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 16, 2020*

# 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 introduced in section 1.1 do not change, as they are independent of the types and terms in the language.

## 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 classic simply-typed lambda calculus with product types and real numbers.

Both the language constructs and the typing rules for this language are common for a simply typed lambda calculus, as shown in figure 1. As expected, we include variables, applications and abstractions in the language using the `var`, `app` and `abs` constructors. Product types are added to the language in the form of binary projections. The `first` and `second` constructors 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 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 abstractions are able to reference their respective arguments.

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

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

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

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

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

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

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

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

```
Inductive tm (Γ : Ctx) : ty → Type :=
  ...
  (* Binary projection products *)
  | tuple : forall τ σ,
    tm Γ τ →
    tm Γ σ →
    tm Γ (τ × σ)
  | first : forall τ σ, tm Γ (τ × σ) → tm Γ τ
  | second : forall τ σ, tm Γ (τ × σ) → tm Γ σ
  (* Operations on reals *)
  | rval : forall r, tm Γ R
  | add : tm Γ R → tm Γ R → tm Γ τ
  | mul : tm Γ R → tm Γ R → tm Γ σ
```

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.

Due to the intrinsic nature of our language representation, the macro also needs to be applied to both the types and typing context to ensure that the terms remain well-typed. In other words, for any well-typed term $\Gamma \vdash t : \tau$, applying the forward-mode macro results in a well-typed term in the macro-expanded context, $\overrightarrow{\mathcal{D}}(\Gamma) \vdash \overrightarrow{\mathcal{D}}(t) : \overrightarrow{\mathcal{D}}(\tau)$.

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

Figure 2: Macro on base simply-typed lambda calculus

Applying the macro to a term gives the syntactic counterparts of both their primal and tangent denotations as a tuple. These terms can be accessed with

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

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

projections to implement the various derivative implementations of the operations on real terms included in the language. Note that applying the macro to the case for variables does nothing as the macro is also applied to the typing context, so variables implicitly already reference macro-applied terms.

Due to restricting our language to total constructions and excluding concepts such as general recursion and iteration, it suffices to give our language a set-theoretic denotational semantics. In this case the types $\mathtt{R}, \Rightarrow, \times$ directly correspond to their Coq equivalent, respectively $\mathcal{R}, \to, \star$. Like the type evidences, well-typed terms will denotate to functions $[\![\Gamma]\!] \to [\![\tau]\!]$.

Denotating the terms in our language now corresponds to finding the appropriate inhabitants in the denotated types. As typing contexts, $\Gamma$, are represented by lists of types. The appropriate way to denotate these would be to map the denotation function over the list. The resulting heterogeneous list contains the denotations of each type in the list in the correct order. The specific implementation of heterogeneous lists used in the proof, corresponds to the one given by Adam Chlipala[3]. In this implementation, heterogeneous lists consist of an underlying list of some type $A$ and an accompanying function $A \to Set$, which in our use case are, respectively, the typing context and the denotation function.

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

```
Equations denote_v Γ τ (v: τ ∈ Γ) : ⟦Γ⟧ → ⟦τ⟧ :=
denote_v (Top Γ τ) (HCons h t) := h;
denote_v (Pop Γ τ σ v) (HCons h t) := denote_v v t.
```

**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.

$$\begin{aligned}
&\llbracket abs\ (mul\ (var\ Top)\ (var\ Top))\rrbracket\ [] \\
&\quad \equiv \lambda x.\llbracket mul\ (var\ Top)\ (var\ Top)\rrbracket\ [x] \\
&\quad \equiv \lambda x.\llbracket var\ Top\rrbracket\ [x] * \llbracket var\ Top\rrbracket\ [x] \\
&\quad \equiv \lambda x.x * x
\end{aligned}$$

□

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 $f$.

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]. Likewise, our proof of correctness will follow a similar logical relations argument. 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. Note that while both the arguments and result type of $t$ are restricted to $\mathsf{R}$, $t$ itself can consist of higher order types. This hints at the need for a more generalized statement.

The logical relation will ensure that both the smoothness 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 $\tau$, $S_\tau$ is the relation between functions $R \to \llbracket\tau\rrbracket$ and $R \to \llbracket\overrightarrow{\mathcal{D}}(\tau)\rrbracket$.

When $\tau = \mathsf{R}$, the denotation of the macro-applied term should give both the original denotation and its derivative. With function types, as long as the relation is valid for the argument, applying these argument functions to the tracked denotations should preserve the relation. Some care has to be taken in the case for products. Notably, the denotations of the subterms, $R \to \llbracket\tau\rrbracket$ and $R \to \llbracket\sigma\rrbracket$,

should be existentially quantified as these are dependent on the original denotation $R \to [\![\tau \times \sigma]\!]$.

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

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 (1) 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 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 $S_\tau([\![t]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(t)]\!] \circ s_D)$.*

The proof proceeds by induction on the typing derivation of the well-typed term $t$. The majority of cases follow from the induction hypothesis. The case for `var` follows from $inst$ which ensures that any term referenced is semantically well-typed with respect to the relation. Proving the cases used to encode the operators on reals such as `add` and `mul` involve proving both smoothness and giving the witness of the derivative.

We can derive the fundamental property of the base logical relation directly from the substitution lemma. This involves proving the prerequisite $inst$ we used previously. Note that the correctness of both the macro and the fundamental property is dependent on the requirement that the denotations supplied by the argument function are smooth.

**Lemma 2** (Fundamental property)**.** *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 $S_\tau([\![t]\!] \circ f, [\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_n \circ f)$.*

This is proven using the substitution lemma. The remaining goal of $inst_{R^n}$ is proven by induction on the number of arguments, $n$. With the case where $n = 0$, the goal is trivial due to the argument function $f$ being extensionally equal

to $const\ []$, which directly corresponds to $inst_{[]}$. Similarly the induction step is proven by both the induction hypothesis and the assumption that the denotations of the arguments supplied are smooth.

**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 (2) for well-typed terms.

$$
\begin{aligned}
&[\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_n \circ f = \lambda x.([\![t]\!] \circ f, {}^{\partial([\![t]\!] \circ f)}/_{\partial x}) \\
&\qquad \Vdash \text{(By definition of } S_R \text{ with } f := [\![t]\!] \circ f \text{ and } g := [\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_n \circ f) \\
&\qquad S_R([\![t]\!] \circ f, [\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_n \circ f) \\
&\qquad \Vdash \text{(Fundamental property (2))}
\end{aligned}
$$

□

## 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.

```
Inductive tm (Γ : Ctx) : ty → Type :=
  ...
  (* Sums *)
  | case : forall τ σ ρ,
    tm Γ (τ <+> σ) →
    tm Γ (τ ⇒ ρ) →
    tm Γ (σ ⇒ ρ) →
    tm Γ ρ
  | inl : forall τ σ,
    tm Γ τ → tm Γ (τ <+> σ)
  | inr : forall τ σ,
    tm Γ σ → tm Γ (τ <+> σ)
```

$$\frac{\Gamma \vdash e : \tau + \sigma \qquad \Gamma \vdash t1 : \tau \to \rho \qquad \Gamma \vdash t2 : \sigma \to \rho}{\Gamma \vdash case\ e\ t1\ t2 : \rho}\ \textsc{TCase}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash inl\ t : \tau + \sigma}\ \textsc{TInl} \qquad \frac{\Gamma \vdash t : \sigma}{\Gamma \vdash inr\ t : \tau + \sigma}\ \textsc{TInr}$$

$$\frac{n \in \mathcal{N}}{\Gamma \vdash nval\ n : \mathsf{N}}\ \textsc{TNVal} \qquad \frac{\Gamma \vdash t : \mathsf{N}}{\Gamma \vdash nsucc\ t : \mathsf{N}}\ \textsc{TNSucc}$$

$$\frac{\Gamma \vdash f : \tau \to \tau \qquad \Gamma \vdash n : \mathsf{N} \qquad \Gamma \vdash t : \tau}{\Gamma \vdash nrec\ f\ n\ t : \tau}\ \textsc{TPrim}$$

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

Binary sum types are included in the language using `inl` and `inr` as introducing terms. The `case` term encodes case-analysis given a function term for each possibility. Primitive recursion is implemented using simple integers, where a endomorphic function is recursively applied a bounded number of times to a start value. For convenience, an additional successor function is added in the form of the `nsucc` term.

```
Inductive tm (Γ : Ctx) : ty → Type :=
  ...
  (* Primitive recursion *)
  | nval : forall n, tm Γ N
  | nsucc : tm Γ N → tm Γ N
  | nrec : forall τ,
    tm Γ (τ ⇒ τ) → tm Γ N → tm Γ τ → tm Γ τ
```

In terms of denotations, `case` expressions will follow the same lines as `app` as they both involve applying a function to an argument. Note that the sum term first needs to be destructed to be able to determine which function branch to apply. Both `inl` and `inr` will map to their **Coq** counterparts. For `nrec`, the number of applications should be dependent on the input integer.

Both sums and integer terms are structure preserving with respect to the forward-mode macro. Note that we only take the derivative of values of type

$$
\begin{aligned}
[\![\tau \texttt{<+>} \sigma]\!] &= [\![\tau]\!] + [\![\sigma]\!] \\
[\![N]\!] &= \mathcal{N}
\end{aligned}
$$

$$
\begin{aligned}
[\![case\ e\ t_1\ t_2]\!] &= \lambda x. \begin{cases} ([\![t_1]\!](x))(t) & : [\![e]\!](x) = inl(t) \\ ([\![t_2]\!](x))(t) & : [\![e]\!](x) = inr(t) \end{cases} \\
[\![inl\ t]\!] &= \lambda x. inl([\![t]\!](x)) \\
[\![inr\ t]\!] &= \lambda x. inr([\![t]\!](x)) \\
[\![nval\ n]\!] &= n \\
[\![nsucc\ t]\!] &= \lambda x. [\![t]\!](x) + 1 \\
[\![nrec\ f\ i\ t]\!] &= \lambda x. fold([\![f]\!](x), [\![i]\!](x), [\![t]\!](x)) \\
fold(f, i, t) &= \begin{cases} t & : i = 0 \\ f(fold(f, i', t)) & : i = i' + 1 \end{cases}
\end{aligned}
$$

Figure 5: Denotations of the sum and integer terms

R, so when integers are encountered. More specifically, we do not keep track of derivatives at integer types as the tangent space is 0-dimensional.

For a similar reason, the logical relation at integer types only needs establish that denotations of integer terms are constant. For sum terms, the functions tracked are either the left or right tag of the sum. This is neatly defined using a logical disjunction.

$$
S_\tau(f, g) = \begin{cases}
f = g \wedge \exists n. f = const(n) & : \tau = \texttt{N} \\
(\exists f_1, f_2, & : \tau = \sigma \texttt{<+>} \rho \\
\quad S_\sigma(f_1, f_2) \wedge & \\
\quad f = inl \circ f_1 \wedge & \\
\quad g = inl \circ g_1) \vee & \\
(\exists f_1, f_2, & \\
\quad S_\rho(f_1, f_2) \wedge & \\
\quad f = inr \circ f_1 \wedge & \\
\quad g = inr \circ g_1) &
\end{cases} \tag{4}
$$

The only lemma or theorem that requires extension to deal with these new terms is the substitution lemma, as the validity of every other statement is independent of the additional types added to our language. With terms of integer type, the proof for the substitution lemma is trivial using the definition of our

$$
\begin{aligned}
\overrightarrow{\mathcal{D}}(\mathsf{N}) &= \mathsf{N} \\
\overrightarrow{\mathcal{D}}(\tau \texttt{<+>} \sigma) &= \overrightarrow{\mathcal{D}}(\tau) \texttt{<+>} \overrightarrow{\mathcal{D}}(\sigma) \\
\overrightarrow{\mathcal{D}}(inl\ t) &= inl\ \overrightarrow{\mathcal{D}}(t) \\
\overrightarrow{\mathcal{D}}(inr\ t) &= inr\ \overrightarrow{\mathcal{D}}(t) \\
\overrightarrow{\mathcal{D}}(case\ e\ t_1\ t_2) &= case\ \overrightarrow{\mathcal{D}}(e)\ \overrightarrow{\mathcal{D}}(t_1)\ \overrightarrow{\mathcal{D}}(t_2) \\
\overrightarrow{\mathcal{D}}(nval\ n) &= nval\ n \\
\overrightarrow{\mathcal{D}}(nsucc\ nm) &= nsucc\ \overrightarrow{\mathcal{D}}(n) \overrightarrow{\mathcal{D}}(m) \\
\overrightarrow{\mathcal{D}}(nrec\ f\ i\ t) &= nrec\ \overrightarrow{\mathcal{D}}(f)\ \overrightarrow{\mathcal{D}}(i)\ \overrightarrow{\mathcal{D}}(t)
\end{aligned}
$$

Figure 6: Macro on the sum and integer terms

denotation functions. The `nrec` case for primitive recursion is only slightly more difficult as case-analysis on the denotation of the iteration term is required, where the $0$ and $n + 1$ case are proven using the induction hypotheses derived from respectively the initial and function terms. As expected with the `case` term for sums, the denotation of the term under scrutiny needs to be destructed to properly apply the two disjunct induction hypotheses to their respective cases.

### 1.3 Arrays

Rarely is automatic differentiation done on mono-valued real numbers, due to the massive computational power available in GPUs in the form of array operations. So the next extension worth considering in our language are the array types. To be more specific we will be considering the array types and terms presented by Shaikhha, et. al.[4]. The well-typed nature of our presentation allows us to avoid much of the hairy details associated with bounds checking both indexing and array creation constructions. This is possible using the finite datatype, `Fin`, which is indexed by the upper-bound and represents for some $n$, the range of $[1..n]$.

Both the macro and denotation functions deviate slightly from how the previous terms were defined. When looking at the macro, while it previously sufficed to recursively call the macro on subterms, this is not possible as the subterm of interest is now a function. This can be solved by substituting the function by a composition of itself combined with the forward-mode macro, essentially applying the macro to every possible result of the function.

Similarly for denotations, the denotation function, instantiated to the correct

$$\frac{\Gamma \vdash f : Fin\ n \to tm\Gamma\tau}{\Gamma \vdash build\ n\ f : Array\ n\ \tau}\ \textsc{TBuild}$$

$$\frac{\Gamma \vdash t : Array\ n\ \tau \qquad \Gamma \vdash i : Fin\ n}{\Gamma \vdash get\ i\ t : \tau}\ \textsc{TGet}$$

Figure 7: Type-inference rules for array construction and indexing

$$\begin{aligned}\overrightarrow{\mathcal{D}}(Array\ n\ R) &= Array\ n\ \overrightarrow{\mathcal{D}}(R)\\ \overrightarrow{\mathcal{D}}(build\ n\ f) &= build\ n\ (\overrightarrow{\mathcal{D}} \circ f)\\ \overrightarrow{\mathcal{D}}(get\ i\ t) &= get\ i\ \overrightarrow{\mathcal{D}}(t)\end{aligned}$$

Figure 8: Macro on array construction and indexing terms

type, has to be passed along to an auxiliary function that builds up a vector of denotation terms. Appropriately, array types will denotate to vectors indexed by length. There is some additional boilerplate necessary to circumvent the structurally recursive requirement imposed by the **Coq** type checker. Note that in the $S$ case of $vectorize$, the $Fin$ and $nat$ types are treated interchangeably, where $Fin\ n$ is interpreted as $n$. The recursive call to $vectorize$ also transforms the function by incrementing the bounded integer given as its argument.

The logical relation for array types needs to exhibit the same behavior with

$$\begin{aligned}[\![Array\ n\ \tau]\!] &= vector(n, [\![\tau]\!])\\ [\![build\ n\ f]\!] &= \lambda x.vect(n, [\![]\!] \circ f, x)\\ [\![get\ i\ t]\!] &= \lambda x.[\![t]\!](x)!i\end{aligned}$$

$$vect(i, f, x) = \begin{cases}[] & : i = 0\\ f(i, x) :: vect(i', \lambda j.f(j+1), x) & : i = S(i')\end{cases}$$

Figure 9: Denotations of the array construction and indexing terms

respect to both construction and indexing in how it preserves the relation on subterms. This is accomplished by first and foremost, quantifying over the indices possible for the vector denotation. Next, each subdenotation needs to both preserve the relation and be extensionally equal to the appropriate projection of the term.

$$S_\tau(f,g) = \begin{cases} \forall i.\exists f_1, g_1. & : \tau = Array\ n\ \sigma \\ \quad S_\sigma(f_1, g_1) \wedge & \\ \quad\quad f_1 = \lambda x.f(x)!i \wedge & \\ \quad\quad f_1 = \lambda x.g(x)!i & \end{cases} \tag{5}$$

As was the case for sum types, only the proof of the substitution lemma needs to be extended. The proof for the array terms proceeds as follows. For `build`, we first do induction on $n$, the length of the array. The base case is trivial, as $Fin\ 0$ contains $0$ inhabitants. For the induction step, we first do case-analysis on the indices, $i$, where the $(+1)$ case follows from the induction hypothesis. For $i = 1$ it suffices to give the proper inhabitants using the induction hypothesis derived from the function used for construction.

## 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. 7).

[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 pp. 6, 14).

[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, 8).

[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, 8).