Using Theorem Proving for Numerical Analysis

Correctness Proof of an Automatic Differentiation Algorithm

Micaela Mayero*

Programming Logic Group
Chalmers University of Technology**

Abstract. In this paper, we present a formal proof, developed in the Coq system, of the correctness of an automatic differentiation algorithm. This is an example of interaction between formal methods and numerical analysis (involving, in particular, real numbers). We study the automatic differentiation tool, called O∂yssée, which deal with FORTRAN programs, and using Coq we formalize the correctness proof of the algorithm used by O∂yssée for a subset of programs. To do so, we describe briefly the library of real numbers in Coq including real analysis, which has been developed originally for this purpose, and we formalize a semantics for a subset of FORTRAN programs. We discuss also about the relevance of such a proof.

1 Introduction

In this work[1], we consider an example of the application of formal methods to prove the correctness of numerical analysis programs and more generally programs dealing with real numbers. The correctness of such programs is often critical, for example we can mention the critical code of the on-board software for planes, trains, real time programs used in medicine, etc. Many of this numerical analysis programs use a notion of differentiation, for example a *gradient*. All these algorithms are well known, but they are also very tedious to implement, because, in particular, a mistake may easily arise, whose detection and identification will be difficult, which means great additional work of test proceedings. Thus, some `automatic differentiation (AD)` tools (like[2] `ADIFOR` [3], `PADRE2` [2], `ADIC` [1], O∂yssée [4]) are used by numericians for the programming of a gradients or another similar differentiation. These tools offer some advantages: speed of development, genericity of the code, and, in particular, the absence of bug which

* mayero@cs.chalmers.se, http://www.cs.chalmers.se/~mayero/.

** Chalmers University of Technology, Department of Computing Science, S-412 96 Göteborg, Sweden.

[1] This work has been realised within the LogiCal project (INRIA-Rocquencourt, France. http://logical.inria.fr/)

[2] A list of automatic differentiation systems has been presented in ICIAM 1995 in Hambourg and is available at http://www-unix.mcs.anl.gov/autodiff/AD_Tools/

would be introduced by the programmer. Regarding this last point, this does not mean that the final program is totally safe because the implementation of the algorithms implemented in those tools can potentially contain some bugs.

Actually, there are two main ways of finding bugs in programs: to make tests or to prove programs. In our case, we prefer the second method which is exhaustive and thus completely secured. One possible solution, to prove formally the correctness of a program, consists in writing the program and proving some properties regarding this code. In the context of Coq [11] this could be done in a functional way using the specification language but also in an imperative way by means of the tactic `Correctness` [5]. Here, we will use naturally the specification language to implement a functional version of the algorithm, which deals with imperative programs but does not require to be coded in an imperative way (moreover, this will be closer to the tool that we propose to study and which provides a functional implementation).

Subsequently, we take a particular interest in O∂yssée, which presents the advantage of having, in addition to an usual differentiation algorithm (direct mode), an inverse mode, which is, in fact, a gradient (even if this work consists only here in the study of the direct mode). With the aim to prove the correctness of the algorithm, we propose using the Coq proof assistance system, in order to prove the commutation of a diagram of figure 1. This diagram does mean the derivative of the semantical interpretation of input program (in FORTRAN) is equal to the semantical interpretation of the output program (in FORTRAN) given by O∂yssée. First, we present O∂yssée and the development which has

$$\begin{tikzcd}
\text{Ass, Seq} \arrow[r, "semantics"] \arrow[d, "O\partial yss\acute{e}e"'] & \mathbb{R} \to \mathbb{R} \arrow[d, "Coq\ Derivative"] \\
\text{Ass, Seq} \arrow[r, "semantics"] & \mathbb{R} \to \mathbb{R}
\end{tikzcd}$$

Fig. 1.

been carried out in Coq regarding real numbers and, more particularly, regarding the derivative. Then, we give a semantics for a subset of FORTRAN programs, a formalization of the differentiation algorithm used by O∂yssée and a correctness proof for this subset. We also discuss, in section 6, on the possibility to deal with a more large set of programs.

2 Presentation of O∂yssée

O∂yssée is a software package for automatic differentiation, developed at INRIA Sophia Antipolis, in the former `SAFIR` project, now the `TROPICS` project.

It implements both derivations: direct mode and inverse mode. It takes input a program written in Fortran-77, computing a differentiable piecewise function, and returns a new program, which computes a tangent or cotangent derivative. The tangent derivative, called direct mode, corresponds to a usual differentiation, and the cotangent derivative, called inverse mode, is a gradient. Here, we will mainly deal with the direct mode, because it is easier to start.

A preliminary step to both modes is the one which to marks the *active variables*. A variable v (or value) is called active:

– if v is an input value of REAL type and if the user specifies it as being an active one
  or
– if v is computed from, at least, one active variable.

In general, the generation of derived code follows, for both modes, the following rule: the structure of the program remains unchanged. In other words, the derived code of a subroutine will be an subroutine, the one of an "if" will be an "if", and so on for the loops and other (imperative) structures which form the imperative programs.

Direct mode

The direct mode may be summarized as follows: let P be a program, X and Y the vectors which respectively represent the input and output variables. Then we have:

$$P(X \to Y) \; \underrightarrow{\textit{direct mode}} \; \bar{P}((X, \bar{X}) \to (Y, \bar{Y}))$$

with $\bar{Y} = J \cdot \bar{X}$ where J represents the Jacobian matrix.

In other words, if the active input variables are v_i, the active output variables are v_o and if P computes $v_o = f(v_i)$, then this mode will compute v_o and $\bar{v}_o$ in v_i in the $\bar{v}_i$ direction , that is to say $\bar{v}_o = f'(v_i) \cdot \bar{v}_i$.

Let us consider the following small example.

```
subroutine cube (x,z)
real y
  y=x*x
  z=y*x
end
```

We will differentiate this routine with respect to `x`. So, `y` and `z` become active variables too. The following command of O∂yssée differentiates `cube` with respect to `x` in the direct mode (`tl`):

```
diff -head cube -vars x -tl -o std
```

What was previously written $\bar{v}$ is now written `VTTL` by O∂yssée. Then we have:

```
COD Compilation unit : cubetl
COD Derivative of unit :  cube
COD Dummys:  x z
COD Active IN   dummys:  x
COD Active OUT  dummys:  z
COD Dependencies between IN and OUT:
COD z <--   x

      SUBROUTINE CUBETL (X, Z, XTTL, ZTTL)
      REAL Y
      REAL YTTL
      YTTL = XTTL*X+X*XTTL
      Y = X*X
      ZTTL = XTTL*Y+X*YTTL
      Z = Y*X
      END
```

For more details, we can refer to [4].

3 Presentation of real analysis in Coq.

As the formalization of that kind of proof requires notions of real analysis, a first step has been to develop a real number library (including basic analysis properties) in Coq. Seeing that our purpose is not a construction of real numbers, the feasibility to formalize and prove the standard analysis properties seems to us, in this case, to be sufficient. So, our entire formalization is based on 17 axioms, which can be proved, if necessary, by a similar construction to that made in HOL [8]. Our axioms express the fact that $\mathbb{R}$ is a commutative, ordered, Archimedian and complete field. From these axioms and some others basic properties, we have defined the notions of limit and derivative. We just present them briefly, seeing that the rest of the proof depends on the choices taken during these formalizations. This development is available in the V6.3 and V7 versions of Coq.

First, the limit is defined in a metric space which is defined in the following way:

```
Record Metric_Space:Type:= {
  Base:Type;
  dist:Base->Base->R;
  dist_pos:(x,y:Base)``(dist x y) >= 0``;
  dist_sym:(x,y:Base)``(dist x y) == (dist y x)``;
  dist_refl:(x,y:Base)``(dist x y) == 0``<->x==y;
  dist_tri:(x,y,z:Base)``(dist x y) <= (dist x z)+(dist z y)``}.
```

The limit definition we used is:

$$\forall \varepsilon > 0 \; \exists \alpha > 0 \; s.t. \; \forall x \in D, \; |\, x - x_0 \,| \; < \alpha \; \rightarrow \; |\, f(x) - l \,| \; < \varepsilon$$

that is to say l is the limit of $f(x)$ when x tends to x_0 in the domain D.

Using the Coq syntax, the limit is expressed as follows:

```
Definition limit_in :=
  [X,X':Metric_Space;f:(Base X)->(Base X');D:(Base X)->Prop;
   x0:(Base X);l:(Base X')]
   (eps:R)``eps > 0`` ->
   (EXT alp:R |``alp > 0``/\(x:(Base X))(D x)/\
              ``(dist X x x0) < alp``->
              ``(dist X' (f x) l) < eps``).
```

Once we have shown that $\mathbb{R}$ is really a metric space, the limit for the functions of type $\mathbb{R} \rightarrow \mathbb{R}$ can be defined in the following way:

```
Definition limit1_in:(R->R)->(R->Prop)->R->R->Prop:=
  [f:R->R; D:R->Prop; l,x0:R](limit_in R_met R_met f D x0 l).
```

Before formalizing the derivative, it is interesting to note that this definition of the limit does not absolutely prescribe $x \neq x_0$. Hence, in order to get a correct definition of the derivative, we must take this first choice[3] into consideration. To formalize the derivative, we chose the following definition:

$$f'(x_0) = \lim_{x \rightarrow x_0, x \neq x_0} \frac{f(x) - f(x_0)}{x - x_0}$$

Considering our definition of limit, we must express that $x \neq x_0$ using the definition domain of the function. The limit which will allow us to build the derivative uses the following notion of domain:

```
Definition D_x: (R->Prop)->R->R->Prop :=
   [D:(R->Prop); y,x:R](D x)/\``y <> x``.
```

In particular, x∈ (D_x D x0) means x∈ D\x0.

Now, we can give the definition of derivative of a function of type $\mathbb{R} \rightarrow \mathbb{R}$. Here as for the limit, we do not compute the derivative d of f, but we show that d is really the derivative of f in x_0 in a domain D.

```
Definition D_in:(R->R)->(R->R)->(R->Prop)->R->Prop:=
  [f,d:(R->R); D:(R->Prop); x0:R]
  (limit1_in [x:R]``((f x)-(f x0))/(x-x0)`` (D_x D x0) (d x0) x0).
```

This definition allows us to show the basic properties which will be useful to prove our correctness lemma, namely, properties about the derivative of sum, product, composition of functions, etc. Some of these properties can be found in appendix A.

For more details regarding the real library of Coq, we can refer to [9, 11].

[3] This choice allow us to prove some lemmas of standard analysis more easily.

4 Formalization of the algorithm in direct mode

In this part, we give a formalization of the algorithm in direct mode for a subset of FORTRAN programs that we will prove its correctness. We consider the issues raised, some of which will be answered during the discussion in section 6.

4.1 A semantic for FORTRAN

Our semantics kernel will have, among other aspects, an environment, an abstract syntax of a subset of FORTRAN expressions, the interpretation of this abstract syntax in Coq, and an interpretation of the differentiation algorithm.

Environment
In this example, our environment is an associated list between the real type variables, indexed by natural numbers, and their associated value. Therefore we have:

```
Inductive rvar:Type:=varreal:nat->rvar.
Definition Env:=(listT (prodT rvar R)).
```

where `listT` and `prodT` represent respectively the type of lists and the type of couples.

Expressions
Now we define the type of the arithmetic expressions, followed by the type of some imperative structures. The arithmetic expressions will include constants, real variables and the usual operators of addition, multiplication, minus, power. In this example, we do not consider the transcendental functions, because their implementation is still incomplete in Coq. For our example, the minimal type of the arithmetic expressions is simply expressed as follows:

```
Inductive expr:Type:=
          numR:R->expr
         |var:rvar->expr
         |Fplus:expr->expr->expr
         |Fmult:expr->expr->expr
         |Fminus:expr->expr->expr
         |Fopp:expr->expr
         |Fpow:expr->nat->expr.
```

With regard to the structures of the FORTRAN language, we only consider here the assignment and the sequence. We will explain this choice during the discussion.

```
Inductive For:Type:= Aff:rvar->expr->For
                    |Seq:For->For->For.
```

Remark 1. It is interesting to note that, even if our abstract syntax represents that of FORTRAN (to deal with O∂yssée programs), it also represents every subset of imperative programs and that correctness proof could be applied to other automatic differentiation tools (which deal with programs of another imperative language).

The semantics itself

We chose to give a natural semantics [7], that is to say big step semantics, which interprets our language towards some functions of the $\mathbb{R} \to \mathbb{R}$ type. This semantics is quite adapted because there is no recursive functions involved. This choice allows our definitions and derivative properties in Coq to be used almost immediately. First, we give a semantics to the variables, the arithmetic expressions, then to our mini language. The evaluation of a variable will be its associated value in the environment. The function `myassoc`, a function written in Coq using a `Fixpoint`, looks for the value associated to a given variable in the environment.

```
Definition sem_var [v:rvar;l:Env]:R:=(myassoc v l).
```

The evaluation of the mathematical expressions is as usual:

```
Fixpoint sem_arith [l:Env;arith:expr]:R:=
  Cases arith of
     (numR x)=> x
    |(var r) =>  (sem_var r l)
    |(Fplus e1 e2)=>``(sem_arith l e1)+(sem_arith l e2)``
    |(Fmult e1 e2)=>``(sem_arith l e1)*(sem_arith l e2)``
    |(Fminus e1 e2)=>``(sem_arith l e1)-(sem_arith l e2)``
    |(Fopp e)=>``-(sem_arith l e)``
    |(Fpow e i)=>(pow (sem_arith l e) i)
  end.
```

The interpretation of assignments and sequences modifies the environment. Regarding the assignment, we create a new assignment with the interpretation of the right part and regarding the sequence `f1;f2` we give an interpretation of `f2` in the environment which has been modified by the interpretation of `f1`:

```
Fixpoint sem_For [l:Env;for:For]:Env:=
  Cases for of
   (Aff v e)=>(replace v (sem_arith l e) l)
  |(Seq f1 f2) => (sem_For (sem_For l f1) f2)
  end.
```

where `replace` is a function which replaces or assigns a value associated to a variable of the environment by another one, which makes the successive assignments possible.

Now, we need an interpretation towards a real function. For this, we can, for example, recover the value associated to an output variable after the evaluation of the program:

```
Definition sem [for:For;l:Env;vi,vo:rvar]:R->R:=
 [r:R](myassoc vo (sem_For (consT (pairT rvar R vi r) l) for)).
```

where `vi` and `vo` are respectively the significant input variable (given active by the user) and the output variable.

4.2 Derivation algorithm

Fisrt of all, to help us to understand the algorithm, we start giving a mathematical transcription of the previous example (`cube`) of section 2.

The instruction `y=x*x` is mathematically defined by the fonction f:
$f : \mathbb{R}^3 \to \mathbb{R}^3$
$f : (x, y, z) \mapsto (f_1(x, y, z), f_2(x, y, z), f_3(x, y, z)) = (x, x \times x, z)$

The instruction `z=y*x` is defined by the fonction g:
$g : \mathbb{R}^3 \to \mathbb{R}^3$
$g : (x, y, z) \mapsto (g_1(x, y, z), g_2(x, y, z), g_3(x, y, z)) = (x, y, y \times x)$

The sequence is the *composition* of this two functions: $g \circ f$. The differentiation of $g \circ f$ in x_0 is $D_{x_0}(g \circ f) = D_{f(x_0)}g \circ D_{x_0}f$. We have the tangente derivative applaying this linear function from $\mathbb{R}^3$ to $\mathbb{R}^3$ with the vector $(\dot{x}, \dot{y}, \dot{z})$ given by the user. To do so, we begin computing the Jacobi matrices of f in (x, y, z) and g in $f(x, y, z)$:

$$J_f(x, y, z) = \begin{pmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} & \frac{\partial f_1}{\partial z} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} & \frac{\partial f_2}{\partial z} \\ \frac{\partial f_3}{\partial x} & \frac{\partial f_3}{\partial y} & \frac{\partial f_3}{\partial z} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2x & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

and

$$J_g(x, y, z) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ y & x & 0 \end{pmatrix}$$

Thus,

$$J_g(x, x^2, z) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x^2 & x & 0 \end{pmatrix}$$

The Jacobi matrice $g \circ f$ is the following:

$$J_{g \circ f}(x, y, z) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x^2 & x & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 2x & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2x & 0 & 0 \\ x^2 + 2x^2 & 0 & 0 \end{pmatrix}$$

Making the projection on $(\dot{x}, 0, 0)$, which is the active vector at the begining (noted `IN` by O∂yssée):

$$\begin{pmatrix} 1 & 0 & 0 \\ 2x & 0 & 0 \\ 3x^2 & 0 & 0 \end{pmatrix} \begin{pmatrix} \dot{x} \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \dot{x} \\ 2x\dot{x} \\ 3x^2\dot{x} \end{pmatrix}$$

We can now check that this corresponds to the O∂yssée program (CUBETL): YTTL is exactly the result given by our mathematical formalization.
Regarding ZTTL we have:
ZTTL=XTTL*Y+X*YTTL=XTTL*X*X+X*(XTTL*X+X*XTTL)=3*X*X*XTTL
which is also the expected result.

We have seen that the direct mode is close to a symbolic derivative. In particular, in O∂yssée, to derive a real mathematical expression consists in calling a function, written in Ocalm, which makes this "symbolic" derivative. Therefore, we have a similar function in Coq, but which only contains what corresponds to our language. In order to add the transcendental functions, for example, adding the corresponding derivation rules is sufficient. Moreover, although the notion of derivative is enough to deal with our subset of programs, sometimes we will use the notion of differention. In particular, it is the case for the variables. If we only derive variables, the sequence of instructions are independent and this solution does not give the expected result. To formalize the O∂yssée algorithm, we must define a association list $(var, dvar)$ of type lvar (which will be built later):

```
Definition lvar:=(listT (prodT rvar rvar)).
```

The derivative of an expression is another expression which has been symbolically derivated. The derivative of a variable is its corresponding derivated variable in the list l of type lvar, given by the myassocV function. The following Coq function gives the derivative of our expressions:

```
Fixpoint deriv_expr [l:lvar;term:expr]:expr:=
  Cases term of
    (numR r)=>(numR R0)
   |(var x) => (var (myassocV x l))
   |(Fplus a1 a2)=>
     (Fplus (deriv_expr l a1) (deriv_expr l a2))
   |(Fmult a1 a2)=>
     (Fplus (Fmult (deriv_expr l a1) a2)
            (Fmult a1 (deriv_expr l a2)))
   |(Fminus a1 a2)=>
     (Fminus (deriv_expr l a1) (deriv_expr l a2))
   |(Fopp a)=>(Fopp (deriv_expr l a))
   |(Fpow a i)=>
      (Fmult (Fmult (numR (INR i)) (Fpow a (minus i (S O))))
             (deriv_expr l a))
  end.
```

As can be seen previously, the derivative algorithm of our FORTRAN subset can be summarized as follow:

– the derivative of an assignment is a sequence between the assignment to a new variable of the right-hand side expression derivative and the initial assignment

– the derivative of a sequence is the sequence of the expression derivatives of the initial sequence

In Coq, the O∂yssée differentiation algorithm can be formalized as follows:

```
Fixpoint deriv_for_aux [l:lvar; termfor:For]:For:=
  Cases termfor of
    (Aff v e) =>
      (Seq (Aff (myassocV v l) (deriv_expr l e)) (Aff v e))
   |(Seq f1 f2) => (Seq (deriv_for_aux l f1)
                        (deriv_for_aux l f2))
  end.

Definition deriv_for [termfor:For]:For:=
       (deriv_for_aux (make_list termfor) termfor).
```

where `make_list` builds the association list $(var, dvar)$.

An example of application of this function can be found in [9] (chapter 5).

5 Proof of correctness

Now, we can give and prove the correctness lemma regarding the algorithm used by O∂yssée to differentiate FORTRAN programs involving sequences and assignments. As said previously, this is equivalent to show that the interpretation in Coq of such a program is derivated (in Coq) into the interpretation (in Coq) of the program differentiated by O∂yssée (which is expressed by the diagram of figure 1).

Before giving the unformal (mathematical) statement and the formal version (Coq), let us remind that the semantical interpretation depends on the variable which is chosen as active, as well as the variable which is chosen as output value. Those two variables must be in the environment (which is a list of variables with their associated values). This method allows us to deal only with functions of type $\mathbb{R} \to \mathbb{R}$, which can be derived in Coq. Thus, the theorem can expressed as follows:

Theorem 1 (Correctness). *Given a function sem, the semantics of a* FORTRAN *program involving sequences and affectations. Given v_i, the input active variable, and v_o, the output variable. Given env, the environment.*
For every program p, if $v_i \in env$ and $v_o \in env$ then $sem'(p\ env\ v_i\ v_o) = (sem\ \bar{p}\ env\ v_i\ v_o)$ where $\bar{p}$ is the differentiated program returned by O∂yssée.

In Coq, this theorem is expressed in the following way:

```
Theorem corr_deriv:(l:Env)(p:For)(vi,vo:rvar)(D:R->Prop)(x0:R)
    (memenv vi l)->(memenv vo l)->
    (D_in (sem p l vi vo) (sem (deriv_for p) l vi vo) D x0).
```

To prove this theorem, we start proving two similar auxiliary lemmas regarding the affectation and the assignment:

```
Lemma corr_deriv_Aff:(l:Env)(e:expr)(r,vi,vo:rvar)(D:R->Prop)(x0:R)
  (memenv vi l)->(memenv vo l)->
  (D_in (sem (Aff r e) l vi vo)
     (sem (deriv_for (Aff r e)) l vi vo) D x0).

lemma corr_deriv_Seq:(l:Env)(f,f0:For)(vi,vo:rvar)(D:R->Prop)(x0:R)
       (memenv vi l)->(memenv vo l)->
  (D_in (sem (Seq f f0) l vi vo)
     (sem (deriv_for (Seq f f0)) l vi vo) D x0).
```

In the following proofs, the assignment will be noted ":=" and the sequence ";". These proofs are described in such a way that they are rather close to the formal proofs built in Coq. The specific lemmas regarding derivative in Coq used in these proofs, can be found in appendix A.

Proof. of `corr_deriv_Aff`
By induction on the expression e.
Let H0 and H1 be the hypotheses $v_i \in l$ and $v_o \in l$.

1. Constant case: we must prove, using the previous hypotheses, that:
   $\forall c : \mathbb{R}, sem'((r := c)\ l\ v_i\ v_o) = (sem\ \overline{(r := c)}\ l\ v_i\ v_o)$ i.e. that
   $\forall c : \mathbb{R}, sem'((r := c)\ l\ v_i\ v_o) = (sem\ (dr := 0; r := c)\ l\ v_i\ v_o)$ where c is a real constant.
   In the case of $v_i = r$ and $v_o = dr$, it is equivalent to prove, after simplifications and using H0, H1, that $c' = 0$. We only have to apply the lemma `Dconst` of Coq.
   In the case of $v_i \neq r$ or $v_o \neq dr$, we have a contradiction with H0 or H1, because in these cases $v_i \notin l$ or $v_o \notin l$.
2. Variable case: we must prove, using the previous hypotheses, that:
   $\forall c : \mathbb{R}, sem'((r := x)\ l\ v_i\ v_o) = (sem\ \overline{(r := x)}\ l\ v_i\ v_o)$ i.e. that
   $\forall c : \mathbb{R}, sem'((r := x)\ l\ v_i\ v_o) = (sem\ (dr := 1; r := x)\ l\ v_i\ v_o)$ where x is a real variable.
   In the case of $v_i = r$ and $v_o = dr$, it is equivalent to prove, after simplifications and using H0, H1, that $x' = 1$. We only have to apply the lemma `Dx` of Coq.
   In the case of $v_i \neq r$ or $v_o \neq dr$, we have a contradiction with H0 or H1, because in these cases $v_i \notin l$ or $v_o \notin l$. $v_o \notin l$.
3. Addition case: we have also two induction hypotheses:
   - H2: $\forall v_i, v_o, r, sem'((r := e_0)\ l\ v_i\ v_o) = (sem\ \overline{(r := e_0)}\ l\ v_i\ v_o)$ where e_0 is an expression
   - H3: $\forall v_i, v_o, r, sem'((r := e_1)\ l\ v_i\ v_o) = (sem\ \overline{(r := e_1)}\ l\ v_i\ v_o)$ where e_1 is an expression
   We must prove that
   $sem'((r := e_0 + e_1)\ l\ v_i\ v_o) = (sem\ \overline{(r := e_0 + e_1)}\ l\ v_i\ v_o)$ i.e. that

$sem'((r := e_0 + e_1)\ l\ v_i\ v_o) = (sem\ (dr := e'_0 + e'_1; r := e_0 + e_1)\ l\ v_i\ v_o)$.
After simplifications of H2 and H3 we have:
- H2': $\forall v_i, v_o, r, sem'((r := e_0)\ l\ v_i\ v_o) = (sem\ (dr := e'_0; r := e_0)\ l\ v_i\ v_o)$
- H3': $\forall v_i, v_o, r, sem'((r := e_1)\ l\ v_i\ v_o) = (sem\ (dr := e'_1; r := e_1)\ l\ v_i\ v_o)$

In the case of $v_i = r$ and $v_o = dr$, it is equivalent to prove, after simplifications and using H0, H1, H2', H3', that $(e_0 + e_1)' = e'_0 + e'_1$. We only have to apply the lemma `Dadd` of `Coq`.
In the case of $v_i \neq r$ or $v_o \neq dr$, we have a contradiction with H0 or H1, because in these cases $v_i \notin l$ or $v_o \notin l$.

4. Multiplication case: similar to the addition case.
5. Minus case: similar to the addition case.
6. Opposite case: we have also an induction hypothesis:
   - H2: $\forall v_i, v_o, r, sem'((r := e_0)\ l\ v_i\ v_o) = (sem\ \overline{(r := e_0)}\ l\ v_i\ v_o)$ where e_0 is an expression

   We must prove that $sem'((r := -e_0)\ l\ v_i\ v_o) = (sem\ \overline{(r := -e_0)}\ l\ v_i\ v_o)$ i.e. that $sem'((r := -e_0)\ l\ v_i\ v_o) = (sem\ (dr := -e'_0; r := -e_0)\ l\ v_i\ v_o)$.
   After simplifications of H2 we have:
   - H2': $\forall v_i, v_o, r, sem'((r := e_0)\ l\ v_i\ v_o) = (sem\ (dr := e'_0; r := e_0)\ l\ v_i\ v_o)$

   In the case of $v_i = r$ and $v_o = dr$, it is equivalent to prove, after simplifications and using H0, H1, H2', that $(-e_0)' = -e'_0$. We only have to apply the lemma `Dopp` of `Coq`.
   In the case of $v_i \neq r$ or $v_o \neq dr$, we have a contradiction with H0 or H1, because in these cases $v_i \notin l$ or $v_o \notin l$.
7. Power case: similar to the opposite case, using the composition and power lemmas.

Proof. of `corr_deriv_Seq`
Double induction on the programs f and f_0.
Let H0 and H1 be the hypotheses $v_i \in l$ and $v_o \in l$.

1. Assigment-assigment case: we must prove, using the previous hypotheses, that $sem'((r_0 := e_0; r := e)\ l\ v_i\ v_o) = (sem\ \overline{(r_0 := e_0; r := e)}\ l\ v_i\ v_o)$ i.e. that
   $sem'((r_0 := e_0; r := e)\ l\ v_i\ v_o) = (sem\ ((dr_0 := e'_0; r_0 := e_0); (dr := e'; r := e))\ l\ v_i\ v_o)$ where e_0 and e are expressions.
   In the case of $v_i = r$ and $v_o = dr$ or $v_i = r_0$ and $v_o = dr_0$ or $v_i = r$ and $v_o = dr_0$ or $v_i = r_0$ and $v_o = dr$, after some substitutions from the environment and some simplifications, we can apply the previous lemma.
   In the other cases, we have either $v_i \notin l$ or $v_o \notin l$, which is a contradiction with respect to the hypotheses.
2. Assigment-sequence case: we have also two induction hypotheses:
   - H2: $\forall e, v_i, v_o, r, sem'((r := e; f_1)\ l\ v_i\ v_o) = (sem\ \overline{(r := e; f_1)}\ l\ v_i\ v_o)$ where f_1 is a program
   - H3: $\forall e, v_i, v_o, r, sem'((r := e; f_2)\ l\ v_i\ v_o) = (sem\ \overline{(r := e; f_2)}\ l\ v_i\ v_o)$ where f_2 is a program

   we must prove, using the previous hypotheses, that
   $sem'((r := e; (f_1; f_2))\ l\ v_i\ v_o) = (sem\ \overline{(r := e; (f_1; f_2))}\ l\ v_i\ v_o)$ i.e. that

$sem'((r := e;(f_1;f_2))\ l\ v_i\ v_o) = (sem\ ((dr := e';r := e);\overline{(f_1;f_2)})\ l\ v_i\ v_o)$
where f_1 and f_2 are programs.
After simplifications of H2 and H3 we have:
- H2': $\forall e, v_i, v_o, r, sem'((r := e; f_1)\ l\ v_i\ v_o) = (sem\ (dr := e'; r := e; \overline{f_1})\ l\ v_i\ v_o)$
- H3': $\forall e, v_i, v_o, r, sem'((r := e; f_2)\ l\ v_i\ v_o) = (sem\ (dr := e'; r := e; \overline{f_2})\ l\ v_i\ v_o)$
Using H2' et H3', we are in the previous case.

3. Sequence-assigment case: symmetric w.r.t. the previous case.
4. Sequence-sequence case: we have 4 induction hypotheses:
   - H2: $\forall v_i, v_o, sem'(((f_4;f_3);f_0)\ l\ v_i\ v_o) = (sem\ \overline{((f_4;f_3);f_0)}\ l\ v_i\ v_o)$ where f_4, f_3 and f_0 are programs
   - H3: $\forall v_i, v_o, sem'(((f_4;f_3);f_1)\ l\ v_i\ mbv_o) = (sem\ \overline{((f_4;f_3);f_1)}\ l\ v_i\ v_o)$ where f_4, f_3 and f_1 are programs
   - H4: si $\forall f_1, sem'((f_1;f_0))\ l\ v_i\ v_o) = (sem\ \overline{(f_1;f_0)}\ l\ v_i\ v_o)$ et si $\forall f_2, sem'((f_2;f_0))\ l\ v_i\ v_o) = (sem\ \overline{(f_2;f_0)}\ l\ v_i\ v_o)$ alors $\forall v_i, v_o, sem'(((f_2;f_1);f_0))\ l\ v_i\ v_o) = (sem\ \overline{((f_2;f_1);f_0)}\ l\ v_i\ v_o)$
   - H5: si $\forall f_2, sem'((f_2;f_0))\ l\ v_i\ v_o) = (sem\ \overline{(f_2;f_0)}\ l\ v_i\ v_o)$ et si $\forall f_0, sem'((f_0;f_1))\ l\ v_i\ v_o) = (sem\ \overline{(f_0;f_1)}\ l\ v_i\ v_o)$ alors $\forall v_i, v_o, sem'(((f_0;f_2);f_1))\ l\ v_i\ v_o) = (sem\ \overline{((f_0;f_2);f_1)}\ l\ v_i\ v_o)$

   Applying H4 and H5 with f_3 H2 f_4 H3, we have these two following hypotheses:
   - H4': $\forall v_i, v_o, sem'(((f_4;f_3);f_0))\ l\ v_i\ v_o) = (sem\ \overline{((f_4;f_3);f_0)}\ l\ v_i\ v_o)$
   - H5': $\forall v_i, v_o, sem'(((f_4;f_3);f_1))\ l\ v_i\ v_o) = (sem\ \overline{((f_4;f_3);f_1)}\ l\ v_i\ v_o)$

   Thus, we must prove that
   $\forall v_i, v_o, sem'(((f_4;f_3);(f_0;f_1))\ l\ v_i\ v_o) = (sem\ \overline{((f_4;f_3);(f_0;f_1))\ l\ v_i}\ v_o)$ i.e. that $\forall v_i, v_o, sem'(((f_4;f_3);(f_0;f_1))\ l\ v_i\ v_o) = (sem\ \overline{((f_4;f_3)};\overline{(f_0;f_1)}\ l\ v_i\ v_o)$, and using H4' and H5' we obtain the previous cases.

Now, the main theorem can be directly proved:

Proof. of theorem 1
By cases on the program p.

1. Assignment case: we apply directly the lemma `corr_deriv_Aff`.
2. Sequence case: we apply directly the lemma `corr_deriv_Seq`.

6 Discussion

In the previous section, we have shown the correctness of programs composed of variables, real arithmetic expressions, assignments and sequences, in order words straight line real programs. Now, we can wonder if we can prove the correctness of the whole algorithm, that is to say for every program. There are several difficulties:

What's happen with control structures? let us study the case of the conditional. This is an interesting case, as it introduces a notion of discontinuity. The correctness of the algorithm is not ensured near the discontinuity points. In particular, the correctness cannot be proved for programs which compute functions by gaps, or functions of Dirac[4]. On the other hand, it is possible, if we are able to choose our branch (either the `then` or the `else`), to show the correctness. This remark essentially expresses the fact that the correctness cannot be proved for all imperative programs, which is represented by the following diagram[5] 2. We will denote S_i the sequence number i, p a program and $\bar{A}$ the result given by O∂yssée.

The conditional is not the only cause of discontinuity in the programs. The

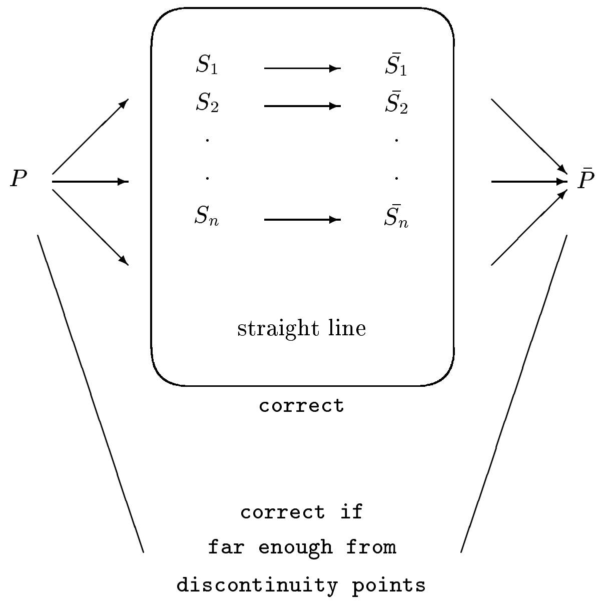

Fig. 2.

`goto, dowhile` are potential causes too. This problem could be solved either giving a straight line program or defining a valid space for the program.

What's happen with several active variables? We have only dealt with programs having a functional interpretation which corresponds to the diagram 1.

Actually, the semantics we gave is a functional interpretation ($\mathbb{R} \rightarrow \mathbb{R}$). From the definition of our current derivative, we cannot give give an interpretation towards multidimensional spaces like $\mathbb{R}^n$. In order to deal with these cases, it would be necessary to begin by extending our development of the real analysis by a notion of differentiation and of Jacobian matrix. This will have to be done if we want to extend this experience to the computation of the gradient.

[4] Let us note that the differentiated programs expressed by O∂yssée are not absurd in these cases, since they return zero values almost everywhere

[5] due to Laurent Hascoët INRIA-SOPHIA-antipolis, TROPICS project

What's happen with others data types? Currently, we do not know if it is possible to find a general way to proceed[6]. For example, another problem, which we do not deal on here, (it is not considered in O∂yssée either), is a code in which there is an array of variable size. A notion of derivative must be found for that kind of object.
Moreover, the other basic data types (integer or boolean) of FORTRAN can be also considered by the program semantics. To do so, we have to define specific types of variables for each data type. This more complete semantics has been also formalized in Coq.

7 Conclusion

The application of formal methods to numerical analysis problems is quite new. The formalization and the correctness proof for our subset of programs derivated by the O∂yssée system has shown that formal proof systems can be appropriate to deal with that kind of problems. This proof could have been also built in some other proof assistants (like HOL[6] or PVS[10]), as well as for some other automatic differentiation tools for imperative programs. The use of a proof assistance tool allows us to isolate the different cases which must be considered in order to prove the correctness of such algorithms. This makes it possible to emphasize the particular case of programs which compute discontinuous functions.
However, although we are convinced that formal methods are suitable for such a critical domain, we have noticed that we are still rather far from dealing with significant problems and there are two main explanations. The first one is that the real analysis formalization is not immediate and that kind of library is not developed in every system. The second one is due to numerical analysis itself. Most of the programs written for numerical analysis use approximation principles, which are controlled or not. There is a kind of conventional *probabilist tolerance*, in the sense that, as can be seen in figure 2 for instance, problems arise rarely or even never. This view is obviously opposed to our motivations, which are to prove formally that no problem can arise. In this particular case we have studied, we can see that the problem of discontinuity points is exactly an example of what we would like to avoid.

Regarding the application of formal proofs to numerical analysis domain, the research activity is still quite open. However, this formalization, on a subset of imperative straight line programs, is a good start for a more advanced formalization, which would take into consideration more the differentiation than the derivative. A next step could be to study algorithms for gradient computation, which are considered as critical in their implementation but which might be dealt using our methods.

[6] By the way, automatic differentiation tools consider only some of them.

References

1. The ADIC Home Page. http://www-fp.mcs.anl.gov/adic/.
2. The PADRE2 Home Page. http://warbler.ise.chuo-u.ac.jp/Padre2/.
3. C. Bischof, A. Carle, P. Khademi, A. Mauer, and P. Hovland. ADIFOR2.0 user's guide. Technical Report ANL/MCS-TM-192,CRPC-TR95516-S, Argonne National Laboratory Technical Memorandum, June 1998.
4. Christèle Faure and Yves Papegay. Odyssée user's guide version 1.7. Technical Report 0224, INRIA Sophia-Antipolis, September 1998.
5. Jean-Christophe Filliâtre. *Preuves de programmes impératifs en théories des types.* PhD thesis, Université Paris-Sud, Juillet 1999.
6. M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.
7. Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press, 1992.
8. John Harrison. *Theorem Proving with the Real Numbers.* Springer-Verlag, 1998.
9. Micaela Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique.* PhD thesis, Université Paris 6, Décembre 2001. http://logical.inria.fr/~mayero/.
10. Sam Owre, Natarajan Shankar, and John Rushby. PVS: A prototype verification system. In *Proceedings of CADE 11, Saratoga Springs, New York*, June 1992.
11. LogiCal Project The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 7.2.* INRIA-Rocquencourt, December 2001. http://coq.inria.fr/doc-eng.html.

A Main derivative properties (in Coq) used in the proofs

```
Lemma Dconst:(D:R->Prop)(y:R)(x0:R)(D_in [x:R]y [x:R]``0`` D x0).
Lemma Dx:(D:R->Prop)(x0:R)(D_in [x:R]x [x:R]``1`` D x0).
Lemma Dadd:(D:R->Prop)(df,dg:R->R)(f,g:R->R)(x0:R)
  (D_in f df D x0)->(D_in g dg D x0)->
  (D_in [x:R]``(f x)+(g x)`` [x:R]``(df x)+(dg x)`` D x0).
Lemma Dmult:(D:R->Prop)(df,dg:R->R)(f,g:R->R)(x0:R)
  (D_in f df D x0)->(D_in g dg D x0)->
  (D_in [x:R]``(f x)*(g x)``
      [x:R]``(df x)*(g x)+(f x)*(dg x)`` D x0).
Lemma Dopp:(D:R->Prop)(f,df:R->R)(x0:R)(D_in f df D x0)->
  (D_in [x:R](Ropp (f x)) [x:R]``-(df x)`` D x0).
Lemma Dminus:(D:R->Prop)(df,dg:R->R)(f,g:R->R)(x0:R)
  (D_in f df D x0)->(D_in g dg D x0)->
  (D_in [x:R](Rminus (f x) (g x)) [x:R]``(df x)-(dg x)`` D x0).
Lemma Dcomp:(Df,Dg:R->Prop)(df,dg:R->R)(f,g:R->R)(x0:R)
  (D_in f df Df x0)->(D_in g dg Dg (f x0))->
  (D_in [x:R](g (f x)) [x:R]``(df x)*(dg (f x))`` (Dgf Df Dg f) x0).
Lemma D_pow_n:(n:nat)(D:R->Prop)(x0:R)(expr,dexpr:R->R)
 (D_in expr dexpr D x0)-> (D_in [x:R](pow (expr x) n)
   [x:R]``(INR n)*(pow (expr x) (minus n (S O)))*(dexpr x)``
        (Dgf D D expr) x0).
```