Chassez le naturel dans la formalisation des mathématiques

Yves Bertot[1] et Thomas Portet[1]

[1]Centre Inria de l'Université Côte d'Azur, France

Nous nous intéressons à l'utilisation des systèmes de preuves basés sur la théorie des types pour l'enseignement des mathématiques. Nous voulons remettre en question l'approche répandue qui consiste à introduire d'abord les nombres naturels, puis d'autres types de nombres.

Nous explorons une approche où le type des nombres réels est le seul type utilisé pour toutes les définitions concernant des nombres. Ceci nous oblige à reconsidérer les outils fournis pour définir des fonctions (en particulier la récursion), et pour calculer avec ces fonctions. L'une des caractéristiques de ce travail est de redonner de la place à la notion d'ensemble.

Nous illustrons cette approche avec quelques exercices mêlant suites récurrentes et nombres réels, où la facilité à manipuler ensemble des nombres habituellement séparés dans des types distincts permet des expérimentations enrichissantes pour les étudiants.

1 Introduction

Il y a maintenant de nombreux systèmes de preuves interactifs utilisables pour faire des preuves mathématiques. Même si ces systèmes de preuves ont été initialement conçus pour vérifier la correction d'algorithmes et de logiciels, il est souvent pratique d'inclure des capacités de raisonnement sur des objets mathématiques pour justifier la correction de logiciels. Par exemple, les primitives cryptographiques peuvent reposer sur des propriétés mathématiques d'objets remarquables comme des courbes elliptiques et la spécification même de la correction de tels algorithmes peut reposer sur le concept mathématique de probabilité.

Historiquement, pratiquement tous les systèmes de preuve basés sur la théorie des types ont commencé par décrire les entiers naturels, en spécifiant un type pour cette catégorie de nombres, puis d'autres types de nombres sont ajoutés au fur et à mesure que des concepts de plus en plus avancés sont fournis. Dans la présentation des données fournies aux utilisateurs finaux, ces nombres naturels apparaissent donc comme la première donnée disponible. C'est le cas dans le système Rocq [Tea24] que nous utilisons dans cette expérience.

Dans les systèmes basés sur la théorie des types avec des types inductifs, comme Agda, Rocq, ou Lean, la situation est renforcée par le fait que les entiers naturels se décrivent très bien comme un type inductif. Cette approche permet de disposer de capacités de calcul relativement efficaces.

Lorsque l'on fait des mathématiques plus avancées, on est amené à utiliser d'autres types de nombres, en particulier une grande partie des mathématiques étudiées à l'école et dans les premières années universitaires repose sur les nombres réels. Si l'on veut couvrir dans

la bibliothèque d'un système de preuve des connaissances correspondant à ce programme, on est naturellement amené à écrire des formules où nombres naturels et nombre réels se côtoient.

L'essor des systèmes de preuve interactifs est tel que l'on peut maintenant s'interroger sur leur apport dans l'enseignement des mathématiques. En particulier, l'étudiante en mathématique doit apprendre à raisonner. Pour raisonner, il faut savoir appliquer des syllogismes, faire la différence entre les hypothèses et la conclusion d'une phrase, comprendre des phrases énonçant l'existence d'un objet ou des phrases énonçant qu'une propriété est universellement satisfaite, et faire la différence entre ces deux types de phrases.

Pour de nombreux étudiants en mathématiques, le langage utilisé est une langue étrangère. Il faut pratiquer cette langue étrangère fréquemment pour progresser. En particulier, il faut apprendre à décrire des raisonnements en suivant correctement les implications fournies par les théorèmes fournis, et en faisant correctement les transformations de formules permises par les égalités prouvées. En fin de compte, l'apprentissage est réussi si l'étudiante est capable de faire un raisonnement qui sera considéré comme correct par une lectrice humaine. En d'autres termes, il s'agit d'apprendre le langage qui permet de s'inclure dans la communauté humaine des mathématiciens.

Un système de preuve interactif est justement conçu pour vérifier que toutes les étapes de raisonnement sont effectuées correctement. Il est donc naturel de vouloir explorer l'utilisation d'un tel système de preuve pour l'enseignement des mathématiques. En revanche, il faut éviter que ce nouvel outil impose une charge d'apprentissage supplémentaire, contre-productive si l'étudiant passe trop de temps à s'adapter au système de preuve mais n'a plus le temps d'apprendre les concepts mathématiques de son cursus. Il faut éviter que le langage mathématique imposé par le système de preuve soit trop éloigné du langage usuel pour le niveau d'apprentissage de l'étudiant.

Les nombres naturels, tels que manipulés dans les systèmes de preuve, sont fournis comme un type inductif. La théorie des types inductifs fournit la possibilité de définir des fonctions par cas et par récurrence de telle que sorte que les opérations de base peuvent être définies, comme l'addition et la multiplication et la soustraction. Mais pour la soustraction il apparaît une dissonance entre les mathématiques telle qu'elles apparaissent dans le système de preuve et dans la tradition humaine. Le système de preuve impose que la fonction soit totale, donc la conceptrice de la bibliothèque est obligée de donner un sens aux formules où le deuxième argument est supérieur au premier, comme $3 - 5$. Le choix fait dans tous les systèmes de preuve est que le résultat est le nombre 0, parce que la valeur réelle ou entière relative n'est pas un entier naturel.

La tradition des mathématiques et d'associer à chaque fonction son domaine de définition. Lorsque l'on écrit une formule composée, on doit expliciter le fait que toutes les fonctions sont bien utilisées dans leur domaine de définition. La tradition des mathématiques formalisées en théorie des types est que toute fonction doit être totale. Les fonctions sont donc généralement prolongées avec une valeur par défaut aux endroits où elles devraient être non définies. La modélisation reste fidèle dans la mesure où les théorèmes qui permettent de raisonner sur la fonction utilisent généralement l'appartenance des arguments au domaine de définition comme hypothèse. Ainsi, une mathématicienne utilisera naturellement le raisonnement :

$$(m - n) + n = m$$

en reposant sur le fait que la formule de gauche a été validée ($n \leq m$) plus tôt dans le discours. En revanche, l'outil de preuve fournit un théorème avec l'énoncé suivant :

```
forall m n, n <= m -> (m - n) + n = m
```

Ainsi, la condition de bonne formation de la formule n'est pas vérifiée a priori, mais uniquement au moment où l'on veut exploiter les propriétés de cette fonction de soustraction. Les bonnes vérifications sont faites, mais à un moment différent.

Il reste que l'utilisatrice finale doit mémoriser le fait que lorsqu'elle voit l'opération $m - n$ dans une formule mathématique, il ne s'agit pas de la soustraction usuelle, mais de cette soustraction si n est plus petit que m et 0 sinon.

Le coût pour l'utilisatrice finale n'est pas seulement un coût de mémorisation. Les outils de démonstration (appelés des tactiques) fournis dans le système de preuve sont également impactés par la présence de cette représentation irrégulière de la soustraction. Dans le système Rocq, il existe une tactique appelée `ring`, qui est capable de prouver des égalités entre formules polynomiales, en exploitant les propriétés des différents opérateurs. Cette tactique est très pratique et son comportement très facile à comprendre, sauf qu'elle est "bloquée" en présence de soustractions sur les entiers naturels. Par exemple, la formule suivante est une formule évidente à l'œil nu que cette tactique ne sait pas résoudre.

$$(n + m - (m + n)) * n = 0$$

Quand on avance dans les mathématiques, on est de toutes façons amené à étudier des ensembles de nombres plus riches que l'ensemble des entiers naturels. Par exemple, si l'on considère la suite de Fibonacci $\mathcal{F}$, définie par les équations suivantes :

$$\mathcal{F}(0) = 0 \qquad \mathcal{F}(1) = 1 \qquad \mathcal{F}(n+2) = \mathcal{F}(n) + \mathcal{F}(n+1),$$

un résultat mathématique connu est donné par la formule suivante :

$$\mathcal{F}(n) = \frac{\phi^n - (\frac{-1}{\phi})^n}{\sqrt{5}},$$

où ϕ est le nombre d'or.

$$\phi = \frac{\sqrt{5} + 1}{2}$$

Alors que le membre gauche de l'égalité fait intervenir une fonction définie seulement par récurrence sur les entiers naturels et en utilisant des additions de nombres naturels, la formule de droite fait intervenir des opérations qui ne sont pas stables pour les nombres naturels, les nombres entiers relatifs, ou même les nombres rationnels.

Nous proposons de développer une librairie d'initiation à la preuve mathématique dans laquelle on n'utilise qu'un seul type de nombres, pour que les étudiants puissent travailler confortablement avec les concepts mathématiques qu'ils doivent apprendre à maîtriser. Ce travail est complémentaire des efforts proposés par d'autres chercheurs pour fournir un environnement où les étudiants peuvent bénéficier d'une approche leur permettant de rédiger leurs preuves dans un style de "langue naturelle contrôlée" parce que ces efforts visent à améliorer le langage utilisé pour décrire les raisonnements logiques, sans modifier le cadre de travail ; les faits utilisés restent proches de la théorie des types traditionnelle et reposent sur plusieurs types de nombres.

Dans notre bibliothèque, les nombres naturels sont présents, mais présentés comme un sous-ensemble des nombres réels. De même nous fournissons un ensemble des nombres entiers. L'ensemble des nombres naturels est décrit par un prédicat inductif de type $\mathbb{R} \to \mathbb{P}rop$. Ceci nous permet de bénéficier directement des fonctionnalités existantes pour les preuves par récurrence. En revanche, cette approche ne nous permet pas de définir des fonctions récursives sur les entiers naturels. Nous fournissons une commande pour remplir cette lacune, et nous montrons sur quelques exemples que cette commande permet de retrouver la capacité de raisonner sur des suites récurrentes, un sujet qui est souvent abordé dans les premiers cours de mathématiques.

Nous décrirons les grandes lignes d'implémentation des différentes commandes que nous fournissons, nous donnons quelques exemples qui illustrent les gains que permettent cette approche, en termes de simplicité d'utilisation pour un étudiant découvrant à la fois les mathématiques et les systèmes de preuve interactifs.

Les expériences décrites dans cet article sont visibles sur le site suivant :
`https://github.com/ybertot/one_num_type/tree/jfla25`

2 Décrire des sous-ensembles des nombres réels

Dans le système de preuve, l'ensemble des nombres réels $\mathbb{R}$ est représenté par un type dénoté `R`. Les opérations usuelles sur les nombres sont représentées par des fonctions `+`, `-`, `*`, `/`. Les constantes entières de type `R` s'écrivent directement dans leur représentation décimale. De cette manière, une utilisatrice finale peut se contenter d'écrire la formule suivante `3 - 5 * 2`.

Nous reviendrons plus tard sur le dispositif qui permet cette utilisation de notations décimales, car il repose sur des nombres entiers dans un type distinct, d'une façon dont l'utilisatrice ne devrait pas avoir conscience.

L'approche usuelle pour définir un ensemble dans le système preuve Rocq est d'identifier l'ensemble avec la propriété caractéristique de cet ensemble. Cette propriété caractéristique est une fonction, de telle sorte que l'on écrit `Rnat n` pour exprimer que `n` est dans l'ensemble `Rnat`. Le système de preuve est basé sur la théorie des types, donc cette fonction doit avoir un type de départ (où elle prend ses arguments). Ici, nous considérons des sous-ensembles de $\mathbb{R}$ donc la fonction a le type `R -> Prop`.

Les nombres naturels peuvent être décrits comme le sous-ensemble minimal de $\mathbb{R}$ qui contient 0 et qui est stable par l'opération d'ajouter 1. Le fait que l'ensemble considéré soit minimal donne naturellement l'existence d'un principe de récurrence qui s'énonce de la façon suivante : *toute propriété qui est vraie pour* 0 *et qui est vraie pour tout nombre de la forme* $n + 1$ *si elle est déjà vraie pour* n *est vraie pour tous les entiers naturels.*

Dans les systèmes de preuve, une telle propriété définie par minimalité peut généralement être construire à l'aide d'une définition inductive. Pour les nombres naturels, la voici :

```
Inductive Rnat : R -> Prop :=
| Rnat0 : Rnat 0
| Rnat_suc : forall n, Rnat n -> Rnat (n + 1).
```

Grâce à cette définition, on prouve aisément que la somme de deux nombres naturels est un nombre naturel (par récurrence sur l'un des deux arguments), de même pour le produit. En revanche la soustraction de deux nombres naturels ne satisfait pas nécessairement cette propriété. La différence de deux nombres est un entier relatif, et cet entier relatif est un nombre naturel sous une condition d'ordre.

Ceci nous amène naturellement à considérer le sous-ensemble de `R` qui contient les entiers relatifs. Plusieurs façons de définir les entiers positifs sont disponibles, nous avons choisi la suivante :

```
Inductive Rint : R -> Prop :=
| Rint1 : Rint 1
| Rint_sub : forall x y, Rint x -> Rint y -> Rint (x - y).
```

Cette définition inductive permet de montrer que la somme de deux nombres entiers est un nombre entiers, de même pour le produit et la soustraction.

L'utilisatrice finale ne sera probablement pas amenée à utiliser le principe de récurrence associé à la définition inductive `Rint`. En revanche les preuves par récurrence usuelles d'un cours de mathématiques seront habituellement faisable directement avec le principe de récurrence associé avec `Rnat`.

Enfin, l'utilisatrice finale ne devra pas le voir, mais pour nos développements, nous aurons besoin d'une correspondance entre l'ensemble `Rnat` et les type inductifs `nat` et `Z`.

La bibliothèque standard des nombres réels de Rocq fournit des fonctions `INR` et `IZR` de type `nat -> R` et `Z -> R`. Ces fonctions sont définies récursivement, de telle sorte que l'on peut montrer facilement que l'image de `INR` coïncide avec l'ensemble `Rnat`. Les énoncés sont les suivants :

```
Rnat_INR n : Rnat (INR n).
```

```
Rnat_exists_nat x {xnat : Rnat x} :
  exists n, x = IZR (Z.of_nat n).

Rnat_cst x : Rnat (IZR (Z.pos x)).
```

Les différents théorèmes exprimant que l'ensemble des nombres naturels est stable pour les opérations `+`, `*`, `Rabs`, sont déclarés comme des instances de classes de types. Ainsi, lorsque l'utilisatrice finale a besoin de vérifier qu'un nombre réel satisfait la propriété d'être un nombre naturel, cette preuve peut souvent être effectuée automatiquement.

Un des éléments remarquables de ces théorèmes de stabilité est le théorème `Rnat_cst` qui exprime que toute constante numérique comme 1, 3, 42 appartient à l'ensemble `Rnat`. Ce théorème ne devrait pas être montré aux utilisateurs finaux, sous peine d'avoir à expliquer l'existence de plusieurs types de nombres, mais il doit être déclenché automatiquement chaque fois que l'ont veut vérifier qu'une constante entière (de type `R`) est bien dans l'ensemble `Rnat`. Ce théorème a pour énoncé :

```
Rnat_cst : forall p : positive, Rnat (IZR (Z.pos p))
```

Ainsi, le nombre `42` de type `R` est affiché `42`, mais est représenté en réalité par l'expression `IZR(Z.pos 42%positive)`. La sous expression `42%positive` représente le nombre `42` dans le type `positive`. La preuve que le nombre réel `42` est bien élément de `Rnat` est fournie par le théorème `IZR_cst 42%positive`.

La notation `42%positive` mérite une explication. L'analyse syntaxique effectuée dans le système de preuve est paramétrable pour indiquer que l'on veut lire données de différentes manières. Ici, le suffixe `%positive` permet d'indiquer que l'on veut mentionner un nombre de type `%positive`. Nous verrons plus tard que nous utiliserons également le suffixe `%nat` pour mentionner un nombre naturel de type `nat`. L'existence de ces notations obscurcit notre présentation, mais c'est justement notre objectif de faire disparaitre ce besoin de notations distinctes pour différentes représentations du même nombre mathématique.

En revanche, le théorème qui permet de prouver qu'une soustraction retourne un nombre naturel a la forme suivante :

```
Rnat_sub : forall n m, Rnat n -> Rnat m -> m <= n -> Rnat(n - m)
```

Ce théorème doit souvent être invoqué interactivement.

3 Définitions Récursives de Fonctions

L'une des caractéristiques essentielles des types inductifs dans les systèmes de preuve est qu'ils permettent de définir des fonctions récursives. En décidant de représenter tous les nombres utilisés dans le discours mathématique par des nombres réels, nous perdons cette caractéristique, parce que les nombres réels ne sont pas définis par un type inductif (et ils ne peuvent pas l'être).

Pour remédier à cela, nous fournissons une commande qui permet de définir des fonctions de `R` dans `R` de telle sorte que leur valeur est définie précisément pour tous les nombres naturels par un procédé récursif et qu'elles sont *indéfinies* ailleurs. Cet usage de valeurs indéfinies est très similaire à ce qui se produit déjà pour l'inverse, dont la valeur n'est pas définie en 0. L'inverse de 0 peut être manipulé comme un nombre réel, mais aucune propriété ne peut être obtenue par preuve pour ce nombre.

Donnons un exemple de définition obtenue par notre commande :

```
Recursive (def factorial such that factorial 0 = 1 /\
  forall n, Rnat (n - 1) -> factorial n = n * factorial (n - 1)).
```

À l'exécution de cette commande, deux nouveaux objets Rocq sont engendrés, le premier est une fonction nommée `factorial` de type `R -> R` et le deuxième est un théorème appelé `factorial_eqn` dont l'énoncé est exactement la conjonction décrite ci-dessus.

Il est évident que la spécification donnée ci-dessus ne permet pas de décrire la valeur de `factorial` pour l'entrée `0.5`. En revanche, elle permet bien d'obtenir le résultat que `factorial 3` est égal à `6`, en reposant sur plusieurs réécritures avec le deuxième membre de la conjonction, sur les preuves que `3 - 1`, `3 - 1 - 1`, `3 - 1 - 1 - 1` sont des nombres naturels, et sur la vérification que `3 * (3 - 1) * (3 - 1 - 1) = 6`. Cette égalité est prouvée en une seule étape par la tactique `ring`, parce que les nombres considérés sont des nombres réels et la tactique `ring` n'a pas de restriction vis-à-vis de la soustraction pour ce type de nombres. La tactique `ring` permet également de remplacer `3 - 1`, `3 - 1 - 1`, `3 - 1 - 1 - 1` par `2`, `1`, `0`, respectivement, et la preuve que ces nombres sont dans `Rnat` est automatique.

La commande permet également de définir des suites récurrentes d'ordre arbitrairement élevé (mais fini). Les suites récurrentes les plus fréquentes, où la valeur en $n+1$ est déterminée seulement en utilisant la valeur de n et la valeur de la suite en n, sont des suites d'ordre 1. Lorsque la valeur en $n+k$ est déterminée seulement en utilisant la valeur de n et les valeurs déjà connues en n, $n+1$, $\ldots n+(k-1)$, on dit que la suite est d'ordre k. Un exemple connu de suite d'ordre 2 est la suite de Fibonacci, où les valeurs en 0 et 1 sont 0 et 1 respectivement, et la valeur en $n+2$ est la somme des deux valeurs précédentes.

Avec notre commande, la suite de Fibonacci peut se définir de la façon suivante (nous nommons la fonction `fib` pour gagner de la place).

```
Recursive (def fib such that
    fib 0 = 0 /\
    fib 1 = 1 /\
    forall n : R, Rnat (n - 2) ->
    fib n = fib (n - 2) + fib (n - 1)).
```

3.1 Techniques d'implémentation

Ce que nous allons décrire dans cette section est la méthode que nous avons employée pour concevoir notre commande de définition récursive. L'utilisatrice finale n'a pas besoin de connaître ces détails : les seuls éléments mis à sa disposition sont la fonction définie avec son type, et le théorème qui collecte la spécification de comportement sous forme d'une conjonction d'égalités, dont une est quantifiée universellement sur un nombre dont un prédécesseur doit être un nombre naturel.

Puisque l'utilisatrice finale ne voit pas ce code, nous avons le droit d'utiliser les nombres naturels, et nous pouvons utiliser un outil de méta-programmation pour manipuler la spécification et construire une définition qui la satisfait.

3.1.1 Forme des spécifications récursives

Notre commande prend en argument un nom de fonction, disons `f`, et une proposition qui spécifie le comportement de cette fonction. Le nom de la variable est donc lié dans la spécification. Notre commande n'accepte que des spécifications qui respectent une forme syntaxique très précise : il doit s'agir d'une conjonction, dont les membres sont de deux formes différentes.

La première forme spécifie les valeurs pour les constantes 0, jusqu'à $k-1$. La forme syntaxique doit donc être `f i = V`. Dans la suite, nous appellerons ce nombre k *l'ordre de la récursion.*

La deuxième forme possible pour un membre est une quantification universelle sur une variable, disons `n`, le corps doit être une implication, dont la prémisse est `Rnat (n - k)`, la conclusion de cette implication doit être une égalité de la forme `f n = B`. La formule `B`

doit être une expression bien typée dans le contexte où `f` est une fonction de type `R -> R` et `n` est un nombre de type `R`. Nous appellerons cette formule `B` *le corps du pas récursif*. Ce corps peut contenir des appels récursifs de la forme `f (n - i)` où i est une constante entière comprise entre `1` et k. Il doit y avoir un seul membre de la conjonction avec cette deuxième forme.

3.1.2 Analyse des spécifications

Un premier parcours récursif de la spécification, paramétré par `f`, permet de retrouver l'ensemble des membres de la première forme et de construire une liste de paires (i, v_i) où les i sont des constantes entières supérieures ou égales à 0 et les v_i sont les termes représentant les valeurs spécifiées pour `f` i. Cette liste est triée, et on vérifie qu'il n'y a ni duplications ni manques. La longueur de cette liste nous donne la valeur de k. À partir de cette liste de paires triée, on construit une seconde liste qui ne contient que les valeurs v_i dans le même ordre. Nous appellerons cette liste *la liste des valeurs initiales*.

Un deuxième parcours de la spécification permet de trouver le seul membre qui est une quantification universelle sur une valeur `n`. Un parcours récursif du membre droit `B` de l'égalité en conclusion de ce membre droit, paramétré par `f` et `n` permet de retrouver l'ensemble des sous-termes de la forme `f (n - i)` et de construire la liste des valeurs i, que nous appellerons *la liste des décalages d'appels récursifs*. Pour cette liste, il n'est pas nécessaire de montrer l'absence de duplications, mais il est nécessaire de vérifier que le maximum est inférieur ou égal à k.

Si nous regardons la définition de `factorial` fournie dans la section précédente, il y a un seul membre de la première forme `factorial 0 = 1`, l'ordre de la récursion est 1, la liste des valeurs initiales est `[1]`, le corps du pas récursif est `n * factorial (n - 1)`, et la liste des décalages d'appels récursifs est `[1]`.

Si nous regardons la définition de `fib` fournie dans la section précédente, les membres de la première forme sont `fib 0 = 0` et `fib 1 = 1`, l'ordre de la récursion est 2, la liste des valeurs initiales est `[0; 1]`, le corps du pas récursif est `fib (n - 2) + fib (n - 1)`, et la liste des décalages d'appels récursifs est `[2,1]`.

3.1.3 Utilisation du type `nat` et son récurseur

L'implémentation repose sur le récurseur fourni pour le type `nat`. Nous allons définir une fonction de type `nat -> list R`, avec l'intention que la liste retournée pour l'entrée `n` contienne les valeurs `f n`, `f (n + 1)` ...`f (n +` $k-1$`)`, où k est l'ordre de la récursion.

Le récurseur pour le type `nat` est une fonction appelée `nat_rect` avec le type suivant :

```
nat_rect : forall (A : nat -> Type) (Init : A 0)
  (step : forall n, A n -> A (n + 1)) (n : nat), A n
```

Dans notre cas, nous utiliserons `nat_rect` de façon non dépendante, en instanciant `A` avec `fun _ => list R`, de telle sorte que la valeur `Init` devra être de type `list R` et `step` devra être de type `nat -> list R -> list R`. La fonction `step` a pour objectif d'expliquer comment construire la valeur retournée pour `n + 1`, lorsque l'on connaît la valeur de `n` (reçue en argument) et la valeur déjà calculée pour `n` (également reçue en argument). La fonction `step` commencera toujours par une abstraction de la forme `fun n (l : list R) => ...`.

Rappelons ici que la bibliothèque standard de Rocq contient une fonction `nth` qui permet de retourner le $n$ième élément d'une liste ou une valeur par défaut si la liste a moins de n éléments. Il y a aussi une fonction `INR` qui envoie tout nombre naturel vers le nombre `R` qui lui correspond.

Pour la fonction factorielle, on voit que la valeur retournée pour `n + 1` doit être la liste contenant un seul élément, le produit de `n + 1` avec la première valeur trouvée dans la liste en entrée, `nth 0%nat l 0`.

```
(INR n + 1) * nth 0%nat l 0
```

ceci doit être comparé avec le corps du pas récursif de la spécification, qui est :

```
n * factorial (n - 1)
```

Pour la suite de Fibonacci, , la liste retournée doit contenir `fib (n + 1)` et `fib (n + 2)`. La liste reçue en argument contient `fib n` et `fib (n + 1)`. Le premier élément de la liste retournée est tout simplement le deuxième élément de la liste reçue en argument, ce qui s'écrit `nth 1%nat l 0`. Le deuxième élément de la liste retournée n'est pas présent dans la liste reçue en argument, mais la spécification nous indique comment calculer cette valeur en utilisant les valeurs présentes dans `l`. La liste est donc de la forme suivante :

```
[nth 1%nat l 0; nth 0%nat l 0 + nth 1%nat l 0]
```

Maintenant que nous avons vu ce que nous voulons sur deux exemples, nous pouvons généraliser le traitement au cas d'une fonction récurrente d'ordre k arbitraire.

— La fonction `step` doit retourner une liste dont la longueur est l'ordre k.
— Les $k-1$ premiers éléments de la liste retournée doivent être `nth 1%nat l 0`, `nth 2%nat l 0`, et ainsi de suite jusqu'à $k-1$ (dans le cas de la fonction factorielle, il n'y a pas d'éléments de cette forme dans la liste).
— Le dernier élément de la liste doit être une copie du corps du pas récursif, modifiée des deux manières suivantes.
— Dans le corps du pas récursif, toute instance de `f (n -` i`)` doit être remplacée par `nth` $(k-i)$`%nat l 0`.
— Dans le corps du pas récursif, toute instance restante de `n` doit être remplacée par `INR n +` k.

Il faut noter qu'après les modifications au corps du pas de récursif, il ne doit plus rester d'instance de la fonction en cours de définition `f`. Par ailleurs, dans la spécification `n` a le type `R`, mais dans la fonction `step`, `n` a le type `nat`, c'est pourquoi la fonction `INR` doit être utilisée. Il est aussi remarquable qu'à l'endroit où une copie modifiée du corps du pas récursif est utilisé, on est en train de définir la valeur pour `f (n +` k`)`.

3.1.4 méta-programmation en `Elpi`

Pour faire les analyses et la génération de code, nous utilisons le langage de programmation `Elpi`, fourni comme une extension de Rocq. Ce langage est basé sur λ-prolog, étendu avec des techniques de résolution de contraintes. Un point qui rend ce langage particulièrement concis pour la conception de notre algorithme est que l'unité de base pour la programmation est la règle logique, au lieu de la fonction, comme on a l'habitude de le voir dans les langages de programmation fonctionnelle avec construction filtrage.

Ainsi, une opération de remplacement, comme celles que nous décrivons dans les quatrième et cinquième points de l'algorithme peut se décrire en disant que l'on prend une fonction existante de copie de termes, sauf que les instance du schéma à remplacer sont traitées par une règle prioritaire, qui fait le remplacement au lieu de faire une simple copie à l'identique.

Le texte suivant montre comment les remplacements sont effectués. Le principe mis en œuvre se résume de la façon suivante : faire un remplacement dans une expression, c'est produire une copie légèremet modifiée de cette expression, où le comportement du prédicat de copie est adapté pour reconnaitre les expressions que l'on veut remplacer et mettre à leur place les expressions cibles. L'infrastructure du langage `Elpi` pour Rocq fournit déjà le prédicat de copie (appelée `copy`) nous n'avons plus qu'à spécifier une modification de ce prédicat en ajoutant temporairement des règles de comportement pour ce prédicat, qui ont priorité sur les règles existantes. La construction `=>` permet d'ajouter des règles temporaires. La première règle temporaire est quantifiée universellement sur les variables `I`, `I'`, `In` et `In'` grâce à la construction `pi`, Elle exprime qu'au lieu de copier, on va remplacer les instances de termes où la fonction `F` est appliquée à `N` moins `I` par l'instance correspondante construite

avec `nth I'` et la liste de valeurs `L`. Le calcul de `I'` est exprimé par les lignes de rang 4 à 6 dans ce texte, et le point d'exclamation `!`, exprime que dès que le motif est reconnu, il ne doit pas y avoir de retour arrière dans l'exécution.

La ligne de rang 7 décrit comment les instances restantes de `N` seront remplacées par `N_plus_Order`, la valeur de `N_plus_Order` ayant été calculée une fois pour toutes auparavant.

```
(pi L \
  ((pi I I' In In' \ copy {{lp:F (lp:N - lp:I)}}
     {{nth lp:I' lp:L 0}} :-
     !,
      real_to_int I In,
      In' is Order - In,
      int_to_nat In' I'),
   copy N N_plus_Order) =>
copy RHS (RHS' L)),
```

Lorsque ce code est exécuté, une nouvelle constante `L` est créée pour représentée temporairement une variable liée dont dépendra le résultat. le terme dans la variable `RHS` est copié dans un nouveau terme. Alors que `RHS` ne contenait aucune occurrence de la constante `L`, le nouveau terme en contient, donc il existe une fonction `RHS'` tel que le terme obtenu soit égal à `(RHS' L)`. En conséquence, le terme `(RHS' L)` dépend de la constante temporaire `L` mais la fonction `RHS'` n'en dépend pas. Fournir cette fonction `RHS'` est une opération typique fournie par λ-prolog, et donc Elpi, et que l'on ne retrouve pas dans les autres langages de programmation. Ceci nous permet de gérer de façon concise la notion de variable liée.

Ce fragment permet également d'illustrer la fonctionnalité de *quotation* qui permet d'écrire directement des expressions dans la syntaxe usuelle de Rocq. Ici, le terme

```
{{lp:F (lp:N - lp:I)}}
```

permet de décrire la représentation en Elpi d'un terme de Rocq comportant l'application d'une fonction à une soustraction de nombres réels. Le préfixe `lp` dans `lp:N` (pour *lambda-prolog*) sert à indiquer que la variable λ-prolog `N` va être instanciée avec la valeur qui se trouve à cet endroit dans l'expression observée.

Lorsque la fonction basée sur `nat_rect` est finalement construite, on dispose d'une fonction de type `nat -> list R` et il reste deux opérations à effectuer : prendre le premier élément du résultat et fournir la bonne entrée. Dans l'environnement de travail pour l'utilisatrice finale, les seuls nombres sont de type `R`, donc nous avons besoin d'une fonction de type `R -> nat` qui associe chaque nombre réel qui est un nombre naturel au nombre correspondant de type `nat`. Nous définissons une telle fonction à l'aide de l'opérateur `epsilon` fourni par la bibliothèque de logique classique.

Plus précisément, nous définissons une fonction `IRZ` de type `R -> Z` qui associe tout nombre réel du sous-ensemble `Rint` au nombre entier de type `Z` correspondant, puis nous composons cette fonction avec une fonction de valeur absolue et de conversion vers `nat` pour obtenir une fonction `IRN` de type `R -> nat` et le théorème de correction suivant :

```
INR_IRN x {xnat : Rnat x} : INR (IRN x) = x.
```

Avec les contraintes que nous avons exprimées sur la spécification d'une fonction récursive, il est assez facile de prouver que les différents membres de la spécification sont satisfaits par la fonction obtenue. Nous avons écrit une tactique de quelques lignes pour cela et la preuve est sauvegardée dans un théorème dont le nom est obtenu en ajoutant le suffixe `_eqn` au nom de la fonction. Le code qui décrit toute cette commande de définition consiste en 340 lignes dans le langage Elpi (incluant des commentaires), plus la preuve de quelques théorèmes annexes, plus une tactique écrite en 16 lignes du langage `Ltac` pour prouver que la définition est bien satisfaite.

3.2 Effectuer des preuves avec les fonctions récursives

Pour effectuer des preuves sur les fonctions définies par notre commande, nous avons deux outils : le principe de récurrence associé au prédicat inductif `Rnat` et la spécification donnée par l'utilisateur. En effet notre commande fournit le théorème qui montre que cette fonction satisfait cette spécification.

Pour illustrer l'utilisation de ces outils nous avons pris la propriété déjà citée dans cet article, qui relie la suite de Fibonacci et le nombre d'or. Ce travail repose essentiellement sur les tactiques `field` et `lra`, avec une petite difficulté causée par le fait que `field` ne connaît pas la fonction de racine carrée. C'est donc l'utilisateur final qui doit trouver le moyen d'éliminer la présence de cette racine dans les formules.

Nous avons donc prouvé l'énoncé suivant (où `phi'` est l'opposé de l'inverse de `phi`).

```
Lemma Fibonacci_and_golden_ratio n:
    Rnat n -> fib n = (phi ^ n - phi' ^ n)/ sqrt 5.
```

La preuve fait moins de 200 lignes écrites dans un style accessible à des novices, avec des commentaires.

3.3 Ajouter des capacités de calcul

Pour les utilisateurs confirmés de Rocq, les nombres de types `nat` ou `Z` ont l'avantage notable de pouvoir être utilisés pour faire des calculs expérimentaux, surtout les nombres entiers, dont la structure de données exploite le codage binaire des nombres et permet donc la représentation et le calcul de très grand nombres. Cette capacité de calcul est perdue pour les nombres réels.

Par exemple, la spécification de la fonction `fib` est suffisamment expressive pour prouver la valeur de la fonction sur des entrées constantes, mais le procédé devient rapidement impraticable. Pour permettre à l'utilisatrice finale de tester la fonction définie sur des entrées assez grandes, nous avons fourni deux commandes.

La première de ces commandes s'appelle `R_compute` et elle permet de demander qu'un calcul soit effectué avec les opérations usuelles.

La seconde s'appelle `mirror_recursive_definition` et permet d'ajouter les fonctions récursives définies par notre première commande parmi les fonctions connues par `R_compute`.

Nous illustrons la première commande avec cet exemple.

```
R_compute (3 + (5 * 7)).
```

Le résultat de cette commande est, comme on peut s'y attendre, 38.

Il est possible de donner un second argument à notre commande qui doit être un nom. Si ce second argument est donné, un théorème est créé pour exprimer l'égalité entre l'expression donnée à calculer et le résultat obtenu. Ainsi, si l'on entre la commande.

```
R_compute (3 + (5 * 7)) name1.
```

Le théorème `name1` est créé par la commande, avec l'énoncé `3 + 5 * 7 = 38`.

Dans l'état courant du développement, la commande `R_compute` ne fait que des calculs en nombres entiers, pour les opérations d'addition, multiplication, soustraction, et opposé. Le principe est de faire les calculs dans `Z` au lieu de les faire dans `R`, en utilisant les correspondances prouvées déjà présentes dans la bibliothèque de Rocq. Ainsi la puissance de calcul des nombres inductifs est mise en œuvre, mais l'utilisatrice finale ne donne en entrée que des expressions réelles et ne reçoit en sortie que des nombres réels.

La deuxième commande s'appelle `mirror_recursive_definition`. L'objectif de cette commande est d'ajouter à la commande `R_compute` la possibilité de faire des calculs en utilisant les fonctions récursives définies grâce à la commande présentée dans la section précédente. Ainsi, lorsque les fonctions `factorial` et `fib` sont déjà définies, la séquence

de commande suivante se déroule avec succès, elle affiche un nombre très grand et ajoute un théorème dans l'environnement qui conserve une preuve du calcul. La réponse vient en moins d'une seconde (le calcul est instantané, mais la production du théorème prend un peu plus de temps).

```
mirror_recursive_definition fib.

mirror_recursive_definition factorial.

R_compute (factorial 10 + fib 1000) fact10_fib1000_eq.
```

En pratique, il y a quelques contraintes à respecter, parce que l'ensemble des calculs doit pouvoir être accompagné d'une preuve. La contrainte la plus forte est que les fonctions récursives ne doivent être appelées que sur des nombres naturels. L'utilisatrice ne sait pas nécessairement à priori si une expression est bien définie, mais la commande `R_compute` fournit un message d'erreur pour indiquer qu'une des fonctions considérées est appelée avec un argument négatif lorsque c'est le cas et aucun théorème n'est engendré.

3.3.1 Réaliser la capacité de calcul

La première chose qu'il est nécessaire de savoir, c'est que les nombres réels dans Rocq ont déjà une relation privilégiée avec les nombres entiers. Toute constante entière qui s'écrit sous forme décimale est déjà marquée comme l'application d'une fonction appelée `IZR` sur une constante de type `Z`. Cette fonction `IZR` est donc déjà présente en plusieurs endroits dans la formule, mais elle n'est jamais affichée (on parle de coercion implicite).

Pour effectuer le calcul, le principe est de transformer la formule à calculer v, qui est de type `R`, en une formule w de type `Z`, en garantissant que $v =$ `IZR` w. Ceci est fait de la façon suivante :

— Chaque instance du schéma `IZR` v est remplacée par v,
— Chaque instance d'une opération élémentaire sur les réels (comme `+`, `*`, `-`, `Rabs`) est remplacée par l'opération correspondante sur les entiers.

Une fois que la formule de type `Z` est obtenue, il suffit de demander son calcul par Rocq (on parle aussi de réduction en forme normale). Ce calcul retourne une constante entière z et le résultat affiché est le nombre réel `IZR` z, mais en accord avec le système de coercion implicite, le symbole `IZR` n'est pas affiché.

Pour justifier qu'une constante entière de la forma `IZR` v doit être traduite en v, il suffit d'utiliser la réflexivité de l'égalité, car dans ce cas la garantie à obtenir est simplement `IZR` $v =$ `IZR` v.

En pratique, le remplacement des fonctions élémentaires sur `R` par leur correspondant sur `Z` est fait en exploitant une table comportant des enregistrements à trois champs.

1. le premier champ contient une fonction f sur `R`
2. le deuxième champ contient une fonction f_z sur `Z`
3. le troisième champ contient un théorème de *morphisme*, exprimant que la fonction `IZR` est compatible avec les deux fonctions f et f_z :
   — si f est une fonction à un argument le théorème a l'énoncé

$$\forall x : \texttt{Z}, f\ (\texttt{IZR}\ x) = \texttt{IZR}(f_z\ x)$$

   — si f est une fonction à deux arguments le théorème a l'énoncé

$$\forall xy : \texttt{Z}, f\ (\texttt{IZR}\ x)\ (\texttt{IZR}\ y) = \texttt{IZR}(f_z\ x\ y)$$

Pour l'addition et la multiplication, les entrées dans la table ont la forme suivante :

```
Rplus Z.add add_compute
Rmult Z.mul mul_compute
```

Pour utiliser les théorèmes de morphisme, nous les encapsulons dans deux théorèmes qui facilitent l'usage récursif sans réécriture, un théorème pour les fonctions à un argument et un théorème pour les fonctions à deux arguments. Nous montrons ici le théorème pour les fonctions à deux arguments.

```
private.IZR_map2 :
forall (opr : R -> R -> R) (opz : Z -> Z -> Z),
  (forall a b : Z, opr (IZR a) (IZR b) = IZR (opz a b)) ->
  forall (a b : R) (c d : Z), a = IZR c -> b = IZR d ->
  opr a b = IZR (opz c d)
```

L'opération de traduction, que nous avons programmée à l'aide d'Elpi, prend en entrée une expression E de type `R` et retourne une expression E_z de type `Z` et un théorème dont l'énoncé est $E =$ `IZR` E_z. La règle Elpi qui traite les fonctions à deux arguments est la suivante :

```
translate_prf (app [F, A, B]) (app [F1, A1, B1])
  {{private.IZR_map2 lp:F lp:F1 lp:PFF1 lp:A lp:B lp:A1 lp:B1
           lp:PFRA lp:PFRB}}
  :-
  std.do! [
  thm_table F F1 PFF1,
  translate_prf A A1 PFRA,
  translate_prf B B1 PFRB
  ].
```

Cette règle exprime que l'expression où F est appliquée à A et B est traduite en l'expression où $F1$ est appliquée à $A1$ et $B1$, si F est en correspondance avec $F1$, et si A et B sont traduites respectivement en $A1$ et $B1$. La preuve que cette traduction est correcte est composée à l'aide de `private.IZR_map2`, du théorème de morphisme pour F et $F1$, ici appelé `PFF1`, et des preuves pour A et B.

Lorsque l'on affiche la preuve obtenue pour l'expression `3 + (5 * 7)`, on voit donc le texte suivant.

```
th38 =
  private.IZR_map2 Rplus Z.add add_compute 3 (5 * 7) 3 (5 * 7)
   (eq_refl : 3 = 3)
   (private.IZR_map2 Rmult Z.mul mul_compute 5 7 5 7
     (eq_refl : 5 = 5)
     (eq_refl : 7 = 7))
      : 3 + 5 * 7 = 38
```

les nombres qui apparaissent en 6e et 7e argument de `private.IZR_map2` sont de type `Z`, mais comme le système de preuve sait que des nombres entiers sont attendus à cet endroit, ils sont affichés comme les nombres qui précèdent, qui sont de type `R`.

3.3.2 Réaliser le calcul pour les fonctions récursives

La traduction des expressions peut également être effectuée sur le corps de la fonction `step` utilisée pour le récurseur `nat_rect`. Il y a quelques éléments supplémentaires utilisés dans le corps de cette fonction : la variable n de type `nat` et la variable l de type `list R`.

Nous supposons que la valeur n est seulement utilisée dans les appels récursifs et dans des appels à `INR` afin que cette valeur numérique soit utilisée dans opérations du type `R`. Ces usages de n doivent être remplacés par la valeur `Z.of_nat` n. Ces remplacements sont justifiés par un théorème fourni dans la bibliothèque nommé `INR_IZR_INZ`, avec l'énoncé suivant :

```
forall n : nat, INR n = IZR (Z.of_nat n)
```

Dans la version actuelle de notre code, nous supposons que la variable l est seulement utilisée dans des appels à la fonction `nth`. Pour ces appels, le premier argument qui est un entier naturel doit être laissé inchangé, la liste l également, mais le dernier argument, qui décrit la valeur par défaut qui serait utilisée si le premier argument est supérieur à la longueur de la liste, doit être remplacé par une valeur de type `Z`.

La correction du remplacement est justifiée par le théorème général suivant sur le récurseur, qui se prouve par récurrence sur l'argument entier naturel. L'invariant de la récurrence est que les listes de valeurs considérées à chaque étape sont les mêmes modulo application de la fonction `IZR` pour convertir les valeurs numérique à chaque rang dans la liste (exprimé par la fonction `map`).

```
private.nat_rect_list_IZR
     : forall (l0 : list Z) (l' : list R)
         (f : nat -> list Z -> list Z)
         (f' : nat -> list R -> list R) (n : nat),
       l' = List.map IZR l0 ->
       (forall (k : nat) (lR : list R) (lZ : list Z),
        lR = List.map IZR lZ -> f' k lR = List.map IZR (f k lZ)) ->
       nat_rect (fun _ : nat => list R) l' f' n =
       List.map IZR (nat_rect (fun _ : nat => list Z) l0 f n)
```

La cinquième ligne de cet énoncé exprime que l'une des conditions utilisée est que la liste initiale de valeur est la même (modulo application d'`IZR`). La sixième et la septième ligne expriment que les fonctions utilisées pour l'étape récursive transforment des listes égales en des listes égales (à nouveau modulo application d'`IZR`).

Une difficulté apparaît si la fonction récursive que l'on veut traiter fait appel à une autre fonction récursive avec un argument qui n'est pas de façon évidente un nombre naturel. Il est possible que l'utilisatrice qui définit cette fonction ait des connaissances précises sur l'algorithme et puisse en déduire que la valeur passée en argument soit toujours un nombre naturel. Notre traducteur impose que dans ce cas, un appel à la fonction `Rabs` soit inséré sur l'argument de l'appel à cette autre fonction récursive. Si la valeur donnée en argument était déjà un nombre naturel, cette fonction `Rabs` la laisse inchangée.

Les fonctions définies par `nat_rect` sont essentiellement conçues pour recevoir des arguments de type `nat`, mais les fonctions utilisées dans les formules calculées doivent prendre en entrée des nombres de `Z`. Pour concilier les deux, les fonctions que notre commande définit commencent par convertir le nombre entier en un nombre naturel en faisant une valeur absolue. Au final, la correspondance entre le calcul en nombre réel et le calcul en nombre entier n'est garantie que pour les entrées qui sont des nombres naturels. La commande `mirror_recursive_definition` créée une nouvelle fonction dont le nom est obtenu en accolant le suffixe `_Z_mirror` au nom de la fonction donnée en argument et un théorème de correction dont le nom est obtenu en accolant le suffixe `_Z_prf` avec l'énoncé de la forme suivante :

```
forall x y, x = IZR y -> f (Rabs x) = IZR (f_Z_mirror y)
```

Ce théorème n'a pas la même forme que les théorème de morphismes utilisés pour les opérations usuelles. Premièrement il prend directement la forme d'un théorème transformant une égalité sur les entrées en une égalité sur les sorties. Deuxièmement, il contient un appel à la valeur absolue sur l'argument de la valeur réelle. Puisque `x` est l'image de `y` par la fonction `IZR` nous savons que c'est un entier et que `(Rabs x)` est un nombre naturel.

Les fonction f, `f_Z_mirror`, et le théorème `f_Z_prf` sont ajoutés dans une autre table de correspondance.

Pour accepter quand même des formules dans lesquelles la fonction récursive n'est pas appelée sur le résultat d'une valeur absolue, nous utilisons le théorème de conversion suivant :

```
private.Rnat_Rabs : forall {f : R -> R} {fz : Z -> Z} :
   (forall (n : R) (z : Z), n = IZR z -> f (Rabs n) = IZR (fz z)) ->
   forall (n : R) (z : Z), Rnat n -> n = IZR z -> f n = IZR (fz z)
```

Ce théorème a une hypothèse supplémentaire, qui demande de vérifier que que l'argument de la fonction sur `R` est bien dans `Rnat`. Dans la commande `R_compute` cette vérification peut être faite à la volée en calculant simplement l'argument et en vérifiant qu'il est positif. Lorsque la preuve de correction est utilisée dans `mirror_recursive_definition`, seule le théorème *f*`_Z_prf` est utilisé.

Il y a également une tactique équivalente à `R_compute` qui fait le calcul et le remplacement d'une formule dans un but en cours de preuve, sans sauvegarder le théorème d'égalité dans un théorème, mais en incluant la preuve dans le terme en cours au moyen d'une réécriture.

4 Séquences d'entiers naturels et grandes opérations

Un idiome supplémentaire qui utilise des entiers naturels est celui des séquences finies d'objets. Par exemple, une famille linéaire de vecteurs ou un vecteur de `R ^ n`. Pour rendre ce type d'objets pratique à utiliser, nous proposons d'introduire une fonction avec une notation intuitive pour décrire une telle séquence, de fournir des notations spécifiques pour une collection d'opérations des listes vers les listes, en s'inspirant de la bibliothèque Mathematical Components [MT22] et de reposer sur des *grandes opérations* [BGBP08], sommes itérées ou produits itérés pour les opérations qui demanderaient normalement de la programmation récursive.

A un niveau intermédiaire pour l'audience finale, ou pour un public également formé à l'informatique, le cours pourrait également contenir une introduction à la programmation récursive, mais seulement sur les listes. Il faudrait alors étendre l'outil de calcul pour fournir une simulation calculatoire également pour les listes de nombres réels. Commencer par la programmation récursive sur les listes présente un avantage sur la programmation récursive sur les nombres naturels.

En effet, même pour des étudiants en informatique, nous avons constaté que les novices ont du mal à comprendre que l'on utilise le symbole `S` comme un schéma de structure (et non comme l'opération d'addition avec le nombre 1), et donc ils ont du mal à comprendre l'apparition de ce symbole dans le membre gauche d'une règle de filtrage en programmation récursive. Lorsque l'on introduit la construction de filtrage sur les listes, il est plus naturel de voir `(_ :: _)` comme un schéma de donnée, qui peut être utilisé à la fois pour faire du filtrage et pour construire une liste.

4.1 Séquences d'entiers naturels

Nous proposons de fournir la fonction `Rseq` à deux arguments n et m pour désigner la séquence de nombres réels de longueur m et contenant les nombres n, $n+1$, ... Comme pour les fonctions récursives, cette notation n'est bien définie que si m est un nombre naturel. Cette séquence peut bien sûr être définie à l'aide d'un schéma récursif, comme nous le faisons précédemment pour les suites récurrentes, mais il s'agit ici d'une fonction produisant des listes.

Il est important de fournir deux théorèmes exprimant le comportement calculatoire de cette fonctions récursive, plus des théorèmes de décomposition, exprimés essentiellement à l'aide des constructeurs du type des listes et la concaténation.

```
Rseq0 : forall n 0 : R, Rseq n 0 = nil

Rseq_suc : forall n m : R, Rnat m ->
  Rseq n (m + 1) = n :: Rseq (n + 1) m.
```

A l'usage, le théorème `Rseq_suc` n'est pas toujours pratique à utiliser, car il demande de préparer la formule sur laquelle on travaille pour faire apparaître l'expression de la forme m `+ 1` avant de permettre l'utilisation. La forme suivante est souvent plus facile à mettre en œuvre.

```
Rseq_suc' : forall n m : R, Rnat (m - 1) ->
  Rseq n m = n :: Rseq (n + 1) (m - 1).
```

La présentation donnée jusqu'ici n'impose pas que le premier argument de `Rseq` soit un entier naturel. Il y a donc un gain par rapport à l'approche reposant sur le type `nat`, puisque l'on peut utiliser la même fonction et le même jeu de notations pour des listes de nombres réels et pour des listes de nombres naturels. A ce stade préliminaire de nos expériences, ce gain est peut-être illusoire, parce qu'il est plus difficile d'exprimer que si le premier argument de `Rseq` est un entier naturel, alors tous les éléments de la séquence sont des entiers naturels (une propriété qui était donnée naturellement par le typage sur les entiers naturels).

4.2 Grands opérateurs

Les grands opérateurs apparaissent fréquemment dans les mathématiques sous deux formes. Soit il s'agit de présentations avec ellipses, où la séquence de plusieurs nombres est présentée, mais séparés par un opérateur binaire :

$$1 + \cdots + n \qquad 1^2 + \cdots + n^2$$

Soit on utilise une notation avec variable liée et un symbole conventionnel (parfois une version agrandie de l'opérateur binaire, parfois un symbole ad hoc connu de tous), la notation mathématique ressemble à ceci :

$$\sum_{i=0}^{n} f(i)$$

Dans Mathematical Components, la deuxième présentation est préférée, en utilisant des notations qui s'inspirent de ce que l'on doit écrire en LaTeX pour obtenir la présentation ci-dessus. Nous voulons nous inspirer de Mathematical Components, mais en réduisant le nombre de possibilités, pour réduire aussi les sources de confusion pour les novices.

Nous proposons de réduire l'usage des grands opérateurs aux deux opérateurs d'addition et multiplication (notation `\sum` ou `\prod`) et au seul cas où l'indice parcourt les entiers compris entre m et $n-1$, en utilisant la notation m `<=` i `<` n. Cette notation exprime que le domaine sur lequel s'applique l'opération itérée est vide si $n \leq m$. Le choix d'utiliser une comparaison stricte fait encore l'objet de réflexions. Il provient de la bibliothèque Mathematical Components, où il est motivé par des questions de régularité dans les calculs, mais les mathématiciens utilisent peut-être plus naturellement des comparaisons au sens large.

Pour les preuves, nous pouvons utiliser les théorèmes de base qui expriment le comportement récursif des opérations itérées en enlevant de la somme ou du produit itéré des valeurs en commençant par l'une ou l'autre des extrémités. Dans la bibliothèque Mathematical Components qui nous sert d'inspiration, ces théorèmes sont génériques et s'appliquent sur des types quelconques. Dans notre bibliothèque destinée à des novices, nous donnons l'implémentation pour les opérations usuelles. Ainsi, la somme d'une collection d'entiers s'écrit `\sum_(a <= i < b) ...` et le théorème suivant permet de montrer qu'une telle somme peut se décomposer en l'addition de la somme d'une collection plus petite et de la dernière valeur : on enlève la valeur prise à droite, d'où l'usage du suffixe `r` dans le nom du théorème. Nous avons également un théorème `sum0` qui exprime que la somme d'une collection vide d'entiers est nulle.

```
sum_recr (E : R -> R) (a b : R) :
```

```
  Rnat (b - a) -> a < b ->
  \sum_(a <= i < b) E i =
   (\sum_(a <= i < (b - 1)) E i) + E (b - 1).

sum0
     : forall (E : R -> R) (a : R), \sum_(a <= i < a) E i = 0
```

Nous avons testé ces définitions et théorèmes sur des preuves élémentaires.

$$\sum_{0\leq i<n} i = \frac{n(n-1)}{2} \qquad \sum_{0\leq i<n} i^2 = \frac{n(n-1)(2n-1)}{6} \qquad n! = \prod_{1\leq i<n+1} i$$

Dans les trois cas, l'avantage notable par rapport aux preuves utilisant le type des entiers naturels est que les preuves d'égalité sont généralement résolues par les tactiques `ring` ou `field`, cette dernière étant nécessaire lorsqu'une division apparaît. Le prix à payer est la nécessité de prouver des conditions annexes, sous forme d'appartenance à l'ensemble `Rnat` ou des comparaisons entre entiers. Pour résoudre ces conditions, il est important d'exploiter le fait que tout nombre naturel est positif ou nul et le fait qu'un nombre entier strictement inférieur à un nombre entier n est inférieur ou égal au prédécesseur de n.

La bibliothèque Mathematical Components est une source très riche de lemmes supplémentaires pour effectuer des raisonnements généraux sur les opérations itérées. En particulier, elle permet de faire des preuves où l'on exploite l'information que dans la somme itérée, l'index d'itération est compris entre les bornes de la somme. Dans nos expériences préliminaires, nous n'avons pas encore mis au point un théorème exprimant cette propriété.

5 La fonction binomiale

La fonction binomiale peut être présentée alternativement comme une fraction d'expressions obtenues avec des factorielles ou comme une fonction récursive sur les entiers naturels. La bibliothèque standard des réels utilise la définition suivante :

```
C =
fun n p : nat => INR (fact n) / (INR (fact p) * INR (fact (n - p)))
```

Cette définition repose sur la définition récursive dans le type `nat` de la fonction factorielle (ici appelée `fact`), puis une coercion dans les nombres réels, puis une division dans le type des nombres réels.

La bibliothèque Mathematical Components utilise plutôt la définition récursive suivante :

```
Fixpoint binomial n m :=
  match n, m with
  | S n', S m' => binomial n' m + binomial n' m'
  | _, 0 => 1
  | 0, S _ => 0
  end.
```

Le nombre `binomial n m` est donc une valeur entier naturel, susceptible d'être injectée dans n'importe quel anneau ou corps grâce à une fonction générique qui joue un rôle similaire à `INR`.

La motivation pour montrer ces deux définitions est de remarquer qu'elles ne coïncident pas lorsque le premier argument est strictement inférieur au second. Dans la définition de `C`, dans le cas où `n` est strictement inférieur à `p`, l'expression `n - p)` est égale à 0, la factorielle de la soustraction est égale à 1, et le résultat est un nombre rationnel strictement compris entre 0 et 1. Dans la definition de binomial, on peut démontrer par récurrence que

la valeur retournée est 0 (c'est le théorème `bin_small` dans la bibliothèque Mathematical Components).

S'il y a une différence, c'est que l'endroit où les deux définitions divergent n'est pas significatif.

La page Wikipedia en anglais (en octobre 2024) présente une troisième définition, plus efficace que les deux autres (mais pas encore optimale si k est proche de n).

$$\frac{n \times (n-1) \cdots \times (n-k+1)}{k \times (k-1) \cdots \times 1}$$

Nous pouvons définir cette fonction dans notre librairie de la façon proposée par cette page :

```
Definition choose (n k : R) :=
  (\prod_(m - k + 1 <= i < n + 1) i) / factorial k
```

Pour montrer que la valeur est bien entière, nous pouvons commencer par réconcilier la définition avec les deux autres, du moins pour les arguments qui le permettent.

```
choose_n_factorial : forall n m, Rnat n -> Rnat m, m <= n ->
  chose n k = factorial n / (factorial (n - k) * factorial k)

choose_n_0 : forall n, Rnat n -> choose n 0 = 1

chose_n_n : forall n, Rnat n -> chose n n = 1

choose_suc k n : Rnat k -> Rnat n -> k < n ->
   choose (k + 1) (n + 1) = choose k n + choose (k + 1) n
```

Le premier lemme met en relation notre définition avec la définition basée sur trois factorielles. Les trois autres lemmes sont très similaires aux règle de réduction donnée pour la définition de la fonction `binomial` par récursion sur le type inductif `nat` comme on le trouve dans la bibliothèque Mathematical Components.

La multiplicité des points de vue sur la fonction binomiale permet de faire de nombreux exercices sur cette fonction. En particulier, nous avons fait quelques essais sur des exercices suggérés par Pierre Rousselin et ses collègues [Rou24].

6 Travaux similaires et reliés

D'autres efforts existent pour rendre les systèmes de preuves basés sur la théorie des types pratiques pour l'enseignement. Waterproof [WAB+24] utilise Rocq pour former les étudiants à écrire des preuves qui sont acceptables pour la communication avec des humains, à commencer par leur professeurs de mathématiques. De façon similaire, Massot [Mas24] propose de modifier les tactiques de Lean pour que les utilisateurs emploient un langage proche du langage naturel pour intéragir avec le système de preuve.

Notre travail est complémentaire de ces travaux. En rendant l'intéraction avec le type de nombre plus automatique, il permet de faciliter leurs efforts pour produire un texte mathématique qui semble naturel à la lecture. En revanche, nous avons volontairement limité notre exploration du langage de tactique, justement parce que notre intention est de bénéficier de leurs approches.

Pierre Rousselin et ses collègues développent également des jeux d'exercices pour apprendre l'utilisation des systèmes formelles dans les premières années universitaire [Rou24]. Les exercices qu'ils ont produit nous ont servi d'inspiration à plusieurs reprises.

Les systèmes de preuve Mizar [BBG+15] et PVS [ORS92] fournissent aussi la possibilité d'utiliser un seul type de nombre, mais la question ne se pose pas avec la même acuité car les frontières entre les types ne sont pas aussi contraignantes que dans Rocq.

7 Conclusion

Chassez le naturel, il revient au galop. De nombreuses commandes et tactiques de Rocq sont conçues dans l'esprit que les nombres naturels sont toujours là à disposition des utilisateurs. Par exemple, la tactique `ring_simplify` crée spontanément des puissances où l'exposant est un nombre naturel.

Le travail que nous suggérons et commençons dans cet article est de faire l'inventaire de tous les endroits où le type des entiers naturels est utile, et d'enlever cet usage de la présentation des mathématiques. Cela ne veut pas dire que l'on veut faire disparaître le type des entiers naturels de la construction des objets sur lesquels nous reposons, mais que chaque concept présenté à l'utilisatrice finale doit utiliser les nombres réels et le prédicat `Rnat` comme interface.

Plus tard dans le déroulement des cours, il est probablement judicieux de réintroduire les nombres naturels et les autres types inductifs de nombres dans l'espace de travail, de façon à bénéficier directement des bénéfices que chacun peut apporter. Le point de vue de cet article est surtout d'éviter de surcharger l'utilisateur dans ses premiers pas d'apprentissage des mathématiques à l'aide des outils de preuve formelle.

Pour l'objectif d'enseignement, il est important d'ajouter plus d'outils pour faciliter les preuves d'appartenance aux sous-ensembles choisis. Il apparait également qu'un travail plus systématique devrait être fait pour associer à des fonctions un domaine de définition et systématiser les preuves pour vérifier qu'une fonction est bien appelée sur un argument dans son domaine de définition.

Références

[BBG+15] Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, Karol Pak et Josef Urban : Mizar : State-of-the-art and beyond. *In* Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe et Volker Sorge, éditeurs : *Intelligent Computer Mathematics*, pages 261–279, Cham, 2015. Springer International Publishing.

[BGBP08] Yves Bertot, Georges Gonthier, Sidi Ould Biha et Ioana Paşca : Canonical Big Operators. *In Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 de *Lecture Notes in Computer Science*, pages 12–16. Springer, août 2008.

[Mas24] Patrick Massot : Teaching Mathematics Using Lean and Controlled Natural Language. *In* Yves Bertot, Temur Kutsia et Michael Norrish, éditeurs : *15th International Conference on Interactive Theorem Proving (ITP 2024)*, volume 309 de *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27 :1–27 :19, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[MT22] Assia Mahboubi et Enrico Tassi : *Mathematical Components*. Zenodo, 2022.

[ORS92] S. Owre, J. M. Rushby et N. Shankar : Pvs : A prototype verification system. *In* Deepak Kapur, éditeur : *Automated Deduction—CADE-11*, pages 748–752, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[Rou24] Pierre Rousselin : Initiation aux preuves formelles, 2024. `https://www.math.univ-paris13.fr/~rousselin/ipf.html`.

[Tea24] The Coq Development Team : The coq proof assistant, juin 2024.

[WAB+24] Jelle Wemmenhove, Dick Arends, Thijs Beurskens, Maitreyee Bhaid, Sean McCarren, Jan Moraal, Diego Rivera Garrido, David Tuin, Malcolm Vassallo, Pieter Wils et Jim Portegies : Waterproof : Educational software

for learning how to write mathematical proofs. *Electronic Proceedings in Theoretical Computer Science*, 400:96–119, avril 2024.