N \Rightarrow ordre\_total(n, m)$.

### La fonction *last*

Cette fonction est définie de manière similaire à la fonction $first$ :

**Lemme 3.2.2 (Existence de last)** *Si $N \geq 2$ alors il existe un entier $last(N) \in \mathbb{N}$ t.q. $0 < last(N) < N$ et $\forall m \in \mathbb{N}$ si $0 < m < N$ alors $\{m.\alpha\} \leq \{last(N).\alpha\}$.*

**Preuve** symétrique à celle de l'existence de $first$. ■

### La fonction *after*

Le successeur d'un point sur le cercle (résultat de la fonction $after$) vérifie la propriété suivante (on pourra se référer à la figure 3.2) :

**Lemme 3.2.3 (Propriété des points pour after)** *$\forall L \in \mathbb{R}$ si $0 \leq L < 1$ alors on a soit :*

1. *il existe un entier $I \in \mathbb{N}$ t.q. $0 < I < N$ et $L < \{I.\alpha\}$ et $\forall m \in \mathbb{N}$ si $0 \leq m < N$ et si $\{m.\alpha\} > L$ alors $\{m.\alpha\} \geq \{I.\alpha\}$*
2. *$\forall m \in \mathbb{N}$ si $0 \leq m < N$ alors $0 \leq \{m.\alpha\} \leq L$*

**Preuve** Par induction sur $N$.
. $N = 1$ : 0 vérifie la propriété.
. Supposons le lemme vrai pour $N$ et montrons-le pour $N + 1$ : l'hypothèse de récurrence est la suivante :
soit
1. il existe un entier que nous noterons $I(N) \in \mathbb{N}$ t.q. $0 < I(N) < n$ et $L < \{I(N).\alpha\}$ et $\forall m \in \mathbb{N}$ si $0 \leq m < n$ et si $\{m.\alpha\} > L$ alors $\{m.\alpha\} \geq \{I(N).\alpha\}$
2. $\forall m \in \mathbb{N}$ si $0 \leq m < n$ alors $0 \leq \{m.\alpha\} \leq L$

Montrons que :
soit
1. il existe un entier $I(N+1) \in \mathbb{N}$ t.q. $0 < I(N+1) < N+1$ et $L < \{I(N+1).\alpha\}$ et $\forall m \in \mathbb{N}$ si $0 \leq m < N+1$ et si $\{m.\alpha\} > L$ alors $\{m.\alpha\} \geq \{I(N+1).\alpha\}$
2. $\forall m \in \mathbb{N}$ si $0 \leq m < N+1$ alors $0 \leq \{m.\alpha\} \leq L$

Nous procédons par cas :

- si $0 \leq L < \{N.\alpha\}$ nous nous trouvons dans le cas 1. et nous procédons à nouveau par cas :

[2] Il serait également possible de récupérer la preuve

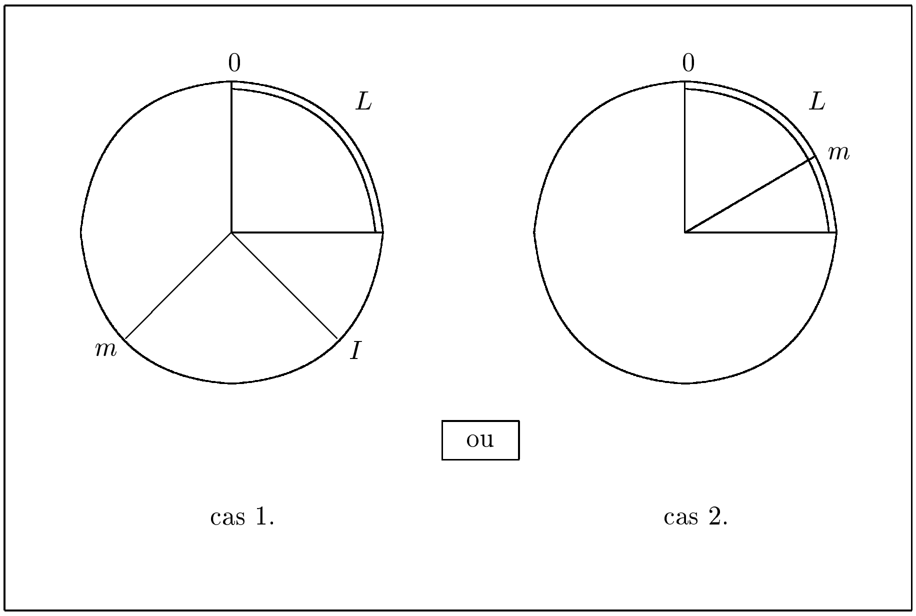



FIG. 3.2 – Propriété des points pour $after$ (lemme 3.2.3)

- si $\{N.\alpha\} < \{I(N).\alpha\}$ alors $I(N+1) = N$.
- si $\{I(N).\alpha\} \leq \{N.\alpha\}$ alors $I(N+1) = I(N)$.

- si $\{N.\alpha\} \leq L < 1$ nous nous trouvons dans le cas 2. et la preuve est immédiate par hypothèse de récurrence. ■

La fonction $after$ peut donc se définir de la façon suivante :

**Définition 3.2.1 (after)** *Pour tout point n du cercle, le point after(N,n) vérifie la propriété des points (lemme 3.2.3) pour* $L = \{n.\alpha\}$ *et est défini tel que :*
*si l'on est dans la cas 1. alors after(N,n)=I*
*si l'on est dans le cas 2. alors after(N,n)=0.*

La propriété sur laquelle repose la définition de $after$ s'exprime ainsi en Coq :

```
Lemma exist_after_L:(L:R)``0 <= L < 1`` -> (N:nat)
 (sumorT
   (sigT nat [I:nat](lt (0) I)/\(lt I N)
              /\``L < (frac_part_n_alpha I)``
              /\((m:nat)(le (0) m)->(lt m N) ->
       ``(frac_part_n_alpha m) > L`` ->
       ``(frac_part_n_alpha m) >= (frac_part_n_alpha I)``))
   (m:nat)(le (0) m)->(lt m N)->
       ``0 <= (frac_part_n_alpha m) <= L``).
```

Nous pouvons à présent formaliser en Coq la définition 3.2.1, en en prenant $L = \{n.\alpha\}$ et en destructurant le "ou" (sumorT), puis le "il existe" (sigT) pour le cas 1. :

```
Definition after:=[N,n:nat]
  (Cases (exist_after_L (frac_part_n_alpha n) (P1 n) (P2 n) N) of
```

```
    (inleftT p) => (<nat>Cases p of (existT a b)=>a end)
   | _         => 0
  end).
```

où (P1 n) et (P2 n) sont les preuves de $0 \leq \{n.\alpha\}$ et $\{n.\alpha\} < 1$.

**Remarque 3.2.2** *En* Coq*, les cas de base des preuves par récurrence sur $N$, que nous avons données précédemment, sont les cas $N = 0$ puisque $N$ est un entier de type* nat*. De ce fait, les preuves des lemmes énoncés pour $N \geq 2$ commencent par des cas absurdes. Par exemple, la preuve* Coq *de* exist_last *commence ainsi :*

```
Induction N; Intro.
Absurd (ge (0) (2));Inversion H;Auto with arith real.
...
```

## 3.3 Enoncés et preuve du théorème

### 3.3.1 Deux énoncés du théorème

Nous commencerons par donner une formulation intuitive du théorème en langage naturel, puis nous donnerons la formulation mathématique utilisée, pour le montrer, par Tony Van Ravenstein. Nous pourrons nous référer à la figure 3.1.

**Théorème 3.3.1 (Enoncé intuitif)** *Soient $N$ points placés consécutivement sur un cercle modulo un angle $\alpha$. Alors pour tout irrationnel $\alpha$ et naturel $N$, les points partitionnent le cercle en au plus trois longueurs différentes d'intervalle.*

Comme nous le montre le théorème 3.3.1 les points sont numérotés dans l'ordre de leur apparition ; or si nous effectuons plus d'un tour de cercle, de nouveaux points viennent s'interposer entre les anciens. Il est possible alors, une fois le dernier point $(N-1)$ placé, de les renuméroter consécutivement dans le sens des aiguilles d'une montre. De cette nouvelle numérotation nous n'utiliserons que les définitions de $first(N)$, $last(N)$ et $after(N,n)$ issues des lemmes 3.2.1 et 3.2.2 et de la définition 3.2.1.

Si nous posons $\|x\|=min(\{x\}, 1-\{x\})$, la distance entre un point $n$ et son successeur $after(N,n)$ est alors donnée par $\|after(N,n)-n\|$. Pour montrer que cette fonction ne peut prendre au plus que trois valeurs on montre que la fonction $(after(N,n)-n)$ elle-même ne prend au plus que trois valeurs.

Démontrer le théorème 3.3.1 revient alors à montrer la formulation mathématique qui va suivre et que nous allons prouver dans le paragraphe suivant.

**Théorème 3.3.2 (Le théorème des 3 intervalles)**

$$after(N,m) - m = \begin{cases} first(N) & si\ 0 \leq m < N - first(N) \\ first(N) - last(N) & si\ N - first(N) \leq m < last(N) \\ -last(N) & si\ last(N) \leq m < N \end{cases}$$

**Remarque 3.3.1**

1. *Cette transcription signifie que le cercle de $N$ points est divisé en $N - first(N)$ intervalles de longueur $\|\ first(N).\alpha\ \|$, $N - last(N)$ intervalles de longueur $\|\ last(N).\alpha\ \|$ et $first(N) + last(N) - N$ intervalles de longueur $\|\ first(N).\alpha\ \| + \|\ last(N).\alpha\ \|$.*

2. *Le théorème 3.3.2 est vrai pour $\alpha$ rationnel et irrationnel. Cependant nous ne présenterons, ici, que la preuve pour $\alpha$* **irrationnel**. *En effet, la plupart des résultats intermédiaires sont faux pour $\alpha$ rationnel (entre autres puisque $first$, $last$ et $after$ ne sont plus des fonctions puisque l'on pourra avoir, par exemple, $first(18) = 7 = 17$ pour $\alpha = 3/10$). D'autre part, le théorème est trivialement vrai pour $\alpha$ rationnel*[3] *: si $\alpha = p/q$ alors le cercle comprend, au plus, trois longueurs d'intervalle si $N < q$ et une longueur d'intervalle si $N \geq q$.*

### 3.3.2 Preuve

Nous rappelons que la preuve est détaillée pour $\alpha$ irrationnel et $N \geq 2$.

Commençons par donner l'idée générale de la preuve. On considère qu'il y a $first(N) + last(N)$ points sur le cercle (cette valeur sera notée $M$). Il s'agit du cas particulier où il y a, au plus, deux longueurs d'intervalles. Ce cas particulier étant montré, il s'agit alors de supprimer les $M - N$ points excédents et de montrer qu'il y a alors une troisième distance.

**Cas particulier : au plus 2 longueurs**

**Lemme 3.3.1 (Cas particulier)** *Si $N = first(N) + last(N)$ alors*

$$after(N, m) - m = \begin{cases} first(N) & si\ 0 \leq m < last(N) \\ -last(N) & si\ last(N) \leq m < N \end{cases}$$

La figure 3.3 illustre ce cas.

**Preuve**

1. Cas $0 \leq m < last(N)$ :
   Pour $m = 0$, par définition de $first$ nous avons que $after(N, 0) = first(N)$.

   Nous voulons prouver que $m + first(N)$ est le successeur de $m$ sur le cercle. Commençons par montrer que $m + first(N)$ appartient au cercle de $N$ points :
   si $0 < m < last(N)$ alors $0 < 0 + first(N) \leq m + first(N) < last(N) + first(N) = N$.

   Montrons maintenant que : soit $i$ un point quelconque du cercle ($0 < i < N$) alors on a soit $\{i.\alpha\} < \{m.\alpha\}$ soit $\{i.\alpha\} > \{(m + first(N)).\alpha\}$. Par l'absurde et par cas : supposons que $\{m.\alpha\} < \{i.\alpha\} < \{(m + first(N)).\alpha\}$

   – si $i > m$ alors $0 < \{i.\alpha\} - \{m.\alpha\} < \{(m + first(N)).\alpha\} - \{m.\alpha\}$ donc $0 < \{(i - m).\alpha\} < \{first(N).\alpha\}$, ce qui contredit la définition de $first$ puisque $(i - m) \in$ Cercle car $0 < i - m < N$.

   – si $i \leq m$ alors $\{(m + first(N)).\alpha\} - \{i.\alpha\} < \{(m + first(N)).\alpha\} - \{m.\alpha\}$ donc $\{(m + first(N) - i).\alpha\} < \{first(N).\alpha\}$, ce qui contredit la définition de $first$ puisque $(m + first(N) - i) \in$ Cercle car $0 < m + first(N) - i < N$.

   Dans les deux cas précédents $\{(m + first(N)).\alpha\} - \{m.\alpha\} = \{first(N).\alpha\}$ si $\{m.\alpha\} < \{(m + first(N)).\alpha\}$. Montrons-le par l'absurde :
   si $\{(m + first(N)).\alpha\} \leq \{m.\alpha\}$ alors

[3] Un bon exemple pour s'en convaincre est de choisir $\alpha = 3/10$

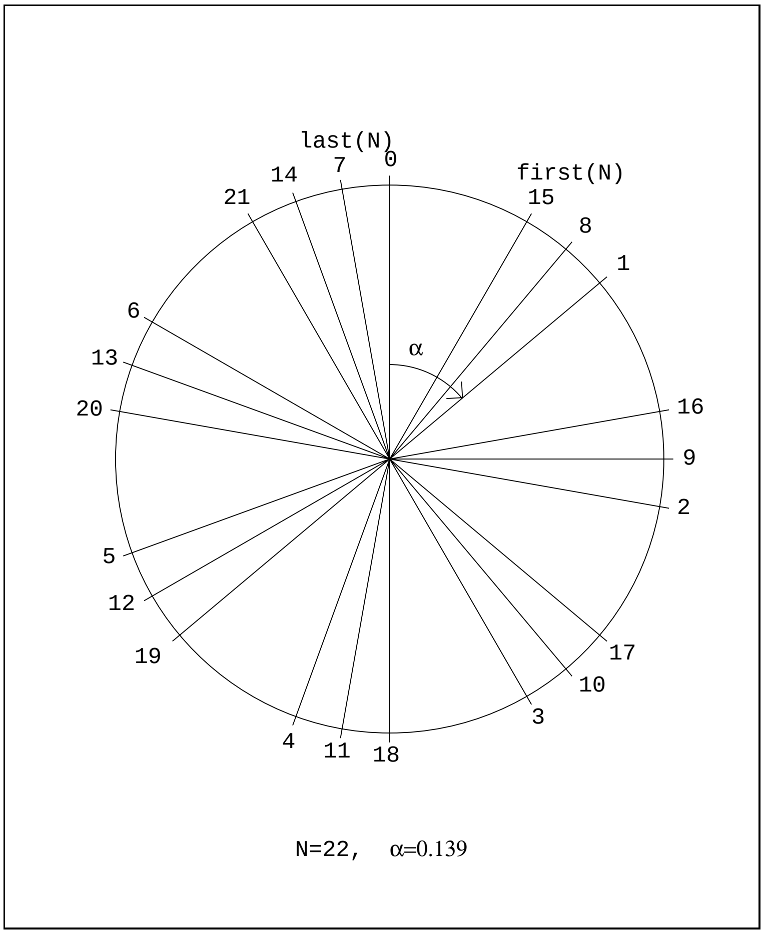



FIG. 3.3 – Cas particulier : $N = first(N) + last(N)$

$\{(m + first(N)).\alpha\} - \{first(N).\alpha\} \leq \{m.\alpha\} - \{first(N).\alpha\}$ donc par définition de $first$, $\{m.\alpha\} \leq \{m.\alpha\} - \{first(N).\alpha\}$ ce qui est absurde car $\{first(N).\alpha\} > 0$ pour $N \geq 2$.

2. Cas $last(N) \leq m < N$ :
   Pour $m = last(N)$, par définition de $last$ nous avons que $after(N, last(N)) = 0$.

   Nous voulons prouver que $m - last(N)$ est le successeur de $m$. Commençons par montrer que $m - last(N)$ appartient au cercle de $N$ points : si $last(N) < m < N$ alors $0 \leq last(N) - last(N) < m - last(N) < N - last(N) < N$ car $last(N) > 0$.

   Montrons maintenant que : soit $i$ un point quelconque du cercle $(0 < i < N)$ alors on a soit $\{i.\alpha\} < \{m.\alpha\}$ soit $\{i.\alpha\} > \{(m - last(N)).\alpha\}$. Par l'absurde et par cas : supposons que $\{m.\alpha\} < \{i.\alpha\} < \{(m - last(N)).\alpha\}$

   – si $i < m$ alors $\{m.\alpha\} - \{i.\alpha\} + 1 > \{m.\alpha\} - \{(m - last(N)).\alpha\} + 1$ donc $\{(m - i).\alpha\} > \{last(N).\alpha\}$, ce qui contredit la définition de $last$ puisque

$(m-i) \in$ Cercle car $0 < m-i < N$.

– si $m \leq i$ alors $\{m.\alpha\} - \{(m - last(N)).\alpha\} + 1 < \{i.\alpha\} - \{(m - last(N)).\alpha\} + 1$ donc $\{last(N).\alpha\} < \{(i + m - last(N)).\alpha\}$ ce qui contredit la définition de $last$ puisque $(i + m - last(N)) \in$ Cercle car $0 < i + m - last(N) < N$.

Dans les deux cas précédents $\{m.\alpha\} - \{(m - last(N)).\alpha\} + 1 = \{last(N).\alpha\}$ si $\{m.\alpha\} < \{(m - last(N)).\alpha\}$. Comme $\alpha$ est irrationnel et par définition de $last$ on a $\{m.\alpha\} < \{last(N).\alpha\}$ et donc $\{(m - last(N)).\alpha\} = \{m.\alpha\} - \{last(N).\alpha\} + 1$ et l'on a bien $\{m.\alpha\} < \{m.\alpha\} - \{last(N).\alpha\} + 1$ puisque $\{last(N).\alpha\} < 1$.

**Remarque 3.3.2** *Le fait que $\alpha$ soit irrationnel est essentiel pour montrer que $\{m.\alpha\} \neq \{last(N).\alpha\}$. En effet, car dans ce cas on a $m \neq last(N)$ donc $\{m.\alpha\} \neq \{last(N).\alpha\}$. Montrons-le par contradiction. Pour ce, supposons que $\{m.\alpha\} = \{last(N).\alpha\}$. Alors $\{m.\alpha\} - \{last(N).\alpha\} = 0$ et donc $\{(m - last(N)).\alpha\} = 0$ et comme seule un nombre entier a une partie fractionnaire nulle on a $(m - last(N)).\alpha = k$, $k \in \mathbb{N}$ d'où $\alpha = \frac{k}{m - last(N)}$, ce qui contredit $\alpha$ irrationnel.*

■

En Coq, nous formaliserons cette propriété en deux fois et en utilisant le principe de `Section` où nous définissons les hypothèses nécessaires, en particulier $\alpha$ irrationnel, $0 < \alpha < 1$ et $N \geq 2$ :

```
Section particular.
Hypothesis alpha_irr:(n,p:Z)``alpha*(IZR p) <> (IZR n)``.
Hypothesis prop_alpha:``0 < alpha < 1``.
Hypothesis prop_N:(N:nat)(ge N (2)).
```

Chacun des deux cas est exprimé séparément en posant $M(N) = first(N) + last(N)$ :

```
Definition M:=[N:nat](plus (first N) (last N)).
```

1. Cas $0 < m < last(N)$ :

```
Lemma inter31a:(N:nat)(n:nat)(lt O n)->(lt n (last (M N)))->
                          (after (M N) n)=(plus n (first (M N))).
```

2. Cas $last(N) \leq m < N$ :

```
Lemma inter31b:(N:nat)(n:nat)(le (last (M N)) n)->(lt n (M N))->
                          (after (M N) n)=(minus n (last (M N))).
```

Comme nous avons pu le constater précédemment lors de la preuve mathématique, un certain nombre de propriétés auxiliaires concernant les fonctions $first$, $last$ et $after$ sont utilisées. Nous pouvons citer par exemple :

– $after(N, (last(N))) = 0$
– $\forall n \in Cercle \Rightarrow \{first(N).\alpha\} \leq \{n.\alpha\}$
– $n \neq m \Rightarrow \{n.\alpha\} \neq \{m.\alpha\}$

**Relations entre $N$ et $M$.**

La démonstration du cas général repose également sur des propriétés particulières concernant le nombre de points réellement sur le cercle $N$ et le nombre de points $M(N)$ tel qu'il a été défini précédemment c'est-à-dire $M(N) = first(N) + last(N)$.

Par souci de simplicité, nous noterons $M$ au lieu de $M(N)$. Nous présenterons les cinq principaux lemmes nécessaires à la généralisation du théorème. Nous donnerons leur formalisation en Coq. Celle-ci s'appuie, comme pour le cas particulier précédent, sur le principe de `Section` afin de factoriser les trois hypothèses nécessaires (les mêmes qu'à la page 59).

**Lemme 3.3.2** *$N \leq M$.*

Ou en Coq :

```
Lemma le_N_M:(N,M:nat)(M=(plus (first N) (last N)))->(le N M).
```

**Preuve** Par l'absurde. On suppose que $M < N$ et on montre que dans ce cas le point $first(N) + last(N)$ se trouve soit avant $first(N)$ soit après $last(N)$, ce qui contredit leur définition.
Montrons donc que
soit $\{(first(N) + last(N)).\alpha\} < \{first(N).\alpha\}$
soit $\{(first(N) + last(N)).\alpha\} > \{last(N).\alpha\}$ :
Discutons selon les deux cas suivant :

1. $\{first(N).\alpha\} + \{last(N).\alpha\} < 1$ : puisque pour $N \geq 2$ $\{first(N).\alpha\} > 0$ on peut écrire que $\{first(N).\alpha\} + \{last(N).\alpha\} > \{last(N).\alpha\}$ donc que $\{(first(N) + last(N)).\alpha\} > \{last(N).\alpha\}$.
2. $\{first(N).\alpha\} + \{last(N).\alpha\} \geq 1$ : de la même manière on peut écrire, en utilisant le fait qu'une partie fractionnaire est comprise entre 0 et 1 ($0 \leq \{\} < 1$), que $\{first(N).\alpha\} + \{last(N).\alpha\} - 1 < \{first(N).\alpha\}$ donc que $\{(first(N) + last(N)).\alpha\} < \{first(N).\alpha\}$.

■

**Lemme 3.3.3** *$first(N) = first(M)$.*

En Coq :

```
Lemma first_eq_M_N:(N:nat)(M:nat)(M=(plus (first N) (last N)))->
                    (first N)=(first M).
```

**Preuve** Par définition de $first$ nous savons que pour tout $a$ et $b$ t.q.
$0 < a < N$ et $0 < b < N$ on a que si $\{a.\alpha\} \leq \{b.\alpha\}$ alors $a = (first(N))$.

Utilisons cette définitions avec $N = M$ et $a = (first(N))$ et l'on a alors pour tout $b$ t.q.$0 < b < M$ si $\{(first(N)).\alpha\} \leq \{b.\alpha\}$ alors $(first(N)) = (first(M))$.

Il nous suffit maintenant de montrer que $\{(first(N)).\alpha\} \leq \{b.\alpha\}$ $\forall b$, $0 < b < M$ :
Pour $0 < b < N$ il s'agit de la définition de $first$ (lemme 3.2.1).

Pour $N \leq b < M$ raisonnons par l'absurde : supposons que $\{b.\alpha\} < \{(first(N)).\alpha\}$
Puisque $b < M = first(N) + last(N)$ on a immédiatement que $b - first(N) < last(N) < N$ et $b - last(N) < first(N) < N$ et par définition de $first$ et $last$ que
$\{(b - first(N)).\alpha\} \leq \{last(N).\alpha\}$ et $\{first(N).\alpha\} \leq \{(b - last(N)).\alpha\}$. On a donc grâce à l'hypothèse de contradiction et aux lemmes 3.2.1 et 3.2.2 que :
$\{b.\alpha\} - \{first(N).\alpha\} + 1 \leq \{last(N).\alpha\}$ et $\{first(N).\alpha\} \leq \{b.\alpha\} - \{last(N).\alpha\} + 1$ ce qui implique que $\{last(N).\alpha\} + \{first(N).\alpha\} - \{b.\alpha\} - 1 = 0$ donc que

$\{(b - first(N)).\alpha\} = \{last(N).\alpha\}$.
Or comme nous l'avons montré dans la remarque 3.3.2 cette égalité oblige $\alpha$ à être rationnel si $b - first(N) \neq last(N)$ ce qui est le cas. ■

**Lemme 3.3.4** *$last(N) = last(M)$.*

En Coq :

```
Lemma last_eq_M_N:(N:nat)(M:nat)(M=(plus (first N) (last N)))->
                  (last N)=(last M).
```

**Preuve** Preuve symétrique à la précédente. ■

**Lemme 3.3.5** *Pour tout $n$ t.q. $0 < n < N - first(N)$ et $last(N) < n < N$ on a $after(N,n) = after(M,n)$.*

En Coq :

```
Lemma eq_after_M_N1:(N,n:nat)(lt O n)->(lt n (minus N (first N)))->
                  (after (M N) n)=(after N n).

Lemma eq_after_M_N2:(N,n:nat)(le (last N) n)->(lt n N)->
                  (after (M N) n)=(after N n).
```

**Preuve** Nous allons procéder par cas :

1. Cas $0 < n < N - first(N)$ : En utilisant l'irrationalité de $\alpha$ (contraposée de la remarque 3.3.2) on a que prouver ce lemme revient à prouver que $\{after(N,n).\alpha\} = \{after(M,n).\alpha\}$ ce qui équivaut également à montrer que $\{after(N,n).\alpha\} \leq \{after(M,n).\alpha\}$ et $\{after(M,n).\alpha\} \leq \{after(N,n).\alpha\}$.
   Procédons par cas et par l'absurde :

   - Cas $\{after(N,n).\alpha\} \leq \{after(M,n).\alpha\}$ :
     Supposons que $\{after(N,n).\alpha\} > \{after(M,n).\alpha\}$.
     D'après le lemme 3.2.3, on montre immédiatement la propriété suivante : $\forall N \in \mathbb{N}, \forall n,k \in Cercle$, si $\{n.\alpha\} < \{k.\alpha\}$ et si $\{k.\alpha\} \neq \{after(N,n).\alpha\}$ alors $\{after(N,n).\alpha\} < \{k.\alpha\}$. Utilisons cette propriété avec $k = after(M,n)$. Nous obtenons directement la contradiction à condition que les hypothèses soient bien vérifiées :
     - $n \in Cercle$ i.e. $0 < n < N$ ; vrai car cas 1.
     - $after(M,n) \in Cercle$ i.e. $0 < after(M,n) < N$ vrai en utilisant le lemme 3.3.1.
     - $\{n.\alpha\} < \{after(M,n).\alpha\}$ par définition de $after$ (lemme 3.2.3 et définition 3.2.1) + lemme 3.3.1 (pour montrer que $after(M,n) \neq 0$).
     - $\{after(M,n).\alpha\} \neq \{after(N,n).\alpha\}$ vrai par hypothèse de contradiction.

   - Cas $\{after(M,n).\alpha\} \leq \{after(N,n).\alpha\}$ :
     Supposons que $\{after(M,n).\alpha\} > \{after(N,n).\alpha\}$. Nous utilisons la même propriété en prenant $k = after(N,n)$ et $N = M$ (sauf dans k).

2. Cas $last(N) < n < N$ : la preuve se fait de manière identique.

■

Le lemme 3.3.5 a permis de calculer $after(N,n)$ pour deux intervalles :
$0 < n < N - first(N)$ et $last(N) < n < N$.

La valeur de $after(N, n)$, pour $n$ se trouvant dans le troisième intervalle se déduit du lemme suivant :

**Lemme 3.3.6** *Pour tout n, $N - first(N) \leq n < last(N)$ il n'existe pas de $k \in Cercle$ t.q. $\{n.\alpha\} < \{k.\alpha\} < \{(n + first(N) - last(N)).\alpha\}$.*

En Coq :

```
Lemma prop_after:(N,n,m:nat)(after N n)=m->
 (Ex [k:nat] (lt O k)/\(lt k N)/\
 ``(frac_part_n_alpha n) < (frac_part_n_alpha k) < (frac_part_n_alpha m)``)
 -> False.
```

**Preuve** Par l'absurde. Supposons qu'il existe un $k \in Cercle$ t.q.
$\{n.\alpha\} < \{k.\alpha\} < \{(n + first(N) - last(N)).\alpha\}$.
Ce $k$ vérifie l'un des trois cas suivants (ordre total sur les réels) :

1. si $\{k.\alpha\} < \{after(M.n).\alpha\}$ : alors on a $\{n.\alpha\} < \{k.\alpha\} < \{after(M.n).\alpha\}$ ce qui contredit la définition de $after$.
2. si $\{k.\alpha\} = \{after(M.n).\alpha\}$ : d'après les lemmes 3.3.1 et 3.3.3
   $after(M, n) = n + first(N)$.
   Or comme $\alpha$ est irrationnel, on devrait avoir $k = n + first(N)$ ce qui contredit les hypothèses $n < last(N)$ et $k < N$ (en utilisant le lemme 3.3.2).
3. si $\{k.\alpha\} > \{after(M.n).\alpha\}$ : on utilise la propriété déjà vue que
   $\forall N \in \mathbb{N}, \forall j, k \in Cercle$, si $\{j.\alpha\} < \{k.\alpha\}$ et si $\{k.\alpha\} \neq \{after(N, j).\alpha\}$ alors
   $\{after(N, j).\alpha\} < \{k.\alpha\}$ avec $N = M$, $j = n + first(N)$.
   On a alors, en utilisant principalement le lemme 3.3.1 que
   $\{(n + first(N) - last(N)).\alpha\} < \{k.\alpha\}$, ce qui contredit l'hypothèse.

■

### Preuve du théorème dans le cas général (théorème 3.3.2)

Supposons que le cercle comprenne $M$ points. Nous savons alors montrer le théorème (lemme 3.3.1). Il suffit maintenant d'"effacer" les $M - N$ points en trop, ce qui fait apparaître la troisième distance.

Une formalisation intermédiaire, en Coq, du cas général, consiste à énoncer et à prouver les trois cas séparément (en considérant les trois hypothèses nécessaires citées précédemment) :

```
Lemma three_gap1:(N:nat)(n:nat)
                 (lt O n)->(lt n (minus N (first N)))->
                 (after N n)=(plus n (first N)).

Lemma three_gap2:(N:nat)(n:nat)
                 (le (minus N (first N)) n)->(lt n (last N))->
                 (after N n)=(minus (plus n (first N)) (last N)).

Lemma three_gap3:(N:nat)(n:nat)(le (last N) n)->(lt n N)->
                 (after N n)=(minus n (last N)).
```

En utilisant les propriétés démontrées précédemment nous avons la preuve suivante pour le théorème :

**Preuve**

1. $0 \leq n < N - first(N)$ : d'après le lemme 3.3.2 on a que $0 \leq n < N - first(N) < last(N)$. En utilisant les lemmes 3.3.3, 3.3.4, 3.3.5 et 3.3.1 nous obtenons immédiatement le résultat.
2. $N - first(N) \leq n < last(N)$ : En utilisant le lemme 3.3.6 on montre que les $M - N$ points à partir de $N$ n'existent pas et par définition de $after$ (lemme 3.2.3 et définition 3.2.1) nous avons le résultat.
3. $last \leq n < N$ : identique au cas 1.

■

Nous pouvons maintenant donner une formalisation Coq plus compacte et plus lisible du théorème se rapprochant de l'énoncé donné en langage naturel (théorème 3.3.1).

```
Section gap.
Variable N,n:nat.
Hypothesis alpha_irr:(n,p:Z)``alpha*(IZR p) <> (IZR n)``.
Hypothesis prop_alpha:``0 < alpha < 1``.
Hypothesis prop_N:(N:nat)(ge N (2)).
Hypothesis Hn:(lt O n)/\(lt n N).

Definition succes:=(after N n).
Definition num1:=(plus n (first N)).
Definition num2:=(minus (plus n (first N)) (last N)).
Definition num3:=(minus n (last N)).

Theorem three_gap:
      (succes=num1)\/(succes=num2)\/(succes=num3).
End gap.
```

## 3.4 Discussion

La preuve formelle de ce théorème a mis en avant plusieurs aspects que nous allons énumérer et commenter.

### 3.4.1 Le théorème

A partir de la preuve entièrement formalisée en Coq, nous pourrons faire une comparaison entre la preuve détaillée issue de la preuve formelle et celle de Tony van Ravenstein.

#### Les parties fractionnaires

De nombreux lemmes intermédiaires ont dû être prouvés. La preuve formelle a, par exemple, permis d'identifier quatre lemmes concernant la partie fractionnaire[4] restés implicites dans la preuve de Tony van Ravenstein et qui sont au coeur de la démonstration.

[4]nous avons donné une preuve de ces lemmes dans le chapitre 1 concernant la formalisation des propriétés de base pour les nombres réels

- si $\{r1\} + \{r2\} \geq 1$ alors $\{r1 + r2\} = \{r1\} + \{r2\} - 1$
- si $\{r1\} + \{r2\} < 1$ alors $\{r1 + r2\} = \{r1\} + \{r2\}$
- si $\{r1\} \geq \{r2\}$ alors $\{r1 - r2\} = \{r1\} - \{r2\}$
- si $\{r1\} < \{r2\}$ alors $\{r1 - r2\} = \{r1\} - \{r2\} + 1$

**Cas dégénérés**

La preuve formelle permet d'identifier les cas dégénérés tels que $N = 0$, $N = 1$ qui peuvent être passés sous silence lors d'une preuve informelle.
Dans le même ordre d'idées, cette preuve formelle nous a permis de constater que l'axiome de complétude des réels n'est pas nécessaire. L'énoncé ainsi que la preuve de ce théorème sont en fait vrais pour tous les corps commutatifs, ordonnés et archimédiens. L'axiome d'Archimède pourrait d'ailleurs être remplacé par un axiome plus faible, suffisant pour définir la partie fractionnaire. Une généralisation du théorème aux groupes a été donnée par E.Fried et V.T.Sós [36].

**$\alpha$ irrationnel**

$\alpha$ irrationnel est une hypothèse utilisée par Tony van Ravenstein mais la formalisation montre précisément où cette hypothèse intervient (cf. remarque 3.3.2). En particulier, si $\alpha$ est rationnel, certains points du cercle peuvent être confondus et $first$, *last* ou $after$ ne sont alors plus des fonctions.

**Les fonctions $first$, *last* et $after$**

Lors de la preuve informelle de Tony van Ravenstein, on constate que l'on peut tolérer une imprécision sur la dépendance de $first$, *last* et $after$ par rapport à N ou M. Bien que ce ne soit pas une erreur, la preuve formelle a mis en évidence la nécessité de prouver ces lemmes[5] qui ne sont pas absolument triviaux (lemmes 3.3.3, 3.3.4 et 3.3.5). La preuve formelle permet de dire précisément où ces lemmes sont utilisés.

### 3.4.2 La logique

La preuve formelle effectuée dans le système Coq -de par l'utilisation de l'axiomatisation des réels comme corps commutatif, ordonné, archmédien et complet- est une preuve classique vu qu'une propriété d'ordre total intuitionniste impliquerait la décidabilité de l'égalité sur les nombres réels. Néanmoins, on peut s'interroger sur la possibilité de donner une preuve constructive du théorème des trois intervalles.
Nous pourrions probablement donner une preuve intuitionniste pour chacun des deux cas, suivant que $\alpha$ est rationnel ou irrationnel car nous connaissons la longueur d'intervalle entre deux points du cercle. Mais les deux cas ne peuvent être traités en même temps. Il faut donc supposer que $\alpha$ est rationnel ou ne l'est pas et nous ne voyons pas, pour l'instant, comment éviter cette distinction.

### 3.4.3 Pertinence

Montrer un théorème purement mathématique tel que celui-ci en utilisant un système d'aide à la preuve permet de déterminer les éventuelles limites du système utilisé. Dans le cas particulier de Coq, nous avons pu confirmer qu'un tel système est tout à fait adapté.

[5] qui ne le sont pas dans [89]

Nous nous sommes particulièrement intéressée à l'axiomatisation des nombres réels afin de déterminer si une axiomatisation est suffisante pour faire de tels développements.

Comme nous avons pu le constater, il nous a été possible de formaliser et de montrer un théorème faisant intervenir à la fois des notions de géométrie, des propriétés numériques et d'analyse standard. Ceci nous permet de conclure que notre axiomatisation est suffisante. Cette affirmation est également étayée par notre expérience concernant l'utilisation d'autres systèmes d'aide à la preuve où une bibliothèque des nombres réels est également disponible. Prenons le cas d'HOL [42] et de sa bibliothèque des réels développée par John Harrison [47]. Comme nous l'avons dit au chapitre 1, il s'agit d'une construction classique des nombres réels. Néanmoins, son utilisation n'est pas différente de la nôtre : tout comme notre type `R`, le type des réels de HOL n'est pas un type inductif. De ce fait, les preuves se font également à grand renfort de réécritures, l'unique différence étant que, dans HOL, les propriétés de commutativité, associativité, etc. ont étés prouvées, tandis que dans notre formalisation, ce sont des axiomes. Ils pourraient d'ailleurs être montrés ultérieurement, en utilisant la même méthode que dans HOL, ou une autre.

Même si cette axiomatisation est suffisante, en théorie, pour développer certains théorèmes des mathématiques, elle s'avère être trop pauvre en particulier pour ce qui est de l'automatisation. Dans le chapitre suivant nous développerons des tactiques d'automatisation rendant son utilisation plus simple et agréable.

# Chapitre 4

# Automatisation pour les nombres réels

Dans ce chapitre nous traiterons des tactiques d'automatisation spécialement conçues et adaptées pour la bibliothèque des nombres réels que nous avons développée et présentée aux chapitres précédents. Pour cela, nous commencerons par présenter brièvement le langage de tactiques utilisé. Nous donnerons ensuite quelques exemples de tactiques simples avant de détailler la tactique réflexive *Field*.

## 4.1 Motivations

Dans cette section nous allons expliquer pourquoi il nous a été nécessaire d'implanter des tactiques spécifiques aux nombres réels.

Les opérateurs tels que l'addition ou la multiplication ne sont pas définis explicitement en utilisant les constructeurs de type, comme c'est le cas pour les plus des entiers ou des relatifs. Montrons, par exemple, que $0+2=1+1$ pour les entiers naturels et pour les réels. Le `plus` du type `nat`$\rightarrow$`nat`$\rightarrow$`nat` est défini de la manière suivante :

```
Fixpoint plus [n:nat] : nat -> nat :=
   [m:nat]Cases n of
      O   => m
  | (S p) => (S (plus p m)) end.
```

De ce fait, la preuve du but suivant, dans les naturels, est triviale en `Coq`. `(plus (0) (2))` se réduira ($\delta$ puis $\iota$ puis $\beta$ conversions) en `(2)` tandis que `(plus (1) (1))` se réduira en `(S (plus (0) (1)))` qui se réduira encore en `(S (1))` qui est `(2)` :

```
Coq < Goal (plus (0) (2))=(plus (1) (1)).

Unnamed_thm < Simpl.
1 subgoal

  ============================
   (2)=(2)
```

**Remarque 4.1.1** *La tactique* `Simpl` *équivaut ici à la tactique* `Cbv Delta Iota Beta`*.*

Pour montrer le même but[1], mais avec `0,1,2 : R`, nous ne pouvons utiliser les règles de conversion. Seule une tactique de réécriture peut s'appliquer :

```
Coq < Goal ``0+2==1+1``.

Unnamed_thm < Rewrite Rplus_Ol.
1 subgoal

  ============================
   ``2==2``
```

Même si, dans ce cas très simple, l'utilisation d'un `Rewrite` ne semble pas particulièrement laborieuse, il n'en est rien pour des propriétés plus grandes en taille mais qui restent simples, telles que, par exemple, des propriétés purement arithmétiques. En effet, si une seule tactique permet de montrer ces propriétés sur les entiers ou les relatifs, il faut souvent dix ou quinze tactiques de réécriture en utilisant les lemmes appropriés lorsqu'il s'agit des nombres réels.

## 4.2 Préliminaire : la tactique `Ring`

Au long de ce chapitre, nous ferons souvent référence à la tactique `Ring`. Cette tactique a été développée en Ocaml par Samuel Boutin et Patrick Loiseleur [10]. Elle traite les égalités modulo la théorie des anneaux abéliens.

Initialement, cette tactique ne pouvait être utilisée que pour des anneaux et semi-anneaux définis en Coq dans la sorte `Set`. Le type des réels étant défini dans `Type`, nous ne pouvions instancier la tactique avec l'anneau sur les nombres réels. Cette tactique a donc été adaptée à cette fin. Grâce à la propriété de cumulativité, son comportement ne change pas en ce qui concerne `Set`. Nous donnons ci-dessous deux exemples d'utilisation sur les réels.

Ce premier exemple représente l'utilisation la plus souvent utilisée : la preuve d'une égalité. Comme nous le verrons, nous en faisons une telle utilisation lors du codage de `Field`.

```
Goal (a,b,c:R)``2*(a+b)*c-2*a*c==2*b*c``.
...

Unnamed_thm < Intros.
1 subgoal

  a : R
  b : R
  c : R
  ============================
   ``2*(a+b)*c-2*a*c == 2*b*c``

Unnamed_thm < Ring.
Subtree proved!
```

[1] notons que "1+1" équivaut à "2" et est affiché comme tel

Ce second exemple représente l'utilisation de cette tactique en vue d'une normalisation[2] du terme donné en paramètre, ce terme étant alors remplacé dans le but courant. Nous en faisons une telle utilisation dans le codage de `DiscrR`.

```
Coq < Goal ``5+2 < 8``.
1 subgoal

  ============================
   ``5+2 < 8``

Unnamed_thm < Ring ``5+2``.
1 subgoal

  ============================
   ``7 < 8``
```

Dans ce cas, `Ring` normalise le terme $5+2$ qui est représenté par $(1+(1+(1+(1+1))))+(1+1)$. La normalisation faite par la tactique nous donne le terme distribué et associé à droite $(1+(1+(1+(1+(1+(1+1))))))$, ce qui est affiché 7. Remarquons que si nous avions choisi le terme $5-2$, nous aurions obtenu le terme 3, par simplification des termes de type $1-1$.

## 4.3 Le langage de tactiques

Nous ne ferons pas une présentation détaillée de ce nouveau langage de tactiques présent à partir de la version V7 de `Coq`, dont nous pourrons trouver une description dans [23]. Nous allons néanmoins exposer les particularités du langage et ainsi expliquer pourquoi nous avons choisi cette méthode plutôt qu'un codage habituel en `Ocaml`.

Avant la version V7 de `Coq`, écrire une nouvelle tactique était possible, soit en utilisant les combinateurs de tactiques (appelés "tacticals" en `Coq`), soit en utilisant directement le langage d'implantation de `Coq`, c'est-à-dire `Ocaml`. Les combinateurs de tactiques ne permettent pas de développer des automatisations très puissantes. Très rapidement, il était souvent impératif d'utiliser `Ocaml`, ce qui, dans ce cas, représente un langage plus bas niveau, cas moins abstrait que `Coq`. De plus ce choix de programmation des tactiques d'automatisation incontournable demande une bonne connaissance du code d'implantation de `Coq`, ce qui peut sembler rebutant pour certains utilisateurs, d'autant plus que, bien souvent, l'écriture de la tactique souhaitée ne demande pas toute la puissance d'un langage de programmation tel que `Ocaml`. Nous présenterons dans la suite de ce chapitre la tactique `SplitAbsolu`. Cette tactique a d'abord été écrite en `Ocaml` par Loïc Pottier. Son code faisait environ 60 lignes. Nous verrons que notre code, en utilisant le nouveau langage, s'est réduit à 10 lignes. Cette simplification d'écriture des tactiques est la principale raison d'être du nouveau langage de tactiques.

Ce langage de tactique, appelé $\mathcal{L}_{tac}$, étend considérablement et remplace l'ancien langage de tacticals. Il comprend le filtrage sur les termes, le filtrage sur le contexte de preuve, la récurrence, le backtrack, la manipulation de termes (écriture de fonctions manipulant des termes et rendant des termes), la gestion de métavariables, etc., et tout cela disponible au

[2]Cette normalisation comprend une passe de simplification.

toplevel[3] de Coq, c'est-à-dire au même titre que le langage de preuves. Les exemples qui suivent illustrent l'utilisation du langage.

## 4.4 Exemples simples

Dans cette section, nous détaillerons trois exemples simples de développement de tactiques d'automatisation qui nous aideront à résoudre quelques problèmes triviaux, au sens mathématique du terme. D'autre part, ces exemples, dont le code est court, nous permettront de mieux appréhender l'utilisation du langage de tactique en vue de décrire la tactique *Field*.

### 4.4.1 Exemple 1 : inégalité des constantes

Comme nous l'avons dit précédemment lors de la présentation des nombres réels (chapitre 1), les preuves de la forme $c_1 \neq c_2$ où $c_1$ et $c_2$ représentent des termes se ramenant à des constantes entières de type réel, sont à la fois fastidieuses à traiter sans automatisation et à la fois inintéressantes. De plus, ce genre de preuves est assez courant, en particulier dans les preuves concernant les dérivées. Cette tactique permet, par exemple, de montrer l'inégalité $2+3 \neq 3-4$.

Les constantes entières de type réel sont composées exclusivement de 1 et de 0, c'est-à-dire de `R1` et de `R0`. D'autre part, nous rappelons que ces constantes ont une représentation de la forme $1 + (... + (1 + 1))$. Une constante peu également être négative ou formée de soustractions ou de multiplications.

Après s'être ramené à une inégalité de la forme $c \neq 0$ avec $c$ une constante entière réelle, l'algorithme suit exactement la preuve présentée dans le chapitre 1. Soit l'inégalité $t_1 \neq t_2$ où $t_1, t_2$ sont des termes Coq. L'idée principale est la suivante :

- on vérifie que $t_1$ et $t_2$ sont des termes de type réel représentant des constantes entières
- on se ramène à l'inégalité $t_1 - t_2 \neq 0$
- on normalise $t_1 - t_2$ afin d'obtenir la constante $c$ cité précédemment
- on généralise la preuve de la page 24

Nous commençons donc par définir une tactique (`Isrealint`) qui prend un terme de type réel en paramètre et qui retourne l'identité (tactique `Idtac`) si le terme peu se ramener à une constante entière, et échoue (tactique `Fail`) sinon :

```
Recursive Tactic Definition Isrealint trm:=
   Match trm With
   | [``0``] -> Idtac
   | [``1``] -> Idtac
   | [``?1+?2``] -> (Isrealint ?1);(Isrealint ?2)
   | [``?1-?2``] -> (Isrealint ?1);(Isrealint ?2)
   | [``?1*?2``] -> (Isrealint ?1);(Isrealint ?2)
   | [``-?1``] -> (Isrealint ?1)
   | _ -> Fail.
```

[3] boucle intéractive

Il s'agit d'une tactique récursive. Comme une constante peut être formée d'additions, de soustractions ou de multiplications, lors du filtrage, l'appel récursif se fait sur les opérateurs `Rplus, Ropp, Rminus` et `Rmult`. Dans tous les autres cas, la tactique doit échouer. Cette tactique évitera, entre autres, le bouclage de la tactique proprement dite (`DiscrR`) en cas d'appel inapproprié de la part de l'utilisateur.

Le dernier point de l'algorithme, qui consiste en la généralisation de la preuve de la page 24, est fait dans la tactique suivante :

```
Recursive Tactic Definition Sup0 :=
   Match Context With
   | [ |- ``1>0`` ] -> Unfold Rgt;Apply Rlt_R0_R1
   | [ |- ``1+?1>0`` ] ->
     Apply (Rgt_trans ``1+?1`` ?1 ``0``);
     [Pattern 1 ``1+?1``;Rewrite Rplus_sym;Unfold Rgt;
      Apply Rlt_r_r_plus_R1|Sup0].
```

Pour cette fonction[4], nous supposons que nous avons exclusivement des inégalités de la forme $cp > 0$ où $cp$ représente une constante entière de type réel strictement positive. Cela sera effectivement le cas lorsque nous l'appellerons. Cette tactique permet donc de montrer la stricte positivité d'un nombre entier de type réel. Le premier cas de filtrage correspond au cas où $1 > 0$, cas où l'on se ramène lors du second cas de filtrage. En effet, $cp$ est ainsi composée : $1 + reste$. Par transitivité on a alors que $1 + reste > reste > 0$. Il reste à montrer que $1 + reste > reste$, ce qui est quasi immédiat en appliquant le lemme `Rlt_r_r_plus_R1`, et à rappeler récursivement `Sup0` pour montrer que $reste > 0$.

La tactique elle-même `DiscrR`[5] peut se coder ainsi :

```
Tactic Definition DiscrR :=
   Try Match Context With
   | [ |- ~(?1==?2) ] ->
     Isrealint ?1;Isrealint ?2;
     Apply Rminus_not_eq; Ring ``?1-?2``;
       (Match Context With
       | [ |- [``-1``] ] ->
         Repeat Rewrite <- Ropp_distr1;Apply Ropp_neq
       | _ -> Idtac);Apply Rgt_not_eq;Sup0.
```

On commence par transformer $c_1 \neq c_2$ en $c_1 - c_2 \neq 0$ avec `Rminus_not_eq`. Puis on normalise le terme obtenu avec `Ring`. Dans ce cas il s'agit d'obtenir la forme canonique d'une constante entière réelle, en particulier, l'associativité à droite. Le cas de filtrage sur le sous-terme ``` ``-1`` ``` est destiné à distribuer (`Ropp_distr1`) les $-$ dans un terme négatif. Ensuite seulement on peut transformer $-c \neq 0$ en $c \neq 0$ grâce à `Ropp_neq`. Puis, de manière générale, on fait ensuite apparaître le but $c > 0$ au lieu de $c \neq 0$ (car on sait que l'on a bien une constante strictement positive) et on appelle la tactique `Sup0` définie juste avant.

[4]une tactique peut aussi être vue comme une fonction manipulant un terme `Coq` ou un contexte et retournant un terme ou un contexte

[5]Nous avons nommé ainsi cette tactique en référence à la tactique `Discriminate` qui est utilisée pour montrer ce genre d'inégalités lorsqu'il s'agit de constructeurs d'un type inductif, ce qui est le cas, par exemple, pour le type `nat`.

**Exemple d'utilisation**

Montrons, par exemple, que $5 - 7 \times 2 \neq 3 + 4$ où ces nombres sont des nombres réels :

```
Coq < Goal ``5-7*2<>3+4``.
1 subgoal

  ============================
   ``5-7*2 <> 3+4``

Unnamed_thm < DiscrR.
Subtree proved!
```

**Remarque 4.4.1** *Sur une proposition d'algorithme de Laurent Théry, l'implantation de cette tactique a changé récemment. En effet une manière alternative et qui s'avère plus efficace consiste à passer par les entiers naturels de* Coq*. Le type de donné des entiers (*`nat`*) étant un type inductif, il suffit alors d'utiliser la tactique* `Discriminate` *[86].*
*On commence donc par définir une tactique qui transforme une constante entière de type réel (*R*) en une constante entière de type entier (*`nat`*) :*

```
Recursive Meta Definition ToINR trm:=
  Match trm With
  | [ ``1`` ] ->  '(S O)
  | [ ``1 + ?1`` ] -> Let t=(ToINR ?1) In '(S t).
```

*Nous pouvons maintenant réécrire la tactique* `DiscrR` *ainsi :*

```
Tactic Definition DiscrR :=
  Try Match Context With
  | [ |- ~(?1==?2) ] ->
    Isrealint ?1;Isrealint ?2;
    Apply Rminus_not_eq; Ring ``?1-?2``;
      (Match Context With
      | [ |- [``-1``] ] ->
        Repeat Rewrite <- Ropp_distr1;Apply Ropp_neq
      | _ -> Idtac);
      (Match Context With
      | [ |- ``?1<>0``] -> Let nbr=(ToINR ?1) In
        Replace ?1 with (INR nbr);
          [Apply not_O_INR;Discriminate|Simpl;Ring]).
```

*Tout comme précédemment, on commence se ramène à l'inégalité (sur les réels) $t_1 - t_2 \neq 0$ puis on normalise $t_1 - t_2$ (nommons ce nouveau terme $e$). On transforme $e$ en une constante $e_N$ entière grâce à* `ToINR` *et le lemme* `not_O_INR:(n:nat)~n=O->``(INR n)<>0``` *. On aura alors à montrer que $e =$ (*`INR` *$e_N$), ce qui est immédiat en utilisant* `Simpl ;Ring`*. Une fois ce remplacement effectué, il reste à prouver que $e_N \neq 0$ ce qui est immédiat par* `Discriminate`*.*

### 4.4.2 Exemple 2 : déstructuration d'un produit

La tactique suivante, très simplement définie, est extrêmement utile dés que nous voulons montrer qu'un produit est non nul. En particulier, comme nous le verrons plus en avant dans ce chapitre, la tactique `Field` génère ce genre de but. Or, il est fastidieux manuellement de montrer, par exemple, que $x \times y \times 3 \times (x+1) \neq 0$ sachant que $x > 1$ et $y \neq 0$ : Il faudrait, en particulier, commencer par destructurer le produit en appliquant trois fois les mêmes lemme (`mult_non_zero:(r1,r2:R)``r1<>0``/\``r2<>0``->``r1*r2<>0``)` et tactique (`Split`), afin d'obtenir les quatre buts distincts ($x \neq 0$, $y \neq 0$, $3 \neq 0$, et $x+1 \neq 0$), qui eux sont immédiats à montrer.

Après s'être assuré que le but est de la bonne forme, nous pouvons appliquer autant de fois qu'il le faut, ce qui a été dit précédemment :

```
Recursive Tactic Definition SplitRmult :=
  Match Context With
  | [ |- ~(Rmult ?1 ?2)==R0 ] ->
         Apply mult_non_zero; Split;Try SplitRmult.
```

**Exemple d'utilisation**

Reprenons l'exemple cité au paragraphe ci-dessus :

```
1 subgoal

  x : R
  y : R
  H : ``x > 1``
  H0 : ``y <> 0``
  ============================
   ``x*y*3*(x+1) <> 0``

Unnamed_thm < SplitRmult.
4 subgoals

  x : R
  y : R
  H : ``x > 1``
  H0 : ``y <> 0``
  ============================
   ``x <> 0``

subgoal 2 is:
 ``y <> 0``
subgoal 3 is:
 ``3 <> 0``
subgoal 4 is:
 ``x+1 <> 0``
```

### 4.4.3 Exemple 3 : traitement de la valeur absolue

Cette tactique consiste à faire disparaître la valeur absolue en mettant en évidence les deux cas qui permettent de la définir. Nous rappelons que la valeur absolue d'un réel a été définie au chapitre 1. Considérons, par exemple, cette propriété simple : si $|x| = 0$ alors $|x+1| = 1$. Une manière de faire la preuve, sans utiliser d'autres propriétés concernant la valeur absolue, consiste à remplacer les valeurs absolue par les quatre cas :

- si $x < 0$ et $x + 1 < 0$ alors on a $-x = 0$ et on doit montrer que $-(x + 1) = 0$
- si $x < 0$ et $x + 1 \geq 0$ alors on a $-x = 0$ et on doit montrer que $x + 1 = 0$)
- si $x \geq 0$ et $x + 1 < 0$ alors on a $x = 0$ et on doit montrer que $-(x + 1) = 0$
- si $x \geq 0$ et $x + 1 \geq 0$ alors on a $x = 0$ et on doit montrer que $x + 1 = 0$.

Chacun de ces cas se montre ensuite immédiatement. La tactique que nous présentons ci-dessous effectue cette séparation.

```
Recursive Tactic Definition SplitAbs :=
  Match Context With
    | [ |- [(case_Rabsolu ?1)] ] ->
         Case (case_Rabsolu ?1); Try SplitAbs.

Recursive Tactic Definition SplitAbsolu :=
  Match Context With
    | [ id:[(Rabsolu ?)] |- ? ] ->
          Generalize id; Clear id; Try SplitAbsolu
    | [ |- [(Rabsolu ?1)] ] ->  Unfold Rabsolu; Try SplitAbs;Intros.
```

C'est la tactique `SplitAbs` qui effectue cette séparation grâce à la tactique `Case`. `SplitAbsolu` parcourt le but et les hypothèses à la recherche de valeurs absolues, puis appelle `SplitAbs`. La tactique `Generalize` nous permet de ici de rassembler tous les termes contenant `Rabsolu` dans le but à montrer afin de ne faire appel qu'une seule fois à la tactique `Unfold`.

### Exemple d'utilisation

Reprenons l'exemple cité au paragraphe ci-dessus :

```
1 subgoal

  x : R
  H : ``(Rabsolu x) == 0``
  ============================
   ``(Rabsolu (x+1)) == 1``

Unnamed_thm < SplitAbsolu.
4 subgoals
subgoal 1 is:
  x : R
  r : ``x+1 < 0``
  r0 : ``x < 0``
  H : `` -x == 0``
  ============================
   `` -(x+1) == 1``

subgoal 2 is:
  x : R
  r : ``x+1 >= 0``
  r0 : ``x < 0``
  H : `` -x == 0``
  ============================
   ``x+1 == 1``
```

```
subgoal 3 is:
  x : R
  r : ``x+1 < 0``
  r0 : ``x >= 0``
  H : ``x == 0``
  ============================
   `` -(x+1) == 1``

subgoal 4 is:
  x : R
  r : ``x+1 >= 0``
  r0 : ``x >= 0``
  H : ``x == 0``
  ============================
   ``x+1 == 1``
```

## 4.5 La tactique *Field*

La tactique que nous allons présenter ici est un travail commun avec David Delahaye [25].
Nous nous proposons d'automatiser les preuves d'égalités en utilisant la théorie des corps commutatifs. L'idée de l'algorithme consiste à se débarrasser des inverses afin de pouvoir se ramener à l'utilisation de la procédure de décision déjà existante sur les anneaux abéliens (Ring). L'élimination des inverses se fait de manière complètement réflexive et la réflexion est réalisée au moyen du langage de tactiques dont nous avons parlé précédemment. Cette tactique pourra alors être instanciée pour une utilisation sur le corps des nombres réels.

Comme nous l'avons déjà souligné, les preuves se font à grand renfort de réécriture de par le fait que les réels sont axiomatiques dans Coq. La tactique Ring qui permet de décider d'égalités sur les anneaux abéliens a permis de résoudre partiellement ce problème et se révèle très utile dans les preuves ne faisant pas intervenir l'inverse.

On peut l'utiliser dans des théorèmes auxiliaires triviaux servant à compléter la théorie comme :

$$\forall x, y \in \mathbb{R}. x + y = 0 \to y = (-x)$$

où pas moins de cinq réécritures sont nécessaires pour résoudre sans la tactique Ring. Cependant, Ring ne règle pas le cas des égalités avec inverses qui, si elles ne sont pas difficiles à montrer, n'en restent pas moins très fastidieuses à résoudre et ralentissent tout développement. En effet, les théorèmes sur les limites et les dérivées font systématiquement intervenir des égalités avec des inverses dans les "preuves $\varepsilon$"[6] lorsque l'on veut composer, additionner, ... Ces égalités sont relativement triviales et alourdissent considérablement les preuves fondamentales de limites et de dérivées. Un exemple typique est celui de la dérivée de l'addition où l'on doit montrer que la somme des dérivées est égale à la dérivée de la somme (cf chapitre 2). Pour ce faire, on prend deux fonctions $f$ et $g$ ainsi que leurs dérivées en $x_0$, c'est-à-dire $f'(x_0)$ et $g'(x_0)$. Rappelons que, par définition, les deux dérivées peuvent s'exprimer comme suit :

[6] On appelle preuves $\varepsilon$, les preuves utilisant la définition explicite de la limite sous la forme : $\forall \varepsilon > 0. \exists \alpha$..... Dans la littérature anglaise, ces preuves sont plutôt connues sous la dénomination de preuves $\varepsilon/\delta$.

$$f'(x_0) = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0}$$

$$g'(x_0) = \lim_{x \to x_0} \frac{g(x) - g(x_0)}{x - x_0}$$

En utilisant le théorème d'addition des limites, on obtient directement :

$$f'(x_0) + g'(x_0) = \lim_{x \to x_0} \left( \frac{f(x) - f(x_0)}{x - x_0} + \frac{g(x) - g(x_0)}{x - x_0} \right)$$

En utilisant la définition de la limite, on a pour tout domaine $D$ de $\mathbb{R}$ :

$$\forall \varepsilon > 0,\ \exists \alpha > 0,\ \forall x \in D \backslash x_0,\ \text{si } |x - x_0| < \alpha \text{ alors}$$

$$\left| \frac{f(x) - f(x_0)}{x - x_0} + \frac{g(x) - g(x_0)}{x - x_0} - (f'(x_0) + g'(x_0)) \right| < \varepsilon$$

Maintenant, il suffit de montrer l'égalité suivante :

$$\frac{f(x) - f(x_0)}{x - x_0} + \frac{g(x) - g(x_0)}{x - x_0} = \frac{f(x) + g(x) - (f(x_0) + g(x_0))}{x - x_0}$$

pour conclure que $f'(x_0) + g'(x_0)$ et $(f + g)'(x_0)$ coïncident. Cette dernière égalité est clairement triviale mais la preuve formelle l'est beaucoup moins[7]. Il faut d'abord réduire au même dénominateur (4 réécritures) puis montrer l'égalité sur les numérateurs (6 réécritures). L'utilisation de `Ring` sur les numérateurs économise des réécritures et au total, cette égalité nécessite 4 réécritures + 1 tactique (`Ring`). Nous pensons que ce genre d'égalité doit être résolu directement par une tactique, à la fois pour un gain de temps considérable dans les développements, mais aussi de concision dans les scripts où l'on a plutôt envie de rendre implicites de telles preuves.

Nous présenterons, tout d'abord, l'algorithme que nous avons utilisé pour décider les égalités sur les corps commutatifs modulo certaines inégalités (conditions que les dénominateurs doivent être non nuls). Ensuite, nous donnerons une idée assez globale de l'implantation qui se fait de manière totalement réflexive. Enfin, nous nous appuierons sur plusieurs exemples, faciles pour certains et plus réalistes pour d'autres, afin de mettre en évidence la correction de la tactique ainsi que ses performances.

### 4.5.1 Algorithme

#### Principe

Comme nous l'avons dit, l'idée de l'algorithme est de minimiser les opérations de simplification, de manière à se brancher le plus tôt possible sur la procédure de décision sur les anneaux abéliens (`Ring`). Cela signifie qu'il faut éliminer tous les inverses intervenant dans l'égalité qu'il s'agit de résoudre. Pour ce faire, nous proposons la suite d'étapes suivante :

- Transformer les expressions $x - y$ en $x + (-y)$ et $x/y$ en $x * 1/y$.

[7] Rappelons qu'il ne s'agit pas ici d'entamer le leitmotiv bien connu que les preuves formelles sont bien plus difficiles que les preuves que l'on peut trouver dans les meilleurs ouvrages de mathématiques, mais de mettre en évidence une lacune d'automatisation qui, si elle venait à être comblée, pourrait nous permettre une meilleure granularité dans l'intéraction avec le système de preuve, dans le sens où l'utilisateur n'aurait plus à montrer, à la main, certaines propriétés considérées comme triviales.

- Chercher tous les inverses apparaissant dans l'égalité pour en faire un produit.
- Distribuer totalement à gauche et à droite de l'égalité, excepté dans les inverses.
- Associer à droite chaque monôme, excepté dans les inverses[8].
- Multiplier à gauche et à droite par le produit d'inverses, que l'on a construit précédemment, en générant la condition que tous les inverses doivent être non nuls.
- Distribuer seulement le produit sur la somme de monômes à gauche et à droite sans réassocier à droite.
- Éliminer les inverses des monômes en utilisant la règle de corps $x.1/x = 1,$ si $x \neq 0$ et en permutant les éléments du monôme si nécessaire, c'est-à-dire s'il reste des inverses et que la règle de corps ne peut pas s'appliquer[9].
- Recommencer le processus s'il reste encore des inverses.

La dernière étape, qui consiste à réitérer le processus, s'explique par le fait qu'il peut y avoir d'autres inverses dans les inverses, et ce, dans des expressions pouvant être compliquées. Pour éviter la réitération, une idée serait de se lancer dans une simplification directe en utilisant la règle $1/1/x = x,$ si $x \neq 0$. Toutefois, l'expérience a montré que le codage de cette simplification était plutôt complexe et générait un lemme de correction difficile. Par ailleurs, les expressions devant être différentes de 0 n'étaient pas exactement les mêmes que celles nécessaires à l'élimination des inverses dans les monômes. On pouvait certes les déduire, mais le lemme de correction correspondant à l'élimination des inverses devenait alors plus compliqué à montrer. On a donc tout avantage à se limiter à la règle $x.1/x = 1,$ si $x \neq 0$, qui tend à simplifier grandement l'algorithme et ce pour une perte d'efficacité négligeable en pratique[10].

Après ces étapes, nous obtenons une expression débarrassée de tous ses inverses et il suffit d'appeler `Ring` pour conclure.

**Exemple**

Considérons un petit exemple en détaillant les étapes de preuve afin de voir comment la procédure fonctionne ; étant donné $x$ et $y$, deux variables réelles, on se propose de montrer l'égalité suivante :

$$x \times (\frac{1}{x} + \frac{x}{x+y}) = (-\frac{1}{y}) \times y \times (-(\frac{x \times x}{x+y}) - 1)$$

On commence par transformer les moins binaires et les divisions :

$$x \times (\frac{1}{x} + x \times \frac{1}{x+y}) = (-\frac{1}{y}) \times y \times (-(x \times x) \times \frac{1}{x+y} + (-1))$$

On construit le produit d'inverses que l'on appellera $p$ :

$$p = x \times ((x+y) \times (y \times (x+y)))$$

On distribue totalement à gauche et à droite sauf dans les inverses :

[8] Cette étape est clairement non nécessaire mais elle permet un gain d'efficacité en évitant un double appel récursif pour toutes les fonctions manipulant ces expressions.

[9] Il n'est pas nécessaire ici de vérifier que $x \neq 0$ car la condition a déjà été générée lors de la multiplication par le produit de tous les inverses.

[10] On estime, en effet, que les expressions contenant des empilements d'inverses d'inverses seront plutôt rares.

$$x \times \frac{1}{x} + x \times x \times \frac{1}{x+y} = (-1) \times \frac{1}{y} \times y \times ((-1) \times (x \times x) \times \frac{1}{x+y}) + (-1) \times \frac{1}{y} \times y \times (-1)$$

On associe à droite chaque monôme sauf dans les inverses :

$$x \times \frac{1}{x} + x \times (x \times \frac{1}{x+y}) = (-1) \times (\frac{1}{y} \times (y \times ((-1) \times (x \times (x \times \frac{1}{x+y}))))) + (-1) \times (\frac{1}{y} \times (y \times (-1)))$$

On multiplie à gauche et à droite par $p$ en générant la condition de correction :

$$(x \times ((x+y) \times (y \times (x+y)))) \times (x \times \frac{1}{x} + x \times (x \times \frac{1}{x+y}))$$
$$=$$
$$(x \times ((x+y) \times (y \times (x+y)))) \times ((-1) \times (\frac{1}{y} \times (y \times ((-1) \times (x \times (x \times \frac{1}{x+y}))))) + (-1) \times (\frac{1}{y} \times (y \times (-1))))$$

Avec $x \times ((x+y) \times (y \times (x+y))) \neq 0$.
On distribue ce produit sur les monômes sans réassocier à droite :

$$(x \times ((x+y) \times (y \times (x+y)))) \times (x \times \frac{1}{x}) +$$
$$(x \times ((x+y) \times (y \times (x+y)))) \times (x \times (x \times \frac{1}{x+y}))$$
$$=$$
$$(x \times ((x+y) \times (y \times (x+y)))) \times (((-1) \times (\frac{1}{y} \times (y \times ((-1) \times (x \times (x \times \frac{1}{x+y})))))) +$$
$$(x \times ((x+y) \times (y \times (x+y)))) \times ((-1) \times (\frac{1}{y} \times (y \times (-1))))$$

On élimine les inverses en permutant si nécessaire :

$$((x+y) \times (y \times ((x+y)))) \times x + (x \times (y \times (x+y))) \times x \times x$$
$$=$$
$$(x \times (x+y)) \times ((-1) \times (y \times ((-1) \times (x \times x)))) + (x \times ((x+y) \times (x+y))) \times ((-1) \times (y \times (-1)))$$

On obtient alors une égalité sur les anneaux abéliens que `Ring` sait résoudre.

### Remarques

Nous n'avons pas formalisé la preuve que cet algorithme décide bien des égalités sur les nombres réels et sur les corps commutatifs plus généralement modulo certaines preuves d'inégalités. Il semble clair, cependant, qu'il est correct dans la mesure où l'on n'utilise que des axiomes de corps. On peut également se convaincre de la terminaison de la procédure puisque le nombre d'inverses décroît à chaque étape.

Par ailleurs, il est important de souligner que notre démarche ne vise pas à résoudre le problème global de décision sur les corps commutatifs dont on ne sait pas, *a priori*, s'il est décidable ou non. En effet, nous ne cherchons pas à prouver les conditions sur les inverses qui sont laissées à l'utilisateur.

Dans un souci de généralité, notre méthode traite tous les corps commutatifs. Si on avait voulu traiter seulement les nombres réels, notre approche aurait été bien différente et on aurait certainement opté pour des algorithmes résolvant au premier ordre tels que, entre autres, la méthode de Tarski[11] [85], l'algorithme de Kreisel-Krivine[12] [53] ou la décomposition cylindrique de Collins [14].

Toujours dans cette optique plus générale, on peut citer le travail du projet Fundamental Theorem of Algebra [39], avec, dans le cadre d'une axiomatisation constructive des nombres réels en Coq, le codage de la tactique réflexive `Rational` [39], traitant des égalités similaires à celles que nous nous proposons de résoudre. Notre approche se démarque essentiellement du fait de choix différents dans la formalisation des nombres réels. Tout d'abord, le fait que la fonction inverse soit totale permet de faire une réflexion totale des expressions de $\mathbb{R}$, ce qui n'est pas le cas dans `Rational` où tout inverse contient aussi la preuve que le dénominateur est non nul. La réflexion doit donc aussi être partielle, ce qui rend le processus plus complexe. Enfin, nous considérons l'égalité de Leibniz, ce qui permet à l'utilisateur d'appliquer des tactiques de réécriture à n'importe quel prédicat, alors que dans `Rational`, l'égalité est plus large (setoïde) et il est nécessaire de prouver des lemmes de compatibilité afin de passer au contexte.

### 4.5.2 Implantation

Comme nous l'avons dit précédemment, l'implantation de cette procédure de décision sur les nombres réels, que nous avons appelée `Field`, a été réalisée dans la version V7 de Coq afin de pouvoir profiter des nouvelles possibilités du langage de tactiques.

#### À propos de la réflexion

Pour coder `Field`, il y a globalement deux choix possibles. Un codage explicite en utilisant la réécriture ou un codage par réflexion en utilisant la réduction. Le codage explicite (approche à la LCF) est très coûteux du fait de l'utilisation de la réécriture, qui prend du temps, mais aussi et surtout de la place dans le terme preuve[13]. Le codage par réflexion est une alternative complètement satisfaisante, pour laquelle nous avons opté. En effet, les réécritures sont remplacées par des phases de réduction plus efficaces et la taille du terme preuve est de l'ordre de celle du but à résoudre. Par ailleurs, on peut formaliser clairement la correction globale de la tactique ainsi que sa complétude (même si nous ne l'avons pas fait ici), alors que dans l'approche explicite, c'est bien plus difficile, voire impossible.

Avant de donner une idée de l'implantation de `Field`, rappelons rapidement le principe d'une tactique codée par réflexion. Soit un langage $C$ des termes concrets (typiquement un type quelconque) et un langage $A$ des termes abstraits (typiquement un type inductif). Comme on ne peut pas manipuler les termes du langage $C$ comme on voudrait (on ne peut pas filtrer), l'idée est de le réfléchir dans le langage $A$ qui lui est isomorphe. Une première phase, appelée métaification par Samuel Boutin [10], consiste donc à traduire les termes de $C$ vers les termes de $A$. Plus précisément, cela consiste, pour un terme $c$ de $C$, à trouver le terme $a$ de $A$ tel que $(f\ v\ a) = c$, où $f$ est la fonction d'interprétation de $A$ vers $C$ (codable dans Coq), $v$ est une liste d'associations contenant les parties de $C$ que

[11]Cet algorithme ne fonctionne qu'en logique classique [37] mais ce n'est pas gênant dans la mesure où il en est de même pour les nombres réels en Coq à cause de l'axiome d'ordre total.

[12]Kreisel et Krivine se sont aussi intéressés à un algorithme dans d'autres structures telles que les corps algébriquement clos, les anneaux de Boole séparables, ...

[13]La taille du terme preuve est un point auquel il faut être très sensible car il n'est pas rare de rencontrer des scripts de preuves corrects pour lesquels on ne peut pas construire le terme preuve, faute de mémoire suffisante.

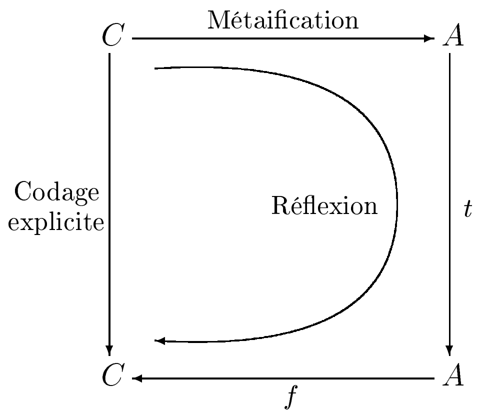
l'on ne réfléchit pas (atomes) et = est l'égalité de Leibniz. On peut se passer de la liste d'associations à condition que l'égalité sur les atomes soit décidable, car on a généralement besoin de comparer les termes de $A$. La métaification s'assimile exactement à une phase d'analyse syntaxique comme on pourrait la trouver dans un langage de programmation.

Ensuite, on peut coder la fonction $t$ de transformation des termes de $A$. Pour l'utiliser, il suffit de prouver un lemme de correction[14] de la forme :

$$\forall a \in A.(f\ v\ (t\ a)) = (f\ v\ a)$$

Enfin, après avoir appliqué ce lemme de correction, il suffit de réduire totalement (`Compute`) pour transformer le terme abstrait (de $A$) et revenir à un terme concret (de $C$). Pour pousser l'analogie avec les langages de programmation, on pourrait voir ces deux étapes comme une phase d'évaluation suivie d'une phase de "pretty-print".

On peut résumer la situation au moyen du schéma suivant :




Pour une description complète de la réflexion, on pourra se reporter à [10] et [46].

### Codage de la tactique

Nous allons maintenant entrer dans les détails de l'implantation de l'algorithme donné précédemment. Nous utiliserons une syntaxe spécifique au langage `Coq`.

#### Paramétrisation

La tactique `Field` doit pouvoir fonctionner pour tout corps commutatif. Elle est donc paramétrée par une théorie de corps commutatif, c'est-à-dire un ensemble et des axiomes sur cet ensemble lui donnant une structure de corps commutatif.

Pour effectuer cette paramétrisation, nous avons choisi d'utiliser une structure d'enregistrement (`Record` en `Coq`), qui contient à la fois l'ensemble et ses axiomes. Nous définissons ainsi un type corps commutatif, appelé `Field_Theory`, de la manière suivante :

```
Record Field_Theory : Type :=
  { A : Type;
    Aplus : A -> A -> A;
```

[14] Il est intéressant de voir ici que, dans le processus de réflexion, la tactique et la preuve de sa correction sont indissociables.

```
    Amult : A -> A -> A;
    Aone : A;
    Azero : A;
    Aopp : A -> A;
    Aeq : A -> A -> bool;
    Ainv : A -> A;
    Aminus : (option A);
    Adiv : (option A);
    RT : (Ring_Theory Aplus Amult Aone Azero Aopp Aeq);
    Th_inv_def : (n:A)~(n==Azero)->(Amult (Ainv n) n)==Aone
  }.
```

Afin de pouvoir instancier par tout corps commutatif, l'ensemble représentant le corps est de type `Type`, ce qui permettra l'instanciation de cette tactique par un corps spécifié dans la sorte `Set`, de par la cumulativité (voir annexe D). Pour ce qui est des opérateurs, les champs correspondant à la soustraction et à la division sont optionnels, car ces constantes s'expriment respectivement en fonction du plus et de l'opposé et en fonction de la multiplication et de l'inverse. Le champs `RT` représente la structure d'anneau au moyen de `Ring_Theory`[15], qui lui aussi est un enregistrement et qui contient, entre autres, tous les axiomes d'anneau. Enfin, on définit l'axiome de simplification de l'inverse, qui permet de passer d'une structure d'anneau à une structure de corps.

### Structure de réflexion

Dans le cas de `Field`, les termes concrets sont exactement les expressions du corps (pour les réels par exemple, des expressions de type $\mathbb{R}$). Les termes abstraits doivent *refléter* les opérateurs qui permettent de donner une structure de corps commutatif et se définissent très simplement par un type inductif que nous avons appelé `ExprA` :

```
Inductive ExprA : Set :=
  | EAzero : ExprA
  | EAone  : ExprA
  | EAplus : ExprA -> ExprA -> ExprA
  | EAmult : ExprA -> ExprA -> ExprA
  | EAopp  : ExprA -> ExprA
  | EAinv  : ExprA -> ExprA
  | EAvar  : nat -> ExprA.
```

Les variables sont des expressions du corps pour lesquelles on ne peut pas, de fait, décider l'égalité de manière générale. On les remplace donc par des indices entiers (`nat`) pour pouvoir décider l'égalité entre variables et on associe à un terme une liste d'associations entre les indices et les expressions du corps.

### Métaification

Pour traduire les expressions du corps vers `ExprA` (métaification), il faut utiliser le métalangage de Coq. Comme nous l'avons dit précédemment, jusqu'à la V7 exclue, le seul moyen était de coder cette traduction dans Ocaml [58] en utilisant un fichier ML que l'on compilait

[15]`Ring_Theory` fait partie de la formalisation de la tactique `Ring`.

avec le système et que l'on pouvait importer dans un toplevel bytecode de Coq. Ce protocole était un peu lourd à mettre en œuvre[16], d'autant que le processus de métaification est extrêmement simple. Dans la V7, le langage de tactiques permet de se libérer de ce genre de contraintes au moyen d'un noyau fonctionnel et d'opérateurs de filtrage élaborés. Par ailleurs, un autre atout intéressant est, qu'étant intégré au toplevel de Coq, il est possible de faire tourner le code au moyen d'un toplevel compilé en natif, sans perdre la modularité[17] du système.

Dans `Field`, pour métaifier, on utilise d'abord une tactique appelée `BuildVarList`, qui construit la liste d'associations des expressions du corps non reflété avec leurs indices entiers, telles que deux expressions égales (au sens de l'égalité de Leibniz) aient le même indice. Cette liste est ensuite passée à la fonction d'interprétation du corps dans `ExprA`, définie de la manière suivante :

```
Recursive Tactic Definition interp_A FT lvar trm :=
  Let AT     = Eval Compute in (A FT)
  And AzeroT = Eval Compute in (Azero FT)
  And AoneT  = Eval Compute in (Aone FT)
  And AplusT = Eval Compute in (Aplus FT)
  And AmultT = Eval Compute in (Amult FT)
  And AoppT  = Eval Compute in (Aopp FT)
  And AinvT  = Eval Compute in (Ainv FT) In
  Match trm With
  | [(AzeroT)] -> EAzero
  | [(AoneT)] -> EAone
  | [(AplusT ?1 ?2)] ->
    Let e1 = (interp_A FT lvar ?1)
    And e2 = (interp_A FT lvar ?2) In
    '(EAplus e1 e2)
  | [(AmultT ?1 ?2)] ->
    Let e1 = (interp_A FT lvar ?1)
    And e2 = (interp_A FT lvar ?2) In
    '(EAmult e1 e2)
  | [(AoppT ?1)] ->
    Let e = (interp_A FT lvar ?1) In
    '(EAopp e)
  | [(AinvT ?1)] ->
    Let e = (interp_A FT lvar ?1) In
    '(EAinv e)
  | [?1] ->
    Let idx = (Assoc ?1 lvar) In
    '(EAvar idx).
```

où `Assoc` est une tactique qui donne l'indice correspondant à l'expression du corps commutatif dans la liste d'associations (`lvar`). Le paramètre `FT` est un corps commutatif (de type `Field_Theory`). De la deuxième ligne à la huitième, on extrait, de `FT`, l'ensemble ainsi

[16] En effet, il faut se procurer Ocaml, Camlp4, compiler les sources de Coq, coder la métaification en comprenant la structure abstraite des termes Coq et enfin compiler le fichier en question.

[17] En effet, le chargement dynamique de fichiers bytecode dans un exécutable natif n'est pas encore très standard et le chargement dynamique de fichiers natifs dans du natif n'est, quant à lui, clairement pas compris. La commande `coqmktop` permet de créer un toplevel "customisé" éventuellement en natif en indiquant une liste de fichiers à inclure au moment de l'édition de liens. Toutefois, le processus est purement statique et on perd la possibilité d'appeler d'autres tactiques qui n'ont pas été "linkées" au moment du `coqmktop`.

que ses opérateurs[18], excepté le moins binaire et la division (champs `Aminus` et `Adiv`). En effet, on suppose, à ce niveau là, que les éventuelles constantes correspondant au moins binaire et à la division ont déjà été expansées (dans une passe préalable). Enfin, on reproduit, dans `ExprA`, la structure de l'expression du corps commutatif et, pour les variables, il suffit de rechercher l'indice associé à l'expression dans la liste produite par `BuildVarList`.

#### Construction du multiplicateur

Ici, il s'agit de construire un produit de facteurs sans doublons inutiles (doublons qui apparaîtront entre monômes après distribution), constitué des inverses de l'égalité (on ne prend pas en compte les inverses dans les inverses, qui seront traités dans d'autres passes de `Field`). Pour ce faire, on a le choix entre utiliser une fonction de `Coq` ou une tactique que l'on peut plus facilement écrire dans la V7. Pour des raisons d'aisance de programmation, nous avons opté pour une tactique, puisque l'on est moins limité, entre autres, dans la récursivité. Le multiplicateur nous est donc donné par la tactique `GiveMult` utilisant la tactique `RawGiveMult` qui donne la liste des inverses :

```
Recursive Tactic Definition RawGiveMult trm :=
  Match trm With
  | [(EAinv ?1)] -> '(cons ExprA ?1 (nil ExprA))
  | [(EAopp ?1)] -> (RawGiveMult ?1)
  | [(EAplus ?1 ?2)] ->
    Let l1 = (RawGiveMult ?1)
    And l2 = (RawGiveMult ?2) In
    (Union l1 l2)
  | [(EAmult ?1 ?2)] ->
    Let l1 = (RawGiveMult ?1)
    And l2 = (RawGiveMult ?2) In
    Eval Compute in (app ExprA l1 l2)
  | _ -> '(nil ExprA).

Tactic Definition GiveMult trm :=
  Let ltrm = (RawGiveMult trm) In
  '(mult_of_list ltrm).
```

où `nil`, `cons` et `app` sont les constructeurs et la concaténation des listes polymorphes. `Union` est une tactique qui concatène deux listes en éliminant les doublons (éléments communs aux deux listes). `mult_of_list` est une fonction `Coq` qui rend un produit associé à droite à partir d'une liste d'expressions de `ExprA`. Le `Eval Compute in` utilisé dans `RawGiveMult` permet de réduire les `app` pour obtenir une liste canonique pouvant être correctement filtrée (par `mult_of_list`).

#### Distributivité et associativité

[18] En `Coq`, les types enregistrements ne sont pas primitifs et utilisent les types inductifs. Un type enregistrement déclare ainsi un type inductif à un seul constructeur contenant tous les champs de l'enregistrement. De plus, les labels des champs sont déclarés comme des constantes fonctionnelles prenant en argument un objet du type inductif déclaré et qui le destructurent pour obtenir les contenus des champs correspondants. Ainsi, accéder à un champ n'est pas immédiat et doit passer par une phase de réduction (d'où la série de `Eval Compute in` sur `FT`).

Il s'agit maintenant de distribuer totalement (sauf dans les inverses) et d'associer à droite (par rapport à l'addition et à la multiplication) dans les membres de l'égalité. Ces deux fonctions se font obligatoirement dans `Coq` car l'idée est de passer du terme initial au terme transformé via la réduction de `Coq` et un lemme de correction à prouver pour chaque fonction.

La distributivité ne peut pas être codée directement et facilement dans `Coq`. En effet, les conditions de garde assurant la normalisation forte obligent à découper le problème de manière à faire des appels récursifs respectant la décroissance de la mesure (ordre sous-terme). Les fonctions peuvent donc sembler un peu compliquées mais il s'agit surtout de respecter ces conditions syntaxiques. Pour distribuer, l'idée est de distribuer d'abord tous les moins unaires `EAopp`. Ensuite, on distribue totalement dans les sous-termes et, pour le cas de la multiplication `EAmult`, on distribue d'abord à gauche puis à droite. Nous ne donnerons pas ici les fonctions en question qui ne présentent pas un intérêt particulier. Le lecteur intéressé pourra se reporter au code source. Pour utiliser la fonction de distributivité `distrib`, on a prouvé le lemme de correction suivant :

```
Lemma distrib_correct:
  (T:Field_Theory; e:ExprA; lvar:(list (Sprod (A T) nat)))
    (interp_ExprA T lvar (distrib e))==(interp_ExprA T lvar e).
```

où `T` est le corps commutatif initial (avec ses axiomes), `interp_ExprA` est la fonction d'interprétation de `ExprA` vers le corps (écrite en `Coq`), `lvar` la liste d'associations des variables.

L'associativité ne pose pas de problèmes dans son codage moyennant quelques précautions. De même que pour `distrib`, nous ne donnerons pas le code de la fonction d'associativité, nommée `assoc`, et il nous a fallu prouver le lemme de correction suivant :

```
Lemma assoc_correct:
  (T:Field_Theory; e:ExprA; lvar:(list (Sprod (A T) nat)))
    (interp_ExprA T lvar (assoc e))==(interp_ExprA T lvar e)
```

Après avoir appliqué `distrib` et `assoc`, on obtient à gauche et à droite de l'égalité à prouver, deux termes qui sont des sommes de monômes associés à droite.

### Multiplier les membres de l'égalité

Pour pouvoir effectuer la simplification des inverses, on multiplie ensuite par le multiplicateur qui a été construit précédemment (produit de tous les inverses excepté ceux qui sont dans d'autres inverses). Pour ce faire, il suffit de montrer le lemme suivant sur les expressions de `ExprA` et qui a directement son équivalent dans le corps reflété :

```
Lemma mult_eq:
  (T:Field_Theory; e1,e2,a:ExprA; lvar:(list (Sprod (A T) nat)))
    ~(interp_ExprA T lvar a)==(Azero T)->
      (interp_ExprA T lvar (EAmult a e1))==
      (interp_ExprA T lvar (EAmult a e2))->
        (interp_ExprA T lvar e1)==(interp_ExprA T lvar e2).
```

Ce lemme permet de générer le but (que l'utilisateur devra prouver) que le multiplicateur doit être non nul. Ceci équivaut à dire que tous ses facteurs doivent être non nuls, ce qui est cohérent dans la mesure où l'on est sûr de devoir simplifier ces expressions.

Une fois le produit effectué, on distribue ce produit sur les monômes, sans réassocier à droite, ce qui est trivialement fait par une fonction `Coq` appelée `multiply` dont le lemme de correction est complètement similaire à ceux donnés précédemment pour la distributivité et l'associativité.

### Élimination des inverses

L'élimination des inverses se fait monôme par monôme. À ce stade, un monôme est un produit du multiplicateur et d'une expression qui est un produit associé à droite (monôme obtenu après la phase de distributivité et d'associativité). L'idée est donc de parcourir le produit du multiplicateur et de repérer les inverses correspondants dans le reste du monôme pour les simplifier. Pour assurer la correction de la simplification, on va paramétrer par un produit de facteurs (supposé non nul) et, avant de simplifier, on vérifiera que ce produit est égal au multiplicateur.

Concrètement, le travail est effectué par les fonctions Coq suivantes :

```
Definition monom_simplif [a,m:ExprA] : ExprA :=
  Cases m of
  | (EAmult a' m') =>
    (Cases (eqExprA a a') of
     | (left _) => (monom_simplif_rem a m')
     | (right _) => m
     end)
  | _ => m
  end.

Fixpoint inverse_simplif [a,e:ExprA] : ExprA :=
  Cases e of
  | (EAplus e1 e2) =>
      (EAplus (monom_simplif a e1) (inverse_simplif a e2))
  | _ => (monom_simplif a e)
  end.
```

où `monom_simplif_rem`, prend deux produits de facteurs en arguments et parcourt le premier pour supprimer d'éventuels inverses dans le second.

On remarque que, comme prévu, `inverse_simplif`, simplifie monôme par monôme et que `monom_simplif`, vérifie bien que le premier produit d'un monôme correspond au multiplicateur (passé en paramètre).

Le lemme de correction d'élimination des inverses s'exprime comme suit :

```
Lemma inverse_correct:
  (T:Field_Theory; e,a:ExprA; lvar:(list (Sprod (A T) nat)))
    ~(interp_ExprA T lvar a)==(Azero T)->
 (interp_ExprA T lvar (inverse_simplif a e))==(interp_ExprA T lvar e).
```

Ce lemme est paramétré par un produit de facteurs (paramètre `a`) que l'on suppose non nul, de manière à ce que les éventuelles simplifications réalisées par `inverse_simplif` soient correctes.

### La tactique globale

La tactique `Field` combine toutes les phases que nous venons de voir. Elle s'exprime directement dans le langage de tactiques de la V7 comme suit :

```
Tactic Definition Field_Gen FT :=
  Let AplusT = Eval Compute in (Aplus FT) In
  Unfolds FT;
  Match Context With
  | [|- ?1==?2] ->
    Let lvar = (BuildVarList FT '(AplusT ?1 ?2)) In
    Let trm1 = (interp_A FT lvar ?1)
    And trm2 = (interp_A FT lvar ?2) In
    Let mul = (GiveMult '(EAplus trm1 trm2)) In
    Cut [ft:=FT][vm:=lvar]
         (interp_ExprA ft vm trm1)==(interp_ExprA ft vm trm2);
    [Compute;Auto
    |Intros;(ApplySimplif ApplyDistrib);(ApplySimplif ApplyAssoc);
     (Multiply mul);[(ApplySimplif ApplyMultiply);
       (ApplySimplif (ApplyInverse mul));
       (Let id = GrepMult In Clear id);Compute;
       First [(InverseTest FT);Ring|(Field_Gen FT)]|Idtac]].
```

La première ligne de `Field`[19] (série d'`Unfold`) permet de déplier les moins binaires et les divisions, s'ils ont été fournis dans le corps commutatif `FT`. Ensuite, on crée la liste d'associations des variables (dans `lvar`) avec `BuildVarList` pour interpréter les deux membres de l'égalité, ce qui donne deux termes `trm1` et `trm2` de `ExprA`. Le multiplicateur `mul` est donné par `GiveMult` en prenant soin de sommer les deux termes pour tenir compte des inverses des deux membres de l'égalité. Le `Cut` permet d'insérer les termes de `ExprA` dans le but à prouver. Par la suite, on peut leur appliquer les différentes transformations dont nous avons parlé précédemment au moyen de tactiques. `ApplyDistrib` applique la distributivité, `ApplyAssoc` l'associativité, `Multiply` la multiplication par le multiplicateur à gauche et à droite de l'égalité, `ApplyMultiply` la distribution du multiplicateur et `ApplyInverse` l'élimination des inverses. `ApplySimplif` permet d'appliquer la tactique à gauche et à droite de l'égalité pour les tactiques qui travaillent sur un terme. On se débarrasse de l'hypothèse que l'interprétation du multiplicateur doit être non nulle au moyen de `GrepMult`, qui rend le nom de cette hypothèse pouvant être ainsi effacées (`Clear`). Enfin, on teste s'il reste encore des inverses dans les termes de l'égalité grâce à la tactique `InverseTest` qui, soit ne fait rien s'il ne reste pas d'inverses permettant ainsi l'appel à `Ring`, soit échoue dans le cas contraire impliquant une nouvelle application de *Field*.

### Instanciation de la tactique

Nous avons opté pour une méthode similaire à celle utilisée pour la tactique `Ring`. Ainsi, on utilise une table de hash stockant les différentes théories avec, comme clé, l'ensemble support. Il n'y a alors plus qu'une seule tactique, appelée `Field`, qui, de l'égalité à résoudre, devine l'ensemble support, puis récupère la théorie correspondante dans la table pour enfin utiliser `Field_Gen` avec. Cependant, tout ceci ne peut pas se faire à toplevel et ce, essentiellement à cause de la table, qui est un trait impératif, de fait, non disponible dans le langage de Coq. On est donc contraint, pour cette partie, de la coder en Ocaml, mais le code en question est plutôt petit, ce qui le rend finalement assez peu significatif sur le développement global.

[19] nommée `Field_Gen` car elle est ici générale et non encore instanciée par un corps particulier, ce qui laissera la possibilité d'utiliser l'identificateur `Field` pour l'instanciation de la tactique appliquée aux corps des nombres réels.

Ainsi, l'instanciation se fait par la commande toplevel suivante :

```
Add Field B Bplus Bmult Bone Bzero Bopp Beq Binv Rth Tinvl.
```

où B est l'ensemble support, Bplus, Bmult, Bone, Bzero, Bopp, Binv, sont ses opérateurs, Beq est une égalité de Leibniz semi-décidable à valeurs dans bool utilisée par Ring, Rth contient les axiomes d'anneau de B (voir le type Ring_Theory) et Tinvl est l'axiome de corps sur l'inverse.

En option, on peut éventuellement préciser le moins binaire ou la division comme suit :

```
Add Field B Bplus Bmult Bone Bzero Bopp Beq Binv Rth Tinvl
         with minus:=Bminus div:=Bdiv.
```

où Bminus et Bdiv sont les constantes représentant respectivement le moins binaire et la division.

Ainsi, si nous prenons l'exemple de l'instanciation pour les nombres réels, nous avons :

```
Add Field R Rplus Rmult R1 R0 Ropp [x,y:R]false Rinv RTheory Rinv_l
         with minus:=Rminus div:=Rdiv.
```

avec

```
Lemma RTheory : (Ring_Theory Rplus Rmult R1 R0 Ropp [x,y:R]false).
```

Cette commande a pour effet de construire la théorie du corps commutatif correspondante (de type Field_Theory) et de la stocker dans une table avec R comme clé. S'il n'y a pas de théorie d'anneau déclarée (commande Add Ring), Add Field la déclare, ce qui permet d'utiliser Ring ensuite[20]. Il suffit alors d'appeler simplement Field, qui, comme on l'a dit précédemment, se charge de récupérer, à partir de l'égalité à résoudre, le R correspondant et peut ensuite retrouver la théorie liée à cette clé pour appeler Field_Gen.

### 4.5.3 Exemples

Nous donnons ici quelques exemples tirés de preuves faisant partie du développement[21] des nombres réels[22].

Dans les exemples qui suivent, certains des buts auxquels nous allons appliquer la tactique ne sont pas montrables tels qu'ils sont présentés car nous n'avons pas exprimé toutes les hypothèses nécessaires. En effet, ces exemples visent essentiellement à montrer l'utilisation de Field et non à montrer les buts générés.

#### Exemple 1

L'exemple que nous donnons ici appartient à une famille d'égalités que nous pouvons qualifier de simples. Néanmoins, ce type d'égalités revient assez souvent dans les preuves et l'accumulation de toutes ces petites preuves devient rapidement fastidieuse et finit par produire un terme preuve plus important qu'il ne devrait. Pour cette raison, il est intéressant de pouvoir utiliser Field fréquemment, tout comme Ring, afin de minimiser le nombre de

[20] En effet, un corps commutatif est aussi un anneau.

[21] Plus précisément, deux fichiers sont principalement concernés : Rlimit.v et Rderiv.v.

[22] Cette tactique n'ayant été développée que très récemment, elle n'est, pour le moment, utilisée que peu souvent dans les preuves. Les preuves seront modifiées de manière à utiliser cette tactique, ce qui réduira la taille des termes preuves.

réécritures.

Nous considérons, par exemple, les parties de preuves présentes de manière récurrente telles que : $b = \frac{b}{a} \times a$.

```
Coq < Goal (a,b:R)``b == b*(1/a)*a``.
...

Unnamed_thm < Field.
1 subgoal

  a : R
  b : R
  ============================
   ``a <> 0``
```

Sans aucune réécriture, il ne nous reste plus qu'à prouver que $a \neq 0$, propriété que nous devions impérativement prouver même dans le cas où nous procédions par réécritures.

### Exemple 2

Nous voulons prouver que $\frac{\varepsilon}{2+2} + \frac{\varepsilon}{2+2} = \frac{\varepsilon}{2}$. Cette opération est utilisée dans la preuve concernant la multiplication des limites (`limit_mul`). La preuve du but énoncé ci-dessous, sans la tactique, fait environ 25 lignes de `Coq`, alors qu'après l'application de `Field` il nous reste à prouver uniquement que $(2+2)*2$ est non nuls[23].

```
Coq < Goal (eps:R)``eps*1/(2+2)+eps*1/(2+2) == eps*1/2``.
...

Unnamed_thm < Field.
1 subgoal

  eps : R
  ============================
   ``(2+2)*2 <> 0``
```

Comme dit précédemment, pour pouvoir simplifier par $2+2$ et par $2$, nous utilisons le fait que $2+2 \neq 0$ et $2 \neq 0$. Ces deux conditions sont générées sous la forme d'un unique sous-but traduisant ces deux propriétés : $(2+2) \times 2 \neq 0$. En effet, si un produit est non nul alors chacun de ses facteurs est également non nul.

### Exemple 3

Revenons sur l'exemple cité en introduction, tiré de la preuve concernant l'addition des dérivées :

```
Coq < Goal (f,g:(R->R); x0,x1:R)
    ``((f x1)-(f x0))/(x1-x0)+((g x1)-(g x0))/(x1-x0) ==
      ((f x1)+(g x1)-((f x0)+(g x0)))/(x1-x0)``.
...

Unnamed_thm < Field.
```

[23] Ce qui, rappelons-le, se fait immédiatement avec `DiscrR`.

```
1 subgoal

  f : R->R
  g : R->R
  x0 : R
  x1 : R
  ============================
   ``x1+-x0 <> 0``
```

Il ne reste plus qu'à prouver que $x1 - x0 \neq 0$, ce qui est une hypothèse de notre lemme d'addition des dérivées.

**Exemple 4**

Nous nous intéressons ici aux preuves concernant les séries entières, utilisées pour définir les fonctions transcendantes telles que $\exp$, $\sin$ ou $\cos$.

Considérons, par exemple, l'application du critère de d'Alembert à la fonction exponentielle. Nous pourrons nous référer au chapitre 2 pour les détails. Il nous faut montrer que

$$\left|\frac{\frac{1}{(n+1)!}}{\frac{1}{n!}}\right| \xrightarrow[n\,\to\,+\infty]{} 0$$

Dans ce but, nous pouvons montrer l'égalité suivante et l'appliquer ultérieurement dans la preuve avec $a = (S\ n)$ et $b = n!$ :

$$\frac{\frac{1}{a.b}}{\frac{1}{b}} = \frac{1}{a}$$

```
Coq < Goal (a,b:R)``a <> 0``->``b <> 0``->``1/(a*b)/(1/b) == 1/a``.
...

Unnamed_thm < Field.
2 subgoals

  a : R
  b : R
  H : ``a <> 0``
  H0 : ``b <> 0``
  ============================
   ``b <> 0``

subgoal 2 is:
 ``a*b*(1/b*a) <> 0``
```

Nous remarquons la génération de deux nouveaux sous-buts. Conformément à l'algorithme utilisé, la première passe génère le sous-but 2 tandis que la seconde génère le sous-but 1. En effet, la première passe commence par multiplier par $a.b \times \frac{1}{b} \times a$ et, après simplifications, il reste alors un $\frac{1}{b}$. La seconde passe multiplie donc par $b$. Cela équivaut alors à montrer $a.b \neq 0$, $\frac{1}{b} \neq 0$, $a \neq 0$ et $b \neq 0$, ce qui se déduit des hypothèses.

**Exemple 5**

Enfin, nous pouvons donner l'exemple de la section 4.5.1, qui n'a pas de sens particulier mais qui est un bon test pour `Field` :

```
Coq < Goal (x,y:R)``x*(1/x+x/(x+y)) == -1/y*y*(-(x*x)/(x+y)-1)``.
...

Unnamed_thm < Field.
1 subgoal

  x : R
  y : R
  ============================
   ``x*((x+y)*y) <> 0``
```

### 4.5.4 Discussion

La tactique `Field` contribue grandement au développement de la théorie des nombres réels en Coq. Elle permet une économie de temps précieux ainsi qu'un gain non négligeable de concision dans les scripts de preuves. Par ailleurs, étant intégralement réflexive, elle permet la construction de termes preuves plus petits que dans l'approche directe en utilisant les réécritures. L'utilisateur peut maintenant se désintéresser de certaines parties de preuves comme il le ferait dans une preuve informelle. Cette tactique a, par ailleurs, servi de base à l'implantation d'une *strategy* en PVS, nommée `FIELD`, développée conjointement avec César Muñoz au centre de recherche ICASE Nasa-Langley [67].

Une extension prévue de cette tactique en Coq serait d'avoir une option similaire à celle de `Ring` où l'on peut lui donner un terme, lequel est normalisé et remplacé dans le but courant. On pourrait faire de même avec `Field` qui simplifierait tous les inverses du terme avant de le remplacer dans le but courant. Étant donné une égalité, on est sûr de pouvoir simplifier tous les inverses, mais, pour un terme, il se peut que des inverses ne se simplifient pas et le test d'arrêt de `Field` (lorsqu'il n'y a plus d'inverses) devra être différent dans ce cas.

Comme nous l'avons vu, l'automatisation de la librairie permet de traiter les égalités et quelques autres propriétés simples sur les constantes ou la valeur absolue. D'autres tactiques ont également été implantées par des utilisateurs de Coq. Nous pouvons citer, par exemple, la tactique `Fourier`, implantée par Loïc pottier. Elle résout les inégalités linéaires en utilisant l'algorithme de Fourier [35]. Une prochaine étape serait de développer une tactique plus générale d'élimination des quantificateurs (non générale cette fois) se rapprochant de celle existant en HOL.

# Chapitre 5

# Différentiation Automatique

De nombreux exemples existent, qui sont en mesure d'illustrer la nécessité de programmes prouvés corrects, dans des secteurs variés de la recherche et de l'industrie. Depuis quelques années, certains secteurs, dans lesquels on utilise des programmes critiques, commencent à s'intéresser aux techniques issues des méthodes formelles. Néanmoins, la complexité des programmes numériques rend souvent difficile l'utilisation des systèmes de preuves formelles, ces derniers n'étant pas encore considérés comme suffisamment efficaces et développés pour ces problématiques. Le but de ce chapitre est, à partir d'un besoin pratique particulier, d'étudier la faisabilité d'une utilisation des systèmes de preuves afin d'obtenir des programmes numériques zéro défaut.

L'exemple que nous avons choisi nous a été proposé par des chercheurs des projets Ondes et Estime de l'INRIA. Ces projets écrivent, en particuliers, des programmes effectuant des calculs de gradient [13]. Ces programmes ne présentent pas de difficultés théoriques pour leur concepteur, mais sont importants en taille et leur phase de recherche d'erreur s'effectuant par des méthodes de test est très longue et, de surcroît, non fiables, puisqu'il s'agit de tests, non exhaustifs, et non de preuves. Pour pallier ce problème, des *systèmes de différentiation automatiques* ont été mis au point, tels que, entre autres, `ADIFOR` [5], `PADRE2` [3], `ADIC` [1] ou O$\partial$yssée [32]. Seulement, si la correction de ces systèmes n'est pas formellement établie, les programmes différentiés obtenus ne peuvent être considérés corrects, et de ce fait, la phase de test n'est pas évitée. L'idée est donc, ici, de montrer formellement, en utilisant la bibliothèque des nombres réels développée dans cette thèse, la correction du système de différentiation automatique O$\partial$yssée, par rapport à sa sémantique. Nous en étudierons la faisabilité en prenant, tout d'abord, un exemple simple, en nous restreignant aux programmes composés exclusivement d'*affectations* et de *séquences*. Les preuves présentées dans ce chapitre sont quasiment achevées en Coq. Nous verrons ensuite comment il serait possible d'étendre cette méthode à un ensemble plus vaste de programmes.

Prouver la correction de cet algorithme revient, dans ce cas, à montrer la commutation du diagramme de la figure 5.1. Ce diagramme traduit le fait que la dérivée de l'interprétation sémantique d'un programme FORTRAN est égale à l'interprétation sémantique du programme FORTRAN généré par O$\partial$yssée.

Nous verrons dans ce chapitre dans quelle mesure la commutation de ce diagramme peut être montrée.

$$\begin{array}{ccc}
\text{Prog. FORTRAN} & \overset{\textit{sémantique}}{\longrightarrow} & (\mathbb{R} \to \mathbb{R}) \\
O\partial yss\acute{e}e \downarrow & & \downarrow \textit{Dérivée formelle} \\
\text{Prog. FORTRAN} & \underset{\textit{sémantique}}{\longrightarrow} & (\mathbb{R} \to \mathbb{R})
\end{array}$$

FIG. 5.1 – Diagramme de commutation

## 5.1 Présentation d'O∂yssée

O∂yssée [32] est un logiciel de différentiation automatique, développé à l'INRIA Sophia Antipolis. Il met en œuvre à la fois la dérivation en mode direct et en mode inverse. Il prend en entrée un programme, écrit en Fortran-77, calculant une fonction, de $\mathbb{R}^n$ dans $\mathbb{R}^p$, différentiable par morceaux, et fournit un nouveau programme qui calcule une dérivée tangente ou cotangente : la dérivée tangente, appelée mode direct, correspond à une dérivée directionnelle, tandis que la cotangente, appelée mode inverse, est un gradient. Nous nous intéresserons principalement au mode direct, qui est plus simple. Néanmoins, nous ferons parfois allusion au mode inverse, car c'est ce mode, calculant un gradient, qui est, à terme, intéressant parmi nos objectifs futurs.

### 5.1.1 Préliminaires mathématiques

O∂yssée étant un système de différentiation, commençons par décrire cette notion ainsi que sa représentation au moyen de la notion de dérivée, puisque c'est cette dernière qui est formalisée formellement dans Coq, et non la différentielle.

Soit une fonction $f : \mathbb{R}^n \to \mathbb{R}^p$.

**Définition 5.1.1 (différentiabilité)** *On dit que $g$ est différentiable au point $a$, s'il existe une application linéaire continue $L$ telle que* $\lim_{u\to 0} \frac{1}{|u|}[g(a+u) - g(a) - L(u)] = 0$.

Notons $f(x_1, ..., x_n) = (f_1(x_1, ..., x_n), ..., f_p(x_1, ..., x_n))$ où $f_k : \mathbb{R}^n \to \mathbb{R},\ k = 1, ..., p$.

**Proposition 5.1.1** *La fonction $f$ est différentiable en $a$ si et seulement si chaque $f_k$ est différentiable en $a$ et on a alors $df(a) = (df_1(a), ..., df_p(a))$.*

**Preuve** Nous pourrons nous référer, par exemple, à [57]. ■

$df(a)$ est représentée dans la base canonique de $\mathbb{R}^n$ et $\mathbb{R}^p$ par la matrice Jacobienne de $f$ en $a$ notée $J_f(a) : \left( \frac{\partial f_i}{\partial x_j}(a) \right)_{1\leq i\leq p,\ 1\leq j\leq n}$

La notion de différentiabilité peut se représenter au moyen de la dérivée. En effet, une dérivée partielle en $x_j$ n'est autre qu'une dérivée par rapport à la variable $x_j$ en considérant les autres variables comme des constantes.

### 5.1.2 Le mode direct

Une étape préliminaire est la phase de repérage des *variables actives* dans un programme FORTRAN, définies de la façon suivante :

**Définition 5.1.2 (variable active)** *Une variable v (ou valeur) est dite active :*

1. *si v est une valeur de type $REAL$[1] d'entrée du programme et si l'utilisateur la spécifie comme étant active*
   *ou*
2. *si v est calculée à partir d'au moins une variable active*

Cette notion de variable active intervient de manière significative dans le résultat obtenu, nous verrons comment l'incorporer dans notre formalisation.

De manière générale, la génération de code dérivé suit la règle suivante : la structure du programme est conservée. En d'autres termes, le code dérivé d'une sous-procédure sera une sous-procédure, celui d'un if sera un if, et ainsi de suite pour les boucles et les autres structures de contrôle qui composent les programmes impératifs.

Le mode direct pourrait se résumer de la manière suivante : soit $P$ un programme, $X$ et $Y$ les vecteurs représentant respectivement les variables d'entrée et de sortie. On a alors :

$$P(X \to Y) \underrightarrow{\text{direct mode}} \dot{P}((X,\dot{X}) \to (Y,\dot{Y}))$$

avec $\dot{Y} = J \cdot \dot{X}$ où $J$ représente la matrice jacobienne en $X$.

$\dot{P}$ retourne donc, en ayant un vecteur quelconque $\dot{X}$, un nouveau vecteur $J \cdot \dot{X}$. L'utilisateur a, de plus, la possibilité de déclarer que certaines composantes de $\dot{X}$ sont nulles, auquel cas, elles n'apparaissent pas comme argument de $\dot{P}$ (cela permet de définir le vecteur actif). En d'autres termes, si les variables d'entrée actives sont $v_i$, celles de sortie actives sont $v_o$ et que $P$ calcule $v_o = f(v_i)$, alors ce mode calculera $v_o$ et $\dot{v_o}$ en $v_i$ dans la direction $\dot{v_i}$ c'est-à-dire $\dot{v_o} = f'(v_i) \cdot \dot{v_i}$. Prenons l'exemple simple suivant, sur lequel nous reviendrons par la suite.

```
subroutine cube (x,z)
real y
  y=x*x
  z=y*x
end
```

Nous allons différencier cette routine par rapport à la variable x. La commande suivante d'O∂yssée différentie `cube` par rapport à `x` en mode direct (`tl`) :

```
diff -head cube -vars x -tl -o std
```

Ce que nous avons noté précédemment sous la forme $\dot{v}$ est noté `VTTL` par O∂yssée. Nous obtenons alors :

```
COD Compilation unit : cubetl
COD Derivative of unit :  cube
COD Dummys:  x z
COD Active IN   dummys:  x
COD Active OUT  dummys:  z
COD Dependencies between IN and OUT:
COD z <--   x

      SUBROUTINE CUBETL (X, Z, XTTL, ZTTL)
```

[1] $REAL$ est le type FORTRAN représentant les nombres flottants.

```
REAL Y
REAL YTTL
YTTL = XTTL*X+X*XTTL
Y = X*X
ZTTL = XTTL*Y+X*YTTL
Z = Y*X
END
```

Donnons une transcription mathématique de ce programme dérivé.

Le programme que nous souhaitons différentier, par rapport à `x` est le suivant :

```
y=x*x;
z=y*x
```

L'instruction `y=x*x` est définie mathématiquement par la fonction $f$ :

$f : \mathbb{R}^3 \to \mathbb{R}^3$
$f : (x, y, z) \mapsto (f_1(x, y, z), f_2(x, y, z), f_3(x, y, z))$ où
$f_1(x, y, z) = x$
$f_2(x, y, z) = x \times x$
$f_3(x, y, z) = z$

L'instruction `z=y*x`, quant à elle, est définie par :

$g : \mathbb{R}^3 \to \mathbb{R}^3$
$g : (x, y, z) \mapsto (g_1(x, y, z), g_2(x, y, z), g_3(x, y, z))$ où
$g_1(x, y, z) = x$
$g_2(x, y, z) = y$
$g_3(x, y, z) = y \times x$

La séquence est alors la *composition* de ces deux fonctions, notée $g \circ f$. La différentielle de $g \circ f$ en $x_0$ est $D_{x_0}(g \circ f) = D_{f(x_0)} g \circ D_{x_0} f$. La dérivée tangente s'obtient en appliquant cette fonction linéaire de $\mathbb{R}^3$ dans $\mathbb{R}^3$ au vecteur $(\dot{x}, \dot{y}, \dot{z})$ donné par l'utilisateur. Pour cela, commençons par calculer les matrices Jacobiennes de $f$ en $(x, y, z)$ et de $g$ en $f(x, y, z)$ :

$$J_f(x, y, z) = \begin{pmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} & \frac{\partial f_1}{\partial z} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} & \frac{\partial f_2}{\partial z} \\ \frac{\partial f_3}{\partial x} & \frac{\partial f_3}{\partial y} & \frac{\partial f_3}{\partial z} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2x & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

et

$$J_g(x, y, z) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ y & x & 0 \end{pmatrix}$$

et donc

$$J_g(x, x^2, z) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x^2 & x & 0 \end{pmatrix}$$

La matrice jacobienne de $g \circ f$ est donc la suivante :

$$J_{g \circ f}(x, y, z) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x^2 & x & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 2x & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2x & 0 & 0 \\ x^2 + 2x^2 & 0 & 0 \end{pmatrix}$$

En projetant sur $(\dot{x}, 0, 0)$, qui est le vecteur actif au début (noté `IN` par O∂yssée) :

$$\begin{pmatrix} 1 & 0 & 0 \\ 2x & 0 & 0 \\ 3x^2 & 0 & 0 \end{pmatrix} \begin{pmatrix} \dot{x} \\ 0 \\ 0 \end{pmatrix}$$

on obtient le vecteur suivant :

$$\begin{pmatrix} \dot{x} \\ 2x\dot{x} \\ 3x^2\dot{x} \end{pmatrix}$$

Vérifions maintenant que le programme rendu par O∂yssée correspond bien à cette formalisation. La séquence résultat d'O∂yssée, en notant `XTTL` pour $\dot{x}$, `YTTL` pour $\dot{y}$ et `ZTTL` pour $\dot{z}$ (`X, Y` et `Z` restant inchangés) est la suivante :

```
YTTL = XTTL*X+X*XTTL
Y = X*X
ZTTL = XTTL*Y+X*YTTL
Z = Y*X
```

`YTTL` est exactement le résultat donné par la formalisation mathématique. Vérifions-le pour `ZTTL` : `ZTTL=XTTL*Y+X*YTTL=XTTL*X*X+X*(XTTL*X+X*XTTL)=3*X*X*XTTL`

L'information concernant les variables par rapport auxquelles la différentiation est effectuée est utilisée pour distinguer les variables actives, qui, elles-mêmes sont utilisées pour faire la projection. L'exemple qui va suivre fait apparaître plus clairement cette notion. Nous pourrons nous référer à l'annexe C pour les programmes complets rendus par O∂yssée concernant cet exemple.

Considérons la procédure suivante :

```
subroutine exple (x,y,z)
 z=y*z
 y=x*z
 x=2*x
end
```

Nous commencerons par la différentier par rapport à `x` puis par rapport à `x` et `y`. Comme dit précédemment, ces notions n'interviennent que lors de la projection de la matrice jacobienne. De ce fait, l'étape suivante est commune aux deux dérivations de code. Cette séquence de trois affectations peut être définie mathématiquement de la manière suivante :

$f : \mathbb{R}^3 \to \mathbb{R}^3$
$f : (x, y, z) \mapsto (f_1(x, y, z), f_2(x, y, z), f_3(x, y, z))$ où
$f_1(x, y, z) = x$
$f_2(x, y, z) = y$
$f_3(x, y, z) = y \times z$

$g : \mathbb{R}^3 \to \mathbb{R}^3$
$g : (x, y, z) \mapsto (g_1(x, y, z), g_2(x, y, z), g_3(x, y, z))$ où
$g_1(x, y, z) = x$
$g_2(x, y, z) = x \times z$
$g_3(x, y, z) = z$

$h : \mathbb{R}^3 \to \mathbb{R}^3$
$h : (x, y, z) \mapsto (h_1(x, y, z), h_2(x, y, z), h_3(x, y, z))$ où
$h_1(x, y, z) = 2 \times x$
$h_2(x, y, z) = y$
$h_3(x, y, z) = z$

Nous obtenons la jacobienne suivante :

$$J_{h \circ g \circ f}(x, y, z) = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ yz & 0 & x \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & z & y \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ yz & xz & xy \\ 0 & z & y \end{pmatrix}$$

Il nous reste à différentier par rapport aux variables choisies. Pour ce faire, il suffit de projeter la matrice jacobienne calculée précédemment sur le vecteur actif correspondant. Le vecteur actif comprend les différentielles des variables qui sont actives *au début* de la séquence d'instructions.

1. Différentiation par rapport à `x` :
   Dans ce cas,le vecteur actif est $(\dot{x}, 0, 0)$. En faisant la projection sur ce vecteur, nous obtenons :

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ yz & xz & xy \\ 0 & z & y \end{pmatrix} \begin{pmatrix} \dot{x} \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2\dot{x} \\ yz\dot{x} \\ 0 \end{pmatrix}$$

2. Différentiation par rapport à `x` et à `y` :
   Dans ce cas,le vecteur actif est $(\dot{x}, \dot{y}, 0)$. En faisant la projection sur ce vecteur, nous obtenons :

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ yz & xz & xy \\ 0 & z & y \end{pmatrix} \begin{pmatrix} \dot{x} \\ \dot{y} \\ 0 \end{pmatrix} = \begin{pmatrix} 2\dot{x} \\ yz\dot{x} + xz\dot{y} \\ z\dot{y} \end{pmatrix}$$

Nous pourrons vérifier aisément que ce résultat et celui fourni par O∂yssée à l'annexe C correspondent. Nous essaierons de montrer que l'algorithme utilisé par O∂yssée consistant à différentier instruction par instruction calcule bien une différentielle.

## 5.2 Un exemple préliminaire simple

Cet exemple va nous permettre d'appréhender la démarche à suivre pour notre preuve de correction d'algorithme. Il s'agit pour cet exemple, de considérer les programmes à une variable réelle, ne contenant qu'une affectation. Soit le programme calculant la fonction carré ainsi que le programme dérivé `CARRETL` rendu par O∂yssée :

```
subroutine carre (x)        SUBROUTINE CARRETL (X, XTTL)
  x=x*x                       XTTL = XTTL*X+X*XTTL
end                           X = X*X
                            END
```

Dans ce cas particulier, il nous faut montrer que le diagramme de la figure 5.2 commute. Nous appellerons "*O∂yssée modifié*" le résultat donné par O∂yssée mais affecté à la même variable d'entrée $x$ et où XTTL vaut 1.

$$\begin{array}{ccc} \text{x=x*x : prog} & \overset{\textit{sémantique}}{\longrightarrow} & \lambda x.(x \times x) : \ \mathtt{R}\!\to \mathtt{R} \\ O\partial yss\acute{e}e\ modifi\acute{e} \downarrow & & \downarrow \textit{Dérivée de } \mathsf{Coq} \\ \text{x=x+x : prog} & \underset{\textit{sémantique}}{\longrightarrow} & \lambda x.(x + x) : \ \mathtt{R}\!\to \mathtt{R} \end{array}$$

FIG. 5.2 – Diagramme de commutation de $x^2$

En réalité, dans le développement ci-après, nous montrerons un peu plus que la correction pour la fonction $x^2$. Notre formalisation s'étend aux fonctions polynomiales à une variable réelle et coefficients réels.

### 5.2.1 Sémantique

Nous définissons d'abord un type pour les expressions FORTRAN. Une expression FORTRAN est, dans ce cas, une affectation d'une expression arithmétique à une variable. Une expression arithmétique peut comprendre des constantes réelles, une variable et des opérations sur les expressions arithmétiques. Nous commençons donc par définir le type des expressions arithmétiques. Suivra le type de la variable (`tvar`), à valeur dans `Type` de par la sorte de $\mathbb{R}$ dans Coq, puis la définition d'une expression FORTRAN :

```
Inductive expr:Type:=
          numR:R->expr
         |var:expr
         |Fplus:expr->expr->expr
         |Fmult:expr->expr->expr
         |Fminus:expr->expr->expr
         |Fopp:expr->expr
         |Fpow:expr->nat->expr.

Inductive tvar:Type:=var1:tvar.

Inductive For:Type:=Aff:tvar->expr->For.
```

De manière similaire, nous pouvons donner une sémantique à ces expressions. Nous donnons une sémantique naturelle à valeur dans l'espace des fonctions de type $\mathbb{R} \to \mathbb{R}$. Une interprétation de l'affectation consiste simplement ici en l'interprétation de l'expression affectée :

```
Fixpoint sem_arith [arith:expr]:R->R:=
  Cases arith of
     (numR x)=> [y:R]x
    |var   => [x:R]x
    |(Fplus e1 e2)=>[x:R]``((sem_arith e1) x)+((sem_arith e2) x)``
    |(Fmult e1 e2)=>[x:R]``((sem_arith e1) x)*((sem_arith e2) x)``
    |(Fminus e1 e2)=>[x:R]``((sem_arith e1) x)-((sem_arith e2) x)``
```

```
    |(Fopp e)=>[x:R]``-((sem_arith e) x)``
    |(Fpow e i)=>[x:R](pow ((sem_arith e) x) i)
  end.

Definition sem_For [for:For]:R->R:=
  Cases for of (Aff v e)=>(sem_arith e) end.
```

### 5.2.2 Algorithme de dérivée

Comme nous l'avons dit lors de la présentation d'O∂yssée, la structure du programme est conservée. La dérivée d'une expression arithmétique est une expression arithmétique et la dérivée d'une affectation reste une affectation. Etant donné que cette formalisation ne contient qu'une seule variable, la dérivée de l'affectation sera l'affectation de la dérivée de l'expression à la même variable de départ :

```
Fixpoint deriv_expr [term:expr]:expr:=
  Cases term of
           (numR r)=>(numR ``0``)
          |var => (numR ``1``)
          |(Fplus a1 a2)=>
             (Fplus (deriv_expr a1) (deriv_expr a2))
          |(Fmult a1 a2)=>
      (Fplus (Fmult (deriv_expr a1) a2) (Fmult a1 (deriv_expr a2)))
          |(Fminus a1 a2)=>
             (Fminus (deriv_expr a1) (deriv_expr a2))
          |(Fopp a)=>(Fopp (deriv_expr a))
          |(Fpow e i)=>(Fmult
      (Fmult (numR (INR i)) (Fpow e (minus i (S O))))
                      (deriv_expr e))
  end.

Definition deriv_For [termfor:For]:For:=
  Cases termfor of (Aff v e)=>(Aff v (deriv_expr e)) end.
```

### 5.2.3 Preuve de correction

La correction de cet algorithme, dans ce cas simple, consiste à montrer les deux lemmes de correction correspondant aux définitions énoncées précédemment. Le premier lemme concerne les expressions arithmétiques. Nous considérons toutes les expressions vivant dans tout l'espace des réels et non dans un domaine $D$ uniquement, ce qui s'énonce en Coq par `[_ :R]True`. Nous verrons lors de la preuve comment appliquer le lemme de composition dans le cas de la fonction puissance.

```
Lemma corr_deriv_arith:(e:expr)(x0:R)
     (D_in (sem_arith e) (sem_arith (deriv_expr e)) [_:R]True x0).
```

**Preuve** La preuve se fait par induction sur l'expression `e`.
Nous avons donc 7 propriétés à montrer, qui l'ont été antérieurement, lors du développement de la bibliothèque des nombres réels que nous avons présenté au chapitre 2.

- Cas de la constante : application du lemme `Dconst`.
- Cas de la variable : application du lemme `Dx`.

- Cas de l'addition : application du lemme `Dadd`.
- Cas de la multiplication : application du lemme `Dmult`.
- Cas de la soustraction : application du lemme `Dminus`.
- Cas de l'opposé : application du lemme `Dopp`.
- Cas de la puissance : application du lemme `D_pow_n`. Ce lemme étant un lemme de composition, il nous faut montrer, pour pouvoir l'utiliser, que, si $x \in \mathbb{R}$ alors toute expression arithmétique est aussi dans $\mathbb{R}$, ce qui est trivialement vrai dans notre cas.

■

Il nous est maintenant plus aisé de montrer le lemme de correction pour l'affectation :

```
Theorem corr_deriv:(e:For)(x0:R)
       (D_in (sem_For e) (sem_For (deriv_For e)) [_:R]True x0).
```

**Preuve** La preuve se fait également par induction sur `e` puis, de par la définition de l'affectation, par application directe du lemme de correction précédent. ■

Nous allons suivre la même démarche afin de tenter de montrer la correction de l'algorithme d'O∂yssée sur un ensemble de programmes FORTRAN plus conséquent.

## 5.3 Une sémantique pour FORTRAN

Nous nous intéresserons à un noyau de FORTRAN composé de *variables réelles* ainsi que l'*affectation* et de la *séquence*. Nous étudierons ultérieurement les autres structures de contrôle car, comme nous le verrons, leur traitement est différent. Les programmes manipulant d'autres types de données que les réels, tels que les entiers ou les booléens pourront également être pris en compte ultérieurement[2]. Nous présenterons, à la fin de cette partie, une proposition de sémantique permettant de prendre en compte de tels programmes. Notre sémantique ne prendra pas en compte les spécificités particulières au langage FORTRAN -sur le prétypage des variables par le nommage, par exemple. Elle peut donc être utilisée pour n'importe quel noyau de langage impératif.

Notre noyau de sémantique contiendra, entre autres, un environnement, une syntaxe abstraite d'un sous-ensemble d'expressions de FORTRAN, l'interprétation de cette syntaxe abstraite dans Coq et l'interprétation de l'algorithme de différentiation.

### 5.3.1 L'environnement

Notre environnement est un vecteur de $\mathbb{R}^n$ ou plus précisément, une liste d'association entre les variables de type réelles, indexées par des entiers, et leur valeur associée. Nous avons donc en Coq :

```
Inductive rvar:Type:=varreal:nat->rvar.
Definition Env:=(listT (prodT rvar R)).
```

`rvar` représente les variables (réelles).
`listT` et `prodT` sont définies similairement à `list` et `prod` mais à valeur dans la sorte `Type` puisque, comme nous l'avons vu antérieurement, `R` est de type `Type`.

[2]Remarquons que la prise en compte de ces types de données n'ont pas une influence significative de par le fait que les dérivées se font par rapport à des variables réelles et que, de ce fait, introduire des variables entières ou booléennes ne revient qu'à introduire des constantes.

### 5.3.2 Les expressions

Nous allons maintenant définir le type des expressions arithmétiques suivi du type des structures impératives que nous souhaitons utiliser. Les expressions arithmétiques seront composées de constantes, variables réelles et des opérateurs usuels d'addition, de multiplication, de soustraction, de puissance. Le type minimal, pour notre exemple, des expressions arithmétiques s'exprime de la même façon que pour l'exemple préliminaire simple (page 97) exepté pour les variables où l'on a :

```
|var:rvar->expr
```

Pour ce qui est des structures du langage FORTRAN, nous nous intéresserons ici exclusivement à l'affectation et à la séquence.

```
Inductive For:Type:= Aff:rvar->expr->For
                    |Seq:For->For->For.
```

### 5.3.3 La sémantique

Nous avons choisi de donner une sémantique naturelle [44], c'est-à-dire à grands pas, qui interprète notre langage vers des fonctions de type $\mathbb{R} \rightarrow \mathbb{R}$. Cette sémantique est adaptée car elle élimine les programmes qui bouclent[3]. Ce choix permet une utilisation quasi immédiate de nos définitions et propriétés de dérivée faites dans Coq. Nous commencerons par donner une sémantique aux variables, aux expressions arithmétiques, puis à notre mini-langage lui-même. L'évaluation d'une variable sera sa valeur associée dans l'environnement. La fonction `myassoc`, écrite en Coq en utilisant un `Fixpoint`, va justement rechercher dans l'environnement la valeur associée à une variable donnée.

```
Definition sem_var [v:rvar;l:Env]:R:=(myassoc v l).
```

L'évaluation des expressions arithmétiques est sans surprise :

```
Fixpoint sem_arith [l:Env;arith:expr]:R:=
  Cases arith of
     (numR x)=> x
    |(var r) =>  (sem_var r l)
    |(Fplus e1 e2)=>``(sem_arith l e1)+(sem_arith l e2)``
    |(Fmult e1 e2)=>``(sem_arith l e1)*(sem_arith l e2)``
    |(Fminus e1 e2)=>``(sem_arith l e1)-(sem_arith l e2)``
    |(Fopp e)=>``-(sem_arith l e)``
    |(Fpow e i)=>(pow (sem_arith l e) i)
  end.
```

L'interprétation de l'affectation et de la séquence modifient l'environnement : pour l'affectation on crée une nouvelle affectation avec l'interprétation de la partie droite et pour la séquence `f1;f2` on donne une interprétation de `f2` dans l'environnement modifié après l'interprétation de `f1`.

```
Fixpoint sem_For [l:Env;for:For]:Env:=
  Cases for of
   (Aff v e)=>(replace v (sem_arith l e) l)
  |(Seq f1 f2) => (sem_For (sem_For l f1) f2)
  end.
```

[3]Bien que nous ne puissions nous trouver dans ce cas puisque nous ne considérons ici que des programmes contenant la séquence et l'affectation, le choix de cette sémantique nous servira lorsque nous étendrons notre formalisation à de plus vastes programmes (contenant des boucles, par exemple) (section 5.6).

où `replace` est une fonction qui remplace ou affecte une valeur associée à une variable de l'environnement par une autre, ce qui permet les affectations successives.

Il nous faut maintenant une interprétation vers une fonction réelle. Pour cela nous pouvons, par exemple, récupérer la valeur associée à une variable de sortie après évaluation du programme :

```
Definition sem [for:For;l:Env;vi,vo:rvar]:R->R:=
 [r:R](myassoc vo (sem_For (consT (pairT rvar R vi r) l) for)).
```

où `vi` et `vo` représentent respectivement la variable d'entrée significative (donnée active par l'utilisateur) et la variable de sortie.

## 5.4 Formalisation de l'algorithme de différentiation

Comme nous l'avons dit précédemment, il ne s'agit pas, ici non plus, d'une véritable différentiation mais plutôt d'une dérivation. En effet, nous utiliserons l'algorithme de différentiation d'O∂yssée, mais nous l'appliquerons en ne différentiant que par rapport à une variable et en considérant les `*TTL`=1 après substitution. Nous obtenons ainsi la dérivée d'une fonction de type $\mathbb{R} \to \mathbb{R}$. Néanmoins, comme nous le verrons un peu plus loin, nous pourrons parler soit de *dérivée* soit de *différentiation*, suivant les cas, car nous aurons parfois besoin d'exploiter la différence entre les deux notions, bien que, comme nous l'avons vu à la section 5.1.1, la seconde notion puisse s'exprimer par rapport à la première. Notons qu'il s'agit de faciliter notre utilisation des propriétés montrées dans Coq précédemment concernant la dérivée. Pour cela nous avons donc choisi de projeter le vecteur d'entrée sur la variable qui nous intéresse. Nous commencerons donc par montrer, informellement, que cette projection sur une variable est bien équivalente aux compositions des matrices Jacobiennes décrites lors de la présentation d'O∂yssée.

Considérons la fonction `cube` présentée antérieurement. Nous pourrons nous reporter au résultat rendu par O∂yssée pour la suite de ce paragraphe. La variable d'entrée qui nous intéresse est `x` et celle de sortie, représentant $x^3$, est `z`. Selon notre sémantique et notre algorithme de dérivation, notre formalisation consistera à renvoyer la valeur de la variable qui contient la dérivée (`ZTTL`) en tenant compte de l'environnement. Dans cet exemple, le lemme de correction consiste donc à vérifier que $3x^2$ (valeur de `ZTTL` lorsque `XTTL`=1) est bien la dérivée de $x^3$ (valeur de `Z`).

La fonction qui nous donne la dérivée des expressions arithmétiques est quasiment identique à celle que nous avons donnée dans l'exemple précédent. Seul le cas des variables diffère. Puisque nous pouvons avoir plusieurs variables, et que, comme nous le verrons par la suite, nous dérivons le séquence instruction par instruction, nous ne pouvons faire une dérivée naïve des variables[4]. Cette solution n'est pas satisfaisante car, alors, les instructions d'une séquence seraient totalement indépendantes. Pour respecter l'algorithme d'O∂yssée, nous devons donc, en ce qui concerne les variables, utiliser ici plutôt un algorithme de différentiation. L'expression dérivée d'une variable sera donc sa variable dérivée associée, que nous commencerons par calculer avant toute chose en créant une liste d'association $(var, dvar)$. Cette liste sera de type `lvar`, ainsi défini :

```
Definition lvar:=(listT (prodT rvar rvar)).
```

[4]Cette dérivée naïve consisterait à comparer la variable avec celle qui est active et, si ce sont les mêmes (mêmes indices), alors la dérivée vaudrait 1, sinon nous la considérerions comme une constante et la dérivée vaudrait 0.

Le cas de filtrage de la variable doit donc être modifié par rapport à l'exemple de la section 5.2 et se traduit donc ainsi dans la fonction `deriv_expr`, ayant une liste `l` de type `lvar` en paramètre supplémentaire :

```
|(var x) => (var (myassocV x l))
```

où `(myassocV x l)` retourne la variable dérivée correspondant à `x` dans la liste `l`.

La fonction Coq représentant l'algorithme de dérivée énonce ce que nous avons présenté précédemment, à savoir, que :

- la dérivée de l'affectation est une séquence entre l'affectation à une nouvelle variable de la dérivée de l'expression de droite et l'affectation d'origine
- la dérivée d'une séquence est la séquence de la dérivée des expressions qui formaient la séquence d'origine

En ce qui concerne l'affectation, nous avons donc besoin d'une nouvelle variable à laquelle nous pourrons affecter la dérivée. Cette variable se trouve dans la liste d'associations que nous avons construite auparavant. Considérons une nouvelle fois l'exemple sur la fonction `cube`. Le programme d'origine, une fois traduit en expression `For` de Coq, est :

```
(Seq (Aff (varreal (1)) (Fmult  (varreal (0)) (varreal (0))))
     (Aff (varreal (2)) (Fmult  (varreal (1)) (varreal (0)))))
```

où `(varreal (i))`, `i`$=0,1,2$ représentent respectivement `x,y,z`.

La liste d'associations correspondant à ce programme est la suivante :

```
[(((varreal (0)),(varreal (3)));
 (((varreal (1)),(varreal (4)));
 (((varreal (2)),(varreal (5)))]
```

où, pour garder la correspondance avec notre programme, `(varreal (i))`, `i`$=3,4,5$ représentent respectivement `XTTL,YTTL,ZTTL`.

En suivant l'algorithme, notre fonction dérivée nous donnera donc :

```
(Seq (Seq (Aff (varreal (4)) (Fplus
       (Fmult (varreal (3)) (varreal (0)))
       (Fmult (varreal (0)) (varreal (3)))))
         (Aff (varreal (1)) (Fmult  (varreal (0)) (varreal (0)))))
     (Seq (Aff (varreal (5)) (Fplus
       (Fmult (varreal (4)) (varreal (0)))
       (Fmult (varreal (1)) (varreal (3)))))
         (Aff (varreal (2)) (Fmult  (varreal (1)) (varreal (0))))))
```

En Coq, l'algorithme de différentiation d'O∂yssée pourra se formaliser comme suit.

```
Fixpoint deriv_for_aux [l:lvar; termfor:For]:For:=
  Cases termfor of
    (Aff v e) =>
      (Seq (Aff (myassocV v l) (deriv_expr l e)) (Aff v e))
   |(Seq f1 f2) => (Seq (deriv_for_aux l f1)
                        (deriv_for_aux l f2))
  end.

Definition deriv_for [termfor:For]:For:=
       (deriv_for_aux (make_list termfor) termfor).
```

où `make_list` construit la liste d'associations $(var, dvar)$.

**Remarque 5.4.1** *Deux passes sont faites sur le programme avant d'appliquer notre fonction de différentiation. Une première passe permet de déterminer le dernier indice de variable utilisé (ce qui équivaut à un nom de variable). La seconde permet de construite la liste d'association à partir de ce dernier indice.*

## 5.5 Preuve de correction

Nous pouvons maintenant énoncer et montrer la correction de l'algorithme utilisé par O∂yssée afin de différentier les programmes FORTRAN composés de la séquence et de l'affectation. Comme nous l'avons dit précédemment, cela équivaut à montrer que l'interprétation dans Coq d'un tel programme se dérive (dans Coq) en l'interprétation dans Coq du programme différentié par O∂yssée(ce qui est exprimé par le diagramme 5.1).

Avant d'en donner un énoncé informel (mathématique) puis formel (Coq), rappelons que l'interprétation sémantique dépend de la variable que nous choisissons active, ainsi que de la variable que nous choisissons comme étant valeur de sortie. Ces deux variables doivent donc impérativement appartenir à l'environnement (contenant les variables et leur valeur associée). Cette méthode nous permet de ne gérer que des fonctions de types $\mathbb{R} \rightarrow \mathbb{R}$ que nous savons dériver en Coq. Le théorème peut donc s'énoncer ainsi :

**Théorème 5.5.1 (Correction)** *Soit une fonction sem donnant la sémantique d'un programme* FORTRAN *composé de séquences et d'affectations. Soient $v_i$ la variable d'entrée active et $v_o$ la variable de sortie. Soit env l'environnement.*
*Pour tout programme $p$, si $v_i \in env$ et $v_o \in env$ alors $sem'(p\ env\ v_i\ v_o) = (sem\ \dot{p}\ env\ v_i\ v_o)$ où $\dot{p}$ est le programme différentié correspondant à l'algorithme utilisé par* O∂yssée.

Ce théorème s'énonce en Coq de la manière suivante :

```
Theorem corr_deriv:(l:Env)(p:For)(vi,vo:rvar)(D:R->Prop)(x0:R)
    (memenv vi l)->(memenv vo l)->
    (D_in (sem p l vi vo) (sem (deriv_for p) l vi vo) D x0).
```

Afin de montrer ce théorème nous commencerons par montrer deux lemmes intermédiaires similaires, un, concernant l'affectation et l'autre, concernant la séquence :

```
Lemma corr_deriv_Aff:(l:Env)(e:expr)(r,vi,vo:rvar)(D:R->Prop)(x0:R)
  (memenv vi l)->(memenv vo l)->
  (D_in (sem (Aff r e) l vi vo)
     (sem (deriv_for (Aff r e)) l vi vo) D x0).

lemma corr_deriv_Seq:(l:Env)(f,f0:For)(vi,vo:rvar)(D:R->Prop)(x0:R)
       (memenv vi l)->(memenv vo l)->
  (D_in (sem (Seq f f0) l vi vo)
     (sem (deriv_for (Seq f f0)) l vi vo) D x0).
```

Pour les deux preuves qui vont suivre, nous noterons l'affectation " $:=$" et la séquence " ;". D'autre part, pour des raisons de lisibilité, nous n'utiliserons pas la notation $\dot{x}$ pour la différentielle d'O∂yssée, mais plutôt $\overline{x}$.

**Preuve** de `corr_deriv_Aff`
Par récurrence sur l'expression $e$.
Soient $H_0$ et $H_1$ les hypothèses $v_i \in l$ et $v_o \in l$.

1. Cas de la constante : nous devons montrer, sous les hypothèses précédentes, que
$\forall c : \mathbb{R}, sem'((r := c)\ l\ v_i\ v_o) = (sem\ \overline{(r := c)}\ l\ v_i\ v_o)$ c'est-à-dire que
$\forall c : \mathbb{R}, sem'((r := c)\ l\ v_i\ v_o) = (sem\ (dr := 0; r := c)\ l\ v_i\ v_o)$ où $c$ est une constante réelle.
Dans le cas $v_i = r$ et $v_o = dr$, cela revient à montrer, après simplifications grâce à $H_0$ et $H_1$, que $c' = 0$. Il suffit d'appliquer le lemme `Dconst` montré lors du développement concernant la dérivée (chapitre 2).
Dans les cas où $v_i \neq r$ ou $v_o \neq dr$, nous avons une contradiction avec $H_0$ ou $H_1$ car dans ces cas $v_i \notin l$ ou $v_o \notin l$.

2. Cas de la variable : nous devons montrer, sous les hypothèses précédentes, que
$\forall c : \mathbb{R}, sem'((r := x)\ l\ v_i\ v_o) = (sem\ \overline{(r := x)}\ l\ v_i\ v_o)$ c'est-à-dire que
$\forall c : \mathbb{R}, sem'((r := x)\ l\ v_i\ v_o) = (sem\ (dr := 1; r := x)\ l\ v_i\ v_o)$ où $x$ est une variable réelle.
Dans le cas $v_i = r$ et $v_o = dr$, cela revient à montrer, après simplifications grâce à $H_0$ et $H_1$, que $x' = 1$. Il suffit d'appliquer la lemme `Dx` montré lors du développement concernant la dérivée (chapitre 2).
Dans les cas où $v_i \neq r$ ou $v_o \neq dr$, nous avons une contradiction avec $H_0$ ou $H_1$ car dans ces cas $v_i \notin l$ ou $v_o \notin l$.

3. Cas de l'addition : nous avons les deux hypothèses de récurrence supplémentaires :
- $H_2$ : $\forall v_i, v_o, r, sem'((r := e_0)\ l\ v_i\ v_o) = (sem\ \overline{(r := e_0)}\ l\ v_i\ v_o)$ où $e_0$ est une expression
- $H_3$ : $\forall v_i, v_o, r, sem'((r := e_1)\ l\ v_i\ v_o) = (sem\ \overline{(r := e_1)}\ l\ v_i\ v_o)$ où $e_1$ est une expression
Nous devons montrer que $sem'((r := e_0 + e_1)\ l\ v_i\ v_o) = (sem\ \overline{(r := e_0 + e_1)}\ l\ v_i\ v_o)$ c'est-à-dire que $sem'((r := e_0 + e_1)\ l\ v_i\ v_o) = (sem\ (dr := e_0' + e_1'; r := e_0 + e_1)\ l\ v_i\ v_o)$.
Les hypothèses $H_2$ et $H_3$ se transforment, après simplification, en :
- $H_2$' : $\forall v_i, v_o, r, sem'((r := e_0)\ l\ v_i\ v_o) = (sem\ (dr := e_0'; r := e_0)\ l\ v_i\ v_o)$
- $H_3$' : $\forall v_i, v_o, r, sem'((r := e_1)\ l\ v_i\ v_o) = (sem\ (dr := e_1'; r := e_1)\ l\ v_i\ v_o)$
Dans le cas $v_i = r$ et $v_o = dr$, cela revient à montrer, après simplifications grâce à $H_0$ et $H_1$ et en utilisant $H_2$' et $H_3$', que $(e_0 + e_1)' = e_0' + e_1'$. Il suffit d'appliquer la lemme `Dadd` montré lors du développement concernant la dérivée (chapitre 2).
Dans les cas où $v_i \neq r$ ou $v_o \neq dr$, nous avons une contradiction avec $H_0$ ou $H_1$ car dans ces cas $v_i \notin l$ ou $v_o \notin l$.

4. Cas de la multiplication : cas similaire à l'addition.

5. Cas de la soustraction : cas similaire à l'addition.

6. Cas de l'opposé : nous avons une hypothèse de récurrence :
- $H_2$ : $\forall v_i, v_o, r, sem'((r := e_0)\ l\ v_i\ v_o) = (sem\ \overline{(r := e_0)}\ l\ v_i\ v_o)$ où $e_0$ est une expression
Nous devons montrer que $sem'((r := -e_0)\ l\ v_i\ v_o) = (sem\ \overline{(r := -e_0)}\ l\ v_i\ v_o)$ c'est-à-dire que $sem'((r := -e_0)\ l\ v_i\ v_o) = (sem\ (dr := -e_0'; r := -e_0)\ l\ v_i\ v_o)$.
L'hypothèse $H_2$ se transforme, après simplification, en :
- $H_2$' : $\forall v_i, v_o, r, sem'((r := e_0)\ l\ v_i\ v_o) = (sem\ (dr := e_0'; r := e_0)\ l\ v_i\ v_o)$
Dans le cas $v_i = r$ et $v_o = dr$, cela revient à montrer, après simplifications grâce à $H_0$ et $H_1$ et en utilisant $H_2$', que $(-e_0)' = -e_0'$. Il suffit d'appliquer la lemme `Dopp` montré lors du développement concernant la dérivée (chapitre 2).

Dans les cas où $v_i \neq r$ ou $v_o \neq dr$, nous avons une contradiction avec $H_0$ ou $H_1$ car dans ces cas $v_i \notin l$ ou $v_o \notin l$.

7. Cas de la puissance : cas similaire à l'opposé, en utilisant le lemme de composition pour la puissance.

■

**Preuve** de `corr_deriv_Seq`
Par double récurrence sur les programmes $f$ et $f_0$.
Soient $H_0$ et $H_1$ les hypothèses $v_i \in l$ et $v_o \in l$. Nous noterons l'affectation " :=" et la séquence " ;".

1. Cas affectation-affectation : nous devons montrer, sous les hypothèses précédentes, que $sem'((r_0 := e_0; r := e)\ l\ v_i\ v_o) = (sem\ \overline{(r_0 := e_0; r := e)}\ l\ v_i\ v_o)$ c'est-à-dire que $sem'((r_0 := e_0; r := e)\ l\ v_i\ v_o) = (sem\ ((dr_0 := e_0'; r_0 := e_0); (dr := e'; r := e))\ l\ v_i\ v_o)$ où $e_0$ et $e$ sont des expressions.
   Dans le cas $v_i = r$ et $v_o = dr$ ou $v_i = r_0$ et $v_o = dr_0$ ou $v_i = r$ et $v_o = dr_0$ ou $v_i = r_0$ et $v_o = dr$, après remplacement des valeurs à partir de l'environnement puis simplifications, nous nous pouvons appliquer le lemme précédent.
   Dans les autres cas on a soit $v_i \notin l$ soit $v_o \notin l$, ce qui nous donne une contradiction.
2. Cas affectation-séquence : nous avons les deux hypothèses de récurrence supplémentaires :
   - $H_2$ : $\forall e, v_i, v_o, r, sem'((r := e; f_1)\ l\ v_i\ v_o) = (sem\ \overline{(r := e; f_1)}\ l\ v_i\ v_o)$ où $f_1$ est un programme
   - $H_3$ : $\forall e, v_i, v_o, r, sem'((r := e; f_2)\ l\ v_i\ v_o) = (sem\ \overline{(r := e; f_2)}\ l\ v_i\ v_o)$ où $f_2$ est un programme
   Nous devons montrer, sous les hypothèses précédentes, que
   $sem'((r := e; (f_1; f_2))\ l\ v_i\ v_o) = (sem\ \overline{(r := e; (f_1; f_2))}\ l\ v_i\ v_o)$ c'est-à-dire que
   $sem'((r := e; (f_1; f_2))\ l\ v_i\ v_o) = (sem\ ((dr := e'; r := e); \overline{(f_1; f_2)})\ l\ v_i\ v_o)$ où $f_1$ et $f_2$ sont des programmes.
   Les hypothèses $H_2$ et $H_3$ se transforment, après simplification, en :
   - $H_2$' : $\forall e, v_i, v_o, r, sem'((r := e; f_1)\ l\ v_i\ v_o) = (sem\ (dr := e'; r := e; \overline{f_1})\ l\ v_i\ v_o)$
   - $H_3$' : $\forall e, v_i, v_o, r, sem'((r := e; f_2)\ l\ v_i\ v_o) = (sem\ (dr := e'; r := e; \overline{f_2})\ l\ v_i\ v_o)$
   En utilisant les deux hypothèses $H_2$' et $H_3$', on se ramène au cas précédent.
3. Cas séquence-affectation : symétrique au cas précédent.
4. Cas séquence-séquence : nous avons les quatre hypothèses de récurrence supplémentaires :
   - $H_2$ : $\forall v_i, v_o, sem'(((f_4; f_3); f_0)\ l\ v_i\ v_o) = (sem\ \overline{((f_4; f_3); f_0)}\ l\ v_i\ v_o)$ où $f_4$, $f_3$ et $f_0$ sont des programmes
   - $H_3$ : $\forall v_i, v_o, sem'(((f_4; f_3); f_1)\ l\ v_i\ mbv_o) = (sem\ \overline{((f_4; f_3); f_1)}\ l\ v_i\ v_o)$ où $f_4$, $f_3$ et $f_1$ sont des programmes
   - $H_4$ : si $\forall f_1, sem'((f_1; f_0))\ l\ v_i\ v_o) = (sem\ \overline{(f_1; f_0)}\ l\ v_i\ v_o)$ et si
   $\forall f_2, sem'((f_2; f_0))\ l\ v_i\ v_o) = (sem\ \overline{(f_2; f_0)}\ l\ v_i\ v_o)$ alors
   $\forall v_i, v_o, sem'(((f_2; f_1); f_0))\ l\ v_i\ v_o) = (sem\ \overline{((f_2; f_1); f_0)}\ l\ v_i\ v_o)$
   - $H_5$ : si $\forall f_2, sem'((f_2; f_0))\ l\ v_i\ v_o) = (sem\ \overline{(f_2; f_0)}\ l\ v_i\ v_o)$ et si
   $\forall f_0, sem'((f_0; f_1))\ l\ v_i\ v_o) = (sem\ \overline{(f_0; f_1)}\ l\ v_i\ v_o)$ alors
   $\forall v_i, v_o, sem'(((f_0; f_2); f_1))\ l\ v_i\ v_o) = (sem\ \overline{((f_0; f_2); f_1)}\ l\ v_i\ v_o)$
   En appliquant $H_4$ et $H_5$ avec $f_3$ $H_2$ $f_4$ $H_3$, nous obtenons les deux hypothèses suivantes :
   - $H_4$' : $\forall v_i, v_o, sem'(((f_4; f_3); f_0))\ l\ v_i\ v_o) = (sem\ \overline{((f_4; f_3); f_0)}\ l\ v_i\ v_o)$
   - $H_5$' : $\forall v_i, v_o, sem'(((f_4; f_3); f_1))\ l\ v_i\ v_o) = (sem\ \overline{((f_4; f_3); f_1)}\ l\ v_i\ v_o)$

Nous devons alors montrer que
$\forall v_i, v_o, sem'(((f_4;f_3);(f_0;f_1))\ l\ v_i\ v_o) = (sem\ \overline{((f_4;f_3);(f_0;f_1))\ l}\ v_i\ v_o)$ c'est-à-dire que $\forall v_i, v_o, sem'(((f_4;f_3);(f_0;f_1))\ l\ v_i\ v_o) = (sem\ (\overline{(f_4;f_3)};\overline{(f_0;f_1)}\ l\ v_i\ v_o)$, ce qui se fait en utilisant les deux hypothèses $H_4$' et $H_5$' afin de nous ramener à des cas similaires traités auparavant.

■

Le théorème principal se montre maintenant simplement[5] :

**Preuve** du théorème 5.5
Par cas sur le programme $p$.

1. Cas de l'affectation : par application directe du lemme `corr_deriv_Aff`.
2. Cas de la séquence : par application directe du lemme `corr_deriv_Seq`.

■

## 5.6 Discussion : un ensemble plus vaste de programmes

Nous avons montré qu'il était possible de montrer la correction de l'algorithme pour les programmes composés de l'affectation et de la séquence. Qu'en est-il pour des ensembles plus vastes de programmes, c'est-à-dire des programmes contenant des structures de contrôle telles que la conditionnelle, les boucles ou encore des structures de données particulières comme les tableaux ? Ce genre de structures de contrôle posent le problème de la *continuité* et de la *dérivabilité*. En effet, dans le cas des programmes définissant des fonctions discontinues, nous ne pouvons donner une dérivée en chaque point (en particulier aux points de discontinuité). Prenons, par exemple, la fonction valeur absolue définie par le programme suivant :

```
subroutine g (x,y)
 if (x.ge.0.) then
  y = x
 else
  y = -x
 end if
end
```

Nous savons que cette fonction n'est pas dérivable en 0. Néanmoins l'algorithme de différentiation d'O∂yssée donne tout de même une valeur en tous les points (voir annexe C). Il semble que, dans la pratique, ces programmes sont utilisés avec des valeurs assez éloignées des points de discontinuité. Par contre, même si cette "approximation" ne pose pas de problème majeur dans la plupart des cas, il est impossible d'en montrer la correction telle quelle. Il serait possible de contourner ce problème de deux manières :
- Soit en *linéarisant* le programme. Il s'agit de transformer un programme contenant des structures de contrôle en des programmes linéaires (ou "straight line"), c'est-à-dire contenant exclusivement l'affectation et la séquence. Par exemple, pour la conditionnelle, il suffit de séparer le programme d'origine en deux programmes distincts, un pour chaque branche (`then` et `else`).
- Soit en définissant un espace sur lequel la fonction est dérivable, ce qui semble être une solution plus satisfaisante car elle présente l'avantage de conserver la structure du programme

[5]Cette preuve est également immédiate en Coq

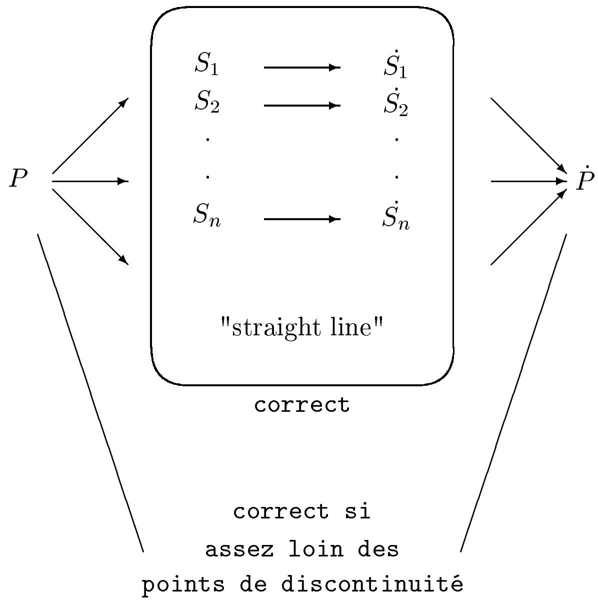



FIG. 5.3 –

intacte.
Le diagramme ci-dessous[6] résume la situation.

Cette méthode de linéarisation de programme permettrait de traiter un nombre important de programmes. Quant aux différents types de données, nous ne savons pas, pour le moment, s'il est possible de trouver une manière générale de procéder[7].

Comme nous l'avons vu, nous avons fait en sorte de nous ramener à des fonctions de type $\mathbb{R} \rightarrow \mathbb{R}$, ce qui nous restreint sur le nombre de variables actives donnée en entrée du programme. Nous pensons néanmoins qu'il est possible d'étendre cette preuves, dans un premier temps, aux fonctions de type $\mathbb{R}^n \rightarrow \mathbb{R}$ en formalisant en Coq une véritable notion de différentiation (et non de dérivée), puis, dans un second temps, aux fonctions de type $\mathbb{R}^n \rightarrow \mathbb{R}^m$.

De plus, comme nous l'avons indiqué au début de ce chapitre, les autres types de FORTRAN simples de donné (integer ou boolean) peuvent également être pris en compte lors de la sémantique du programme[8]. Nous devons pour cela définir des types spécifiques de variables pour chaque type de donnée. Il en va de même pour la sémantique. Nous en donnons ci-dessous un bref aperçu :

```
Inductive var_arith:Set:=
           Realvar:nat->var_arith
          |Integervar:nat->var_arith.
Inductive var:Set:=
```

[6] dû à Laurent Hascoët
[7] D'autant que tous ne sont pas encore pris en compte par les outils de différentiation.
[8] Cette sémantique plus complète a été entièrement formalisée en Coq.

```
          varith:var_arith->var
         |Booleanvar:nat->var.
Inductive numbers:Set:=
          Integer:Z->numbers
         |Real:R->numbers.
Inductive value_val:Set:=
                     Vnumbers:numbers->value_val
                    |Vbool:bool->value_val.
Inductive value:Set:=
                     Val:value_val->value
                    |Error:value.

Definition Env:=(list (var*value_val)).

Inductive expr_arith:Set:=
          num:numbers->expr_arith
         |vararith:var_arith->expr_arith
         |Fplus:expr_arith->expr_arith->expr_arith
         |Fmult:expr_arith->expr_arith->expr_arith
         |Fminus:expr_arith->expr_arith->expr_arith
         |Fmoins:expr_arith->expr_arith.

Inductive expr_Boolean:Set:=...

Inductive expr:Set:=
          Arith:expr_arith->expr
         |Boolean:expr_Boolean->expr.

Inductive Fortran:Set:=
          Aff:var->expr->Fortran
         |Seq:Fortran->Fortran->Fortran.

Fixpoint sem_arith [env:Env;arith:expr_arith]:value:=...

Fixpoint sem_Boolean [env:Env;Boolean:expr_Boolean]:value:=
  Cases Boolean of
     (Booleanc l)=>(Val (Vbool l))
    |(varBoolean v)=>....
    |(Fle n1 n2)=>
        (Cases (sem_arith env n1) (sem_arith env n2) of
          (Val v1) (Val v2)=>
              (Cases v1 v2 of
                 (Vnumbers vn1) (Vnumbers vn2)=>
                    (Cases vn1 vn2 of
                       (Integer i1) (Integer i2) =>
                          (Val (Vbool (Zle_bool i1 i2)))
                      |(Real r1) (Real r2) =>
                          (Val (Vbool (Rleb r1 r2)))
                      |(Integer i1) (Real r2) =>
                          (Val (Vbool (Rleb (IZR i1) r2)))
```

```
                   |(Real r1) (Integer i2) =>
                       (Val (Vbool (Rleb r1 (IZR i2))))
                  end)
               |_ _=>Error
            end)
        |_ _=>Error
      end)
  |...

Fixpoint sem [env:Env;term:Fortran]:Env:=...
```

# Conclusion

Une caractéristique commune à l'ensemble de ce travail est l'utilisation de nombres réels. Bien que notre but principal n'ait pas été une formalisation des nombres réels en Coq, c'est un préliminaire incontournable à l'utilisation des outils de preuves formelles sur des applications numériques ainsi que, bien souvent, géométriques. De ce fait, notre premier travail a consisté en un développement d'une bibliothèque sur les nombres réels. Il s'agissait de trouver un compromis satisfaisant entre le temps de développement[9] et une utilisation satisfaisante, ce qui nous a amenée à choisir une axiomatisation. Lors de l'utilisation effective de ce développement, nous avons pu constater que le résultat est plutôt positif. En effet, tant en ce qui concerne l'automatisation qu'en ce qui concerne la possibilité de montrer des théorèmes non triviaux, une axiomatisation est satisfaisante. L'implantation de la tactique réflexive Field ainsi que la preuve du théorème des trois intervalles en sont une bonne illustration. Nous n'avons d'ailleurs pas noté de différence significative d'utilisation entre cette axiomatisation et la construction des nombres réels faite en HOL, plus particulièrement dans le sens où les preuves utilisent massivement les tactiques de réécriture, confirmant notre intuition mathématique que la manière dont sont construits les nombres réels n'influe que sur les prémisses du développement de la théorie. Cette bibliothèque, accompagnée des tactiques d'automatisation que nous avons présentées la concernant, fait maintenant partie de la librairie standard distribuée avec la version courante de Coq.

L'application des méthodes formelles aux problèmes d'analyse numérique n'en est qu'à ses débuts. Le premier aspect de notre travail sur ce sujet a donc consisté en la recherche d'un sujet d'étude, en essayant, à la fois, de choisir un exemple *raisonnable*[10] et issu d'un problème réel. La formalisation et la preuve de correction pour notre sous-ensemble de programmes du système O∂yssée a montré que les systèmes de preuves formelles commencent à être adaptés à cet effet. Néanmoins, bien que nous soyons convaincus de l'importance de méthodes formelles dans ce domaine critique, nous avons pu constater que nous sommes encore assez loin de pouvoir les appliquer à des problèmes conséquents, et cela pour deux raisons principales. La première concerne les systèmes d'aide à la preuve. La formalisation de l'analyse réelle n'est pas immédiate et tous les systèmes n'en possèdent pas ou peu, ce qui, comme nous l'avons vu est indispensable pour ces problèmes. La seconde concerne le domaine de l'analyse numérique lui-même. Une grande majorité des programmes qui en sont issus utilisent des principes d'approximation, certains contrôlés mais d'autres pas. Une certaine *tolérance probabiliste* est, en effet, usuelle, dans le sens où, comme résumé dans la figure 5.3 par exemple, le problème ne peut se produire que rarement ou même jamais.

[9] Nous rappelons ici que le fait de traduire une bibliothèque déjà existante d'un autre système (en HOL par exemple) vers Coq est malheureusement loin d'être plus rapide, de par les spécificités propres à chaque système.

[10] c'est-à-dire *pas trop difficile* sachant qu'il n'est pas toujours aisé d'évaluer ce genre de considération à l'avance.

Cette optique est, bien évidemment, tout à fait contraire à notre but qui est de montrer formellement qu'aucun problème ne peut survenir. Dans le cas particulier que nous avons étudié, nous voyons que le problème des points de discontinuité en est un exemple frappant.

Au vu de ce travail, plusieurs futures directions apparaissent clairement. Le développement concernant les nombres réels doit se poursuivre. En particulier, nous pensons, à très court terme, aux fonctions transcendantes, puis à une notion de différenciation et d'intégration. Nous pensons qu'il est également souhaitable, dans le cas de `Coq`, d'étudier l'intégration d'une bibliothèque des réels intuitionniste, ce qui permettrait l'utilisation de la librairie actuelle, plus adaptée aux mathématiques standard, pour l'analyse, tout en ayant la possibilité d'utiliser une librairie intuitionniste en cas d'extraction. L'automatisation est, d'autre part, un point important de cette bibliothèque. En effet, ne pouvant profiter de l'un des points forts de `Coq`, qui est son principe d'induction, une automatisation en est donc tout à fait utile. Les tactiques `Ring, Field` et `Fourier`,entre autres, sont particulièrement utiles, mais nous pensons qu'il manque une tactique supplémentaire similaire à la tactique `Omega` (tactique développée par Pierre Crégut et implantant une procédure de décision pour l'arithmétique de Pressburger).
En ce qui concerne l'application des preuves formelles au domaine numérique, le champ d'exploration est encore très vaste. Outre les perspectives présentées au chapitre 5, une prochaine étape pourrait être une étude des algorithmes de calcul du gradient, considérés comme critiques dans leur implantation, paraissant toutefois abordables actuellement par nos méthodes.

# Annexes

# Annexe A

# Axiomatisation

```
Require Export ZArith.
Require Export TypeSyntax.

Parameter R:Type.
Parameter R0:R.
Parameter R1:R.
Parameter Rplus:R->R->R.
Parameter Rmult:R->R->R.
Parameter Ropp:R->R.
Parameter Rinv:R->R.
Parameter Rlt:R->R->Prop.
Parameter up:R->Z.

(**********)
Definition Rgt:R->R->Prop:=[r1,r2:R](Rlt r2 r1).

(**********)
Definition Rle:R->R->Prop:=[r1,r2:R]((Rlt r1 r2)\/(r1==r2)).

(**********)
Definition Rge:R->R->Prop:=[r1,r2:R]((Rgt r1 r2)\/(r1==r2)).

(**********)
Definition Rminus:R->R->R:=[r1,r2:R](Rplus r1 (Ropp r2)).

(**********)
Definition Rdiv:R->R->R:=[r1,r2:R](Rmult r1 (Rinv r2)).

(*********************************************************)
(*             {Field axioms}                           *)
(*********************************************************)
(*********************************************************)
(*s     Addition                                        *)
(*********************************************************)
```

```
(**********)
Axiom Rplus_sym:(r1,r2:R)``r1+r2==r2+r1``.
Hints Resolve Rplus_sym : real.

(**********)
Axiom Rplus_assoc:(r1,r2,r3:R)``(r1+r2)+r3==r1+(r2+r3)``.
Hints Resolve Rplus_assoc : real.

(**********)
Axiom Rplus_Ropp_r:(r:R)``r+(-r)==0``.
Hints Resolve Rplus_Ropp_r : real v62.

(**********)
Axiom Rplus_Ol:(r:R)``0+r==r``.
Hints Resolve Rplus_Ol : real v62.

(***********************************************************)
(*s      Multiplication                                    *)
(***********************************************************)

(**********)
Axiom Rmult_sym:(r1,r2:R)``r1*r2==r2*r1``.
Hints Resolve Rmult_sym : real v62.

(**********)
Axiom Rmult_assoc:(r1,r2,r3:R)``(r1*r2)*r3==r1*(r2*r3)``.
Hints Resolve Rmult_assoc : real v62.

(**********)
Axiom Rinv_l:(r:R)``r<>0``->``(1/r)*r==1``.
Hints Resolve Rinv_l : real.


(**********)
Axiom Rmult_1l:(r:R)``1*r==r``.
Hints Resolve Rmult_1l : real v62.

(**********)
Axiom R1_neq_R0:``1<>0``.
Hints Resolve R1_neq_R0 : real.

(*********************************************************)
(*s     Distributivity                                   *)
(*********************************************************)

(**********)
Axiom Rmult_Rplus_distr:(r1,r2,r3:R)``r1*(r2+r3)==(r1*r2)+(r1*r3)``.
Hints Resolve Rmult_Rplus_distr : real v62.
```

```
(*******************************************************)
(*      Order axioms                                   *)
(*******************************************************)
(*******************************************************)
(*s     Total Order                                    *)
(*******************************************************)

(**********)
Axiom total_order_T:(r1,r2:R)(sumorT (sumboolT ``r1<r2`` r1==r2)
                                              ``r1>r2``).

(*******************************************************)
(*s     Lower                                          *)
(*******************************************************)

(**********)
Axiom Rlt_antisym:(r1,r2:R)``r1<r2`` -> ~ ``r2<r1``.

(**********)
Axiom Rlt_trans:(r1,r2,r3:R)
  ``r1<r2``->``r2<r3``->``r1<r3``.

(**********)
Axiom Rlt_compatibility:(r,r1,r2:R)``r1<r2``->``r+r1<r+r2``.

(**********)
Axiom Rlt_monotony:(r,r1,r2:R)``0<r``->``r1<r2``->``r*r1<r*r2``.

Hints Resolve Rlt_antisym Rlt_compatibility Rlt_monotony : real.

(********************************************************)
(*s     Injection from N to R                           *)
(********************************************************)

(**********)
Fixpoint INR [n:nat]:R:=(Cases n of
                          O      => ``0``
                         |(S O)  => ``1``
                         |(S n)  => ``(INR n)+1``
                        end).

(********************************************************)
(*s     Injection from Z to R                           *)
(********************************************************)

(**********)
Definition IZR:Z->R:=[z:Z](Cases z of
                          ZERO      => ``0``
                        |(POS n) => (INR (convert n))
                        |(NEG n) => ``-(INR (convert n))``
```

```
                        end).

(*******************************************************)
(*s      R Archimedian                                  *)
(*******************************************************)

(**********)
Axiom archimed:(r:R)``(IZR (up r)) > r``/\``(IZR (up r))-r <= 1``.

(*******************************************************)
(*s      R Complete                                      *)
(*******************************************************)

(**********)
Definition is_upper_bound:=[E:R->Prop][m:R](x:R)(E x)->``x <= m``.

(**********)
Definition bound:=[E:R->Prop](ExT [m:R](is_upper_bound E m)).

(**********)
Definition is_lub:=[E:R->Prop][m:R]
    (is_upper_bound E m)/\(b:R)(is_upper_bound E b)->``m <= b``.

(**********)
Axiom complet:(E:R->Prop)(bound E)->
              (ExT [x:R] (E x))->
              (ExT [m:R](is_lub E m)).
```

# Annexe B

# Lemmes de base sur ℝ

```
(**********************************************************************)
(*  v      *   The Coq Proof Assistant  /  The Coq Development Team    *)
(* <O___,, *        INRIA-Rocquencourt  &  LRI-CNRS-Orsay              *)
(*   \VV/  **************************************************************)
(*    //   *      This file is distributed under the terms of the      *)
(*         *       GNU Lesser General Public License Version 2.1       *)
(**********************************************************************)
(********************************************************************)
(*             Basic lemmas for the classical reals numbers         *)
(********************************************************************)
(********************************************************************)
(*   Relation between orders and equality                           *)
(********************************************************************)

Lemma Rlt_antirefl:(r:R)~``r<r``.

Lemma Rlt_not_eq:(r1,r2:R)``r1<r2``->``r1<>r2``.

Lemma Rgt_not_eq:(r1,r2:R)``r1>r2``->``r1<>r2``.

Lemma imp_not_Req:(r1,r2:R)(``r1<r2``\/ ``r1>r2``) -> ``r1<>r2``.

Lemma Req_EM:(r1,r2:R)(r1==r2)\/``r1<>r2``.

Lemma total_order:(r1,r2:R)``r1<r2``\/(r1==r2)\/``r1>r2``.

Lemma not_Req:(r1,r2:R)``r1<>r2``->(``r1<r2``\/``r1>r2``).

(******************************************************************)
(*        Order Lemma  : relating [<], [>], [<=] and [>=]          *)
(******************************************************************)

Lemma Rlt_le:(r1,r2:R)``r1<r2``-> ``r1<=r2``.

Lemma Rle_ge : (r1,r2:R)``r1<=r2`` -> ``r2>=r1``.

Lemma Rge_le : (r1,r2:R)``r1>=r2`` -> ``r2<=r1``.
```

```
Lemma not_Rle:(r1,r2:R)~(``r1<=r2``)->``r1>r2``.

Lemma Rlt_le_not:(r1,r2:R)``r2<r1``->~(``r1<=r2``).

Lemma Rle_not:(r1,r2:R)``r1>r2`` -> ~(``r1<=r2``).

Lemma Rle_not_lt: (r1, r2:R) ``r2 <= r1`` ->~ (``r1<r2``).

Lemma Rlt_ge_not:(r1,r2:R)``r1<r2``->~(``r1>=r2``).

Lemma eq_Rle:(r1,r2:R)r1==r2->``r1<=r2``.

Lemma eq_Rge:(r1,r2:R)r1==r2->``r1>=r2``.

Lemma eq_Rle_sym:(r1,r2:R)r2==r1->``r1<=r2``.

Lemma eq_Rge_sym:(r1,r2:R)r2==r1->``r1>=r2``.

Lemma Rle_antisym : (r1,r2:R)``r1<=r2`` -> ``r2<=r1``-> r1==r2.

Lemma Rle_le_eq:(r1,r2:R)(``r1<=r2``/\``r2<=r1``)<->(r1==r2).

Lemma Rle_trans:(r1,r2,r3:R) ``r1<=r2``->``r2<=r3``->``r1<=r3``.

Lemma Rle_lt_trans:(r1,r2,r3:R)``r1<=r2``->``r2<r3``->``r1<r3``.

Lemma Rlt_le_trans:(r1,r2,r3:R)``r1<r2``->``r2<=r3``->``r1<r3``.

Lemma total_order_Rlt:(r1,r2:R)(sumboolT ``r1<r2`` ~(``r1<r2``)).

Lemma total_order_Rle:(r1,r2:R)(sumboolT ``r1<=r2`` ~(``r1<=r2``)).

Lemma total_order_Rgt:(r1,r2:R)(sumboolT ``r1>r2`` ~(``r1>r2``)).

Lemma total_order_Rge:(r1,r2:R)(sumboolT (``r1>=r2``) ~(``r1>=r2``)).

Lemma total_order_Rlt_Rle:(r1,r2:R)(sumboolT ``r1<r2`` ``r2<=r1``).

Lemma Rle_or_lt: (n, m:R)(Rle n m) \/ (Rlt m n).

Lemma total_order_Rle_Rlt_eq :(r1,r2:R)``r1<=r2``->
                              (sumboolT ``r1<r2`` ``r1==r2``).

Lemma inser_trans_R:(n,m,p,q:R)``n<=m<p``->
                    (sumboolT ``n<=m<q`` ``q<=m<p``).

(*****************************************************************)
(*       Field Lemmas                                            *)
(* This part contains lemma involving the Fields operations     *)
(*****************************************************************)
(*******************************************************)
(*      Addition                                       *)
(*******************************************************)
```

```
Lemma Rplus_ne:(r:R)``r+0==r``/\``0+r==r``.

Lemma Rplus_Or:(r:R)``r+0==r``.

Lemma Rplus_Ropp_l:(r:R)``(-r)+r==0``.

Lemma Rplus_Ropp:(x,y:R)``x+y==0``->``y== -x``.

Lemma Rplus_plus_r:(r,r1,r2:R)(r1==r2)->``r+r1==r+r2``.

Lemma r_Rplus_plus:(r,r1,r2:R)``r+r1==r+r2``->r1==r2.

Lemma Rplus_ne_i:(r,b:R)``r+b==r`` -> ``b==0``.

(*********************************************************)
(*       Multiplication                                  *)
(*********************************************************)

Lemma Rinv_r:(r:R)``r<>0``->``r* (/r)==1``.

Lemma Rinv_l_sym:(r:R)``r<>0``->``1==(/r) * r``.

Lemma Rinv_r_sym:(r:R)``r<>0``->``1==r* (/r)``.

Lemma Rmult_Or :(r:R) ``r*0==0``.

Lemma Rmult_Ol:(r:R)(``0*r==0``).

Lemma Rmult_ne:(r:R)``r*1==r``/\``1*r==r``.

Lemma Rmult_1r:(r:R)(``r*1==r``).

Lemma Rmult_mult_r:(r,r1,r2:R)r1==r2->``r*r1==r*r2``.

Lemma r_Rmult_mult:(r,r1,r2:R)(``r*r1==r*r2``)->``r<>0``->(r1==r2).

Lemma without_div_Od:(r1,r2:R)``r1*r2==0`` -> ``r1==0`` \/ ``r2==0``.

Lemma without_div_Oi:(r1,r2:R) ``r1==0``\/``r2==0`` -> ``r1*r2==0``.

Lemma without_div_Oi1:(r1,r2:R) ``r1==0`` -> ``r1*r2==0``.

Lemma without_div_Oi2:(r1,r2:R) ``r2==0`` -> ``r1*r2==0``.

Lemma without_div_O_contr:(r1,r2:R)``r1*r2<>0`` ->
                         ``r1<>0`` /\ ``r2<>0``.

Lemma mult_non_zero :(r1,r2:R)``r1<>0`` /\ ``r2<>0`` -> ``r1*r2<>0``.

Lemma Rmult_Rplus_distrl:
   (r1,r2,r3:R) ``(r1+r2)*r3 == (r1*r3)+(r2*r3)``.

Definition Rsqr:R->R:=[r:R]``r*r``.
```

```
Lemma Rsqr_O:(Rsqr ``0``)==``0``.

Lemma Rsqr_r_R0:(r:R)(Rsqr r)==``0``->``r==0``.

(*********************************************************)
(*      Opposite                                         *)
(*********************************************************)

Lemma eq_Ropp:(r1,r2:R)(r1==r2)->``-r1 == -r2``.

Lemma Ropp_O:``-0==0``.

Lemma eq_RoppO:(r:R)``r==0``-> ``-r==0``.

Lemma Ropp_Ropp:(r:R)``-(-r)==r``.

Lemma Ropp_neq:(r:R)``r<>0``->``-r<>0``.

Lemma Ropp_distr1:(r1,r2:R)``-(r1+r2)==(-r1 + -r2)``.

Lemma Ropp_mul1:(r1,r2:R)``(-r1)*r2 == -(r1*r2)``.

Lemma Ropp_mul2:(r1,r2:R)``(-r1)*(-r2)==r1*r2``.

Lemma minus_R0:(r:R)``r-0==r``.

Lemma Rminus_Ropp:(r:R)``0-r==-r``.

Lemma Ropp_distr2:(r1,r2:R)``-(r1-r2)==r2-r1``.

Lemma Ropp_distr3:(r1,r2:R)``-(r2-r1)==r1-r2``.

Lemma eq_Rminus:(r1,r2:R)(r1==r2)->``r1-r2==0``.

Lemma Rminus_eq:(r1,r2:R)``r1-r2==0`` -> r1==r2.

Lemma Rminus_eq_right:(r1,r2:R)``r2-r1==0`` -> r1==r2.

Lemma Rplus_Rminus: (p,q:R)``p+(q-p)``==q.

Lemma Rminus_eq_contra:(r1,r2:R)``r1<>r2``->``r1-r2<>0``.

Lemma Rminus_not_eq:(r1,r2:R)``r1-r2<>0``->``r1<>r2``.

Lemma Rminus_not_eq_right:(r1,r2:R)``r2-r1<>0`` -> ``r1<>r2``.

Lemma Rminus_distr:  (x,y,z:R) ``x*(y-z)==(x*y) - (x*z)``.

Lemma Rinv_R1:``/1==1``.

Lemma Rinv_neq_R0:(r:R)``r<>0``->``(/r)<>0``.

Lemma Rinv_Rinv:(r:R)``r<>0``->``/(/r)==r``.
```

```
Lemma Rinv_Rmult:(r1,r2:R)``r1<>0``->``r2<>0``->
                         ``/(r1*r2)==(/r1)*(/r2)``.

Lemma Ropp_Rinv:(r:R)``r<>0``->``-(/r)==/(-r)``.

Lemma Rinv_r_simpl_r : (r1,r2:R)``r1<>0``->``r1*(/r1)*r2==r2``.

Lemma Rinv_r_simpl_l : (r1,r2:R)``r1<>0``->``r2*r1*(/r1)==r2``.

Lemma Rinv_r_simpl_m : (r1,r2:R)``r1<>0``->``r1*r2*(/r1)==r2``.

Lemma Rinv_Rmult_simpl:(a,b,c:R)``a<>0``->``(a*(/b))*(c*(/a))==c*(/b)``.

Lemma Rlt_compatibility_r:(r,r1,r2:R)``r1<r2``->``r1+r<r2+r``.

Lemma Rlt_anti_compatibility:  (r,r1,r2:R)``r+r1 < r+r2`` -> ``r1<r2``.

Lemma Rle_compatibility:(r,r1,r2:R)``r1<=r2`` -> ``r+r1 <= r+r2 ``.

Lemma Rle_compatibility_r:(r,r1,r2:R)``r1<=r2`` -> ``r1+r<=r2+r``.

Lemma Rle_anti_compatibility: (r,r1,r2:R)``r+r1<=r+r2`` -> ``r1<=r2``.

Lemma sum_inequa_Rle_lt:(a,x,b,c,y,d:R)``a<=x`` -> ``x<b`` ->
         ``c<y`` -> ``y<=d`` -> ``a+c < x+y < b+d``.

Lemma Rplus_lt:(r1,r2,r3,r4:R)``r1<r2`` -> ``r3<r4`` ->
                             ``r1+r3 < r2+r4``.

Lemma Rplus_le:(r1,r2,r3,r4:R)``r1<=r2`` -> ``r3<=r4`` ->
                             ``r1+r3 <= r2+r4``.

Lemma Rplus_lt_le_lt:(r1,r2,r3,r4:R)``r1<r2`` -> ``r3<=r4`` ->
                    ``r1+r3 < r2+r4``.

Lemma Rplus_le_lt_lt:(r1,r2,r3,r4:R)``r1<=r2`` -> ``r3<r4`` ->
                                    ``r1+r3 < r2+r4``.

Lemma Rgt_Ropp:(r1,r2:R) ``r1 > r2`` -> ``-r1 < -r2``.

Lemma Rlt_Ropp:(r1,r2:R) ``r1 < r2`` -> ``-r1 > -r2``.

Lemma Ropp_Rlt: (x,y:R) ``-y < -x`` ->``x<y``.

Lemma Rlt_Ropp1:(r1,r2:R) ``r2 < r1`` -> ``-r1 < -r2``.

Lemma Rle_Ropp:(r1,r2:R) ``r1 <= r2`` -> ``-r1 >= -r2``.

Lemma Ropp_Rle: (x,y:R) ``-y <= -x`` ->``x <= y``.

Lemma Rle_Ropp1:(r1,r2:R) ``r2 <= r1`` -> ``-r1 <= -r2``.

Lemma Rge_Ropp:(r1,r2:R) ``r1 >= r2`` -> ``-r1 <= -r2``.
```

```
Lemma Rlt_RO_Ropp:(r:R) ``0 < r`` -> ``0 > -r``.

Lemma Rgt_RO_Ropp:(r:R) ``0 > r`` -> ``0 < -r``.

Lemma Rle_RO_Ropp:(r:R) ``0 <= r`` -> ``0 >= -r``.

Lemma Rge_RO_Ropp:(r:R) ``0 >= r`` -> ``0 <= -r``.

Lemma  Rlt_monotony_r:(r,r1,r2:R)``0<r`` -> ``r1 < r2`` ->
                   ``r1*r < r2*r``.

Lemma Rlt_monotony_contra:
  (x, y, z:R) ``0<z`` ->``z*x<z*y`` ->``x<y``.

Lemma Rlt_anti_monotony:(r,r1,r2:R)``r < 0`` -> ``r1 < r2`` ->
                                ``r*r1 > r*r2``.

Lemma Rle_monotony:
  (r,r1,r2:R)``0 <= r`` -> ``r1 <= r2`` -> ``r*r1 <= r*r2``.

Lemma Rle_monotony_r:
  (r,r1,r2:R)``0 <= r`` -> ``r1 <= r2`` -> ``r1*r <= r2*r``.

Lemma Rle_monotony_contra:
  (x, y, z:R) ``0<z`` ->``z*x<=z*y`` ->``x<=y``.

Lemma Rle_anti_monotony
       :(r,r1,r2:R)``r <= 0`` -> ``r1 <= r2`` -> ``r*r1 >= r*r2``.

Lemma Rle_Rmult_comp:(x, y, z, t:R) ``0 <= x`` -> ``0 <= z`` ->
                   ``x <= y`` -> ``z <= t`` -> ``x*z <= y*t``.

Lemma Rmult_lt:(r1,r2,r3,r4:R)``r3>0`` -> ``r2>0`` ->
  `` r1 < r2`` -> ``r3 < r4`` -> ``r1*r3 < r2*r4``.

Lemma Rlt_minus:(r1,r2:R)``r1 < r2`` -> ``r1-r2 < 0``.

Lemma Rle_minus:(r1,r2:R)``r1 <= r2`` -> ``r1-r2 <= 0``.

Lemma Rminus_lt:(r1,r2:R)``r1-r2 < 0`` -> ``r1 < r2``.

Lemma Rminus_le:(r1,r2:R)``r1-r2 <= 0`` -> ``r1 <= r2``.

Lemma tech_Rplus:(r,s:R)``0<=r`` -> ``0<s`` -> ``r+s<>0``.

Lemma pos_Rsqr:(r:R)``0<=(Rsqr r)``.

Lemma pos_Rsqr1:(r:R)``r<>0``->``0<(Rsqr r)``.

Lemma Rlt_RO_R1:``0<1``.

Lemma Rlt_Rinv:(r:R)``0<r``->``0</r``.
```

```
Lemma Rlt_Rinv2:(r:R)``r < 0``->``/r < 0``.

Lemma Rlt_monotony_rev:
  (r,r1,r2:R)``0<r`` -> ``r*r1 < r*r2`` -> ``r1 < r2``.

Lemma Rinv_lt:(r1,r2:R)``0 < r1*r2`` -> ``r1 < r2`` -> ``/r2 < /r1``.

Lemma Rlt_Rinv_R1: (x, y:R) ``1 <= x`` -> ``x<y`` ->``/y< /x``.

(*******************************************************)
(*      Greater                                        *)
(*******************************************************)

Lemma Rge_ge_eq:(r1,r2:R)``r1 >= r2`` -> ``r2 >= r1`` -> r1==r2.

Lemma Rlt_not_ge:(r1,r2:R)~(``r1<r2``)->``r1>=r2``.

Lemma Rgt_not_le:(r1,r2:R)~(``r1>r2``)->``r1<=r2``.

Lemma Rgt_ge:(r1,r2:R)``r1>r2`` -> ``r1 >= r2``.

Lemma Rlt_sym:(r1,r2:R)``r1<r2`` <-> ``r2>r1``.

Lemma Rle_sym1:(r1,r2:R)``r1<=r2``->``r2>=r1``.

Lemma Rle_sym2:(r1,r2:R)``r2>=r1`` -> ``r1<=r2``.

Lemma Rle_sym:(r1,r2:R)``r1<=r2``<->``r2>=r1``.

Lemma Rge_gt_trans:(r1,r2,r3:R)``r1>=r2``->``r2>r3``->``r1>r3``.

Lemma Rgt_ge_trans:(r1,r2,r3:R)``r1>r2`` -> ``r2>=r3`` -> ``r1>r3``.

Lemma Rgt_trans:(r1,r2,r3:R)``r1>r2`` -> ``r2>r3`` -> ``r1>r3``.

Lemma Rge_trans:(r1,r2,r3:R)``r1>=r2`` -> ``r2>=r3`` -> ``r1>=r3``.

Lemma Rgt_RoppO:(r:R)``r>0``->``(-r)<0``.

Lemma Rlt_RoppO:(r:R)``r<0``->``-r>0``.

Lemma Rlt_r_plus_R1:(r:R)``0<=r`` -> ``0<r+1``.

Lemma Rlt_r_r_plus_R1:(r:R)``r<r+1``.

Lemma tech_Rgt_minus:(r1,r2:R)``0<r2``->``r1>r1-r2``.

Lemma Rgt_plus_plus_r:(r,r1,r2:R)``r1>r2``->``r+r1 > r+r2``.

Lemma Rgt_r_plus_plus:(r,r1,r2:R)``r+r1 > r+r2`` -> ``r1 > r2``.

Lemma Rge_plus_plus_r:(r,r1,r2:R)``r1>=r2`` -> ``r+r1 >= r+r2``.

Lemma Rge_r_plus_plus:(r,r1,r2:R)``r+r1 >= r+r2`` -> ``r1>=r2``.
```

```
Lemma Rge_monotony:
 (x,y,z:R) ``z>=0`` -> ``x>=y`` -> ``x*z >= y*z``.

Lemma Rgt_minus:(r1,r2:R)``r1>r2`` -> ``r1-r2 > 0``.

Lemma minus_Rgt:(r1,r2:R)``r1-r2 > 0`` -> ``r1>r2``.

Lemma Rge_minus:(r1,r2:R)``r1>=r2`` -> ``r1-r2 >= 0``.

Lemma minus_Rge:(r1,r2:R)``r1-r2 >= 0`` -> ``r1>=r2``.

Lemma Rmult_gt:(r1,r2:R)``r1>0`` -> ``r2>0`` -> ``r1*r2>0``.

Lemma Rmult_lt_0:(r1,r2,r3,r4:R)``r3>=0``->``r2>0``->
               ``r1<r2``->``r3<r4``->``r1*r3<r2*r4``.

Lemma Rmult_lt_pos:(x,y:R)``0<x`` -> ``0<y`` -> ``0<x*y``.

Lemma Rplus_eq_R0_l:(a,b:R)``0<=a`` -> ``0<=b`` ->
                          ``a+b==0`` -> ``a==0``.

Lemma Rplus_eq_R0:(a,b:R)``0<=a`` -> ``0<=b`` ->
                        ``a+b==0`` -> ``a==0``/\``b==0``.

Lemma Rplus_Rsr_eq_R0_l:(a,b:R)``(Rsqr a)+(Rsqr b)==0``->``a==0``.

Lemma Rplus_Rsr_eq_R0:(a,b:R)``(Rsqr a)+(Rsqr b)==0``->
                            ``a==0``/\``b==0``.

(*******************************************************)
(*      Injection from N to R                          *)
(*******************************************************)

Lemma S_INR:(n:nat)(INR (S n))==``(INR n)+1``.

Lemma S_O_plus_INR:(n:nat)
    (INR (plus (S O) n))==``(INR (S O))+(INR n)``.

Lemma plus_INR:(n,m:nat)(INR (plus n m))==``(INR n)+(INR m)``.

Lemma minus_INR:(n,m:nat)(le m n)->
       (INR (minus n m))==``(INR n)-(INR m)``.

Lemma mult_INR:(n,m:nat)(INR (mult n m))==(Rmult (INR n) (INR m)).

Lemma lt_INR_0:(n:nat)(lt O n)->``0 < (INR n)``.

Lemma lt_INR:(n,m:nat)(lt n m)->``(INR n) < (INR m)``.

Lemma INR_lt_1:(n:nat)(lt (S O) n)->``1 < (INR n)``.

Lemma INR_pos : (p:positive)``0<(INR (convert p))``.
```

```
Lemma pos_INR:(n:nat)``0 <= (INR n)``.

Lemma INR_lt:(n,m:nat)``(INR n) < (INR m)``->(lt n m).

Lemma le_INR:(n,m:nat)(le n m)->``(INR n)<=(INR m)``.

Lemma not_INR_O:(n:nat)``(INR n)<>0``->~n=O.

Lemma not_O_INR:(n:nat)~n=O->``(INR n)<>0``.

Lemma not_nm_INR:(n,m:nat)~n=m->``(INR n)<>(INR m)``.

Lemma INR_eq: (n,m:nat)(INR n)==(INR m)->n=m.

Lemma INR_le: (n, m : nat) (Rle (INR n) (INR m)) ->  (le n m).

Lemma not_1_INR:(n:nat)~n=(S O)->``(INR n)<>1``.

(**********************************************************)
(*      Injection from Z to R                             *)
(**********************************************************)

Definition INZ:=inject_nat.

Lemma IZN:(z:Z)(`0<=z`)->(Ex [n:nat] z=(INZ n)).

Lemma INR_IZR_INZ:(n:nat)(INR n)==(IZR (INZ n)).

Lemma plus_IZR_NEG_POS : (p,q:positive)
   (IZR `(POS p)+(NEG q)`)==``(IZR (POS p))+(IZR (NEG q))``.

Lemma plus_IZR:(z,t:Z)(IZR `z+t`)==``(IZR z)+(IZR t)``.

Lemma mult_IZR:(z,t:Z)(IZR `z*t`)==``(IZR z)*(IZR t)``.

Lemma Ropp_Ropp_IZR:(z:Z)(IZR (`-z`))==``-(IZR z)``.

Lemma Z_R_minus:(z1,z2:Z)``(IZR z1)-(IZR z2)``==(IZR `z1-z2`).

Lemma lt_O_IZR:(z:Z)``0 < (IZR z)``->`0<z`.

Lemma lt_IZR:(z1,z2:Z)``(IZR z1)<(IZR z2)``->`z1<z2`.

Lemma eq_IZR_R0:(z:Z)``(IZR z)==0``->`z=0`.

Lemma eq_IZR:(z1,z2:Z)(IZR z1)==(IZR z2)->z1=z2.

Lemma not_O_IZR:(z:Z)`z<>0`->``(IZR z)<>0``.

Lemma le_O_IZR:(z:Z)``0<= (IZR z)``->`0<=z`.

Lemma le_IZR:(z1,z2:Z)``(IZR z1)<=(IZR z2)``->`z1<=z2`.

Lemma le_IZR_R1:(z:Z)``(IZR z)<=1``-> `z<=1`.
```

```
Lemma IZR_ge: (m,n:Z) `m>= n` -> ``(IZR m)>=(IZR n)``.

Lemma IZR_le: (m,n:Z) `m<= n` -> ``(IZR m)<=(IZR n)``.

Lemma IZR_lt: (m,n:Z) `m< n` -> ``(IZR m)<(IZR n)``.

Lemma one_IZR_lt1 : (z:Z)``-1<(IZR z)<1``->`z=0`.

Lemma one_IZR_r_R1 : (r:R)(z,x:Z)``r<(IZR z)<=r+1``->
                     ``r<(IZR x)<=r+1``->z=x.

Lemma single_z_r_R1:
   (r:R)(z,x:Z)``r<(IZR z)``->``(IZR z)<=r+1``->``r<(IZR x)``->
        ``(IZR x)<=r+1``->z=x.

Lemma tech_single_z_r_R1
        :(r:R)(z:Z)``r<(IZR z)``->``(IZR z)<=r+1``
         -> (Ex [s:Z] (~s=z/\``r<(IZR s)``/\``(IZR s)<=r+1``))->False.
```

# Annexe C

# O∂yssée : Programmes complets du chapitre 5

## C.1 Exemple *exple*

Programme FORTRAN donné en entrée :

```
      subroutine exple (x,y,z)
      z=y*z
      y=x*z
      x=2*x
      end
```

Commande O∂yssée pour différentier par rapport à `x` :

```
diff -head exple -vars x -tl -o std
```

Programme fourni par O∂yssée :

```
COD Compilation unit : expletl
COD Derivative of unit :  exple
COD Dummys:  x y z
COD Active IN   dummys:  x
COD Active OUT  dummys:  x y
COD Dependencies between IN and OUT:
COD x <--   x
COD y <--   x


      SUBROUTINE EXPLETL (X, Y, Z, XTTL, YTTL)

      IMPLICIT NONE
      REAL X
      REAL Y
      REAL Z
```

```
      REAL YTTL
      REAL XTTL

      Z = Y*Z
      YTTL = Z*XTTL
      Y = X*Z
      XTTL = 2*XTTL
      X = 2*X
      END
```

Commande O∂yssée pour différentier par rapport à `x` et `y` :

```
diff -head exple -vars x y -tl -o std
```

Programme fourni par O∂yssée :

```
COD Compilation unit : expletl
COD Derivative of unit :  exple
COD Dummys:  x y z
COD Active IN   dummys:  x y
COD Active OUT  dummys:  x y z
COD Dependencies between IN and OUT:
COD x <--   x
COD y <--   x y
COD z <--   y


      SUBROUTINE EXPLETL (X, Y, Z, XTTL, YTTL, ZTTL)

      IMPLICIT NONE
      REAL ZTTL
      REAL X
      REAL Y
      REAL Z
      REAL YTTL
      REAL XTTL

      ZTTL = Z*YTTL
      Z = Y*Z
      YTTL = ZTTL*X+Z*XTTL
      Y = X*Z
      XTTL = 2*XTTL
      X = 2*X
      END
```

## C.2 Exemple g (*Valeur Absolue*)

Programme FORTRAN donné en entrée :

```
      subroutine g (x,y)
       if (x.ge.0.) then
        y = x
       else
        y = -x
       end if
      end
```

Commande O∂yssée pour différentier par rapport à `x` :

```
diff -head g -vars x -tl -o std
```

Programme fourni par O∂yssée :

```
COD Compilation unit : gtl
COD Derivative of unit :  g
COD Dummys:  x y
COD Active IN   dummys:  x
COD Active OUT  dummys:  y
COD Dependencies between IN and OUT:
COD y <--   x


      SUBROUTINE GTL (X, Y, XTTL, YTTL)

      IMPLICIT NONE
      REAL X
      REAL Y
      REAL YTTL
      REAL XTTL

      IF (X.GE.0.) THEN
        YTTL = XTTL
        Y = X
      ELSE
        YTTL = -XTTL
        Y = -X
      END IF
      END
```

# Annexe D

# Quelques notions de Coq

Cet annexe n'est pas une présentation de `Coq` mais plutôt un répertoire des notions qui sont utilisées dans cette thèse. De nombreuses présentations du système `Coq`, auxquelles nous pourrons nous référer pour plus de détails, ont déjà été écrites [4, 20, 8].

## D.1 Préliminaires

### D.1.1 Vocabulaire

Pour les lecteurs ne connaissant pas le système `Coq`, nous allons présenter quelques termes propres au système :

- Nous utiliserons le mot *sorte* pour désigner "les types des types". Il y en a trois en `Coq` : `Prop`, `Set` et `Type`.
- Nous parlerons de *but* et de *sous-but* lorsqu'il s'agit du mode interactif. Il s'agit d'un raisonnement arrière. La conclusion est le but et les prémisses sont les sous-buts.
- Une *tactique* implante le raisonnement arrière. Lorsqu'une tactique est appliquée, elle remplace le but par le sous-but généré. Les tactiques de `Coq` peuvent être comparées aux *strategies* de `PVS`.

### D.1.2 Notations

Le tableau suivant établit la correspondance entre la syntaxe usuelle mathématique ou logique et la syntaxe de `Coq` :

| | | |
|---|---|---|
| $\perp$ | `False` | Faux |
| $\top$ | `True` | Vrai |
| $\neg$ | `~` | Non |
| $\vee$ | `\/` | Ou (dans `Prop`) |
| $\vee$ | {...}+{...} | Ou (dans `Set`) |
| $\wedge$ | `/\` | Et |
| $\Rightarrow$ | `->` | Implique |
| $\Leftrightarrow$ | `<->` | Si et seulement si |
| $\forall x : A$ | `(x :A)` | Pour tout x de type A |
| $\exists x : A, ...$ | `(Ex [x :A] ...)` | Il existe x de type A tel que (dans `Prop`) |
| $\exists x : A, ...$ | `{x :A \|...}` | Il existe x de type A tel que (dans `Set`) |
| $\lambda x : A.$ | `[x :A]` | Lambda abstraction |

Les règles d'affichage (*pretty print*) pour les nombres sont différentes pour les nombres suivant qu'ils sont de type entier, relatif ou réel. Ces règles sont implantées directement en Coq, grâce au "prettyprinter" et, éventuellement, en Ocaml (pour introduire des compteurs, par exemple), ce qui est le cas pour les nombres.
Le tableau suivant donne la correspondance entre l'arbre de syntaxe abstraite (AST) et l'affichage. Nous prendrons les exemples de la constante 3 et de l'addition $3 + x$ à valeur dans les trois types `nat` ($\mathbb{N}$), `Z` ($\mathbb{Z}$) et `R` ($\mathbb{R}$) :

| | | |
|---|---|---|
| `nat` | 3 | (3) |
| `nat` | $3 + x$ | (plus (3) x) |
| `Z` | 3 | '3' |
| `Z` | $3 + x$ | '3+x' |
| `R` | 3 | "3" |
| `R` | $3 + x$ | "3+x" |

Etant donné que nous allons donner des exemples pris directement du mode interactif de Coq (toplevel), nous allons en faire une présentation générale. Les hypothèses apparaissent nommées au-dessus d'un double trait et la propriété à montrer (appelée le *but*) en dessous. S'il y a plusieurs buts (par exemple après la séparation d'un ET logique) ils apparaissent en dessous. Seules les hypothèses du but courant sont visibles en permanence, les hypothèses des autres buts restant cachées, sauf avis contraire de l'utilisateur (commande `Show` ou `Focus`). Voici un exemple où il reste deux buts à montrer :

```
nom_du_lemme < Tactique.
2 subgoals

 P : ....
 H : ....
 x : ....
 ============================
   ......

subgoal 2 is:
   ......

nom_du_lemme <
```

## D.2 La logique et les mathématiques

Tout système de preuves formelles repose sur un système logique précis. Nous nous intéresserons particulièrement aux logiques *classique* et *intuitionniste*.

La logique classique est caractérisée par un axiome, appelé *axiome du tiers exclu* : pour toute proposition $A$, la proposition "$A$ *ou non* $A$" est vraie. C'est cette logique qui est utilisée pour montrer (valider) toutes les propriétés mathématiques usuelles de l'analyse standard telles qu'elles sont énoncées dans [57, 15, 12, 27], par exemple. En effet, il est toujours possible de donner une valeur de vérité à une proposition dans un modèle du système logique, mais nous ne pourrons pas forcément établir une preuve. Nous pouvons, par exemple, montrer (valider) que toute fonction des entiers dans les entiers admet un minimum. Supposons l'existence d'une fonction $f$ des entiers dans les entiers n'ayant pas de minimum. On peut alors construire une suite $(S_i)_{i\in}$ $\mathbb{N}$ d'entiers infinie et strictement décroissante. On prend $S_0 = f(0)$ et $S_0$ n'étant pas le minimum de $f$, il existe $k$ tel que $f(k) < S_0$. On prend $S_1 = f(k)$ et ainsi de suite on construit la suite $(S_i)$ par récurrence, ce qui nous donne une contradiction, car il n'existe pas de suite infinie strictement décroissante sur les entiers. Nous obtenons la proposition voulue en utilisant la propriété que *non non* $A$ *implique* $A$, qui est quasi directement issue du tiers exclus. Comme nous avons pu le voir, cette preuve ne permet pas de donner un minimum, ni aucune autre puisque ce problème est indécidable. Nous venons de voir un exemple de propriété qui n'est montrable que de manière classique, puisqu'il est impossible de trouver une fonction qui donnerait un minimum (qui serait une preuve et non plus une validation).
Les systèmes HOL et PVS, par exemple, reposent sur cette logique.

La logique intuitionniste diffère de la précédente par le fait que l'axiome du tiers exclus n'est pas valide. De ce fait, si nous reprenons l'exemple développé précédemment, nous voyons que nous pouvons montrer qu'il n'existe pas de fonction des entiers dans les entiers qui n'admette pas de minimum, mais nous ne pouvons en déduire la propriété telle que nous l'avons énoncée à l'origine. Ainsi, prouver la proposition $A \vee B$ signifie que l'on est capable de savoir si l'on se trouve dans le cas où $A$ est vrai ou bien $B$, tout comme prouver une proposition de la forme *il existe* $x$ *tel que*... signifie que l'on sait construire[1] un tel $x$.

Le système Coq s'appuie sur cette logique. C'est grâce à cette particularité que Coq possède un principe d'extraction, ce qui est impossible pour les "systèmes classiques". En effet, la logique constructive permet de faire une analogie entre preuves et programmes, appelée *isomorphisme de Curry-Howard*. Le type d'une preuve correspond exactement à la propriété montrée, en identifiant les principaux connecteurs logiques aux opérateur appropriés du langage de programmation. Le principe d'extraction de Coq consiste donc en l'obtention d'un programme Ocaml à partir du terme preuve obtenu puisque nous sommes en logique constructive. On pourra se référer à [70] pour les détails concernant ce principe.

Le formalisme intuitionniste de Coq (Calcul des Constructions Inductives [17, 18, 19]) accompagné de son principe d'extraction, demande l'introduction d'une sorte particulière, la sorte `Set`.
Nous ne ferons pas ici une présentation de ces sortes. En effet, leur étude est très importante pour la formalisation des nombres réels. Pour cette raison, nous préférons en faire une présentation liée aux nombres réels, au chapitre 1.

[1] c'est la raison pour laquelle cette logique est aussi appelée logique constructive

## D.3 Représentation des nombres : raisonnement ou calcul

Dans un système de preuves formelles, la manière de représenter les nombres influence fortement les preuves qui seront faites par la suite sur ces nombres.
Sur ce sujet, nous comparerons deux systèmes dont les représentations sont différentes :
- PVS : représentation interne (type primitif)
- Coq : définition dans le langage du système

En PVS, les entiers naturels et relatifs, ainsi que les nombres rationnels sont, en fait, les nombres de Lisp, le langage d'implantation de PVS.
En Coq, les nombres sont définis en tant que types. Les entiers naturels de Peano, par exemple, sont représentés par le type inductif à deux constructeurs (zéro et successeur) suivant :

```
Inductive nat : Set := O : nat
                     | S : nat->nat.
```

Ces deux formalisations sont différentes à plusieurs points de vue. Les preuves formelles sont basées sur la notion de *raisonnement*. Faire une preuve signifie donc trouver toutes les étapes de raisonnement qui conduisent au résultat. Néanmoins, l'on souhaite également profiter, tout comme en mathématique, de la notion de *calcul*. Considérons l'égalité $2+2=4$. Dans ce genre de cas, il est inintéressant (et fastidieux) de détailler la preuve. Nous préférons, dans la mesure du possible, utiliser la puissance du calcul que l'on trouve dans tout langage de programmation. Dans le cas d'un système comme PVS, qui s'apparente plutôt à un outil décisionnel, l'utilisation du calcul ne pose pas de problème pour les nombres rationnels[2] puisque ce sont les rationnels de Lisp, ce qui permet d'utiliser le principe de calcul de ce langage. En Coq, il ne s'agit pas seulement de valider des propriétés mais également de construire une preuve qui les démontre. Pour cela, en reprenant l'égalité $2 + 2 = 4$, la preuve se fait soit par réduction, soit par réécriture en utilisant les propriétés usuelles de commutativité, associativité... Cependant, on cherche à remplacer ces étapes par du calcul, par exemple en donnant une règle de calcul plutôt qu'un axiome [28].

### D.3.1 Egalité et Décidabilité

#### L'égalité de Coq

L'égalité de Coq est une égalité intentionnelle définie par le type inductif ci-dessous et correspondant à l'*égalité de Leibniz*, engendrée par l'axiome de réflexivité :

```
Inductive eq [A:Set; x:A] : A->Prop :=
  | refl_equal : (eq A x x).
```

Cette égalité est peu étendue en Coq par la notion de convertibilité (voir section D.5). En effet, deux termes seront également considérés égaux s'ils sont interconvertibles, c'est-à-dire s'ils se peuvent se réduire (après application des règles de conversion) sur le même terme irréductible, ce qui permet alors d'appliquer la réflexivité. Remarquons que, de ce fait, deux fonctions calculant le même résultat ne seront nécessairement pas égales[3] en Coq (deux fonctions de tri, par exemple).

[2] Nous nous intéressons aux nombres irrationnels en tant que nombres réels
[3] ce serait alors une égalité extentionnelle

### D.3.2 La décidabilité de l'égalité

Nous ne parlerons ici que de la décidabilité de l'égalité, car c'est cette notion qui nous intéressera particulièrement lors de la formalisation des nombres réel.

Cette notion est particulièrement simple à formaliser en Coq. En effet, étant donné que nous avons une sorte dédiée aux preuves constructives (`Set`), dont le modèle est la réalisabilité (voir le chapitre 1 pour plus de détails), dire que l'égalité définie sur un ensemble $A$ particulier est décidable revient à montrer que $\forall x, y \in A, x = y \textit{ ou } x \neq y$ avec *ou* à valeur dans la sorte `Set`, celle-ci ayant donc un contenu calculatoire. Bien évidemment nous ne pouvons montrer cette propriété pour tout $A$. Par contre, nous pouvons, par exemple, la montrer pour les entiers, les booléen..., mais pas pour les réels (cf. chapitre 1).

## D.4 Typage

### D.4.1 Elimination forte

Le principe d'élimination est lié au principe de définitions inductives. De par la nécessité de gérer l'information calculatoire, il existe plusieurs schémas d'élimination, issus d'une déclaration inductive et suivant la sorte dans laquelle le type est défini. Nous nous intéresserons en priorité au `Cases`, qui est un type inductif de Coq et que nous utiliserons souvent dans nos formalisations.

Lorsqu'un terme est défini à l'aide d'un `Cases`, l'idée de base pour le destructurer est d'avoir un objet $m$ du type inductif $I$ et de montrer une propriété $(P\ m)$, qui, en général, dépend de $m$. Pour cela, il suffit de montrer la propriété pour chaque $m = (c_i\ u_1...u_{pi})$ de chaque constructeur de $I$.

Le terme preuve est noté :

```
<P>Cases m of (c1 x11 ... x1p1) =>
        f1 ... (cn xn1 | ... | xnpn) => fn  end
```

Si $m$ est un terme construit à partir du constructeur $(c_i\ u_1...u_{pi})$ alors l'expression se comporte comme celle définie à la $i^{\text{ème}}$ branche et se réduit en $f_i$ où les $x_{i1}...x_{ipi}$ sont remplacés par $u_1...u_p$ (règle de $\iota$-réduction ci-après).

L'élimination construit un objet $(P\ m)$ à partir d'un objet $m$ de type $I$. De ce fait, toutes les combinaisons entres sortes ne sont pas autorisées.
De manière générale, on ne peut avoir $I$ de type `Prop` et $P$ de type $I \rightarrow$`Set`, car cela permettrait de construire une preuve ayant un contenu calculatoire à partir d'un objet ne contenant pas de contenu informatif, ce qui poserait de nombreux problèmes lors de l'extraction.

Nous appelons cette élimination particulière, qui donne la possibilité de calculer un type défini inductivement sur une structure de terme, l'*élimination forte*.
Nous ne donnerons pas les règles d'élimination que nous pourrons trouver dans [86]. Par contre, nous résumerons ces règles ainsi :

- La sorte `Prop` ne s'élimine que dans elle-même (un seul schéma de récurrence). Elle s'élimine par rapport à `Set` pour les types inductifs singletons (un seul constructeur) tel que l'égalité par exemple.

- Les sortes {`Prop`,`Set`} s'éliminent dans `Type` uniquement si le type inductif est un petit inductif, c'est-à-dire si chaque constructeur est de la forme `(x :T)P` avec `T` de type `Prop` ou `Set`.
- La sorte `Type` s'éliminent dans toutes les sortes (trois schémas de récurrence)
- La sorte `Set` s'élimine dans elle-même et dans `Prop` (deux schémas de récurrence)

### D.4.2 Cumulativité

Le principe de cumulativité est lié aux sortes et aux univers. Comme nous l'avons vu précédemment, le système `Coq` possède une hiérarchie d'univers indexés par des entiers $(\texttt{Type})_i$ et deux sortes `Prop` et `Set` correspondant à $\texttt{Type}_0$. La cumulativité est tout simplement le principe de "stratification"[4] des sortes, qui permet, par exemple, de dire que si un objet est de type `Set` alors il est aussi de type `Type`. Nous utiliserons, en particulier, ce principe dans le chapitre 4 pour généraliser des tactiques aux deux sortes.

### D.4.3 Coercions

Le principe de coercion [76] est lié au sous-typage et à l'injection. En particulier, une coercion se définit entre deux classes de `Coq`. Une classe est un terme dont le type est de la forme $(x_1 : A_1)...(x_n : A_n)s$ où $s$ est une sorte. Un objet de classe $C$ est un terme de type $(C\ t_1...t_n)$. Pour définir une coercion $f$ de $C$ vers $D$, la notation est `f :C >-> D`. Pour coercer un objet $t : (C\ t1..tn)$ de $C$ vers $D$, il suffit d'appliquer la coercion $f$ sur cet objet et l'on obtient alors le terme $(f\ t1..tn\ t)$ qui est bien un objet de $D$.

Le principe de coercion est souvent utilisé afin de simplifier la syntaxe. Pour cela, nous pouvons, par exemple, définir une fonction d'injection comme une coercion, ce qui évite de l'écrire lorsqu'elle est utilisée.

## D.5 Conversion

Le Calcul des Constructions Inductives comprend un mécanisme de conversion qui permet de décider si deux termes sont intentionnellement égaux (convertibles).

Il existe plusieurs règles de réduction dans `Coq` ($\beta$, $\delta$,$\iota$,$\zeta$,$\eta$) mais nous n'en présenterons que certaines.

### D.5.1 $\beta$-réduction

Soit un environnement $E$ et un contexte $\Gamma$. Cette règle permet d'identifier un objet $\alpha$ de type $T$ et une application $\lambda x : T.x\ \alpha$ :

$$E[\Gamma] \vdash (\lambda x : T.t\ u) \triangleright_\beta t\{x/u\}$$

### D.5.2 $\delta$-réduction

Soit un environnement $E$ et un contexte $\Gamma$. Cette réduction concerne les variables de contexte ou les constantes de l'environnement global. Il s'agit alors de les expanser :

$$E[\Gamma] \vdash x \triangleright_\delta t \text{ si } (x := t : T) \in \Gamma$$

[4] énoncé plus précisément dans le chapitre 1

$$E[\Gamma] \vdash c \rhd_{\delta} t \text{ si } (c := t : T) \in E$$

### D.5.3 $\iota$-réduction

Cette règle est spécifiquement associée aux objets inductifs. Un $\iota$-rédex est un terme de la forme : $< P > Cases\ (c_{pi}\ q_1...q_r\ a_1...a_m)\ of\ f_1...f_l\ end$ avec $c_{pi}$ le $i^{\text{ème}}$ constructeur du type inductif $I$ à $r$ paramètres. La $\iota$-réduction de ce terme est $(f_i\ a_1...a_m)$, ce qui donne la règle :

$$< P > Cases\ (c_{pi}\ q_1...q_r\ a_1...a_m)\ of\ f_1...f_n\ end\ \rhd_{\iota}\ (f_i\ a_1...a_m)$$

## D.6 Principe de sections

Ce principe est un principe de $\lambda$-abstraction. L'utilisateur peut ainsi factoriser des hypothèses ou des paramètres afin, par exemple, d'éviter les généraliser tous les théorèmes utilisant ces paramètres ou hypothèses. Un exemple d'utilisation typique est celui que nous faisons lors de notre développement sur les suites et les séries.

# Index

# Index Général

# Index des Symboles et Identificateurs

# Bibliographie


[1] The ADIC Home Page.
http ://www-fp.mcs.anl.gov/adic/.

[2] The Calculemus Project Home Page.
http ://http ://www.mathweb.org/calculemus/.

[3] The PADRE2 Home Page.
http ://warbler.ise.chuo-u.ac.jp/Padre2/.

[4] Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives.* Thèse de Doctorat, Université Paris 7, Novembre 1999.

[5] C. Bischof, A. Carle, P. Khademi, A. Mauer, and P. Hovland. ADIFOR2.0 User's Guide. Technical Report ANL/MCS-TM-192,CRPC-TR95516-S, Argonne National Laboratory Technical Memorandum, June 1998.

[6] E. Bishop. *Foundations of Constructive Analysis.* McGraw-Hill, 1967.

[7] H.J. Boehm. Constructive Real Interpretation of Numerical Programs. In *1987 ACM conference on interpreters and interpretives techniques*, 1987.

[8] Sylvain Boulmé. *Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de calcul formel.* Thèse de Doctorat, Université Paris 6, Décembre 2000.

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

[10] Samuel Boutin. *Réflexions sur les quotients.* Thèse de Doctorat, Université Paris 7, Avril 1997.

[11] G Cagnac, E Ramis, and J Commeau. *Traité de mathématique spéciales - Analyse 2.* Masson, 1967.

[12] Gustave Choquet. *Cours de Topologie.* Masson, 1992.

[13] François Clément. *Une formulation en temps de parcours par migration pour la détermination des vitesses de propagation acoustique à partir de données sismiques bidimensionnelles.* Thèse de Doctorat, Université Paris 9, 1994.

[14] G. E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In *Second GI Conference on Automata Theory and Formal Languages*, volume 33, pages 134–183. Springer-Verlag LNCS, 1976.

[15] Jean Combes. *Suites et séries.* Puf, 1982.

[16] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the Nuprl Proof development System.* Computing Science Department, Cornell University, 1986.

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

[18] Thierry Coquand. The Calculus of Constructions. In *Information and Computation*, volume 76, pages 95–120, 1988.

[19] Thierry Coquand and Christine Paulin-Mohring. Inductively Defined Types. In *Proceedings of Colog'88*, volume 417. Springer-Verlag LNCS, 1990.

[20] Cristina Cornes. *Conception d'un langage de haut niveau de représentation de preuves : récurrence par filtrage de motifs, unification en présence de types inductifs primitifs, syntèse de lemmes d'inversion.* Thèse de Doctorat, Université Paris 7, Novembre 1997.

[21] Y. Coscoy. A Natural Language Explanation for Formal Proofs. In C. Retoré, editor, *Proceedings of Int. Conf. on Logical Aspects of Computational Liguistics (LACL), Nancy*, volume 1328. Springer-Verlag LNCS/LNAI, September 1996.

[22] Marc Daumas, Laurence Rideau, and Laurent Théry. A Generic Library for Floating-Point Numbers and its Application to Exact Computing. In *Proc. TPHOL2001*, volume 2152. Springer-Verlag LNCS, 2001.

[23] David Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island*, volume 1955, pages 85–95. Springer-Verlag LNCS/LNAI, November 2000.

[24] David Delahaye. *Conception de langages pour décrire les preuves et les automatisations dans les outils d'aide à la preuve. Une étude dans le cadre du système Coq.* Thèse de Doctorat, Université Paris VI, Décembre 2001.

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

[26] Pietro Di Giannantonio and Alberto Ciaffaglione. A Co-Inductive Approach to Real Numbers. In *Proc. of TYPES 99*, volume 1956, pages 114–130. Springer-Verlag LNCS, 1999.

[27] J Dieudonné. *Eléments d'analyse.* Gauthier-Villard, 1968.

[28] Gilles Dowek. *La part du calcul.* Habilitation à diriger les recherches, Université Paris 7, 1999.

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

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

[31] Abbas Edalat and Peter John Potts. A New Representation for Exact Real Numbers. In *Theoretical Computer Science 6.* Elsevier Science B.V., 1997.

[32] Christèle Faure and Yves Papegay. Odyssée User's Guide Version 1.7. Technical Report 0224, INRIA Sophia-Antipolis, September 1998.

[33] Jean-Christophe Filliâtre. *Preuves de programmes impératifs en théories des types.* Thèse de Doctorat, Université Paris-Sud, Juillet 1999.

[34] J. D. Fleuriot and L. C. Paulson. Mechanizing Nonstandard Real Analysis. In *London Mathematical Society Journal of Computation and Mathematics 3*, 2000.

[35] Jean-Baptiste-Joseph Fourier. *Oeuvres de Fourier.* Numérisation BnF de l'édition de Paris, Guathier-Villars, 1890. Vomule 2,Mémoires publiés dans divers recueils / publ. par les soins de M. Gaston Darboux, pages 326-327, http ://gallica.bnf.fr.

[36] E. Fried and V.T. Sós. A Generalization for the Three-Distance Theorem for Groups. In *Algebra Universalis*, volume 29, pages 136–149, 1992.

[37] D. Gabbay. The Undecidability of Intuitionnistic Theories of Algebraically Closed Fields and Real Closed Fields. In *Journal of Symbolic Logic*, volume 38, pages 86–92, 1973.

[38] Ruben Gamboa. *Mechanically Verifying Real-Valued Algorithms in ACL2.* PhD thesis, University of Texas at Austin, May 1999.

[39] Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. Equational Reasoning via Partial Reflection. In *Proceedings of TPHOL.* Springer-Verlag, August 2000.

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

[41] Jean Yves Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur.* Doctorat d'État, Université Paris VII, Juin 1972.

[42] M.J.C. Gordon and T.F. Melham. *Introduction to HOL : A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

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

[44] Carl A. Gunter. *Semantics of Programming Languages : Structures and Techniques.* Foundations of Computing. MIT Press, 1992.

[45] J.H. Halton. The Distribution of the Sequence $\{\eta\xi\}$ (n=0,1,2,...). In *Cambridge Phil. Soc.*, volume 71, pages 665–670, 1965.

[46] John Harrison. Metatheory and Reflection in Theorem Proving : a Survey and Critique. Technical Report CRC-053, SRI Cambridge, UK, 1995.

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

[48] John Harrison. A Machine-Checked Theory of Floating Point Arithmetic. In *Proc. TPHOL99*, volume 1690, pages 113–130. Springer LNCS, 1999.

[49] Douglas Howe. Iporting Mathematics from HOL into Nuprl. In *Proc. TPHOL96*, volume 1125, pages 267–281. Springer LNCS, 1996.

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

[51] Claire Jones. Completing the Rationals ans Metric Spaces in LEGO. Cambridge University Press, 1993.

[52] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. Computer-Aided Reasoning : an Approach. Kluwer Academic, 2000.

[53] G. Kreisel and J.-L. Krivine. *Éléments de logique mathématique : théorie des modèles.* Dunod, 1964.

[54] Edmund Landau. *Foundations of Analysis. The Arithmetic of Whole, Rational, Irrational and Complex Numbers.* Chelsea Publishing Company, 1951.

[55] Vincent Lefèvre. An Algorithm that Computes a Lower Bound on the Distance between a Segment and $\mathbb{Z}^2$. Technical Report 97-18, LIP, Lyon, 1997.

[56] Vincent Lefèvre, Jean-Michel Muller, and Arnaud Tisserand. The Table Maker's Dilemma. In *IEEE Transactions on Computers*, volume 47, 1998.

[57] Jacqueline Lelong-Ferrand and Jean-Marie Arnaudiès. *Cours de mathématiques-Tome 2-Analyse.* Dunod Université, 1977.

[58] Xavier Leroy et al. *The Objective Caml System release 3.00.* INRIA-Rocquencourt, April 2000.
http ://caml.inria.fr/ocaml/htmlman/.

[59] P Loiseleur. Formalisation en Coq de la norme IEEE 754 sur l'arithmétique à virgule flottante. Rapport de stage, LIP, ENS Lyon, 1997.

[60] H. Lombardi and M.F. Roy. Constructive Elementary Theory of Ordered Field. In *Effective Methods in Algebraic Geometry*, volume 94, pages 249–262. Mora T., Traverso C., Birkhäuser Progress in Math., 1991.

[61] Micaela Mayero. The Three Gap Theorem (Steinhauss Conjecture). In *Proceedings of TYPES'99, Lökeberg*. Springer-Verlag LNCS.

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

[63] Alexandre Miquel. A Model for Impredicative Type Systems with Universes, Intersection Types and Subtyping. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science (LICS'00)*, 2000.

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

[65] C. Muñoz, R.W. Butler, V.A. Carreño, and G. Dowek. On the Formal Verification of Conflict Detection Algorithms. Technical Report TM-2001-210864, Nasa-Langley Research Center, May 2001.

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

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

[68] E. Nelson. Internal Set Theory. In *Bull. Amer. Math. Soc. 83*, pages 1165–1198, 1977.

[69] C. Paulin-Mohring. *Définitions inductives en théorie des types d'ordre supérieur.* Habilitation à diriger les recherches, Université Claude Bernard Lyon I, Décembre 1996.

[70] Chistine Paulin-Mohring. *Extraction de programmes dans le calcul des constructions.* Thèse de Doctorat, Université Paris VII, Janvier 1989.

[71] J.C. Reynolds. Polymorphism is not Set Theoretic. In Kahn, MacQueen, and Plotkin, editors, *Semantics of Data Types*, volume 173, pages 145–156. Springer-Verlag LNCS, 1984.

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

[73] Abraham Robinson. *Non-Standard Analysis.* Princeton University Press, 1996. Revised Edition.

[74] John M. Rushby, Natajaran Shankar, and Mandayam Srivas. PVS : Combining Specification, Proof Checking, and Model Checking. In *Proc. CAV'96*, volume 1102. Springer-Verlag LNCS, July 1996.

[75] David M. Russinoff. A Mechanically Checked Proof of IEEE Compliance of the AMD K5 Floating-Point Square Root Microcode. In *Proc. Formal Methods in System Desingn*, volume 14, pages 75–125, Junuary 1999.

[76] Amokrane Saïbi. *Outils génériques de modélisation et de démonstration pour la formalisation des mathématiques en théorie des types ; application à la théorie des catégories.* Thèse de Doctorat, Université Paris 6, 1999.

[77] N. Shankar, S. Owre, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Language Reference.* Computer Science Laboratory, SRI International, 1999.

[78] N. Shankar, S. Owre, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, 1999.

[79] N. Shankar, S. Owre, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, 1999.

[80] N.B. Slater. Gap and Step for the Sequence $\eta\theta$ *mod* 1. In *Cambridge Phil. Soc.*, volume 73, pages 1115–1122, 1967.

[81] V.T. Sós. On the Distribution *mod* 1 of the Sequence $\eta\alpha$. In *Ann. Univ. Sci. Budapest. Eötvös Sect. Math. 1*, pages 127–134, 1958.

[82] G. Stolzenberg. A New Courses in Analysis. 1972-1987.

[83] J. Surányi. Über der Anordnung der Vielfachen einer reelen Zahl mod 1. In *Ann. Univ. Sci. Budapest. Eötvös Sect. Math. 1*, pages 107–111, 1958.

[84] S. Sẃierckowski. On Successing Settings of an Arc on the Circumference of a Circle. In *Fund. Math.*, volume 48, pages 187–189, 1958.

[85] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951. Previous version published as a technical report by the RAND Corporation, 1948 ; prepared for publication by J. C. C. McKinsey.

[86] LogiCal Project The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 7.2*. INRIA-Rocquencourt, December 2001.
http ://coq.inria.fr/doc-eng.html.

[87] A.S. Troelstra and D. Van Dalen. *Constructivism in Mathematics - An Introduction - Volume I*. Studies in Logics and the Foundations of Mathematics. Elsevier science, 1988.

[88] Andrzej Trybulec. The Mizar-QC/6000 Logic Information Language. In *ALLC Bulletin (Association for Literary and Linguistic Computing)*, volume 6, pages 136–140, 1978.

[89] T. Van Ravenstein. The Three Gap Theorem (Steinhaus Conjecture). In *J. Austral. Math. Soc. (Serie A)*, volume 45, pages 360–370, 1988.

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

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

**Résumé.** Cette thèse concerne l'étude d'interactions entre les méthodes formelles et le domaine de l'analyse numérique. Beaucoup de programmes critiques sont, en effet, issus de cette science, mais les travaux traitant des applications des méthodes formelles aux programmes d'analyse numérique sont rares. La principale raison à cet état de fait est l'utilisation importante des nombres réels dans les programmes numériques, alors que les méthodes formelles manipulent plutôt des nombres entiers, ou plus généralement des structures discrètes. Néanmoins, nous défendons, le point de vue que les outils de méthodes formelles, en particulier les systèmes de preuves formelles tels que Coq, sont aujourd'hui de plus en plus adaptés pour que l'on s'intéresse aux nombres réels, ce qui ouvre une brèche pour l'application de ces systèmes aux programmes d'analyse numériques réels. Nous illustrons cette affirmation par l'étude de deux exemples issus, l'un de la géométrie réelle, et l'autre de l'analyse numérique. Le premier exemple est la preuve formelle du théorème des trois intervalles (conjecture de Steinhaus). Le second exemple concerne la preuve de corrections, en Coq, de l'algorithme de différentiation automatique de programmes FORTRAN utilisé dans le mode direct du système O∂yssée. Afin de pouvoir traiter ces deux exemples, nous développons, dans Coq et suite à une étude comparative des différentes méthodes, une bibliothèque des nombres réels, accompagnée de tactiques d'automatisation, qui fait maintenant partie de la distribution du système.

**Mots-clés:** Preuves formelles, Nombres réels, Analyse, Coq, Odyssée, Analyse numérique, Théorème des 3 intervalles, Automatisation

**Abstract.** This thesis studies the interactions between formal methods and the numerical analysis domain. Although, many critical programs come from this domain, work on applying formal methods to numerical analysis programs is quite rare. The main reason for this is the intensive use of real numbers in numerical programs, whereas formal methods handle integers, or more generally discrete structures. However, we believe that formal method tools, particulary formal proof systems such as Coq, are becomming increasingly appropriate and that it is therefore worth focusing on real numbers in order to make the application of these systems to real numerical analysis programs possible. We illustrate this with two examples from, respectively, real geometry and numerical analysis. The first example is the formal proof of the three gap theorem (Steinhaus conjecture). The second example is the correctness proof, in Coq, of the automatic differentiation algorithm of FORTRAN programs, which is used in the direct mode of the O∂yssée system. In order to deal with these two examples, after a comparative study of several methods, we develop, in Coq a library of real numbers, and some automation tactics, which have now been integrated in the system distribution.

**Keywords:** Formal Proofs, Real Numbers, Analysis, Coq, Odyssée, Numérical Analysis, 3 Gap Theorem, Automation