Computer Science • 15 (2) 2014 http://dx.doi.org/10.7494/csci.2014.15.2.215

Emmanuel M. Tadjouddine
Wenjin Lv

FOUNDATIONAL CERTIFICATION OF CODE TRANSFORMATIONS USING AUTOMATIC DIFFERENTIATION

Abstract *Automatic Differentiation (AD) is concerned with the semantics augmentation of an input program representing a function to form a transformed program that computes the function's derivatives. To ensure the correctness of the AD transformed code (particularly for safety-critical applications), we aim at certifying the algebraic manipulations at the heart of the AD process. We have considered a WHILE-language, and have shown how such proofs can be constructed by using appropriate relational Hoare logic. In particular, we have shown how such inference rules can be constructed for both the forward- and reverse-mode AD by using an abductive logical reasoning.*

Keywords Certification, relational Hoare logic, abductive reasoning, Automatic Differentiation

1. Introduction

Automatic Differentiation (AD) [16] is now standard technology for computing derivatives of a (vector) function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ defined by a computer code. Such derivatives may be used as sensitivities with respect to design parameters, Jacobians for use in Newton-like iterations or in optimization algorithms, or coefficients for Taylor-series generation. Compared to the numerical finite differencing scheme, AD is accurate for machine precision and presents opportunities for efficient derivative computation. There is already a large body of literature on the use of AD in solving engineering problems.

However, the application of AD on large-scale applications is not straightforward for at least the following reasons:

- AD relies on the assumption that the input code is piecewise differentiable.
- Prior to AD, certain language constructs may need be rewritten or the input code be massaged for the specifics of the AD tool, see for example [28].
- The input code may contain non-differentiable functions; e.g., `abs` or functions such as `sqrt` whose derivative values may overflow for very small numbers [27].

In principle, AD preserves the semantics of the input code provided that has not been altered prior to AD transformation. Given this semi-automatic usage of AD, can we trust AD for safety-critical applications?

Although the chain rule of calculus and the analyses used in AD have been proven to be correct, the correctness of AD-generated code is tricky to establish. First, AD may locally replace some part B of the input code by B' that is not observationally equivalent to B, even though both are semantically equivalent in that particular context. Second, the input code may not be piecewise differentiable in contrast to the AD assumption. Finally, AD may use certain common optimizing transformations used in compiler-construction technology and for which formal proofs are not straightforward [4, 20]. To ensure trust in the AD process, we propose shifting the burden of proof from the AD client to the AD producer by using the proof-carrying code paradigm [22]: an AD tool must provide proof for the correctness of an AD-generated code or a counter-example demonstrating, for example, that the input code is not piecewise differentiable; an AD user can check correctness proof using a simple checker. Note that in this work, it is not our intention to certify real arithmetic, but to certify the symbolic manipulations carried out by the AD process. In this perspective, we have shown that, at least in some simple cases, one can establish the correctness of a mechanical AD transformation (involving mainly algebraic manipulations) used to that end by using a variant of Hoare logic [18, Chap. 4]. For that purpose, we have constructed inference rules based on relational Hoare logic [4] to establish the correctness of the forward-mode AD. We have also investigated an abductive approach [25, 19, 5, 8], aiming at finding preconditions given post-conditions for the correctness of the reverse-mode AD. Besides that, we aim to put forward a viewpoint that distinguishes between performance and correctness (or

safety) aspects of AD transformations; correctness aspects have yet to be explored in AD literature.

2. Background and problem statement

This section gives a background on automatic differentiation, proof-carrying code, and states the problem of certifying AD transformations.

2.1. Automatic Differentiation

AD is a semantics-augmentation framework based on the idea that a source program S representing $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m, \mathbf{x} \mapsto \mathbf{y}$ can be viewed as a sequence of instructions; each representing a function ϕ_i that has a continuous first derivative. This assumes the program S is piecewise differentiable; therefore, we can conceptually fix the program's control flow to view S as a sequence of q assignments. An assignment $v_i = \phi_i\big(\{v_j\}_{j \prec i}\big), \quad i = 1, \ldots, q$ wherein $j \prec i$ means v_i depends on v_j, computes the value of a variable v_i in terms of previously defined v_j. Thus, S represents a composition of functions

$$\phi_q \circ \phi_{q-1} \circ \ldots \circ \phi_2 \circ \phi_1 \tag{1}$$

Differentiating $\mathbf{f}$ yields the following chain of matrix multiplications that compute the derivative of the function f represented by the program S.

$$\mathbf{f}'(\mathbf{x}) = \phi'_q(\{v_j\}_{j \prec q-1}) \cdot \phi'_{q-1}(\{v_j\}_{j \prec q-2}) \cdot \ldots \cdot \phi'_1\big(\mathbf{x}\big) \tag{2}$$

The variables $\mathbf{x}, \mathbf{y}$ are called *independents, dependents* respectively. A variable that depends on an independent and influences a dependent is called *active*.

2.1.1. A simple AD example

The calculation of $\mathbf{f}$ will be described by a code list [16], equivalent to static single-assignment form [10]. It is a sequence of equations

$$v_i = \varphi_i(\text{relevant previous } v_j), \tag{3}$$

for $i = 1, \ldots, p + m$. The φ_i are given elementary functions and "relevant previous v_j" denotes those variables v_j that are the actual arguments of φ_i – necessarily all having $j < i$. Here, $v_{1-n}, \ldots, v_0$ are aliases for $\mathbf{f}$'s *input variables* $x_1, \ldots, x_n$, while $v_{p+1}, \ldots, v_{p+m}$ are aliases for $\mathbf{f}$'s *output variables* $y_1, \ldots, y_m$, and $v_1, \ldots, v_p$ are *intermediate variables*. That is, there are n inputs, p intermediates, and m outputs.

A code list describes the values calculated by a single execution-trace through the program code of $\mathbf{f}$. To illustrate this, let us consider the function $\mathbf{f} : \mathbb{R}^2 \to \mathbb{R}^2, (x_1, x_2) \mapsto (y_1 = (\sin(x_2) - x_1)x_1, y_2 = \sqrt{\sin(x_2)})$. The left column of the table below shows a code list for $\mathbf{f}$, written in MATLAB-like notation. On the right, the

code list variables (written in normal mathematical notation) are shown as functions of the inputs x_1, x_2.

```
function [y1,y2] = f(x1,x2)
v1 = sin(x2)                    v1 = sin x2
v2 = v1-x1                      v2 = sin(x2) − x1
y1 = v2*x1                      y1 = x1 (sin(x2) − x1)
y2 = sqrt(v1)                   y2 = √sin(x2)
```
(4)

The independents are `x1,x2`, the dependents are `y1,y2`, and the intermediates are `v1,v2`. We wish to generate code to calculate $J = \partial(y_1, y_2)/\partial(x_1, x_2)$, comprising $\partial y_1/\partial x_1$, $\partial y_1/\partial x_2$, etc.

The basic linear relations of AD are obtained by differentiating the code line-by-line.

$$\begin{array}{lll} v_1 = \sin(v_2) & \mathrm{d}v_1 = \cos(x_2)\,\mathrm{d}x_2 \\ v_2 = v_1 - x_1 & \mathrm{d}v_2 = \mathrm{d}v_1 - \mathrm{d}x_1 \\ y_1 = x_1 v_2 & \mathrm{d}y_1 = x_1\,\mathrm{d}v_2 + v_2\,\mathrm{d}x_1 \\ y_2 = \sqrt{v_1} & \mathrm{d}y_2 = \frac{1}{2\sqrt{v_1}}\,\mathrm{d}v_1 \end{array} \tag{5}$$

The d's mean "derivatives with respect to whatever input variables we are interested in". Eliminating intermediate $\mathrm{d}v_k$ to get the $\mathrm{d}y_i$ as linear combinations of the $\mathrm{d}x_j$,

$$\mathrm{d}y_i = \sum_j J_{ij}\,\mathrm{d}x_j,$$

one obtains $J = [J_{ij}]$, the desired Jacobian matrix.

One way of computing J is by the classic *forward mode AD*. Here, d means *gradient with respect to the input variables*. In our example, $\mathrm{d} = (\partial/\partial x_1, \partial/\partial x_2)$. The process is shown in the table below.

Initialize with				
$\mathrm{d}x_1$	=		(1	0)
$\mathrm{d}x_2$	=		(0	1)
and continue				
$\mathrm{d}v_1$	=	$\cos(x_2)\,\mathrm{d}x_2$	=	(0 $\cos(x_2)$)
$\mathrm{d}v_2$	=	$\mathrm{d}v_1 - \mathrm{d}x_1$	=	(−1 $\cos(x_2)$)
ending with				
$\mathrm{d}y_1$	=	$x_1\,\mathrm{d}v_2 + v_2\,\mathrm{d}x_1$	=	...
$\mathrm{d}y_2$	=	$\frac{1}{2\sqrt{v_1}}\,\mathrm{d}v_1$	=	...

(formulae for entries in last two rows omitted)

(6)

This of course is done numerically at run time, wherein input values are not given in the symbolic way suggested in this table. The process amounts to eliminating the $\mathrm{d}v_k$ by *forward substitution*. If done straightforwardly, not taking into account the sparsity in the row vectors on the right, the cost (in terms of floating-point operations)

of computing $\nabla\mathbf{f}$ is about $3n$ times the cost of computing $\mathbf{f}$ [16]. Note that $\mathrm{d}x$ is also called the directional derivative of a given variable x, and that the forward-mode AD will augment the input code to produce a new code, which simultaneously evaluates the value of the function as well as its directional derivative $(\mathrm{d}y_1, \mathrm{d}y_2) = \nabla\mathbf{f} \cdot \mathbf{e}$ wherein $\mathbf{e}$ is a vector in the standard basis $\mathbb{R}^2$.

Another classical way of computing J is by the *reverse mode AD*. To illustrate this technique, let us consider our example $\mathbf{f}$ function for which a computer code and its *linearised computational graph* are shown on the left and right respectively of Figure 1. A vertex of the linearized computational graph represents an input, intermediate, or output variable; an edge (v_j, v_i) represents a dependency relationship stating that the calculation of v_i depends on v_j and is labeled by the partial derivative $\partial v_i/\partial v_j$.

$$\begin{aligned} v_1 &= \sin(x_2) \\ v_2 &= v_1 - x_1 \\ y_1 &= v_2 x_1 \\ y_2 &= \sqrt{v_1} \end{aligned}$$

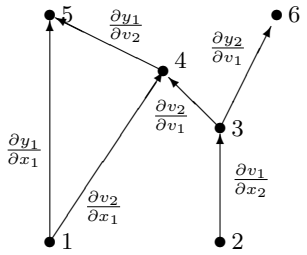


Figure 1. An example of code fragment and its linearised computational graph.

Denoting $\overline{v} = \frac{\partial y_1}{\partial v}$ or $\frac{\partial y_2}{\partial v}$, the reverse-mode AD augment the input code in order to evaluate the code of the left of equation (7) to get the function value $\mathbf{f}(x_1, x_2)$ and then the code on its right to calculate a directional derivative $(\overline{x_1}, \overline{x_2}) = \mathbf{e} \cdot \nabla\mathbf{f}$ wherein $\mathbf{e}$ is a vector in the standard basis $\mathbb{R}^2$.

$$\begin{aligned} v_1 &= \sin(x_2) & \overline{v_1} &= \tfrac{\partial y_2}{\partial v_1}\overline{y_2} \\ v_2 &= v_1 - x_1 & \overline{v_2} &= \tfrac{\partial y_1}{\partial v_2}\overline{y_1} \\ y_1 &= v_2 x_1 & \overline{x_1} &= \tfrac{\partial y_1}{\partial x_1}\overline{y_1} \\ y_2 &= \sqrt{v_1} & \overline{v_1} &= \overline{v_1} + \tfrac{\partial v_2}{\partial v_1}\overline{v_2} \\ & & \overline{x_1} &= \overline{x_1} + \tfrac{\partial v_2}{\partial x_1}\overline{v_2} \\ & & \overline{x_2} &= \tfrac{\partial v_1}{\partial x_1}\overline{v_1} \end{aligned} \tag{7}$$

The cost of computing $\nabla\mathbf{f}$ is about $3m$ times the cost of computing $\mathbf{f}$ [16], but the memory requirement may be excessive without the use of sophisticated check-pointing or recalculation strategies [16]. It follows that gradients with $m = 1$ use fewer floating-point operations with the reverse-mode AD.

Traditionally, numerical analysts use classic finite differencing schemes to estimate the derivative of a mathematical function. This estimation involves guessing a suitable step-size and incurs truncation errors, giving derivatives with an error of $O(\sqrt{\epsilon})$ at best while AD is exact within machine precision ϵ [12, 16]. Moreover, AD can lead to fast derivative computation by exploiting the structure of the code. For

example, the reverse-mode AD can evaluate the gradient of functions with a large number number of inputs with a cost that is proportional to that of evaluating the original function [16]. AD has been successfully applied to CFD, aerospace, finance, design optimization, or sensitivity analysis; see `www.autodiff.org` for more details.

Note also that AD is not a symbolic differentiation tool à la MAPLE or MATHEMATICA. While AD can differentiate an implicit function (it suffices to have a computer code that calculates the function), a symbolic differentiation tool requires an explicit formula of the function. It is worth observing that, while AD uses symbolic manipulations to differentiate the code, the evaluation of derivatives is carried out numerically. In that sense, AD is a good example of the combination of symbolic and numerical evaluations.

In terms of implementation, an AD tool can be written by using operator overloading or source-to-source transformation. The source-transformation approach of AD relies on compiler-construction technology. It parses the original code into an abstract syntax tree, as in the front end of a compiler, see [1]. Certain constructs in the abstract syntax tree may be transformed into a semantically equivalent one, suitable to applying the AD technique. This is termed *canonicalization*. Then, the code statements that calculate real-valued variables are augmented with additional statements to calculate their derivatives. Data-flow analyses can be performed in order to improve the performance of the AD transformed code, which can be compiled and ran for numerical simulations. A standard data-flow analysis is the activity analysis aimed at finding the set of active variables, since non-active variables will have a zero derivative; see for example [17].

2.1.2. About non-differentiability

A real-life application may contain mathematical functions that are not differentiable in some points in their domain. A computer code that models such an application may contain intrinsic functions (e.g., `abs` or `arccos`) or branching constructs used to treat physical constraints; for instance, non-physical values of model parameters. We will now describe three situations which may cause non-differentiability problems.

First, let us consider the case related to non-differentiable intrinsic functions. For instance, the derivative of cos^{-1} is not defined at $x = 0$ since

$$\frac{\mathrm{d}\cos^{-1}(x=1)}{\mathrm{d}x} = \infty.$$

Moreover, consider the function `abs`. Its derivative evaluated at point $x = 0$ has more than one possible value, including $-1, 0, 1$. Choosing one of these values depends upon the numerical application. This suggests that there is no "automatic" way of treating such a pathological case, and that code insight is crucial in guiding sensible choices. To date, the best thing an AD tool can do is provide an exception-handling mechanism that can be turned on in order to track down intrinsic related non-differentiable points. ADIFOR is a primary example of such a mechanism, and to our knowledge is unique in that respect (at the time of this writing).

Second, consider an engineering application in which the independent or dependent variables are real-valued, but complex-valued data has been used for computational purposes. Using the equivalence between $\mathbb{R}^2$ and $\mathbb{C}$, a complex function $h : a + ib \mapsto f(a, b) + ig(a, b)$ of a complex variable $a + ib$, where a, b are real values and f, g are real-valued functions, is differentiable if and only if h is *analytic* meaning $\frac{\partial f}{\partial a}=\frac{\partial g}{\partial b}$ and $\frac{\partial f}{\partial b}=-\frac{\partial g}{\partial a}$. It follows that the conjugate operator $z \mapsto \overline{z}$ is not differentiable. The application of AD into such complex-valued functions is discussed in [23]. This may raise subtle issues for the application of AD, which relies on the assumption that the input code is piecewise differentiable.

2.1.3. Iterative numerical solvers

An important question in using AD concerns differentiating through iterative processes. Typically, AD augments a given iteration with statements calculating derivatives. Empirically, AD provides the desired derivatives. However, questions remained as to whether the AD-generated iteration converges and to what it converges. Consider Fischer's example as discussed in [11]. The iterative constructor $x_{k+1} = g_k(x_k)$ with

$$g_k(x) = x \exp(-kx^2) \tag{8}$$

converges to $g \equiv 0$ when $k \to \infty$ whilst its derivative $g_k^{'}(x) \to 0$ but $g^{'}(0) = 1$. The issues of derivative convergence for iterative solvers in relation to AD are discussed in detail in [14, 15] for the forward-mode AD and in [9] for the adjoint mode. In [15], it has been shown that the mechanical application of AD to a fixpoint iteration gives a derivative fixpoint iteration that converges R-linearly to the desired derivative for a large class of nicely contractive iterates or secant updating methods.

Usually, current AD tools generate derivative code using the same number of iterations as the original solver. However, if the initial guess is close to the solution, then this adjoint solver no longer converges to the adjoint of the solution. For example, let us consider the following implicit iterative solver:

$$z_0 = z_0(x, y), \quad z_i = g(x, y, z_{i-1}) \ \text{ for } \ i = 1 \ldots l, \tag{9}$$

for l a non negative integer and the function g defined as:

$$\begin{aligned} g : \quad & \mathbb{R}^3 \to \mathbb{R} \\ & (x, y, z) \mapsto (y^2 + z^2)/x \end{aligned}$$

$z_0 = z_0(x, y)$ is meant z_0 is initialized for some values of x and y. For given values $x = 3, y = 2$ and an initial guess $z = 0.5$, the implicit equation

$$z = g(x, y, z)$$

has a solution $z_*{=}z_*(x, y){=}1$ and $\nabla g(x, y, z_*){=}(-1, 1)$. When the code in equation 9 is mechanically differentiated using, for exampl,e TAPENADE, we observed:

- If the initial guess is within a radius of the solution that leads to convergence, then the AD-generated iteration converged to the correct derivative.

- If the initial guess is closer to the solution, say the initial value of z=1, then the derivative iteration converged in one iteration to $\nabla g(x, y, z_*)$=$(-1/3, 1/3)$, which is wrong.

This means the assumption made by most AD tools to use the same number of iterations taken by the original iterative process for the derivative one is fair, but it may lead to wrong derivatives in certain cases. As suggested in [9], the AD tool ought to augment the convergence criterion to account for derivative convergence.

In summary, validating derivative calculation via AD can be difficult in the presence of non-differentiable functions and iterative solvers. It is hoped that future AD tools will help spot such anomalies and raise warnings to the AD user since, to our knowledge, there are no automatic ways of solving these issues.

2.2. Validating AD transformations

By validating a derivative code T from a source code S ($T = \mathrm{AD}(S)$), we mean that T and S have to satisfy the following property $p(S, T)$:

$$P(S) \Rightarrow Q(S, T), \tag{10}$$

wherein $P(S)$ means S has a well-defined semantics and represents a numerical function $\mathbf{f}$ and $Q(S, T)$ means $T = \mathrm{AD}(S)$ has a well-defined semantics and calculates a derivative $\mathbf{f}'(\mathbf{x}) \cdot \dot{\mathbf{x}}$ or $\overline{\mathbf{y}} \cdot \mathbf{f}'(\mathbf{x})$. Checking $p(S, T)$ implies the AD tool must ensure the function represented by the input code is piecewise differentiable prior to differentiation.

Traditionally, AD-generated codes are validated using software-testing recipes. The derivative code is run for a wide range of input data. For each run, we test the consistency of the derivative values using a combination of the following methods:

- Evaluate $\dot{\mathbf{y}} = \mathbf{f}'(\mathbf{x}) \cdot \dot{\mathbf{x}}$ using the forward mode and $\overline{\mathbf{x}} = \overline{\mathbf{y}} \cdot \mathbf{f}'(\mathbf{x})$ using the reverse mode and check the equality $\overline{\mathbf{y}} \cdot \dot{\mathbf{y}} = \overline{\mathbf{x}} \cdot \dot{\mathbf{x}}$.
- Evaluate $\mathbf{f}'(\mathbf{x}) \cdot \mathbf{e}_i$ for all vectors $\mathbf{e}_i$ in the standard basis of $\mathbb{R}^n$ using Finite Differencing (FD)
$$\dot{\mathbf{y}} = \mathbf{f}'(\mathbf{x}) \cdot \mathbf{e}_i \approx \frac{\mathbf{f}(\mathbf{x} + h\mathbf{e}_i) - \mathbf{f}(\mathbf{x})}{h}, \tag{11}$$
and then monitor the difference between the AD and FD derivative values against the FD's step size. For the 'best' step size, the difference should be of the order of the square root of machine-relative precision [16].
- Evaluate $\mathbf{f}'(\mathbf{x})$ using other AD tools or a hand-coded derivative code (if available) and compare the different derivative values; this should be the same (or within a few multiples) of machine precision.

The question is what actions should be taken if at least one of these tests does not hold. If we overlook the implementation quality of the AD tool, incorrect AD-derivative values may result from a violation of the piecewise differentiability assumption. The AD tool ADIFOR [7] provides an exception-handling mechanism, allowing the user to locate non-differentiable points at runtime for codes containing

non-differentiable intrinsic functions, such as `abs` or `max`. However, these intrinsic functions can be rewritten using branching constructs as performed by the TAPENADE AD tool [17]. To check the correctness of AD codes, one can use a *checker*, a Boolean-valued function $check(S,T)$ that formally verifies the validating property $p(S,T)$ by statically analyzing both codes to establish the following logical proposition:

$$check(S,T) = \text{true} \Rightarrow p(S,T) \tag{12}$$

In this approach, the checker itself must be validated. To avoid validating a possibly large code, we adopt a framework that relies on Necula's proof-carrying code [22].

2.3. Proof-Carrying Code

Proof-Carrying Code (PCC) [22] is based on the idea that the complexity of ensuring code safety can be shifted from the code consumer to the code producer by providing proof that the code satisfies some safety rules defined by the code consumer. Safety rules are verification conditions that must hold in order to guarantee the safety of the code. Verification conditions can be, for example, that the code cannot access a forbidden memory location, the code is memory-safe or type-safe, or the code executes within a well-specified time or resource usage limits. In the PCC paradigm, certification is about generating a formal proof that the code adheres to a well-defined safety policy, and validation consists of checking if the generated proof is correct by using a simple and trusted proof-checker.

3. Unifying PCC and AD validation

Unifying PCC and AD validation implies that it is the responsibility of the AD producer to ensure the correctness of the AD code T from a source S by providing a proof of the property $p(S,T)$ in equation (10) along with the generated code T or a counter-example (possibly an execution trace leading to a point of the program where the derivative function represented by T is not well-defined). For a given source code S, certifying AD software will return either nothing or a couple $(T = \text{AD}(S), C)$ wherein C is a certificate that should be used along with both codes S and T by the verifier *check* in order to establish the property $p(S,T)$ of equation (10). If we can generate the certificate C with the help of a proof-generator tool, then the correctness proof of the derivative code becomes

$$check(S,T,C) = \text{true} \Rightarrow p(S,T). \tag{13}$$

In this case, the AD user must run the verifier *check*, which is simply a proof-checker, a small and easy to certify program that checks whether the generated proof C is correct. There are variants of the PCC framework. For example, instead of generating an entire proof, it may be sufficient for the AD software to generate enough annotations or hints so that the proof can be constructed cheaply by a specialized theorem prover at the AD user site.

3.1. The Piecewise Differentiability Hypothesis (PDH)

The PDH (piecewise differentiability hypothesis) is the AD assumption that the input code is piecewise differentiable. This may be violated even in cases where the function represented by the input code is differentiable. A classic example is the identity function $y = f(x) = x$ written symbolically as

$$\text{if x = 0 then y = 0 else y = x endif.} \tag{14}$$

Applying AD to this code will give $f'(0) = 0$ in lieu of $f'(0) = 1$. This unfortunate scenario can happen whenever a control variable in a guard (logical expression) of an IF construct or a loop is active. Recall that a variable is active if it depends on an independent variable and it impacts a dependent variable. These scenarios can be tracked by computing the intersection between the set $V(e)$ of variables in each guard e and the set A of active variables in the program. If $V(e) \cap A = \emptyset$ for each guard e in the program, then the PDH holds; otherwise, the PDH may be violated, in which case an AD tool should at least issue a warning to the user that an identified construct in the program may cause non-differentiability of the input program.

Ideally, one would like to check the PDH for a given computer code to be differentiated. The following scheme outlines such a procedure:

1. Compute A the set of active variables of the program.
2. For each guard e, compute $V(e)$ the set of variables in e.
3. If $V(e) \cap A = \emptyset$ then the PDH holds,
   Else find the boundary values B described by the guard e,
   For each value $b \in B$, check if the local derivative obtained by AD is the same as that obtained using the standard definition of derivative evaluation,

$$f'(x_0) = \lim_{x \to x_0,\, x \neq x_0} \frac{f(x) - f(x_0)}{x - x_0}. \tag{15}$$

One can notice that, by applying the standard definition of derivative evaluation to the code in Equation (14), we can recover that $f'(0) = 1$, while an AD-generated code will produce $f'(0) = 0$. The remainder of this paper is devoted to how AD-generated codes can be certified using a variant of Hoare logic.

3.2. Certifying AD Code Properties

In differentiating a computer code, an AD user may wish to ensure confidence in the AD-generated code by specifying desirable properties to be checked. A property can be that the PD hypothesis holds or the AD-generated code is memory or type safe if the original code is. Generally speaking, AD software may have a canonicalization mechanism. That is, it may silently rewrite certain constructs within the input code prior to differentiation. The transformed input code should be proven semantically equivalent to the original one so that the AD user can trust the AD-generated code. This is even more necessary for legacy codes for which maintenance is crucial, and

the cost of maintaining different versions of the same code is not simply acceptable. In addition to the piecewise differentiability hypothesis, any prior transformation of the input code must be proven correct, and all extra statements involving derivative calculation must adhere to a safety policy defined by the AD user. For example, if the input code is thread-safe, then the differentiated one must also be thread-safe.

The following describes the idea behind our PCC framework. An AD user sends a program along with a configuration file with specifications regarding the differentiation process (e.g., independents, dependents) and possibly the safety properties to be checked. The AD server has a well-defined safety policy for generating derivatives. If the AD tool rewrote parts of the input code, then there should be proof that the transformed code fragment is computationally equivalent to the original. Moreover, an AD user might be interested in finding out if some parts of code are pierce-wise differentiable, in which case the AD-generated code must be proven correct. We can also specify desirable safety properties such as thread-safety (as discussed at the beginning of this section).

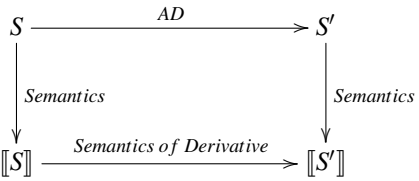
With the help of a theorem prover, the AD server should produce the AD code along with a certificate showing that the property holds or a counter-example invalidating it. This is sent to the AD user who has to check the given certificate by a simple proof-checker before using the AD-generated code or simulates the given counter-example. A proof assistant candidate is Coq [6], which has been used to develop a proof-carrying code approach to certify game-theoretic mechanisms in [3]. Coq is an interactive theorem prover based on the calculus of inductive constructions, allowing definitions of data types, predicates, and functions. It provides a meta-language enabling us to define different logics including Hoare Logic and higher-order logics [18]. Because of the use of higher-order logic, checking a proof within Coq can take an exponential time. On the other hand, the PCC framework works on the premise that the certificate will be expressed in a formalism, enabling its checking to be tractable. Nonetheless, if we check small code fragments, we can rely on Coq's proof checker to verify the certificates in our PCC approach; see for example [3]. Leaving aside the PCC implementation issues, we focus on the proof rules by using relational Hoare logic [4].

4. Foundational Certification of AD Transformations

In this section, we use Hoare logic [18, Chap. 4], a foundational formalism for program verification, to certify local code replacements or canonicalizations, and the forward and reverse modes of AD.

Given an input computer code S and its AD transformed S', we would like to show that $[\![S']\!]$, the semantics of S', coincide with that obtained using numerical differentiation as defined in equation (15). In essence, we wish to show the commutation

of the following diagram:



By *semantics of derivative*, we mean the mathematical meaning of a derivative $f'(x_0)$ defined as the limit in equation (15). The standard interpretation of limit tells us that:

$$\lim_{x \to x_0} f(x) = l$$

is defined as

$$\forall \epsilon > 0, \exists \eta > 0, \forall x \in D, |x - x_0| < \eta \rightarrow |f(x) - l| < \epsilon. \tag{16}$$

In other words, we wish to establish that if a computer code S representing a function f has well-defined semantics, then its AD-transformed code S' has well-defined semantics and represents the derivative f' viewed as the limiting process defined in equation (16).

4.1. Language used

In this work, we consider a WHILE-language composed of assignments, `if`, and `while` statements, and which expressions are formed using basic arithmetic or logical operations. We denote $\mathbb{V}$, a set of program variables, $\mathbb{E}$ the set of arithmetic expressions, $\mathbb{B}$ the set of Boolean expressions, and C the set of commands or statements. This language can be described as:

$$\begin{array}{lcl}
x & \in & \mathbb{V} \\
aop & \in & \{+, -, \times, /\} \\
rop & \in & \{<, >, ==, \leq, \ldots\} \\
lop & \in & \{\wedge, \vee, \neg, \ldots\} \\
e \in \mathbb{E} & ::= & const \mid x \mid e \text{ aop } e \\
b \in \mathbb{B} & ::= & true \mid false \mid e \text{ rop } e \mid b \text{ lop } b \\
c \in C & ::= & \text{skip} \mid x := e \mid c; c \mid \text{if } b \text{ then } c \text{ else } c \mid \text{while } b \text{ do } c
\end{array}$$

The states $\sigma \in \mathbb{S} = \mathbb{V} \rightarrow \mathbb{R}$ are defined as associations of values to variables, and the evaluation of expressions remains standard in the natural semantics. We denote $\sigma \rightarrowtail C \rightarrowtail \sigma'$ to mean a command c evaluated at an pre-state σ leads to a post-state σ'. This allows us to reason on the program by using Hoare logic.

Hoare logic is a sound and complete formal system providing logical rules for reasoning about the correctness of computer programs. For a given statement c, the Hoare triple $\{\phi\}c\{\psi\}$ means the execution of c in a state satisfying the pre-condition

ϕ will terminate in a state satisfying the post-condition ψ. The conditions ϕ and ψ are first order logical formulae called *assertions*. Hoare proofs are *compositional* in the structure of the language in which the program is written. For a given statement c, if the triple $\{\phi\}c\{\psi\}$ can be proven in the Hoare calculus, then the judgment $\vdash \{\phi\}c\{\psi\}$ is *valid*.

4.2. A Hoare logic for AD Canonicalizations

An AD canonicalization consists in locally replacing a piece of code C_1 by a new one C_2 suitable for the AD transformation [28]. For example, non-differentiable intrinsic functions can be rewritten using IF constructs [17]. In this case, one must ensure that $C_1 \sim C_2$ meaning C_1 and C_2 are computationally equivalent. That is, for any states $\sigma,\ \sigma'$ if $\sigma \rightarrowtail C_1 \rightarrowtail \sigma'$, then $\sigma \rightarrowtail C_2 \rightarrowtail \sigma'$.

The inference rules for AD canonicalization are given in Figure 2. They use a variant of the relational Hoare logic [4], wherein commands are run over one state in lieu of a couple of states as in [4]. The judgment $\vdash C_1 \sim C_2 : \phi \Rightarrow \psi$ means simply $\{\phi\}C_1\{\psi\} \Rightarrow \{\phi\}C_2\{\psi\}$. In the assignment rule (*asgn*), the lhs variable may be different but is kept the same for clarity. Also, notice that the same conditional branches must be taken (see the *if* rule) and that loops be executed the same number of times (see the *while* rule) on the source and target to guarantee their computational equivalence.

$$\frac{}{\vdash v := e_1 \ \sim \ v := e_2 : \ \ \phi[v/e_1] \wedge \phi[v/e_2] \Rightarrow \phi} \ asgn$$

$$\frac{\vdash s_1 \ \sim \ c_1 : \ \ \phi \Rightarrow \phi_0 \quad \vdash s_2 \ \sim \ c_2 : \ \ \phi_0 \Rightarrow \psi}{\vdash s_1; s_2 \ \sim c_1; c_2 : \ \ \phi \Rightarrow \psi} \ seq$$

$$\frac{\vdash s_1 \sim c_1 : \ \phi \wedge (b_1 \wedge b_2) \Rightarrow \psi \quad \vdash s_2 \sim c_2 : \ \phi \wedge \neg(b_1 \vee b_2) \Rightarrow \psi}{\vdash \text{if } b_1 \text{ then } s_1 \text{ else } s_2 \ \sim \text{if } b_2 \text{ then } c_1 \text{ else } c_2 \ : \ \ \phi \wedge (b_1 = b_2) \Rightarrow \psi} \ if$$

$$\frac{\vdash s \sim c : \phi \wedge (b_1 \wedge b_2) \Rightarrow \phi \wedge (b_1 = b_2)}{\vdash \text{while } b_1 \text{ do } s \ \ \sim \text{while } b_2 \text{ do } c \ : \ \ \phi \wedge (b_1 = b_2) \Rightarrow \phi \wedge \neg(b_1 \vee b_2)} \ while$$

$$\frac{\vdash \phi \Rightarrow \phi_0 \quad \vdash s \sim c : \ \phi_0 \Rightarrow \psi_0 \quad \vdash \psi_0 \Rightarrow \psi}{\vdash s \sim c : \ \phi \Rightarrow \psi} \ imp$$

Figure 2. Hoare logic for AD Canonicalization.

The relational Hoare logic is appropriate in the sense that it is both sound and complete with respect to the intended interpretation. We define $\sigma \models \phi$ to mean ϕ holds at the state σ.

Theorem 1 (Soundness of AD Canonicalization)**.** *If $C_1 \sim C_2 : \phi \Rightarrow \psi$, then for any states σ, σ' such that $\sigma \rightarrowtail C_1 \rightarrowtail \sigma'$, we have $\sigma \rightarrowtail C_2 \rightarrowtail \sigma'$, and $\sigma \models \phi \Rightarrow \sigma' \models \psi$*

Proof. As in [13], the proof of this statement is carried out by induction on the relation $\sim$: $C_1 \mapsto C_2$ and subordinate induction on $\sigma \rightarrowtail C \rightarrowtail \sigma'$ for the while loop. □

Theorem 2 (Completeness of AD Canonicalization)**.** *If, for any states* σ, σ' *such that* $\sigma \rightarrowtail C_1 \rightarrowtail \sigma'$, *we have* $\sigma \rightarrowtail C_2 \rightarrowtail \sigma'$ *and* $\sigma \models \phi \Rightarrow \sigma' \models \psi$, *then* $C_1 \sim C_2 : \phi \Rightarrow \psi$.

Proof. The proof is similar to that of [13]. □

4.3. A Hoare logic for the forward mode AD

The forward-mode AD can be used to compute the function $\mathbf{f}$'s derivative $\dot{\mathbf{y}}$ given a directional derivative $\dot{\mathbf{x}}$. Usually, $\dot{\mathbf{x}}$ is a vector of the standard basis of $\mathbb{R}^n$. Figure 3 shows how a program written in WHILE-language can be transformed using the forward-mode AD. It shows how an assignment, IF-construct, or Loop-construct can be augmented using the chain rule of calculus in order to evaluate derivative information. For example, given assignment S, its derivative S' is constructed using the chain rule and inserted just before S to form the sequence $T \equiv S'; S$. These transformation rules provide us with a recipe to build up a derivative code from an input code representing a mathematical function. In Figure 3, S_1 and S_2 are assignments.

Assignment

$$S: \quad z := e(\mathbf{x}) \quad \Rrightarrow \quad T: \quad \underbrace{dz := \sum_{i=1}^{n} \frac{\partial e\,(\mathbf{x})}{\partial x_i} \cdot dx_i \;;}_{S'} \; z := e(\mathbf{x})$$

Sequence of Assignments

$$S: \; S_1 \;;\; S_2 \;\Rrightarrow T: \; S_1' \;;\; S_1 \;;\; S_2' \;;\; S_2$$

If statement

$$S: \text{ if } b \text{ then } S_1 \text{ else } S_2 \;\Rrightarrow T: \text{ if } b \text{ then } S_1' \;;\; S_1 \text{ else } S_2' \;;\; S_2$$

Loop

$$S: \text{ while } b \text{ do } S_1 \text{ end} \;\Rrightarrow T: \text{ while } b \text{ do } S_1' \;;\; S_1 \text{ end}$$

Figure 3. Transformation rules for the forward-mode AD.

For a given source code S and its transformed $T = \mathrm{AD}(S)$ obtained by the transformation rules in Figure 3, we aim to establish the property $p(S,T)$ given in equation (10) in which $P(S)$ is understood as a Hoare triple $\{\phi\}S\{\psi\}$ establishing that S has a well-defined semantics and represents a function $\mathbf{f}$ and $Q(S,T)$ is understood as a derived triple $\{\phi'\}T\{\psi'\}$ establishing that T has a well-defined semantics and computes $\mathbf{f}'(\mathbf{x}) \cdot \dot{\mathbf{x}}$. Observe that the pre-conditions and post-conditions have changed from the source code to the transformed code in opposition to the basic rules of Figure 2. This reflects the fact that AD augments the semantics of the input code.

The relational Hoare logic rules for the forward-mode AD are given in Figure 4, in which $PDH == true$ tests if the PDH (piecewise differentiability hypothesis) holds. This condition is the first premise to be checked in the proof rules, and its checking amounts to checking that $V(e) \cap A = \emptyset$ wherein $V(e)$ is the set of variables in the logical

expression e and A the set of active variables. If it does not hold, then the correctness of an AD-generated code cannot be guaranteed. Furthermore, we sometimes gave names to certain long commands by preceding them with an identifier followed by ':'. The notation $S \Mapsto T$ means S is transformed into T. To give an idea of the proof rules, consider the assignment rule. It states that if in a pre-state, a statement S, $z := e(\mathbf{x})$, wherein $e(\mathbf{x})$ is an expression depending on $\mathbf{x}$, is transformed into the sequence T of the two assignments $dz := \sum_{i=1}^{n} \frac{\partial e}{\partial x_i} \cdot dx_i \ ; \ z := e(\mathbf{x})$, then we get the value of the lhs z and its derivative $dz = \frac{\partial e}{\partial x_i} \cdot \dot{\mathbf{x}}$ in a post-state.

$$\frac{}{\vdash S : z := e(\mathbf{x}) \ \Mapsto \ T : \ Q(S,T)[z/e(\mathbf{x}), dz/\sum_{i=1}^{n} \frac{\partial e\,(\mathbf{x})}{\partial x_i} \cdot dx_i] \Rightarrow Q(S,T)} \ asgn$$

$$\frac{\vdash S_1 \ \Mapsto \ T_1 : \ \ P(S_1) \Rightarrow Q(S_1,T_1) \quad \vdash S_2 \ \Mapsto \ T_2 : \ \ Q(S_1,T_1) \wedge P(S_2) \Rightarrow Q(S_2,T_2)}{\vdash S : \ S_1; S_2 \ \Mapsto T : \ T_1 \, ; \, T_2 : \ \ P(S) \Rightarrow Q(S,T)} \ seq$$

$$\frac{\vdash S_1 \Mapsto T_1 : \ P(S_1) \wedge b \Rightarrow Q(S_1,T_1) \quad PDH == true}{\vdash S : \text{if } b \ S_1 \text{ else } S_2 \ \Mapsto T : \text{if } b \text{ then } T_1 \text{ else } T_2 \ : \ P(S) \Rightarrow Q(S,T)} \ if\_true$$

$$\frac{\vdash S_2 \Mapsto T_2 : \ P(S_2) \wedge \neg b \Rightarrow Q(S_2,T_2) \quad PDH == true}{\vdash S : \text{if } b \ S_1 \text{ else } S_2 \ \Mapsto T : \text{if } b \text{ then } T_1 \text{ else } T_2 \ : \ P(S) \Rightarrow Q(S,T)} \ if\_false$$

$$\frac{\vdash S_1 \Mapsto T_1 : P(S_1,T_1) \wedge b \Rightarrow P(S_1,T_1) \quad PDH == true}{\vdash S : \text{while } b \text{ do } S_1 \ \Mapsto T : \text{while } b \text{ do } T_1 \ : \ P(S,T) \Rightarrow P(S,T) \wedge \neg b} \ while$$

$$\frac{\vdash P(S) \Rightarrow P_0 \quad \vdash S \Mapsto T : \ P_0 \Rightarrow Q_0 \quad \vdash Q_0 \Rightarrow Q(S,T)}{\vdash S \Mapsto T : \ P(S) \Rightarrow Q(S,T)} \ imp$$

Figure 4. Relational Hoare logic for the forward-mode AD.

The forward-mode AD transformation rules give a Hoare logical framework that is sound and complete.

Theorem 3 (Soundness of forward mode AD)**.** *If $S \Mapsto T : \phi' \Rightarrow \psi'$ and $\{\phi\}S\{\psi\}$, then for any states σ, σ', δ, δ' such that $\sigma \rightarrowtail S \rightarrowtail \sigma'$ and $\delta \rightarrowtail T \rightarrowtail \delta'$, we have $(\sigma \models \phi \Rightarrow \sigma' \models \psi) \Rightarrow (\delta \models \phi' \Rightarrow \delta' \models \psi')$.*

Theorem 4 (Completeness of forward mode AD)**.** *If for any states σ, σ', δ, δ' such that $\sigma \rightarrowtail S \rightarrowtail \sigma'$ and $\delta \rightarrowtail T \rightarrowtail \delta'$, we have $(\sigma \models \phi \Rightarrow \sigma' \models \psi) \Rightarrow (\delta \models \phi' \Rightarrow \delta' \models \psi')$, then $\{\phi\}S\{\psi\}$ and $S \Mapsto T : \phi' \Rightarrow \psi'$.*

5. Abductive Hoare logic for the reverse-mode AD

The reverse-mode AD can be implemented in order to compute the function **f**'s derivative $\overline{\mathbf{x}}$ given a directional derivative $\overline{\mathbf{y}}$. Usually, $\overline{\mathbf{y}}$ is a vector of the standard basis of $\mathbb{R}^m$. Figure 5 shows how a program in WHILE-language can be transformed using the reverse-mode AD assuming we have a trusted code implementing a `Stack` with the usual function `push` and `pop`.

The stack can be used to preserve the original value of the tests involved in the loop or branching instructions, but is irrelevant if those values cannot change during the evaluation of the input function. One may also recompute those values in lieu of storing them, but we omit this discussion here and refer the reader to the work reported elsewhere; for example, in [17].

As seen in the forward-mode AD, Figure 5 shows how the basic constructs of WHILE-language can be augmented in order to evaluate the derivative of a function encoded by a computer code in that language. Note how the reverse-mode AD evaluates first the original function before going backwards to evaluate the partial derivatives. The use of the Stack allows us to store information in the forward sweep and then use that information in the reverse sweep when it is needed. In Figure 5, S_1 and S_2 are assignments.

Assignment

$$S:\ z := e(\mathbf{x}) \ \Rightarrow\ T:\ z := e(\mathbf{x})\ ;\ \underbrace{\bar{x_i} = \bar{x_i} + \bar{z} \cdot \frac{\partial e\,(\mathbf{x})}{\partial x_i} \text{ for each active variable } x_i}_{\overline{S}}$$

Sequence of assignments

$$S:\ S_1\ ;\ S_2\ \Rightarrow T:\ S_1\ ;\ S_2\ ;\ \overline{S_2}\ ;\ \overline{S_1}$$

If statement

$$S:\ \text{if } b \text{ then } S_1 \text{ else } S_2\ \Rightarrow$$
$$T:\ \mathtt{push}(b);\ \text{if } b \text{ then } S_1\ \text{ else } S_2\ ;\ \mathtt{pop}(b);\ \text{if } b \text{ then } \overline{S_1}\ \text{ else } \overline{S_2}$$

Loop

$$S:\ \text{while } b \text{ do } S_1 \text{ end}\ \Rightarrow$$
$$T:\ \mathtt{push}(b);\ \text{while } b \text{ do } S_1\ ;\ \mathtt{push}(b);\ \text{end}\ ;\ \mathtt{pop}(b);\ \text{while } b \text{ do } \overline{S_1}\ ;\ \mathtt{pop}(b);\ \text{end}$$

Figure 5. Transformation rules for the reverse-mode AD.

5.1. Abductive Hoare logic

In logic, abduction is a kind of logical inference that seeks hypotheses in order to satisfy given observations or conclusions; see for example [19, 25]. Given that the reverse-mode AD uses a backward sweep to propagate sensitivities from the outputs to the inputs, it can be viewed as a way of finding origins for given anomalies or causes for given consequences. Abductive reasoning can be used to verify a derivative code obtained by the reverse-mode AD as follows: the execution of a computer program S representing a function $\mathbf{f}$ is a sequence of states from an initial state σ_i to a final state σ_f

$$\sigma_i = \sigma_1\ \rightarrow\ \sigma_2\ \rightarrow\ \sigma_3\ \rightarrow\ \ldots\ \rightarrow\ \sigma_q = \sigma_f$$

Each transition changes the state. The final state is reached after a finite number of transitions $q-1$. The reverse-mode AD augments the given program S by using a forward sweep to evaluate the function $\mathbf{f}$ and a backward sweep to accumulate the

partial derivatives of $\mathbf{f}$ with respect to its inputs. This augmentation gives rise to a new sequence of states associated with the transformed program T composed of the sequence $S; \overline{S}$. To verify that the property $p(S,T)$ given in equation (10) in which $P(S)$ is understood as a Hoare triple $\{\phi\}S\{\psi\}$ establishing that S has a well-defined semantics and represents a function $\mathbf{f}$ and $Q(S,T)$ is understood as a derived triple $\{\overline{\phi}\}T\{\overline{\psi}\}$ establishing that T has a well-defined semantics and computes $\overline{\mathbf{y}} \cdot \mathbf{f}'(\mathbf{x})$, an abductive approach may be to assert that the property $p(S,T)$ holds at the final state σ_q for some q and then finds the *weakest precondition wp* satisfied by the preceding state σ_{q-1}. The weakest precondition is one that describes the maximal set of possible preceding states such that the execution of T leads to a state satisfying the post-condition. In case a post-state can be reached from more than one pre-state, we compute the MOP (Meet Over all Paths) upper bound as the union of all pre-states. Repeatedly applying this reasoning to all intermediate states leading to σ_q, we can calculate a weakest precondition wp_0. If we have

$$wp_0 \Rightarrow \overline{\phi} \quad \text{and} \quad p(S,T) = true,$$

then the code generated by the reverse-mode AD is correct.

Example: Consider the code fragment in equation (14). By setting up the post-condition to be that the derivative of the output y is 1 and leaving aside the real arithmetic implementation, we can find the precondition: the value of x should be different from 0. In this case, we ensure the correctness of the AD-generated code for well-defined conditions; this should assist the AD user by pointing out cases wherein the AD-generated code may not be correct.

5.2. Generating preconditions by abduction

In this section, we explain how abduction is used to discover preconditions in order to verify that transformations performed by the reverse-mode AD are correct. In our analysis, abduction can be expressed as follows.

Abduction. Given an assumption A and a goal G, we aim to find a missing hypothesis H making the entailment

$$\vdash A \wedge H \Rightarrow G \tag{17}$$

We can always return the false assertion for the hypothesis H, but we need to find the best-possible solution. We say that H is a better solution of (17) than H', $H \precsim H'$, if $A \wedge H \Rightarrow G$ and $A \wedge H' \Rightarrow G$ and $H' \Rightarrow H$. In other words, we seek solutions that are minimal and consistent with the meaning of the relationship $\precsim$. Figure 6 shows the proof rules based on abduction in order to establish the correctness of the reverse-mode AD. Reading these proof rules from conclusion-to-premises can be viewed as a way for finding missing hypotheses H. This gives us a way of obtaining preconditions for some post-conditions to hold.

The key to reading the proof rules of Figure 6 is that they are of the following form:

$$\frac{\vdash H' \wedge A \Rightarrow G' \quad \text{Cond}}{\vdash H \wedge A \Rightarrow G} \tag{18}$$

In the equation (18) Cond represents a condition. This rule should be read as follows. In order to establish the entailment $\vdash H \wedge A \Rightarrow G$, the condition Cond is checked first. If it holds, then we make a recursive call to establish the smaller but related entailment $\vdash H' \wedge A \Rightarrow G'$. The solution H' of this simpler question is then used to compute the solution H of the original question. For example, the sequence rule seq expresses that in order to prove the correctness of the transformation of a source code S that is a sequence of two statements S_1 and S_2 into a target code T_1 ($S : S_1; S_2 \Mapsto T_1 : S_1; S_2; \overline{S_2}; \overline{S_1}$), we need to first prove that $S : S_1; S_2 \Mapsto T_2 : S_1; S_2; \overline{S_2}$ is correct. This provides us with an abductive procedure aimed at establishing that the reverse-mode AD evaluates the correct derivative.

$$\frac{}{\vdash S : z := e(\mathbf{x}) \ \Mapsto \ T : \ S; \ \overline{S} : \quad Q(S,T)[z/e(\mathbf{x}), \bar{x_i}/\frac{\partial e}{\partial x_i}\bar{z}] \Rightarrow Q(S,T)} \ asgn$$

$$\frac{\vdash S : \ S_1; S_2 \ \Mapsto T_2 : S_1; S_2; \overline{S_2} : \quad P(S) \wedge H' \Rightarrow Q(S,T_2)}{\vdash S : \ S_1; S_2 \ \Mapsto T_1 : S_1; S_2; \overline{S_2}; \overline{S_1} : \quad P(S) \wedge H \Rightarrow Q(S,T_1)} \ seq$$

$$\frac{\vdash S_1 \Mapsto T_1 : \ P(S_1) \wedge H' \wedge b \Rightarrow Q(S_1,T_1) \quad PDH == true}{\vdash S : \text{if } b \ S_1 \text{ else } S_2 \ \Mapsto T : \ P(S) \wedge H \Rightarrow Q(S,T)} \ if\_true$$

$$\frac{\vdash S_2 \Mapsto T_2 : \ P(S_2) \wedge H' \wedge \neg b \Rightarrow Q(S_2,T_2) \quad PDH == true}{\vdash S : \text{if } b \ S_1 \text{ else } S_2 \ \Mapsto T : \ P(S) \wedge H \Rightarrow Q(S,T)} \ if\_false$$

$$\frac{\vdash S_1 \Mapsto T_1 : P(S_1,T_1) \wedge b \wedge H' \Rightarrow P(S_1,T_1) \quad PDH == true}{\vdash S : \text{while } b \text{ do } S_1 \quad \Mapsto T : \quad P(S,T) \wedge H \Rightarrow P(S,T) \wedge \neg b} \ while$$

$$\frac{\vdash P(S) \Rightarrow P_0 \quad \vdash S \Mapsto T : \ P_0 \Rightarrow Q_0 \quad \vdash Q_0 \Rightarrow Q(S,T)}{\vdash S \Mapsto T : \ P(S) \Rightarrow Q(S,T)} \ imp$$

Figure 6. Abductive proof rules for the reverse-mode AD.

Note that abductive reasoning can also be applied to the forward-mode AD. As for the forward-mode AD, the inference rules for the reverse-mode AD are sound for the intended interpretation.

Theorem 5 (Correctness of the reverse-mode AD)**.** *$\{\phi\}S\{\psi\}$ and $S \Mapsto T : \phi' \Rightarrow \psi'$ iff for any states σ, σ', δ, δ' such that $\sigma \rightarrowtail S \rightarrowtail \sigma'$ and $\delta \rightarrowtail T \rightarrowtail \delta'$, we have $(\sigma \models \phi \Rightarrow \sigma' \models \psi) \Rightarrow (\delta \models \phi' \Rightarrow \delta' \models \psi')$.*

6. Related work

The idea of certifying AD derivatives is relatively new. Probably, this idea was first investigated in [21], wherein Coq has been used to develop a correctness proof of

the forward-mode AD. Araya and Hascoët [2] proposed a method that computes a valid neighborhood for a given directional derivative by looking at all branching tests, and finding a set of constraints that the directional derivative must satisfy. However, applying this method for every directional derivative may be very expensive for large codes. Our approach to validation, previously introduced in [26], is derived from work on certifying compiler optimizations and transformation validation for imperative languages [4, 20]. Our correctness proofs of AD canonicalizations are somewhat similar to Benton's relational Hoare logic for semantics equivalence between two pieces of code [4]. Our logical framework is inspired by that of compiler optimization techniques in [13]. In [24], a formalization of AD rules on basic functions is implemented in ACL2(r) in order to produce algebraic proofs of derivatives. Our approach for certifying AD transformation is based on the idea that the AD producer should be able to produce direct evidence in the form of a certificate for an AD-generated code and that the certificate can be easily checked by the AD user prior to using the derivative code. Our foundational certification of the forward-mode AD is an extension of relational Hoare logic calculus, since the assertions for the input code are augmented for the AD-transformed code. However, we have relied on abductive logic [25, 19] to construct the proof rules for the reverse-mode AD. Our abductive-Hoare-logic approach is inspired by work done on Separation Logic [5, 8], although our work does not use Separation Logic at all.

7. Conclusions and future work

We have highlighted the need to ensure trust in the AD-transformation framework and have presented an approach for that purpose. We then focused on the foundational aspects of providing such proof that an AD-transformed code is correct. We have shown that simple code transformations (or AD canonicalizations) and the actual semantics augmentation performed by the forward-mode AD can be certified using a Hoare-style calculus. We have also devised inference rules based on abductive logic for the correctness the reverse-mode AD. This first step is a small step compared to the work that needs to be done in order to fully certify an AD back-end.

The use of relational Hoare logic in this context has simplified proof rules. This formalism has potential and deserves further study. The use of abduction in the proof rules of the reverse-mode AD can be thought of as a natural way of understanding the reverse-mode AD, in the sense as the reverse-mode AD uses a backwards sweep to propagate sensitivities, the abduction procedure starts from the conclusion to search for an appropriate hypothesis. Our approach can be used by the proof-carrying code paradigm: an AD tool must provide a machine-checkable certificate for an AD-generated code, which can be checked by an AD user in polynomial time in the size of the certificate by using a simple and easy-to-certify program. Our theoretical approach needs to be implemented using an AD tool and a theorem prover such as Coq for at least the WHILE-language considered in this work.

References

[1] Aho A., Sethi R., Ullman J., Lam M.: *Compilers: principles, techniques, and tools*. Addison-Wesley Publishing Company, Boston, USA, Second ed., 2006.

[2] Araya-Polo M., Hascoët L.: *Certification of Directional Derivatives Computed by Automatic Differentiation. WSEAS Transactions on Circuits and Systems*, 2005.

[3] Bai W., Tadjouddine E.M., Payne T., Guan S.: A Proof-Carrying Code Approach to Certificate Auction Mechanisms. In: *The 10th International Symposium on Formal Aspects of Component Software*. 2013.

[4] Benton N.: Simple relational correctness proofs for static analyses and program transformations. In: *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 14–25. ACM Press, New York, NY, USA, 2004. ISBN 1-58113-729-X.

[5] Berdine J., Calcagno C., O'Hearn P. W.: Symbolic Execution with Separation Logic. In: *In APLAS*, pp. 52–68. Springer, 2005.

[6] Bertot Y., Castéran P.: *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Texts in theoret. comp. science. Springer-Verlag, 2004. ISBN 3-540-20854-2 (hardcover).

[7] Bischof C.H., Carle A., Khademi P., Mauer A.: ADIFOR 2.0: Automatic Differentiation of Fortran 77 Programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

[8] Calcagno C., Distefano D., O'Hearn P., Yang H.: Compositional shape analysis by means of bi-abduction. In: *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on POPL*, pp. 289–300. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-379-2.

[9] Christianson B.: *Reverse Accumulation and Attractive Fixed Points. Optimization Methods and Software*, 3 311–326, 1994.

[10] Cytron R., Ferrante J., Rosen B. K., Wegman M. N., Zadeck F. K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4): 451–490, 1991.

[11] Fischer H.: Special Problems in Automatic Differentiation. In: *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, A. Griewank, G. F. Corliss, eds., pp. 43–50. SIAM, Philadelphia, PA, 1991. ISBN 0–89871–284–X.

[12] Forth S. A., Tadjouddine M., Pryce J. D., Reid J. K.: Jacobian Code Generated by Source Transformation and Vertex Elimination can be as Efficient as Hand-Coding. *ACM Transactions on Mathematical Software*, 30(3): 266–299, 2004.

[13] Frade M. J., Saabas A., Uustalu T.: Foundational certification of data-flow analyses. In: *TASE*, pp. 107–116. 2007.

[14] Gilbert J. C.: Automatic Differentiation and Iterative Processes. *Optimization Methods and Software*, 1: 13–21, 1992.

[15] Griewank A., Bischof C., Corliss G., Carle A., Williamson K.: Derivative Convergence for Iterative Equation Solvers. *Optimization Methods and Software*, 2: 321–355, 1993.

[16] Griewank A., Walther A.: *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Frontiers in Appl. Math. SIAM, Philadelphia, PA, second ed., 2008.

[17] Hascoët L., Pascual V.: *TAPENADE 2.1 user's guide*. INRIA Sophia Antipolis, 2004, Route des Lucioles, 09902 Sophia Antipolis, France, 2004.
See `http://www.inria.fr/rrrt/rt-0300.html`.

[18] Huth M. R. A., Ryan M. D.: *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England, 2000. ISBN Hardback: ISBN 0521652006, Paperback: ISBN 0521656028.

[19] Inoue K.: Induction, Abduction, and Consequence-Finding. In: *ILP '01: Proceedings of the 11th Int' Conf. on Inductive Logic Programming*, pp. 65–79. Springer, London, UK, 2001. ISBN 3-540-42538-1.

[20] Leroy X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: *Proceedings of POPL'06*, pp. 42–54. 2006.

[21] Mayero M.: *Formalisation et automatisation de preuves en analyses rélle et numérique*. Ph.D. thesis, Université Paris VI, 2001.
`ftp://ftp.inria.fr/INRIA/LogiCal/Micaela.Mayero/papers/these-mayero.ps.gz`.

[22] Necula G. C.: Proof-carrying code. In: *Proceedings of POPL'97*, pp. 106–119. ACM Press, New York, NY, USA, 1997. ISBN 0-89791-853-3.

[23] Pusch G.D., Bischof C., Carle A.: *On Automatic Differentiation of Codes with COMPLEX Arithmetic with Respect to Real Variables*. Technical Memorandum ANL/MCS-TM-188, Argonne National Laboratory, Mathematics and Computer Science Division, 9700 South Casss Avenue, Argonne, IL 60439, 1995.

[24] Reid P., Gamboa R.: Automatic Differentiation in ACL2. In: *ITP*, M.C.J.D. van Eekelen, H. Geuvers, J. Schmaltz, F. Wiedijk, eds., *Lecture Notes in Computer Science*, vol. 6898, pp. 312–324. Springer, 2011.

[25] Ross B. J.: Running Programs Backwards: The Logical Inversion of Imperative Computation. *Formal Aspects of Computing*, 9: 331–348, 1998.

[26] Tadjouddine E. M.: On Formal Certification of AD Transformations. In: *Advances in Automatic Differentiation*, C. Bischof, M. Bücker, P. Hovland, U. Naumann, J. Utke, eds., *Lecture Notes in Computational Science and Engineering*, vol. 64, pp. 23–34. Springer, Berlin, 2008.

[27] Tadjouddine E. M., Cao Y.: An Option Price Model Calibration Using Algorithmic Differentiation. In: *ISCIS 2011*, E. Gelenbe, ed., Computer and Information Sciences, pp. 577–581. Springer, 2011.

[28] Tadjouddine M., Forth S. A., Keane A. J.: Adjoint differentiation of a Structural Dynamics Solver. In: *Automatic Differentiation: Applications, Theory, and Implementations*, M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris, eds., LNCSE, pp. 309–319. Springer, Berlin, Germany, 2005.

Affiliations

Emmanuel M. Tadjouddine
Xi'an Jiaotong-Liverpool University, Department of Computer Science & Software Engineering, 111 Ren Ai Road, Suzhou Industrial Park, Suzhou, P.R. China 215123, Emmanuel.Tadjouddine@xjtlu.edu.cn

Wenjin Lv
Xi'an Jiaotong-Liverpool University, Department of Computer Science & Software Engineering, 111 Ren Ai Road, Suzhou Industrial Park, Suzhou, P.R. China 215123

Received: 30.10.2013
Revised: 15.01.2014
Accepted: 15.01.2014