Tactique réflexive de dérivation formelle*

Tahina Ramananandro
École Normale Supérieure – Paris

16 février 2006

1 Objectifs

On veut écrire une fonction de dérivation formelle, `deriv`, dans un langage restreint de fonctions de R dans R, puis prouver la correction du dérivateur formel, i.e. qu'il est conforme avec la définition.

Définition 1.1 (Correction de l'algorithme `deriv`) *On dit que l'algorithme `deriv` est correct sur le langage de fonctions $\mathcal{L}$ si, et seulement si, pour tout f terme du langage de fonctions $\mathcal{L}$, on a :*

$$\forall x \in \mathbb{R}, \forall \epsilon > 0, \exists \delta > 0, \forall h \in]0, \delta[: \frac{[f](x+h) - [f](x)}{h} - [\mathtt{deriv} f](x) < \epsilon$$

où $[.]$ est une fonction d'interprétation du langage $\mathcal{L}$ vers les fonctions de $\mathbb{R}$ dans $\mathbb{R}$, vérifiant $[f+g] := [f]+[g]$, etc.

On utilise pour cela le système d'aide à la preuve (ou *assistant de preuve*) Coq[1].

Le but de ce projet est non seulement de nous familiariser avec la preuve de théorèmes en Coq (avec en particulier l'utilisation de la *librairie standard* de Coq, contenant une formalisation des réels dans les modules `Reals` et associés), mais aussi avec le langage de *tactiques* $\mathcal{L}_{tac}$ du système Coq. On se reportera au sujet du projet, par Bruno Barras[2], pour les objectifs détaillés.

Les fichiers source Coq constituant le travail que j'ai réalisé pour ce projet sont disponibles sur mon site web[3].

2 Fonctions infiniment dérivables en tout point (`derivation_formelle.v`)

Dans un premier temps, on considère un langage décrivant les fonctions que l'on peut obtenir avec :

– l'identité, les fonctions constantes, le sinus, le cosinus, l'exponentielle
– somme, différence, produit, composition et élévation à une puissance constante (au sens du produit $\times$)[4] de telles fonctions

Cela nous conduit à la définition suivante du type informatif inductif `fonction`, qui décrit donc en quelque sorte la *grammaire* donnant la *syntaxe* des termes du langage de fonctions considéré :

* Mastère Parisien de Recherche en Informatique (MPRI, `http://mpri.master.univ-paris7.fr`), année 2005–2006, cours 2-7-2 : «Assistants de preuve»

[1] `http://coq.inria.fr`

[2] `http://pauillac.inria.fr/~barras/mpri/projet.ps`

[3] `http://www.eleves.ens.fr/home/ramanana/travail/coqderiv/`

[4] Dans le sujet, on parle de «puissance de fonctions». Cette notion est ambiguë *a priori*, car elle peut être vue de trois façons :
– la puissance f^g de deux fonctions est définie comme $x \mapsto \exp(g(x) \ln f(x))$. Cette quantité n'est pas toujours définie (que se passe-t-il si $f(x) < 0$?)
– la puissance f^n en tant qu'itérée n-ième de fonctions (i.e. au sens de la composition $\circ$) est définie sur tout $\mathbb{R}$ si f l'est. Cependant, la dérivée d'une telle fonction est $x \mapsto \prod_{i=0}^{n-1} f'(f^i(x))$ (notation puissance toujours au sens de $\circ$, mais produit Π au sens de $\times$), formule dont la preuve ne serait pas aisée ne serait-ce qu'à cause de sa représentation en Coq
– en revanche, la puissance f^n au sens du produit $\times$, dont la dérivée est $x \mapsto nf'(x)f^{n-1}(x)$, est un objet beaucoup plus courant et la preuve est beaucoup plus simple, c'est pourquoi nous avons choisi cette interprétation de la puissance.


```
Inductive fonction : Set :=
| Id : fonction
| Const : R -> fonction
| Exp : fonction
| Sin : fonction
| Cos : fonction
| Plus : fonction -> fonction -> fonction
| Moins : fonction -> fonction -> fonction
| Fois : fonction -> fonction -> fonction
| Compo : fonction -> fonction -> fonction
| Puiss : fonction -> nat -> fonction
.
```

Qui dit syntaxe dit *sémantique*, et donc *interprétation* des termes de ce langage vers les fonctions de $\mathbb{R}$ dans $\mathbb{R}$. Cette fonction d'interprétation est définie par récurrence sur la structure du terme du langage de fonctions :

```
Fixpoint interp (f : fonction) : R -> R :=
match f with
| Id => fun x => x
| Const k => fun _ => k
| Exp => exp
| Sin => sin
| Cos => cos
| Plus g h => fun x => interp g x + interp h x
| Moins g h => fun x => interp g x - interp h x
| Fois g h => fun x => interp g x * interp h x
| Compo g h => fun x => interp g (interp h x)
| Puiss f n => fun x => pow (interp f x) n
end.
```

L'algorithme de dérivation formelle est également défini par récurrence structurelle :

```
Fixpoint deriv (f : fonction) : fonction :=
match f with
| Id => Const 1
| Const _ => Const 0
| Exp => Exp
| Sin => Cos
| Cos => Moins (Const 0) Sin
| Plus g h => Plus (deriv g) (deriv h)
| Moins g h => Moins (deriv g) (deriv h)
| Fois g h => Plus (Fois (deriv g) h) (Fois g (deriv h))
| Compo g h => Fois (Compo (deriv g) h) (deriv h)
| Puiss f O => Const 0
| Puiss f (S n) => Fois (Fois (Const (INR (S n))) (deriv f)) (Puiss f n)
end.
```

Le théorème de correction s'énonce alors très simplement :

```
Theorem deriv_correcte : forall (f : fonction) (x : R),
derivable_pt_lim (interp f) x (interp (deriv f) x).
```

On le prouve par induction sur la structure du terme f du langage de fonctions.

Les cas du sinus, cosinus,…sont déjà montrés, en quelque sorte, dans la librairie standard (théorèmes `derivable_pt_lim_sin`, `derivable_pt_lim_cos`…qu'il suffit d'appliquer immédiatement). Ceux des constantes et de l'identité aussi, mais j'ai choisi de traiter ces deux cas à la main.

Pour les termes plus complexes comme la somme, on se ramène à appliquer des théorèmes comme `derivable_pt_lim_plus` et on utilise l'hypothèse d'induction.

La puissance constitue un cas particulier : comme le théorème n'est prédéfini que pour la fonction puissance elle-même, $x \mapsto x^n$, je suis obligé de dire que f^n est en fait la composée $(x \mapsto x^n) \circ f$, et d'utiliser alors les deux théorèmes prédéfinis, celui pour la fonction puissance et celui pour la composée.

Ce *background* théorique sur la correction de la fonction de dérivation formelle étant jeté, il s'agit maintenant d'écrire une tactique permettant d'*automatiser* le calcul de la dérivée f' d'une fonction f *a priori* quelconque (de $\mathbb{R} \to \mathbb{R}$) et l'ajout direct en cours de démonstration (sans avoir aucune preuve supplémentaire à faire) d'une hypothèse correspondante.

Il s'agit donc de traduire f en un terme h du langage de fonctions s'interprétant sur f, afin de calculer la dérivée de f de manière automatique en utilisant l'algorithme de dérivation formelle dont on a montré la correction.

J'ai choisi de distinguer la tactique d'«inversion» d'une fonction f, cette étape étant *a priori* indépendante de la dérivation (elle pourrait être intéressante pour d'autres manipulations) :

```
Ltac inversion_fonction f :=
 match constr:f with
  | (comp ?f1 ?f2) =>
      let g1 := inversion_fonction f1 with g2 := inversion_fonction f2 in constr:(Compo g1 g2)
  | (plus_fct ?f1 ?f2) =>
      let g1 := inversion_fonction f1 with g2 := inversion_fonction f2 in constr:(Plus g1 g2)
  | (minus_fct ?f1 ?f2) =>
      let g1 := inversion_fonction f1 with g2 := inversion_fonction f2 in constr:(Moins g1 g2)
  | (mult_fct ?f1 ?f2) =>
      let g1 := inversion_fonction f1 with g2 := inversion_fonction f2 in constr:(Fois g1 g2)
  | (pow_fct ?f ?m) =>
      let g := inversion_fonction f in constr:(Puiss g m)
  | exp => constr:Exp
  | sin => constr:Sin
  | cos => constr:Cos
  | id => constr:Id
  | (fun x => x) => constr:Id
  | (fun _ => ?y) => constr:(Const y)
  | fct_cte ?x => constr:(Const x)
  | (fun x => ?f x) => inversion_fonction f
end.
```

Remarque : L'intérêt de $\mathcal{L}_{tac}$ est de pouvoir analyser la structure de n'importe quel terme *a priori* (plus familièrement, «faire du *pattern-matching*»). Cependant, à l'heure actuelle (2006 : Coq 8.0), ce langage de tactiques n'est pas assez puissant car l'analyse «sous les `fun`» est très limitée. Par exemple, si j'écris directement la fonction que je veux analyser comme ceci : `fun x => cos x + sin (x + Pi)`, alors je n'aurai aucun moyen de reconnaître des «sous-termes sous le contexte [`x`]», c'est-à-dire qu'en pratique je ne pourrai pas isoler les deux parties `cos x` et `sin (x + Pi)`.[5] Je dois donc entrer cette fonction sous une forme moins lisible : `plus_fct cos (comp sin (plus_fct id (fct_cte Pi)))`.

J'intègre ensuite cette étape d'«interprétation à l'envers» dans la tactique de dérivation, qui, moyennant quelques réécritures, rajoute une hypothèse sur f et sa dérivée, grâce à l'application du théorème de correction :

```
Ltac derive f :=
  let f0 := inversion_fonction f in
  let h := fresh "h" in (
  assert (h: forall y, derivable_pt_lim f y (interp (deriv f0) y));
  [ apply (deriv_correcte f0) | simpl in h ]
).
```

3 Fonctions dérivables sur un sous-ensemble de $\mathbb{R}$ (`derivation_formelle_avec_domaine.v`)

[5] Il ressort d'une discussion avec Jean-Christophe Filliâtre (`http://www.lri.fr/~filliatr`), que l'on pourrait éventuellement étendre les capacités de $\mathcal{L}_{tac}$ en en écrivant «des bouts en ML», c'est-à-dire en intégrant du code OCaml (`http://caml.inria.fr/ocaml`) qui pourrait effectuer une telle analyse sous les `fun` en manipulant les termes à plus bas niveau. Mais par manque de temps, je n'ai pas pu développer cette approche.

L'objectif de cette partie est d'intégrer au développement (c'est-à-dire, en l'occurrence, de dériver) des fonctions non dérivables en tout point : logarithme, valeur absolue, division, inverse. Il faut donc définir, en plus d'un algorithme de dérivation formelle, un algorithme de «dérivabilité», en fait de détermination d'une condition *suffisante* de dérivabilité.

Alors que dans le développement précédent, j'avais défini le terme `Moins` dans la grammaire, ici je considère plutôt des termes `Oppose` et `Inverse`, et je redéfinis `Moins` et `Sur` comme du *sucre syntaxique* :

```
Inductive fonction : Set :=
| Const : R -> fonction
| Id : fonction
| Exp : fonction
| Sin : fonction
| Cos : fonction
| Log : fonction
| Valeur_absolue : fonction
| Racine : fonction
| Oppose : fonction -> fonction
| Inverse : fonction -> fonction
| Plus : fonction -> fonction -> fonction
| Fois : fonction -> fonction -> fonction
| Compo : fonction -> fonction -> fonction
| Puiss : fonction -> nat -> fonction
.

Definition Moins f g := Plus f (Oppose g).
Definition Sur f g := Fois f (Inverse g).
```

Ce choix pour `Moins` et `Sur` est d'ailleurs proche de la méthode que l'on rencontre dans la librairie standard de Coq, pour la définition par exemple de `Rminus` et `Rdiv`, fondée sur les préexistants `Ropp` et `Rinv`.

Alors, la fonction d'interprétation est définie par récurrence sur la structure des termes de la grammaire, et je montre, *à côté* de cette définition, les propriétés immédiates que l'on attend pour les sucres syntaxiques `Moins` et `Sur`.

```
Fixpoint interp (f : fonction) : R -> R :=
match f with
| Id => id
| Const k => fct_cte k
| Exp => exp
| Sin => sin
| Cos => cos
| Racine => sqrt
| Valeur_absolue => Rabs
| Log => ln
| Oppose f => opp_fct (interp f)
| Inverse f => inv_fct (interp f)
| Plus g h =>  plus_fct  (interp g) (interp h)
| Fois g h => mult_fct (interp g) (interp h)
| Compo g h => comp (interp g) (interp h)
| Puiss g n => fun x => pow (interp g x) n
end.

Theorem interp_moins : forall f g,
interp (Moins f g) = minus_fct (interp f) (interp g).
auto with real.
Qed.

Theorem interp_sur : forall f g,
interp (Sur f g) = div_fct (interp f) (interp g).
```

```
auto with real.
Qed.
```

Remarque : contrairement au développement précédent, j'ai choisi d'utiliser les fonctions prédéfinies `plus_fct`, etc. au lieu des structures `fun`. En effet, ce choix m'a permis d'alléger la preuve du théorème de correction de l'algorithme de dérivation : moins de réécritures, car c'est sous cette forme que sont formulés les théorèmes de dérivation dans la librairie standard.

Alors, l'algorithme de dérivation formelle s'écrit à nouveau par récurrence sur la structure du terme du langage de fonctions :

```
Fixpoint deriv (f : fonction) : fonction :=
match f with
| Id => Const 1
| Const _ => Const 0
| Exp => Exp
| Sin => Cos
| Cos => Oppose Sin
| Racine => Inverse (Fois (Const 2) Racine)
| Log => Inverse Id
| Valeur_absolue => Sur Id Valeur_absolue
| Oppose g => Oppose (deriv g)
| Inverse g => Oppose (Sur (deriv g) (Fois g g))
| Plus g h => Plus (deriv g) (deriv h)
| Fois g h => Plus (Fois (deriv g) h) (Fois g (deriv h))
| Compo g h => Fois (Compo (deriv g) h) (deriv h)
| Puiss f O => Const 0
| Puiss f (S n) => Fois (Fois (Const (INR (S n))) (deriv f)) (Puiss f n)
end.
```

Avec l'arrivée de fonctions comme la valeur absolue, il faut parler des *domaines de dérivabilité* des fonctions considérées. La connaissance de ces domaines, bien qu'inutile pour l'algorithme de dérivation lui-même, est en revanche nécessaire pour montrer sa correction. C'est pourquoi ces domaines sont exprimés par des prédicats P d'une variable réelle (`R -> Prop`). Ils sont définis par induction sur la structure du terme de fonction.

```
Inductive derivable : fonction -> R -> Prop :=
| derivable_id : forall (x : R), derivable Id x
| derivable_const : forall (x y : R), derivable (Const y) x
| derivable_exp : forall (x : R), derivable Exp x
| derivable_cos: forall (x:R), derivable Cos x
| derivable_sin : forall (x : R), derivable Sin x
| derivable_log : forall (x : R),
  x > 0 -> derivable Log x
| derivable_racine : forall (f : fonction) (x : R),
  x > 0 -> derivable Racine x
| derivable_valeur_absolue : forall (x : R),
  x <> 0 -> derivable Valeur_absolue x
| derivable_oppose : forall (f : fonction) (x: R),
derivable f x -> derivable (Oppose f) x
| derivable_inverse : forall (f : fonction) (x: R),
derivable f x -> interp f x <> 0 -> derivable (Inverse f) x
| derivable_plus : forall (f g : fonction) (x : R),
derivable f x -> derivable g x -> derivable (Plus f g) x
| derivable_fois : forall (f g : fonction) (x : R),
derivable f x -> derivable g x -> derivable (Fois f g) x
| derivable_compo : forall (f g : fonction) (x : R),
  derivable f (interp g x) -> derivable g x -> derivable (Compo f g) x
| derivable_puiss_O : forall (f : fonction) (x : R),
  derivable (Puiss f O) x
```



```
| derivable_puiss_S : forall (f : fonction) (x : R) (n : nat),
  derivable f x -> derivable (Puiss f (S n)) x
.
```

Il s'agit là de *conditions suffisantes* de dérivabilité : pour x réel, si $P(x)$ est vraie, alors la fonction est dérivable en x. Mais ces conditions ne sont pas toujours également nécessaires. En effet, dans le cas général la fonction $f \times g$ est dérivable en x dès que f et g le sont en x, condition que l'on trouvera algorithmiquement avec l'aide de la définition ci-dessus. En revanche, des fonctions comme $x \mapsto 0 \times |x|$ sont dérivables en tout point (alors que $|.|$ n'est pas dérivable en 0), mais pour le découvrir algorithmiquement, il aurait peut-être fallu raffiner la définition de `derivable` en «poussant» un petit peu plus l'analyse.

La correction de l'algorithme de dérivation formelle s'énonce donc simplement en intégrant ces conditions suffisantes de dérivabilité :

```
Theorem deriv_correcte : forall (f : fonction) (x : R),
derivable f x ->
derivable_pt_lim (interp f) x (interp (deriv f) x).
```

On le prouve également par induction sur la structure du terme f. Lorsque j'ai besoin d'utiliser l'hypothèse d'induction, j'ai besoin de connaître l'hypothèse de dérivabilité pour un sous-terme de f. Je peux obtenir cette hypothèse grâce à la tactique `inversion` appliquée à l'hypothèse `derivable f x`.

Moyennant la prise en compte de cette nouvelle hypothèse de dérivabilité, la preuve s'effectue comme avant grâce aux théorèmes de la librairie standard, sauf pour l'inverse d'une fonction, pour laquelle il n'existe pas de théorème prédéfini : je suis obligé de me ramener à la division en passant de `inv_fct h` à `div_fct (fct_cte 1) h` (qui ne sont pas convertibles car il faut un théorème[6] en plus d'un dépliage de définition pour aller de l'un à l'autre), afin d'utiliser le théorème existant `derivable_pt_lim_div`.

L'arrivée des fonctions supplémentaires étend la définition de la tactique de traduction («interprétation inverse») des fonctions de `R -> R` :

```
Ltac inversion_fonction f :=
 match constr:f with
  | (plus_fct ?f1 ?f2) =>
    let g1 := inversion_fonction f1 with g2 := inversion_fonction f2 in constr:(Plus g1 g2)
  | (minus_fct ?f1 ?f2) =>
    let g1 := inversion_fonction f1 with g2 := inversion_fonction f2 in constr:(Moins g1 g2)
  | (mult_fct ?f1 ?f2) =>
    let g1 := inversion_fonction f1 with g2 := inversion_fonction f2 in constr:(Fois g1 g2)
  | (pow_fct ?f ?m) =>
    let g := inversion_fonction f in constr:(Puiss g m)
  | (div_fct ?f1 ?f2) =>
    let g1 := inversion_fonction f1 with g2 := inversion_fonction f2 in constr:(Sur g1 g2)
  | (inv_fct ?f) =>
    let g := inversion_fonction f in constr:(Inverse g)
  | (opp_fct ?f) => let g := inversion_fonction f in constr:(Oppose g)
  | exp => constr:Exp
  | sin => constr:Sin
  | cos => constr:Cos
  | ln => constr:Log
  | Rabs => constr:Valeur_absolue
  | sqrt => constr:Racine
  | (fun x => x) => constr:Id
  | id => constr:Id
  | (fun _ => ?x) => constr:(Const x)
  | fct_cte ?x => constr:(Const x)
  | (fun x => ?f x) => inversion_fonction f (* eta-conversion *)
end.
```

[6] En l'occurrence : le fait que 1 soit neutre pour $\times$...

Si cette extension est simple, en revanche, l'introduction des prédicats de dérivabilité corse le problème de la dérivation automatique. Certes, si la tactique est quasi inchangée :

```
Ltac eta_contraction f := match f with
| (fun x => ?g x) => eta_contraction g
| _ => constr:f
end.

Ltac derive f p :=
  let f1 := eta_contraction f in
  let f0 := inversion_fonction f1 in
  let g0 := constr:(interp (deriv f0)) in
  let h := fresh "h" in (
  assert (h: forall x : R, p x -> derivable_pt_lim f1 x (g0 x)) ; [
   intro x ; intro ; apply (deriv_correcte f0 x) ; auto
   | simpl in h]
  ).
```

La dérivation automatique n'est plus guère possible. Si par exemple on exécute la tactique :

```
derive (div_fct (fct_cte 2) (fct_cte 2)) (fun _:R => True).
```

alors on se retrouve avec le but à prouver :

```
derivable (Sur (Const 2) (Const 2) x)
```

qui, avec l'utilisation de la définition inductive de `derivable`, amène à prouver que $2 \neq 0$...

4 Variables de fonctions (`derivation_formelle_varmap.v`)

Cette fois-ci, on veut étendre notre langage de fonctions avec des variables[7]. Donc, lorsque l'on voudra interpréter ces termes de fonctions avec variables, il faudra joindre à notre terme une *valuation*.

Cette valuation prend la forme d'un *arbre* (le type `varmap`) dont les noeuds sont étiquetés par des fonctions de `R -> R`. Dans les termes, les variables (type `index`) sont en fait les *chemins* qu'il faudra suivre dans l'arbre lors de l'interprétation, pour trouver la fonction correspondante.

```
Inductive fonction : Set :=
...
| Var : index -> fonction
.
```

Du coup, un problème majeur s'est posé : que se passe-t-il si je tente d'interpréter une fonction dans une valuation incomplète ? En d'autres termes, si une variable représente un chemin inexistant dans l'arbre ?

J'ai d'abord souhaité écrire une fonction d'interprétation partielle, en considérant plusieurs solutions :

– *La fonction d'interprétation renvoie un type somme : la fonction interprétée, ou bien une variable dans le terme qui n'a pas pu être instanciée (en guise d'indication d'erreur).* Cette solution alourdit considérablement les preuves ; en outre, elle aboutit au problème grave suivant. Considérons que j'applique une transformation T aux variables d'un terme f (par exemple, rajouter un pas vers la gauche), et que l'arbre de valuation A initial subisse une transformation T' compatible avec T (c'est-à-dire telle que pour toute variable i, l'élément de chemin $T(i)$ dans $T'(A)$ soit le même que l'élément de chemin i dans A). Alors je souhaite que l'interprétation de $T(f)$ sous la valuation $T'(A)$ soit la même que l'interprétation de f sous A. Mais ceci est faux si f a des variables non instanciées sous A, car les variables renvoyées en indication d'erreur ne sont pas les mêmes.

– *La fonction d'interprétation renvoie un type option : la fonction interprétée, ou rien si une variable dans le terme n'a pas pu être instanciée.* Cette solution alourdit tout autant les preuves. Certes, elle permet de contourner le problème posé par la solution précédente, mais elle nécessite d'écrire les théorèmes de la forme `interp v f = Some h -> ...`, ce qui pose des problèmes de convertibilité (que je n'ai pas vu venir, alors qu'ils auraient aussi été soulevés par la solution précédente) : j'ai abouti à `Some h = Some (-f) -> Some f = Some (-h)`, ce qui est bien sûr indémontrable. Introduire alors une notion d'équivalence de fonctions (égalité en tout point) rend le développement rigoureusement illisible à ce niveau.

[7]Il ne s'agit pas des variables de Coq, mais d'un concept de variables au sein du langage de fonctions

– *La fonction d'interprétation dépend d'un prédicat inductif dit d'«interprétabilité» (i.e. toutes les variables de la fonction sont instanciées sous la valuation considérée), et est construite par induction sur la structure de la preuve de ce prédicat pour la fonction à interpréter.* Cette solution alourdit les preuves encore plus que les deux précédentes (ne serait-ce que la définition de la fonction d'interprétation, avec `Definition ...Defined` et en mode preuve plutôt qu'avec `Fixpoint`), et les raisonnements font apparaître les fonctions d'induction et de récursion `interpretable_rec`, etc. ce qui est très malcommode.

J'ai donc revu mes ambitions à la baisse, et j'ai préféré définir une fonction d'interprétation *a priori* totale, associant la fonction nulle à toute variable non instanciée. Pourquoi la fonction nulle ? Car, supposant (généreusement) que si une variable n'est pas instanciée sous une certaine valuation, sa «dérivée» ne l'est pas non plus dans la valuation où on va interpréter la dérivée de la fonction, il faut trouver une fonction dont la dérivée soit elle-même[8].

```
Fixpoint interp (v : varmap (R -> R)) (f : fonction) {struct f} : (R -> R) :=
match f with
| Id => id
| Const k => (fct_cte k)
| Exp => exp
| Sin => sin
| Cos => cos
| Racine => sqrt
| Valeur_absolue => Rabs
| Log => ln
| Oppose g => opp_fct (interp v g)
| Inverse g => inv_fct( interp v g )
| Plus g h => plus_fct (interp v g) (interp v h)
| Fois g h => mult_fct (interp v g) (interp v h)
| Compo g h => comp (interp v g) (interp v h)
| Puiss g n => fun x => pow (interp v g x) n
| Var x => varmap_find (fct_cte 0) x v
end.
```

On veut que si f s'interprète dans une valuation v, alors sa dérivée f' s'interprète dans une valuation $V =$ `Node_vm` $\alpha\ v\ v'$, où α est arbitraire et v' est une valuation telle que pour toute variable i, une fonction de chemin i dans v soit interprétée avec une fonction de chemin i dans v'.

Fixons deux telles valuations v et v'. Considérons deux variables i et j instanciées dans v et v'. Alors la fonction $i \times j$ admet une interprétation dans v. Dériver cette fonction naïvement donnerait alors $i' \times j + i \times j'$, avec encore les variables i et j présentes, mais à interpréter dans V et non dans v. C'est absurde. En fait, on aurait voulu remplacer i par `Left_idx` i, c'est-à-dire rajouter un pas vers la gauche devant le chemin représenté par i, ce pas vers la gauche permettant de passer de V à son sous-arbre gauche v où on pourra instancier la variable représentée par le reste du chemin, à savoir i.

Cela nécessite donc que, avant de dériver formellement f, on modifie toutes ses variables en leur rajoutant un pas vers la gauche. On dit que l'on va «gauchir» f :

```
Fixpoint gauchir (f : fonction) : fonction := match f with
| Var v => Var (Left_idx v)
| Oppose g => Oppose (gauchir g)
| Inverse g => Inverse (gauchir g)
| Plus g h => Plus (gauchir g) (gauchir h)
| Fois g h => Fois (gauchir g) (gauchir h)
| Compo g h => Compo (gauchir g) (gauchir h)
| Puiss g n => Puiss (gauchir g) n
| e => e
end.
```

Alors, je montre que si j'interprète une fonction f dans une varmap v, j'obtiens le même résultat qu'en interprétant `gauchir` f dans une varmap `Node_vm` $\alpha\ v\ z$, avec α et z arbitraires. Cette égalité me servira plusieurs fois dans la suite.

[8]Les fonctions $k \exp$ conviennent donc –tiens, même en informatique on doit résoudre des équations différentielles !–, mais il est extrêmement peu naturel de prendre $k \neq 0$; voir par exemple les langages BASIC qui instancient par défaut à 0 toutes les variables numériques non définies.

```
Theorem gauchir_correcte : forall f v bidon_f bidon_v,
interp v f = interp (Node_vm bidon_f v bidon_v) (gauchir f).
```

On notera que cette égalité est vraie même si f contient des variables non instanciées dans v.

Revenant au cas général, on veut alors que i' renvoie, dans V, à la dérivée de la fonction de chemin i dans v. Or, cette dérivée admet un chemin i dans v', et v' est le sous-arbre droit de V. Donc on a $i' =$ `Right_idx` i. C'est-à-dire que, dans `gauchir` f, où i est remplacée par `Left_idx` i , il suffit de modifier toutes les variables en changeant leur premier pas, qui est un pas vers la gauche, en un pas vers la droite.

D'où la définition suivante de la dérivation formelle `deriv`, qui s'appuie sur `derivg`, fonction supposant que son entrée est «gauche» (i.e. a toutes ses variables commençant par un pas vers la gauche) et effectuant l'opération ci-dessus ainsi que toutes les autres dérivations formelles «classiques».

```
Fixpoint derivg  (f : fonction) : fonction :=
match f with
| Id => Const 1
| Const _ => Const 0
| Exp => Exp
| Sin => Cos
| Cos => Oppose Sin
| Racine => (Inverse (Fois (Const 2) Racine))
| Log => Inverse Id
| Valeur_absolue => Sur Id Valeur_absolue
| Oppose g => Oppose (derivg g)
| Inverse g => Oppose (Sur (derivg g) (Fois g g))
| Plus g h => Plus (derivg g) (derivg h)
| Fois g h => Plus (Fois (derivg g) h) (Fois g (derivg h))
| Compo g h => Fois (Compo (derivg g) h) (derivg h)
| Puiss _ O => Const 0
| Puiss g (S n) => Fois (Fois (Const (INR (S n))) (derivg g)) (Puiss g n)
| Var (Left_idx i) => Var (Right_idx i)
| Var v => Var v
end.

Definition deriv f := derivg (gauchir f).
```

On ne s'occupe pas du comportement de `derivg` sur une variable qui ne commencerait pas par un pas vers la gauche.

Les domaines de dérivabilité posent également problème. L'arrivée des variables exige que la condition suffisante de dérivabilité dépende non pas d'une mais de *deux* valuations :

- la valuation v pour pouvoir instancier les variables : en effet, par exemple pour exprimer la condition de dérivabilité de l'inverse d'une fonction, on a besoin de son interprétation. Plus simplement, si i est une variable quelconque, la dérivabilité de $1/i$ dépend de la valeur de i, et donc de la valuation v
- une valuation u de prédicats (u de type `varmap (R -> Prop)`), telle que pour tout chemin i, si le prédicat de chemin i dans u est vrai en x réel, alors la fonction de chemin i dans v est dérivable en x

Ayant introduit une nouvelle valuation u, il faut s'attacher à ce qui peut arriver si i n'est pas un chemin valide dans u. En fait, dans ce cas, on considère que le prédicat de dérivabilité est `False`. En effet, c'est la condition suffisante la plus faible dont on dispose ; elle convient toujours.

```
Inductive derivable (v : varmap (R -> Prop)) (w : varmap (R -> R)) : fonction -> R -> Prop :=
| derivable_id : forall (x : R), derivable v w Id x
| derivable_const : forall (x y : R), derivable v w (Const y) x
| derivable_exp : forall (x : R), derivable v w Exp x
| derivable_cos: forall (x:R), derivable v w Cos x
| derivable_sin : forall (x : R), derivable v w Sin x
| derivable_log : forall (x : R),
  x > 0 -> derivable v w Log x
| derivable_racine : forall (f : fonction) (x : R),
```



```
  x > 0 -> derivable v w Racine x
| derivable_valeur_absolue : forall (x : R),
  x <> 0 -> derivable v w Valeur_absolue x
| derivable_oppose : forall (f : fonction) (x: R),
derivable v w f x -> derivable v w (Oppose f) x
| derivable_inverse : forall (f : fonction) (x: R),
derivable v w f x -> interp w f x <> 0 -> derivable v w (Inverse f) x
| derivable_plus : forall (f g : fonction) (x : R),
derivable v w f x -> derivable v w g x -> derivable v w (Plus f g) x
| derivable_fois : forall (f g : fonction) (x : R),
derivable v w f x -> derivable v w g x -> derivable v w (Fois f g) x
| derivable_compo : forall (f g : fonction) (x : R),
  derivable v w f (interp w g x) -> derivable v w g x -> derivable v w (Compo f g) x
| derivable_puiss_O : forall (f : fonction) (x : R),
  derivable v w (Puiss f O) x
| derivable_puiss_S : forall (f : fonction) (x : R) (n : nat),
  derivable v w f x -> derivable v w (Puiss f (S n)) x
| derivable_var : forall (i : index) (x : R),
varmap_find (fun _=>False) i v x -> derivable v w (Var i) x
.
```

Le théorème de correction de l'algorithme de dérivation formelle s'écrit alors (les noms des variables ayant changé, v désignant la valuation des prédicats de dérivabilité, w la valuation des fonctions et w' celle des dérivées) :

```
Theorem deriv_correcte : forall (f : fonction) (w w' : varmap (R->R))
 (v : varmap (R -> Prop)) (x : R),
derivable v w f x ->
(forall (i:index), forall x : R, varmap_find (fun _=>False) i v x ->
  derivable_pt_lim (interp w (Var i)) x (interp w' (Var i) x)) ->
forall (bidon_fonct : R -> R),
derivable_pt_lim (interp w f) x (interp (Node_vm bidon_fonct w w')
(deriv f) x).
```

Il exige les hypothèses suivantes :

- f est dérivable sous la valuation v des prédicats de dérivabilité et la valuation w d'interprétation des fonctions
- pour tout chemin i, toute fonction de chemin i dans w se dérive, en x réel, en la valeur en x de la fonction de chemin i dans w', sous la condition suffisante indiquée par le prédicat de chemin i dans v

La preuve (par induction sur la structure de f) n'est guère plus compliquée que dans les développements précédents, si ce n'est quelques réécritures supplémentaires. Il se rajoute le cas de la variable, qui somme toute est vite traité : grâce à la tactique `inversion` sur l'hypothèse f dérivable, j'obtiens que la condition suffisante indiquée par la valuation v est vérifiée, ce qui me permet d'utiliser l'autre hypothèse du théorème.

Il est à noter que ce théorème reste vrai même si les variables ne sont pas instanciées dans les valuations considérées.

L'introduction des variables permet le traitement automatique de *toutes* les fonctions. En effet, jusque-là, les tactiques d'inversion de fonctions échouaient lorsque l'on indiquait des fonctions non reconnues : elles ne passaient pas le *pattern-matching*. Maintenant, il est possible de les prendre en compte en les intégrant aux valuations au fur et à mesure que l'on analyse la structure de la fonction.

On voudrait construire des arbres «équilibrés». À première vue, cette notion n'a pas grand sens, s'agissant d'arbres de fonctions, éléments dépourvus d'ordre total. Mais on veut simplement construire les arbres de façon à assurer un accès indexé *logarithmique* en la taille de l'arbre (i.e. en le nombre de fonctions dans la valuation). Voilà ce qui motive le choix de l'arbre plutôt que de la liste, dont le temps d'accès est linéaire.[9]

On voudrait remplir l'arbre «ligne par ligne». En programmation pratique, on peut écrire des algorithmes efficaces de construction en mémorisant un pointeur vers la prochaine «case libre». Mais n'oublions pas que $\mathcal{L}_{tac}$

[9]En Coq, il n'existe pas de «tableau», dont l'accès serait constant.

est purement fonctionnel[10]. On pourrait prendre une approche «monadique» (i.e. prendre l'ancien pointeur en entrée et renvoyer le nouveau en sortie), mais ce serait plutôt lourd à écrire.

Mon algorithme de construction par ajouts successifs n'utilise donc ni pointeurs ni monades. Pour ajouter un objet o à un arbre A :

1. si A est vide, je le rajoute à sa racine
2. sinon, soient g et d ses sous-arbres gauche et droit. Je regarde si g est complet.
3. si g n'est pas complet, alors j'ajoute récursivement o à g
4. sinon, je regarde si d est complet. S'il ne l'est pas, alors j'ajoute récursivement o à d
5. sinon, g et d sont complets. Dans ce cas, je compare la taille de g et de d, en mesurant leur hauteur, et j'ajoute récursivement o au plus petit des deux.

Or, vérifier qu'un arbre est complet est linéaire en la taille de l'arbre (en gros, on compte ses éléments). Donc, mon algorithme est clairement sous-optimal : chaque ajout est linéaire.

On sacrifie donc la construction, linéaire, à l'accès, logarithmique.

```
Ltac gauche_plus_grand g d :=
match constr:g with
| Empty_vm _ => constr:False
| Node_vm _ ?g2 _ =>
match constr:d with
| Empty_vm _ => constr:True
| Node_vm _ ?d2 _ => gauche_plus_grand g2 d2
end end.

Ltac complet v :=
match constr:v with
| Empty_vm _ => constr:True
| Node_vm _ ?g ?d =>
match complet g with
| True => complet d
| False => constr:False
end end.

Ltac ajouter o v :=
match constr:v with
| Empty_vm _ => constr:(Node_vm o (Empty_vm _) (Empty_vm _))
| Node_vm ?r ?g ?d =>
match complet g with
| True =>
match complet d  with
| True =>
match gauche_plus_grand g d with
| True => let d' := ajouter o d in constr:(Node_vm r g d')
| False => let g' := ajouter o g in constr:(Node_vm r g' d)
end
| False => let d' := ajouter o d in constr:(Node_vm r g d')
end
| False => let g' := ajouter o g in constr:(Node_vm r g' d)
end
end.
```

Remarque : il est possible de modifier légèrement la fonction d'ajout en rajoutant un cas de filtrage :

```
Ltac ajouter_sans_doublon o v :=
match constr:v with
| Empty_vm _ => constr:(Node_vm o (Empty_vm _) (Empty_vm _))
| Node_vm o _ _ => constr:v
| Node_vm ?r ?g ?d => ...
```

[10]C'est pourquoi la tactique `auto`, utilisant les bases de `Hint`, n'est pas exprimable en $\mathcal{L}_{tac}$.

En effet, $\mathcal{L}_{tac}$ permet de faire du *pattern-matching* plus fort que ce que l'on pourrait faire par exemple en OCaml (sans les clauses `when`), puisqu'il nous permet de tester, en même temps que le filtrage, l'égalité de deux termes non constants désignés par des variables. En l'occurrence, on rajoute la clause permettant de filtrer un arbre dont la racine serait étiquetée par l'objet que l'on voudrait ajouter (et qui est donc déjà présent).

Disposant de primitives pour ajouter un élément dans un arbre, on peut alors écrire la fonction qui permet de compléter un arbre avec les fonctions non reconnues que l'on trouve dans la fonction f. L'idée est de rajouter dans l'arbre, systématiquement mais sans doublon, toute fonction inconnue :

```
Ltac completer v f :=
match constr:f with
| (comp ?f1 ?f2) => let w := completer v f1 in  completer w f2
| (plus_fct ?f1 ?f2) => let w := completer v f1 in  completer w f2
| (minus_fct ?f1 ?f2) => let w := completer v f1 in  completer w f2
| (mult_fct ?f1 ?f2) => let w := completer v f1 in  completer w f2
| (pow_fct ?f ?m) => completer v f
| (div_fct ?f1 ?f2) => let w := completer v f1 in  completer w f2
| (inv_fct ?f) => completer v f
| (opp_fct ?f) => completer v f
| exp => constr:v
| sin => constr:v
| cos => constr:v
| ln => constr:v
| Rabs => constr:v
| sqrt => constr:v
| (fun x => x) => constr:v
| id => constr:v
| (fun _ => ?x) => constr:v
| fct_cte ?x => constr:v
| (fun x => ?f (?g x)) => let w := completer v f in completer w g (* eta-conversion avec composition *)
| (fun x => ?f x) => completer v f (* eta-conversion simple *)
| _ => ajouter_sans_doublon f v (* autres cas *)
end.
```

L'arbre des fonctions étant ainsi construit, il est alors possible d'étendre la tactique de traduction («interprétation inverse») en intégrant cet arbre. Cette tactique, s'appuie sur une tactique qui permet de rechercher une fonction dans l'arbre et de construire son index (i.e. le chemin qui y mène).

```
Ltac trouver f v :=
match constr:v with
| Node_vm f _ _ => constr:End_idx
| Node_vm _ ?g _ => let i := trouver f g in constr:(Left_idx i)
| Node_vm _ _ ?d => let i := trouver f d in constr:(Right_idx i)
| _ => fail "Une fonction n'a pu être trouvée dans la varmap. Complétez d'abord
la varmap avant d'inverser la fonction."
end.

Ltac inversion_fonction v f :=
 match constr:f with
  | (comp ?f1 ?f2) =>
     let g1 := inversion_fonction v f1 with g2 := inversion_fonction v f2 in constr:(Compo g1 g2)
  | (plus_fct ?f1 ?f2) =>
     let g1 := inversion_fonction v f1 with g2 := inversion_fonction v f2 in constr:(Plus g1 g2)
  | (minus_fct ?f1 ?f2) =>
     let g1 := inversion_fonction v f1 with g2 := inversion_fonction v f2 in constr:(Moins g1 g2)
  | (mult_fct ?f1 ?f2) =>
     let g1 := inversion_fonction v f1 with g2 := inversion_fonction v f2 in constr:(Fois g1 g2)
  | (pow_fct ?f ?m) =>
     let g := inversion_fonction v f in constr:(Puiss g m)
```

```
  | (div_fct ?f1 ?f2) =>
     let g1 := inversion_fonction v f1 with g2 := inversion_fonction v f2 in constr:(Sur g1 g2)
  | (inv_fct ?f) =>
     let g := inversion_fonction v f in constr:(Inverse g)
  | (opp_fct ?f) => let g := inversion_fonction v f in constr:(Oppose g)
  | exp => constr:Exp
  | sin => constr:Sin
  | cos => constr:Cos
  | ln => constr:Log
  | Rabs => constr:Valeur_absolue
  | sqrt => constr:Racine
  | (fun x => x) => constr:Id
  | id => constr:Id
  | (fun _ => ?x) => constr:(Const x)
  | fct_cte ?x => constr:(Const x)
  | (fun x => ?f (?g x)) =>
let k := constr:(comp f g) in inversion_fonction v k (* eta-conversion avec composition *)
  | (fun x => ?f x) => inversion_fonction v f (* eta-conversion simple *)
  | _ => let i := trouver f v in constr:(Var i) (* autres cas *)
  | _ => fail "Une fonction n'a pu être trouvée dans la varmap. Complétez d'abord
la varmap avant d'inverser la fonction."
end.
```

Ces deux tactiques présentent la particularité suivante de posséder plusieurs branches filtrant exactement les mêmes termes. Elles ne sont pas redondantes, mais au contraire, exploitent une particularité de $\mathcal{L}_{tac}$, attrayante pour certains, déroutante pour d'autres, à savoir le *backtracking* dans un filtrage : sachant qu'une séquence de tactiques peut échouer, si une branche de filtrage échoue, alors au lieu de faire échouer toute la tactique, on passe à la prochaine branche du filtrage, jusqu'à réussir, ou bien à épuiser toutes les branches, auquel cas toute la tactique échoue. Cet aspect pourrait être rapproché du fonctionnement de PROLOG. Cette particularité, que je trouve personnellement séduisante, devrait toutefois être doublée d'un mécanisme de `fail` (tactique pour faire échouer la séquence) plus «ergonomique», qui permettrait de placer des «points d'arrêt» dans les tactiques pour les déboguer (en les faisant échouer à des points précis). En l'état actuel des choses (2006, Coq 8.0), le contrôle des échecs, et donc du *backtracking*, est encore difficile.

On remarque également que la structure de `inversion_fonction` est très voisine de celle de `completer`, ce qui laisse penser que ces deux opérations auraient pu s'effectuer en même temps :

```
(* On a besoin d'un produit dans Type *)
Inductive prod2 (A B : Type) : Type :=
| pair2 : A -> B -> prod2 A B.
Implicit Arguments pair2.

Ltac deux_en_un v f :=
 match constr:f with
  | (comp ?f1 ?f2) =>
     let k1 := deux_en_un v f1 in
     match constr:k1 with
     | pair2 ?w ?g1 => let k2 := deux_en_un w f2 in
     match constr:k2 with
     | pair2 ?x ?g2 => constr:(pair2 x (Compo g1 g2))
    end end
  | (plus_fct ?f1 ?f2) =>
     let k1 := deux_en_un v f1 in
     match constr:k1 with
     | pair2 ?w ?g1 => let k2 := deux_en_un w f2 in
     match constr:k2 with
     | pair2 ?x ?g2 => constr:(pair2 x (Plus g1 g2))
    end end
  | (minus_fct ?f1 ?f2) =>
```



```
     let k1 := deux_en_un v f1 in
     match constr:k1 with
     | pair2 ?w ?g1 => let k2 := deux_en_un w f2 in
     match constr:k2 with
     | pair2 ?x ?g2 => constr:(pair2 x (Moins g1 g2))
    end end
  | (mult_fct ?f1 ?f2) =>
     let k1 := deux_en_un v f1 in
     match constr:k1 with
     | pair2 ?w ?g1 => let k2 := deux_en_un w f2 in
     match constr:k2 with
     | pair2 ?x ?g2 => constr:(pair2 x (Fois g1 g2))
    end end
  | (pow_fct ?f1 ?m) =>
     let k1 := deux_en_un v f1 in
     match constr:k1 with
     | pair2 ?w ?g1 => constr:(pair2 w (Puiss g1 m))
    end
  | (div_fct ?f1 ?f2) =>
     let k1 := deux_en_un v f1 in
     match constr:k1 with
     | pair2 ?w ?g1 => let k2 := deux_en_un w f2 in
     match constr:k2 with
     | pair2 ?x ?g2 => constr:(pair2 x (Sur g1 g2))
    end end
  | (inv_fct ?f1) =>
     let k1 := deux_en_un v f1 in
     match constr:k1 with
     | pair2 ?w ?g1 => constr:(pair2 w (Inverse g1))
    end
  | (opp_fct ?f1) =>
     let k1 := deux_en_un v f1 in
     match constr:k1 with
     | pair2 ?w ?g1 => constr:(pair2 w (Oppose g1))
    end
  | exp => constr:(pair2 v Exp)
  | sin => constr:(pair2 v Sin)
  | cos => constr:(pair2 v Cos)
  | ln => constr:(pair2 v Log)
  | Rabs => constr:(pair2 v Valeur_absolue)
  | sqrt => constr:(pair2 v Racine)
  | (fun x => x) => constr:(pair2 v Id)
  | id => constr:(pair2 v Id)
  | (fun _ => ?x) => constr:(pair2 v (Const x))
  | fct_cte ?x => constr:(pair2 v (Const x))
  | (fun x => ?f (?g x)) =>
let k := constr:(comp f g) in deux_en_un v k (* eta-conversion avec composition *)
  | (fun x => ?f x) => deux_en_un v f (* eta-conversion simple *)
  | _ => (* sous-optimal : tenter d'ajouter f sans doublon puis séparément trouver son index
dans la varmap obtenue *)
let w := ajouter_sans_doublon f v in
let i := trouver f w in constr:(pair2 w (Var i))
end.
```

Certes, grâce à la construction de l'arbre des fonctions non reconnues, on arrive à capturer l'ensemble de toutes les fonctions réelles. Cependant, avec l'indétermination soulevée par l'apparition de valuations supplémentaires coup sur coup, l'automatisation du calcul s'arrête là : le calcul automatique de la dérivée est clairement impossible, puisque cela reviendrait à calculer les dérivées des fonctions ajoutées dans l'arbre, c'est-à-dire les fonctions que *précisément* l'on n'a pas pu reconnaître et réécrire dans notre langage de fonctions, *langage hors* *duquel on ne sait pas dériver.*

Le mieux que l'on puisse faire, c'est d'instancier le théorème de correction de la dérivée avec l'arbre de valuation des fonctions et le terme dans le langage de fonctions, et de le «laisser en plan», c'est-à-dire de laisser à la charge de l'utilisateur le soin de fournir l'arbre de valuation des prédicats de dérivabilité, et l'arbre des dérivées. On voudrait quand même évincer autant que possible `interp` : j'utilise alors `simpl`, qui est pourtant un peu violent car il déplie d'autres définitions, rendant le but moins lisible :

```
Ltac derive v f :=
  let f1 := eta_contraction f in
  let w := completer v f in
  let f0 := inversion_fonction w f1 in (
  generalize (deriv_correcte f0 w) ;
  unfold Moins ; unfold Sur ; simpl interp ; unfold_fct
).
```

Conclusion

Ce projet m'a permis d'approfondir mes connaissances en Coq, sous son aspect de prouveur de théorèmes : j'ai découvert de nombreuses astuces de preuve.

Mais il m'a aussi permis de mieux étudier l'automatisation de manipulation de termes *via* le langage de tactiques $\mathcal{L}_{tac}$.[11]

Il est vrai que $\mathcal{L}_{tac}$ est assez difficile à maîtriser, car son comportement n'est pas bien spécifié, et la documentation est plutôt obscure. Certes, il admet des caractéristiques très intéressantes comme le *backtracking* – de plus, son pouvoir d'expression est très étonnant, par le fait qu'il soit (presque) non typé ; mais également des limitations dans le filtrage (impossibilité d'analyser sous les `fun`), ou des limitations plus conceptuelles comme l'impossibilité d'accéder aux bases de `Hint`. Enfin, l'automatisation de la réécriture n'est pas très satisfaisante en pratique : les tactiques sont trop ou pas assez puissantes selon le contexte, au point qu'il est difficile de cerner le comportement d'un `simpl` ou d'un `auto with real`. On aimerait bien, par exemple, effectuer un pas de β-réduction à un endroit précis du terme considéré, ou bien réécrire une égalité à un endroit précis en spécifiant le moins d'informations possibles (car sinon, cela reviendrait à utiliser `change`, par exemple)...

[11] Ce projet m'a aussi permis de réviser mes théorèmes sur les dérivées...