Fonctions récursives générales par itération en théorie des types

Antonia Balaa & Yves Bertot

INRIA Sophia-Antipolis
2004 Route des lucioles, B.P. 93
06902 Sophia Antipolis Cedex, FRANCE
`Antonia.Balaa, Yves.Bertot@sophia.inria.fr`

Résumé

Dans le calcul des constructions inductives, les outils de construction de fonctions sont soumis à une condition de terminaison: Il est impossible de construire des fonctions qui ne terminent pas. Les fonctions structurellement récursives sont bien intégrées dans le système car la condition s'exprime par une propriété syntaxique, ce n'est pas le cas pour les fonctions récursives générales. Nous proposons une méthode pour construire des fonctions récursives générales, à partir d'une définition à la `ML` ou plus précisèment de l'équation de point fixe de la fonction, et d'une relation bien fondée permettant de prouver la terminaison de la fonction. Nous produisons ainsi, la fonctionnelle associée, la définition de la fonction à partir du théorème de terminaison et la preuve que la fonction ainsi définie vérifie bien l'équation de point fixe donnée. Un outil de construction de fonctions a été développé dans le système Coq et a été experimenté sur des exemples de fonctions dont la fonction de logarithme discret à base deux détaillée dans ce document.

Mots Clés: fonctions récursives générales, calcul des constructions, Coq

1. Introduction

Les systèmes de preuves basés sur la théorie des types sont des langages de programmation fonctionnels. Mais paradoxalement, la programmation de fonctions récursives peut paraître très difficile pour les utilisateurs. La difficulté n'apparaît pas pour les fonctions récursives structurelles qui sont simples à décrire et simples à utiliser. En revanche elle devient flagrante pour les fonctions récursives générales. Pour de nombreux utilisateurs, même expérimentés, la seule technique de programmation qui ait été pratiquée est celle de la récursion structurelle, et la récursion bien fondée fait partie des outils dont on connaît l'existence, mais dont on redoute le besoin.

Lorsque l'on programme une fonction récursive en théorie des types, on doit souvent fournir simultanément l'algorithme et la preuve qu'il termine. Pour les fonctions récursives structurelles, ceci se fait en isolant l'argument principal, tel que les appels récursifs ne pourront se produire que sur des sous-termes stricts de cet argument principal. Pour les fonctions récursives bien fondées, ceci se fait également en isolant un argument principal, mais aussi en exhibant une relation bien fondée et en montrant que les appels récursifs n'ont lieu que sur des arguments qui sont des prédécesseurs de l'argument principal pour cette relation bien fondée. Le système de preuve assure cette propriété au moment où il vérifie que l'algorithme est bien typé [8], donc on est encore dans la situation où l'algorithme et sa terminaison doivent être vérifiés simultanément.

Pour le programmeur et l'enseignant, séparer les contraintes représente un progrès considérable: on peut se concentrer d'une part sur l'algorithme en le lisant comme un programme dans un langage de programmation usuel (comme les langages de la famille ML) et d'autre part sur les arguments de terminaison. C'est à cet objectif que le travail présenté dans cet article tente de répondre.

L'idée développée ici tire son origine du théorème du point fixe. On sait que toute fonction F monotone et continue dans un ordre partiel complet admet un plus petit point fixe. Il est de tradition en sémantique dénotationelle de justifier la construction des fonctions récursives par ce théorème. Le point le plus important est que ce plus petit point fixe est la limite, lorsque n tend vers l'infini de $F^n(\perp)$, où $\perp$ est l'élément minimal de l'ordre partiel complet considéré. Notre première contribution est d'éviter de construire explicitement un ordre partiel complet en remplaçant l'utilisation de l'élément $\perp$ par une quantification universelle. Notre seconde contribution est de simuler la notion de limite par une suite de quantifications universelles et existentielles bien choisies.

Le résultat de l'étude se concrétise en un outil automatisable (nous en avons construit une maquette) qui permet la programmation en Coq [3] de fonctions récursives générales presque aussi naturellement que dans un langage de programmation fonctionnel. Nous pensons que la technique s'appliquera avec des variations mineures dans tous les systèmes de démonstration basés sur la théorie des types.

Bien sûr, ce travail mérite une comparaison avec le travail de Parent [9] dont la tactique `Program` fournit une construction syntaxique adaptée pour la définition de fonctions récursives bien fondées. Ce travail contribue également à séparer la construction de l'algorithme et la preuve de la terminaison: l'algorithme est donné dans une première phase de la preuve permettant la définition de la fonction récursive cherchée, tandis que les preuves de correction ou de terminaison sont laissées pour plus tard. Néanmoins, la fonction obtenue synthétise l'ensemble des informations en un seul et même objet, difficile à observer pour le programmeur. Dans notre travail, nous définissons plusieurs objets, dont l'un décrit les calculs effectués dans la fonction récursive, le second décrit la preuve que l'on définit bien une fonction récursive, le troisième est la fonction récursive elle-même, et le dernier est la preuve que cette fonction satisfait bien l'équation censée la définir: l'équation de point fixe.

Par comparaison, les fonctions récursives obtenues par la tactique `Program` ne sont pas fournies avec une équation de point fixe, de sorte que l'on doit s'assurer dès la définition de la fonction qu'elle vérifiait bien la spécification attendue. Par conséquent la démonstration définissant la fonction récursive doit combiner non seulement la description de l'algorithme et la preuve qu'il termine, mais aussi la preuve qu'il satisfait la spécification. En conséquence, la construction de la fonction récursive n'en est que plus complexe et il est difficile de montrer qu'une fonction répond à plusieurs spécifications.

Dans un travail plus récent [2], nous nous étions déjà intéressés à l'équation de point fixe, pour comprendre comment celle-ci pouvait être construite à partir de la fonction obtenue, par exemple, à l'aide de la tactique `Program`. La solution exposée dans [2] est excessivement complexe et n'atteint pas l'objectif de séparer les contraintes: description de l'algorithme, preuve de terminaison. Pourtant, cette solution reste valable car elle s'applique plus souvent, en particulier si la fonction présente de la récursion imbriquée, que la solution décrite dans le présent article et fournit bien une équation de point fixe. En outre, les travaux de [2] et ceux présentés ici ont la même approche: la construction des différentes démonstrations nécessaires à la définition d'une fonction récursive suit la structure de la fonction récursive elle-même. Nous citons d'autres travaux reliés en conclusion, lorsque nous abordons les évolutions possibles de ce travail.

2. Technique de la Récursion par itération

Le principe de cette technique est d'isoler les calculs qui sont effectués entre deux appels successifs d'une fonction récursive. Ensuite nous introduisons un paramètre k supplémentaire qui limite le nombre de répétitions de ces calculs. En faisant varier ce paramètre, nous obtenons des approximations de la fonction récursive étudiée. C'est sur les valeurs de ces approximations que nous raisonnons.

En général, la définition d'une fonction fait intervenir une expression de la forme suivante:

$$(f\ x) = expr$$

où x apparaît dans l'expression $expr$, pour parler de fonctions récursives nous autorisons aussi l'apparition de f dans cette expression. Nous associons à cette définition récursive une fonction du second ordre, la fonctionnelle, que nous notons F, qui associe la valeur $expr$ à f et x. Elle aura la forme suivante:

$$F \;=\; \lambda f.\lambda x.\ expr$$

L'équation définissant notre fonction récursive est alors équivalente à l'équation suivante:

$$(f\ x) = (F\ f\ x)$$

Tout au long de l'article, nous allons utiliser comme exemple conducteur la fonction `logarithme` discret à base deux. Cette fonction est définie par l'équation de point fixe suivante:

```
(x:nat)(log x)=
       Cases x of
             0 => 0
          | (S 0) => 0                                            (1)
          | (S (S x')) => (S (log (div2 (S (S x')))))
       end
```

La fonctionnelle `F_log` associée peut être construite sur le même modèle par la définition suivante:

```
Definition F_log : (nat -> nat) -> (nat -> nat) :=
   [log:nat->nat][x:nat]Cases x of
                              0 => 0
                           | (S 0) => 0
                           | (S (S x')) => (S (log (div2 (S (S x')))))
                        end.
```

Si la fonction f est du type: $A \rightarrow B$ et x a le type A alors $(f\ x)$ a le type B, et la fonctionnelle F a le type: $(A \rightarrow B) \rightarrow (A \rightarrow B)$.
Pour notre exemple conducteur, `log` a le type: `nat -> nat` et `F_log` a le type: `(nat -> nat) -> (nat -> nat)`.

2.1. Itérations de la fonctionnelle

Dire que la fonction f termine pour x c'est dire qu'il y a un nombre k maximal d'appels récursifs imbriqués de la fonction f pour obtenir la valeur finale.
Intuitivement, la même valeur finale est obtenue par la fonction

$$(F^k\ g)\ x$$

où g est une fonction arbitraire de type: $A \rightarrow B$ et F^k est définie par les deux équations suivantes:

$$\begin{array}{rcl}(F^0\ g) & = & g\\ (F^{(k+1)}\ g) & = & (F\ (F^k\ g)).\end{array}$$

En fait, si k existe tel que $(F^k\ g\ x)$ ne dépend pas de g alors $(F^k\ g\ x) = (F^k\ f\ x) = (f\ x)$.

Cette remarque permet également de construire f. Montrer que f est bien définie partout, c'est justement montrer qu'on peut associer à toute valeur en entrée x une valeur v telle que $(F^k\ g\ x) = v$

quelque soit g et quelque soit k *«assez grand»*; et cette valeur v est précisèment $(f\ x)$.
Dans le cadre de notre exemple conducteur, effectuons la démonstration de cette propriété fondamentale: $(\texttt{F\_log}^{\texttt{k}}\ g\ x) = (\texttt{log}\ x)$ et examinons les deux fonctions `log` et `F_log`:
Pour $x = 1$ on a:
$(\texttt{F\_log}\ g\ x) =$ `Cases (S 0) of`

```
                 0 => 0
               | (S 0) => 0
               | (S (S x')) => (S ( g (div2 (S (S x')))))
             end
```

en simplifiant l'expression `Cases` on obtient `0`, donc $(\texttt{F\_log}\ g\ x) = \texttt{0}$ et ne dépend pas de g.
De même,

```
(log x)= Cases (S 0) of
               0 => 0
             | (S 0) => 0
             | (S (S x')) => (S (log (div2 (S (S x')))))
         end
```

en simplifiant, on obtient encore la valeur `0`.
Le même comportement apparaît pour $x = 0$.

Si $x > 1$ nous savons qu'il existe un x' tel que $x = x' + 2$. Si l'on dispose de k tel que, pour tout g on ait:

$$(\texttt{F\_log}^{\texttt{k}}\ g\ (\texttt{div2}\ x)) = (\texttt{log}\ (\texttt{div2}\ x))$$

alors $(\texttt{F\_log}^{\texttt{k}}\ g\ (\texttt{div2}\ x))$ ne dépend pas de g et l'on a:

$(\texttt{F\_log}^{(\texttt{k+1})}\ g\ (\texttt{div2}\ x)) =$ `Cases` x `of`

```
                                  0 => 0
                                | (S 0) => 0
                                | (S (S x')) => (S (F_log^k g (div2 (S (S x')))))
                              end
```

En simplifiant l'expression de filtrage, on obtient

$$(\texttt{F\_log}^{(\texttt{k+1})}\ g\ x) = (\texttt{S}\ (\texttt{F\_log}^{\texttt{k}}\ g\ (\texttt{div2}\ x))) = (\texttt{S}\ (\texttt{log}\ (\texttt{div2}\ x))).$$

De même,

```
(log x)= Cases x of
              0 => 0
            | (S 0) => 0
            | (S (S x')) => (S (log (div2 (S (S x')))))
         end
```

et en simplifiant on a:

```
(log x) = (S (log (div2 (S (S x'))))) = (S (log (div2 x))).
```

En d'autres termes, pour les k premiers "instants" du calcul, les deux fonctions $(F^k\ g)$ et f ont la même structure et la valeur retournée par la fonction f coïncide avec la valeur de $(F^k\ g)$ pour certaines valeurs de x. La fonctionnelle F^k nous donne à chaque étape une approximation de la valeur de f. C'est quand $(F^k\ g\ x)$ ne dépend pas de g que l'approximation est exacte. Ceci nous mène au *théorème de terminaison.*

2.2. Théorème de terminaison

Le théorème de terminaison a le type suivant:

$$Definition\ \ f\_terminates :$$
$$\forall x : A.\{v : B | (\exists p : nat.\ \ \forall k : nat.(p < k) \rightarrow \ \ \forall g : A \rightarrow B\ \ ((F^k\ g\ x) = v)\}.$$

Ce théorème exprime que pour tout x, en itérant k fois la fonctionnelle F, avec k assez grand, les approximations de $(F^k\ g\ x)$ se stabilisent et nous donnent une valeur v indépendante de la fonction de départ g. Notons que nous utilisons ici deux types de quantifications existentielles, la première est notée par $\{v : B|...\}$ et exprime que la valeur v est importante, tandis que la deuxième, notée $\exists p : nat.\ x...$, indique que la valeur de p n'est pas importante en soi-même. Cette distinction est pertinente pour les outils d'extraction [10] : après extraction la fonction obtenue calcule effectivement v mais ne calcule pas p.

Pour décrire F^k, nous utilisons la fonction d'itération `iter` définie de la façon suivante:

```
Fixpoint iter [A:Set;n:nat] : (A -> A) -> A -> A :=
  [f:A->A][g:A]Cases n of
                   O => g
                 | (S n') => (f (iter A n' f g))
              end.
```

Dans notre exemple conducteur, on trouve donc ce type:

```
Theorem log_terminates :
  (x:nat)
   {v:nat |(Ex [p:nat]
              (k:nat)(lt p k)->
              (g:nat -> nat)
              (iter (nat -> nat) k F_log g x)=v)}.
```

Nous détaillerons la preuve de ce théorème dans la section 3.

2.3. Construction de la fonction visée et de son équation de point fixe

Le théorème de terminaison est une fonction qui retourne la paire constituée d'une valeur et d'une preuve que cette valeur est le résultat de k itérations de F pour k *assez grand*. Nous pouvons extraire la fonction f définie à partir de la fonction $(f\_terminates)$ qui n'est autre que la preuve du théorème de terminaison, simplement en oubliant la partie preuve.

$$Definition\ f : A \rightarrow B := \lambda x : A.\ \ Case\ (f\_terminates\ x)\ of\ [v : B;\ h :?]\ v\ \ end.$$

Pour notre exemple conducteur du `logarithme`, elle sera définie par:

```
Definition log := [x:nat]Case (log_terminates x) of [v:nat;h:?]v end.
```

Nous construisons maintenant l'équation de point fixe dont nous rappelons qu'elle est de la forme:

$$\forall x.(f\ x) = expr.$$

On peut prouver que le comportement de la fonction ainsi définie satisfait cette équation de point fixe. Nous détaillerons cette preuve dans la section 4. Pour notre exemple conducteur, c'est l'équation (1) donnée en section 2.

3. Preuve du théorème de terminaison

Nous effectuons la preuve du théorème de terminaison par récurrence bien fondée. Une relation bien fondée est une relation pour laquelle il n'existe pas de suite infinie décroissante. La preuve par récurrence bien fondée passe par la vérification que les arguments des appels recursifs sont des prédécesseurs de l'argument initial pour une relation bien fondée. Ceci nous assure que les appels récursifs ne forment pas une chaîne infinie. Pour guider le processus de preuve, l'utilisateur doit fournir la relation bien fondée et la preuve qu'elle est bien fondée. La relation choisie doit être adaptée à la structure de la fonction.

Pour notre exemple conducteur, la relation choisie est `lt`, la preuve qu'elle est bien fondée est `lt_wf`. Ces deux objets sont fournis dans les librairies de Coq.

La structure de la preuve du théorème de terminaison est construite par analyse syntaxique du code de la fonctionnelle F qui comporte:

- Une descente récursive dans les traitements par cas imbriqués, représentée dans la suite par une fonction ϕ,
- Un traitement particulier pour les expressions correspondant aux cas de base, représentée dans la suite par une fonction ψ,
- Un traitement particulier pour les expressions correspondant aux appels récursifs de la fonction étudiée, représentée dans la suite par ρ.

Pour notre exemple conducteur, la structure de la fonctionnelle apparaissant dans la figure 1 est décomposée par le système Coq en la structure de la figure 2.

```
Cases x of
       O => O
      |(S O) => O
      |(S (S p)) => (S (log (div2 (S (S p)))))
end.
```

Figure 1: Expression fournie par l'utilisateur

```
Cases x of
      O => O
      |(S p') =>  Cases p' of
                      O => O
                    |(S p) =>  (S (log (div2 (S (S p)))))
                   end
end.
```

Figure 2: Expression générée par le système Coq

Pour la structure de la figure 2, les boîtes en trait plein correspondent aux arguments successifs de la fonction ϕ; les boîtes en pointillés simples correspondent aux cas de base, traités par ψ; et celles en pointillés alternés correspondent au cas de récurrence, traités par ρ. Nous avons développé ces fonctions ϕ, ψ et ρ en OCAML.

3.1. Construction de filtrage

Pour un type inductif T défini avec les constructeurs c_1, c_2, ... d'arité n_1, n_2, ..., la construction du filtrage prend la forme suivante:

```
Cases x of
      (c1 x1,1 ... xn1,1) => e1
    |  ...
    | (ck x1,k ... xnk,k) => ek
    ...
end
```

Le système COQ transforme toute construction de filtrage en une construction `Case` qui permet de voir chaque règle de filtrage comme une λ-abstraction. Elle prend la forme suivante:

```
Case x of
      λx1,1 ... xn1,1 . e1
    |  ...
    | λx1,k ... xnk,k . ek
    ...
end
```

Notons e cette expression, la fonction ϕ prend en argument un fragment de la fonction F et un contexte décrivant l'ensemble des λ-abstractions qui l'entourent. Si ϕ est appelée avec e comme argument alors elle devra se rappeler récursivement avec $e_1 \ldots e_k$, avec les environnements mis à jour pour tenir compte des λ-abstractions dans chaque branche. En plus, nous ajoutons dans le contexte, pour la démonstration correspondant à chaque branche, une hypothèse *heq* de la forme $x = (c_i\ x_{1,i}\ \ldots\ x_{n_i,i})$. La preuve a alors la forme suivante:

$$
\begin{array}{l}
\phi(\mathtt{e},\sigma) = \text{Case } x \text{ of} \\
\qquad \lambda x_{1,1} : t_{1,1}\ \ldots\ x_{n_1,1} : t_{n_1,1}.\ \lambda heq : x = (c_1\ x_{1,1}\ \ldots\ x_{n_1,1}) \\
\qquad (\phi(e_1, \sigma + [x_{1,1}; \ldots; x_{n_1,1}; heq])) \\
\qquad \mid\ \ldots \\
\qquad \mathtt{end}
\end{array}
$$

Raisonner sur l'expression e dans un environnement donné σ c'est raisonner sur chaque expression $e_1 \ldots e_k$ dans un nouvel environnement après avoir ajouté dans l'environnement initial σ toutes les variables introduites par les λ-abstractions, ainsi qu'une égalité qui conserve la trace du cas dans lequel on se trouve. Nous remarquons que la fonction ϕ suit exactement la structure de la fonction et que la preuve suit aussi cette structure. Nous retrouvons ici la correspondance entre structure de fonction et structure de preuve déjà utilisée dans [2].

La tactique utilisée pour ajouter les égalités peut être définie en COQ sans faire appel à OCAML. En voici la définition:

```
Tactic Definition CaseEq x :=
        Generalize (refl_equal ? x);
        Pattern -1 x;
        Case x.
```

3.1.1. Le code ML correspondant

La fonction `main_terminates` représente la fonction ϕ. Elle utilise deux fonctions auxiliaires `mk_intros_and_continue` et `find_call_occs`.

- La fonction `mk_intros_and_continue` est appelée pour chaque règle d'une construction de filtrage et a pour but d'introduire toutes les λ-abstractions dans le contexte. Le premier argument de cette fonction est un booléen, si sa valeur est `true` alors il faut introduire une hypothèse supplémentaire appelée `heq` en plus des lambdas qui est l'équation d'égalité entre la variable et le cas produit par la tactique `CaseEq`.

- La fonction `find_call_occs` est appelée lorsque l'expression n'est pas une construction de filtrage. Elle prend en argument le nom de la fonction à chercher (dans notre exemple c'est `log`) et en deuxième argument l'expression courante (c'est-à-dire la partie droite de chaque règle de filtrage du `Cases`), elle nous retourne une valeur d'un type option: None pour indiquer le cas de base et (Some x) pour indiquer le cas récursif. Dans le cas récursif, $expr$ a la forme suivante (pour cette expérience nous nous sommes limités aux fonctions à un seul argument):

  $$expr = C[(f \quad arg)]$$

  où C est un contexte. La fonction `find_call_occs` nous retourne une paire d'arguments: $(\chi,\ arg)$, le premier correspond au contexte dans lequel se trouve la fonction:

  $$\chi = \lambda x.\ C[x]$$

  et le second est l'argument de cette fonction arg.

  Dans notre exemple conducteur, la fonction `find_call_occs` est appelée trois fois.

  1. Sur `O` elle retourne None,
  2. Le deuxième appel est comme le premier, en d'autres termes, sur `(S O)` le résultat est le même que précédemment,
  3. Sur `(S (log (div2 (S (S p)))))` elle retourne (Some $(\lambda x.$(S $x)$, (`div2` (S (S p))))), nous avons $\chi = \lambda x.$(S x) et $arg =$ (`div2` (S (S x p))).

La fonction `main_terminates` prend en argument l'hypothèse de récurrence $Hrec$ obtenue grâce à la récurrence bien fondée sur la valeur en entrée et qui a la forme suivante:

$$Hrec : \forall y : A.(R\ y\ x) \Rightarrow \{v : B | \exists p : nat. \forall k.p < k \Rightarrow \forall g.(F^k\ g\ y) = v\}$$

La fonction `main_terminates` prend aussi en argument la sous-expression de F considérée, que nous notons e.

- Si e est une construction de filtrage, `main_terminates` applique la tactique `CaseEq` sur l'expression filtrée et la fonction `mk_intros_and_continue` sur toutes les règles de filtrage avec la valeur booléenne `true`. Ensuite la fonction `main_terminates` est rappelée récursivement sur le corps de chacune des λ-abstractions présentes dans la construction de filtrage.

- Sinon, `main_terminates` appelle la fonction `find_call_occs` sur e qui distingue deux cas: Soit l'expression considérée est un cas de base de la fonction étudiée, soit l'expression considérée contient un appel récursif.

3.2. Cas de base

Si la fonction `find_call_occs` indique qu'il s'agit bien d'un cas de base, la valeur à donner pour v est la valeur de e reçue en argument et la valeur à donner pour p est `0`. Nous obtenons un but qui est de la forme:

```
(k:nat; g:(nat->nat))  (lt (0) k)->(iter nat->nat F_log g k x)= v
```

Ce but est dépendant du nombre d'itérations `k` de la fonction nécessaire pour obtenir un résultat:

- Pour (`k = 0`), nous avons une hypothèse fausse ($0 < 0$) et la preuve se résout aisément (théorème (lt_n_n) fourni dans la librairie de Coq).
- Pour (`k = (S n)`), la partie gauche de l'égalité se réduit à l'expression e. On peut alors réécrire par les hypothèses heq introduites précédemment par la fonction `mk_intros_and_continue`. La preuve devient alors une application de la réflexivité de l'égalité.

Pour notre exemple conducteur, la valeur v à fournir est `0` dans les deux cas de base. Les équations engendrées par la fonction `mk_intros_and_continue` sont seulement `x=0` pour le premier cas de base et `x=(S x'')` et `x''=0` pour le second cas de base.

3.3. Cas de Récurrence

Si la fonction `find_call_occs` indique que l'expression e contient un appel récursif, elle retourne une paire (χ, arg) (pour cette expérience, nous ne considérons que les fonctions qui présentent un seul appel récursif). Il faut alors utiliser l'hypothèse de récurrence $Hrec$ sur arg. Ceci impose à l'utilisateur de fournir une preuve que arg est bien plus petit que l'argument initial après réécriture de cet argument initial par les égalités $heq \dots heq_n$ introduites par `mk_intros_and_continue`.

Pour notre exemple conducteur, il faut appliquer $Hrec$ à `(div2 (S (S p)))`, ce qui nécessite de démontrer `(lt (div2 (S (S p))) x)`. Les utilisations successives de la tactique `CaseEq` fournissent deux égalités `heq:x = (S x'')` et `heq0:x'' = (S x')`. Après réécriture par ces égalités, il faut donc démontrer `(lt (div2 (S (S p))) (S (S p)))`. C'est une démonstration que l'utilisateur fait sur le côté.

L'utilisation de $Hrec$ nous donne une valeur v' et une preuve que v' est le résultat de k itérations de F sur arg, pour tout k assez grand. Nous introduisons la valeur à fournir pour v qui est alors $(\chi\ v')$.

Pour notre exemple conducteur, la valeur à fournir pour v est `(S v')`.

A ce stade l'expression à démontrer est équivalente à la suivante:

$$(\forall k.p' < k \Rightarrow \forall g.(F^k\ g\ arg) = v')) \Rightarrow \exists p.\forall k.p < k \Rightarrow \forall g.(F^k\ g\ x) = v$$

et l'on dispose dans le contexte de plusieurs égalités qui imposent que

$$(F^k\ g\ x) = (\chi(F^{k-1}\ g\ arg))$$

Il suffit donc de fournir $(S\ p')$ comme témoin pour la quantification existentielle sur p et l'équation en prémisse permet de résoudre la preuve.

4. Preuve de l'équation de point fixe

Pour se convaincre que f est bien la fonction attendue et pour raisonner ultérieurement sur cette fonction, il est nécessaire d'obtenir une égalité de la forme

$$\forall x : A.(f\ x) = expr.$$

C'est l'équation de point fixe.

Pour prouver cette équation, nous utilisons le fait que f est définie à partir de $f\_terminates$ à l'aide d'un traitement par cas. Pour effectuer le même traitement par cas dans la preuve, nous devons seulement envoyer la tactique:

```
Intros x; Unfold f; Case (f_terminates x).
```

Notre équation de point fixe devient la formule logique suivante:

$$(\exists p : nat.\forall k.p < k \Rightarrow \forall g.(F^k\ g\ x) = v) \Rightarrow v = expr.$$

Ayant fixé p, nous savons que $p < k$ implique l'existence de k' tel que $k = (S\ k')$ et $(F^k\ g\ x) = (F\ (F^{k'}\ g)\ x)$. Notre équation devient donc:

$$(\forall k'.p < (S\ k') \Rightarrow \forall g.(F\ (F^{k'}\ g)\ x) = v) \Rightarrow v = (F\ f\ x).$$

le membre gauche de la première égalité et le membre droit de la seconde ont la même structure. Comme pour la preuve de terminaison, la preuve de l'équation peut suivre cette structure, en effectuant récursivement les traitements par cas qui apparaissent dans l'expression $expr = (F\ f\ x)$. Et encore comme pour la preuve de terminaison, ce traitement récursif s'arrête sur des expressions correspondant aux cas de base de la fonction f ou aux cas d'appels récursifs.

4.1. Descente récursive

La fonction `main_eq` a la même structure que la fonction `main_terminates` à deux variantes près. Premièrement les traitements par cas ne sont pas effectués à l'aide de la tactique `CaseEq` mais simplement à l'aide de la tactique `Case`. Ainsi nous ne créons pas d'hypothèses d'égalités dans le contexte. Deuxièmement, la fonction `mk_intros_and_continue` est appelée avec la valeur booléenne `false` pour tenir compte de l'absence de ces hypothèses d'égalité.

Lorsque l'on quitte la descente récursive dans les constructions de filtrage, c'est encore la fonction `find_call_occs` qui est utilisée pour distinguer les expressions correspondant à des cas de base et les expressions correspondant à des appels récursifs de f.

4.2. Cas de base

Pour les expressions correspondant à des cas de base, $(F\ h\ x)$ se réduit en une expression e indifféremment que h soit égal à $(F^{k'}\ g)$ ou à f, on peut choisir une valeur arbitraire de k' supérieure ou égale à p, par exemple p lui-même. Notre équation se simplifie alors dans la forme suivante:

$$e = v \Rightarrow v = e$$

Ce qui se démontre sans peine.

4.3. Cas de récurrence

Pour les expressions correspondant à des cas d'appel récursif, la fonction auxiliaire `find_call_occs` retourne une paire (χ, arg) telle que $(F\ h\ x) = (\chi\ (h\ arg))$.

Notre équation prend alors la forme suivante:

$$(\forall k'.p < k' \Rightarrow \forall g.(\chi\ (F^{k'}\ g\ arg)) = v) \Rightarrow v = (\chi\ (f\ arg))$$

Ici encore, nous utilisons le fait que f est défini à partir de $f\_terminates$ à l'aide d'un traitement par cas. Ceci mène notre équation à la forme suivante:

$$(\forall k'.p < k' \Rightarrow \forall g.(\chi\ (F^{k'}\ g\ arg)) = v) \Rightarrow$$
$$(\exists p'.\forall k.p' < k \Rightarrow \forall g.(F^k\ g\ arg) = v') \Rightarrow v = (\chi\ v')$$

En fixant p', par le procédé connu sous le nom de *skolémisation*, on obtient la formule suivante:

$$(\forall k'.p < k' \Rightarrow \forall g.(\chi\ (F^{k'}\ g\ arg)) = v) \Rightarrow$$
$$(\forall k.p' < k \Rightarrow \forall g.(F^k\ g\ arg) = v') \Rightarrow v = (\chi\ v')$$

En choisissant d'instancier les deux formules quantifiées universellement avec une même valeur choisie pour être à la fois strictement supérieure à p' et à p (nous utilisons la valeur $(Sp + p')$), on obtient la formule suivante:

$$(\chi\ (F^k\ g\ arg)) = v \Rightarrow (F^k\ g\ arg) = v' \Rightarrow v = (\chi\ v').$$

Ce qui se démontre encore sans peine.

5. Mise en œuvre pratique

Dans notre travail expérimental, nous avons construit une commande qui prend la syntaxe suivante:

`Recursive Definition` *f type R wf_ thm thm eqn*

Les arguments que l'utilisateur doit fournir sont les suivants:

1. *f,* le nom de la fonction à définir,
2. *type,* le type de la fonction à définir,
3. *R,* la relation bien fondée qui sera utilisée pour justifier la terminaison,
4. *wf_ thm,* le théorème qui prouve que R est bien-fondée,
5. *thm,* le théorème qui sera utilisé pour montrer que l'appel récursif se fait bien sur un prédécesseur de l'argument principal (on suppose qu'il n'y a qu'un seul appel récursif dans la fonction),
6. *eqn,* l'équation qui est censée définir la fonction récursive. Cette équation doit avoir la forme suivante:
$$\forall x : A.(f\ x) = \ldots$$

A partir de ces données notre outil expérimental engendre la fonctionnelle F, la preuve de terminaison $f\_terminates$, la fonction f, et l'équation de point fixe $f\_equation$.

Les lignes qui suivent sont la transcription d'une session utilisant cet outil pour notre exemple conducteur. L'expérience a été réalisée avec la version "beta" de Coq version 7. L'adaptation à une version plus récente nécessite simplement de s'adapter au changement de types des fonctions internes de Coq. Le texte fourni par l'utilisateur est écrit dans cette police de caractères (sans-sérif), tandis que le texte produit par le système de preuve est écrit dans `cette police (machine à écrire)`.

```
bahia$  /usr/local/coq-7.0beta/bin/coqtop -byte
Welcome to Coq 7.0beta (December 2000)
Coq <  Load Term_const.
[Loading ML file term_const.cmo ...done]

Coq <  Parameter div2:nat->nat;
```

```
       div2_lt : (n:nat)(lt (div2 (S n))(S n)).
div2 is assumed
div2_lt is assumed

Coq <  Recursive Definition log nat->nat lt lt_wf div2_lt
       (x:nat)(log x) = (Cases x of O => O | (S O) => O
                     | (S (S p)) => (S (log (div2 (S (S p))))) end).
log_F is defined

  x : nat
  hrec : (y:nat)
          (lt y x)
          ->v:nat |
               (EX p:nat |
                  (k:nat)
                   (lt p k)
                   ->(def:(nat->nat))(iter nat->nat k log_F def y)=v)
  n : nat
  teq : x=(S n)
  p0 : nat
  teq0 : n=(S p0)
  ============================
   (lt (div2 (S (S p0))) (S (S p0)))

log_terminates is defined
log is defined
log_equation is defined
log_equation : (x:nat) (log x) =(Cases x of
                                       0 => (0)
                                     | (S y0) => Cases y0 of
                                                   0 => (0)
                                                 | (S p0) => (S (log (div2 (S (S p0)))))
                                             end
                                 end)
```

Il faut noter que notre outil affiche le but correspondant à la preuve de décroissance des appels récursifs, même si l'argument *thm* est mal choisi par l'utilisateur. Ainsi, l'outil peut être utilisé pour prévoir l'énoncé exact du théorème à fournir.
L'équation de point fixe peut ensuite être utilisée pour prouver des propriétés générales de la fonction définie, en conjonction avec la récurrence bien fondée sur la relation adaptée. Nous donnons ici un exemple de démonstration:

```
Coq <  Fixpoint power [n,m:nat]:nat := Cases m of O => (1) | (S p) => (mult n (power n p)) end.

Coq <  Axiom le_mult2: (x,y:nat)(le x y) -> (le (mult (2) x) (mult (2) y)).
       Axiom le_S_mult2 : (x:nat) (le (mult (2) (S (div2 x))) (S (S x))).

Coq <  Theorem power_log : (x:nat) (lt O x) -> (le (power (2) (log x)) x).
       Intros x; Elim x using (well_founded_ind nat lt lt_wf).
```

Cette élimination nous génère une hypothèse de récurrence qui est vraie pour tous les y inférieurs à x et le but devient de la forme:

```
   (x0:nat) ((y:nat)(lt y x0)->(lt (0) y)->(le (power (2) (log y)) y))
    ->(lt (0) x0)
    ->(le (power (2) (log x0)) x0)
```

```
Coq <  Clear x; Intros x.
       Rewrite log_equation.
```

Cette réécriture remplace (log x0) par sa définition, donnée à la ML, et on obtient alors le but suivant:

```
((y:nat)(lt y x)->(lt (0) y)->(le (power (2) (log y)) y))
   ->(lt (0) x)
   ->(le (power (2)
         Cases x of  0 => (0)
                  | (S y0) => Cases y0 of  0 => (0)
                                         | (S p0) => (S (log (div2 (S (S p0)))))
                              end
         end) x)
```

Pour résoudre notre but, nous effectuons un traitement par cas sur la variable x. Cela nous donne deux buts dont le premier se vérifie trivialement sous l'hypothèse fausse Hlt: (lt 0 0).

```
Coq <  Case x.
       Intros H Hlt; Inversion Hlt.
```

Le deuxième est de la forme suivante, après un renommage de y0 par n:

```
(n:nat)
    ((y:nat)(lt y (S n))->(lt (0) y)->(le (power (2) (log y)) y))
    ->(lt (0) (S n))
    ->(le (power (2)
          Cases n of  0 => (0)
                   | (S p0) => (S (log (div2 (S (S p0)))))
          end) (S n))
```

De la même manière que pour x, nous effectuons un traitement par cas sur n.

```
Coq <  Intros n; Case n.
```

Pour n=0 nous prouvons que (le (power (2) 0) (1)) est vrai, en appliquant les tactiques Simpl et Auto.

```
Coq <  Simpl; Auto.
```

Pour n différent de 0, qui correspond au cas de récurrence, nous introduisons les hypothèses Hrec et Hlt, puis nous effectuons une expansion de la fonction power sans modifier les autres fonctions.

```
Coq <  Clear n; Intros n Hrec Hlt.
       Lazy Beta Iota Delta [power]; Fold power.
```

Le but devient:

```
(le (mult (2) (power (2) (log (div2 (S (S n)))))) (S (S n)))
Coq <  Apply le_trans with (mult (2) (S (div2 n))).
```

Nous obtenons deux buts à prouver dont le deuxième:(le (mult (2) (S (div2 n))) (S (S n))) , est une conséquence du résultat posé ici comme axiome le_S_mult2.

```
Coq <  2: Apply le_S_mult2.
```

Le premier but est:

```
(le (mult (2) (power (2) (log (div2 (S (S n))))))
  (mult (2) (S (div2 n))))
```

```
Coq <  Apply le_mult2; Simpl.
```

Ainsi le but devient:

```
(le (power (2) (log (S (div2 n)))) (S (div2 n)))
```

On pourra appliquer l'hypothèse
`Hrec:(y:nat) (lt y (S (S n)))->(lt (0) y)->(le (power (2) (log y)) y)`

```
Coq <  Apply Hrec; Auto with arith.
```

`Auto with arith` résout la deuxième hypothèse:`(lt (0) (S (div2 n)))`. Il reste le but suivant provenant de la première hypothèse:`(lt (S (div2 n)) (S (S n)))`. Nous remplaçons `(S (div2 n))` par `(div2 (S (S n)))` avec:

```
Coq <  Replace (S (div2 n)) with (div2 (S (S n))).
```

Le but se transforme en `(lt (div2 (S (S n))) (S (S n)))` résolu par

```
Coq <  Apply div2_lt; Auto with arith.
```

et `Auto with arith` est capable de résoudre le deuxième but qui n'est autre que l'égalité entre `(S (div2 n))` et `(div2 (S (S n)))`.

```
Coq <  Qed.
```

6. Conclusion

Les travaux présentés dans cet article comportent deux parties. D'une part nous décrivons une méthode pour définir des fonctions récursives générales dans le système Coq, d'autre part nous montrons que cette méthode est automatique, en nous concentrant sur une classe restreinte de fonctions récursives: le corps de la fonction récursive doit consister d'abord en une arborescence de constructions de filtrage, et un seul des cas doit présenter une seule récursion (ou bien le même théorème doit s'appliquer pour tous les cas de récursion). Les restrictions prises pour cette première version de notre outil expérimental ne doivent pas faire oublier le caractère général de la méthode. Les auteurs ont déjà eu l'occasion d'utiliser cette méthode pour un algorithme issu de la géométrie algorithmique: l'algorithme du paquet cadeau pour le calcul d'enveloppes convexes en dimension 2 étudié dans [11] et pour la fonction d'Ackermann. D'autres expériences ont été menées avec la même méthode par Nicolas Baudru et David Sanchez, encadrés par Solange Coupet-Grimal sur un problème d'interprétation abstraite.

Les développements futurs pour cette méthode concernent son utilisation pour les fonctions partielles: même si la fonction récursive ne peut pas être définie en Coq parce que l'on ne peut prouver qu'elle termine, il est quand même possible de parler de la valeur que prennent les approximations successives obtenues par itérations de la fonctionnelle, lorsque ces approximations se stabilisent. Ainsi, l'ensemble de définition est l'ensemble des x tels que $(F^k\ g\ x)$ ne dépend plus de g lorsque k grandit. Nous pensons également que la méthode s'appliquera avec élégance pour les fonctions récursives imbriquées, dans lesquelles on voit apparaître des expressions de la forme $f(h(f(arg)))$. La fonction d'Ackermann est un exemple de fonction récursive imbriquée, mais elle reste un cas facile, grâce à la structure particulière de l'ordre lexical qui justifie sa terminaison. Les travaux sur les fonctions imbriquées devront bien sûr être comparés aux travaux effectués par Bove et Capretta [5] et Dubois et Viguié Donzeau-Gouge [6] en théorie des types. Ces deux travaux proposent de faire apparaître la relation liant les arguments successifs des appels récursifs: le sous-ensemble du type de départ où la fonction partielle considérée est bien définie correspondant à la partie bien-fondée (selon Aczel [1]) de cette relation. Dans le travail [5], Ana Bove et Venanzio Capretta ont développé une méthode qui construit un prédicat inductif d'accessibilité pour chaque algorithme. Ce prédicat répond à la question suivante: Ces valeurs d'entrées vérifient-elles les conditions de terminaison de l'algorithme? Le calcul de l'algorithme peut être fait dès que l'on a une preuve que les valeurs d'entrées vérifient le prédicat, alors que la preuve de l'algorithme peut être faite par récursion structurelle sur une preuve que les valeurs d'entrées vérifient le prédicat. Cette méthode dissocie clairement le calcul de la preuve de l'algorithme. Notons que leur méthode s'adapte aux fonctions mutuellements récursives, aux fonctions partielles et aux fonctions récursives imbriquées, pour lesquelles notre technique reste à étudier. D'autres travaux reliés sont ceux de Slind en logique d'ordre supérieur [12], qui fournit un outil plus complet que celui que nous présentons, mais dans un autre cadre puisque les démonstrations n'ont pas à être décrite comme des termes de théorie des types.

Remerciements

Ces travaux bénéficient du soutien financier de Dassault-Aviation et de la communauté européenne dans le cadre du groupe de travail TYPES. Les auteurs tiennent également à remercier les membres du projet Lemme à l'INRIA Sophia Antipolis pour leurs commentaires et suggestions.

Bibliographie

[1] Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, 1977.

[2] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In Harrison and Aagaard [7], pages 1–16.

[3] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Mu noz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997. revised version distributed with Coq.

[4] Richard J. Boulton and Paul B. Jackson, editors. *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*. Springer-Verlag, September 2001.

[5] Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. In Boulton and Jackson [4], pages 121–135.

[6] Catherine Dubois and Véronique Viguié Donzeau-Gouge. A step towards the mechanization of partial functions: domains as inductive predicates. CADE-15, Workshop on Mechanization of Partial Functions, July 1998. `http:/www.cs.bham.ac.uk/~ mmk/cade98-partiality`.

[7] J. Harrison and M. Aagaard, editors. *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[8] Bengt Nordström. Terminating general recursion. *BIT*, 28(3):605–619, Octobre 1988.

[9] Catherine Parent. Synthesizing proofs from programs in the Calculus of Inductive Constructions. In *Mathematics of Program Construction*, volume 947 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1995.

[10] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[11] David Pichardie and Yves Bertot. Formalizing convex hull algorithms. In Boulton and Jackson [4], pages 346–361.

[12] Konrad Slind. Another look at nested recursion. In Harrison and Aagaard [7], pages 497–517.