Une Plate Forme Logique de Spécification et de Preuve de Programmes ML

Sylvain Baro

Une Plate Forme Logique de Spécification et de Preuve de Programmes ML

S. Baro[1]

1: Laboratoire Preuves, Programmes et Système
Université Paris 7
Paris, France
Sylvain.Baro@pps.jussieu.fr

Résumé

Nous verrons ici la présentation d'un système logique dédié à la spécification et la preuve de programmes ML. La spécificité de ce système est de chercher à avoir une approche simple et *orientée programmation* de la preuve. Ce système est fondé sur un calcul des prédicats au premier ordre, enrichi par des constructions ML habituelles. Nous montrerons ensuite que ce système est cohérent en le plongeant dans une théorie des types au deuxième ordre.

1. Introduction

Une des difficultés majeures de l'informatique aujourd'hui est d'avoir la certitude qu'un programme est « correct ». Par correct, on entends que le comportement de ce programme va être conforme à sa spécification. Une des méthodes habituellement utilisée est de tester le programme dans certaines configurations bien précises (en général, aux *limites*), mais cela n'est généralement pas possible.

Une autre possibilité est d'essayer de prouver (au sens mathématique) que le programme correspond effectivement à sa spécification. Pour cela il sera nécessaire d'avoir un modèle (une sémantique) du langage de programmation, ainsi qu'une transcription formelle du langage. Si l'on choisit de faire la preuve formellement, il sera nécessaire d'avoir une *plate-forme* logique cohérente pour pouvoir garantir la correction des preuves.

Il y a principalement deux méthodes pour développer des programmes certifiés : avec la première méthode, appelée *programmation par preuve*, on extrait un programme fonctionnel d'une preuve de la consistance de notre spécification. Cette méthode a pour inconvénient principal que l'algorithme qui sera utilisé dans le programme sera fonction de la preuve, et il pourra être difficile de choisir la preuve qui conduira à un algorithme optimal.

Avec la deuxième méthode, que nous appellerons *preuve de programmes*, on commence par écrire le programme ainsi que sa spécification formelle, puis on prouvera que le programme correspond effectivement à sa spécification. Cela permet d'avoir un contrôle plus aisé sur l'implémentation. Par opposition à la première méthode où l'on donne un sens calculatoire à la preuve, dans la seconde méthode, on donne un sens mathématique au langage de programmation (cf. Boyer et Moore [2], ou Abrial [1]).

Nous choisirons la seconde approche car nous estimons qu'il est important pour l'utilisateur de pouvoir choisir exactement quel sera le code écrit dans un programme. Nous présenterons ici une plate-forme logique de spécification, de développement et de preuves de programmes ML définie dans la continuité des travaux de Krivine et Leivant sur AF2 [6, 7] (où la spécification du programme est donnée sous la forme d'un ensemble d'équations et de son type, et le programme est extrait de la preuve de typage). A la différence d'AF2, nous donnerons le programme sous la forme de fonctions

ML, mais les spécifications seront du même type que celles données dans AF2. Nous voulons que ce support logique soit aussi simple que possible à comprendre et à utiliser pour quelqu'un qui ne sera pas forcément familiarisé avec les notions de preuves formelles, de langage d'ordre supérieur, et c'est la raison qui a guidé certains de nos choix. Cependant nous montrerons ici de quelle manière nous pouvons garantir la correction de notre système, relativement à un calcul des prédicats du deuxième ordre : TTR de Michel Parigot [9]. Cette plate-forme logique est utilisée comme base pour la création d'un logiciel d'aide à la preuve de programme.

Tout d'abord, nous présenterons nos langages de programmation et de spécification, puis nous donnerons la sémantique du langage (statique et dynamique) ; enfin, nous rappellerons brièvement certains aspects de TTR, pour finir par un plongement de notre théorie dans TTR, prouvant ainsi sa correction.

2. Présentation

Nous avons appelé le système logique que nous présentons ici TT1.5, pour Théorie des Types à l'ordre 1, avec « quelque chose de plus ». Ce système est un calcul des prédicats au premier ordre, avec pour langage des termes un mini-ML, et des types récursifs. Notre but est d'encapsuler tout les éléments *au second ordre* sous forme de constructions ML (par exemple les définitions de types). De cette manière, l'utilisateur n'aura à manipuler que des constructions logiques au premier ordre (ce qui reste raisonnable) et du code ML (ce qu'il est supposé connaître). Ce système doit être perçu en fait comme une sorte d'interface utilisateur (*frontend*) pour un langage qui se situe en fait au deuxième ordre. Le but de ces travaux est de développer un Système d'Aide à la Preuve qui soit utilisable par des personnes ayant l'habitude de programmer, mais pas forcément classique minimale de logique.

Le langage de programmation que nous avons choisi est un mini-ML (pour l'instant, un sous ensemble strictement fonctionnel), avec la possibilité de définir des types algébriques, récursifs et polymorphes. Il sera possible de définir des fonctions récursives, avec filtrage, et fonctions d'ordre supérieures (fonctionnelles).

Le langage de spécification sera les formules du calcul des prédicats au premier ordre, avec pour termes du langage, les termes de ML. Les types de données ML seront vus comme des prédicats, et la quantification sera bornée par les types.

Le langage de preuve sera un calcul des prédicats au premier ordre avec l'égalité, l'évaluation des fonctions ML, des règles de typage, et bien sûr d'inductions.

2.1. Langage de programmation

Nous allons maintenant définir notre ML : ce langage permet la définition de types de données algébriques (y compris inductifs et polymorphes), ainsi que la définition de fonctions et fonctionnelles.

Remarque : Tous les ensembles de variables, noms, *etc.* utilisés dans les définitions qui suivent sont dénombrables.

Définition 1 (Termes)
Soit X un ensemble de variable, C un ensemble de constructeurs de type munis d'une arité et F un ensemble de constantes. On défini l'ensemble des termes ML *noté T_{ML} comme le plus petit ensemble tel que :*

- *si* $\mathtt{x} \in X$, *alors* $\mathtt{x} \in T_{ML}$,
- *si* $\mathtt{c} \in C$, $\mathtt{c}$ *est n-aire, et* $\mathtt{t_1}, \ldots, \mathtt{t_n} \in T_{ML}$, *alors* $\mathtt{c}(\mathtt{t_1}, \ldots, \mathtt{t_n}) \in T_{ML}$,
- *si* $\mathtt{f} \in F$, *alors* $\mathtt{f} \in T_{ML}$,
- *si* $\mathtt{t_1}, \mathtt{t_2} \in T_{ML}$, *alors l'application* $(\mathtt{t_1}\ \mathtt{t_2}) \in T_{ML}$,

- *si* $\mathtt{t} \in T_{ML}$, *et* $\mathtt{x} \in X$, *alors l'abstraction* $\mathtt{fun\ x} \rightarrow \mathtt{t} \in T_{ML}$,
- *si* $\mathtt{f} \in X$, *et* $t \in T_{ML}$, *alors* $\mathtt{fix\ f} = \mathtt{t} \in T_{ML}$, *et dénote le plus petit point fixe de* f,
- *si* $\mathtt{x} \in X$, *et* $\mathtt{t_1}, \mathtt{t_2} \in T_{ML}$, $\mathtt{let\ x} = \mathtt{t_1\ in\ t_2} \in T_{ML}$.
- *si* $\mathtt{x} \in X$ *et* $\mathtt{p_1}, \ldots, \mathtt{p_n}$ *sont des* constructeurs linéaires *de* T_{ML}, *et* $\mathtt{t_1}, \ldots, \mathtt{t_n}$ *appartiennent à* T_{ML}, *alors* $\mathtt{match\ x\ with\ p_1} \rightarrow \mathtt{t_1} \mid \cdots \mid \mathtt{p_n} \rightarrow \mathtt{t_n} \in T_{ML}$.

On appelle constructeurs linéaires *le sous-ensemble de* T_{ML} *engendré uniquement avec les deux premières règles, et où pour tout terme, chaque variable a une seule occurrence, auquel on ajoute* _ *qui est un motif de filtage universel.*

Note : Seuls les motifs de filtrage exhaustifs peuvent être définis.

Ces termes sont les termes du langage de programmation, ainsi que les termes du premier ordre du langage logique. Nous montrerons dans la suite que cela ne pose aucun problème.

Définition 2 (Types de données ML)
Soit $\mathcal{V}$ *l'ensemble des variables de types, soit* A *l'ensemble des types de données algébriques munis d'une arité (paramètres de type). On définit l'ensemble des* types de données ML *noté* $\mathcal{T}_{ML}$ *comme étant le plus petit ensemble tel que :*

- *si* $'\mathtt{a} \in \mathcal{V}$, *alors* $'\mathtt{a} \in \mathcal{T}_{ML}$,
- *si* t *est un type de donnée algébrique d'arité* n *de* A, *et* $\mathtt{t_1}, \ldots, \mathtt{t_n} \in \mathcal{T}_{ML}$, *alors* $\mathtt{t}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ *appartiens à* $\mathcal{T}_{ML}$,
- *si* $\mathtt{t}, \mathtt{t}' \in \mathcal{T}_{ML}$, *alors* $\mathtt{t} \rightarrow \mathtt{t}' \in \mathcal{T}_{ML}$,
- *si* $\mathtt{t_1}, \ldots, \mathtt{t_n} \in \mathcal{T}_{ML}$, *alors* $\mathtt{t_1} * \cdots * \mathtt{t_n}$ *appartiens à* $\mathcal{T}_{ML}$.

Pour obtenir le système de type complet, il faut maintenant définir les types de données algébriques :

Définition 3 (Types de données algébriques)
Pour chaque élément t *n-aire de* A, *l'ensemble des types de données algébriques, on associe :*

- *les constructeurs* $\mathtt{c_1}, \ldots, \mathtt{c_p}$ *pris dans l'ensemble* C, *munis d'une signature* $\mathtt{t_1} * \cdots * \mathtt{t_q} \rightarrow \mathtt{t}$, *où* q *est l'arité du constructeur,* $\mathtt{t_1}, \ldots, \mathtt{t_q} \in \mathcal{T}_{ML}$, *tel que chaque constructeur ne soit associé qu'à un seul type,*
- n *paramètres de types* $'\mathtt{a_1}, \ldots, '\mathtt{a_n}$ *tels que* $'\mathtt{a_1}, \ldots, '\mathtt{a_n}$ *sont les seules variables utilisées dans les signatures des constructeurs.*

Ces définitions (mutuellement récursives) offrent un pouvoir d'expression suffisant pour pouvoir écrire :

Exemple 1 (Termes et types)
On peut définir le type des arbres planaires polymorphes comme

```
type 'a ptree =
   Node of 'a * (('a ptree) list) ;;
```

ce qui nous donne une syntaxe simple (et bien connue) pour la définitions des types formels. Il est également possible d'exprimer une fonction telle que le map *sur les listes, qui applique une fonction prise en paramètre sur chaque élément de la liste.*

```
fix map = fun  f -> fun xs  ->
 match xs with
       Nil -> Nil
     | Cons(a,ys) -> Cons((f a),(map f ys)) ;;
```

que l'on peut également écrire de la manière suivante (sucre syntaxique) :

```
let rec map  f =
  fun [] -> []
    | a::ys -> (f a)::(map f ys) ;;
```

ce qui est la définition usuelle dans ML.

Il est maintenant possible de définir le comportement des programmes ML.

2.2. Sémantique

Il est maintenant nécessaire d'exprimer une autre restriction sur les fonctions définies dans notre sous-ensemble de ML : toute les fonctions définies devront être *totales*. D'un autre côté, nous voudrions que le système soit aussi simple que possible, c'est pourquoi nous voudrions avoir l'inférence de type usuelle, de manière à éviter d'avoir à présenter à l'utilisateur des preuves de typage triviales.
Nous allons donc intégrer à la plate-forme un système d'inférence de types à la Milner, mais lorsque la fonction définie sera susceptible de ne pas terminer (par exemple si elle est définie avec l'opérateur de point fixe), l'utilisateur se verra présenté une *obligation* de prouver que la fonction est bien totale sûr le type inféré.

Pour la fonction `map` définie ci-dessus, le système de typage déterminera
`map :('a->'b)->'a list->'b list`, et l'utilisateur aura à prouver :
$\forall \alpha, \beta. \forall (f : \alpha \to \beta), a : \alpha\ list. (map\ f\ a) : \beta\ list$. On verra en 2.3 la définition du langage utilisé.

2.2.1. Sémantique statique

Le système de type est basé sur des règles de typage similaires à la méthode de Milner. Nous écrirons Γ pour le contexte de typage (qui sera en fait une liste d'hypothèses restreintes à des prédicats de typage), τ pour des types et `t` ou `e` pour des termes ML.

On définit d'abord la primitive $Gen(\tau, \Gamma)$ qui généralise (quantifie universellement) les variables de types de τ libres dans le contexte Γ. Nous utilisons aussi la primitive $\phi(\mathtt{t_1}, \mathtt{t_2})$ qui aura pour résultat la liste de substitution engendrée par le filtrage de $\mathtt{t_2}$ par $\mathtt{t_1}$.

$$\frac{\Gamma \vdash t_1 : \tau' \to \tau \quad \Gamma \vdash t_2 : \tau'}{\Gamma \vdash (t_1\ t_2) : \tau}\ App$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (fun\ x \to t) : \tau_1 \to \tau_2}\ Abst$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma, x : Gen(\tau_1, \Gamma) \vdash t_2 : \tau_2}{\Gamma \vdash (let\ x = t_1\ in\ t_2) : \tau_2}\ Let$$

$$\frac{\Gamma, f : \tau \vdash t : \tau}{\Gamma \vdash fix f = t : \tau}\ Fix$$

$$\frac{\Gamma \vdash t : \tau' \quad \Gamma \vdash t_1 : \tau' \quad \cdots \quad \Gamma \vdash t_n : \tau' \quad \overrightarrow{\overrightarrow{\Gamma \vdash u_i : \tau_i^u}} \quad \Gamma, (\overrightarrow{x^1 : \tau_i^u})_1 \vdash e_1 : \tau \quad \cdots \quad \Gamma, (\overrightarrow{x^n : \tau_i^u})_n \vdash e_n : \tau}{\Gamma \vdash match\ t\ with\ t_1 \to e_1 \mid \cdots \mid t_n \to e_n : \tau}\ Match$$

Où $(\overrightarrow{x^i : \tau_i^u})_i$ signifie que pour chaque variable x de t_i, si $x^i \mapsto u_i$ dans $\phi(t_i, t)$, τ_i^u est le type de u_i, et $\overrightarrow{\overrightarrow{\Gamma \vdash u_i : \tau_i^u}}$ signifie que pour tous les sous-termes u_i de t unifiés à des variables de chaque t_i, on aura un type τ_i^u.

$$\frac{\Gamma \vdash t : \forall \alpha. \tau}{\Gamma \vdash t : \tau[\tau'/\alpha]}\ Sub_1$$

Où τ' est un type ML.

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash t : \tau[\tau'/\alpha]}\ Sub_2$$

Où τ' est un type de ML, et α n'apparaît pas dans Γ.

Remarque : Nous n'introduisons pas ici de règles pour typer les constructeurs, parce qu'elles seront introduites ultérieurement lorsque nous verrons la définitions de nouveaux types ML (cf. 2.3.3).

Remarque importante : Il est possible d'inférer avec ces règles un type pour les termes utilisant `fix`, mais quand cet opérateur apparaît, une preuve de terminaison devra être fournie.

2.2.2. Sémantique dynamique

Pour définir l'évaluation dans notre langage, nous utiliserons une sémantique naturelle (dite *big-step*), parce que nous pensons qu'il est plus simple de comprendre comment chaque évaluation sera interprétée. Nous allons utiliser une sémantique pour mini-ML telle que celle définie par Joëlle Despeyroux [3] étendue par le filtrage. Cette sémantique remplace les équations utilisées dans AF2,et nous permet d'utiliser directement les programmes comme terme.

Tout d'abord, il faut définir l'ensemble des *valeurs* de notre sémantique :
$v :=$ $[\texttt{fun x} \to \texttt{e ; } \rho] \mid [\texttt{fun x} \to \texttt{e ; } \rho]_{\texttt{f}} \mid \texttt{C}^{\texttt{k}}(\texttt{v}_1, \ldots, \texttt{v}_{\texttt{k}}) \mid \texttt{x (si x} \in \texttt{V}_{\texttt{v}})$
Où ρ est un contexte d'évaluation, les $\texttt{v}_{\texttt{i}}$ sont des valeurs, les `e` sont des expressions, et les `x` des variables.

V_v est l'ensemble des variables que l'on considérera comme des valeurs, en général, les variables liées dans les hypothèses, mais pas dans le contexte d'évaluation (variables de récurrences, etc.). On définit également les contextes comme étant : $\rho := \epsilon \mid (\texttt{x} \mapsto \texttt{v}) :: \rho$ munis d'un opérateur $\oplus$ concaténant deux contextes recouvrant les variables déjà liées dans le contexte de gauche par celle redéfinie dans le contexte de droite.

On ajoute également deux constructions qui ne sont pas des expressions : la première est `let x = e ; ;` qui liera une valeur (`e` après avoir été évaluée) à un nom, et la seconde `type t = ... ; ;` qui permet de définir un nouveau type algébrique. Nous verrons ces constructions en détail en 2.4.

Les règles d'évaluation de nos termes ML seront donc :

$$\frac{\rho \models e_1 \hookrightarrow v_1 \quad \cdots \quad \rho \models e_k \hookrightarrow v_k}{\rho \models C^k(e_1, \ldots, e_k) \hookrightarrow C^k(v_1, \ldots, v_k)}\ Constructeur$$

$$\frac{}{\rho \models (fun\ x \to e) \hookrightarrow [fun\ x \to e; \rho]}\ fun$$

$$\frac{\rho \models e_1 \hookrightarrow [fun\ x \to e; \rho_1] \quad \rho \models e_2 \hookrightarrow v_2 \quad (\rho_1, x \mapsto v_2) \models e \hookrightarrow v}{\rho \models (e_1\ e_2) \hookrightarrow v}\ App$$

$$\frac{\rho \models e' \hookrightarrow v' \quad (\rho, x \mapsto v') \models e \hookrightarrow v}{\rho \models let\ x = e'\ in\ e \hookrightarrow v}\ let\ in$$

$$\frac{(\rho, x \mapsto [fun\ x \to e; \rho]_f) \models e \hookrightarrow v}{\rho \models fix\ x = e \hookrightarrow v}\ fix$$

Maintenant les règles pour parcourir le contexte d'évaluation :

$$\frac{}{\epsilon \models x \hookrightarrow x}\ var_\epsilon$$

$$\frac{}{(\rho, x \mapsto v) \models x \hookrightarrow v}\ var_1$$

$$\frac{\rho \models x \hookrightarrow v \quad \text{if } x \neq y}{(\rho, y \mapsto v') \models x \hookrightarrow v}\ var_0$$

$$\frac{(\rho, y \mapsto [e; (\rho_1, y \mapsto [fun\ y \to e; \rho_1]_f)]) \models x \hookrightarrow v}{(\rho, y \mapsto [fun\ y \to e; \rho_1]_f) \models x \hookrightarrow v}\ fix\ env$$

En ce qui concerne le filtrage, on définit une fonction $\phi_\rho(\mathtt{x}, \mathtt{y})$, où $\mathtt{x}, \mathtt{y}$ sont des termes, ρ un contexte, et qui signifie : « le terme `x` filtre le terme `y` dans le contexte ρ ». Cette fonction calcule une liste de substitution ρ' présentée sous la même forme qu'un contexte. En cas d'échec, elle s'évaluera en $\bot$. Pour un `match` à n branches, il est nécessaire de créer n règles d'évaluation :

$$\frac{(\text{si } \phi_\rho(v, t_0) = \rho') \quad \rho \oplus \rho' \models e_0 \hookrightarrow v'}{\rho \models\ match\ v\ with\ t_0 \to e_0 \mid \cdots \mid t_n \to e_n \hookrightarrow v'}\ match_0$$

$$\frac{(\text{si } \phi_\rho(v, t_i) = \rho' \text{ et } \forall p.p < i,\ \phi_\rho(v, t_p) = \bot) \quad \rho \oplus \rho' \models e_i \hookrightarrow v'}{\rho \models\ match\ v\ with\ t_0 \to e_0 \mid \cdots \mid t_n \to e_n \hookrightarrow v'}\ match_i$$

Maintenant que nous avons défini le langage et son comportement, nous pouvons passer à la définition du langage de spécification, et de preuve.

2.3. Langage de spécification et de preuve

Nous allons commencer par la définition des formules, puis nous verrons comment sont gérés les types de données dans notre système et nous finirons par les règles d'inférences.

2.3.1. Formules

Nous voulons pouvoir écrire des formules au premier ordre avec la quantification bornée par les types. Nous voudrions aussi pouvoir faire la différence entre les propositions « de typage » et celle correspondant à des propriétés plus spécifiques. Cela permettra de pouvoir utiliser des tactiques plus adaptées à l'un ou à l'autre lors de l'implémentation.

Définition 4 (Formules)
Soient Prop et Type deux types d'ordre supérieur. Nous définissons le type des prédicats, quantificateurs et connecteurs comme suit :

- *Si $\tau \in \mathcal{T}_{ML}$, on défini le prédicat de typage :*
  $(\_ : \_) : \forall\alpha.\alpha \to Type \to Prop.$
- *On définit pour chaque prédicat P son type $P : \tau_1 * \cdots * \tau_n \to Prop$ avec $\tau_1, \ldots, \tau_n$ des types de $\mathcal{T}_{ML}$. On considérera aussi les prédicats 0-aires, qui seront de type Prop.*
- *Pour les connecteurs : $\_ \vee \_ : Prop \to Prop \to Prop$, $\_ \wedge \_ : Prop \to Prop \to Prop$, $\neg\_ : Prop \to Prop$, et $\_ \to \_ : Prop \to Prop \to Prop$.*
- *Pour la quantification bornée :$(\forall x : \_.\_) : \forall\alpha.(\alpha \to Type) \to Prop \to Prop$, et $(\exists x : \_.\_) : \forall\alpha.(\alpha \to Type) \to Prop \to Prop$.*

Remarque : Quand on écrit $\forall\alpha.\tau$ pour un type polymorphe, cela signifie que α ne peut être instancié que par un type $(\forall\alpha : Type.\tau)$.

2.3.2. Règles

Le système de règles utilisé est la déduction naturelle au premier ordre.

- Règle axiome :

$$\frac{}{\Gamma, A \vdash A}\ axiome$$

– Connecteurs :

$$\frac{\Gamma \vdash A \quad \Gamma' \vdash B}{\Gamma, \Gamma' \vdash A \wedge B} \wedge - intro$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge - elim1$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge - elim2$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee - intro1$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee - intro2$$

$$\frac{\Gamma \vdash A \vee B \quad \Gamma', A \vdash C \quad \Gamma'', B \vdash C}{\Gamma \vdash C} \vee - elim$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to - intro$$

$$\frac{\Gamma \to A \quad \Gamma' \vdash A \to B}{\Gamma, \Gamma' \vdash B} \to - elim$$

$$\frac{\Gamma \vdash \neg A}{\Gamma, A \vdash \bot} \neg - intro$$

$$\frac{\Gamma \vdash A \quad \Gamma' \vdash \neg A}{\Gamma, \Gamma' \vdash \bot} \neg - elim$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A} \bot - elim$$

– Quantificateurs bornés :

$$\frac{\Gamma, x : A \vdash B \quad (\text{si } x \notin FV(\Gamma))}{\Gamma \vdash \forall x : A.B} \forall - intro$$

$$\frac{\Gamma \vdash \forall x : A.B \quad \Gamma' \vdash t : A}{\Gamma, \Gamma' \vdash B[t/x]} \forall - elim$$

$$\frac{\Gamma \vdash t : A \quad \Gamma' \vdash B[t/x]}{\Gamma, \Gamma' \vdash \exists x : A.B} \exists - intro$$

$$\frac{\Gamma \vdash \exists x : A.B \quad \Gamma', x : A, B \vdash C \quad (\text{si } x \notin FV(\Gamma, \Gamma', C))}{\Gamma, \Gamma' \vdash C} \exists - elim$$

– Évaluation :

$$\frac{\Gamma \vdash A(t) \quad (\text{si } \rho \models t \hookrightarrow t')}{\Gamma \vdash A(t')} \; eval\ gauche$$

$$\frac{\Gamma, A(t) \vdash B \quad (\text{si } \rho \models t \hookrightarrow t')}{\Gamma, A(t') \vdash B} \; eval\ droite$$

Nous devons ajouter à ces règles les axiomes de l'égalité. On défini le prédicat au premier ordre $\_ = \_$ de type $\_ = \_ : \forall\alpha.\alpha \to \alpha \to Prop$, avec les schémas d'axiomes suivants :

Réflexivité : pour tout type τ

$$\forall x : \tau.x = x$$

Extensionalité sur les propositions : pour tout type τ, pour tout $P : \tau \to Prop$

$$\forall x : \tau.\forall y : \tau.((x = y) \to (P[x/z] \iff P[y/z]))$$

Extensionalité sur les fonctions : pour tout type τ, τ'

$$\forall f : (\tau \to \tau').\forall g : (\tau \to \tau').(\forall x : \tau.(f\ x) = (g\ x)) \to f = g$$

Avec $A \iff B$ signifiant $A \to B \wedge B \to A$. L'évaluation est prise en charge par les règles *eval*.

Il ne reste plus qu'à ajouter à notre système les règles d'induction, la possibilité de typer les constructeurs, et la possibilité de définir de nouveaux types algébriques.

2.3.3. Types de données algébriques

Chaque définition d'un nouveau type de données algébrique va engendrer (si la définition est acceptable) un certain nombre de règles permettant de manipuler les objets de ce type.

Avant d'introduire un nouveau type de données, il est nécessaire de s'assurer que ce type est bien défini : la première chose à vérifier est que le type est bien habité par au moins un élément (que le type a au moins un constructeur non récursif), ensuite il faut vérifier les critères de bonne définition du point fixe, c'est à dire les conditions de positivités dans le cas d'un type récursif.

Pour chaque type de donnée algébrique, nous allons avoir des paramètres de type, et un ensemble de constructeurs munis de leurs signatures (cf. définitions 2 et 3).

La définition d'un nouveau type de donnée τ va engendrer pour chaque constructeur $\mathtt{c}$ de signature $\mathtt{c} : \tau_1 * \cdots * \tau_n \to \tau$ de l'ensemble des constructeurs de τ (C_τ) la règle suivante :

$$\frac{\Gamma_1 \vdash t_1 : \tau_1 \quad \cdots \quad \Gamma_n \vdash t_n : \tau_n}{\Gamma_1, \ldots, \Gamma_n \vdash c(t_1, \ldots, t_n) : \tau}\ c - intro$$

ainsi qu'une règle de récurrence sur le type nouvellement défini :

$$\frac{\overrightarrow{\Gamma_i \vdash \forall t_j : \tau_j.\forall u_k : \tau.P(u_k).P(c_i(t_1, \ldots, t_n, u_1, \ldots, u_m))}}{\Gamma, x : \tau \vdash P(x)}\ rec_\tau$$

Nous justifierons dans la section 4 la raison pour laquelle ces extensions sont conservatives.

2.4. Évènements, interactions

Pour modéliser la notion de « top level », nous introduisons ce que Boyer et Moore [5] appellent des *évènements*. Les évènements vont nous permettre de formaliser le fait que dans le déroulement d'une session de spécification/programmation/preuve, nous allons définir de nouveaux types de données, poser de nouveaux axiomes et de nouvelles définitions, écrire des fonctions, annoncer des théorèmes et les prouver. Le but ici n'étant pas de définir la syntaxe du « langage », on se contentera de voir deux formes en détail : les définitions de types et de fonctions.

Pour les définitions de type, on introduit la forme :

```
type <params> <nom> =
     <Constr0>
   | <Constr1>
   | <Constr2> of <type2>
   | <Constr3> of <type3> ;;
```

où l'on définit le type algébrique <*nom*>, les constructeurs 0-aires <*Constr0*> et <*Constr1*> de type <*nom*>, et deux constructeurs de type : <*type2*> -> <*nom*>, et <*type3*> -> <*nom*>, où <*nom*> est susceptible d'avoir des occurrences dans <*type2*> et <*type3*>.

A partir de cette définition, on obtient toutes les informations nécessaires pour engendrer le type de données, et les règles d'introduction et d'élimination qui lui sont associées. Il est cependant nécessaire de vérifier que les conditions de positivité sont respectées.

On acceptera les définitions de types inductifs si les conditions de positivités sont respectées pour tous les constructeurs : pour un type τ ayant un constructeur défini comme `c of` τ', on aura τ' de la forme $\tau_1 * \cdots * \tau_n$. Si les τ_i ne sont pas fonctionnels, la définition est correcte, et s'ils sont fonctionnels, toutes les occurrences de τ dans τ dans τ_i doivent être positives.

Définition 5 (Positivité)
Si τ est positif (resp. négatif), et $\tau \equiv \tau' \to \tau''$ alors τ'' est positif (resp. négatif) et τ' est négatif (resp. positif).
Si `c of` $\tau_1 * \cdots * \tau_n$, *alors $\tau_1, \ldots, \tau_n$ sont positifs.*

On ajoute également la constructions suivante :

```
let <nom> = <expr> ;;
```

qui a pour effet d'inférer un type pour l'expression <*expr*>, puis, si l'évaluation de celle-ci est susceptible de ne pas terminer (voir plus haut), une obligation de preuve va être engendrée. Le système ne va donc accepter les définitions de fonctions que si elles sont bien typées. Si l'expression est acceptée, <*expr*> sera liée au nom <*nom*>.

Nous avons donc maintenant la plate-forme logique complète, dans laquelle les aspects d'ordre supérieur sont tous encapsulés dans des constructions ML habituelles (polymorphisme, définition de types, etc.). Nous avons choisi de définir ce système d'une façon naïve pour permettre à un utilisateur de manipuler des concepts simples et proche de la programmation habituelle. D'un autre côté, nous voudrions pouvoir garantir la correction de ce système relativement à une théorie formalisée et considérée comme correcte. Nous montrerons dans la section 4 comment nous parvenons à cette fin en plongeant TT1.5 dans la théorie TTR de Michel Parigot, mais avant cela, nous donnerons un bref récapitulatif de cette théorie.

3. Présentation de TTR

TTR est un calcul des prédicats au deuxième ordre, étendu avec un opérateur de plus petit point fixe sur les prédicats. Cette théorie est décrite dans [9]. Nous oublierons la distinction qui est faite dans TTR entre les preuves ayant un contenu calculatoire et celle n'en ayant pas, dans la mesure où nous ne nous occupons pas ici d'extraction de code, mais de preuve de programmes.

Les termes d'individus sont construits de façon classique à partir d'un ensemble de variables et de constantes fonctionnelles, auxquelles on ajoute les constantes **Ap**, **K** et **S** et les équations usuelles sur ces combinateurs. On notera $(t_1\ t_2)$ pour $\mathbf{Ap}(t_1, t_2)$. On utilisera également un ensemble de variables de prédicats munis d'une arité.

On définira les formules du second ordre comme suit :
– si P est un symbole de prédicat n-aire, et $t_1, \ldots, t_n$ sont des termes, alors $P(t_1, \ldots, t_n)$ est une formule,

- si A et B sont des formules, $A \to B$, $\neg A$, $A \vee B$ et $A \wedge B$ sont des formules.
- if x est une variable d'individu et A une formule, $\forall x.A$ est une formule.
- si X est une variable de prédicat et A une formule, $\forall X.A$ est une formule,
- si A est une formule, X^n une variable de prédicat n-aire n'ayant que des occurrences positives dans A, $x_1, \ldots, x_n$ sont des variables d'individus et $t_1, \ldots, t_n$ des termes, alors $\mu X^n x_1 \ldots x_n A < t_1, \ldots, t_n >$ est une formule,
- si e est une équation, alors c'est une formule.

Intuitivement, $\mu X^n x_1 \ldots x_n A < t_1, \ldots, t_n >$ dénote le plus petit prédicat X^n tel que $X^n(x_1, \ldots, x_n)$ est équivalent à A pour tout $x_1, \ldots, x_n$. Les $t_1, \ldots, t_n$ sont les paramètres.

Les règles de TTR sont les règles habituelles de la déduction naturelle au deuxième ordre, étendues avec les règles d'introduction et d'élimination du μ. Bien que nous ne nous occupons pas ici d'extraction de programme, nous allons avoir besoin du fait que TTR permet l'extraction de λ-terme (réalisabilité).

Les règles ajoutées pour l'opérateur de point fixe sont :

$$\frac{}{\Gamma \vdash A[\lambda y.\mu X x A < y > /X] \to \mu X x A < x >} \ \mu_i$$

$$\frac{}{\Gamma \vdash \mu X x A < x > \to A[\lambda y.\mu X x A < y > /X]} \ \mu'_i$$

$$\frac{\Gamma \vdash A[\lambda y.B[y]/X] \to B[X]}{\Gamma \vdash \mu X x A < x > \to A[\lambda y.\mu X x A < y > /X]} \ \mu'_i$$

On aura également les axiomes de l'égalité au deuxième ordre.

3.1. Types de données formels

En utilisant l'opérateur μ, il est possible de définir aisément les types de données formels. Nous ne donnerons pas ici tous les détails, nous prendrons juste comme exemple le type des listes polymorphes. On définit le prédicat $List(T, l)$ (où T est un prédicat) tel que :

$$List(T, l) \equiv \mu L(T, l)\Phi < T, l >$$

avec
$\Phi \equiv \forall X.(X(T, Nil) \to (\forall a.T(a) \to \forall x.L(T, x) \to X(T, Cons(a, x))) \to X(T, l))$

En prouvant les théorèmes suivants :
$List(T, Nil)$ et $\forall a.T(a) \to \forall l.List(T, l) \to List(T, Cons(a, l))$ on extrait les représentations sous forme de λ-termes de Nil et $Cons$. On peut ensuite engendrer les λ-termes qui représentent l'opérateur $case_{List}$ (qui nous sera nécessaire ultérieurement) en prouvant le théorème :
$\forall X.(X(Nil) \to (\forall a.T(a) \to \forall x.List(T, x) \to X(Cons(a, x))) \to \forall l.List(T, l) \to X(l))$.

On se servira des ces représentations pour justifier la sémantique naturelle.

Remarque : Lors de l'extraction d'un λ-terme à partir d'une preuve faisant appel à la règle μ, on obtiendra un terme avec un point fixe (on peut utiliser par exemple celui de Church). Nous n'utiliserons donc pas de termes de type rec_{List}.

Nous pouvons maintenant passer à une description rapide des propriétés de TTR qui nous intéressent (les preuves sont dans [9]).

3.2. Propriétés

Le modèle que nous utilisons pour TTR ($\mathcal{M}$) est Λ^*, l'ensemble des λ-termes modulo $\beta\eta$-équivalence. On a l'interprétation habituelle des combinateurs dans le λ-calcul, et nous l'étendons avec

l'interprétation des constructeurs (à chaque ajout d'un nouveau type formel), et avec l'interprétation des constantes.

On écrit t^* l'interprétation d'un terme de TTR t dans $\mathcal{M}$.

On a les propriétés suivantes :

Proposition 1 (Correction)
Soit $A_1, \ldots, A_n, B$ un type de donnée formel. Si t réalise $\forall x_1, \ldots, x_n.(A_1(x_1) \to A_n(x_n) \to B(f\ x_1 \ldots x_)$, alors, pour tout $u_1, \ldots, u_n$ satisfaisant $A_1, \ldots, A_n$, on a $(t\ u_1 \ldots u_n) = (f\ u_1 \ldots u_n)$.

Proposition 2 (Conservation)
Soit E un ensemble d'équations satisfaites dans $\mathcal{M}$, et A une proposition du second ordre : si $E \vdash A(t)$, alors t^* réalise A.

Proposition 3 (Types de données formels)
Les types de données définis comme expliqué ci-dessus sont des types de données formels. Informellement, pour τ un type de données, on a :
- *$\tau(t)$ implique que t^* réalise $\tau(t)$,*
- *t' réalise $\tau(t)$ implique $\tau(t)$ et $t = t'$,*
- *on peut calculer les destructeurs avec une réduction en temps constant.*

Remarque : On pourra étendre TTR en utilisant les règles de la déduction libre classique (et le $\lambda\mu$-calcul) pour pouvoir traiter certains aspects impératifs de ML.

4. Plongement

Dans cette section, nous allons présenter comment plonger notre théorie (TT1.5) dans TTR. Cela nous donne une méthode pour prouver la correction de notre système, et cela nous permet de prouver que les programmes ML sont bien des termes du langage. L'usage des programmes ML comme termes de la logique est donc bien justifié.

Nous allons commencer par voir comment les types de données algébriques de ML peuvent être plongés dans TTR, et les restrictions que cela va engendrer.

4.1. Types

On divisera le plongement des types ML en plusieurs parties : en effet, les types algébriques et les types fonctionnels ne pourront pas être traités de la même manière.

4.1.1. Le prédicat $Type(\_)$

Dans TTR, il est possible d'utiliser un prédicat quelconque de la même manière qu'un prédicat de type, mais ce n'est pas ce que nous voulons dans TT1.5. Par exemple, lorsque l'on définit le type des listes polymorphes, on ne souhaite pas que le paramètre de type soit instancié avec un prédicat quelconque.

On définit le prédicat du premier ordre $type(\_)$ (en minuscule), comme étant le plus petit prédicat tel que :

1. si $\tau(x)$ et τ est un type de données formel défini, alors $type(x)$,
2. si $\forall x.\tau_1(x) \to \exists y.\tau_2(y) \wedge y = (f\ x)$, où τ_1 et τ_2 sont tels que $\forall a.\tau_1(a) \to type(a)$ et $\forall a.\tau_2(a) \to type(a)$, alors $type(f)$.

Intuitivement, cela signifie qu'un élément d'un type de données vérifie le prédicat $type(\_)$, et que ce que l'on pourrait appeler « un élément d'un type flèche de ML » vérifie également $type(\_)$.

On définit maintenant quelques notations et abréviations :
- $x : \tau$ pour $\tau(x)$, $\forall x : \tau.P$ pour $\forall x.\tau(x) \rightarrow P$, et $\exists x : \tau.P$ pour $\exists x.\tau(x) \wedge P$,
- $f : \tau_1 \rightarrow \tau_2$ pour $\forall x : \tau_1.\exists y : \tau_2.y = (f\ x)$,
- $Type(\tau)$ pour $\forall x : \tau.type(x)$, et $Type(\tau_1 \rightarrow \tau_2)$ pour $\forall x : \tau_1 \rightarrow \tau_2.type(x)$.

Ces définitions et notations ne sont pas des extensions de TTR, mais en forment une théorie de TTR : si on considère que tous les types de données formels ont été définis avant toute chose, on peut définir le prédicat *type* comme étant :

$$type(x) \equiv \mu T x \Psi < x >$$

avec
$\Psi \equiv \forall X.(\overrightarrow{(\forall x.\tau_i(y) \rightarrow X(y))} \rightarrow$
$(\forall f.(\forall \tau_1.(\forall a.\tau_1(a) \rightarrow T(a)) \rightarrow (\forall \tau_2.(\forall b.\tau_2(b) \rightarrow T(b)) \rightarrow (\forall y.\tau_1(y) \rightarrow$
$\exists z.\tau_2(z) \wedge .z = (f\ y))) \rightarrow X(f))) \rightarrow X(x))$

Lorsque l'on écrit $\overrightarrow{(\forall x.\tau_i(y) \rightarrow X(y))}$, les τ_i seront tous les types de données formels que l'on a défini.

On traduira donc les types polymorphes, par exemple $\forall$ `'a.t['a]` en
$\forall \alpha : Type.\tau[\alpha]$.

Remarque : Si on choisit d'accepter de pouvoir « encapsuler » une fonction dans un type algébrique (comme on le fait ici), on ne peut plus garantir l'existence d'un représentant canonique pour chaque objet d'un type.

4.1.2. Interprétation des types ML

On a vu en 2.3.3 comment définir les types algébriques ML. Par exemple le type `'a list`.

```
type 'a list =
     Nil
   | Cons of 'a * 'a list ;;
```

On va transformer chaque type avec n paramètres de types en un prédicat $n+1$-aire. Pour le type des listes polymorphes, on obtiendra le prédicat $List(A, l)$ avec A un type de données.

Un type algébrique ML aura la forme générale :

```
type 'a1,...,'an t =
     C0 of t01 *...* t0q
   ...
   | Cp of tp1 *...* tpr ;;
```

On transformera ceci en un type de données formel τ tel que
$\tau(\alpha_1, \ldots, \alpha_n, x) \equiv \mu T(\alpha_1, \ldots, \alpha_n, x)\Phi < \alpha_1, \ldots, \alpha_n, x >$ où

$\Phi \equiv \forall X.((\forall x_1 : \tau_1^0, \ldots, x_q : \tau_q^0.X(\alpha_1, \ldots, \alpha_n, C_0(x_1, \ldots, x_q))) \rightarrow$
$\cdots \rightarrow (\forall x_1 : \tau_1^p, \ldots, x_r : \tau_r^p.X(\alpha_1, \ldots, \alpha_n, C_p(x_1, \ldots, x_r))) \rightarrow X(\alpha_1, \ldots, \alpha_n, x))$

dans lequel τ_j^i est la traduction de `tij` où toutes les occurrences de `t` on était traduite par T.

Proposition 4 (Bonne fondation des types récursifs)
Si les types ML ont été définis en respectant les conditions de positivités et tels qu'ils soient non vides (cf. définition 5), alors leurs traductions dans TTR respectera aussi les conditions de positivités et seront non vide.

A partir de cette traduction, on est capable d'extraire la représentation des constructeurs en TTR, ainsi que l'opérateur d'analyse par cas sur le type ($case_\tau$).

Pour les types fonctionnels ML, on utilisera la forme décrite plus haut : la proposition `f : t1 → t2` ML sera traduite en TTR sous la forme
$\forall x : \tau_1.\exists y : \tau_2.y = (f\ x)$.

4.2. Termes

La transformation des termes de TT1.5 en TTR ne présente aucune surprise : il suffit de transformer les termes de ML dans le λ-calcul étendu avec un opérateur de point fixe. Le λ-calcul pouvant lui même être codé sous la forme de combinateurs, on aura ainsi la preuve que les programmes ML (munis de leur mécanisme d'évaluation) peuvent vraiment être considérés comme des termes du premier ordre.

On note Λ l'ensemble des termes de TTR. On traduira les termes de TT1.5 comme suit :
- si `x` est une variable de T_{ML}, c'est une variable dans Λ,
- si `f` est un symbole de constante de T_{ML}, c'est un symbole de constante de Λ,
- si `c(t1,...,tn)` et `c` est un constructeur n-aire, `t1,...,tn` sont des termes de T_{ML}, on obtiens (cf. 4.1.2) : $(c\ t_1, \ldots, t_n)$ avec $t_1, \ldots, t_n$ la traduction de `t1,...,tn`,
- `(t1 t2)` si `t1` et `t2` sont des termes de T_{ML}, sera transformé en $(t_1\ t_2)$, où t_1 et t_2 sont les traductions de `t1` et `t2`,
- `fun x -> t` où `x` est une variable et `t` un terme de T_{ML} va être traduit en $\lambda x.t$, dans lequel x et t sont les traductions de `x` et `t`,
- `let x = t1 in t2` où `x` est une variable et `t1`, `t2` sont des termes de T_{ML} sera traduit en $(\lambda x.t2\ t_1)$, dans lequel x, t_1 et t_2 sont les traductions de `x`, `t1` et `t2` (on pourra raffiner en utilisant un opérateur de mise en mémoire),
- `fix x = t` sera traduit en $!\lambda x.t$, où ! est un opérateur de plus petit point fixe dans Λ,
- `match t with t1 -> e1 | ...| tn -> en` sera transformé [1] en `case t of t1' -> e1' | ...| tm' -> em'` Ensuite, si le type de t (la traduction de `t`) est τ, ce sera traduit par $(case_\tau\ t\ e'_1\ \ldots\ e'_m)$ tel que si le motif de filtrage p est `t[x1,...,xn] -> e[x1,...,xn]` on obtiendra $e_p : \lambda x_1 \ldots \lambda x_n.e[x_1, \ldots, x_n]$ avec e la traduction de `e`.

4.3. Formules

Les formules sont très simple à traduire : c'est un plongement d'un calcul du premier ordre au second ordre.

Les prédicats de types algébriques ML, seront traduits par les prédicats des types formels de TTR, et les prédicats de typage pour les types fonctionnels (`f :t -> t'`) seront traduits par une formule de la forme $\forall x : \tau.\exists y : \tau'.y = (f\ x)$.

On considérera la quantification bornée comme une notation (cf. 4.1.1) : $\forall x : P.Q$ signifie $\forall x.P(x) \rightarrow Q$ et $\exists x : P.Q$ signifie $\exists x.P(x) \wedge Q$.

Du point de vue des règles, il n'y a rien à faire.

4.4. TT1.5 $\subset$ TTR

Toutes les fonctions utilisées dans TT1.5 sont totales. Leur type est inféré, et dans le cas de fonctions potentiellement partielles (*ie* récursives), il est nécessaire d'en prouver la terminaison. De plus, les filtrages utilisés devront être exhaustifs.

[1] Tout filtrage *en profondeur* exhaustif peut être transformé en un arbre de plusieurs filtrage *en surface*, c'est à dire des opérateurs *case*, en conservant la même sémantique.

Nous allons montrer que TT1.5 est réellement un sous-ensemble clos de TTR, ou plus exactement, une représentation d'une partie de TTR.

Nous avons déjà montré que les termes ML peuvent être considérés comme des λ-termes et que ceux-ci peuvent être transcrits en combinateurs. Nos termes ML sont donc des termes de TTR. Il est cependant nécessaire de prouver que notre système a bien le comportement attendu, c'est à dire que notre sémantique naturelle est un sous-ensemble de la β-réduction. Nous prouverons ensuite que les règles de notre système de typage sont correctes par rapport au typage de TTR. On déduira de ces propriétés que TT1.5 est bien un sous-ensemble clos de TTR.

A cause de l'opérateur `fix` nous ne pourront avoir la normalisation forte de notre système, mais nous aurons la normalisation faible.

Les lemmes suivants sont nécessaires pour pouvoir manipuler les contextes.

Lemme 1 (Évaluation de `fix`)
$\rho \models fix\ x = e \hookrightarrow v$ si et seulement si $\rho \models e[(fix\ x = e)/x] \hookrightarrow v$

Lemme 2
Les expressions de TT1.5 s'évaluent vers les même valeurs si les substitutions du contextes sont appliquées immédiatement, et si on identifie les fermetures avec les abstractions, et les $[\texttt{fun x} \rightarrow \texttt{e}; \epsilon]_f$ avec `fix x = e`.

La preuve est triviale en utilisant le lemme 1 dans le cas du point fixe.

Proposition 5 (Sémantique naturelle $\subset$ β-réduction)
Pour tout `t` un terme de TT1.5, $\texttt{t} \hookrightarrow \texttt{t}'$ implique $t \triangleright_\beta t'$ avec t, t' respectivement les interprétations de `t` et `t'` dans TTR.

PREUVE: *Par induction structurelle sur le terme `t` et en appliquant toutes les substitutions du contexte d'évaluation à chaque étape de réduction (modulo α-conversion) :*

`t` est une variable ou un nom : *S'il est dans le contexte, s'évalue en sa valeur, sinon, rien.*

`t` est un constructeur : $\texttt{C}(\texttt{t}_0, \ldots, \texttt{t}_n)$ *s'évalue en* $\texttt{C}(\texttt{t}'_0, \ldots, \texttt{t}'_n)$ *si* $\texttt{t}_0 \hookrightarrow \texttt{t}'_0, \ldots, \texttt{t}_n \hookrightarrow \texttt{t}'_n$. *Puis avec les hypothèse d'induction sur les arguments de `t`.*

`t` est une abstraction : *On identifie la fermeture avec l'abstraction (voir lemme 2) où la substitution a été appliquée. Dans TT1.5, il n'y a pas d'évaluation sous le λ.*

`t` est une application : $\texttt{t} \equiv (\texttt{t}_1\ \texttt{t}_2)$. *Par les hypothèses d'induction.*

`t` est un point fixe : *L'environnement est « déroulé » de la même manière que les substitutions sont appliquées. On conclue par le lemme 2, puis par les hypothèses d'induction.*

`t` est `let ...in` : *Identique à une application et une abstraction.*

`t` est un `match` : *Par réduction vers l'opérateur $case_\tau$, puis par les hypothèses d'induction.* □

Proposition 6 (Correction de l'induction)
Les règles d'inductions de TT1.5 sont dérivables à partir des types de données formels de TTR.

PREUVE: *On fera la preuve sur l'exemple des listes polymorphes.*

```
type 'a list =
     Nil
   | Cons of 'a * 'a list ;;
```

Le type de donnée formel sera : $List(\alpha, l) \equiv \mu L(T, l)\Phi < T, l >$ avec
$\Phi \equiv \forall X.(X(T, Nil)$
$\rightarrow (\forall a.T(a) \rightarrow \forall x.L(T, x) \rightarrow X(T, Cons(a, x))) \rightarrow X(T, l))$

On obtiendra :

$$\frac{\Gamma_1 \vdash P(Nil) \quad \Gamma_2 \vdash \forall a : \alpha.\forall l : \alpha\ list.P(l) \rightarrow P(Cons(a,l))}{\Gamma_1, \Gamma_2, x : \alpha\ list \vdash P(x)}\ rec_{list}$$

Pour prouver $List(T,x) \rightarrow P(x)$ *dans TTR, on élimine le* μ*, il reste à prouver :*
$(\forall X.(X(T,Nil) \rightarrow (\forall a : T.\forall l.P(l) \rightarrow X(T,Cons(a,l))) \rightarrow X(x))) \rightarrow P(x)$*. En instanciant* X *avec* P*, il reste à prouver :*
$P(Nil)$ *et* $\forall a : T.\forall l.P(l) \rightarrow P(Cons(a,l))$*, ce qui est exactement la règle* rec_{list}*.* □

La dernière proposition nous permet de dire que si nous avons un terme de TT1.5, et que son évaluation termine, l'interprétation de ce terme est bien typée dans TTR.

Proposition 7 (Correction du typage ML)
Pour tout terme ML `t` *et tout type ML* `A`*, si on a* `t : A` *et l'évaluation de* `t` *termine, alors nous pouvons prouver dans TTR* $A(t)$*, avec* A *et* t *les interprétations respectives dans TTR de* `A` *et* `t`*.*

On prouve aisément cette proposition avec une induction structurelle sur l'arbre de typage dans TT1.5.

Les deux dernières extensions ont pour corollaire que les types algébriques ML sont bien une extension conservative.

Nous pouvons maintenant affirmer que notre système logique, décrit à dessein de manière naïve est correct. Le plongement de celui-ci dans une théorie bien connue nous permet de garantir la correction du système, et toutes les bonnes propriétés qu'il est souhaitable d'avoir.

Nous avions exprimé dans une remarque dans 4.1.1 que si nous acceptions la possibilité d'unifier une variable de type ML avec un type fonctionnel ML, nous perdions la propriété d'avoir un représentant canonique pour chaque objet d'un type. Nous pensons que ce n'est pas gênant dans le cadre de la preuve de programme, dans la mesure où dans les langages de programmation, on n'a pas d'égalité extensionnelle sur les fonctions.

5. Conclusion

Le but de ce travail était de construire une plate-forme de travail logique qui soit d'usage « aisé » pour l'utilisateur final. L'objectif à terme est d'implémenter ce système sous forme d'un logiciel d'aide à la preuve de programmes ML, où l'utilisateur pourrait écrire ses programmes ML, et utiliser des tactiques simples (mais non primitives) pour construire ses preuves. Nous voudrions que des preuves simples soient simples à écrire. Ce système pourra certainement être moins puissant (logiquement) que les systèmes basés sur les logiques d'ordre supérieur, mais nous pensons qu'il sera suffisant.

Les travaux futurs comprennent notamment l'extension du sous-ensemble de ML que l'on traitera : nous voudrions utiliser comme langage sous-jacent le $\lambda\mu$-calcul de M. Parigot [8] pour pouvoir exprimer certains des aspects impératifs de ML. Nous voudrions également pouvoir introduire une notion de théorie, similaire à celles que l'on a dans EML [4] ou ACL2 [5].

Références

[1] Jean-Raymond Abrial. *The B-Book : Assigning Programs to Meanings.* Cambridge Univ Press, 1996.

[2] R. Boyer and J. Moore. *A Computational Logic.* Academic Press, 1979.

[3] Joëlle Despeyroux. Sémantique Naturelle : Spécifications et Preuves. Technical report, INRIA, 1998.

[4] Stephan Kahrs, Donald Sannella, and Andrzej Tarlecki. The Definition of Extended ML : a gentle introduction. 1995.

[5] Matt Kaufmann and J Strother Moore. A Precise Description of the ACL2 Logic. Technical report, Computational Logic, Inc., 1997.

[6] Jean-Louis Krivine and Michel Parigot. Programming with proofs. In *SCT 87*.

[7] Daniel Leivant. Reasonning about functional programs and complexity classes associated with type discipline. In *FOCS*.

[8] Michel Parigot. $\lambda\mu$-calculus : an Algorithmic Interpretation of Classical Natural Deduction. In *LNCS 624*.

[9] Michel Parigot. Recursive Programming with Proofs.