

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 8, 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

<b>1</b>	<b>Formalizing Forward-Mode AD</b>	<b>4</b>
1.1	Simply Typed Lambda Calculus . . . . .	4
1.2	Adding Sums and Primitive Recursion . . . . .	21
1.3	Arrays . . . . .	22
<b>2</b>	<b>Optimization</b>	<b>22</b>
2.1	Program Transformations . . . . .	22
<b>3</b>	<b>Reverse-Mode AD</b>	<b>22</b>
<b>4</b>	<b>Discussion</b>	<b>22</b>
4.1	Problems . . . . .	22
4.2	Future Work . . . . .	22
<b>5</b>	<b>Conclusion</b>	<b>22</b>
<b>A</b>	<b>Language Definitions</b>	<b>22</b>
<b>B</b>	<b>Forward-Mode Macro</b>	<b>22</b>
<b>C</b>	<b>Denotations</b>	<b>22</b>

## 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-Brujin 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  $\Lambda_{\delta}^{\times, \rightarrow, R}$ , where  $\delta$  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

$$\begin{array}{c}
\frac{elem\ n\ \Gamma = \tau}{\Gamma \vdash var\ n : \tau} \text{TVAR} \qquad \frac{(\sigma, \Gamma) \vdash t : \tau}{\Gamma \vdash abs\ t : \sigma \rightarrow \tau} \text{TAbs} \\
\\
\frac{\Gamma \vdash t1 : \sigma \rightarrow \tau \quad \Gamma \vdash t2 : \sigma}{\Gamma \vdash app\ t1\ t2 : \tau} \text{TAPP} \\
\\
\frac{\Gamma \vdash t1 : \tau \quad \Gamma \vdash t2 : \sigma}{\Gamma \vdash tuple\ t1\ t2 : \tau \times \sigma} \text{TTUPLE} \\
\\
\frac{\Gamma \vdash t : \tau \times \sigma}{\Gamma \vdash first\ t : \tau} \text{TFST} \qquad \frac{\Gamma \vdash t : \tau \times \sigma}{\Gamma \vdash second\ t : \sigma} \text{TSND} \\
\\
\frac{r \in \mathcal{R}}{\Gamma \vdash rval\ r : R} \text{TRVAL} \\
\\
\frac{\Gamma \vdash r1 : R \quad \Gamma \vdash r2 : R}{\Gamma \vdash add\ r1\ r2 : R} \text{TADD} \qquad \frac{\Gamma \vdash r1 : R \quad \Gamma \vdash r2 : R}{\Gamma \vdash mul\ r1\ r2 : R} \text{TMULL}
\end{array}$$

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

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

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,  $\vec{\mathcal{D}}$ , 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 denote to functions  $\llbracket \Gamma \rrbracket \rightarrow \llbracket \tau \rrbracket$ .

For any type  $\tau$ , simply swap out the syntactic type to its corresponding **Coq** variant in **Type**. So the syntactic  $R$  will denote to the type in **Coq**. Denoting the terms in our language now corresponds to finding the appropriate inhabitants in the denotated types. We denote our typing contexts  $\Gamma$ , 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 will correspond to lookups into

$$\begin{array}{ll}
\llbracket \mathbf{R} \rrbracket = & \llbracket \text{var } v \rrbracket = \lambda x. \text{lookup } \llbracket v \rrbracket x \\
\llbracket \tau \times \sigma \rrbracket = \llbracket \tau \rrbracket * \llbracket \sigma \rrbracket & \llbracket \text{app } t_1 t_2 \rrbracket = \lambda x. (\llbracket t_1 \rrbracket(x))(\llbracket t_2 \rrbracket(x)) \\
\llbracket \tau \rightarrow \sigma \rrbracket = \llbracket \tau \rrbracket \rightarrow \llbracket \sigma \rrbracket & \llbracket \text{abs } t \rrbracket = \lambda x y. \llbracket t \rrbracket(y :: x) \\
\llbracket \text{Top} \rrbracket = \text{hd} & \llbracket \text{add } t_1 t_2 \rrbracket = \lambda x. \llbracket t_1 \rrbracket(x) + \llbracket t_2 \rrbracket(x) \\
\llbracket \text{Pop } v \rrbracket = \text{tl} \circ \llbracket v \rrbracket & \llbracket \text{mul } t_1 t_2 \rrbracket = \lambda x. \llbracket t_1 \rrbracket(x) * \llbracket t_2 \rrbracket(x) \\
& \llbracket \text{tuple } t_1 t_2 \rrbracket = \lambda x. (\llbracket t_1 \rrbracket(x), \llbracket t_2 \rrbracket(x)) \\
& \llbracket \text{first } t \rrbracket = \lambda x. \text{let } (x, y) = \llbracket t \rrbracket(x) \text{ in } x \\
& \llbracket \text{second } t \rrbracket = \lambda x. \text{let } (x, y) = \llbracket t \rrbracket(x) \text{ in } y
\end{array}$$

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

our heterogeneous lists to their appropriate types.

As mentioned by Barthe, et. al.[5], this small calculus,  $\Lambda_{\delta}^{\times, \rightarrow, \mathbb{R}}$ , 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)) denotes to the square function  $\lambda x. x * x$ .*

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

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

As we work with denotations, smooth functions  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  can be interpreted as the denotations of a corresponding syntactic term  $x_1, \dots, x_n \vdash t : R$ . 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  $x_1 : R, \dots, x_n : R \vdash t : R$  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  $\tau$ ,  $S_\tau$  is the relation between functions  $R \rightarrow \llbracket \tau \rrbracket$  and  $R \rightarrow \llbracket \vec{\mathcal{D}}(\tau) \rrbracket$ .

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

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  $x_1 : R, \dots, x_n : R \vdash t : \tau$  and argument function  $f : R \rightarrow R^n$  such that  $S_\tau(\llbracket t \rrbracket \circ f, \llbracket \vec{\mathcal{D}}(t) \rrbracket \circ \vec{\mathcal{D}}_n \circ f)$ .

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  $f : R \rightarrow \llbracket R^n \rrbracket$  into  $\vec{\mathcal{D}}_n(f, x) : R \rightarrow \llbracket \vec{\mathcal{D}}(R^n) \rrbracket$ . The full type signature of the function becomes  $\vec{\mathcal{D}}_n : (R \rightarrow \llbracket R^n \rrbracket) \rightarrow R \rightarrow \llbracket \vec{\mathcal{D}}(R^n) \rrbracket$ , which essentially accompanies each argument supplied by  $f$  with its accompanying derivative.

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

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  $f : R \rightarrow R^n$ . Previously the function was used to indicate the open variables or function arguments. If generalized to  $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$ , this same function could be seen as a function which supplies terms for each open variable  $x_1, \dots, x_n$  with their appropriate types. So the argument function now becomes the pair of functions  $s : R \rightarrow \llbracket \Gamma \rrbracket$  and  $s_D : R \rightarrow \llbracket \vec{D}(\Gamma) \rrbracket$ . 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  $s : R \rightarrow \llbracket \Gamma \rrbracket$  and  $s_D : R \rightarrow \llbracket \vec{D}(\Gamma) \rrbracket$  are inductively instantiated such that they follow

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

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  $s : R \rightarrow \llbracket \Gamma \rrbracket$  and  $s_D : R \rightarrow \llbracket \vec{D}(\Gamma) \rrbracket$  such that they follow  $inst_{\Gamma}(s, s_D)$ , then  $S_{\tau}(\llbracket t \rrbracket \circ s, \llbracket \vec{D}(t) \rrbracket \circ s_D)$ .*

*Proof.* This is proven by induction on the typing derivation of  $t$ . Unless otherwise specified, the type of  $s$  and  $s_D$  are respectively  $R \rightarrow \llbracket \Gamma \rrbracket$  and  $R \rightarrow \llbracket \vec{D}(\Gamma) \rrbracket$ .

1. (var)

Prove:  $S_{\tau}(\llbracket var\ v \rrbracket \circ s, \llbracket \vec{D}(var\ v) \rrbracket \circ s_D)$ .

Proceed by induction on the type evidence  $v$ .

- (Top) Base case

Prove:  $S_\tau(\llbracket \text{var } Top \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(\text{var } Top) \rrbracket \circ s_D)$ , where  $s : R \rightarrow \llbracket \tau :: \Gamma \rrbracket$  and  $s_D : R \rightarrow \llbracket \tau :: \Gamma \rrbracket$

In this case the referenced  $\tau$  exists at the top of the list. So both  $\llbracket \text{var } Top \rrbracket$  and  $\llbracket \vec{\mathcal{D}}(\text{var } Top) \rrbracket$  denote to fetching the top term. This is now proven by definition of *inst*, which states that the term is semantically well-typed.

$$\begin{aligned}
& S_\tau(\llbracket \text{var } Top \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(\text{var } Top) \rrbracket \circ s_D) \\
& \quad \Vdash (\text{Definition of } \vec{\mathcal{D}}) \\
& S_\tau(\llbracket \text{var } Top \rrbracket \circ s, \llbracket \text{var } Top \rrbracket \circ s_D) \\
& \quad \Vdash (\text{Definition of } \circ) \\
& S_\tau(\lambda x. \llbracket \text{var } Top \rrbracket(s(x)), \lambda x. \llbracket \text{var } Top \rrbracket(s_D(x))) \\
& \quad \Vdash (\text{Definition of } \llbracket \cdot \rrbracket) \\
& S_\tau(\lambda x. \text{lookup } \llbracket Top \rrbracket(s(x)), \lambda x. \text{lookup } \llbracket Top \rrbracket(s_D(x))) \\
& \quad \Vdash (\text{Rewrite using } s = \lambda x. \text{hd}(s(x)) :: \text{tl}(s(x))) \\
& S_\tau(\lambda x. \text{lookup } \llbracket Top \rrbracket(\text{hd}(s(x)) :: \text{tl}(s(x))), \\
& \quad \lambda x. \text{lookup } \llbracket Top \rrbracket(\text{hd}(s_D(x)) :: \text{tl}(s_D(x)))) \\
& \quad \Vdash (\text{Simplify with lookup and } \llbracket Top \rrbracket) \\
& S_\tau(\lambda x. \text{hd}(s(x)), \lambda x. \text{hd}(s_D(x))) \\
& \quad \Vdash (\text{By definition of } inst_{\tau :: \Gamma})
\end{aligned}$$

□

- (Pop) Induction step

Prove:  $S_\tau(\llbracket \text{var } (Pop \ v) \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(\text{var } (Pop \ v)) \rrbracket \circ s_D)$ , where  $s : R \rightarrow \llbracket \sigma :: \Gamma \rrbracket$  and  $s_D : R \rightarrow \llbracket \vec{\mathcal{D}}(\sigma :: \Gamma) \rrbracket$ .

Induction hypothesis:

$$\begin{aligned}
& (a) \ \forall (f : R \rightarrow \llbracket \Gamma \rrbracket), (g : R \rightarrow \llbracket \vec{\mathcal{D}}(\Gamma) \rrbracket). \\
& \quad S_\tau(\llbracket \text{var } v \rrbracket \circ f, \llbracket \vec{\mathcal{D}}(\text{var } v) \rrbracket \circ g)
\end{aligned}$$

Note that the var term now denotes to ignoring the arbitrary unrelated type  $\sigma$  and looking up  $v$  in the rest of the list  $\Gamma$ . So  $S_\tau(\llbracket \text{var } v \rrbracket \circ \text{tl} \circ s, \llbracket \vec{\mathcal{D}}(\text{var } v) \rrbracket \circ \text{tl} \circ s_D)$ , which is proven using the

induction hypothesis by respectively instantiating  $f$  and  $g$  as  $tl \circ s$  and  $tl \circ s_D$ .

$$\begin{aligned}
& S_\tau(\llbracket var (Pop v) \rrbracket \circ s, \llbracket \vec{D}(var (Pop v)) \rrbracket \circ s_D) \\
& \quad \vdash \text{(Definition of } \vec{D}) \\
& S_\tau(\llbracket var (Pop v) \rrbracket \circ s, \llbracket var (Pop v) \rrbracket \circ s_D) \\
& \quad \vdash \text{(Definition of } \circ) \\
& S_\tau(\lambda x. \llbracket var (Pop v) \rrbracket(s(x)), \lambda x. \llbracket var (Pop v) \rrbracket(s_D(x))) \\
& \quad \vdash \text{(Definition of } \llbracket \cdot \rrbracket) \\
& S_\tau(\lambda x. lookup \llbracket Pop v \rrbracket(s(x)), \lambda x. lookup \llbracket Pop v \rrbracket(s_D(x))) \\
& \quad \vdash \text{(Rewrite using } s = \lambda x. hd(s(x)) :: tl(s(x))) \\
& S_\tau(\lambda x. lookup \llbracket Pop v \rrbracket(hd(s(x)) :: tl(s(x))), \\
& \quad \lambda x. lookup \llbracket Pop v \rrbracket(hd(s_D(x)) :: tl(s_D(x)))) \\
& \quad \vdash \text{(Simplify with lookup and } \llbracket Pop v \rrbracket) \\
& S_\tau(\lambda x. lookup \llbracket v \rrbracket(tl(s(x))), \lambda x. lookup \llbracket v \rrbracket(tl(s_D(x)))) \\
& \quad \vdash \text{(Use IH. 1 with } f = tl(s(x)) \text{ and } g = tl(s_D(x)))
\end{aligned}$$

□

## 2. (app)

Prove:  $S_\tau(\llbracket app t_1 t_2 \rrbracket \circ s, \llbracket \vec{D}(app t_1 t_2) \rrbracket \circ s_D)$

Induction hypotheses:

- (a)  $S_{\sigma \rightarrow \tau}(\llbracket t_1 \rrbracket \circ s, \llbracket \vec{D}(t_1) \rrbracket \circ s_D)$
- (b)  $S_\sigma(\llbracket t_2 \rrbracket \circ s, \llbracket \vec{D}(t_2) \rrbracket \circ s_D)$

First it is useful to rewrite the induction hypothesis 2a in a more usable format. Rewrite the statement using the definition of  $S$  at function types.

$$\begin{aligned}
& S_{\sigma \rightarrow \tau}(\llbracket t_1 \rrbracket \circ s, \llbracket \vec{D}(t_1) \rrbracket \circ s_D) \\
& \quad \vdash \text{(Definition of } \circ) \\
& S_{\sigma \rightarrow \tau}(\lambda x. \llbracket t_1 \rrbracket(s(x)), \lambda x. \llbracket \vec{D}(t_1) \rrbracket(s_D(x))) \\
& \quad \vdash \text{(Definition of } S_{\rightarrow})
\end{aligned}$$

$$\forall f_1, f_2. S_\sigma(f_1, f_2) \rightarrow \\ S_\tau(\lambda x. (\llbracket t_1 \rrbracket(s(x))) (f_1(x)), \lambda x. (\llbracket \vec{\mathcal{D}}(t_1) \rrbracket(s_D(x))) (f_2(x)))$$

The case for app is now proven by applying the induction hypothesis 2a for the function term using the induction hypothesis 2b for the argument term to satisfy its premise.

$$\begin{aligned} & S_\tau(\llbracket \text{app } t_1 \ t_2 \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(\text{app } t_1 \ t_2) \rrbracket \circ s_D) \\ & \quad \vdash \text{(Definition of } \vec{\mathcal{D}}) \\ & S_\tau(\llbracket \text{app } t_1 \ t_2 \rrbracket \circ s, \llbracket \text{app } \vec{\mathcal{D}}(t_1) \ \vec{\mathcal{D}}(t_2) \rrbracket \circ s_D) \\ & \quad \vdash \text{(Definition of } \circ) \\ & S_\tau(\lambda x. \llbracket \text{app } t_1 \ t_2 \rrbracket(s(x)), \lambda x. \llbracket \text{app } \vec{\mathcal{D}}(t_1) \ \vec{\mathcal{D}}(t_2) \rrbracket(s_D(x))) \\ & \quad \vdash \text{(Definition of } \llbracket \cdot \rrbracket) \\ & S_\tau(\lambda x. (\llbracket t_1 \rrbracket(s(x))) (\llbracket t_2 \rrbracket(s(x))), \lambda x. (\llbracket \vec{\mathcal{D}}(t_1) \rrbracket(s_D(x))) (\llbracket \vec{\mathcal{D}}(t_2) \rrbracket(s_D(x)))) \\ & \quad \vdash \text{(Induction hypothesis 2a)} \\ & S_\sigma(\lambda x. \llbracket t_2 \rrbracket(s(x)), \lambda x. \llbracket \vec{\mathcal{D}}(t_2) \rrbracket \circ s_D(x))) \\ & \quad \vdash \text{(Induction hypothesis 2b)} \end{aligned}$$

□

### 3. (abs)

$$\text{Prove: } S_{\sigma \rightarrow \tau}(\llbracket \text{abs } t \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(\text{abs } t) \rrbracket \circ s_D)$$

Induction hypothesis:

$$(a) \ S_\sigma(\llbracket t \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(t) \rrbracket \circ s_D), \text{ where } s : R \rightarrow \llbracket \sigma :: \Gamma \rrbracket \text{ and } s_D : R \rightarrow \llbracket \sigma :: \Gamma \rrbracket$$

As is the case for 2a, simplify the goal statement using the definition of  $S_{\rightarrow}$ . So the proof obligation now becomes.

$$\text{Prove: } S_\tau(\lambda x. (\llbracket \text{abs } t \rrbracket(s(x))) (f_1(x)), \lambda x. (\llbracket \vec{\mathcal{D}}(\text{abs } t) \rrbracket(s_D(x))) (f_2(x)))$$

Assume:

$$\begin{aligned} (a) \ & f_1 : R \rightarrow \llbracket \sigma \rrbracket \\ (b) \ & f_2 : R \rightarrow \llbracket \vec{\mathcal{D}}(\sigma) \rrbracket \end{aligned}$$

(c)  $S_\sigma(f_1, f_2)$

The proof proceeds by rewriting the goal until we can apply the induction hypothesis. Note that the assumption 3c:  $S_\sigma(f_1, f_2)$  ensures that the requirement of  $inst_{\sigma::\Gamma}$  in the induction hypothesis 3a is satisfied.

$$\begin{aligned}
& S_\tau(\lambda x.(\llbracket abs\ t \rrbracket(s(x)))(f_1(x)), \lambda x.(\llbracket \vec{\mathcal{D}}(abs\ t) \rrbracket(s_D(x)))(f_2(x))) \\
& \quad \Vdash (\text{Definition of } \vec{\mathcal{D}}) \\
& S_\tau(\lambda x.(\llbracket abs\ t \rrbracket(s(x)))(f_1(x)), \lambda x.(\llbracket abs\ \vec{\mathcal{D}}(t) \rrbracket(s_D(x)))(f_2(x))) \\
& \quad \Vdash (\text{Definition of } \llbracket \cdot \rrbracket) \\
& S_\tau(\lambda x.(\llbracket t \rrbracket(f_1(x) :: s(x))), \lambda x.(\llbracket \vec{\mathcal{D}}(t) \rrbracket(f_2(x) :: s_D(x)))) \\
& \quad \Vdash (\text{Induction hypothesis 2a})
\end{aligned}$$

□

#### 4. (rval)

Prove:  $S_R(\llbracket rval\ n \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(rval\ n) \rrbracket \circ s_D)$

This is proven by noting that the corresponding denotations of `rval` are constant functions, which are both smooth and whose derivatives are equal to 0.

$$\begin{aligned}
& S_R(\llbracket rval\ n \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(rval\ n) \rrbracket \circ s_D) \\
& \quad \Vdash (\text{Definition of } \vec{\mathcal{D}}) \\
& S_R(\llbracket rval\ n \rrbracket \circ s, \llbracket tuple\ (rval\ n)\ (rval\ 0) \rrbracket \circ s_D) \\
& \quad \Vdash (\text{Definition of } \llbracket \cdot \rrbracket) \\
& S_R(const\ n, (const\ n, const\ 0)) \\
& \quad \Vdash (\text{Definition of } S_R) \\
& smooth\ (const\ n) \wedge const\ 0 = \partial const\ n / \partial x \\
& \quad \Vdash (\text{split goals: goal 1}) \\
& \quad smooth\ (const\ n) \\
& \quad \quad \Vdash (f(x) = n \text{ is continuously differentiable}) \\
& \quad \Vdash (\text{split goals: goal 2}) \\
& \quad const\ 0 = \partial const\ n / \partial x
\end{aligned}$$

$$\Vdash (\text{if } f(x) = n, \text{ then } \partial f / \partial x = 0)$$

□

5. (add)

Prove:  $S_R(\llbracket \text{add } t_1 \ t_2 \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(\text{add } t_1 \ t_2) \rrbracket \circ s_D)$

Induction hypotheses:

$$(a) \ S_R(\llbracket t_1 \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(t_1) \rrbracket \circ s_D)$$

$$(b) \ S_R(\llbracket t_2 \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(t_2) \rrbracket \circ s_D)$$

The proof proceeds by simplifying the denotations and proving the smoothness and derivative requirements for  $S_R$ .

$$\begin{aligned} & S_R(\llbracket \text{add } t_1 \ t_2 \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(\text{add } t_1 \ t_2) \rrbracket \circ s_D) \\ & \Vdash (\text{Definition of } \vec{\mathcal{D}}) \\ & S_R(\llbracket \text{add } t_1 \ t_2 \rrbracket \circ s, \llbracket \text{tuple} \\ & \quad (\text{add } (\text{first } \vec{\mathcal{D}}(t_1)) (\text{first } \vec{\mathcal{D}}(t_2))) \\ & \quad (\text{add } (\text{second } \vec{\mathcal{D}}(t_1)) (\text{second } \vec{\mathcal{D}}(t_2))) \rrbracket \circ s_D) \\ & \Vdash (\text{Definition of } \llbracket \cdot \rrbracket, \text{ using} \\ & \quad (d_1, d'_1) = \llbracket \vec{\mathcal{D}}(t_1) \rrbracket s(x) \text{ and } (d_2, d'_2) = \llbracket \vec{\mathcal{D}}(t_2) \rrbracket s_D(x)) \\ & S_R(\lambda x. d_1(x) + d_2(x), \lambda x. (d_1(x) + d_2(x), d'_1(x) + d'_2(x))) \\ & \Vdash (\text{Definition of } S_R) \\ & \text{smooth } (\lambda x. d_1(x) + d_2(x)) \wedge \\ & \quad \lambda x. d'_1(x) + d'_2(x) = \partial(\lambda x. d'_1(x) + d'_2(x)) / \partial x \\ & \Vdash (\text{split goals: goal 1}) \\ & \text{smooth } (\lambda x. d_1(x) + d_2(x)) \\ & \Vdash (\text{Addition is smooth, if subterms are smooth}) \\ & \text{smooth } d_1 \wedge \text{smooth } d_2 \\ & \Vdash (\text{Induction hypothesis 5a for } d_1 \text{ and 5b for } d_2) \\ & \Vdash (\text{split goals: goal 2}) \\ & \lambda x. d'_1(x) + d'_2(x) = \partial(\lambda x. d'_1(x) + d'_2(x)) / \partial x \\ & \Vdash (\text{By definition of taking the derivative of addition}) \\ & d'_1 = \partial d_1 / \partial x \wedge d'_2 = \partial d_2 / \partial x \end{aligned}$$

$\Vdash$  (Induction hypothesis 5a for  $d_1$  and 5b for  $d_2$ )

□

6. (mul)

Prove:  $S_R(\llbracket mul\ t_1\ t_2 \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(mul\ t_1\ t_2) \rrbracket \circ s_D)$

Induction hypotheses:

(a)  $S_R(\llbracket t_1 \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(t_1) \rrbracket \circ s_D)$

(b)  $S_R(\llbracket t_2 \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(t_2) \rrbracket \circ s_D)$

Proof goes through almost identically as for the case for add.

$S_R(\llbracket mul\ t_1\ t_2 \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(mul\ t_1\ t_2) \rrbracket \circ s_D)$

$\Vdash$  (Definition of  $\vec{\mathcal{D}}$ )

$S_R(\llbracket mul\ t_1\ t_2 \rrbracket \circ s, \llbracket tuple$

$(mul\ (first\ \vec{\mathcal{D}}(t_1))\ (first\ \vec{\mathcal{D}}(t_2)))$

$(add$

$(mul\ (first\ \vec{\mathcal{D}}(t_1))\ (second\ \vec{\mathcal{D}}(t_2)))$

$(mul\ (first\ \vec{\mathcal{D}}(t_2))\ (second\ \vec{\mathcal{D}}(t_1)))) \rrbracket \circ s_D)$

$\Vdash$  (Definition of  $\llbracket \cdot \rrbracket$ , using

$(d_1, d'_1) = \llbracket \vec{\mathcal{D}}(t_1) \rrbracket s(x)$  and  $(d_2, d'_2) = \llbracket \vec{\mathcal{D}}(t_2) \rrbracket s_D(x))$

$S_R(\lambda x. d_1(x) * d_2(x),$

$\lambda x. (d_1(x) * d_2(x), d_1(x) * d'_2(x) + (d_2(x) * d'_1(x))))$

$\Vdash$  (Definition of  $S_R$ )

$smooth\ (\lambda x. d_1(x) * d_2(x)) \wedge$

$\lambda x. d_1(x) * d'_2(x) + d_2(x) * d'_1(x) = \partial(\lambda x. (d_1(x) * d_2(x)))/\partial x$

$\Vdash$  (split goals: goal 1)

$smooth\ (\lambda x. d_1(x) * d_2(x))$

$\Vdash$  (Multiplication is smooth, if subterms are smooth)

$smooth\ d_1 \wedge smooth\ d_2$

$\Vdash$  (Induction hypothesis 6a for  $d_1$  and 6b for  $d_2$ )

$\Vdash$  (split goals: goal 2)

$$\lambda x. d_1(x) * d'_2(x) + d_2(x) * d'_1(x) = \partial(\lambda x. (d_1(x) * d_2(x)) / \partial x$$

$\Vdash$  (By definition of taking the derivative of multiplications)

$$d'_1 = \partial d_1 / \partial x \wedge d'_2 = \partial d_2 / \partial x$$

$\Vdash$  (Induction hypothesis 6a for  $d_1$  and 6b for  $d_2$ )

□

## 7. (tuple)

Prove:  $S_{(\tau \times \sigma)}(\llbracket tuple\ t_1\ t_2 \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(tuple\ t_1\ t_2) \rrbracket \circ s_D)$

Induction hypotheses:

$$(a) \ S_{\tau}(\llbracket t_1 \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(t_1) \rrbracket \circ s_D)$$

$$(b) \ S_{\sigma}(\llbracket t_2 \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(t_2) \rrbracket \circ s_D)$$

A recurring pattern will become apparent in later sections when continuing to prove the substitution lemma 1 for types consisting of other types. In this case, due to the carefull attention spent on the logical relation, only the witnesses of the subterms of the tuple need to be supplied to finish the proof.

Note that the witnesses of  $S_{\tau}$  and  $S_{\sigma}$  that need to be given here are supplied by the induction hypotheses. While these witnesses are not exactly relevant to finish this proof for tuple, they are needed in the proofs for projections.

$$S_{(\tau \times \sigma)}(\llbracket tuple\ t_1\ t_2 \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(tuple\ t_1\ t_2) \rrbracket \circ s_D)$$

$\Vdash$  (Definition of  $\vec{\mathcal{D}}$ )

$$S_{(\tau \times \sigma)}(\llbracket tuple\ t_1\ t_2 \rrbracket \circ s, \llbracket tuple\ \vec{\mathcal{D}}(t_1)\ \vec{\mathcal{D}}(t_2) \rrbracket \circ s_D)$$

$\Vdash$  (Definition of  $\llbracket \cdot \rrbracket$ )

$$S_{(\tau \times \sigma)}(\lambda x. (\llbracket t_1 \rrbracket(s(x)), \llbracket t_2 \rrbracket(s(x))), \\ \lambda x. (\llbracket \vec{\mathcal{D}}(t_1) \rrbracket(s'(x)), \llbracket \vec{\mathcal{D}}(t_2) \rrbracket(s'(x))))$$

$\Vdash$  (Definition of  $S_{\tau \times \sigma}$ )

$$\exists f_1, f_2, g_1, g_2,$$

$$S_{\tau}(f_1, f_2), S_{\sigma}(g_1, g_2).$$



$$\begin{aligned}
& \lambda x.(\llbracket t_1 \rrbracket(s(x)), \llbracket t_2 \rrbracket(s(x))) = \lambda x.(f_1(x), g_1(x)) \wedge \\
& \lambda x.(\llbracket \vec{\mathcal{D}}(t_1) \rrbracket(s'(x)), \llbracket \vec{\mathcal{D}}(t_2) \rrbracket(s'(x))) = \lambda x.(f_2(x), g_2(x)) \\
& \Vdash (\text{Give witnesses: } f_1 := \llbracket t_1 \rrbracket \circ s, f_2 := \llbracket t_2 \rrbracket \circ s, \\
& \quad g_1 := \llbracket \vec{\mathcal{D}}(t_1) \rrbracket \circ s', g_2 := \llbracket \vec{\mathcal{D}}(t_2) \rrbracket \circ s') \\
& \exists S_\tau(f_1, f_2), S_\sigma(g_1, g_2). \\
& \lambda x.(\llbracket t_1 \rrbracket(s(x)), \llbracket t_2 \rrbracket(s(x))) \\
& \quad = \lambda x.(\llbracket t_1 \rrbracket(s(x)), \llbracket t_2 \rrbracket(s(x))) \wedge \\
& \lambda x.(\llbracket \vec{\mathcal{D}}(t_1) \rrbracket(s'(x)), \llbracket \vec{\mathcal{D}}(t_2) \rrbracket(s'(x))) \\
& \quad = \lambda x.(\llbracket \vec{\mathcal{D}}(t_1) \rrbracket(s'(x)), \llbracket \vec{\mathcal{D}}(t_2) \rrbracket(s'(x))) \\
& \Vdash (\text{Give witnesses of } S_\tau \text{ and } S_\sigma \text{ using respective IHs 7a and 7b}) \\
& \lambda x.(\llbracket t_1 \rrbracket(s(x)), \llbracket t_2 \rrbracket(s(x))) \\
& \quad = \lambda x.(\llbracket t_1 \rrbracket(s(x)), \llbracket t_2 \rrbracket(s(x))) \wedge \\
& \lambda x.(\llbracket \vec{\mathcal{D}}(t_1) \rrbracket(s'(x)), \llbracket \vec{\mathcal{D}}(t_2) \rrbracket(s'(x))) \\
& \quad = \lambda x.(\llbracket \vec{\mathcal{D}}(t_1) \rrbracket(s'(x)), \llbracket \vec{\mathcal{D}}(t_2) \rrbracket(s'(x))) \\
& \Vdash (\text{Reflexivity})
\end{aligned}$$

□

## 8. (first)

Prove:  $S_\tau(\llbracket first\ t \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(first\ t) \rrbracket \circ s_D)$

Induction hypotheses:

$$(a) \ S_{\tau \times \sigma}(\llbracket t \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(t) \rrbracket \circ s_D)$$

Simplifying the induction hypothesis 9a using the definition of  $S_{\tau \times \sigma}$  gives rise to a number of useful assumptions containing:  $f_1 : R \rightarrow \llbracket \tau \rrbracket$ ,  $f_2 : R \rightarrow \llbracket \vec{\mathcal{D}}(\tau) \rrbracket$ ,  $g_1 : R \rightarrow \llbracket \sigma \rrbracket$  and  $g_2 : R \rightarrow \llbracket \vec{\mathcal{D}}(\sigma) \rrbracket$ .

Assumptions:

- (a)  $S_\tau(f_1, f_2)$
- (b)  $S_\sigma(g_1, g_2)$
- (c)  $\llbracket t \rrbracket \circ s = \lambda x.(f_1(x), g_1(x))$
- (d)  $\llbracket \vec{\mathcal{D}}(t) \rrbracket \circ s = \lambda x.(f_2(x), g_2(x))$

$$\begin{aligned}
& S_\tau(\llbracket first\ t \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(first\ t) \rrbracket \circ s_D) \\
& \quad \Vdash (\text{Rewrite using definition of } \vec{\mathcal{D}}) \\
& S_\tau(\llbracket first\ t \rrbracket \circ s, \llbracket first\ \vec{\mathcal{D}}(t) \rrbracket \circ s_D) \\
& \quad \Vdash (\text{Rewrite using definition of } \llbracket \cdot \rrbracket) \\
& S_\tau(\lambda x. fst(\llbracket t \rrbracket(s(x))), \lambda x. fst(\llbracket \vec{\mathcal{D}}(t) \rrbracket(s_D(x)))) \\
& \quad \Vdash (\text{Rewrite using 8c and 8d}) \\
& S_\tau(\lambda x. fst(f_1(x), g_1(x)), \lambda x. fst(f_2(x), g_2(x))) \\
& \quad \Vdash (\beta\eta\text{-equality}) \\
& S_\tau(f_1, f_2) \\
& \quad \Vdash (\text{Assumption 8a})
\end{aligned}$$

□

## 9. (second)

Prove:  $S_\tau(\llbracket first\ t \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(first\ t) \rrbracket \circ s_D)$

Induction hypotheses:

$$(a) \ S_{\tau \times \sigma}(\llbracket t \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(t) \rrbracket \circ s_D)$$

Proof goes the same as the case for `first` with the same assumptions following from the induction hypothesis, with  $f_1 : R \rightarrow \llbracket \tau \rrbracket$ ,  $f_2 : R \rightarrow \llbracket \vec{\mathcal{D}}(\tau) \rrbracket$ ,  $g_1 : R \rightarrow \llbracket \sigma \rrbracket$  and  $g_2 : R \rightarrow \llbracket \vec{\mathcal{D}}(\sigma) \rrbracket$ .

Assumptions:

- (a)  $S_\tau(f_1, f_2)$
- (b)  $S_\sigma(g_1, g_2)$
- (c)  $\llbracket t \rrbracket \circ s = \lambda x. (f_1(x), g_1(x))$
- (d)  $\llbracket \vec{\mathcal{D}}(t) \rrbracket \circ s = \lambda x. (f_2(x), g_2(x))$

$$\begin{aligned}
& S_\sigma(\llbracket second\ t \rrbracket \circ s, \llbracket \vec{\mathcal{D}}(second\ t) \rrbracket \circ s_D) \\
& \quad \Vdash (\text{Rewrite using definition of } \vec{\mathcal{D}})
\end{aligned}$$

$$\begin{aligned}
& S_\sigma(\llbracket \text{second } t \rrbracket \circ s, \llbracket \text{second } \vec{\mathcal{D}}(t) \rrbracket \circ s_D) \\
& \vdash \text{(Rewrite using definition of } \llbracket \cdot \rrbracket \text{)} \\
& S_\sigma(\lambda x. \text{snd}(\llbracket t \rrbracket(s(x))), \lambda x. \text{snd}(\llbracket \vec{\mathcal{D}}(t) \rrbracket(s_D(x)))) \\
& \vdash \text{(Rewrite using 9c and 9d)} \\
& S_\sigma(\lambda x. \text{snd}(f_1(x), g_1(x)), \lambda x. \text{snd}(f_2(x), g_2(x))) \\
& \vdash (\beta\eta\text{-equality}) \\
& S_\sigma(f_1, f_2) \\
& \vdash \text{(Assumption 9a)}
\end{aligned}$$

□

□

The proof of the fundamental property of the logical relation now follows from the substitution lemma.

**Lemma 2** (Fundamental property). *For any well-typed term  $x_1 : R, \dots, x_n : R \vdash t : \tau$ , and argument function  $f : R \rightarrow \llbracket R^n \rrbracket$ , such that each argument is continuously derivable, then  $S_\tau(\llbracket t \rrbracket \circ f, \llbracket \vec{\mathcal{D}}(t) \rrbracket \circ \vec{\mathcal{D}}_n \circ f)$ .*

*Proof.* This is proven by instantiating the substitution lemma 1 with the proper variables and proving the resulting judgement of  $\text{inst}_{R^n}$  by induction on  $n$ .

$$\begin{aligned}
& S_\tau(\llbracket t \rrbracket \circ f, \llbracket \vec{\mathcal{D}}(t) \rrbracket \circ \vec{\mathcal{D}}_n \circ f) \\
& \vdash \text{(Apply substitution lemma with } s := f, s_D := \vec{\mathcal{D}}_n \circ f \text{ and } \Gamma := R^n) \\
& \text{inst}_{R^n}(f, \vec{\mathcal{D}}_n \circ f)
\end{aligned}$$

Proceed by induction on  $n$ , intuitively building up the environment with denotation of terms such that they follow  $S$ .

- Base case:  $n = 0$

$$\begin{aligned}
& \text{inst}_{R^0}(f, \vec{\mathcal{D}}_0 \circ f) \\
& \vdash \text{(Induction on } n, \text{ base case } n = 0)
\end{aligned}$$

$$\begin{aligned}
& inst_{\square}(f, \vec{\mathcal{D}}_0 \circ f) \\
& \vdash (\text{Singleton instance of } R \rightarrow R^0, f = const(\square)) \\
& inst_{\square}(const(\square), \vec{\mathcal{D}}_0 \circ const(\square)) \\
& \vdash (\text{Definition of } \vec{\mathcal{D}}_0) \\
& inst_{\square}(const(\square), const(\square)) \\
& \vdash (\text{Definition of } inst_{\square})
\end{aligned}$$

- Induction case:  $n = S(n')$

Induction hypothesis:  $inst_{R^{n'}}(f', D \circ f')$ , where  $f' : R \rightarrow \llbracket R^{n'} \rrbracket$ .

$$\begin{aligned}
& inst_{R::R^{n'}}(f, \vec{\mathcal{D}}_{R::R^{n'}} \circ f) \\
& \vdash (\text{Unfold } \circ) \\
& inst_{R::R^{n'}}(\lambda x.f(x), \lambda x.\vec{\mathcal{D}}_{R::R^{n'}}(f, x)) \\
& \vdash (\text{Rewrite using } f = \lambda x.hd(f(x)) :: tl(f(x)) \\
& \quad \text{and definition of } \vec{\mathcal{D}}_{R::R^{n'}}) \\
& inst_{R::R^{n'}}(\lambda x.hd(f(x)) :: tl(f(x)), \\
& \quad \lambda x.(hd(f(x)), \partial(hd \circ f)/\partial x(x)) :: \vec{\mathcal{D}}_{R^{n'}}(tl \circ f, x)) \\
& \vdash (\text{By definition of } inst_{R::R^{n'}}, \text{ rest proven by IH})
\end{aligned}$$

□

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  $x_1 : R, \dots, x_n : R \vdash t : R$ ,  $\llbracket \vec{\mathcal{D}}(t) \rrbracket$  gives the dual number representation of  $\llbracket t \rrbracket$ , such that for any argument function  $f : R \rightarrow R^n$ , then  $\llbracket \vec{\mathcal{D}}(t) \rrbracket \circ \vec{\mathcal{D}}_n \circ f = \lambda x.(\llbracket t \rrbracket \circ f, \partial(\llbracket t \rrbracket \circ f)/\partial x)$ .*

*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.

$$\llbracket \vec{\mathcal{D}}(t) \rrbracket \circ \vec{\mathcal{D}}_n \circ f = \lambda x.(\llbracket t \rrbracket \circ f, \partial(\llbracket t \rrbracket \circ f)/\partial x)$$

$\Vdash$  (By definition of  $S_R$  with  $f := \llbracket t \rrbracket \circ f$  and  $g := \llbracket \vec{\mathcal{D}}(t) \rrbracket \circ \vec{\mathcal{D}}_n \circ f$ )  
 $S_R(\llbracket t \rrbracket \circ f, \llbracket \vec{\mathcal{D}}(t) \rrbracket \circ \vec{\mathcal{D}}_n \circ f)$   
 $\Vdash$  (Fundamental property (2))

□

## 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

...

$$\begin{array}{c}
 \frac{\Gamma \vdash e : \tau + \sigma \quad \Gamma \vdash t1 : \tau \rightarrow \rho \quad \Gamma \vdash t2 : \sigma \rightarrow \rho}{\Gamma \vdash \text{case } e \text{ } t1 \text{ } t2 : \rho} \text{TCASE} \\
 \\
 \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \text{inl } t : \tau + \sigma} \text{TI}_{\text{NL}} \qquad \frac{\Gamma \vdash t : \sigma}{\Gamma \vdash \text{inr } t : \tau + \sigma} \text{TI}_{\text{NR}} \\
 \\
 \frac{n \in \mathcal{N}}{\Gamma \vdash \text{nval } n : \mathbb{N}} \text{TN}_{\text{VAL}} \qquad \frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \text{nsucc } t : \mathbb{N}} \text{TN}_{\text{SUCC}} \\
 \\
 \frac{\Gamma \vdash f : \tau \rightarrow \tau \quad \Gamma \vdash n : \mathbb{N} \quad \Gamma \vdash t : \tau}{\Gamma \vdash \text{nrec } f \text{ } n \text{ } t : \tau} \text{TP}_{\text{PRIM}}
 \end{array}$$

Figure 4: Type-inference rules for language constructs for sum 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).