Un système X
Raisonner formellement sur les programmes ML

S. Baro[1] & P. Manoury[1]

1: P.P.S.
Université Denis Diderot – Paris 7
Case 7014
2 Place Jussieu
75521 PARIS cedex 05
{Sylvain.Baro|Pascal.Manoury}@pps.jussieu.fr

Résumé

Nous proposons dans cet article un système de types dit *renforcé* permettant de garantir la terminaison de programmes exprimés dans le noyau fonctionnel pur de ML. La stratégie d'évaluation est celle de ML. Le langage de types est celui de ML. Le caractère « renforcé » du système proposé vient de l'ajout aux règles de typage traditionnelles d'un principe d'induction structurelle sur les types de données qui apporte la garantie de terminaison.

Nous montrons, pour une version du langage volontairement limitée aux seuls entiers comme structure de données (type `nat`), la correction de notre système de types : tout terme typable (au sens renforcé) de type τ est réductible à une valeur selon la stratégie d'évaluation de ML. Notre preuve de normalisation suit la tradition des preuves par réductibilité au sens de Tait.

Le typage statique confère aux programmes que l'on soumet à cette discipline un certain degré de sûreté d'exécution. Lorsque le langage de programmation est de la famille des langages applicatifs ML, la discipline de type peut être formellement définie, grosso-modo comme une extension du lambda-calcul simplement typé : le langage des types s'enrichit des types paramétrés ; le langage des termes (*i.e.* des programmes) s'enrichit de constantes, d'un opérateur de point fixe et de constructions originales comme le filtrage. Dès lors, la propriété de sûreté d'exécution est elle-même formellement établie comme conservation du type par réduction (voir [16]).

Néanmoins, un tel système de type ne garantit qu'un degré minimal de correction et l'on peut souhaiter vouloir établir des propriétés de programme plus élaborées. Il faut, pour cela, disposer d'un formalisme permettant : d'une part d'exprimer ces propriétés ; d'autre part de les établir formellement. Un certain nombre de systèmes issus des travaux des logiciens portant sur les rapports entre preuves et fonctions calculables ont été élaborés et, pour certains d'entre eux, concrètement mis en œuvre. Ces systèmes se présentent pour la plus grande part comme des systèmes de lambda-calcul typé, avec un langage de types beaucoup plus riche que celui du lambda-calcul simplement typé. Citons [7, 18, 15, 13, 17].

Malheureusement, ce gain d'expression des propriétés des programmes a un prix : le langage des termes, en particulier pour ce qui est de l'écriture des (définitions de) fonctions récursives s'éloigne de la facilité d'expression des langages ML. Ceci est dû, pour l'essentiel, à ce que les systèmes de type plus élaborés proposés réclament que soit garantie la terminaison ou la totalité des fonctions programmées.

C'est, en effet, cet impératif de totalité qui fonde le raisonnement par induction permettant d'établir formellement les propriétés attendues des fonctions – *ie* des programmes (voir [5] ou [4]).

Le but de l'esquisse que nous présentons ici est de définir un système de types qui permette de garantir la terminaison de programmes récursifs exprimés dans un langage à la syntaxe aussi proche que possible de celle du noyau fonctionnel de ML. Pour conserver la facilité syntaxique de ML, notre langage contiendra un opérateur de définitions récursives (à la `let rec`) ainsi qu'une structure de filtrage (à la `match with`). Le langage des types est le langage des types de ML. Le processus d'évaluation des programmes est fixé par une *stratégie de réduction* faible analogue à celle de ML en appel par valeur. Les règles de typage usuelles de ML sont bien entendu valables pour les termes de ce langage. Les nouvelles règles de typage que nous introduisons pour notre *typage renforcé* et qui doivent assurer la terminaison des programmes récursifs ajoutent aux règles traditionnelles permettant de typer le λ-calcul

- une règle exprimant l'invariance des types par expansion[1] ;
- les règles d'introduction pour les (constructeurs des) types de données ;
- une règle d'induction structurelle pour chaque type de données.

Notre système de type renforcé à la puissance du système T de Gödel [11].

Mais contrairement à ce qui se passe dans le système T, nous n'avons aucune règle explicite de typage de l'opérateur de définition récursive ni de l'opérateur de filtrage. Le seul moyen dont nous disposions pour typer les définitions récursives de fonction est d'utiliser la règle d'expansion et les règles d'induction structurelle (voir l'exemple de typage donné en 4). Ainsi, une dérivation de typage, au sens renforcé, est nécessairement une démonstration, par induction structurelle sur un ou plusieurs types de données, que le terme à typer est toujours réductible. C'est d'ailleurs ce que nous établissons formellement en prouvant la normalisation des termes typables au sens renforcé par la méthode de réductibilité de Tait [20].

Notre assignation de type renforcée est notée $t{\downarrow}\tau$ et se prononcera donc « *t est τ-réductible* » ou encore « *t se réduit dans τ* ».

Plan de l'article La suite notre article se décompose en cinq sections :

1. une présentation informelle des possibilités d'expression de notre système en regard du problème de la terminaison et que nous confrontons à d'autres formalismes.
2. la définition du fragment simplifié du langage avec ses règles de réduction dont nous faisons l'étude dans ce papier.
3. un rapide rappel du typage simple pour notre langage.
4. la présentation de notre système de types renforcé.
5. la preuve de correction de notre système, c'est-à-dire, la preuve de normalisation (pour notre stratégie de réduction) des termes typés au sens renforcé.

1. Avant propos

Plusieurs méthodes sont utilisées pour exprimer et montrer la terminaison de fonctions récursives. Dans le système T de Gödel, les fonctions récursives terminent par construction : elles sont définies avec un récurseur dont l'emploi est syntaxiquement régit par les règles de typage. Dans le langage de spécification de Coq, les fonctions récursives sont exprimées en utilisant un combinateur de point fixe gardé garantissant la bonne fondation de la récursion par un critère syntaxique simple (voir [10]). Dans le langage `fœtus` de [1], syntaxiquement proche du notre, la terminaison est assurée par une

[1] En fait, nous appelons cette règle *Eval* car nous l'utilisons au sens d'une tactique : pour prouver qu'un terme est dans un certain type, on le montre pour un de ses réduits.

analyse statique d'un graphe des appels récursifs permettant de contrôler la décroissance structurelle des arguments (une approche similaire, pour un langage du premier ordre, est proposée dans [12]).

À l'exception du système T, ces systèmes décident de la terminaison d'une plus large classe de définitions récursives que celui que nous présentons dans ce papier. De plus, ils le font automatiquement. Il nous paraît cependant utile de tenter une comparaison qui, quoiqu'à notre désavantage, nous permettra de mettre l'accent sur l'originalité de notre approche, les traits non présentés dans ce papier ainsi que les extensions réalisées ou à l'étude.

Un objectif important de notre système est de faciliter la définition de fonctions récursives en rendant celles-ci proches de ce que l'on écrirait en ML. Prenons l'exemple la fonction d'Ackermann dont nous donnons, pour mémoire, une définition en ML

```
let rec ack n m =
  match n, m with
    0, m -> (succ m)
  | n, 0 -> (ack (pred n) 1)
  | n, m -> (ack (pred n) (ack n (pred m)))
```

Dans le système T, on la définit par imbrication de deux récurseurs (notés ici nat_rec), le plus extérieur itérant la fonctionnelle définie par le plus intérieur :

$$
\begin{array}{l}
\lambda n.(nat\_rec\ n \\
\qquad \lambda m.(S\ m) \\
\qquad \lambda p.\lambda a_1.\lambda m.(nat\_rec\ m \\
\qquad\qquad (a_1\ (S\ 0)) \\
\qquad\qquad \lambda q.\lambda a_2.(a_1\ a_2)))))
\end{array}
$$

Dans cette définition, l'analyse de cas des argument (0 ou non) est implicite et les appels récursifs sont cachés dans les variables a_1 et a_2.

Le langage de spécification de Coq explicite l'analyse de cas par l'opérateur de filtrage `Cases` et permet la récursion par l'opérateur de définition par point fixe `Fix`[2]. On y définit la fonction d'Ackermann de la façon suivante :

```
Fixpoint ack [n:nat] : nat -> nat :=
   Cases n of
      O    => [m:nat](S m)
   | (S p) => Fix aux {aux/1 : nat -> nat :=
                      [m:nat] Cases m of
                            O    => (ack p (S O))
                         | (S q) => (ack p (aux q))
                             end }
   end.
```

On retrouve ici la nécessité de définir explicitement la fonction récursive itérée (`Fix aux`) et pour chaque point fixe, la position de l'argument sur lequel est fait la récurrence est précisée, ici, par l'indication `/1`. Cette définition est acceptée par Coq car pour chacun des deux points fixes, on peut vérifier syntaxiquement la décroissance des arguments : on trouve à la position de récurrence indiquée un sous terme, issu du filtrage, de l'argument d'appel.

Dans notre langage, on écrira une définition encore plus proche de ML :

```
let rec ack =
```

[2]La commande de définition `Fixpoint ack [n :nat] ...` est macro-expansée vers `Fix ack {ack/1 : nat -> ...}`

```
  fun n m ->
    case n of
      0 -> S(m)
    | S(p) ->
           case m of
             0 -> (ack p S(0))
           | S(q) -> (ack n (ack S(p) q))
```

La commande de définition `let rec ack ...` est macro-expansée en `rec ack ...` où `rec` est notre combinateur de point fixe. La seule distance, outre l'usage explicite du constructeur `S`, entre notre définition et celle donnée en ML est la survivance de l'imbrication des opérateurs de filtrage `case`. Dans l'état actuel de l'implantation de notre système, la terminaison doit être établie en dérivant interactivement le jugement de typage renforcé. Pour ce, nous avons implanté chacune des règles de typage renforcé et les règles de réduction comme des tactiques. Ces tactiques permettent de dériver, pour cette fonction, une assignation de type (au sens renforcé) de la forme : $\mathsf{ack}{\downarrow}\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$. Notons que le type assigné est celui inféré par le typage à la ML. Nous donnons en 4 l'exemple de la dérivation du type renforcé du récurseur du système T.

Notre système, limité au seul type de base `nat`, est équivalent au système T. Du point de vue intentionnel, on y programme tous les algorithmes dits *primitifs récursifs* de [9]. À ce stade, le gain est essentiellement la facilité d'écriture des définitions récursives : inutilité des indications de type, appels récursifs naturels. Ce défaut d'automatisation est cependant temporaire puisqu'il existe des techniques éprouvées dans [14] que l'on peut adapter à la synthèse automatique de la dérivation des jugements de typage renforcés.

Il est possible d'étendre le système présenté ici aux types algébriques avec paramètres (les types somme de ML). Il suffit pour cela de poser les règles d'introduction pour les constructeurs du type et d'élimination pour le principe de récurrence structurelle[3]. Pour ce qui est des paramètres de type, leur introduction n'ajoute pas de difficulté : nous utilisons le langage des types inférés de ML et le système de type renforcé n'a besoin que d'instancier les paramètres de type des fonctions polymorphes. Si l'on définit donc le type des listes comme

```
type ['a]list = Nil | Cons of 'a * ['a]list
```

dont notre implantation sait extraire les règles attendues, on peut, par exemple écrire la fonction de fusion de deux listes selon une relation `r`

```
let rec merge =
  fun r l1 l2 ->
   case l1 of
     Nil -> l2
   | Cons(c1,l1s) ->
     case l2 of
       Nil -> l1
     | Cons(c2,l2s) ->
              if (r c1 c2)
              then Cons(c1,(merge l1s l2))
              else Cons(c2,(merge l1 l2s))
```

La dérivation de $\mathtt{merge}{\downarrow}[\mathtt{'a}]\mathtt{list} \to [\mathtt{'a}]\mathtt{list} \to [\mathtt{'a}]\mathtt{list}$ est obtenue par deux utilisations de la règle de récurrence structurelle sur les listes.

[3]Naturellement, les principes de récurrence structurelle ne sont engendrés que pour les définitions de type satisfaisant les conditions de positivité d'occurrence du type défini dans le type des arguments des constructeurs.

Ces extensions ne posent théoriquement pas de problème. En effet, nos preuves (et en particulier celle du lemme clef 7) utilisent essentiellement (au sens propre) le fait que l'ensemble des termes réductibles sur un entier (termes de type `nat`) est ou peut être[4] défini inductivement. Une telle définition peut être fournie pour chaque type de données dont nous admettons la définition, et, comme le note T. Coquand dans [8] à propos d'une preuve de normalisation du système T, dans un tel cadre, « il est presque immédiat d'étendre [la preuve] au cas d'autres types de données ».

Nous échappent les définitions naturelles de fonctions faisant usage de décroissances structurelles plus profondes comme la fonction de Fibonacci que nous écririons :

```
let rec fib =
  fun n ->
    case n of
      0 -> S(0)
    | S(p) -> case p of
                0 -> S(0)
              | S(q) -> (fib p) + (fib q)
```

Une première réponse à ce manque est d'ajouter le produit au langage de types avec les règles d'introduction et d'élimination correspondantes. On retrouve alors la possibilité de construire une dérivation faisant usage de la récurrence structurelle simple. Mais une telle extension est loin d'être la plus générale et son usage alourdit les preuves de dérivation.

Les systèmes Coq ou fœtus valident directement de telles définitions en utilisant des conditions syntaxiquement dont la justification est donnée par la méta-théorie du système (validité du prédicat de garde de [10] ou du graphe de décroissance de [1]). Pour gagner en pouvoir d'expression, de telles approchent « externalisent » l'argument décisif de bonne fondation des définitions récursives : le principe d'induction utilisé n'est pas formulé dans le système. Dans [21] et [2] est proposée une utilisation *ad hoc* de types dépendants qui permet de faire revenir l'argument de décroissance dans le giron des preuves de typage. Intuitivement, on indexe les types de données par une indication de taille. Il est ainsi possible d'exprimer dans les règles de typage ces variations de taille. Nous conjecturons qu'une telle approche est adaptable à notre système et, si ce point est encore à l'étude, il n'y a pas de raison que notre technique de preuve par réductibilité ne puisse pas être reconduite dans un cadre où, pour l'essentiel, nous aurons généralisé le principe d'induction structurelle pour les types de données.

2. Langage des termes

Pour répondre à nos exigences de proximité avec ML, notre langage doit, en particulier, offrir la possibilité de construire des expressions récursives (comme avec le `let rec`) ainsi que des expressions par filtrage (opérateur `match`).

Soit donc la grammaire des termes t où x désigne une variable ; $\mathtt{0}$, $\mathtt{S}$, λ, `rec` et `case` (`of`) sont des constantes.

$$t ::= x \mid \mathtt{0} \mid \mathtt{S}t \mid (t\ t) \mid \lambda x.t \mid \mathtt{rec}\ x.t \mid (\mathtt{case}\ t\ \mathtt{of}\ \mathtt{0} \to t \mid \mathtt{S}x \to t)$$

On pourra utiliser l'abréviation $(\mathtt{case}\ t\ \mathtt{of}\ c_0 \mid c_1)$ pour $(\mathtt{case}\ t\ \mathtt{of}\ \mathtt{0} \to t_0 \mid \mathtt{S}x \to t_1)$ ainsi que le raccourci usuel $(t_1\ t_2\ t_3)$ pour $((t_1\ t_2)\ t_3)$.

La notation $\mathtt{rec}\ x.t$ vient intuitivement d'une contraction du codage des définitions récursives en λ-calcul avec un combinateur de point fixe : $(FIX\ \lambda x.t)$ où x est une variable fonctionnelle et FIX un combinateur de point fixe. Nous donnons à cette notation le statut d'une construction primitive

[4]Par facilité, nous n'avons pas explicité cette définition inductive, mais tous nos arguments sur la réductibilité des termes de type `nat` reposent sur l'induction structurelle sur les entiers.

du langage avec sa règle de réduction et notre système de types renforcé permet de s'assurer de la réductibilité des expressions récursives.

Règles de réduction

On se donne une stratégie de réduction faible proche de celle des langages à appel par valeur. La construction `rec` fait, bien entendu exception, ainsi que le `case` que l'on traite comme une structure de contrôle conditionnelle[5].

Valeurs On définit l'ensemble v des valeurs du langage comme le sous-ensemble de termes suivant :

$$v ::= x \mid \lambda x.t \mid \mathtt{0} \mid \mathtt{S}v$$

Radicaux On donne l'ensemble des radicaux du langage et leur règle de réduction notée $\overset{\epsilon}{\hookrightarrow}$. Nous notons $t[u/x]$ pour désigner le terme obtenu par substitution de u aux occurrences libres de x dans t avec sa définition standard (voir par exemple [19]).

$$\begin{array}{ll} \mathit{RRec} & \mathtt{rec}\ x.t \overset{\epsilon}{\hookrightarrow} t[\mathtt{rec}\ x.t/x] \\ \mathit{RBeta} & (\lambda x.t\ u) \overset{\epsilon}{\hookrightarrow} t[u/x] \\ \mathit{RCase0} & (\mathtt{case}\ \mathtt{0}\ \mathtt{of}\ \mathtt{0} \to t_0 \mid \mathtt{S}x \to t_1) \overset{\epsilon}{\hookrightarrow} t_0 \\ \mathit{RCaseS} & (\mathtt{case}\ \mathtt{S}u\ \mathtt{of}\ \mathtt{0} \to t_0 \mid \mathtt{S}x \to t_1) \overset{\epsilon}{\hookrightarrow} t_1[u/x] \end{array}$$

Contextes et relation de réduction On fixe la stratégie de réduction en définissant l'ensemble E des *contextes*. Ici, v désigne une valeur.

$$E ::= \langle\rangle \mid (E\ v) \mid (t\ E) \mid (\mathtt{case}\ E\ \mathtt{of}\ c_0 \mid c_1)$$

On note $E\langle t\rangle$ le terme obtenu en plaçant t dans le contexte E. La relation de réduction d'un pas entre termes quelconques, notée $\hookrightarrow^1$, est définie à l'aide des contextes :

$$\mathit{RCtx} \quad E\langle t\rangle \hookrightarrow^1 E\langle u\rangle \quad \text{si } t \overset{\epsilon}{\hookrightarrow} u$$

La réduction ainsi définie est *déterministe*. C'est-à-dire que pour tout t, si $t \hookrightarrow^1 u$, il existe un seul contexte E tel que $t = E\langle r\rangle \hookrightarrow^1 E\langle r'\rangle = u$.

On se donne alors la relation itérée $\hookrightarrow^n$ où n est un entier :

$$\begin{array}{lll} \mathit{RVal} & v \hookrightarrow^0 v & \text{si } v \text{ est une valeur} \\ \mathit{RIter} & t \hookrightarrow^{n+1} u & \text{si } t \hookrightarrow^1 t' \text{ et } t' \hookrightarrow^n u \end{array}$$

On note $t \hookrightarrow^* u$, ou plus simplement $t \hookrightarrow u$, s'il existe un entier n, possiblement nul, tel que $t \hookrightarrow^n u$.
Le déterminisme de la réduction nous donne immédiatement le petit résultat de confluence :
(i) si $t \hookrightarrow \mathtt{S}^n\mathtt{0}$ et $t \hookrightarrow t'$ alors $t' \hookrightarrow \mathtt{S}^n\mathtt{0}$.
où v désigne une valeur.

On a, toujours pour les termes réductibles à des valeurs du type `nat`, le lemme suivant qui nous servira pour montrer la propriété de stabilité restreinte énoncée par le lemme 6.

Lemme 1 (de confluence suffisante) *Si* $u \hookrightarrow^* \mathtt{S}^k\mathtt{0}$ *et si* $t[\mathtt{S}^k\mathtt{0}/x] \hookrightarrow^* \mathtt{S}^n\mathtt{0}$ *alors* $t[u/x] \hookrightarrow^* \mathtt{S}^n\mathtt{0}$.

Preuve (Idée de) La preuve se fait par induction sur la longueur de $t[\mathtt{S}^k\mathtt{0}/x] \hookrightarrow^* \mathtt{S}^n\mathtt{0}$. Dans le cas récursif, où $t[\mathtt{S}^k\mathtt{0}/x] \hookrightarrow^1 w \hookrightarrow^* \mathtt{S}^n\mathtt{0}$, on utilise la remarque suivante que si $t[\mathtt{S}^k\mathtt{0}/x]$ s'écrit $E\langle r\rangle[\mathtt{S}^k\mathtt{0}/x]$ alors w s'écrit $E\langle r'\rangle[\mathtt{S}^k\mathtt{0}/x]$ avec $r \overset{\epsilon}{\hookrightarrow} r'$.

[5] On se passe du coup d'une construction conditionnelle native que l'on peut définir par un `case` en codant les booléens par des entiers.

3. Système de types simples

Nous rappelons brièvement dans ce paragraphe le système de typage à la ML pour notre langage ainsi que ses principales propriétés.

L'ensemble des types simples τ est donné par :

$$\tau ::= \mathsf{nat} \mid \tau \to \tau$$

On dote notre ensemble de termes d'un système de règles d'assignation de type étendant les types simples du λ-calcul aux nouvelles constructions : `0`, `S`, `rec` et `case` .

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\, TVar$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2}\, TAbs \qquad \frac{\Gamma \vdash t : \tau' \to \tau \qquad \Gamma \vdash u : \tau'}{\Gamma \vdash (t\ u) : \tau}\, TApp$$

$$\frac{\Gamma, x : \tau \vdash t : \tau}{\Gamma \vdash \mathtt{rec}\ x.t : \tau}\, TRec$$

$$\frac{}{\Gamma \vdash \mathsf{0} : \mathsf{nat}}\, TZero \qquad \frac{\Gamma \vdash t : \mathsf{nat}}{\Gamma \vdash \mathsf{S}t : \mathsf{nat}}\, TSucc$$

$$\frac{\Gamma \vdash t : \mathsf{nat} \qquad \Gamma \vdash t_0 : \tau \qquad \Gamma, x : \mathsf{nat} \vdash t_1 : \tau}{\Gamma \vdash (\mathtt{case}\ t\ \mathtt{of}\ \mathtt{0} \to t_0 \mid \mathrm{S}x \to t_1) : \tau}\, TCase$$

Avec les restrictions suivantes : dans les règles $TAbs$, $TRec$ et $TCase$, la variable x n'apparaît pas dans Γ.

Les termes typés dans ce système jouissent de la propriété fondamentale de *conservation du type par réduction* qui s'établit de façon standard en démontrant dans un premier temps la préservation du type par substitution.

Lemme 2 (de préservation du type par substitution) *Si* $\Gamma, x : \tau' \vdash t : \tau$ *et si* $\Gamma \vdash u : \tau'$ *alors* $\Gamma \vdash t[u/x] : \tau$.

Ce lemme se prouve par induction sur la dérivation de $\Gamma, x : \tau' \vdash t : \tau$. Nous n'en donnerons pas le détail (voir par exemple [19]). Les constructions originales du langage de termes doivent se traiter comme le cas de l'abstraction : dans $\mathtt{rec}\ x.t$ et $\mathtt{case}\ t\ \mathtt{of}\ \mathtt{0} \to t_0 \mid \mathrm{S}x \to t_1$ la variable x est liée.

Lemme 3 (de conservation du type par réduction) *Si* $\Gamma \vdash t : \tau$ *et si* $t \hookrightarrow^* t'$ *alors* $\Gamma \vdash t' : \tau$.

Il suffit de montrer par induction sur la dérivation de $\Gamma \vdash t : \tau$ que le type est conservé en un pas ($t \hookrightarrow^1 t'$).

4. Système de types renforcé

On définit une seconde catégorie d'assignation de type que l'on note $t \downarrow \tau$. L'intention de cette assignation est de garantir la *terminaison* de la réduction, selon notre stratégie, du terme t. Mais attention, on peut établir que $t{\downarrow}\tau$ sans que pour autant $t : \tau$ (voir la remarque ci-dessous). Pour que le jeu soit complet, il faut : typer le terme t au sens des types simples ; puis montrer sa terminaison par typage renforcé. Dans le cadre d'un système de preuve de programme, ce double travail n'en est pas un pour l'utilisateur car le typage au sens ML, décidable, sera automatisé.

Système de règles

Les règles s'inspirent de celles du système T de Gödel. On y ajoute une règle intégrant la réduction. Mais, contrairement aux règles du système T, les règles de notre système ne sont pas nécessairement guidées par la syntaxe des termes. Ce qui complique un peu la preuve de correction mais préserve la facilité d'expression du langage.

Un second trait de notre système de typage renforcé et qui l'éloigne d'un système traditionnel de dérivation de type pour le rapprocher d'un système logique de dérivation de séquents, est la présence, dans les contextes manipulés, de déclarations, et il faudrait plutôt dire d'*hypothèses*, de la forme $t{\downarrow}\tau$, où t n'est pas une variable.

Nos règles de typage renforcé sont donc :

$$\frac{}{\Gamma, t{\downarrow}\tau \vdash t{\downarrow}\tau}\,Ax$$

$$\frac{\Gamma, x{\downarrow}\tau' \vdash t{\downarrow}\tau}{\Gamma \vdash \lambda x.t{\downarrow}\tau' \to \tau}\,Fun \qquad \frac{\Gamma \vdash t{\downarrow}\tau' \to \tau \qquad \Gamma \vdash u{\downarrow}\tau'}{\Gamma \vdash (t\ u){\downarrow}\tau}\,App$$

$$\frac{}{\Gamma \vdash \mathsf{0}{\downarrow}\mathsf{nat}}\,Zero \qquad \frac{\Gamma \vdash t{\downarrow}\mathsf{nat}}{\Gamma \vdash \mathsf{S}t{\downarrow}\mathsf{nat}}\,Succ$$

$$\frac{\Gamma \vdash t[\mathsf{0}/x]{\downarrow}\tau \qquad \Gamma, y{\downarrow}\mathsf{nat}, t[y/x]{\downarrow}\tau \vdash t[\mathsf{S}y/x]{\downarrow}\tau}{\Gamma, x{\downarrow}\mathsf{nat} \vdash t{\downarrow}\tau}\,NatRec$$

$$\frac{\Gamma \vdash t'{\downarrow}\tau \qquad t \hookrightarrow t'}{\Gamma \vdash t{\downarrow}\tau}\,Eval$$

On a les restrictions suivantes :
- dans la règle Fun, la variable x n'apparaît pas dans Γ ;
- dans la règle $NatRec$, la variable x n'apparaît pas dans Γ et la variable y n'apparaît ni dans Γ ni dans t.

Exemple

On se donne

$$\Phi = \mathtt{rec}\ \phi.\lambda x.\lambda f.\lambda a.\mathtt{case}\ x\ \mathtt{of}\ \mathtt{0} \to a \mid \mathtt{S}y \to (f\ y\ (\phi\ y\ f\ a))$$

C'est un terme pour représenter le récurseur du système T dont il a le comportement opérationnel :

$$\begin{array}{lcl} (\Phi\ 0\ F\ A) & \hookrightarrow & A \\ (\Phi\ Sn\ F\ A) & \hookrightarrow & (F\ n\ (\Phi\ n\ F\ A)) \end{array}$$

Soit α un type quelconque, on veut s'assurer que pour tout entier n (*i.e.* pour tout $n{\downarrow}\mathsf{nat}$), pour toute fonction totale de α dans α (*i.e.* pour toute $f{\downarrow}\alpha \to \alpha$) et que pour toute valeur dans α (*i.e.* pour tout $a{\downarrow}\alpha$), on a $(\Phi\ n\ f\ a){\downarrow}\alpha$. Pour ce, on dérive l'assignation $\Phi{\downarrow}\mathsf{nat} \to (\alpha \to \alpha) \to \alpha \to \alpha$.

Nous présentons cette dérivation telle qu'on pourrait la saisir à l'aide d'un système d'aide à la preuve dans lequel nos règles sont implantées comme des *tactiques*. Nous utilisons l'indentation pour manifester la structure arborescente de la dérivation. Les trois points $(\ldots)$ indiquent que l'on hérite le contexte de l'étape précédente. On utilise $R*$ pour abréger l'application itérée d'une règle R.

$\vdash \Phi{\downarrow}\mathsf{nat} \to (\alpha \to \alpha) \to \alpha \to \alpha$
$\cdot Eval$
$\vdash \lambda x.\lambda f.\lambda a.(\mathtt{case}\ x\ \mathtt{of}\ 0 \to a \mid \mathtt{S}y \to (f\ y\ (\Phi\ y\ f\ a))){\downarrow}\mathsf{nat} \to (\alpha \to \alpha) \to \alpha \to \alpha$
$\cdot Fun\ *$
$x{\downarrow}\mathsf{nat}, f{\downarrow}\alpha \to \alpha, a{\downarrow}\alpha \vdash (\mathtt{case}\ x\ \mathtt{of}\ 0 \to a \mid \mathtt{S}y \to (f\ y\ (\Phi\ y\ f\ a))){\downarrow}\alpha$
$\cdot NatRec(x)$

$\ldots \vdash (\mathtt{case}\ 0\ \mathtt{of}\ 0 \to a \mid \mathtt{S}y \to (f\ y\ (\Phi\ y\ f\ a))){\downarrow}\alpha$
$\cdot Eval$
$\ldots \vdash a{\downarrow}\alpha$
$\cdot Ax\cdot$

$\ldots, y{\downarrow}\mathsf{nat}, (\mathtt{case}\ y\ \mathtt{of}\ 0 \to a \mid \mathtt{S}z \to (f\ z\ (\Phi\ z\ f\ a))){\downarrow}\alpha$
$\vdash (\mathtt{case}\ \mathtt{S}y\ \mathtt{of}\ 0 \to a \mid \mathtt{S}y \to (f\ y\ (\Phi\ y\ f\ a))){\downarrow}\alpha$
$\cdot Eval$
$\ldots \vdash (f\ y\ (\Phi\ y\ f\ a)){\downarrow}\alpha$
$\cdot App\ *$
$\ldots \vdash f{\downarrow}\alpha \to \alpha$
$\cdot Ax\cdot$
$\ldots \vdash y{\downarrow}\mathsf{nat}$
$\cdot Ax\cdot$
$\ldots \vdash (\Phi\ y\ f\ a){\downarrow}\alpha$
$\cdot Eval\ *$
$\ldots \vdash (\mathtt{case}\ y\ \mathtt{of}\ 0 \to a \mid \mathtt{S}z \to (f\ z\ (\Phi\ z\ f\ a))){\downarrow}\alpha$
$\cdot Ax\cdot$

Remarque : typage renforcé n'est pas simplement typage

Notre système de type renforcé permet de typer des termes qui ne sont pas typable au sens des types ML. En effet, un programme mal typé (au sens des types ML) peut quand même terminer. Cela se produit lorsqu'un sous-terme provoquant une erreur de typage – voire susceptible de boucler – est en fait du *code mort*. Notre système de typage renforcé qui intègre avec la règle *Eval* une possibilité d'*exécution symbolique* des programmes, permet dans certains cas de repérer de telles situations. Nous en donnons un exemple.

Soient les termes A et B tels que $\vdash A : \tau_1$ et $\vdash A{\downarrow}\tau_1$ mais $\vdash B : \tau_2$ avec $\tau_1 \neq \tau_2$, on se donne :

$$\begin{array}{lll}
True & = & \mathtt{S0} \\
False & = & \mathtt{0} \\
If & = & \lambda x.\lambda a.\lambda b.\mathtt{case}\ x\ \mathtt{of}\ 0 \to b \mid \mathtt{S}x \to a \\
Eq & = & \mathtt{rec}\ f.\lambda x.\lambda y. \\
 & & \quad \mathtt{case}\ x\ \mathtt{of} \\
 & & \qquad 0 \to (\mathtt{case}\ y\ \mathtt{of}\ 0 \to True \mid \mathtt{S}y' \to False) \\
 & & \quad \mid \mathtt{S}x' \to (\mathtt{case}\ y\ \mathtt{of}\ 0 \to False \mid \mathtt{S}y' \to (f\ x'\ y')) \\
\Phi & = & \lambda n.(If\ (Eq\ n\ n)\ A\ B)
\end{array}$$

On va pouvoir dériver que $\vdash \Phi{\downarrow}\mathsf{nat} \to \tau_1$ alors que Φ n'est pas typable, au sens ML.

$\vdash \Phi{\downarrow}\mathsf{nat} \to \tau_1$
$\cdot Eval$
$\vdash \lambda n.(If\ (Eq\ n\ n)\ A\ B){\downarrow}\mathsf{nat} \to \tau_1$
$\cdot Fun$

$n{\downarrow}\mathsf{nat} \vdash (\mathit{If}\ (\mathit{Eq}\ n\ n)\ A\ B){\downarrow}\tau_1$
$\cdot NatRec$

$\vdash (\mathit{If}\ (\mathit{Eq}\ \mathtt{0}\ \mathtt{0})\ A\ B){\downarrow}\tau_1$
$\cdot Eval\ *$
$\vdash A{\downarrow}\tau_1$
$\ldots$

$m{\downarrow}\mathsf{nat}, (\mathit{If}\ (\mathit{Eq}\ m\ m)\ A\ B){\downarrow}\tau_1$
$\vdash (\mathit{If}\ (\mathit{Eq}\ \mathtt{S}m\ \mathtt{S}m)\ A\ B){\downarrow}\tau_1$
$\cdot Eval\ *$
$\ldots \vdash (\mathit{If}\ (\mathit{Eq}\ m\ m)\ A\ B){\downarrow}\tau_1$
$\cdot Ax\cdot$

On n'a donc pas en général qu'un terme typable au sens renforcé l'est également au sens usuel. En revanche, lorsque la valeur obtenue par réduction n'est pas fonctionnelle, elle est du type assigné, au sens renforcé comme au sens usuel – nous ne démontrerons pas ce point ici.

5. Correction

Le résultat de correction que nous donnons à présent établit que notre système de typage renforcé satisfait bien son intention : garantir la terminaison des fonctions lorsqu'elles reçoivent en arguments des valeurs du type attendu.

Nous allons donc démontrer que

Théorème 1 *pour tout terme t, si $\vdash t{\downarrow}\tau$ alors t est normalisable.*

où « normalisable » s'entend comme réductible (selon la stratégie fixée) à une forme normale faible.

On utilise pour montrer notre résultat de correction la méthode de réductibilité de Tait [20].

Interprétation

Pour chaque type τ, on définit l'ensemble de termes $\|\tau\|$ par induction sur τ :
– $\|\mathsf{nat}\| = \{t \mid t \hookrightarrow \mathtt{S}^n\mathtt{0}\}$
– $\|\tau' \to \tau\| = \{t \mid \forall u \in \|\tau'\|, (t\ u) \in \|\tau\|\}$
Notons que les éléments des interprétations des types ne sont pas nécessairement des termes clos. Seules les valeurs (formes normales) de $\|nat\|$ sont nécessairement closes.

Fait 1 *si $t \in \|\mathsf{nat}\|$ alors $\mathtt{S}t \in \|\mathsf{nat}\|$.*

Lemme 4 (d'adéquation de l'interprétation) *Si $t \in \|\tau\|$ alors t est normalisable.*

Preuve par induction sur τ.

Si $\tau = \mathsf{nat}$ alors t est normalisable par définition de $\|\mathsf{nat}\|$.

Si $\tau = \tau_1 \to \tau_2$, par définition de $\|\tau_1 \to \tau_2\|$, pour tout $u \in \|\tau_1\|$, $(t\ u) \in \|\tau_2\|$ et, par hypothèse de récurrence, les termes u et $(t\ u)$ sont normalisables. Il existe donc v_1 et v_2 tels que $(t\ u) \hookrightarrow (t\ v_1) \hookrightarrow v_2$. Par définition de la réduction, il faut qu'il existe t', normal, tel que $t \hookrightarrow t'$ et $(t'\ v_1) \hookrightarrow v_2$. Et donc, t est normalisable.

C.Q.F.D.

On montre que les interprétations sont closes par expansion, c'est-à-dire :

Lemme 5 *si* $t \hookrightarrow t'$ *et* $t' \in \|\tau\|$ *alors* $t \in \|\tau\|$

Preuve par induction sur τ.

Si $\tau = \mathsf{nat}$. On suppose $t' \in \|\mathsf{nat}\|$ donc $t' \hookrightarrow \mathsf{S}^n\mathsf{0}$, et $t \hookrightarrow t' \hookrightarrow \mathsf{S}^n\mathsf{0}$. Et donc, par définition de $\|\mathsf{nat}\|$, $t \in \|\mathsf{nat}\|$.

Si $\tau = \tau_1 \to \tau_2$. Supposons $t \hookrightarrow t'$ et $t' \in \|\tau_1 \to \tau_2\|$. On a, par définition de $\|\tau_1 \to \tau_2\|$: pour tout $u \in \|\tau_1\|$, $(t'\ u) \in \|\tau_2\|$. En utilisant le lemme d'adéquation 4, il existe v normal tel que $u \hookrightarrow v$. On montre alors, par induction sur la longueur de la réduction $u \hookrightarrow v$ que $(t\ u) \in \|\tau_2\|$.

- si $u \hookrightarrow^0 v$, c'est-à-dire $u = v$, alors $(t\ u) \hookrightarrow (t'\ u)$ et donc $(t\ u) \in \|\tau_2\|$ par hypothèse d'induction sur les types ;
- si $u \hookrightarrow^1 u' \hookrightarrow v$ alors, par hypothèse d'induction sur la dérivation $u \hookrightarrow v$, on a $(t\ u') \in \|\tau_2\|$ et, comme $(t\ u) \hookrightarrow (t\ u')$, on a par hypothèse d'induction sur les types que $(t\ u) \in \|\tau_2\|$.

C.Q.F.D.

Le lemme technique suivant nous donne une propriété de stabilité pour la substitution restreinte à la réduction des entiers (termes de type nat). On montre donc que :

Lemme 6 *si* $u \hookrightarrow \mathsf{S}^k\mathsf{0}$ *et si* $t[\mathsf{S}^k\mathsf{0}/x] \in \|\tau\|$ *alors* $t[u/x] \in \|\tau\|$.

Preuve par induction sur τ.

Si $\tau = \mathsf{nat}$, alors, par définition de $\|\mathsf{nat}\|$, $t[\mathsf{S}^k\mathsf{0}/x] \hookrightarrow \mathsf{S}^n\mathsf{0}$. Le lemme de confluence suffisante nous donne alors que $t[u/x] \hookrightarrow \mathsf{S}^n\mathsf{0}$, c'est-à-dire : $t[u/x] \in \|\mathsf{nat}\|$.

Si $\tau = \tau_1 \to \tau_2$, supposons que $t[\mathsf{S}^k\mathsf{0}/x] \in \|\tau_1 \to \tau_2\|$, c'est-à-dire, par définition de $\|\tau_1 \to \tau_2\|$, que pour tout $w \in \|\tau_1\|$, $(t[\mathsf{S}^k\mathsf{0}/x]\ w) \in \|\tau_2\|$.

Remarquons l'on peut toujours choisir une variable z telle que $t[\mathsf{S}^k\mathsf{0}/x] = t[z/x][\mathsf{S}^k\mathsf{0}/z]$, $t[u/x] = t[z/x][u/z]$ et $w = w[\mathsf{S}^k\mathsf{0}/z] = w[u/z]$.

On a donc, d'après notre hypothèse que $(t[z/x]\ w)[\mathsf{S}^k\mathsf{0}/z] \in \|\tau_2\|$. D'où, par hypothèse de récurrence, $(t[z/x]\ w)[u/z] \in \|\tau_2\|$; c'est-à-dire $(t[u/x]\ w) \in \|\tau_2\|$, et ce, pour tout $w \in \|\tau_1\|$.

On a donc bien $t[u/x] \in \|\tau_1 \to \tau_2\|$.

C.Q.F.D.

Le dernier lemme dont nous avons besoin, le lemme clef, nous dit que les jugements de typage sont stables par substitution pour l'interprétation des types. Pour le dire intuitivement,

> si $\Gamma \vdash t \downarrow \tau$ est un jugement de typage dérivable et si ρ est une substitution telle que $t_i\rho \in \|\tau_i\|$ pour toute assignation $t_i{\downarrow}\tau_i$ de Γ alors $t\rho \in \|\tau\|$.

On peut toujours écrire un jugement de typage renforcé sous la forme

$$x_1{\downarrow}\tau_1, \ldots, x_n{\downarrow}\tau_n, t_1{\downarrow}\sigma_1, \ldots, t_m{\downarrow}\sigma_m \vdash t{\downarrow}\tau$$

où les x_i sont les variables, toutes différentes, introduites par les règles *Fun* ou *NatRec* et les t_i sont les termes introduits par la règle *NatRec*.

On allégera l'écriture en notant $\bar{\alpha}$ les suites $\alpha_1, \ldots, \alpha_k$ pour α variable, terme ou type. De façon générique, la notation "$\bar{\alpha}, \alpha$" désigne la suite obtenue en ajoutant α à la fin de la suite $\bar{\alpha}$. On peut étendre cette abréviation aux assignations de type, à l'appartenance et aux substitutions. On note ainsi, respectivement, $\bar{x}{\downarrow}\bar{\tau}$, $\bar{v} \in \|\bar{\tau}\|$ ou $t[\bar{v}/\bar{x}]$. Ainsi, un jugement de typage renforcé s'écrit en abrégé : $\bar{x}{\downarrow}\tau, \bar{t}{\downarrow}\bar{\sigma} \vdash t{\downarrow}\tau$; la substitution des $\bar{v}$ aux $\bar{x}$ pour chacun des termes de $\bar{t}$ s'écrit : $\bar{t}[\bar{v}/\bar{x}]$.

Notre lemme clef s'énonce alors

Lemme 7 *Si* $\bar{x}{\downarrow}\bar{\tau}, \bar{t}{\downarrow}\bar{\sigma} \vdash t{\downarrow}\tau$ *alors si* $\bar{v} \in \|\bar{\tau}\|$ *et* $\bar{t}[\bar{v}/\bar{x}] \in \|\bar{\sigma}\|$ *alors* $t[\bar{v}/\bar{x}] \in \|\tau\|$.

Preuve par induction sur la dérivation. On raisonne selon la dernière règle appliquée.

Règle Ax : t est soit une variable dans $\bar{x}$, soit un terme dans $\bar{t}$ et τ est, respectivement, soit l'un des $\bar{\tau}$, soit l'un des $\bar{\sigma}$. On conclut, dans les deux cas, par hypothèse.

Règle Fun : on veut obtenir $(\lambda x.t)[\bar{v}/\bar{x}] \in \|\tau' \to \tau\|$. On a, par hypothèse de récurrence que si $\bar{v} \in \|\bar{\tau}\|$ et $v \in \|\tau'\|$ et $\bar{t}[\bar{v}, v/\bar{x}, x] \in \|\bar{\sigma}\|$ alors $t[\bar{v}, v/\bar{x}, x] \in \|\tau\|$. Or, $t[\bar{v}, v/\bar{x}, x] = t[\bar{v}/\bar{x}][v/x]$ et $(\lambda x.t[\bar{v}/\bar{x}]\ v) \hookrightarrow t[\bar{v}/\bar{x}][v/x]$. En utilisant le lemme 5, on obtient que $(\lambda x.t[\bar{v}/\bar{x}]\ v) \in \|\tau\|$ et ce, pour un v quelconque. Ce qui nous donne $\lambda x.t[\bar{v}/\bar{x}] \in \|\tau' \to \tau\|$. Enfin, comme la restriction sur x dans la règle Fun nous donne que $\lambda x.t[\bar{v}/\bar{x}] = (\lambda x.t)[\bar{v}/\bar{x}]$, on obtient bien $(\lambda x.t)[\bar{v}/\bar{x}] \in \|\tau' \to \tau\|$.

Règle App : on veut $(t\ u)[\bar{v}/\bar{x}] \in \|\tau\|$, c'est-à-dire, $(t[\bar{v}/\bar{x}]\ u[\bar{v}/\bar{x}]) \in \|\tau\|$. Or, on a, par hypothèse de récurrence que $t[\bar{v}/\bar{x}] \in \|\tau' \to \tau\|$ et que $u[\bar{v}/\bar{x}] \in \|\tau'\|$. Donc, par définition de $\|\tau' \to \tau\|$, on obtient $(t[\bar{v}/\bar{x}]\ u[\bar{v}/\bar{x}]) \in \|\tau\|$.

Règle $Eval$: on veut $t[\bar{v}/\bar{x}] \in \|\tau\|$, sachant $t \hookrightarrow t'$. Comme, de plus, par hypothèse de récurrence, on a $t'[\bar{v}/\bar{x}] \in \|\tau\|$. Le lemme 5 nous donne $t[\bar{v}/\bar{x}] \in \|\tau\|$.

Règle $Zero$: on veut $\mathsf{0} \in \|\mathsf{nat}\|$. Ce que l'on a par définition de $\|\mathsf{nat}\|$.

Règle $Succ$: on veut $\mathsf{S}t \in \|\mathsf{nat}\|$. On a, par hypothèse de récurrence que $t \in \|\mathsf{nat}\|$. Le fait 1 nous donne alors que $\mathsf{S}t \in \|\mathsf{nat}\|$.

Règle $NatRec$: on veut $t[\bar{v}, v/\bar{x}, x] \in \|\tau\|$ sachant, en particulier, que $v \in \|nat\|$. On a les deux hypothèses de récurrence suivantes :

(HR0) $t[\mathsf{0}/x][\bar{v}/\bar{x}] \in \|\tau\|$;

(HRS) si $\bar{v} \in \|\bar{\tau}\|$ et $\bar{t}[\bar{v}, v/\bar{x}, y] \in \|\bar{\sigma}\|$ et $t[y/x][\bar{v}, v/\bar{x}, y] \in \|\tau\|$ alors $t[\mathsf{S}y/x][\bar{v}, v/\bar{x}, y] \in \|\tau\|$.

Les restrictions imposées sur les variables x et y dans la règle $NatRec$ permettent de reformuler nos deux hypothèses de récurrence de la façon suivante :

(HR0') $t[\bar{v}/\bar{x}][\mathsf{0}/x] \in \|\tau\|$;

(HRS') si $\bar{v} \in \|\bar{\tau}\|$ et $\bar{t}[\bar{v}/\bar{x}] \in \|\bar{\sigma}\|$ et $t[\bar{v}/\bar{x}][v/, x] \in \|\tau\|$ alors $t[\bar{v}/\bar{x}][\mathsf{S}v/x] \in \|\tau\|$.

Nous allons montrer, que pour tout entier k, si (HR0') et (HRS') alors $t[\bar{v}/\bar{x}][\mathsf{S}^k\mathsf{0}/x] \in \|\tau\|$, par induction sur k.

Si $k = 0$, (HR0') nous donne $t[\bar{v}/\bar{x}][\mathsf{0}/x] \in \|\tau\|$.

Si $k = k' + 1$, on a, par hypothèse de récurrence que $t[\bar{v}/\bar{x}][\mathsf{S}^{k'}\mathsf{0}/x] \in \|\tau\|$ et, par définition, $\mathsf{S}^{k'}\mathsf{0} \in \mathsf{nat}$. D'où, par (HRS'), $t[\bar{v}/\bar{x}][\mathsf{S}^{k'+1}\mathsf{0}/x] \in \|\tau\|$.

Puisque $v \in \|\mathsf{nat}\|$, on a, par définition, que $v \hookrightarrow \mathsf{S}^k\mathsf{0}$. Ce qui, joint au résultat ci-dessus et au lemme 6, nous donne donc que $t[\bar{v}/\bar{x}][v/x] \in \|\tau\|$. Et comme, par les restrictions sur x dans la règle $NatRec$, $t[\bar{v}, v/\bar{x}, x] = t[\bar{v}/\bar{x}][v/x]$, on obtient finalement que $t[\bar{v}, v/\bar{x}, x] \in \|\tau\|$.

C.Q.F.D.

Preuve de correction

Si $\vdash t{\downarrow}\tau$, alors, par le lemme 7, $t \in \|\tau\|$ et, par le lemme d'adéquation de l'interprétation, t est normalisable.

C.Q.F.D.

6. Pour conclure

Nous avons présenté dans cet article un langage de termes formalisant le noyau fonctionnel des langages ML. Nous avons muni ce langage d'une sémantique dynamique sous forme d'un ensemble de règles d'évaluation et d'une double sémantique statique : un typage usuel à la ML et un typage renforcé inspiré du système T. Nous avons ainsi dissocié la propriété de typage usuel des langages de programmations d'avec la propriété de totalité des langages logiques pour les fonctions récursives. Nous avons, pour le système de types renforcé, établi sa propriété de correction : tout terme typable au sens renforcé est réductible.

Le cadre que nous avons posé s'exprime comme une théorie des programmes ML dans le calcul des prédicats du second ordre (voir l'esquisse donnée dans [3]). Les propriétés attendues des programmes ML peuvent donc être posées et dérivées dans un tel système logique. L'implantation d'un système de preuve assistée pour ce formalisme est réalisée par S. Baro dans le cadre de son travail de doctorat pour un langage de termes et de types plus complet que celui volontairement limité que nous avons présenté ici. Dans ce cadre, un bon nombre de preuves de terminaison peuvent être et seront automatisées selon les principes donnés dans [14].

La prochaine étape de ce travail, sera de définir un système de type renforcé analogue pour un principe d'induction plus général utilisant la relation d'ordre bien fondé usuelle sur les entiers. Un tel principe d'induction bien fondée est applicable à d'autres types de données algébriques en utilisant une fonction de mesure adéquate qui « passe aux constructeurs ». Le langage de types utilisera des dépendances restreintes permettant d'exprimer, sur les types de données, des contraintes de décroissance dans une règle d'induction généralisée.

Un effet de bord de cette extension sera de permettre le prise en compte de fonctions partielles définies sur des sous-domaines simplement exprimables (comme les entiers non nul).

Références

[1] A. Abel et T. Altenkirch, *A Predicative Analysis of Structural Recursion*, Journal of Functional Programming 12(1), 1-41.

[2] G. Barthe et *al.*, *Type-based termination of recursive definitions*, à paraître dans Mathematical Structures in Computer Science.

[3] S. Baro, *Une Plate Forme Logique de Spécification et de Preuve de Programmes ML*, prépublication électronique PPS//02/09//n°7 (mn), `http ://www.pps.jussieu.fr/PPS-prepub`.

[4] R. Boyer et J Moore, *A computational logic*, Academic Press, 1979.

[5] R. Burstall, *Proving properties of programs by structural induction*, Computer Journal 12(1) (1969), 41-48.

[6] A. Church, *A formulation of the simple theory of types*, Journal of Symbolic Logic 5(1) (1940), 56-68.

[7] T. Coquand et G. Huet, *The calculus of constructions*, Information and computation 76 (1988), 95-120.

[8] T. COQUAND, *Inductive definitions and Type Theory*, notes de cours pour Types summer school'99 : Theory and Practice of Formal Proofs, `http ://www-sop.inria.fr/certilab/types-sum-school99`.

[9] L. COLSON, *About Primitive Recursive Algorithms*, in Proceedings of the 16th International Colloquim on Automata Languages and Programming, LNCS 372, 194-206.

[10] E. GIMENEZ, *Codifying guarded definitions with recursive schemes*, in Proceedings of the 1994 Workshop on Types for Proofs and Programs, LNCS 996, 39-59.

[11] K. GÖDEL, *Collected works*, Oxford University Press, 1986.

[12] C. LEE, N. JONES ET A. BEN AMRAN, *The Size-Change Principle for Program Termination*, in In ACM Symposium on Principles of Programming Languages, ACM press vol. 28 (2001), 81-92.

[13] J-L. KRIVINE ET M. PARIGOT, *Programming with proofs*, Journal of Information Processing and Cybernetics, 26(3) (1990), 149-167.

[14] P. MANOURY ET M. SIMONOT, *Automatizing termination proof of recursively defined functions*, Theoretical Computer Science, 135 (1995).

[15] P. MARTIN-LÖF, *Intuitionistic Type Theory*, Napoli Bibliopolis, 1984.

[16] R. MILNER, *A theory of type polymorphism in programming*, JCSS 17 (1978), 348-375.

[17] M. PARIGOT, *Recursive programming with proofs*, Theoretical Computer Science, 94 (1992), 335-356.

[18] C. PAULIN-MOHRING, *Inductive definitions in the system Coq - Rules and properties*, in Proceedings of the conference Typed Lambda Calculi and Applications, LNCS 664 (1993).

[19] B. C. PIERCE *Types and Programming Languages*, MIT Press, 2002

[20] W. W. TAIT, *Intentional interpretation of functionals of finite type I*, Journal of Symbolic Logic, 32 (1967).

[21] H. XI ET F. PFENNING, *Dependent types in Practical Programming*, in Proceedings of POPL'99, ACM Press, 214-227.