Une mesure ordinale pour les preuves de terminaison en Coq

P. L. Bégay[1], P. Manoury[2] & I. Rakotonirina[1]

1: École Normale Supérieure de Cachan,
94230 Cachan, France
pierre-leo.begay@ens-cachan.fr, itsaka.rakotonirina@ens-cachan.fr
2: Laboratoire Preuves, Programmes et Système,
75013 Paris, France
pascal.manoury@pps.univ-paris-diderot.fr

Résumé

Nous abordons dans ce papier les preuves de terminaison de fonctions récursives par l'usage d'un ordre bien fondé. Nous proposons une utilisation de l'ordre sur les ordinaux : pour ce faire, nous donnons une représentation des ordinaux de ω^ω, posons une définition de leur relation d'ordre basée sur cette représentation et en montrons la bonne fondation. Nous illustrons le pouvoir d'expression de cette approche sur quelques exemples de définitions de fonctions au schéma de récursivité complexe. Leur terminaison est obtenue en définissant, pour chacune, un plongement des arguments dans un ordinal. L'avantage de cette approche est d'une part la facilité d'expression des fonctions définies par filtrage, et d'autre part son aspect systématique qui ouvre la perspective de l'usage d'une heuristique de décision pour la terminaison.

1. Introduction

La terminaison est une propriété centrale et indécidable pour la validation formelle de programmes. Les systèmes mécaniques d'aide à la preuve doivent affronter ce problème.

Cela s'observe tout particulièrement dans le cadre de l'assistant de preuve Coq [6] où il faut obtenir un certificat de terminaison pour toute fonction définie. La bonne fondation y est *a priori* obtenue par vérification de l'existence d'une décroissance structurelle des arguments lors des éventuels appels récursifs : avec cette approche, des manœuvres techniques sont presque toujours nécessaires pour définir des fonctions allant au delà du trivial schéma primitif récursif. Le mécanisme de définition en devient alors parfois assez peu naturel (voir par exemple la définition de `merge` du module `Coq.Sorting.Mergesort`).

Un procédé alternatif classique pour justifier qu'une fonction termine est de remplacer le contrôle syntaxique de décroissance structurelle par un argument logique reposant sur un ordre bien fondé. Coq propose d'ailleurs cette méthode de définition au travers des clauses `Recursive Definition` [1] et, plus récemment, `Program Fixpoint` [12] : l'utilisateur fournit alors l'ordre en question et justifie sa bonne fondation ainsi que la décroissance effective des arguments lors des appels récursifs. Cette démarche a l'avantage de séparer l'argument de terminaison de l'élaboration du code de la fonction, concentrant ainsi le problème sur la recherche d'ordres bien fondés. Il existe déjà un folklore relativement vaste de tels ordres parmi lesquels les ordres lexicographique, multi-ensemble, ou encore de Dershowitz [4] ou de Knuth-Bendix [7] ; chacun ayant ses instances académiques d'utilisation, cette bibliothèque met à notre disposition quelques armes sur le terrain. Toutefois, la recherche de tels

ordres est souvent manuelle voire ad hoc, et il faut chaque fois justifier de la bonne fondation de l'ordre proposé.

L'idéal serait donc d'obtenir une procédure de génération d'ordres certifiés bien fondés donnant la terminaison d'un large spectre de fonctions.

Une approche générique de la terminaison consiste à utiliser une *mesure* des arguments et un ordre bien fondé sur le domaine de cette mesure. Dans ce contexte, les ordinaux fournissent un domaine d'une grande expressivité. Cette méthode est d'ailleurs utilisée par le système d'aide à la preuve ACL2 (voir, par exemple, [8]). Sur cette idée et à l'aide du système ProPre de Manoury et Simonot [10], Monin et Simonot [11] donnent une procédure heuristique de génération de *mesure ordinale*, qu'ils appellent la *mesure ramifiée*, associant aux arguments des fonctions un ordinal dans ω^ω. Ils montrent comment l'ordre ainsi obtenu hérite des conditions suffisantes de terminaison définies dans le cadre de ProPre.

Nous proposons dans ce papier l'intégration d'éléments de cette heuristique à Coq. Dans une première étape, nous avons défini les ordinaux et leur relation d'ordre. Ceci posé, nous avons mené à bien la preuve de bonne fondation de cet ordre. Enfin, nous avons vérifié sur quelques exemples choisis de définitions récursives la bonne intégration de la méthode en Coq, en définissant «à la main» les mesures dont Monin et Simonot proposent une génération automatisée.

L'ensemble du développement Coq sur lequel repose ce papier est donné en [2].

La suite du papier est organisée comme suit : la section 2 fait office de préliminaires et présente les principes de construction de la mesure ordinale de Monin et Simonot ; la section 3 est dédiée à la définition et à la preuve de bonne fondation de l'ordre des ordinaux de ω^ω en Coq ; la section 4 illustre l'utilisation du procédé avec quelques exemples.

2. Génération d'un mesure ordinale

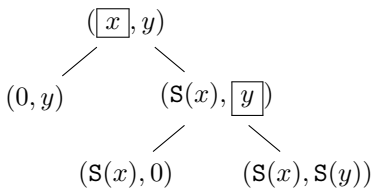
Dans leur article [11], Monin et Simonot exposent comment ils utilisent les structures arborescentes produites par l'heuristique de ProPre afin d'engendrer une mesure ordinale. Par soucis de concision et parce que les détails techniques en question ne sont pas centraux pour notre contribution, nous ne présenterons pas ici de définition formelle des concepts de [11] mais seulement quelques exemples suffisants, nous l'espérons, pour donner une bonne intuition du procédé.

2.1. Analyse de la structure des définitions

Avec le langage ML, le programmeur dispose d'un moyen compact de définir des fonctions par *filtrage* de leurs arguments. Cette possibilité est reconduite dans le système Coq pour des arguments appartenant à un type inductif. L'analyse par filtrage des arguments d'une fonction peut être projetée sur une structure arborescente appelée *arbre de distribution* dans [10]. Cette structure donne une manière de modéliser la façon dont les arguments d'une fonction vont influer sur son comportement afin de leur attribuer un poids adéquat dans une mesure ordinale. Considérons, pour prendre un premier exemple simple, une fonction $f : \mathbb{N}^2 \to F$ définie par le filtrage à la ML suivant :

$$\begin{cases} f(0, y) & = & t_1 \\ f(\mathtt{S}(x), 0) & = & t_2 \\ f(\mathtt{S}(x), \mathtt{S}(y)) & = & t_3 \end{cases} \qquad \text{pour certains termes } t_1,\ t_2,\ t_3$$

On peut représenter l'analyse par filtrage donnée par la définition de la fonction par l'arbre suivant :



On a encadré sur cette figure les positions des arguments analysés.

Cet arbre intègre bien l'intuition que l'on a de la structure des membres gauches de la définition équationnelle de f. Comme x est analysé avant y, x est distingué plus haut que y dans l'arbre : la mesure ordinale associée donnera donc à chaque argument un poids proportionnel à la hauteur [1] à laquelle il est analysé.

La mesure ordinale est alors donnée par une fonction $\Omega : \mathbb{N}^2 \to \omega^\omega$. Pour être générique :

— la définition de Ω suit le filtrage de f et traduit chaque feuille de l'arbre de distribution en une expression ordinale qui associe à chaque argument une puissance de ω d'autant plus élevée qu'il est analysé haut dans l'arbre. La fonction de génération dépend donc d'un arbre de distribution ;

— l'expression de Ω dépend également de fonctions de mesure m_i à valeur dans $\mathbb{N}$ qui donnent le facteur multiplicatif des ω. On considère une fonction m_i par argument.

Pour l'exemple ci-dessus, la fonction Ω est définie ainsi :

$$\begin{cases} \Omega(0, y) & = & \omega * m_1(0) \\ \Omega(\mathtt{S}(x), 0) & = & \omega * m_1(\mathtt{S}(x)) + m_2(0) \\ \Omega(\mathtt{S}(x), \mathtt{S}(y)) & = & \omega * m_1(\mathtt{S}(x)) + m_2(\mathtt{S}(y)) \end{cases}$$

Nontons que, dans la première équation de la définition de Ω, le second argument, qui n'est pas analysé, n'est pas pris en compte.

Dans l'heuristique héritée de ProPre, les fonctions m_i sont des fonctions de calcul de la taille des structures. Pour les entiers, c'est la fonction identité, pour les listes, la fonction de calcul de la longueur, etc.

Choix d'un arbre de distribution Dans l'exemple précédent, le fait que la forme de y ne soit pas précisée dans la première équation a naturellement suggéré de privilégier le premier argument par simple examen des membres gauches des équations. Considérons une définition de la forme suivante :

$$\begin{cases} f(0, 0) & = & t_1 \\ f(0, \mathtt{S}(y)) & = & t_2 \\ f(\mathtt{S}(x), 0) & = & t_3 \\ f(\mathtt{S}(x), \mathtt{S}(y)) & = & t_4 \end{cases} \quad \text{pour certains termes } t_1,\ t_2,\ t_3,\ t_4$$

L'examen des seuls membres gauches n'est pas suffisant ici pour décider de placer en tête la décomposition du premier ou du second argument. Dans l'esprit de mécaniser la génération des fonctions de mesure ordinale on peut envisager soit de recourir à une exploration exhaustive des arbres de distribution, soit de recourir à la stratégie d'analyse définie pour ProPre qui prend en compte à la fois les membres gauches des équations et les appels récursifs pouvant apparaître dans les membres droits. Nous ne développons pas ce point dans le travail présenté ici.

1. La hauteur d'un nœud est la différence entre la hauteur de l'arbre et la profondeur du nœud.

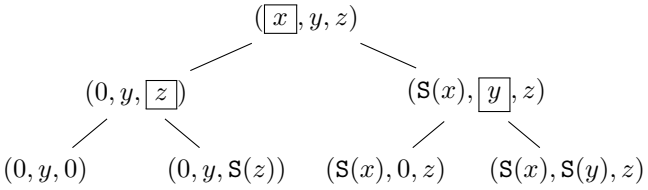
2.2. Autres schémas récursifs

Reprenons maintenant quelques exemples de [11] afin, d'une part, d'illustrer plus en détail la capacité de la procédure de Monin et Simonot à générer des ordres peu intuitifs et, d'autre part, d'aider à la compréhension de la construction de Ω.

Ordre lexicographique composé La comparaison lexicographique classique se rencontre souvent en pratique mais a parfois besoin d'être raffinée. Si on considère une fonction $f : \mathbb{N}^3 \to F$ définie par :

$$\begin{cases} f(0,y,0) & = & t_1 \\ f(0,y,\mathtt{S}(z)) & = & t_2 \\ f(\mathtt{S}(x),0,z) & = & t_3 \\ f(\mathtt{S}(x),\mathtt{S}(y),z) & = & t_4 \end{cases}$$

Son arbre de distribution serait :



Générant par suite la mesure ramifiée ci-dessous :

$$\begin{cases} \Omega(0,y,0) & = & \omega * m_1(0) + m_3(0) \\ \Omega(0,y,\mathtt{S}(z)) & = & \omega * m_1(0) + m_3(\mathtt{S}(z)) \\ \Omega(\mathtt{S}(x),0,z) & = & \omega * m_1(\mathtt{S}(x)) + m_2(0) \\ \Omega(\mathtt{S}(x),\mathtt{S}(y),z) & = & \omega * m_1(\mathtt{S}(x)) + m_2(\mathtt{S}(y)) \end{cases}$$

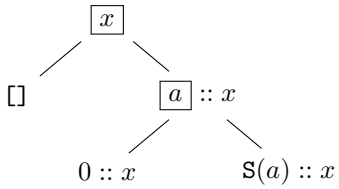
On obtient ainsi une famille (indexée par m_1, m_2, m_3) d'ordres lexicographiques mixtes, comparant (x, z) (deux premières équations) ou (x, y) (deux dernières) selon la situation. C'est ce que l'on peut appeler un ordre lexicographique *ramifié*.

Récursion sur des types paramétrés Au-delà des fonctions prenant en argument des entiers naturels, on peut aussi étudier dans quelle mesure la procédure de Monin et Simonot permet de traiter d'autres types de données. En particulier, on peut considérer le type des listes d'entiers, et des définitions de fonctions données par filtrage tant sur la structure de liste que sur la structure de ses éléments. Considérons par exemple une fonction calculant la somme des éléments d'une liste d'entiers, définie par la spécification suivante :

$$\begin{cases} f(\mathtt{[]}) & = & 0 \\ f(0 :: x) & = & f(x) \\ f(\mathtt{S}(a) :: x) & = & \mathtt{S}(f(a :: x)) \end{cases}$$

Ici, une simple récurrence sur les listes ne suffit pas pour obtenir la terminaison, car, dans la troisième équation, la structure de liste reste inchangée. Toutefois, le paramètre en tête de liste décroît. On peut obtenir ici un argument de terminaison en considérant l'ordre lexicographique sur les couples constitués de la longueur de la liste et de la valeur de son premier élément. Nous montrons ci-dessous comment la méthode de définition de mesures ordinales donne de quoi établir cet argument.

La fonction distingue donc en premier lieu la structure de la liste, puis celle de l'entier en tête. Ce qui donne l'arbre de distribution



La mesure ramifiée associée est alors :

$$\begin{cases} \Omega(\texttt{[]}) & = & \omega * m_1(\texttt{[]}) \\ \Omega(0 :: x) & = & \omega * m_1(0 :: x) + m_2(0) \\ \Omega(\mathtt{S}(a) :: x) & = & \omega * m_1(\mathtt{S}(a) :: x) + m_2(\mathtt{S}(a)) \end{cases}$$

C'est la présence de deux décompositions par filtrage qui donne la forme $\omega \cdot p + q$ à l'expression de Ω, p mesurant la liste et q l'entier par dominance de la liste dans ces décompositions. En choisissant la fonction longueur pour m_1 et l'identité pour m_2, on obtient ainsi l'ordre lexicographique visé, discriminant la taille de la liste puis la valeur de son premier élément.

3. Outils nécessaires à l'intégration à Coq

Nous présentons dans cette section notre codage des ordinaux de ω^ω en Coq, la relation d'ordre correspondante ainsi que la bonne fondation de cette dernière. Nous montrerons dans la section suivante (4) comment utiliser ce résultat pour obtenir la définition de fonctions récursives en conjonction avec la clause `Programm Fixpoint`.

La bonne fondation est formalisée en Coq par un *prédicat d'accessibilité* : `Acc`. Celui-ci est plus restrictif que la notion habituelle de bonne fondation (pas d'existence de suite infinie décroissante) [2]. Plus précisément, pour une relation `R` sur `E`, le *prédicat de bonne fondation* (`well_founded R`) est défini en Coq par :

$$\forall\ \mathtt{x} : \mathtt{E},\ \mathtt{Acc}_\mathtt{R}\ \mathtt{x}$$

où $\mathtt{Acc}_\mathtt{R}$ est le prédicat inductif défini par l'unique constructeur :

$$\mathtt{Acc\_intro}\ :\ \forall\ \mathtt{x} : \mathtt{E},\ (\forall\ \mathtt{y} : \mathtt{E},\ \mathtt{R}\ \mathtt{y}\ \mathtt{x}\ \rightarrow\ \mathtt{Acc}_\mathtt{R}\ \mathtt{y})\ \rightarrow\ \mathtt{Acc}_\mathtt{R}\ \mathtt{x}$$

Autrement dit, l'accessibilité d'un élément s'obtient en déléguant successivement la charge de la preuve aux strates inférieures jusqu'à aboutir à des éléments n'ayant plus de minorant. La bonne fondation s'exprime alors comme l'accessibilité de tout élément.

2. On peut certes prouver en Coq que `well_founded R` implique qu'il n'existe aucune suite infinie décroissante au sens de `R`, mais la réciproque nécessite typiquement l'axiome du choix dépendant.

3.1. Formalisation des ordinaux et de leur ordre

Un ordinal de ω^ω est un polynôme $P(\omega)$ à coefficients dans $\mathbb{N}$, autrement dit une somme de puissances de ω. Dans notre cadre où les polynômes sont souvent pleins, il est naturel de les représenter par la liste des coefficients de P, la puissance de poids fort en tête.

Cette représentation a quelques inconvénients (voir paragraphe ci-dessous : définition de l'ordre) et il en existe d'autres qui ne les possèdent pas. Toutefois, la traduction naïve des polynômes en listes de coefficients nous a semblé plus pertinente dans la mesure où sa simplicité va en effet dans le sens de l'usage que nous comptons faire des ordinaux. À savoir : fournir au programmeur un moyen aussi direct que possible pour exprimer un argument de terminaison en terme de mesure ordinale (voir les exemples section 4).

C'est pourquoi nous avons privilégié une représentation qui, si elle complique un peu la définition de l'ordre et sa preuve de bonne fondation, offrira en revanche à l'utilisateur une «interface» plus immédiate avec les ordinaux. En effet, dans chaque cas de la définition de la fonction de mesure ordinale, il lui suffira de donner la liste des valeurs à mesurer placées dans l'ordre selon lequel elles seront lexicographiquement comparées.

Définition de l'ordre Le parti de la simplicité de l'encodage a un prix : définir un ordre sur les listes d'entiers qui soit fidèle à l'ordre sur ω^ω. Deux points doivent être considérés lors de sa conception :

1. La représentation d'un ordinal n'y est pas unique. En effet, d'une part, une infinité de listes représentent 0 (`[]`, `[0]`, `[0,0]`...) et, plus généralement, `0::w` représente le même ordinal que `w`, pour tout `w:list nat`.
2. Le degré des polynômes représentés est implicite. En effet, le degré des polynômes représentés dépend de la longueur de la liste et de la position de son premier élément non nul (le degré de `[0;1;0;0]` est 2). Il est égal à la longueur de la liste diminuée de 1 lorsque celle-ci commence par un entier non nul.

On note $|\mathtt{w}|$ la longueur de la liste `w`. L'ordre $\prec$ (`wlt` dans [2]), sur les listes d'entiers, est défini inductivement par les clauses suivantes :

a. Ignorer les zéros de tête :
$\mathtt{w}_1 \prec \mathtt{w}_2 \quad \rightarrow \quad \mathtt{0} :: \mathtt{w}_1 \prec \mathtt{w}_2$
$\mathtt{w}_1 \prec \mathtt{w}_2 \quad \rightarrow \quad \mathtt{w}_1 \prec \mathtt{0} :: \mathtt{w}_2$

b. Pour des listes sans zéro de tête :

b.1. La liste vide est minimale :
$[] \prec \mathtt{S(a)} :: \mathtt{w}$

b.2. Sinon, comparer en priorité les degrés :
$|\mathtt{w}_1| <_{\mathbb{N}} |\mathtt{w}_2| \ \rightarrow \ \mathtt{S(a}_1\mathtt{)} :: \mathtt{w}_1 \prec \mathtt{S(a}_2\mathtt{)} :: \mathtt{w}_2$

b.3. Sinon, comparer lexicographiquement les coefficients :
$|\mathtt{w}_1| = |\mathtt{w}_2| \quad \rightarrow \quad \mathtt{a}_1 <_{\mathbb{N}} \mathtt{a}_2 \quad \rightarrow \quad \mathtt{S(a}_1\mathtt{)} :: \mathtt{w}_1 \prec \mathtt{S(a}_2\mathtt{)} :: \mathtt{w}_2$
$|\mathtt{w}_1| = |\mathtt{w}_2| \quad \rightarrow \quad \mathtt{w}_1 \prec \mathtt{w}_2 \quad \rightarrow \quad \mathtt{S(a)} :: \mathtt{w}_1 \prec \mathtt{S(a)} :: \mathtt{w}_2$

Si elle n'est pas « optimale » au sens de la remarque un peu plus haut, cette définition a le mérite de distinguer explicitement les cas de figure à prendre en considération pour la comparaison des listes. Ce caractère verbeux a été un guide pour la preuve de bonne fondation.

La preuve que la relation $\prec$ est bien fondée, transcrit en Coq par l'énoncé $\forall \mathtt{w} : \mathtt{list\ nat},\ \mathtt{Acc}_{\prec}(\mathtt{w})$, présente quelques difficultés liées à l'impossibilité d'utiliser une induction structurelle sur les listes (`a::w` peut encoder le même ordinal que `w`) ou sur leur longueur (même problème). Ces problèmes étant des conséquences de la potentielle présence de zéros inutiles en tête des listes, on pourrait définir une forme normale obtenue en tronquant lesdits zéros. Malheureusement, l'ensemble des formes normales n'est pas stable par extraction de queue (`[1;0;0]` est normale mais pas `[0;0]`) ce qui rend difficile leur utilisation au sein d'une induction.

Pour mener à bien la preuve de bonne fondation de $\prec$, nous sommes passés par la définition de sa restriction (notée $\prec_{|\cdot|}$ ici et `wlt_pad` dans [2]) à des listes de même longueur dont seule la plus petite (au sens de $\prec$) peut commencer par des zéros. Dans ce cas, on obtient plus facilement la bonne fondation par combinaison lexicographique d'ordre bien fondés.

Nous définissons inductivement $\prec_{|\cdot|}$ par les trois clauses suivantes :

a. La comparaison des degrés est une comparaison de taille ignorant le *padding* :
$|\mathtt{w}_1| \leqslant_{\mathtt{nat}} |\mathtt{w}_2| \quad \rightarrow \quad \mathtt{0}^{\mathtt{k}}\mathtt{::w}_1 \prec_{|\cdot|} \mathtt{Sa::w}_2 \qquad$ avec $\mathtt{k} = |\mathtt{w}_2| - |\mathtt{w}_1| + 1$

b. À degré égal, on conserve la comparaison lexicographique :
$|\mathtt{w}_1| = |\mathtt{w}_2| \quad \rightarrow \quad \mathtt{a}_1 <_{\mathtt{nat}} \mathtt{a}_2 \quad \rightarrow \quad \mathtt{S(a}_1\mathtt{)::w}_1 \prec_{|\cdot|} \mathtt{S(a}_2\mathtt{)::w}_2$
$|\mathtt{w}_1| = |\mathtt{w}_2| \quad \rightarrow \quad \mathtt{w}_1 \prec_{|\cdot|} \mathtt{w}_2 \quad \rightarrow \quad \mathtt{S(a)::w}_1 \prec_{|\cdot|} \mathtt{S(a)::w}_2$

Ici encore, si notre définition peut sembler inutilement redondante (*a.* se retrouve dans la première clause de *b.*), c'est qu'il s'est plus agi de donner un intermédiaire *technique* entre la définition de l'ordre général sur les listes ($\prec$) et sa bonne fondation que de caractériser finement une relation sur les listes.

Le schéma de preuve La preuve de bonne fondation de $\prec$ aura donc le squelette suivant :

(*i*) $\forall\ \mathtt{w} : \mathtt{list\ nat},\ \mathtt{Acc}_{\prec_{|\cdot|}}(\mathtt{w}) \rightarrow \mathtt{Acc}_{\prec}(\mathtt{w})$ (lemme `Acc_wlt_pad_Acc_wlt` de [2])
(*ii*) `well_founded` $\prec_{|\cdot|}$ (lemme `Acc_wlt_pad` de [2])
(*iii*) D'où on déduit `well_founded` $\prec$ (lemme `Acc_wlt` de [2])

3.2. Preuves de (*i*), (*ii*) et (*iii*)

Les démonstrations qui suivent sont formalisées dans [2]. Nous nous contentons donc ici de les décrire en français, à un niveau de détail néanmoins suffisant pour que la transcription en Coq apparaisse clairement à la lecture.

(*i*). Étant donné un ordinal `w`, on veut montrer que l'accessibilité restreinte $\mathtt{Acc}_{\prec_{|\cdot|}}(\mathtt{w})$ implique l'accessibilité générale $\mathtt{Acc}_{\prec}(\mathtt{w})$. Par induction sur l'hypothèse $\mathtt{Acc}_{\prec_{|\cdot|}}(\mathtt{w})$, on se donne deux ordinaux arbitraires $\mathtt{w}_1$ et $\mathtt{w}_2$ tels que $\mathtt{w}_1 \prec \mathtt{w}_2$ et, sous l'hypothèse d'induction $\forall \mathtt{w}', \mathtt{w}' \prec_{|\cdot|} \mathtt{w}_2 \rightarrow \mathtt{Acc}_{\prec}(\mathtt{w}')$, on veut démontrer que $\mathtt{Acc}_{\prec}(\mathtt{w}_1)$. Il suffit donc de justifier que $\mathtt{w}_1 \prec_{|\cdot|} \mathtt{w}_2$, le résultat découlant alors de l'hypothèse d'induction pour $\mathtt{w}' := \mathtt{w}_1$. Cela s'obtient aisément en comparant les longueurs des listes :

- Si $|\mathtt{w}_1| = |\mathtt{w}_2|$, alors l'hypothèse $\mathtt{w}_1 \prec \mathtt{w}_2$ se reformule en $\mathtt{w}_1 \prec_{|\cdot|} \mathtt{w}_2$ par analyse des définitions de $\prec$ et $\prec_{|\cdot|}$ (lemmes d'inversion).
- Supposons sinon que $|\mathtt{w}_1| \neq |\mathtt{w}_2|$. Au vu des règles *a.* de la définition de $\prec$, la comparaison par $\prec$ et par suite, l'accessibilité $\mathtt{Acc}_{\prec}$ sont préservées par ajout de zéros en tête des listes considérées. Quitte à compléter de cette manière la plus courte des deux listes, on peut supposer sans perte de généralité que $|\mathtt{w}_1| = |\mathtt{w}_2|$ et on est ramené au premier cas. □

(*ii*). On veut à présent démontrer la bonne fondation de l'ordre restreint, autrement dit, qu'on a $\mathtt{Acc}_{\prec_{|\cdot|}}(\mathtt{w})$ pour tout ordinal `w`. Il s'agit du résultat central de cette section et notre démonstration

repose sur une induction mixte :

— une induction sur la longueur des listes (puisque $\prec_{|\cdot|}$ compare des listes de même taille) ;
— une élimination que nous empruntons à P. Castéran, pour le cas inductif de l'induction sur la longueur.

Le principe de Castéran est le suivant :

Lemme (Castéran [3]). *Soit A et B deux ensembles respectivement munis d'une relation $<_A$ et $<_B$, ainsi qu'une proposition à deux paramètres $P : A \to B \to \texttt{Prop}$. On suppose que pour tout $a \in A$ et $b \in B$, $P(a, b)$ est vraie quand les deux conditions suivantes sont vérifiées :*

1. *$P(a', b')$ est vraie pour tout $a' <_A a$ et $b' <_B$-accessible.*
2. *$P(a, b')$ est vraie pour tout $b' <_B b$.*

Alors $P(a, b)$ est vraie pour tout $a <_A$-accessible et $b <_B$-accessible.

Formellement :

$$\begin{aligned}&(\forall\ a \in A,\ \forall\ b \in B,\\ &\quad (\forall\ a' <_A a,\ \forall\ b' \in B,\ \texttt{Acc}_{<_B}(b') \to P(a', b'))\\ &\quad \to (\forall\ b' <_B b,\ P(a, b'))\\ &\quad \to P(a, b))\\ &\to \forall\ a \in A,\ \forall\ b \in B, (\ \texttt{Acc}_{<_A}(a) \to \texttt{Acc}_{<_B}(b) \to P(a, b))\end{aligned}$$

Dans le cas inductif de l'induction sur la longueur de la liste, ce principe nous permettra de combiner l'accessibilité sur les entiers et la $\prec_{|\cdot|}$-accessibilité, cette dernière étant donnée par l'hypothèse d'induction jusqu'à une certaine longueur n. Formellement :

· On a bien $\texttt{Acc}_{\prec_{|\cdot|}}(\texttt{w})$ si $|\texttt{w}| = \texttt{0}$, autrement dit si `w=[]`, par minimalité de `[]`.

· Supposons à présent pour un certain $\texttt{n} \in \mathbb{N}$ la propriété établie pour toute liste de taille au plus n, autrement dit que :

$$(\texttt{HR}) : \quad \forall\ \texttt{w} : \texttt{list nat},\ |\texttt{w}| \leqslant \texttt{n}\ \to\ \texttt{Acc}_{\prec_{|\cdot|}}(\texttt{w})$$

Considérons alors `w` de taille `S(n)` et montrons que $\texttt{Acc}_{\prec_{|\cdot|}}(\texttt{w})$. De manière équivalente, on veut montrer que pour `w'` ordinal et `a` naturel :

$$|\texttt{a} :: \texttt{w}'| = \texttt{S(n)}\ \to\ \texttt{Acc}_{\prec_{|\cdot|}}(\texttt{a} :: \texttt{w}')$$

Puisque $\texttt{Acc}_{<_{\mathbb{N}}}(\texttt{a})$ (résultat classique) et $\texttt{Acc}_{\prec_{|\cdot|}}(\texttt{w}')$ (par `HR`), il nous suffit d'établir l'hypothèse de l'élimination de Castéran avec P la proposition ci-dessus afin de conclure. Autrement dit, sous les hypothèses suivantes :

$$(\texttt{H1}) \quad : \quad \forall\ \texttt{a}' <_{\mathbb{N}} \texttt{a},\ \forall\ \texttt{w}'',\ \texttt{Acc}_{\prec_{|\cdot|}}(\texttt{w}'')\ \to\ |\texttt{a}' :: \texttt{w}''| = \texttt{S(n)}\ \to\ \texttt{Acc}_{\prec_{|\cdot|}}(\texttt{a}' :: \texttt{w}'')$$

$$(\texttt{H2}) \quad : \quad \forall\ \texttt{w}'' \prec_{|\cdot|} \texttt{w}',\ |\texttt{a} :: \texttt{w}'| = \texttt{S(n)}\ \to\ \texttt{Acc}_{\prec_{|\cdot|}}(\texttt{a} :: \texttt{w}'')$$

on doit montrer que $|\texttt{a} :: \texttt{w}'| = \texttt{S(n)}\ \to\ \texttt{Acc}_{\prec_{|\cdot|}}(\texttt{a} :: \texttt{w}')$. Modulo une introduction et la définition de $\texttt{Acc}_{\prec_{|\cdot|}}$ (à savoir `Acc_intro`), on se donne un ordinal $\texttt{w}''$ ainsi que les deux hypothèses supplémentaires :

$$(\texttt{H3}) \quad : \quad |\texttt{a} :: \texttt{w}'| = \texttt{S(n)}$$

$$(\texttt{H4}) \quad : \quad \texttt{w}'' \prec_{|\cdot|} \texttt{w}'$$

et on est ramené à démontrer que $\texttt{Acc}_{\prec_{|\cdot|}}(\texttt{w}'')$. On conclut par une disjonction de cas sur la définition de $\prec_{|\cdot|}$ dans `H4` :

· Si $\mathtt{w''} = \mathtt{0^k :: w'''}$ avec $\mathtt{k} > \mathtt{0}$ et $\mathtt{a} = \mathtt{S(b)}$: par `H1`, `HR` et `H3`.

· Si $\mathtt{w''} = \mathtt{S(b) :: w'''}$ et $\mathtt{w'} = \mathtt{S(c) :: w''''}$ avec $\mathtt{b} <_{\mathbb{N}} \mathtt{c}$: par `H1`, `HR` et `H3`.

· Si $\mathtt{w''} = \mathtt{S(b) :: w'''}$ et $\mathtt{w'} = \mathtt{S(b) :: w''''}$ avec $\mathtt{w'''} \prec_{|\cdot|} \mathtt{w''''}$: par `H2`. □

(*iii*). Découle de (*i*) et (*ii*). □

Nous avons donc obtenu la bonne fondation de $\prec$. Nous exploitons ce résultat dans la section suivante pour mettre en œuvre notre méthode de preuve de terminaison. Nous y verrons en particulier comment la bonne fondation de $\prec$ nous donnera à très peu de frais la bonne fondation des divers ordres définis par plongement.

4. En pratique

Avant de donner des exemples d'application du résultat de la section précédente, précisons notre procédure pour établir une preuve de terminaison avec Coq.

On considère des fonctions décurryfiées, de type $(A_1 * \ldots * A_n) \to B$ puisqu'on a potentiellement besoin de s'intéresser à plusieurs arguments d'une fonction en même temps. On se donne une fonction $m_i : A_i \to \mathbb{N}$ pour chacun des $A_1, \ldots, A_n$. On procède ensuite aux étapes suivantes :

1. Définir la fonction de mesure $mw : (A_1 * \ldots * A_n) \to \omega^\omega$.
2. Définir la relation d'ordre associée R sur $(A_1 * \ldots * A_n)$.
3. Utiliser la clause `Program Fixpoint` avec R pour définir la fonction.
4. Prouver les obligations engendrées :
   a. appels récursifs effectivement décroissants vis-à-vis de la mesure donnée ;
   b. bonne fondation de R.

4.1. Quelques outils

On peut outiller les étapes **2.** et **4.b.** de manière à rendre leur accomplissement systématique.

Définition de l'ordre En pratique, étant donnée une fonction de mesure ordinale `mw` sur un type A (qui correspond à $(A_1 * \ldots * A_n)$ dans la notation précédente), on crée l'ordre à l'aide de :

```
Definition make_mwlt (A:Set) (mw : A -> list nat) (a1 a2:A) : Prop :=
  (wlt (mw a1) (mw a2)).
```

Bonne fondation La bonne fondation de tout ordre défini à partir d'une mesure ordinale est donnée par la bonne fondation de l'ordre sur les ordinaux. On établit donc le résultat général :

```
Lemma Acc_mwlt : forall (A:Set) (mw: A -> list nat),
  forall (x:A), (Acc (fun x1 x2 => (wlt (mw x1) (mw x2))) x).
```

La bonne fondation de l'ordre défini par `(fun x1 x2 => (wlt (mw x1) (mw x2)))` s'obtient aisément en Coq par une induction bien fondée sur nos ordinaux. Le lemme `well_founded_ind` de Coq nous donne le principe d'induction à partir de l'ordre `wlt` (le $\prec$ de la section précédente) que l'on a montré bien fondé.

Ce résultat nous permet de définir la tactique

```
Ltac by_Acc_mwlt mwlt :=
  unfold Wf.MR; unfold well_founded; intros; unfold mwlt; apply Acc_mwlt.
```

que l'on pourra systématiquement utiliser pour satisfaire l'obligation de preuve de bonne fondation engendrée par l'utilisation de `Program Fixpoint`.

Sur l'ordre `wlt` Pour les obligations de preuve de décroissance, il s'est avéré utile de disposer de deux lemmes généralisant la définition de la relation `wlt` :

```
Lemma wlt_lt_gen : forall (a1 a2:nat) (w1 w2:list nat),
  (length w1) = (length w2) -> (lt a1 a2)
    -> (wlt (cons a1 w1) (cons a2 w2)).

Lemma wlt_wlt_gen : forall (a:nat) (w1 w2:list nat),
  (length w1) = (length w2) -> (wlt w1 w2)
  -> (wlt (cons a w1) (cons a w2)).
```

Ils évitent, en pratique, la décomposition répétitive des entiers en tête des listes représentant les ordinaux. Nous l'avions conservée explicite dans nos définitions pour des raisons de commodité dans la preuve de bonne fondation. Celle-ci état maintenant acquise, il devient plus commode d'utiliser ces énoncés plus généraux dans les preuves de décroissance.

4.2. Exemple simple et complet

Pour donner une idée de l'ensemble des étapes à accomplir, nous revenons sur l'exemple de la fonction d'Ackermann.

La mesure ordinale est définie sur un couple d'entiers comme la liste qui les contient (dans l'ordre du couple) :

```
Definition mw_natxnat (xy:nat*nat) :=
  match xy with
    (x,y) => (cons x (cons y nil))
  end.
```

La relation d'ordre utilise le constructeur de relation `make_wlt` :

```
Definition lex_natxnat (xy1 xy2:nat*nat) :=
    (make_mwlt (nat*nat) mw_natxnat).
```

Cet ordre est bien l'ordre lexicographique sur les couples d'entiers, et nous pouvons montrer :

```
Lemma lex_natxnat_fst : forall (x1 y1 x2 y2:nat),
  (lt x1 x2) -> (lex_natxnat (x1,y1) (x2,y2)).
```

et

```
Lemma lex_natxnat_snd : forall (x y1 y2:nat),
  (lt y1 y2) -> (lex_natxnat (x,y1) (x,y2)).
```

Ces deux lemmes s'obtiennent facilement avec `wlt_lt_gen` et `wlt_wlt_gen`.

La fonction

```
Program Fixpoint ack (xy:nat*nat) {wf lex_natxnat xy} :=
  match xy with
    (0, y) => (S y)
  | (S x, 0) => (ack (x, S 0))
  | (S x, S y) => (ack (x, ack (S x, y)))
  end.
```

Les preuves de décroissances Les deux premières, qui demandent d'établir que `(x, S 0)` est plus petit que `(S x, 0)` et que `(S x, y)` est plus petit que `(S x, S y)`, sont immédiates avec `lex_natxnat_fst` et `lex_natxnat_snd`.

```
Obligation 1.
apply lex_natxnat_fst.
auto with arith.
Qed.

Obligation 2.
apply lex_natxnat_snd.
auto with arith.
Qed.
```

La troisième, quoique formulée de manière troublante :

```
  [..]
  ack : forall xy : nat * nat, lex_natxnat xy (n, n0) -> nat
  [..]
  Heq_xy : (S x, S y) = (n, n0)
  ============================
   lex_natxnat (x, ack (S x, y) (ack_obligation_2 (n, n0) ack x y Heq_xy))
     (n, n0)
```

ne pose guère plus de difficulté. Il faut simplement utiliser l'équation en hypothèse :

```
Obligation 3.
apply lex_natxnat_fst. inversion Heq_xy. auto with arith.
Qed.
```

La bonne fondation Elle est immédiate avec notre tactique `by_Acc_mwlt` :

```
Obligation 4.
by_Acc_mwlt lex_natxnat.
Defined.
```

4.3. Ordre pour les listes

Les exemples de mesure ordinale que nous donnons pour les listes utilisent la fonction `length` comme fonction de projection des listes dans les entiers.

Ordre basé sur la longueur On définit cet ordre par :

```
Definition mw_list (A:Set) (xs: list A) :=
  (cons (length xs) nil).

Definition lt_len_list (A:Set) :=
  (make_mwlt (list A) (mw_list A)).
```

Le passage par l'encapsulation de la longueur dans une liste peut sembler inutilement alambiqué mais souvenons-nous que nous explorons un procédé systématique de définition d'ordres. Le prix à payer pour ce détour est léger (ouverture des définitions) quand la bonne fondation nous est donnée automatiquement.

On peut alors définir la fonction de remontée de la valeur maximale dans une passe de tri à bulle de la manière attendue :

```
Program Fixpoint bubble (xs:list nat) {wf (lt_len_list nat) xs} :=
  match xs with
      nil => nil
    | (cons x nil) => (cons x nil)
    | (cons x1 (cons x2 xs)) =>
      if (ltb x1 x2) then (cons x1 (bubble (cons x2 xs)))
      else (cons x2 (bubble (cons x1 xs)))
  end.
```

où `ltb` est la comparaison booléenne de deux entiers.

Récurrence sur deux listes On peut définit l'ordre lexicographique sur les longueurs de deux listes en utilisant l'ordre lexicographique sur les couples d'entiers :

```
Definition mw_listxlist (A:Set) (xys: list A * list A) :=
  match xys with
    (xs,ys) => (mw_natxnat (length xs, length ys))
  end.

Definition lex_listxlist (A:Set) :=
  (make_mwlt (list A * list A) (mw_listxlist A)).
```

On utilise cet ordre pour la définition de la fusion de listes :

```
Program Fixpoint merge (xys: list nat * list nat) {wf (lex_listxlist nat) xys} :=
  match xys with
      (nil, ys) => ys
    | (xs, nil) => xs
    | (cons x xs, cons y ys) =>
      if (ltb x y) then (cons x (merge (xs, (cons y ys))))
      else (cons y (merge ((cons x xs), ys)))
  end.
```

Ordre lexicographique interne C'est l'analogue de l'ordre exhibé pour l'exemple de la fonction de calcul de la somme des éléments d'une liste (section 2, page 4). Nous l'appliquons ici à la définition de la concaténation de listes :

```
Definition m_listlist (A:Set) (xss : list (list A)) :=
  match xss with
      nil => nil
    | (cons xs _) => (cons (length xss) (cons (length xs) nil))
  end.

Definition lt_listlist (A:Set) (xss yss : list (list A)) :=
  (wlt (m_listlist A xss) (m_listlist A yss)).
```

La définition de la fonction de concaténation est définie directement :

```
Program Fixpoint list_concat (xss : list (list A)) {wf (lt_listlist A) xss} :=
  match xss with
      nil => nil
    | (cons nil xss) => (list_concat xss)
    | (cons (cons x xs) xss) => (cons x (list_concat (cons xs xss)))
  end.
```

4.4. Ordre pour les arbres binaires

Nous donnons la définition suivante des arbres binaires et de la fonction de calcul de leur taille :

```
Inductive btree (A:Set) : Set :=
  Empty : (btree A)
| Node : (btree A) -> A -> (btree A) -> (btree A).

Fixpoint btree_size (A:Set) (bt:btree A) :=
  match bt with
      Empty => 0
    | (Node bt1 x bt2) => S (plus (btree_size A bt1) (btree_size A bt2))
  end.
```

Nous allons traiter deux exemples sur les arbres binaires. Le premier est celui dit du «peigne» d'un arbre binaire et le second est inspiré de [5] : décompte des feuilles dans une liste d'arbres binaires. Ce dernier sera l'exception à notre méthode de génération systématique de mesure ordinale où la mesure des arguments se réduit à une mesure simple de leur taille.

Le peigne La mesure définie donne le couple constitué de la taille de l'arbre et de la taille de son sous-arbre gauche, pour les arbres non vides. La mesure de l'arbre vide n'a pas d'incidence sur la décroissance voulue :

```
Definition m_btree (A:Set) (bt:btree A) :=
  match bt with
      Empty => nil
    | (Node bt1 x bt2) => (cons (btree_size A bt) (cons (btree_size A bt1) nil))
  end.

Definition lt_btree (A:Set) (bt1 bt2:btree A) :=
  (wlt (m_btree A bt1) (m_btree A bt2)).
```

L'ordre induit permet d'obtenir la terminaison de :

```
Program Fixpoint to_list (bt:btree A) {wf (lt_btree A) bt} :=
  match bt with
      Empty => nil
    | (Node Empty x bt) => (cons x (to_list bt))
    | (Node (Node bt1 x1 bt2) x2 bt3) => (to_list (Node bt1 x1 (Node bt2 x2 bt3)))
  end.
```

sans effort particulier.

L'exception La fonction dont nous voulons obtenir la terminaison est définie sur les listes d'arbres binaires (l'ordre est encore à trouver) :

```
Program Fixpoint count_tips (bts:(list (btree A))) {wf ??? } :=
  match bts with
      nil => 0
    | (cons Empty bts) => S (count_tips bts)
    | (cons (Node bt1 x bt2) bts) => (count_tips (cons bt1 (cons bt2 bts)))
  end.
```

La difficulté ici est que la longueur de la liste croît dans l'appel récursif de la troisième équation. Nous ne pouvons donc appliquer la méthode qui donne un ordre lexicographique sur la longueur de la liste et la taille de son premier élément, comme nous l'avions fait pour `list_concat`, par exemple. Toutefois, on peut obtenir la terminaison de `count_tips` en mesurant «en profondeur» la liste d'arbres binaires :

```
Fixpoint list_btree_size (A:Set) (bts:list (btree A)) : nat :=
  match bts with
      nil => 0
    | (cons bt bts) => (plus (btree_size A bt) (list_btree_size A bts))
  end.
```

Nous pouvons alors retrouver un substitut à l'ordre multi-ensemble que Manna et Dershowitz utilisent dans leur papier :

```
Definition mw_list_btree (A:Set) (bts:list (btree A)) : (list nat) :=
  (cons (list_btree_size A bts) (cons (length bts) nil)).

Definition lt_list_btree (A:Set) :=
  (make_mwlt (list (btree A)) (mw_list_btree A)).
```

Jusqu'ici, nous avions utilisé des mesures entières *compatibles avec les constructeurs* (par exemple `(length xs)` est inférieure à `(length (cons x xs))`). Nous étions donc en fait restés dans le cadre d'une récurrence structurelle généralisée. Notre dernier exemple montre que, échappant à ce cadre, la méthode de plongement dans un ordinal intercepte un large éventail de techniques de preuves de terminaison dont la réalisation reste assez simple en Coq.

5. Conclusion

Notre encodage de ω^ω et de leur ordre nous permet de mettre en application dans Coq la méthode de preuve de terminaison de Monin et Simonot. Avoir prouvé la bonne fondation des mesures générées par ce système nous laisse pour seules obligations les preuves de décroissances qui jusqu'ici

ont systématiquement été triviales, permettant ainsi de prouver de façon efficace et élégante des programmes non reconnus par l'analyse structurelle de Coq.

Plus précisément, le système de Monin et Simonot permet de prouver la terminaison de récursions primitives, multiples [3] et générales, même en se limitant à de simples mesures de taille des arguments pour les m_i. Bien que la plupart de nos exemples restent dans ce cadre et malgré les nombreux éléments de systématisation que nous proposons, un certain exotisme peut parfois être requis dans la construction de ces plongements. Le degré de liberté qu'ils représentent est en effet mal maîtrisé et générer automatiquement la mesure que nous employons dans notre dernier exemple sur les arbres binaires serait difficile. Au vu du cadre très général qu'offre cette approche, il serait intéressant de l'intégrer à Coq sans se limiter à des mesures compatibles avec les constructeurs, ce qui nécessite donc encore un travail pour la génération automatique de plongements m_i non standards.

Références

[1] A. Balaa and Y. Bertot. Fonctions récursives générales par itération en théorie des types. In *Journées francophones des langages applicatifs*, 2002.

[2] P.-L. Bégay, P. Manoury, and I. Rakotonirina.
Disponible à : http ://www.pps.univ-paris-diderot.fr/~eleph/Recherche/listordi.v.

[3] P. Castéran. Bibliothèque AccP pour Coq.
Disponible à : http ://www.labri.fr/perso/casteran/Cantor/HTML/AccP.html.

[4] N. Dershowitz. Orderings for term rewriting systems. In *Foundations of Computer Science*, 1979.

[5] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. In *Colloquium on Automata, Languages and Programming (Graz, Austria)*. Springer-Verlag, Berlin, 1979.

[6] G. Huet, G. Kahn, and C. Paulin-Mohring. The coq proof assistant, 1997.

[7] D. Knuth and P. Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*. Springer, 1983.

[8] P. Manolios and D. Vroon. Ordinal arithmetic in ACL2. In *ACL2 workshop*, 2003.

[9] P. Manoury. A user's friendly syntax to define recursive functions as typed λ-terms. In *Types for Proofs and Programs*. Springer, 1995.

[10] P. Manoury and M. Simonot. Automatizing termination proofs of recursively defined functions. *Theoretical Computer Science*, 1994.

[11] F. Monin and M. Simonot. An ordinal-measure-based procedure for termination of functions. *Theoretical Computer Science*, 2001.

[12] M. Sozeau. Subset coercions in Coq. In *Types for Proofs and Programs*, 2007.

3. en anglais : *unnested multiple recursion*