Utrecht University

MASTER THESIS

FORMALIZED PROOF OF AUTOMATIC DIFFERENTIATION IN COQ

Student:
Curtis Chin Jen Sem

Supervisors:
Mathijs Vákár
Wouter Swierstra

Department of Information and Computing Science
Last updated: June 10, 2020

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean quis dapibus lorem. Praesent volutpat feugiat erat. Quisque quis hendrerit lectus, et malesuada nisi. Quisque id elementum lectus. Phasellus sit amet ante ornare, hendrerit orci non, consectetur erat. Phasellus pulvinar orci urna. Donec fringilla fringilla ornare. Sed ut tempus arcu, eget ornare ligula.

Ut varius pretium pellentesque. Nam sit amet sapien lobortis nisl faucibus tempor. Curabitur non enim venenatis, euismod elit convallis, auctor sapien. Praesent eget urna sed justo luctus malesuada. In scelerisque metus nibh, ullamcorper efficitur eros fermentum non. Phasellus accumsan congue diam, non fringilla lorem fringilla sit amet. Ut molestie feugiat sagittis. Integer in lobortis justo, et euismod augue. Suspendisse nec euismod lectus, at condimentum magna. Pellentesque eu elementum dui.

Contents

1 Formalizing Forward-Mode AD

The formalization of the forward-mode automatic differentiation macro will be explained in the following sections. The formal proof will start from a base simply-typed lambda calculus enriched with product types and incrementally add both sum and array types. Also included in the final language is an implementation of primitive recursion using integer types. Many of the theorems and lemmas will stay consistent between sections as the overarching correctness statement does not change. Later section only add their respective cases to the corresponding proofs of the lemmas.

1.1 Simply Typed Lambda Calculus

As mentioned in background section **??**, we will make use of De-Bruijn indices in a intrinsic representation to formulate our language. Both addition and multiplication are included as operations on the real numbers. Our base language consists of the simply-typed lambda calculus with product types and real numbers, hereby referenced as $\Lambda_{\delta}^{\times,\to,R}$, where δ consists of the addition and multiplication operators.

Both the language constructs and the typing rules for this language are standard for a simply typed lambda calculus, as shown in 1. As expected we include variables, applications and abstractions in the `var`, `app` and `abs` constructors. Product types are added to the language in the form of binary projections, `first` and `second` to represent respectively the left and right projections of `tuple` terms. For real numbers, `rval` is used to introduce real numbered constants and `add` and `mul` will be used to respectively encode addition and multiplication.

These can be translated into Coq definitions in a reasonably straightforward manner, with each case keeping track of both how the typing context and types change. In the `var` case we need some way to determine what type the variable is referencing. Like many others previously[2][1], instead of using indices into the list accompanied with a proof that the index does not exceed the length of the list, we make use of an inductively defined type evidence to type our variables as shown in **??**. The cases for `app` and `abs` are as expected, where variables in the body of abstraction are able to reference their respective arguments.

Notably, in the original proof by Huot, Staton, and Vákár [6], they make use of n-ary products accompanied with pattern matching expressions. We opted to implement binary projection products, as they are conceptually simpler while still retaining much of the same functionality expected of

$$\frac{elem\ n\ \Gamma = \tau}{\Gamma \vdash var\ n : \tau}\ \textsc{TVar} \qquad \frac{(\sigma, \Gamma) \vdash t : \tau}{\Gamma \vdash abs\ t : \sigma \to \tau}\ \textsc{TAbs}$$

$$\frac{\Gamma \vdash t1 : \sigma \to \tau \qquad \Gamma \vdash t2 : \sigma}{\Gamma \vdash app\ t1\ t2 : \tau}\ \textsc{TApp}$$

$$\frac{\Gamma \vdash t1 : \tau \qquad \Gamma \vdash t2 : \sigma}{\Gamma \vdash tuple\ t1\ t2 : \tau \times \sigma}\ \textsc{TTuple}$$

$$\frac{\Gamma \vdash t : \tau \times \sigma}{\Gamma \vdash first\ t : \tau}\ \textsc{TFst} \qquad \frac{\Gamma \vdash t : \tau \times \sigma}{\Gamma \vdash second\ t : \sigma}\ \textsc{TSnd}$$

$$\frac{r \in \mathcal{R}}{\Gamma \vdash rval\ r : \mathsf{R}}\ \textsc{TRVal}$$

$$\frac{\Gamma \vdash r1 : \mathsf{R} \qquad \Gamma \vdash r2 : \mathsf{R}}{\Gamma \vdash add\ r1\ r2 : \mathsf{R}}\ \textsc{TAdd} \qquad \frac{\Gamma \vdash r1 : \mathsf{R} \qquad \Gamma \vdash r2 : \mathsf{R}}{\Gamma \vdash mul\ r1\ r2 : \mathsf{R}}\ \textsc{TMull}$$

Figure 1: Type-inference rules for the base simply-typed lambda calculus

$$\begin{aligned}\overrightarrow{\mathcal{D}}(\mathsf{R}) &= \mathsf{R}\times\mathsf{R}\\ \overrightarrow{\mathcal{D}}(\tau\times\sigma) &= \overrightarrow{\mathcal{D}}(\tau)\times\overrightarrow{\mathcal{D}}(\sigma)\\ \overrightarrow{\mathcal{D}}(\tau\to\sigma) &= \overrightarrow{\mathcal{D}}(\tau)\to\overrightarrow{\mathcal{D}}(\sigma)\end{aligned}\qquad \begin{aligned}\overrightarrow{\mathcal{D}}(rval\ n) &= tuple\ (rval\ n)\ (rval\ 0)\\ \overrightarrow{\mathcal{D}}(add\ n\ m) &= tuple\ (add\ n\ m)\ (add\ n'\ m')\\ \overrightarrow{\mathcal{D}}(mul\ n\ m) &= tuple\ (mul\ n\ m)\\ &\qquad (add\ (mul\ n'\ m)\ (mul\ m'\ n)))\end{aligned}$$

Figure 2: Macro on base simply-typed lambda calculus

product types.

We use the same inductively defined macro on types and terms used by many previous authors to implement the forward-mode automatic differentiation macro[6][5][4]. The forward-mode macro, $\overrightarrow{\mathcal{D}}$, keeps track of both primal and tangent traces using tuples as respectively its first and second elements. In most cases, the macro simply preserves the structure of the language. The cases for real numbers such as addition and multiplication are the exception. Here, the element encoding the tangent trace needs to contain the proper syntactic translation of the derivative of the operation.

This is implemented by destructing the recursive calls to $\mathcal{D}$ to access the syntactic counterparts of the primal and tangent denotations. Note that applying the macro to variables do nothing exceptional as the macro is also applied to the typing context, so variables implicitly already reference macro-applied terms.

Due to restricting our language to be total and excluding constructs related to partiality such as general recursion and iteration, it suffices to give our language a set-theoretic denotational semantics. Like the type evidences, well-typed terms will denotate to functions $[\![\Gamma]\!] \to [\![\tau]\!]$.

For any type τ, simply swap out the syntactic type to its corresponding **Coq** variant in `Type`. So the syntactic `R` will denotate to the type in **Coq**. Denotating the terms in our language now corresponds to finding the appropriate inhabitants in the denotated types. We denotate our typing contexts Γ, lists of types, as heterogeneous lists containing their corresponding denotations. The specific implementation of heterogeneous lists used, correspond to the one given by Adam Chlipala[3].

When working through giving the constructs in our language the proper denotations, most of the cases are straightforward. Notable is the case for variables, where we made use of an inductively defined evidence to type our terms. As denotations, these evidences wil correspond to lookups into

$$
\begin{array}{rlrl}
 & & [\![var\ v]\!] &= \lambda x.lookup\ [\![v]\!]\ x \\
[\![\mathsf{R}]\!] &= & [\![app\ t_1\ t_2]\!] &= \lambda x.([\![t_1]\!](x))([\![t_2]\!](x)) \\
[\![\tau \times \sigma]\!] &= [\![\tau]\!] * [\![\sigma]\!] & [\![abs\ t]\!] &= \lambda x\ y.[\![t]\!](y :: x) \\
[\![\tau \to \sigma]\!] &= [\![\tau]\!] \to [\![\sigma]\!] & [\![add\ t_1\ t_2]\!] &= \lambda x.[\![t_1]\!](x) + [\![t_2]\!](x) \\
 & & [\![mul\ t_1\ t_2]\!] &= \lambda x.[\![t_1]\!](x) * [\![t_2]\!](x) \\
[\![\mathsf{Top}]\!] &= \mathsf{hd} & [\![tuple\ t_1\ t_2]\!] &= \lambda x.([\![t_1]\!](x), [\![t_2]\!](x)) \\
[\![\mathsf{Pop}\ v]\!] &= \mathtt{tl} \circ [\![v]\!] & [\![first\ t]\!] &= \lambda x.let\ (x,y) = [\![t]\!](x)\ in\ x \\
 & & [\![second\ t]\!] &= \lambda x.let\ (x,y) = [\![t]\!](x)\ in\ y
\end{array}
$$

Figure 3: Denotations of the base simply-typed lambda calculus

our heterogeneous lists to their appropriate types.

As mentioned by by Barthe, et. al.[5], this small calculus, $\Lambda_{\delta}^{\times,\to,\mathbb{R}}$, accompanied with the arguably simple set-theoretic denotational semantics is expressive enough to encode the higher-order polynomials containing the addition and multiplication operators.

Example 1 (Square)**.** *$abs\ (mul\ (var\ Top)\ (var\ Top))$ denotates to the square function $\lambda x.x * x$.*

Proof. This follows from the definition of our denotation functions.

$$
\begin{aligned}
&[\![abs\ (mul\ (var\ Top)\ (var\ Top))]\!]\ [] \\
&\qquad \equiv \lambda x.[\![mul\ (var\ Top)\ (var\ Top)]\!]\ [x] \\
&\qquad \equiv \lambda x.[\![var\ Top]\!]\ [x] * [\![var\ Top]\!]\ [x] \\
&\qquad \equiv \lambda x.x * x
\end{aligned}
$$ □

As we work with denotations, smooth functions $f : \mathsf{R}^n - > \mathsf{R}$ can be interpreted as the denotations of a corresponding syntactic term $x_1, \ldots, x_n \vdash t : R$. Intuitively, the free variables in the term t denote the usages of the parameters of the function and as such are restricted to terms of type R, same as each of the arguments in the function f. Note that while both the arguments and result type of t are restricted to R, t itself can consist of higher order types.

Although Barthe, et. al.[5] gave a syntactic proof of correctness of the macro, our proof follows the more denotational style of proof given by Huot, Staton and Vákár[6]. Correctness of the forward-mode macro consists of the assertion that the denotation of any macro-applied term of type $x_1 : R, \ldots, x_n : R \vdash t : \mathsf{R}$ will result in a pair of both the denotation of the original term and the derivative of that denotation.

Likewise, the proof of correctness will follow a logical relations argument, first generalizing the statement for both the typing contexts and the result type. The relation will ensure that both the derivability property and the derivatives are preserved over higher-order types. We define the logical relation as a type-indexed relation between denotations of both terms and their macro-applied variants, so for any type τ, S_τ is the relation between functions $R \to [\![\tau]\!]$ and $R \to [\![\overrightarrow{\mathcal{D}}(\tau)]\!]$.

$$S_\tau(f, g) = \begin{cases} \mathit{smooth}\ f \wedge g = \lambda x.(f(x), \frac{\partial f}{\partial x}(x)) & : \tau = R \\ \exists f_1, f_2, g_1, g_2, & : \tau = \sigma \times \rho \\ \quad S_\sigma(f_1, f_2), S_\sigma(g_1, g_2). & \\ \quad f = \lambda x.(f_1(x), g_1(x)) \wedge & \\ \quad g = \lambda x.(f_2(x), g_2(x)) & \\ \forall f_1, f_2. & : \tau = \sigma \to \rho \\ \quad S_\sigma(f_1, f_2) \Rightarrow & \\ \quad S_\rho(\lambda x.f(x)(f_1(x)), \lambda x.f(x)(f_2(x))) & \end{cases} \tag{1}$$

The next step involves proving that syntactically well-typed terms are semantically correct. In other words, the relation needs to be proven valid for any term $x_1 : R, \ldots, x_n : R \vdash t : \tau$ and argument function $f : R \to R^n$ such that $S_\tau([\![t]\!] \circ f, [\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_n \circ f)$.

To properly instantiate the arguments to the denotation of the macro-applied term, an auxiliary function is needed that pairs each constant with their derivative 0. So it transforms $f : R \to [\![R^n]\!]$ into $\overrightarrow{\mathcal{D}}_n(f, x) : R \to [\![\overrightarrow{\mathcal{D}}(R)^n]\!]$. The full type signature of the function becomes $\overrightarrow{\mathcal{D}}_n : (R \to [\![R^n]\!]) \to R \to [\![\overrightarrow{\mathcal{D}}(R)^n]\!]$, which essentially accompanies each argument supplied by f with its accompanying derivative.

$$\overrightarrow{\mathcal{D}}_n(f, x) = \begin{cases} f(x) & : n = 0 \\ ((hd \circ f)(x)), \frac{\partial (hd \circ f)}{\partial x}(x)) :: \overrightarrow{\mathcal{D}}_{n'}(tl \circ f, x) & : n = S(n') \end{cases} \tag{2}$$

Proving this statement directly by induction on the typing derivation, however, does not work. As expected in a logical relations proof, the indicative issue lies in both the case for applications and abstractions. To make this work, the correctness statement needs to be generalized to arbitrary contexts and implicitly, substitutions.

If this were a syntactic proof, one would need to show that relation is preserved when applying substitutions consisting of arbitrary terms, possibly containing higher-order constructs. In this style of proof, the same concept needs to be incorporated in the argument function f, which intuitively speaking, supplies the terms referenced by variables through the typing context.

To prove this statement, it first needs to be generalized to arbitrary substitutions. The key in formulating these denotationally lies in what was previously the argument function $f : R \to R^n$. Previously the function was used to indicate the open variables or function arguments. If generalized to $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, this same function could be seen as a function which supplies terms foreach open variable $x_1, \ldots, x_n$ with their appropriate types. So the argument function now becomes the pair of functions $s : R \to [\![\Gamma]\!]$ and $s_D : R \to [\![\overrightarrow{\mathcal{D}}(\Gamma)]\!]$. Note that the functions s and s_D are built out of the denotations of terms such that these same denotations follow the logical relation (3) for our language. We phrase this requirement as a definition.

Definition 1. (*Instantiation*) *Substitutional functions $s : R \to [\![\Gamma]\!]$ and $s_D : R \to [\![\overrightarrow{\mathcal{D}}(\Gamma)]\!]$ are inductively instantiated such that they follow*

$$inst_\Gamma(f, g) = \begin{cases} f = const([]) \wedge g = const([]) & : \Gamma = [] \\ \forall f_1, f_2, g_1, g_2. & : \Gamma = (\tau :: \Gamma') \\ \quad inst_{\Gamma'}(f_1, g_1) \wedge S_\tau(f_2, g_2) & \\ \quad\quad \implies f = \lambda x.(f_2(x) :: f_1(x)) \wedge & \\ \quad\quad g = \lambda x.(g_2(x) :: g_1(x)) & \end{cases} \tag{3}$$

Using this notion of substitution instantiations we can now formulate our substitution lemma.

Lemma 1 (Substitution)**.** *For any well-typed term $\Gamma \vdash t : \tau$, and instantiation functions $s : R \to [\![\Gamma]\!]$ and $s_D : R \to [\![\overrightarrow{\mathcal{D}}(\Gamma)]\!]$ such that they follow $inst_\Gamma(s, s_D)$, then $S_\tau([\![t]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(t)]\!] \circ s_D)$.*

Note that the correctness of the macro is dependent on the requirement that the denotations supplied by the argument function are all continuously derivable.

Theorem 1 (Macro correctness)**.** *For any term $x_1 : R, ..., x_n : R \vdash t : R$, $[\![\overrightarrow{\mathcal{D}}(t)]\!]$ gives the dual number representation of $[\![t]\!]$, such that for any argument function $f : R \to R^n$, then $[\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_n \circ f = \lambda x.([\![t]\!] \circ f, {}^{\partial([\![t]\!] \circ f)}/_{\partial x})$.*

Proof. This is proven by showing that the goal follows from the logical relation which itself implied by the fundamental property (**??**) for well-typed terms.

$$
\begin{aligned}
&[\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_n \circ f = \lambda x.([\![t]\!] \circ f, {}^{\partial([\![t]\!] \circ f)}/_{\partial x}) \\
&\qquad \Vdash \text{(By definition of } S_R \text{ with } f := [\![t]\!] \circ f \text{ and } g := [\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_n \circ f) \\
&\qquad S_R([\![t]\!] \circ f, [\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_n \circ f) \\
&\qquad \Vdash \text{(Fundamental property (\textbf{??}))}
\end{aligned}
$$

□

1.2 Adding Sums and Primitive Recursion

Now that correctness has been verified for the base simply-typed lambda calculus, the next goal will be to add in both sum and integer types. In the interest of testing the flexibility of both the representation and the proofing technique, integer types and primitive recursion were also added. The inference rules for the new language constructs added for sum and number types are given in figure 4

$$\frac{\Gamma \vdash e : \tau + \sigma \qquad \Gamma \vdash t1 : \tau \to \rho \qquad \Gamma \vdash t2 : \sigma \to \rho}{\Gamma \vdash \mathit{case}\ e\ t1\ t2 : \rho}\ \text{TC{\small ASE}}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathit{inl}\ t : \tau + \sigma}\ \text{TI{\small NL}} \qquad \frac{\Gamma \vdash t : \sigma}{\Gamma \vdash \mathit{inr}\ t : \tau + \sigma}\ \text{TI{\small NR}}$$

$$\frac{n \in \mathcal{N}}{\Gamma \vdash \mathit{nval}\ n : \mathsf{N}}\ \text{TNV{\small AL}} \qquad \frac{\Gamma \vdash t : \mathsf{N}}{\Gamma \vdash \mathit{nsucc}\ t : \mathsf{N}}\ \text{TNS{\small UCC}}$$

$$\frac{\Gamma \vdash f : \tau \to \tau \qquad \Gamma \vdash n : \mathsf{N} \qquad \Gamma \vdash t : \tau}{\Gamma \vdash \mathit{nrec}\ f\ n\ t : \tau}\ \text{TP{\small RIM}}$$

Figure 4: Type-inference rules for language constructs for sum types and primitive recursion

1.3 Arrays

2 Optimization

2.1 Program Transformations

3 Reverse-Mode AD

4 Discussion

4.1 Problems

4.2 Future Work

5 Conclusion

A Language Definitions

B Forward-Mode Macro

C Denotations

References

[1] T. Coquand and P. Dybjer, "Inductive definitions and type theory an introduction (preliminary version)", in *Foundation of Software Technology and Theoretical Computer Science*, P. S. Thiagarajan, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 60–76, ISBN: 978-3-540-49054-8 (cit. on p. 4).

[2] N. Benton, C.-K. Hur, A. J. Kennedy, and C. McBride, "Strongly typed term representations in coq", *Journal of Automated Reasoning*, vol. 49, no. 2, pp. 141–159, Mar. 2011. DOI: `10.1007/s10817-011-9219-0`. [Online]. Available: `https://doi.org/10.1007/s10817-011-9219-0` (cit. on p. 4).

[3] A. Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013, ISBN: 0262026651 (cit. on p. 6).

[4] A. Shaikhha, A. Fitzgibbon, D. Vytiniotis, and S. Jones, "Efficient differentiable programming in a functional array-processing language", *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–30, Jul. 2019. DOI: `10.1145/3341701` (cit. on p. 6).

[5] G. Barthe, R. Crubillé, U. D. Lago, and F. Gavazzo, *On the versatility of open logical relations: Continuity, automatic differentiation, and a containment theorem*, 2020. arXiv: `2002.08489 [cs.PL]` (cit. on pp. 6, 7).

[6] M. Huot, S. Staton, and M. Vákár, *Correctness of automatic differentiation via diffeologies and categorical gluing*, 2020. arXiv: `2001.02209 [cs.PL]` (cit. on pp. 4, 6, 7).